Projeto de Software

PADRÕES DE PROJETO: ESTRUTURA

Artur Monteiro Parreiras
Brenda Stefany de Oliveira Rocha
Bruna Letícia Silva
Eduarda Faria Pinheiro
Elenice Florentina de Oliveira dos Reis
Marco Aurélio Faria Ramos

ADAPTER

Definição:

Conceito

O Adapter (Adaptador) é um padrão estrutural que permite que objetos com interfaces incompatíveis trabalhem juntos.

Ele atua como um tradutor, convertendo a interface de uma classe existente em uma interface esperada pelos clientes.

Definição (GoF):

“Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem juntas.”

Definição (GoF):

“Converte a interface de uma classe em outra interface esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem juntas.”

Problemas que o Adapter resolve:

- Incompatibilidade de interfaces: o cliente espera uma interface, mas o objeto fornece outra.
- Reuso de código legado: aproveita classes antigas sem modificá-las.
- Integração entre sistemas diferentes: útil para APIs, bibliotecas e drivers.

ADAPTER

Estrutura

O Adapter envolve quatro componentes principais:

- Target (Alvo): Interface esperada pelo cliente.
- Adaptee (Adaptado): Classe existente com interface incompatível.
- Adapter (Adaptador): Implementa a interface Target e traduz as chamadas para o Adaptee.
- Client (Cliente): Usa objetos que seguem a interface Target.
- O Adapter funciona como um “meio-termo” que converte chamadas entre interfaces diferentes.

Exemplo Java

```
// 1. Target (Alvo)
interface TomadaTresPinos {
    void conectarTresPinos();
}
// 2. Adaptee (Adaptado)
class TomadaDoisPinos {
    void conectarDoisPinos() {
        System.out.println("Conectado à tomada de dois pinos.");
    }
}
// 3. Adapter (Adaptador)
class AdaptadorTomada implements TomadaTresPinos {
    private TomadaDoisPinos tomadaDoisPinos;

    public AdaptadorTomada(TomadaDoisPinos tomadaDoisPinos) {
        this.tomadaDoisPinos = tomadaDoisPinos;
    }
    @Override
    public void conectarTresPinos() {
        System.out.println("Adaptando conexão de três pinos para dois...");
        tomadaDoisPinos.conectarDoisPinos();
    }
}
// 4. Cliente
public class Main {
    public static void main(String[] args) {
        TomadaDoisPinos doisPinos = new TomadaDoisPinos();
        TomadaTresPinos adaptador = new AdaptadorTomada(doisPinos);
        adaptador.conectarTresPinos();
    }
}
```

Vantagens
Reutiliza código legado sem modificação.
Facilita integração entre sistemas incompatíveis.
Segue o Princípio Aberto/Fechado (OCP) — estende o comportamento sem alterar código existente.
Melhora flexibilidade e manutenção do sistema.

Desvantagens
Pode aumentar a complexidade do sistema.
Pode ocultar problemas de design entre classes incompatíveis.
Introduz uma camada extra de tradução, impactando desempenho em grandes sistemas.

ADAPTER

```
// 1. Target (Alvo)
interface TomadaTresPinos {
    void conectarTresPinos();
}

// 2. Adaptee (Adaptado)
class TomadaDoisPinos {
    void conectarDoisPinos() {
        System.out.println("Conectado à tomada de dois pinos.");
    }
}

// 3. Adapter (Adaptador)
class AdaptadorTomada implements TomadaTresPinos {
    private TomadaDoisPinos tomadaDoisPinos;

    public AdaptadorTomada(TomadaDoisPinos tomadaDoisPinos) {
        this.tomadaDoisPinos = tomadaDoisPinos;
    }

    @Override
    public void conectarTresPinos() {
        System.out.println("Adaptando conexão de três pinos para dois...");
        tomadaDoisPinos.conectarDoisPinos();
    }
}

// 4. Cliente
```

COMPOSITE

Definição:

➡️ Compõe objetos em estruturas de árvore (hierarquias "parte-todo").

➡️ Permite tratar objetos individuais (folhas) e grupos (containers) da mesma maneira.

➡️ Simplifica o cliente, que trata todos os objetos de forma uniforme, sem precisar saber a diferença entre eles.

QUANDO USAR

Quando você tem uma hierarquia de objetos do tipo "parte-todo" (ex: um menu que contém submenus e itens de menu).

Quando você quer que o código cliente ignore a diferença entre objetos simples e compostos.

Para simplificar o código do cliente, evitando um monte de if-else para checar se um objeto é uma folha ou um container.

ESTRUTURA

O Composite envolve três componentes principais:

1. **Componente (Component Interface):** A interface comum para todos os objetos (tanto as folhas quanto os compostos). Define as operações que todos devem ter (ex: exibirNome()).
2. **Folha (Leaf):** É o objeto individual, a "ponta" da árvore. Ele implementa o Componente, mas não pode ter filhos.
3. **Composto (Composite):** É o objeto "container" ou "grupo". Ele implementa o Componente e o mais importante: possui uma lista de filhos (outros Componentes). Quando uma operação é chamada nele, ele a "delega" para todos os seus filhos.

EXEMPLO

```
// 1. Componente: A interface comum
public interface ItemDoSistema {
        void exibirNome();
}

// 2. Folha (Leaf): O objeto individual
public class Arquivo implements ItemDoSistema {
        private String nome;

        public Arquivo(String nome) { this.nome = nome; }
        @Override
        public void exibirNome() {
                System.out.println("Arquivo: " + nome);
        }
}
```

```
// 3. Composto (Composite): O container
import java.util.List;
import java.util.ArrayList;

public class Pasta implements ItemDoSistema {
        private String nome;
        // Possui uma lista de Componentes
        private List<ItemDoSistema> filhos = new ArrayList<>();

        public Pasta(String nome) { this.nome = nome; }

        public void adicionar(ItemDoSistema item) {
                filhos.add(item);
        }

        @Override
        public void exibirNome() {
                System.out.println("--- Pasta: " + nome + " ---");
                for (ItemDoSistema item : filhos) {
                        item.exibirNome();
                }
        }
}
```

```
// 4. Cliente
public class Main {
        public static void main(String[] args) {
                Pasta raiz = new Pasta("Disco C:");
                Pasta docs = new Pasta("Documentos");
                docs.adicionar(new Arquivo("relatorio.pdf"));
                raiz.adicionar(docs);
                raiz.adicionar(new Arquivo("foto.jpg"));

                // O cliente chama a operação na raiz
                raiz.exibirNome();
        }
}
```

5. Saída:

```
 --- Pasta: Disco C: ---
 --- Pasta: Documentos ---
Arquivo: relatorio.pdf
Arquivo: foto.jpg
```

DECORATOR

Definição:

➡️ Anexa responsabilidades adicionais a um objeto dinamicamente (em tempo de execução).

➡️ Fornece uma alternativa flexível à herança (subclasses) para estender a funcionalidade de um objeto.

➡️ Envolve o objeto original em um "wrapper" (decorador) que adiciona novos comportamentos.

QUANDO USAR

- Adicionar ou remover responsabilidades de objetos em tempo de execução.
- Evitar uma "explosão de classes" causada pela herança excessiva de funcionalidades.
- Estender funcionalidades sem modificar o código existente (Princípio Aberto/Fechado).

ESTRUTURA

O Decorator envolve três componentes principais:

1. **Componente (Component Interface):** Define a interface para os objetos que terão responsabilidades adicionadas dinamicamente (tanto o objeto base quanto os decoradores a implementam).
2. **Componente Concreto (Concrete Component):** O objeto central que possui a funcionalidade básica e será decorado.
3. **Decorador (Decorator - Classe Abstrata/Interface):** Mantém uma referência ao objeto Componente que ele decora. Define uma interface compatível com o Componente.

EXEMPLO

```
// 1. Componente: A interface comum para todos os cafés
interface Bebida {
    String getDescricao();
    double custo();
}

// 2. Componente Concreto: O objeto base que será decorado
class CafeExpresso implements Bebida {
    @Override
    public String getDescricao() {
        return "Café Expresso";
    }
    @Override
    public double custo() {
        return 4.50;
    }
}

// 3. Decorator Abstrato: Mantém a referência para o objeto Bebida
abstract class AdicionalDecorator implements Bebida {
    protected Bebida bebida; // A bebida que estamos envolvendo

    public AdicionalDecorator(Bebida bebida) {
        this.bebida = bebida;
    }

    // A descrição é uma combinação da descrição do adicional + a da bebida envolvida
    @Override
    public abstract String getDescricao();
}

// 4. Decoradores Concretos: Adicionam a funcionalidade
class LeiteDecorator extends AdicionalDecorator {
    public LeiteDecorator(Bebida bebida) {
        super(bebida);
    }
    @Override
    public String getDescricao() {
        return bebida.getDescricao() + ", com Leite";
    }
    @Override
    public double custo() {
        // Adiciona o custo do leite ao custo da bebida envolvida
        return bebida.custo() + 1.50;
    }
}
class ChocolateDecorator extends AdicionalDecorator {
    public ChocolateDecorator(Bebida bebida) {
        super(bebida);
    }
    @Override
    public String getDescricao() {
        return bebida.getDescricao() + ", com Chocolate";
    }

    @Override
    public double custo() {
        // Adiciona o custo do chocolate
        return bebida.custo() + 2.00;
    }
}

// Cliente
public class Main {
    public static void main(String[] args) {
        // Pedido 1: Um café expresso simples
        Bebida cafeSimples = new CafeExpresso();
        System.out.println("Pedido 1:");
        System.out.println("Descrição: " + cafeSimples.getDescricao());
        System.out.println("Custo: R$" + cafeSimples.custo());

        System.out.println("\n------------------------\n");

        // Pedido 2: Um café expresso com leite e chocolate
        // Começamos com o objeto base
        Bebida cafeDecorado = new CafeExpresso();
        // Envolvemos com o decorador de Leite
        cafeDecorado = new LeiteDecorator(cafeDecorado);
        // Envolvemos o objeto já decorado com o decorador de Chocolate
        cafeDecorado = new ChocolateDecorator(cafeDecorado);

        System.out.println("Pedido 2:");

        System.out.println("Descrição: " + cafeDecorado.getDescricao());
        System.out.println("Custo: R$" + cafeDecorado.custo()); // 4.50 (café) + 1.50 (leite) + 2.00 (chocolate)
    }
}
```

Saída:

```
- Pedido 1:
Descrição: Café Expresso
Custo: R$4.5
- Pedido 2:
Descrição: Café Expresso, com Leite, com Chocolate
Custo: R$8.0
```

FACADE

Definição:

➡️ Fornece uma interface única e simplificada para um conjunto complexo de classes.

➡️ Oculta a complexidade interna e facilita o uso do sistema.

QUANDO USAR

Sistema com muitas classes complexas.

Desejo de oferecer uma “porta de entrada” simples.

Reduz acoplamento entre cliente e subsistemas.

ESTRUTURA

O Facade envolve três componentes principais:

1. **Facade (Fachada):** A classe que provê a interface simplificada. Ela conhece todas as classes do subsistema e orquestra as chamadas necessárias.
2. **Subsistema (Subsystem Classes):** As classes que implementam a funcionalidade real. Elas não têm conhecimento da Fachada e podem continuar sendo usadas diretamente, se necessário.
3. **Cliente (Client):** O código que usa o sistema. Ele interage **apenas** com a Fachada, chamando um método simples de alto nível.

EXEMPLO

```
// Subsistemas complexos
class DVDPlayer {
    void on() { System.out.println("DVD ligado."); }
    void play(String movie) {
System.out.println("Reproduzindo: " + movie); }
}
class Projector {
    void on() { System.out.println("Projetor ligado."); }
    void setInput(DVDPlayer dvd) {
System.out.println("Entrada configurada para DVD.");
}
}
class SoundSystem {
    void on() { System.out.println("Som ligado."); }
    void setVolume(int level) {
System.out.println("Volume: " + level);
```

```
// Facade
class HomeTheaterFacade {
private DVDPlayer dvd;
private Projector projector;
private SoundSystem sound;
public HomeTheaterFacade(DVDPlayer dvd, Projector projector, SoundSystem
sound)
{
this.dvd = dvd;
this.projector = projector;
this.sound = sound;
}
public void assistirFilme(String filme) {
System.out.println("Preparando para assistir ao filme...");
dvd.on();
projector.on();
projector.setInput(dvd);
sound.on();
sound.setVolume(10);
dvd.play(filme);
}
}
```

EXEMPLO

```
//Cliente
public class Main {
    public static void main(String[] args) {
        DVDPlayer dvd = new DVDPlayer();
        Projector proj = new Projector();
        SoundSystem sound = new SoundSystem();
        HomeTheaterFacade homeTheater = new
HomeTheaterFacade(dvd, proj, sound);
        homeTheater.assistirFilme("Matrix");
    }
}
```

Saída:

Preparando para assistir ao filme...
DVD ligado.
Projetor ligado.
Entrada configurada para DVD.
Som ligado.
Volume: 10
Reproduzindo: Matrix

O cliente só usou uma classe (Facade), sem lidar com toda a complexidade.

Flyweight

Definição:

➡️ Reduzir o consumo de memória entre objetos compartilhados.

➡️ Separado em duas partes uma intrínseca e outra extrínseca.

QUANDO USAR

Muitas instâncias de um objeto que consome muita memória

Sistema com recursos limitados.

Melhorar desempenho e reduzir consumo de memória

ESTRUTURA

O Flyweight envolve quatro componentes principais:

1. **Flyweight (interface abstrata)** – Define operações que podem depender de dados extrínsecos.
2. **ConcreteFlyweight** – Implementa o Flyweight e mantém o estado intrínseco compartilhado.
3. **FlyweightFactory** – Gerencia e fornece instâncias compartilhadas de Flyweight.
4. **Client** – Mantém os estados extrínsecos e interage com os Flyweights.

EXEMPLO

```
class ArvoreTipo {
    private String nome;
    private String corTextura;
    private String texturaArquivo;

    public ArvoreTipo(String nome, String corTextura, String
texturaArquivo) {
        this.nome = nome;
        this.corTextura = corTextura;
        this.texturaArquivo = texturaArquivo;
    }

    public void desenhar(int x, int y) { // dados extrínsecos
vêm de fora
        System.out.println("Desenhando " + nome + " na
posição (" + x + ", " + y + ")");
    }
}

    public static ArvoreTipo getArvoreTipo(String nome,
String cor, String textura) {
        String chave = nome + cor + textura;
        if (!tipos.containsKey(chave)) {
            tipos.put(chave, new ArvoreTipo(nome, cor, textura));
            System.out.println("[Novo Flyweight criado]");
        }
        return tipos.get(chave);
    }
}
```

```
class Arvore {
private int x, y; // dados extrínsecos
private ArvoreTipo tipo; // Flyweight
public Arvore(int x, int y, ArvoreTipo tipo) {
this.x = x;
this.y = y;
this.tipo = tipo; }
public void desenhar() {
tipo.desenhar(x, y); } }
// Uso
public class Main {
public static void main(String[] args) {
Arvore arv1 = new Arvore(10, 20, ArvoreFactory.getArvoreTipo("Ipê", "Amarelo",
"ipe.png"));
Arvore arv2 = new Arvore(50, 60, ArvoreFactory.getArvoreTipo("Ipê", "Amarelo",
"ipe.png"));
Arvore arv3 = new Arvore(70, 80, ArvoreFactory.getArvoreTipo("Ipê", "Amarelo",
"ipe.png"));
arv1.desenhar();
arv2.desenhar();
arv3.desenhar();
}
}
```

PROXY

Definição:

➡️ Cria um objeto substituto (intermediário) que controla o acesso ao objeto real.

➡️ O proxy tem a mesma interface do objeto original.

QUANDO USAR

Controle de acesso ou segurança

Carregamento preguiçoso (lazy loading)

Log, cache ou monitoramento de chamadas

ESTRUTURA

O Proxy envolve três componentes principais:

1. **Assunto/Interface (Subject):** Interface comum que o Objeto Real e o Proxy devem implementar. Garante que o Proxy seja um substituto transparente.
2. **Assunto Real/Objeto Real (Real Subject):** A classe que contém a lógica de negócios principal (o objeto "caro" ou que precisa de proteção).
3. **Proxy (Procurador):** Mantém uma referência ao Objeto Real. Implementa a mesma interface do Assunto e, em seus métodos, executa a lógica de controle antes de delegar a chamada para o Objeto Real.

EXEMPLO

Imagine um serviço caro de carregar imagens grandes da internet:

```
// Interface comum

interface Imagem {
    void exibir();
}
// Objeto real
class ImagemReal implements Imagem {
    private String arquivo;
    public ImagemReal(String arquivo) {
        this.arquivo = arquivo;
        carregarDoDisco();
    }
    private void carregarDoDisco() {
        System.out.println("Carregando " + arquivo + " do disco...");
    }
    public void exibir() {
        System.out.println("Exibindo " + arquivo);
    }
}
```

```
// Proxy
class ImagemProxy implements Imagem {
private ImagemReal imagemReal;
private String arquivo;
public ImagemProxy(String arquivo) {
this.arquivo = arquivo;
}
public void exibir() {
if (imagemReal == null) {
imagemReal = new ImagemReal(arquivo); // só carrega quando necessário
}
imagemReal.exibir();
}
}
```

EXEMPLO

```
// Cliente
public class Main {
    public static void main(String[] args) {
        Imagem img = new ImagemProxy("foto.png");

        // A imagem só será carregada quando realmente for exibida
        System.out.println("Imagem criada, mas ainda não carregada.");
        img.exibir(); // agora carrega e exibe
        img.exibir(); // agora já está em cache
    }
}
```

Saída:

Imagem criada, mas ainda não carregada.
Carregando foto.png do disco...
Exibindo foto.png
Exibindo foto.png

O Proxy controla quando o objeto real é criado e acessado, evitando custos desnecessários.

BRIDGE

Definição:

➡️ Desacopla uma abstração da sua implementação para que as duas possam variar independentemente.

➡️ Em vez de herança, ele usa composição. Ele "conecta" duas hierarquias de classes diferentes com uma "ponte".

➡️ Pense nele como ter uma hierarquia para os "controles" e outra para os "dispositivos".

QUANDO USAR

Quando você quer evitar a "explosão de classes"

Quando você precisa que tanto a abstração quanto a implementação possam ser alteradas de forma independente.

Quando você quer poder trocar a implementação em tempo de execução.

ESTRUTURA

O Composite envolve três componentes principais:

1. **Abstração (Abstraction):** A interface de "alto nível" que o cliente usa. Ela possui uma referência para o Implementor.
2. **Abstração Refinada (Refined Abstraction):** Extensões da Abstração que adicionam mais lógica.
3. **Implementador (Implementor Interface):** A interface de "baixo nível" que define as operações primitivas.
4. **Implementador Concreto (Concrete Implementor):** As classes reais que fazem o trabalho pesado. Elas implementam o Implementador.

EXEMPLO

```
// 1. Implementador (A interface "Plataforma")
public interface Dispositivo {
	void ligar();
	void desligar(); void setCanal(int canal);
}

// 2. Implementador Concreto (Ex: TV)
	public class TV implements Dispositivo {
	@Override
	public void ligar() { System.out.println("TV: ligada"); }
	@Override
	public void desligar() { System.out.println("TV: desligada"); }
	@Override
	public void setCanal(int canal) {
		System.out.println("TV: canal " + canal);
    }
}
```

```
// 3. Abstração (O "Controle")
public abstract class ControleRemoto {
	// A "PONTE" é esta referência!
	protected Dispositivo dispositivo;
	public ControleRemoto(Dispositivo d) {
		this.dispositivo = d;
	}
	public void ligar() { dispositivo.ligar(); }
	public void desligar() { dispositivo.desligar(); }
	// Deixa uma ação para as subclasses definirem
	public abstract void acaoDoBotaoMeio();
}

// 4. Abstração Refinada
public class ControlePadrao extends ControleRemoto {

    public ControlePadrao(Dispositivo d) { super(d); }

    @Override
    public void acaoDoBotaoMeio() {
        System.out.println("Controle: mudando canal para 10");
        dispositivo.setCanal(10);
    }
}
```

```
// Cliente
public class Main {
    public static void main(String[] args) {
        // Conecta a abstração (Controle) com a
        // implementação (TV) na criação
        ControleRemoto controleTV =
            new ControlePadrao(new TV());

        controleTV.ligar();
        controleTV.acaoDoBotaoMeio(); // TV vai pro canal 10

        System.out.println("--- Trocando para o Rádio ---");

        // Agora o mesmo tipo de controle, mas com outra
implementação
        ControleRemoto controleRadio =
            new ControlePadrao(new Radio());

        controleRadio.ligar();
        controleRadio.acaoDoBotaoMeio(); // Rádio vai pra
estação 10
    }
}
```

Saída:

```
TV: ligada
Controle: mudando canal para 10
TV: canal 10
--- Trocando para o Rádio ---
Rádio: ligado
Controle: mudando canal para 10
Rádio: sintonizado na estação 10
```