ARQUITETURA DE SOFTWARE 2/2024

Prof. Dr. Felipe Batista da Silva

Atividade

Padrões de Projetos (Design Patterns)

Nota: _____

Nome: Rafael Oliveira Venancio **RA:22309003**

Instruções

- Leia atentamente o que é pedido em cada questão.
- Entregue sua atividade de maneira ordenada e organizada.
- A atividade pode ser entregue em documento Word ou PDF **em arquivo único.**
- Para a questão que envolve implementação de uma linguagem, pode-se fazer a cópia do código escrito do editor como VS Code, por exemplo. Daí, insira o código em um arquivo como documento no Word.
- Todo código **deve ter um cabeçalho** com a data de escrita, seu nome, e o nome do autor caso use código de terceiros.
- A atividade terá um percentual de 30% da prova 2.
- Entrega 11/11/2024 até 23:59.

Parte 1. Indique quais são os 3 tipos de padrões de projetos e apresente a definição de cada um.

Parte 2. Para cada tipo de padrão de projeto, liste todos os casos.

Parte 3. Para cada um dos 22 padrões de projetos:

- Apresente a definição.
- Apresente uma analogia que ilustre a aplicação do padrão.
- Apresente duas vantagens e duas desvantagens do uso do padrão.
- Implemente o padrão em alguma linguagem de programação.

Parte 4. Escreva um pequeno parágrafo sumarizando o que você aprendeu com esta atividade, e como isso poderá ser utilizado profissionalmente na sua carreira.

O que são os padrões de projeto?

São soluções típicas para problemas comuns em projeto de software. Eles são como plantas de obra pré fabricadas que você pode customizar para resolver um problema de projeto recorrente em seu código.

Como são classificados os padrões de projeto?

Os padrões podem ser categorizados por seu propósito, ou intenção. Segue as três classificações dos padrões de projeto:

Padrões criacionais: fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código. Padrões que compõem esse grupo são: Factory Method, Abstract Factory, Builder, Prototype e Singleton;

Padrões estruturais: explicam como montar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes. Padrões que compõem esse grupo são: Adapter, Bridge, Composite, Decorator, Facade, Flyweight e Proxy;

Padrões comportamentais: cuidam da comunicação eficiente e da assinalação de responsabilidades entre objetos. Padrões que compõem esse grupo são: Chain of responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method e Visitor.

Padrões Criacionais

Factory Method:

-Definição

É um padrão criacional de projeto que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados. O padrão Factory Method sugere que você substitua chamadas diretas de construção de objetos (usando o operador new) por chamadas para um método fábrica especial.

-Analogia

Imagine uma padaria que faz pães. A fábrica de pães (método fábrica) sabe como fazer pão, mas não especifica qual tipo (integral, francês, brioche). Cada tipo específico de pão é uma subclasse com uma implementação específica.

-Vantagens:

Evita acoplamentos firmes entre o criador e os produtos concretos; Aumenta a manutenibilidade do código; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Aumenta a complexidade do código; Exige muitas subclasses para criar objetos específicos, aumentando o número de classes.

-Exemplo em código:

Python

```python
from abc import ABC, abstractmethod

# Interface Pao define o método para criação do pão
class Pao(ABC):
    @abstractmethod
    def criar_pao(self):
        pass

# Subclasse concreta que implementa o pão francês
class PaoFrances(Pao):
    def criar_pao(self):
        return "Pão Francês"

# Subclasse concreta que implementa o pão integral
class PaoIntegral(Pao):
    def criar_pao(self):
        return "Pão Integral"

# Fábrica que cria o pão de acordo com o tipo especificado
def fabrica_pao(tipo):
    if tipo == "frances":
        return PaoFrances()
    elif tipo == "integral":
        return PaoIntegral()
    else:
        raise ValueError("Tipo de pão desconhecido")
    # Tratamento de erro para tipos inválidos

# Criando um pão do tipo francês
pao = fabrica_pao("frances")
print(pao.criar_pao())
```

Java

```java
// Interface Pao define o método para criação do pão
interface Pao {
    String criarPao();
}

// Subclasse concreta que implementa o pão francês
class PaoFrances implements Pao {
    public String criarPao() {
        return "Pão Francês";
    }
}

// Subclasse concreta que implementa o pão integral
class PaoIntegral implements Pao {
    public String criarPao() {
        return "Pão Integral";
    }
}

// Fábrica que cria o pão de acordo com o tipo especificado
public class FabricaDePao {
    public static Pao criarPao(String tipo) {
        if (tipo.equals("frances")) {
            return new PaoFrances();
        } else if (tipo.equals("integral")) {
            return new PaoIntegral();
        } else {
            throw new IllegalArgumentException("Tipo de pão desconhecido");
        }
    }

    public static void main(String[] args) {
        Pao pao = FabricaDePao.criarPao("frances");
        System.out.println(pao.criarPao());
    }
}

```

Factory Method:

-Definição:

É um padrão de projeto criacional que permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.

-Analogia:

Imagine uma loja de móveis que fabrica conjuntos de móveis. Se você escolher uma linha de móveis, todos os móveis serão do mesmo estilo (moderno, clássico, etc.), mas cada linha terá suas próprias cadeiras, mesas, sofás, etc., sem que se misturem estilos.

-Vantagens:

Garante a compatibilidade entre os objetos criados; Facilita a criação de famílias de objetos relacionados; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Gera alta complexidade; Inserção de muitas classes e interfaces; Dificulta a adição de novos produtos sem modificar o código existente.

-Exemplos em código:

Python

```python
from abc import ABC, abstractmethod

# Definindo a interface para um sofá
class Sofa(ABC):
    @abstractmethod
    def get_type(self):
        pass

# Definindo a interface para uma mesa
class Mesa(ABC):
    @abstractmethod
    def get_type(self):
        pass

# Implementação concreta de um sofá moderno
class SofaModerno(Sofa):
    def get_type(self):
        return "Sofá Moderno"

# Implementação concreta de uma mesa moderna
class MesaModerna(Mesa):
    def get_type(self):
        return "Mesa Moderna"

# Implementação concreta de um sofá clássico
class SofaClassico(Sofa):
    def get_type(self):
        return "Sofá Clássico"
```

```python
# Implementação concreta de uma mesa clássica
class MesaClassica(Mesa):
    def get_type(self):
        return "Mesa Clássica"

# Fábrica que cria móveis no estilo moderno
class FabricaModerna:
    def criar_sofa(self):
        return SofaModerno()

    def criar_mesa(self):
        return MesaModerna()

# Fábrica que cria móveis no estilo clássico
class FabricaClassica:
    def criar_sofa(self):
        return SofaClassico()

    def criar_mesa(self):
        return MesaClassica()

# Usando a fábrica moderna para criar móveis
fabrica = FabricaModerna()
sofa = fabrica.criar_sofa()
mesa = fabrica.criar_mesa()
print(sofa.get_type())
print(mesa.get_type())
```

Java

```java
// Definindo a interface para um sofá
interface Sofa {
    String getTipo();
}

// Definindo a interface para uma mesa
interface Mesa {
    String getTipo();
}

// Implementação concreta de um sofá moderno
class SofaModerno implements Sofa {
    public String getTipo() {
        return "Sofá Moderno";
    }
}

// Implementação concreta de uma mesa moderna
class MesaModerna implements Mesa {
    public String getTipo() {
        return "Mesa Moderna";
    }
}

// Implementação concreta de um sofá clássico
class SofaClassico implements Sofa {
    public String getTipo() {
        return "Sofá Clássico";
    }
}

// Implementação concreta de uma mesa clássica
class MesaClassica implements Mesa {
    public String getTipo() {
        return "Mesa Clássica";
    }
}

// Fábrica que cria móveis no estilo moderno
interface FabricaDeMoveis {
    Sofa criarSofa();
    Mesa criarMesa();
}
```

```java
// Fábrica moderna
class FabricaModerna implements FabricaDeMoveis {
    public Sofa criarSofa() {
        return new SofaModerno();
    }

    public Mesa criarMesa() {
        return new MesaModerna();
    }
}

// Fábrica clássica
class FabricaClassica implements FabricaDeMoveis {
    public Sofa criarSofa() {
        return new SofaClassico();
    }

    public Mesa criarMesa() {
        return new MesaClassica();
    }
}

// Usando a fábrica moderna para criar móveis
public class Main {
    public static void main(String[] args) {
        FabricaDeMoveis fabrica = new FabricaModerna();
        Sofa sofa = fabrica.criarSofa();
        Mesa mesa = fabrica.criarMesa();

        System.out.println(sofa.getTipo());  // "Sofá Moderno"
        System.out.println(mesa.getTipo());  // "Mesa Moderna"
    }
}
```

Builder

-Definição:

É um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção. O padrão Builder sugere que você extraia o código de construção do objeto para fora de sua própria classe e mova ele para objetos separados chamados builders.

-Analogia:

Pense em um processo de montagem de hambúrguer em uma lanchonete. O cliente pode escolher diferentes componentes (pão, carne, queijo, molho, etc.). O processo de montagem (construção) é o mesmo, mas os ingredientes variam conforme o pedido.

-Vantagens:

Facilita a criação de objetos complexos com múltiplos componentes; Proporciona flexibilidade na construção de diferentes representações de um mesmo objeto; Implementa o princípio S do acrônimo SOLID.

-Desvantagens:

Complexo para objetos simples; Aumenta a complexidade do código, devido a múltiplas classes.

-Exemplos em código:

Python

```python
# Classe que representa o produto final, Burger
class Hamburguer:
    def __init__(self):
        self.ingredients = []  # Lista para armazenar os ingredientes do hambúrguer

    def add_ingredient(self, ingrediente):
        self.ingredients.append(ingrediente)  # Adiciona um ingrediente ao hambúrguer

# Builder para criar diferentes tipos de Hamburguer com componentes opcionais
class HamburguerBuilder:
    def __init__(self):
        self.hamburguer = Hamburguer()  # Cria uma nova instância de hamburguer

    def add_pao(self):
        self.hamburguer.add_ingredient("Pao")  # Adiciona pão
        return self  # Retorna o builder para facilitar o encadeamento

    def add_carne(self):
        self.hamburguer.add_ingredient("Carne")  # Adiciona carne
        return self

    def add_queijo(self):
        self.hamburguer.add_ingredient("Queijo")  # Adiciona queijo
        return self

    def add_alface(self):
        self.hamburguer.add_ingredient("Alface")  # Adiciona alface
        return self

    def build(self):
        return self.hamburguer  # Retorna o hambúrguer completo

# Usando o builder para construir um hambúrguer com diferentes ingredientes
builder = HamburguerBuilder()
burger = builder.add_pao().add_carne().add_queijo().add_alface().build()
print(burger.ingredients)
```

Java

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

// Classe que representa o produto final, hamburguer
class Hamburguer {
    private List<String> ingredientes = new ArrayList<>();

    public void addIngrediente(String ingrediente) {
        ingredientes.add(ingrediente);
    }

    public List<String> getIngredientes() {
        return ingredientes;
    }
}

// Builder para criar diferentes tipos de hamburguer com componentes opcionais
class HamburguerBuilder {
    private Hamburguer hamburguer = new Hamburguer();

    public HamburguerBuilder addPao() {
        hamburguer.addIngrediente("Pão");
        return this;
    }

    public HamburguerBuilder addCarne() {
        hamburguer.addIngrediente("Carne");
        return this;
    }

    public HamburguerBuilder addQueijo() {
        hamburguer.addIngrediente("Queijo");
        return this;
    }

    public HamburguerBuilder addAlface() {
        hamburguer.addIngrediente("Alface");
        return this;
    }

    public hamburguer build() {
        return hamburguer;
    }
}

// Usando o builder para construir um hambúrguer com diferentes ingredientes
public class Main {
    public static void main(String[] args) {
        Hamburguer hamburguer = new HamburguerBuilder()
            .addPao()
            .addCarne()
            .addQueijo()
            .addAlface()
            .build();

        System.out.println(hamburguer.getIngredientes());
    }
}
```

Prototype

-Definição:

É um padrão de projeto criacional que permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes. O padrão Prototype delega o processo de clonagem para o próprio objeto que está sendo clonado. O padrão declara uma interface comum para todos os objetos que suportam clonagem. Essa interface permite que você clone um objeto sem acoplar seu código à classe daquele objeto.

-Analogia:

Imagine um fotógrafo profissional. Ele faz uma série de fotos e precisa de várias cópias para enviar aos clientes. Em vez de refazer cada foto, ele faz cópias do original.

-Vantagens:

Facilita a criação de objetos a partir de protótipos existentes; Útil para objetos complexos que são custosos de criar diretamente.

-Desvantagens:

Clonar objetos com referências profundas pode requerer uma implementação adicional para garantir uma cópia completa; Algumas linguagens não possuem suporte nativo para clonagem.

-Exemplos em código:

Python

```python
import copy

# Classe que representa o protótipo Carro
class Carro:
    def __init__(self, modelo, ano):
        self.modelo = modelo  # Modelo do carro
        self.ano = ano        # Ano do carro

    def clone(self):
        # Retorna uma cópia profunda do objeto Carro, permitindo a criação de um novo carro com os mesmos atributos
        return copy.deepcopy(self)

# Criando um carro protótipo
carro1 = Carro("Sedan", 2020)
# Clonando o carro protótipo para criar um novo carro
carro2 = carro1.clone()
print(carro1.modelo, carro2.modelo)  # Ambos têm o mesmo modelo e ano
```

Java

```java
// Classe que representa o protótipo Carro
class Carro implements Cloneable {
    private String modelo;
    private int ano;

    public Carro(String modelo, int ano) {
        this.modelo = modelo;
        this.ano = ano;
    }

    // Método clone, permite copiar a instância de Carro
    @Override
    public Carro clone() throws CloneNotSupportedException {
        return (Carro) super.clone();
    }

    public String getModelo() {
        return modelo;
    }

    public int getAno() {
        return ano;
    }
}

// Testando o padrão Prototype
public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Carro carro1 = new Carro("Sedan", 2020);
        Carro carro2 = carro1.clone(); // Clonando o carro

        System.out.println(carro1.getModelo() + " - " + carro2.getModelo()); // Ambos têm o mesmo modelo
    }
}
```

Singleton

-Definição:

É um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

-Analogia:

Imagine uma presidência: em um país, há apenas um presidente em um dado momento. Se alguém quiser consultar ou interagir com o presidente, todos referem-se à mesma pessoa, pois existe apenas um "presidente" no sistema.

-Vantagens:

Garante que há apenas uma instância de uma classe; Facilita o acesso a uma instância global de forma controlada.

-Desvantagens:

Dificulta a testabilidade do código; Viola o princípio S do acrônimo SOLID.

-Exemplos em código:

Python

```python
class Singleton:
    _instance = None  # Armazena a única instância da classe

    def __new__(cls):
        # Se _instance é None, cria uma nova instância
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance  # Retorna a instância existente ou recém-criada

# Testando o Singleton
s1 = Singleton()
s2 = Singleton()
print(s1 == s2)  # True, pois ambos apontam para a mesma instância (Singleton)
```

Java

```java
public class Singleton {
    private static Singleton instance = null; // Armazena a única instância da classe

    private Singleton() {
        // Construtor privado impede a criação de instâncias fora da classe
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // Cria a instância se for a primeira vez
        }
        return instance; // Retorna a instância única
    }

    public void showMessage() {
        System.out.println("Singleton Pattern Example");
    }

    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();

        singleton1.showMessage();
        System.out.println(singleton1 == singleton2); // True, ambas são a mesma instância
    }
}
```

Padrões Estruturais

Adapter

-Definição:

É um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si. É um objeto especial que converte a interface de um objeto para que outro objeto possa entendê-lo.

-Analogia:

Imagine que você tem um computador que usa uma tomada de três pinos, mas você tem um dispositivo com um cabo de dois pinos. Um adaptador é usado para permitir que o dispositivo se conecte ao computador.

-Vantagens:

Implementa o princípio S do acrônimo SOLID, separando a conversão de interface ou de dados da lógica primária do negócio do programa; Implementa o princípio O do acrônimo SOLID, permitindo introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente

-Desvantagens:

Aumenta a complexidade do código ao inserir novas classes e interfaces; Pode diminuir o desempenho devido à camada extra de abstração.

-Exemplos em código:

Python

```python
class Alvo:
    def requisicao(self) -> str:
        return "Alvo: O comportamento padrão do alvo."


class Adaptado:
    """
    O Adaptado contém um comportamento útil, mas sua interface é incompatível
    com o código cliente existente. O Adaptado precisa de alguma adaptação antes
    que o código cliente possa usá-lo.
    """

    def requisicao_especifica(self) -> str:
        return ".eetpadA eht fo roivaheb laicepS"


class Adaptador(Alvo, Adaptado):
    """
    O Adaptador torna a interface do Adaptado compatível com a interface do Alvo
    através da herança múltipla.
    """

    def requisicao(self) -> str:
        return f"Adaptador: (TRADUZIDO) {self.requisicao_especifica()[::-1]}"
```

```python
def codigo_cliente(alvo: "Alvo") -> None:
    """
    O código cliente suporta todas as classes que seguem a interface do Alvo.
    """

    print(alvo.requisicao(), end="")


if __name__ == "__main__":
    print("Cliente: Eu posso trabalhar bem com objetos do Alvo:")
    alvo = Alvo()
    codigo_cliente(alvo)
    print("\n")

    adaptado = Adaptado()
    print("Cliente: A classe Adaptado tem uma interface estranha. "
          "Veja, eu não entendo isso:")
    print(f"Adaptado: {adaptado.requisicao_especifica()}", end="\n\n")

    print("Cliente: Mas eu posso trabalhar com isso através do Adaptador:")
    adaptador = Adaptador()
    codigo_cliente(adaptador)
```

Java

```java
package Estruturais;

// Classe Alvo: Define a interface específica do domínio
class Alvo {
    public String requisicao() {
        return "Alvo: O comportamento padrão do alvo.";
    }
}

// Classe Adaptado: Contém o comportamento útil, mas com uma interface incompatível
class Adaptado {
    public String requisicaoEspecifica() {
        return ".eetpadA eht fo roivaheb laicepS";
    }
}

// Classe Adaptador: Torna a interface do Adaptado compatível com a interface do Alvo
class Adaptador extends Alvo {
    private Adaptado adaptado;

    public Adaptador() {
        adaptado = new Adaptado();
    }

    @Override
    public String requisicao() {
        // Tradução do comportamento do Adaptado
        return "Adaptador: (TRADUZIDO) " + new StringBuilder(adaptado.requisicaoEspecifica()).reverse().toString();
    }
}
```

```java
// Classe Cliente
public class Cliente {
    public static void codigoCliente(Alvo alvo) {
        // O código cliente suporta todas as classes que seguem a interface Alvo
        System.out.print(alvo.requisicao());
    }

    public static void main(String[] args) {
        // Cliente: Eu posso trabalhar bem com objetos do Alvo
        System.out.println("Cliente: Eu posso trabalhar bem com objetos do Alvo:");
        Alvo alvo = new Alvo();
        codigoCliente(alvo);
        System.out.println("\n");

        // Cliente: A classe Adaptado tem uma interface estranha
        Adaptado adaptado = new Adaptado();
        System.out.println("Cliente: A classe Adaptado tem uma interface estranha. "
                + "Veja, eu não entendo isso:");
        System.out.println("Adaptado: " + adaptado.requisicaoEspecifica() + "\n");

        // Cliente: Mas eu posso trabalhar com isso através do Adaptador
        System.out.println("Cliente: Mas eu posso trabalhar com isso através do Adaptador:");
        Adaptador adaptador = new Adaptador();
        codigoCliente(adaptador);
    }
}
```

Bridge

-Definição:

É um padrão de projeto estrutural que permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.

-Analogia:

Imagine uma ponte que conecta duas ilhas, permitindo que veículos (abstração) se movam entre as ilhas, independentemente do tipo de veículo ou do tipo de ilha.

-Vantagens:

O código cliente trabalha com abstrações em alto nível, ele não fica exposto a detalhes de plataforma; Facilita a extensão do sistema, adicionando novas abstrações ou implementações; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Pode aumentar o número de classes no sistema; Pode ser mais complexo de implementar em comparação com soluções diretas.

-Exemplos em código:

Python

```python
# Interface do dispositivo que pode ser ligado e desligado
class Dispositivo:
    def ligar(self):
        pass

    def desligar(self):
        pass

# Implementação do dispositivo TV
class TV(Dispositivo):
    def ligar(self):
        print("TV ligada")

    def desligar(self):
        print("TV desligada")

# Controle remoto universal (abstração)
class ControleRemoto:
    def __init__(self, dispositivo):
        self.dispositivo = dispositivo

    def ligar(self):
        self.dispositivo.ligar()

    def desligar(self):
        self.dispositivo.desligar()

# Exemplo de uso
tv = TV()
controle = ControleRemoto(tv)
controle.ligar()  # TV ligada
controle.desligar()  # TV desligada
```

Java

```java
package Estruturais;

// Interface do dispositivo
interface Dispositivo {
    void ligar();
    void desligar();
}

// Implementação do dispositivo TV
class TV implements Dispositivo {
    public void ligar() {
        System.out.println("TV ligada");
    }

    public void desligar() {
        System.out.println("TV desligada");
    }
}

// Controle remoto que usa qualquer dispositivo que implemente a interface Dispositivo
class ControleRemoto {
    private Dispositivo dispositivo;

    public ControleRemoto(Dispositivo dispositivo) {
        this.dispositivo = dispositivo;
    }

    public void ligar() {
        dispositivo.ligar();
    }

    public void desligar() {
        dispositivo.desligar();
    }
}

public class Bridge {
    public static void main(String[] args) {
        Dispositivo tv = new TV();
        ControleRemoto controle = new ControleRemoto(tv);
        controle.ligar();  // TV ligada
        controle.desligar();  // TV desligada
    }
}
```

Composite

-Definição:

É um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.

-Analogia:

Pense em uma árvore com galhos e folhas. Cada galho e cada folha pode ser tratado de maneira similar, embora sejam objetos diferentes. O galho pode ter outros galhos (subcomponentes) ou folhas (componentes).

-Vantagens:

Permite tratar um conjunto de objetos de maneira uniforme/mais simples; Pode ser facilmente estendido para novos tipos de componentes; Implementa o princípio O do acrônimo SOLID.

-Desvantagens:

A estrutura de composição pode se tornar muito complexa com o aumento do número de componentes; Exige um bom planejamento da hierarquia de objetos.

-Exemplos em código:

Python

```python
# Componente básico
class Componente:
    def operar(self):
        pass

# Folha (um item indivisível)
class Folha(Componente):
    def operar(self):
        print("Folha operando")

# Galho (pode conter outros componentes)
class Galho(Componente):
    def __init__(self):
        self.componentes = []

    def adicionar(self, componente):
        self.componentes.append(componente)
```

```python
    def operar(self):
        print("Galho operando")
        for componente in self.componentes:
            componente.operar()

# Exemplo de uso: uma árvore com galhos e folhas
galho1 = Galho()
folha1 = Folha()
galho1.adicionar(folha1)

galho2 = Galho()
folha2 = Folha()
galho2.adicionar(folha2)

galhoPrincipal = Galho()
galhoPrincipal.adicionar(galho1)
galhoPrincipal.adicionar(galho2)

galhoPrincipal.operar()
```

Java

```java
package Estruturais;

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

// Componente
interface Componente {
    void operar();
}

// Folha
class Folha implements Componente {
    public void operar() {
        System.out.println("Folha operando");
    }
}

// Galho
class Galho implements Componente {
    private List<Componente> componentes = new ArrayList<>();

    public void adicionar(Componente componente) {
        componentes.add(componente);
    }

    public void operar() {
        System.out.println("Galho operando");
        for (Componente componente : componentes) {
            componente.operar();
        }
    }
}

public class Composite {
    public static void main(String[] args) {
        Galho galho1 = new Galho();
        Folha folha1 = new Folha();
        galho1.adicionar(folha1);

        Galho galho2 = new Galho();
        Folha folha2 = new Folha();
        galho2.adicionar(folha2);

        Galho galhoPrincipal = new Galho();
        galhoPrincipal.adicionar(galho1);
        galhoPrincipal.adicionar(galho2);

        galhoPrincipal.operar();  // Galho operando, Folha operando (para cada folha)
    }
}
```

Decorator

-Definição:

É um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos. “Envoltório” é o apelido alternativo para o padrão Decorator que expressa claramente a ideia principal dele. Um envoltório é um objeto que pode ser ligado com outro objeto alvo. O envoltório contém o mesmo conjunto de métodos que o alvo e delega a ele todos os pedidos que recebe. Contudo, o envoltório pode alterar o resultado fazendo alguma coisa ou antes ou depois de passar o pedido para o alvo.

-Analogia:

Imagine uma pizza simples. Você pode adicionar coberturas adicionais como queijo extra, cogumelos ou pepperoni. Cada cobertura é um "decorador" que adiciona um comportamento à pizza.

-Vantagens:

Permite adicionar funcionalidades em tempo de execução; Não é necessário criar uma nova classe para cada combinação de funcionalidades; Implementa o princípio S do acrônimo SOLID.

-Desvantagens:

É difícil remover um invólucro de uma pilha de invólucros; Pode ser difícil entender o fluxo de execução quando muitos decoradores são usados.

-Exemplos em código:

Python

```python
# Pizza base
class Pizza:
    def descricao(self):
        return "Pizza simples"

# Decorador base
class DecoradorPizza(Pizza):
    def __init__(self, pizza):
        self.pizza = pizza

    def descricao(self):
        return self.pizza.descricao()

# Decorador que adiciona queijo extra
class QueijoExtra(DecoradorPizza):
    def descricao(self):
        return self.pizza.descricao() + " com queijo extra"

# Decorador que adiciona cogumelos
class Cogumelos(DecoradorPizza):
    def descricao(self):
        return self.pizza.descricao() + " com cogumelos"

# Exemplo de uso
pizza = Pizza()
pizza_com_queijo = QueijoExtra(pizza)
pizza_com_queijo_e_cogumelos = Cogumelos(pizza_com_queijo)
print(pizza_com_queijo_e_cogumelos.descricao())  # Pizza simples com queijo extra com cogumelos
```

Java

```java
package Estruturais;

// Interface Pizza
interface Pizza {
    String descricao();
}

// Pizza simples
class PizzaSimples implements Pizza {
    public String descricao() {
        return "Pizza simples";
    }
}

// Decorador base
class DecoradorPizza implements Pizza {
    protected Pizza pizza;

    public DecoradorPizza(Pizza pizza) {
        this.pizza = pizza;
    }

    public String descricao() {
        return pizza.descricao();
    }
}

// Decorador queijo extra
class QueijoExtra extends DecoradorPizza {
    public QueijoExtra(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String descricao() {
        return pizza.descricao() + " com queijo extra";
    }
}

// Decorador cogumelos
class Cogumelos extends DecoradorPizza {
    public Cogumelos(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String descricao() {
        return pizza.descricao() + " com cogumelos";
    }
}

// Exemplo de uso
public class Decorator {
    public static void main(String[] args) {
        Pizza pizza = new PizzaSimples();
        Pizza pizzaComQueijo = new QueijoExtra(pizza);
        Pizza pizzaComQueijoECogumelos = new Cogumelos(pizzaComQueijo);

        System.out.println(pizzaComQueijoECogumelos.descricao());  // Pizza simples com queijo extra com cogumelos
    }
}
```

Facade

-Definição:

É um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes. Uma fachada é uma classe que fornece uma interface simples para um subsistema complexo que contém muitas partes que se movem. Uma fachada pode fornecer funcionalidades limitadas em comparação com trabalhar com os subsistemas diretamente. Contudo, ela inclui apenas aquelas funcionalidades que o cliente se importa.

Ter uma fachada é útil quando você precisa integrar sua aplicação com uma biblioteca sofisticada que tem dúzias de funcionalidades, mas você precisa de apenas um pouquinho delas.

-Analogia:

Imagine um controle remoto universal, que permite controlar vários dispositivos, como televisão, DVD e rádio. Ao invés de usar controles separados, você usa o controle remoto para interagir com todos eles.

-Vantagens:

Simplifica a interação com sistemas complexos; Reduz o acoplamento entre os subsistemas e o cliente.

-Desvantagens:

Pode esconder funcionalidades importantes dos subsistemas; Uma fachada pode se tornar um objeto deus acoplado a todas as classes de uma aplicação.

-Exemplos em código:

Python

```python
# Subsistemas complexos
class Amplifier:
    def on(self):
        print("Amplificador ligado.")

    def set_volume(self, volume):
        print(f"Volume ajustado para {volume}.")

class DvdPlayer:
    def on(self):
        print("DVD Player ligado.")

    def play_movie(self, movie):
        print(f"Reproduzindo o filme '{movie}'.")

class Projector:
    def on(self):
        print("Projetor ligado.")

    def set_input(self, source):
        print(f"Fonte de entrada do projetor definida como {source}.")

# Fachada
class HomeTheaterFacade:
    def __init__(self):
        self.amplifier = Amplifier()
        self.dvd_player = DvdPlayer()
        self.projector = Projector()

    def watch_movie(self, movie):
        print("Iniciando o sistema de home theater...")
        self.amplifier.on()
        self.amplifier.set_volume(10)
        self.dvd_player.on()
        self.dvd_player.play_movie(movie)
        self.projector.on()
        self.projector.set_input("DVD")
        print("Aproveite o filme!\n")

# Uso da Fachada
home_theater = HomeTheaterFacade()
home_theater.watch_movie("Inception")
```

Java

```java
package Estruturais;

// Subsistemas complexos
class Amplifier {
    public void on() {
        System.out.println("Amplificador ligado.");
    }

    public void setVolume(int volume) {
        System.out.println("Volume ajustado para " + volume + ".");
    }
}

class DvdPlayer {
    public void on() {
        System.out.println("DVD Player ligado.");
    }

    public void playMovie(String movie) {
        System.out.println("Reproduzindo o filme '" + movie + "'.");
    }
}

class Projector {
    public void on() {
        System.out.println("Projetor ligado.");
    }

    public void setInput(String source) {
        System.out.println("Fonte de entrada do projetor definida como " + source + ".");
    }
}

// Fachada
class HomeTheaterFacade {
    private Amplifier amplifier;
    private DvdPlayer dvdPlayer;
    private Projector projector;

    public HomeTheaterFacade() {
        this.amplifier = new Amplifier();
        this.dvdPlayer = new DvdPlayer();
        this.projector = new Projector();
    }

    public void watchMovie(String movie) {
        System.out.println("Iniciando o sistema de home theater...");
        amplifier.on();
        amplifier.setVolume(10);
        dvdPlayer.on();
        dvdPlayer.playMovie(movie);
        projector.on();
        projector.setInput("DVD");
        System.out.println("Aproveite o filme!\n");
    }
}

// Uso da Fachada
public class Facade {
    public static void main(String[] args) {
        HomeTheaterFacade homeTheater = new HomeTheaterFacade();
        homeTheater.watchMovie("Inception");
    }
}
```

Flyweight

-Definição:

É um padrão de projeto estrutural que permite a você colocar mais objetos na quantidade de RAM disponível ao compartilhar partes comuns de estado entre os múltiplos objetos ao invés de manter todos os dados em cada objeto.

-Analogia:

Imagine que você tem várias árvores em um parque. Em vez de criar uma árvore individual para cada folha, você pode criar um objeto "folha" compartilhado para todas as árvores, economizando espaço.

-Vantagens:

Otimiza o uso de memória RAM; Melhora a performance em sistemas com muitos objetos semelhantes.

-Desvantagens:

Pode ser difícil de implementar corretamente; Você pode estar trocando RAM por ciclos de CPU quando parte dos dados de contexto precisa ser recalculado cada vez que alguém chama um método flyweight.

-Exemplos em código:

Python

```python
# Objeto Flyweight compartilhado
class TreeType:
    def __init__(self, name, color, texture):
        self.name = name
        self.color = color
        self.texture = texture

    def draw(self, x, y):
        print(f"Desenhando a árvore '{self.name}' na posição ({x}, {y}) com cor '{self.color}' e textura '{self.texture}'.")

# Fábrica de Flyweight
class TreeFactory:
    _tree_types = {}

    @classmethod
    def get_tree_type(cls, name, color, texture):
        if (name, color, texture) not in cls._tree_types:
            cls._tree_types[(name, color, texture)] = TreeType(name, color, texture)
        return cls._tree_types[(name, color, texture)]

# Uso do Flyweight
class Tree:
    def __init__(self, x, y, tree_type):
        self.x = x
        self.y = y
        self.tree_type = tree_type

    def draw(self):
        self.tree_type.draw(self.x, self.y)

# Exemplo de uso
tree1 = Tree(10, 20, TreeFactory.get_tree_type("Pine", "Green", "Smooth"))
tree2 = Tree(30, 40, TreeFactory.get_tree_type("Pine", "Green", "Smooth"))
tree3 = Tree(15, 25, TreeFactory.get_tree_type("Oak", "Brown", "Rough"))

tree1.draw()
tree2.draw()
tree3.draw()
```

Java

```java
package Estruturais;

import java.util.HashMap;
import java.util.Map;

// Objeto Flyweight compartilhado
class TreeType {
    private String name;
    private String color;
    private String texture;

    public TreeType(String name, String color, String texture) {
        this.name = name;
        this.color = color;
        this.texture = texture;
    }

    public void draw(int x, int y) {
        System.out.println("Desenhando a árvore '" + name + "' na posição (" + x + ", " + y + ") com cor '" + color + "' e textura '" + texture + "'.");
    }
}

// Fábrica de Flyweight
class TreeFactory {
    private static final Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, String color, String texture) {
        String key = name + color + texture;
        if (!treeTypes.containsKey(key)) {
            treeTypes.put(key, new TreeType(name, color, texture));
        }
        return treeTypes.get(key);
    }
}

// Uso do Flyweight
class Tree {
    private int x, y;
    private TreeType treeType;

    public Tree(int x, int y, TreeType treeType) {
        this.x = x;
        this.y = y;
        this.treeType = treeType;
    }

    public void draw() {
        treeType.draw(x, y);
    }
}

// Exemplo de uso
public class Flyweight {
    public static void main(String[] args) {
        Tree tree1 = new Tree(10, 20, TreeFactory.getTreeType("Pine", "Green", "Smooth"));
        Tree tree2 = new Tree(30, 40, TreeFactory.getTreeType("Pine", "Green", "Smooth"));
        Tree tree3 = new Tree(15, 25, TreeFactory.getTreeType("Oak", "Brown", "Rough"));

        tree1.draw();
        tree2.draw();
        tree3.draw();
    }
}
```

Proxy

-Definição:

É um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original. O padrão Proxy sugere que você crie uma nova classe proxy com a mesma interface do objeto do serviço original. Então você atualiza sua aplicação para que ela passe o objeto proxy para todos os clientes do objeto original. Ao receber uma solicitação de um cliente, o proxy cria um objeto do serviço real e delega a ele todo o trabalho.

-Analogia:

Imagine que você precisa de permissão para acessar uma sala restrita. Um segurança (proxy) decide se você pode entrar ou não, baseando-se nas regras.

-Vantagens:

Permite controlar o objeto do serviço sem os clientes ficarem sabendo; Permite gerenciar o ciclo de vida de um objeto do serviço quando os clientes não se importam mais com ele; Implementa o princípio O do acrônimo SOLID.

-Desvantagens:

O código pode ficar mais complexo; Pode gerar um ponto de falha ou atrasos.

-Exemplos em código:

Python

```python
# Classe que representa o objeto real, neste caso o Banco de Dados.
class Database:
    def __init__(self):
        self.data = "Dados confidenciais do banco de dados"

    def read_data(self):
        print("Acessando o banco de dados...")
        return self.data

# Classe Proxy que controla o acesso ao banco de dados.
class DatabaseProxy:
    def __init__(self, user_role):
        self.user_role = user_role  # Define o papel do usuário (admin, visitante, etc.)
        self._real_database = None  # Inicialmente, o banco de dados real não é carregado.

    def read_data(self):
        # Verifica permissões antes de acessar o banco de dados real.
        if self.user_role == "admin":
            # Carrega o banco de dados real sob demanda, caso ainda não esteja carregado.
            if self._real_database is None:
                self._real_database = Database()
            return self._real_database.read_data()
        else:
            print("Acesso negado. Somente administradores podem acessar o banco de dados.")
            return None

# Exemplo de uso
admin_access = DatabaseProxy("admin")
print(admin_access.read_data())  # Acesso permitido e dados carregados.

visitor_access = DatabaseProxy("visitor")
print(visitor_access.read_data())  # Acesso negado devido a permissões insuficientes.
```

Java

```java
package Estruturais;

// Classe que representa o objeto real, neste caso o Banco de Dados.
class Database {
    private String data;

    public Database() {
        this.data = "Dados confidenciais do banco de dados";
    }

    public String readData() {
        System.out.println("Acessando o banco de dados...");
        return data;
    }
}

// Classe Proxy que controla o acesso ao banco de dados.
class DatabaseProxy {
    private String userRole;
    private Database realDatabase;  // Referência ao banco de dados real.

    public DatabaseProxy(String userRole) {
        this.userRole = userRole;  // Define o papel do usuário (admin, visitante, etc.)
    }

    public String readData() {
        // Verifica permissões antes de acessar o banco de dados real.
        if ("admin".equals(userRole)) {
            // Carrega o banco de dados real sob demanda, caso ainda não esteja carregado.
            if (realDatabase == null) {
                realDatabase = new Database();
            }
            return realDatabase.readData();
        } else {
            System.out.println("Acesso negado. Somente administradores podem acessar o banco de dados.");
            return null;
        }
    }
}

// Exemplo de uso
public class Proxy {
    public static void main(String[] args) {
        DatabaseProxy adminAccess = new DatabaseProxy("admin");
        System.out.println(adminAccess.readData());  // Acesso permitido e dados carregados.

        DatabaseProxy visitorAccess = new DatabaseProxy("visitor");
        System.out.println(visitorAccess.readData());  // Acesso negado devido a permissões insuficientes.
    }
}
```

Padrões Comportamentais

Chain of Responsibility

-Definição:

É um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente. Como muitos outros padrões de projeto comportamental, o Chain of Responsibility se baseia em transformar certos comportamentos em objetos solitários chamados handlers. O padrão sugere que você ligue esses handlers em uma corrente. Cada handler ligado tem um campo para armazenar uma referência ao próximo handler da corrente. Além de processar o pedido, handlers o passam adiante na corrente. O pedido viaja através da corrente até que todos os handlers tiveram uma chance de processá-lo.

-Analogia:

Imagine uma linha de montagem onde cada estação é responsável por uma verificação específica. Se uma estação não consegue resolver um problema, ela passa o item para a próxima estação, até que seja resolvido ou rejeitado.

-Vantagens:

Reduz a dependência entre remetente e receptor do pedido; Permite adicionar ou alterar receptores na cadeia sem modificar os clientes; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Pode ser difícil rastrear onde o pedido foi processado; Alguns pedidos podem acabar sem tratamento.

-Exemplos em código:

Python

```python
# Definindo uma classe base para manipuladores
class Handler:
    def __init__(self, successor=None):
        self.successor = successor  # Próximo manipulador na cadeia

    def handle_request(self, request):
        if self.successor:
            self.successor.handle_request(request)

# Implementando manipuladores específicos
class LowLevelHandler(Handler):
    def handle_request(self, request):
        if request < 10:
            print(f"LowLevelHandler: Processando o pedido {request}")
        else:
            print("LowLevelHandler: Passando para o próximo.")
            super().handle_request(request)

class HighLevelHandler(Handler):
    def handle_request(self, request):
        if request >= 10:
            print(f"HighLevelHandler: Processando o pedido {request}")
        else:
            print("HighLevelHandler: Não pode processar.")

# Configuração da cadeia de responsabilidade
handler = LowLevelHandler(HighLevelHandler())
handler.handle_request(5)  # Espera-se que seja processado pelo LowLevelHandler
handler.handle_request(20)  # Espera-se que seja processado pelo HighLevelHandler
```

Java

```java
package Comportamentais;

// Definindo uma classe abstrata para manipuladores
abstract class Handler {
    protected Handler successor;

    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }

    public abstract void handleRequest(int request);
}

// Implementando manipuladores específicos
class LowLevelHandler extends Handler {
    @Override
    public void handleRequest(int request) {
        if (request < 10) {
            System.out.println("LowLevelHandler: Processando o pedido " + request);
        } else {
            System.out.println("LowLevelHandler: Passando para o próximo.");
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}

class HighLevelHandler extends Handler {
    @Override
    public void handleRequest(int request) {
        if (request >= 10) {
            System.out.println("HighLevelHandler: Processando o pedido " + request);
        } else {
            System.out.println("HighLevelHandler: Não pode processar.");
        }
    }
}

// Configuração da cadeia de responsabilidade
public class ChainOfResponsibility {
    public static void main(String[] args) {
        Handler lowLevelHandler = new LowLevelHandler();
        Handler highLevelHandler = new HighLevelHandler();
        lowLevelHandler.setSuccessor(highLevelHandler);

        lowLevelHandler.handleRequest(5);  // Processado pelo LowLevelHandler
        lowLevelHandler.handleRequest(20); // Processado pelo HighLevelHandler
    }
}
```

Command

-Definição:

É um padrão de projeto comportamental que transforma um pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas. O padrão Command sugere que os objetos GUI não enviem esses pedidos diretamente. Ao invés disso, você deve extrair todos os detalhes do pedido, tais como o objeto a ser chamado, o nome do método, e a lista de argumentos em uma classe comando separada que tem apenas um método que aciona esse pedido.

-Analogia:

Imagine um restaurante onde os garçons recebem pedidos dos clientes e os entregam aos cozinheiros. O pedido é tratado como uma instrução que pode ser executada (ou cancelada).

-Vantagens:

Desacoplamento entre remetente e executor; Capacidade de reverter operações ao armazenar históricos; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Aumento no número de classes para cada comando; Implementação pode ser desnecessária para comandos simples.

-Exemplos em código:

Python

```python
# Classe de comando
class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def execute(self):
        print("A luz foi acesa!")

class LightOffCommand(Command):
    def execute(self):
        print("A luz foi apagada!")

class RemoteControl:
    def submit(self, command):
        command.execute()

# Utilização do comando
control = RemoteControl()
control.submit(LightOnCommand())
control.submit(LightOffCommand())
```

Java

```java
package Comportamentais;

// Interface de comando
interface Comm {
    void execute();
}

class LightOnCommand implements Comm {
    public void execute() {
        System.out.println("A luz foi acesa!");
    }
}

class LightOffCommand implements Comm {
    public void execute() {
        System.out.println("A luz foi apagada!");
    }
}

class RemoteControl {
    public void submit(Comm command) {
        command.execute();
    }
}

// Utilização do comando
public class Command {
    public static void main(String[] args) {
        RemoteControl control = new RemoteControl();
        control.submit(new LightOnCommand());
        control.submit(new LightOffCommand());
    }
}
```

Iterator

-Definição:

É um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.). A ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado chamado um iterador.

-Analogia:

Pense em um controle remoto de uma apresentação de slides, onde você pode avançar ou retroceder pelos slides sem saber como eles estão armazenados.

-Vantagens:

Oculta a estrutura interna da coleção (encapsulamento); Facilita a navegação em várias coleções sem modificar a implementação; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Usar um iterador pode ser menos eficiente que percorrer elementos de algumas coleções especializadas diretamente; Necessidade de criar classes de iterador para cada coleção personalizada.

-Exemplos em código:

Python

```python
class ConcreteIterator:
    def __init__(self, collection):
        self._collection = collection  # Atribui a coleção à variável interna
        self._index = 0  # Inicializa o índice para rastrear o próximo item

    def __next__(self):
        # Verifica se ainda há elementos na coleção
        if self._index < len(self._collection):
            item = self._collection[self._index]  # Obtém o item atual
            self._index += 1  # Avança para o próximo item
            return item
        else:
            # Lança uma exceção quando todos os itens foram percorridos
            raise StopIteration

class ConcreteCollection:
    def __init__(self):
        self._items = []  # Cria uma lista interna para armazenar itens

    def add_item(self, item):
        self._items.append(item)  # Adiciona um item à coleção

    def __iter__(self):
        # Retorna um iterador que percorre os itens da coleção
        return ConcreteIterator(self._items)

# Uso do iterador
collection = ConcreteCollection()
collection.add_item(1)
collection.add_item(2)
collection.add_item(3)
for item in collection:
    print(item)  # Saída: 1, 2, 3
```

Java

```java
package Comportamentais;

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

// Classe que representa a coleção de elementos
class ConcreteCollection implements Iterable<Integer> {
    private List<Integer> items = new ArrayList<>();  // Lista interna para armazenar itens

    public void addItem(int item) {
        items.add(item);  // Adiciona um item à coleção
    }

    @Override
    public Iterator<Integer> iterator() {
        // Retorna o iterador da lista, ocultando a estrutura interna
        return items.iterator();
    }
}

// Uso do iterador
public class MainIterator {
    public static void main(String[] args) {
        ConcreteCollection collection = new ConcreteCollection();
        collection.addItem(1);
        collection.addItem(2);
        collection.addItem(3);

        // Itera sobre os itens da coleção usando o iterador
        for (int item : collection) {
            System.out.println(item);  // Saída: 1, 2, 3
        }
    }
}

```

Mediator

-Definição:

É um padrão de projeto comportamental que permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador. O padrão Mediator sugere que você deveria cessar toda comunicação direta entre componentes que você quer tornar independentes um do outro. Ao invés disso, esses componentes devem colaborar indiretamente, chamando um objeto mediador especial que redireciona as chamadas para os componentes apropriados

-Analogia:

Considere um aeroporto onde a torre de controle é responsável por coordenar as comunicações entre aeronaves, evitando que se comuniquem diretamente.

-Vantagens:

Reduz as dependências diretas entre objetos; Permite reutilizar componentes individuais mais facilmente; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Pode criar um ponto único de falha se o mediador falhar; Pode evoluir para um objeto Deus.

-Exemplos em código:

Python

```python
class Mediator:
    def notify(self, sender, event):
        pass  # Define a interface de notificação para o Mediador

class ConcreteMediator(Mediator):
    def __init__(self, component1, component2):
        self.component1 = component1
        self.component2 = component2
        # Cada componente conhece o mediador, promovendo o desacoplamento
        self.component1._mediator = self
        self.component2._mediator = self

    def notify(self, sender, event):
        # Implementa a lógica de comunicação entre componentes
        if event == "A":
            print("Mediator reage ao evento A e dispara o evento B")
            self.component2.do_b()
        elif event == "B":
            print("Mediator reage ao evento B e dispara o evento A")
            self.component1.do_a()

class Component1:
    def __init__(self, mediator=None):
        self._mediator = mediator  # Define o mediador para o componente

    def do_a(self):
        print("Componente 1 executa A.")
        # Notifica o mediador sobre o evento A
        self._mediator.notify(self, "A")

class Component2:
    def __init__(self, mediator=None):
        self._mediator = mediator  # Define o mediador para o componente

    def do_b(self):
        print("Componente 2 executa B.")
        # Notifica o mediador sobre o evento B
        self._mediator.notify(self, "B")

# Configuração do mediador
mediator = ConcreteMediator(Component1(), Component2())
mediator.component1.do_a()  # Inicia o fluxo com o evento A
```

Java

```java
package Comportamentais;

// Interface Mediator que define o método de notificação
interface Mediator {
    void notify(Object sender, String event);
}

// Implementação concreta do mediador
class ConcreteMediator implements Mediator {
    private Component1 component1;
    private Component2 component2;

    public ConcreteMediator(Component1 component1, Component2 component2) {
        this.component1 = component1;
        this.component2 = component2;
        // Define o mediador para ambos os componentes
        this.component1.setMediator(this);
        this.component2.setMediator(this);
    }

    public void notify(Object sender, String event) {
        // Reage ao evento e aciona as ações dos componentes
        if ("A".equals(event)) {
            System.out.println("Mediator reage ao evento A e dispara o evento B");
            component2.doB();
        } else if ("B".equals(event)) {
            System.out.println("Mediator reage ao evento B e dispara o evento A");
            component1.doA();
        }
    }
}

// Componente 1 que interage através do mediador
class Component1 {
    private Mediator mediator;

    public void setMediator(Mediator mediator) {
        this.mediator = mediator;  // Define o mediador para o componente
    }

    public void doA() {
        System.out.println("Componente 1 executa A.");
        // Notifica o mediador sobre o evento A
        if (mediator != null) {
            mediator.notify(this, "A");
        }
    }
}

// Componente 2 que interage através do mediador
class Component2 {
    private Mediator mediator;

    public void setMediator(Mediator mediator) {
        this.mediator = mediator;  // Define o mediador para o componente
    }

    public void doB() {
        System.out.println("Componente 2 executa B.");
        // Notifica o mediador sobre o evento B
        if (mediator != null) {
            mediator.notify(this, "B");
        }
    }
}

// Configuração do mediador e inicialização do fluxo de eventos
public class MainMediator {
    public static void main(String[] args) {
        Component1 component1 = new Component1();
        Component2 component2 = new Component2();
        ConcreteMediator mediator = new ConcreteMediator(component1, component2);

        component1.doA();  // Inicia o fluxo com o evento A
    }
}
```

Memento

-Definição:

É um padrão de projeto comportamental que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação. O padrão Memento delega a criação dos retratos do estado para o próprio dono do estado, o objeto originador.

-Analogia:

Imagine um videogame que permite salvar o progresso em determinado ponto. O jogador pode, então, restaurar o progresso salvo caso deseje voltar para aquela parte do jogo.

-Vantagens:

Permite produzir retratos do estado de um objeto sem violar seu encapsulamento; Permite simplificar o código da originadora permitindo que a cuidadora mantenha o histórico do estado da originadora.

-Desvantagens:

Pode consumir muita RAM se os clientes criarem mementos com muita frequência; A maioria das linguagens de programação dinâmicas, tais como PHP, Python, e JavaScript, não conseguem garantir que o estado dentro do memento permaneça intacto.

-Exemplos em código:

Python

```python
class Memento:
    def __init__(self, state):
        self._state = state  # Armazena o estado do objeto no memento

    def get_state(self):
        return self._state  # Retorna o estado armazenado no memento

class Originator:
    def __init__(self):
        self._state = ""  # Estado inicial do objeto

    def set_state(self, state):
        print(f"Estado definido para: {state}")
        self._state = state  # Define o estado atual do objeto

    def save_state_to_memento(self):
        return Memento(self._state)  # Cria um memento para salvar o estado atual

    def get_state_from_memento(self, memento):
        self._state = memento.get_state()  # Restaura o estado do memento
        print(f"Estado restaurado para: {self._state}")

# Uso do padrão Memento
originator = Originator()
originator.set_state("Estado1")  # Define um estado
memento = originator.save_state_to_memento()  # Salva o estado atual no memento
originator.set_state("Estado2")  # Altera o estado

originator.get_state_from_memento(memento)  # Restaura o estado anterior a partir do memento
```

Java

```java
package Comportamentais;

class Memento {
    private final String state;  // Armazena o estado do objeto no memento

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;  // Retorna o estado armazenado no memento
    }
}

class Originator {
    private String state;  // Estado atual do objeto

    public void setState(String state) {
        System.out.println("Estado definido para: " + state);
        this.state = state;  // Define o estado do objeto
    }

    public Memento saveStateToMemento() {
        return new Memento(state);  // Cria um memento para salvar o estado atual
    }

    public void getStateFromMemento(Memento memento) {
        state = memento.getState();  // Restaura o estado a partir do memento
        System.out.println("Estado restaurado para: " + state);
    }
}

// Uso do padrão Memento
public class MainMemento {
    public static void main(String[] args) {
        Originator originator = new Originator();
        originator.setState("Estado1");  // Define um estado
        Memento memento = originator.saveStateToMemento();  // Salva o estado atual no memento
        originator.setState("Estado2");  // Altera o estado

        originator.getStateFromMemento(memento);  // Restaura o estado anterior a partir do memento
    }
}
```

Observer

-Definição:

É um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

-Analogia:

Pense em uma conta de redes sociais. Quando um usuário faz uma postagem, todos os seus seguidores recebem uma notificação ou atualização.

-Vantagens:

Os objetos observadores não precisam conhecer os detalhes internos do objeto observado; Permite adicionar e remover observadores facilmente; Implementa o princípio O do acrônimo SOLID.

-Desvantagens:

Não há garantia de ordem de execução entre observadores; Pode gerar alto tráfego de notificações se muitos observadores estiverem conectados.

-Exemplos em código:

Python

```python
class Subject:
    def __init__(self):
        self._observers = []  # Lista para armazenar os observadores

    def attach(self, observer):
        self._observers.append(observer)  # Adiciona um observador

    def detach(self, observer):
        self._observers.remove(observer)  # Remove um observador

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)  # Notifica todos os observadores

class Observer:
    def update(self, message):
        pass  # Método a ser implementado pelos observadores concretos

class ConcreteObserver(Observer):
    def update(self, message):
        print(f"Notificação recebida: {message}")  # Imprime a notificação recebida

# Uso do padrão Observer
subject = Subject()
observer1 = ConcreteObserver()
observer2 = ConcreteObserver()

subject.attach(observer1)  # Anexa o primeiro observador
subject.attach(observer2)  # Anexa o segundo observador

subject.notify("Nova atualização!")  # Notifica todos os observadores anexados
```

Java

```java
package Comportamentais;

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

interface Observer {
    void update(String message);  // Método para notificação
}

class ConcreteObserver implements Observer {
    public void update(String message) {
        System.out.println("Notificação recebida: " + message);  // Exibe a notificação recebida
    }
}

class Subject {
    private List<Observer> observers = new ArrayList<>();  // Lista para armazenar os observadores

    public void attach(Observer observer) {
        observers.add(observer);  // Adiciona um observador
    }

    public void detach(Observer observer) {
        observers.remove(observer);  // Remove um observador
    }

    public void notify(String message) {
        for (Observer observer : observers) {
            observer.update(message);  // Notifica todos os observadores
        }
    }
}

// Uso do padrão Observer
public class MainObserver {
    public static void main(String[] args) {
        Subject subject = new Subject();
        Observer observer1 = new ConcreteObserver();
        Observer observer2 = new ConcreteObserver();

        subject.attach(observer1);  // Anexa o primeiro observador
        subject.attach(observer2);  // Anexa o segundo observador

        subject.notify("Nova atualização!");  // Notifica todos os observadores anexados
    }
}
```

State

-Definição:

É um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe. O padrão State sugere que você crie novas classes para todos os estados possíveis de um objeto e extraia todos os comportamentos específicos de estados para dentro dessas classes.

-Analogia:

Imagine um semáforo que muda de estado entre "verde", "amarelo" e "vermelho", alterando o comportamento de acordo com o estado atual.

-Vantagens:

Separa cada comportamento específico em classes diferentes; Adicionar novos estados é mais fácil sem modificar o código existente; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

O padrão pode ser um exagero se a máquina de estado só tem alguns estados ou raramente muda eles; Pode aumentar a quantidade de objetos instanciados.

-Exemplos em código:

Python

```python
class State:
    def handle(self):
        pass  # Método abstrato para implementação dos estados concretos

class ConcreteStateA(State):
    def handle(self):
        print("Estado A: Executando ação")  # Comportamento específico para o estado A

class ConcreteStateB(State):
    def handle(self):
        print("Estado B: Executando ação")  # Comportamento específico para o estado B

class Context:
    def __init__(self, state):
        self._state = state  # Armazena o estado atual

    def set_state(self, state):
        self._state = state  # Altera o estado do contexto

    def request(self):
        self._state.handle()  # Executa a ação baseada no estado atual

# Uso do padrão State
context = Context(ConcreteStateA())
context.request()  # Estado A: Executando ação

context.set_state(ConcreteStateB())
context.request()  # Estado B: Executando ação
```

Java

```java
package Comportamentais;

interface State {
    void handle();  // Método abstrato para implementação dos estados concretos
}

class ConcreteStateA implements State {
    public void handle() {
        System.out.println("Estado A: Executando ação");  // Comportamento específico para o estado A
    }
}

class ConcreteStateB implements State {
    public void handle() {
        System.out.println("Estado B: Executando ação");  // Comportamento específico para o estado B
    }
}

class Context {
    private State state;  // Armazena o estado atual

    public Context(State state) {
        this.state = state;  // Inicializa o estado do contexto
    }

    public void setState(State state) {
        this.state = state;  // Altera o estado do contexto
    }

    public void request() {
        state.handle();  // Executa a ação baseada no estado atual
    }
}

// Uso do padrão State
public class MainState {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStateA());
        context.request();  // Estado A: Executando ação

        context.setState(new ConcreteStateB());
        context.request();  // Estado B: Executando ação
    }
}
```

Strategy

-Definição:

É um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis. O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas estratégias. A classe original, chamada contexto, deve ter um campo para armazenar uma referência para um dessas estratégias.

-Analogia:

Imagine um navegador GPS que oferece múltiplas rotas para um destino: a mais rápida, a mais curta e a mais econômica. Você pode selecionar a estratégia de rota conforme sua preferência, e o GPS adapta o cálculo de acordo.

-Vantagens:

Permite mudar o algoritmo em tempo de execução; Evita condicionais complexas, mantendo cada estratégia isolada em uma classe; Implementa o princípio O do acrônimo SOLID.

-Desvantagens:

Pode resultar em muitas classes adicionais para cada estratégia; Se as estratégias são semelhantes, pode-se gerar complexidade desnecessária.

-Exemplos em código:

Python

```python
class Strategy:
    # Interface da estratégia
    def execute(self, data):
        pass

class ConcreteStrategyA(Strategy):
    # Estratégia concreta A
    def execute(self, data):
        print(f"Executando estratégia A no dado: {data}")

class ConcreteStrategyB(Strategy):
    # Estratégia concreta B
    def execute(self, data):
        print(f"Executando estratégia B no dado: {data}")

class Context:
    # Contexto que usa a estratégia
    def __init__(self, strategy: Strategy):
        self._strategy = strategy

    def set_strategy(self, strategy: Strategy):
        # Permite mudar a estratégia em tempo de execução
        self._strategy = strategy

    def do_something(self, data):
        # Executa a estratégia atual
        self._strategy.execute(data)

# Uso do Strategy
context = Context(ConcreteStrategyA())  # Inicia com Estratégia A
context.do_something("Informação")  # Executa estratégia A

context.set_strategy(ConcreteStrategyB())  # Muda para Estratégia B
context.do_something("Informação")  # Executa estratégia B
```

Java

```java
package Comportamentais;

interface Strategy {
    // Interface da estratégia
    void execute(String data);
}

class ConcreteStrategyA implements Strategy {
    public void execute(String data) {
        // Estratégia concreta A
        System.out.println("Executando estratégia A no dado: " + data);
    }
}

class ConcreteStrategyB implements Strategy {
    public void execute(String data) {
        // Estratégia concreta B
        System.out.println("Executando estratégia B no dado: " + data);
    }
}

class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(Strategy strategy) {
        // Permite alterar a estratégia em tempo de execução
        this.strategy = strategy;
    }

    public void doSomething(String data) {
        // Executa a estratégia atual
        strategy.execute(data);
    }
}

// Uso do Strategy
public class MainStrategy {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStrategyA());  // Inicia com Estratégia A
        context.doSomething("Informação");  // Executa estratégia A

        context.setStrategy(new ConcreteStrategyB());  // Muda para Estratégia B
        context.doSomething("Informação");  // Executa estratégia B
    }
}
```

Template Method

-Definição:

É um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura. O padrão do Template Method sugere que você quebre um algoritmo em uma série de etapas, transforme essas etapas em métodos, e coloque uma série de chamadas para esses métodos dentro de um único método padrão.

-Analogia:

Imagine uma receita de bolo. A sequência de passos é sempre a mesma (pré-aquecer o forno, preparar a massa, assar), mas os detalhes da receita (ingredientes, tempo de forno) podem variar conforme o tipo de bolo.

-Vantagens:

Permite a reutilização de código, onde a estrutura do algoritmo pode ser reutilizada em várias implementações; Mudanças na estrutura geral do algoritmo são centralizadas na classe base.

-Desvantagens:

Viola o princípio L do acrônimo SOLID, ao suprimir uma etapa padrão de implementação através da subclasse; A ordem dos passos é fixa, o que pode ser limitante.

-Exemplos em código:

Python

```python
class AbstractClass:
    # Método template que define o esqueleto do algoritmo
    def template_method(self):
        self.step_one()
        self.step_two()
        self.hook()  # Gancho opcional para subclasses

    def step_one(self):
        # Passo fixo para todas as subclasses
        print("Passo 1: fixo")

    def step_two(self):
        # Passo a ser implementado pelas subclasses
        pass

    def hook(self):
        # Gancho opcional que pode ser sobrescrito
        pass

class ConcreteClassA(AbstractClass):
    # Implementação do passo 2
    def step_two(self):
        print("Passo 2: implementado pela Classe A")
```

```python
class ConcreteClassB(AbstractClass):
    # Implementação do passo 2 e do gancho
    def step_two(self):
        print("Passo 2: implementado pela Classe B")

    def hook(self):
        print("Gancho implementado pela Classe B")

# Uso do Template Method
a = ConcreteClassA()
b = ConcreteClassB()

a.template_method()
# Saída: Passo 1: fixo
# Passo 2: implementado pela Classe A

b.template_method()
# Saída: Passo 1: fixo
# Passo 2: implementado pela Classe B
# Gancho implementado pela Classe B
```

Java

```java
package Comportamentais;

abstract class AbstractClass {
    // Método template que define o esqueleto do algoritmo
    public final void templateMethod() {
        stepOne();
        stepTwo();
        hook();  // Gancho opcional para subclasses
    }

    public void stepOne() {
        // Passo fixo para todas as subclasses
        System.out.println("Passo 1: fixo");
    }

    public abstract void stepTwo();  // Passo a ser implementado pelas subclasses

    public void hook() {
        // Gancho opcional que pode ser sobrescrito
    }
}

class ConcreteClassA extends AbstractClass {
    // Implementação do passo 2
    public void stepTwo() {
        System.out.println("Passo 2: implementado pela Classe A");
    }
}
```

```java
class ConcreteClassB extends AbstractClass {
    // Implementação do passo 2 e do gancho
    public void stepTwo() {
        System.out.println("Passo 2: implementado pela Classe B");
    }

    public void hook() {
        System.out.println("Gancho implementado pela Classe B");
    }
}

// Uso do Template Method
public class TemplateMethod {
    public static void main(String[] args) {
        AbstractClass a = new ConcreteClassA();
        AbstractClass b = new ConcreteClassB();

        a.templateMethod();
        // Saída: Passo 1: fixo
        // Passo 2: implementado pela Classe A

        b.templateMethod();
        // Saída: Passo 1: fixo
        // Passo 2: implementado pela Classe B
        // Gancho implementado pela Classe B
    }
}
```

Visitor

-Definição:

É um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam. O padrão Visitor sugere que você coloque o novo comportamento em uma classe separada chamada visitante, ao invés de tentar integrá-lo em classes já existentes. O objeto original que teve que fazer o comportamento é agora passado para um dos métodos da visitante como um argumento, desde que o método acesse todos os dados necessários contidos dentro do objeto.

-Analogia:

Imagine um museu com várias exposições (objetos). O visitante (Visitor) pode explorar diferentes exposições, realizando diferentes operações (como tirar fotos, fazer anotações, etc.) sem alterar as exposições em si. O visitante só precisa entender o tipo de cada exposição e o que pode fazer com ela.

-Vantagens:

Você pode adicionar novas funcionalidades sem alterar o código das classes dos objetos; Permite que uma única operação seja aplicada a uma grande quantidade de tipos de objetos diferentes; Implementa os princípios S e O do acrônimo SOLID.

-Desvantagens:

Sempre que a estrutura de objetos mudar, é necessário atualizar o Visitor; Visitantes podem não ter seu acesso permitido para campos e métodos privados dos elementos que eles deveriam estar trabalhando.

-Exemplos em código:

Python

```python
# Classe Element base que aceita um visitante
class Element:
    def accept(self, visitor):
        pass  # Método que permite que o visitante realize uma operação no elemento

# Elemento específico A
class ConcreteElementA(Element):
    def accept(self, visitor):
        visitor.visit_concrete_element_a(self)  # Chama o método específico do visitante para o Elemento A

# Elemento específico B
class ConcreteElementB(Element):
    def accept(self, visitor):
        visitor.visit_concrete_element_b(self)  # Chama o método específico do visitante para o Elemento B

# Interface Visitor com métodos para cada tipo de elemento
class Visitor:
    def visit_concrete_element_a(self, element):
        pass  # Visita um ConcreteElementA

    def visit_concrete_element_b(self, element):
        pass  # Visita um ConcreteElementB

# Visitante específico 1 que implementa as operações para os elementos
class ConcreteVisitor1(Visitor):
    def visit_concrete_element_a(self, element):
        print("Visitor 1 visitou Elemento A")  # Operação realizada no Elemento A

    def visit_concrete_element_b(self, element):
        print("Visitor 1 visitou Elemento B")  # Operação realizada no Elemento B

# Visitante específico 2 que implementa as operações para os elementos
class ConcreteVisitor2(Visitor):
    def visit_concrete_element_a(self, element):
        print("Visitor 2 visitou Elemento A")  # Operação realizada no Elemento A

    def visit_concrete_element_b(self, element):
        print("Visitor 2 visitou Elemento B")  # Operação realizada no Elemento B

# Uso do padrão Visitor
elements = [ConcreteElementA(), ConcreteElementB()]  # Lista de elementos
visitor1 = ConcreteVisitor1()  # Visitante 1
visitor2 = ConcreteVisitor2()  # Visitante 2

# Para cada elemento, aplicamos ambos os visitantes
for element in elements:
    element.accept(visitor1)  # Visitor 1 visita os elementos
    element.accept(visitor2)  # Visitor 2 visita os mesmos elementos
```

Java

```java
package Comportamentais;

// Interface Element que define o método accept para receber um visitante
interface Element {
    void accept(Visitor visitor);  // Método que permite que o visitante realize uma operação no elemento
}

// ConcreteElementA que implementa a operação accept
class ConcreteElementA implements Element {
    public void accept(Visitor visitor) {
        visitor.visitConcreteElementA(this);  // Chama o método do visitante específico para o Elemento A
    }
}

// ConcreteElementB que implementa a operação accept
class ConcreteElementB implements Element {
    public void accept(Visitor visitor) {
        visitor.visitConcreteElementB(this);  // Chama o método do visitante específico para o Elemento B
    }
}

// Interface Visitor com métodos para cada tipo de Elemento
interface Visitor {
    void visitConcreteElementA(ConcreteElementA element);  // Visita um ConcreteElementA
    void visitConcreteElementB(ConcreteElementB element);  // Visita um ConcreteElementB
}

// Visitante específico 1 que implementa os métodos de visitação
class ConcreteVisitor1 implements Visitor {
    public void visitConcreteElementA(ConcreteElementA element) {
        System.out.println("Visitor 1 visitou Elemento A");  // Operação realizada no Elemento A
    }

    public void visitConcreteElementB(ConcreteElementB element) {
        System.out.println("Visitor 1 visitou Elemento B");  // Operação realizada no Elemento B
    }
}

// Visitante específico 2 que implementa os métodos de visitação
class ConcreteVisitor2 implements Visitor {
    public void visitConcreteElementA(ConcreteElementA element) {
        System.out.println("Visitor 2 visitou Elemento A");  // Operação realizada no Elemento A
    }

    public void visitConcreteElementB(ConcreteElementB element) {
        System.out.println("Visitor 2 visitou Elemento B");  // Operação realizada no Elemento B
    }
}

// Uso do padrão Visitor
public class MainVisitor {
    public static void main(String[] args) {
        // Lista de elementos
        Element[] elements = {new ConcreteElementA(), new ConcreteElementB()};

        // Visitantes
        Visitor visitor1 = new ConcreteVisitor1();  // Criando visitante 1
        Visitor visitor2 = new ConcreteVisitor2();  // Criando visitante 2

        // Para cada elemento, ambos os visitantes são aplicados
        for (Element element : elements) {
            element.accept(visitor1);  // Visitor 1 visita os elementos
            element.accept(visitor2);  // Visitor 2 visita os mesmos elementos
        }
    }
}
```

Considerações Finais

Com este projeto pude perceber como os Desing Patterns são fundamentais para o desenvolvimento de sistemas bem estruturados. Antes de entender completamente os padrões, eu costumava resolver os problemas no decorrer do desenvolvimento de maneira mais improvisada, sem pensar nas consequências a longo prazo, agora, vejo como essas soluções reutilizáveis são importantes, especialmente quando estamos lidando com sistemas grandes e complexos. Cada padrão oferece uma maneira testada e comprovada de resolver um tipo específico de problema, como quando precisamos criar objetos, controlar o fluxo de comunicação entre diferentes partes do sistema ou até mesmo gerenciar alterações sem causar muitos efeitos colaterais.

Uma das coisas que mais me chamou a atenção é como os padrões podem aumentar a produtividade. Quando você já sabe qual padrão usar, pode focar mais em como implementar a lógica do sistema, sem precisar se preocupar tanto com a estrutura de baixo nível ou com soluções improvisadas. Além disso, como os padrões de projeto são amplamente conhecidos, a comunicação entre desenvolvedores fica muito mais fácil.

Por outro lado, ao estudar os padrões de projeto, também notei que não podemos simplesmente sair aplicando qualquer padrão para todos os problemas. Às vezes, um padrão pode ser excessivo para o problema em questão, e ele pode acabar complicando o código em vez de simplificá-lo. Isso é algo que nós desenvolvedores precisamos ter em mente: escolher o padrão certo para a situação certa.

Por fim, acredito que os padrões de projeto ajudam os desenvolvedores a escrever códigos mais limpos, eficientes e de fácil manutenção. Além disso, eles facilitam a colaboração entre membros da equipe e tornam o processo de desenvolvimento mais rápido e menos propenso a erros. Agora que entendo melhor os padrões, estou mais confiante para aplicá-los em meus próprios projetos e seguir melhores práticas de design de software.