PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Artur Monteiro Parreiras

Brenda Stefany de Oliveira Rocha

Bruna Letícia Silva

Eduarda Faria Pinheiro

Elenice Florentina de Oliveira dos Reis

Marco Aurélio Faria Ramos

Padrões de Projeto: Estrutura

Belo Horizonte
2025

1. Padrão de Adapter

1. 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 (incompatível) 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."

2. O Problema que o Adapter Resolve

Imagine que você está desenvolvendo um sistema que utiliza uma interface específica, mas precisa integrar uma biblioteca ou classe existente cuja interface é diferente.
Modificar o código original pode não ser possível (por exemplo, é código legado, de terceiros ou muito arriscado).

O Adapter resolve esse problema encapsulando o objeto existente em uma nova classe adaptadora, que expõe a interface esperada e traduz as chamadas para o formato que o objeto original entende.

Problemas resolvidos:

- Incompatibilidade de interfaces: o cliente espera uma interface, mas o objeto fornece outra.

- Reuso de código legado: você pode aproveitar classes antigas sem modificá-las.

- Integração entre sistemas diferentes: muito útil em integração de APIs, bibliotecas e drivers.

3. 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.

4. Exemplo em Java

Imagine que você tem uma tomada de três pinos, mas o seu aparelho tem apenas dois pinos. O Adapter permite conectar o aparelho de dois pinos na tomada de três pinos sem alterar o código do aparelho.

```
// 1. Target (Alvo): Interface esperada pelo cliente
interface TomadaTresPinos {
    void conectarTresPinos();
}
// 2. Adaptee (Adaptado): Classe existente com interface incompatível
class TomadaDoisPinos {
    void conectarDoisPinos() {
        System.out.println("Conectado à tomada de dois pinos.");
    }
}
// 3. Adapter (Adaptador): Traduz a interface do Adaptee para a interface Target
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: Usa a interface Target

public class Main {

    public static void main(String[] args) {

        TomadaDoisPinos doisPinos = new TomadaDoisPinos();

        TomadaTresPinos adaptador = new AdaptadorTomada(doisPinos);



        adaptador.conectarTresPinos();

    }

}
```

Saída:

Adaptando conexão de três pinos para dois...

Conectado à tomada de dois pinos.

5. Vantagens

Permite reutilizar código legado sem modificá-lo.
Facilita a integração entre sistemas incompatíveis.
Segue o Princípio Aberto/Fechado (OCP) — você estende o comportamento sem alterar

código existente.
Melhora a flexibilidade e manutenção do sistema.
6. Desvantagens

Pode aumentar a complexidade do sistema se usado em excesso.
Pode ocultar problemas de design entre classes incompatíveis.
Introduz uma camada adicional de indireção (tradução), o que pode afetar desempenho em sistemas muito grandes.

2. Padrão de Bridge

Definição O Bridge (Ponte) é um padrão estrutural que desacopla uma abstração da sua implementação, permitindo que as duas possam variar independentemente.

Objetivo Separar uma hierarquia de classes grande ou complexa em duas hierarquias independentes: a "abstração" (lógica de controle) e a "implementação" (plataforma de execução). Isso evita a "explosão de classes" quando você precisa estender as duas dimensões.

Estrutura UML Simplificada Client --> Abstraction --[ref]--> Implementor Abstraction <-- RefinedAbstraction Implementor <-- ConcreteImplementor

- **Client (Cliente):** código que usa a Abstraction.
- **Abstraction (Abstração):** interface de alto nível que o cliente usa. Contém uma referência (a "ponte") para um Implementor.
- **Implementor (Implementador):** interface de baixo nível (a "plataforma") que a Abstraction usa.
- **RefinedAbstraction (Abstração Refinada):** Extensões da Abstraction (ex: ControleRemotoAvancado).
- **ConcreteImplementor (Implementador Concreto):** Classes que implementam o Implementor (ex: TV, Radio).

Exemplo prático em Java Suponha que temos diferentes tipos de Controles Remotos (Padrão, Avançado) e diferentes Dispositivos (TV, Rádio). Em vez de criar ControlePadraoParaTV, ControlePadraoParaRadio, etc., o Bridge separa as duas lógicas.

Java

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

Java

```
// 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);
    }
}
```

Java

```
// 2. Implementador Concreto (Ex: Radio)
public class Radio implements Dispositivo {
    @Override
    public void ligar() { System.out.println("Rádio: ligado"); }
    @Override
    public void desligar() { System.out.println("Rádio: desligado"); }
    @Override
    public void setCanal(int canal) {
        System.out.println("Rádio: sintonizado na estação " + canal);
    }
}
```

Java

```
// 3. Abstração (O "Controle" genérico)
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();
}
```

Java

```
// 4. Abstração Refinada (Um tipo de controle específico)
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);
    }
}
```

Java

```
// Cliente
public class Main {
    public static void main(String[] args) {
        // Conecta a abstração (Controle) com a implementação (TV)
        ControleRemoto controleTV = new ControlePadrao(new TV());
        controleTV.ligar();
        controleTV.acaoDoBotaoMeio(); // TV vai pro canal 10

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

```
        // O mesmo 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
```

✅ Vantagens

- Evita a "explosão de classes" e o acoplamento permanente.
- Abstração e Implementação podem evoluir de forma independente.
- Permite trocar a implementação em tempo de execução.
- Segue o Princípio Aberto/Fechado (OCP).

⚠️ Desvantagens

- Aumenta a complexidade do código ao introduzir duas novas hierarquias (Abstração e Implementação).
- Pode ser complexo de desenhar no início.

🌍 Exemplo do mundo real

- Controles remotos para diferentes dispositivos (TVs, Rádios).
- Drivers de sistema operacional (uma API gráfica abstrata que roda em diferentes implementações de SO: Windows, Linux, Mac).
- Aplicações de formas (Círculo, Quadrado) que podem ser desenhadas em diferentes APIs gráficas (API_V1, API_V2).

3. Padrão de Composite

Definição O Composite é um padrão estrutural que permite compor objetos em estruturas de árvore para representar hierarquias "parte-todo". Ele permite que os clientes tratem objetos individuais (folhas) e composições de objetos (grupos) de maneira uniforme.

Objetivo Converter a operação do cliente para funcionar de forma transparente em uma estrutura de árvore. O cliente não precisa saber se está lidando com um objeto individual (Leaf) ou um grupo (Composite). Ele simplesmente chama a operação no "componente" e a própria estrutura resolve a recursividade.

Estrutura UML Simplificada Client --> Component <-- Leaf Client --> Component <-- Composite (que contém uma lista de Components)

- **Client (Cliente):** código que usa a interface Component.
- **Component (Componente):** interface esperada pelo cliente, implementada por todos os objetos na árvore.
- **Leaf (Folha):** classe do objeto individual, que implementa Component mas não pode ter filhos.
- **Composite (Composto):** classe do objeto "container", que implementa Component e armazena uma lista de filhos (Component).

Exemplo prático em Java Suponha que temos um sistema de arquivos. Queremos exibir a estrutura de nomes, mas temos Arquivos (individuais) e Pastas (grupos). O Composite resolve isso fazendo com que ambos implementem a mesma interface.

Java

```
// 1. Componente: A interface comum

public interface ItemDoSistema {

   void exibirNome();

}
```

Java

```
// 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);
    }
}
```

Java

```
// 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 + " ---");
        // A mágica: delega para os filhos
        for (ItemDoSistema item : filhos) {
            item.exibirNome();
        }
    }
}
```

Java

```
// 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();
    }
```

}

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

✅ Vantagens

- Simplifica o código do cliente (trata todos os objetos de forma uniforme).
- Facilita a adição de novos tipos de componentes (novas folhas ou novos compostos).
- Segue o Princípio Aberto/Fechado (OCP).

⚠️ Desvantagens

- Pode criar uma interface muito genérica (ex: a Folha pode ter métodos de adicionar que não fazem sentido).
- Pode dificultar a restrição de tipos de filhos em um Composto.

🌍 Exemplo do mundo real

- Sistemas de arquivos (Pastas e Arquivos).
- Menus de interface gráfica (um Menu contém ItensDeMenu e outros SubMenus).
- Organogramas de empresas (um Departamento contém Funcionarios e outros SubDepartamentos).

4. Padrão Decorator

1. Conceito

O Decorator (decorador) permite adicionar novas funcionalidades a um objeto dinamicamente, sem alterar sua classe. Ele "envolve" o objeto original em um ou mais objetos "decoradores", onde cada um adiciona uma responsabilidade específica. É uma alternativa flexível à herança para estender funcionalidades.

Definição (GoF): "Anexar responsabilidades adicionais a um objeto dinamicamente. Os Decorators fornecem uma alternativa flexível à herança para estender a funcionalidade."

2. O Problema que o Decorator Resolve

Imagine que você tem uma classe e precisa adicionar várias funcionalidades a ela. A abordagem tradicional seria usar herança (subclasses). No entanto, isso leva a problemas:

- Explosão de Classes: Se você tiver várias funcionalidades independentes (ex: A, B, C), precisaria criar subclasses para cada combinação (ClasseComA, ClasseComB, ClasseComAB, ClasseComABC, etc.), o que se torna inviável rapidamente.
- Rigidez: A herança é estática. A funcionalidade é adicionada em tempo de compilação, e você não pode adicionar ou remover responsabilidades de um objeto em tempo de execução.
- Violação do Princípio de Responsabilidade Única: Uma única classe pode acabar com muitas responsabilidades que poderiam ser separadas.

O Decorator resolve isso permitindo que você "empilhe" funcionalidades em um objeto base de forma modular e em tempo de execução.

3. Estrutura

O Decorator envolve quatro componentes principais:

- Component (Componente): A interface ou classe abstrata que define os métodos que serão implementados tanto pelo objeto base quanto pelos decoradores. O cliente usará essa interface para interagir com os objetos.
- ConcreteComponent (Componente Concreto): A classe do objeto original ao qual queremos adicionar funcionalidades. Ela implementa a interface Component.
- Decorator (Decorador): Uma classe abstrata que também implementa a interface Component. Ela contém uma referência (agregação) para um objeto Component e delega as chamadas a ele. É a base para todos os decoradores concretos.
- ConcreteDecorator (Decorador Concreto): As classes que adicionam as novas responsabilidades. Elas herdam do Decorator e adicionam seu comportamento antes ou depois de delegar a chamada para o objeto que estão envolvendo.

4. Exemplo em Java

Imagine um sistema para uma cafeteria. Temos um café simples, mas queremos adicionar ingredientes extras como Leite, Chocolate, etc., onde cada um adiciona um custo e altera a descrição.

```
// 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

5. Padrão Facade

1. Conceito

O Facade (fachada) fornece uma interface simples para um conjunto complexo de classes ou subsistemas. Em vez de o código cliente interagir diretamente com várias classes internas, ele fala apenas com uma “fachada” que simplifica o acesso.
Definição (GoF): "Fornece uma interface unificada para um conjunto de interfaces em um subsistema. A Fachada define uma interface de nível mais alto que torna o subsistema mais fácil de usar."

2. O Problema que o Facade Resolve

Sistemas modernos são frequentemente compostos por múltiplos subsistemas, cada um com dezenas de classes. Quando um código "cliente" precisa realizar uma tarefa de alto nível, ele se depara com a necessidade de:

1. **Conhecer muitas Classes:** O cliente precisa saber quais classes do subsistema existem.
2. **Gerenciar a Orquestração:** O cliente precisa chamar essas classes em uma sequência correta e específica.
3. **Alto Acoplamento:** Se o subsistema muda (uma classe é renomeada ou adicionada), o código cliente deve ser alterado em vários lugares.

O Facade resolve isso fornecendo um único ponto de entrada que encapsula toda essa complexidade.

3. 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.

4. Exemplo em Java

Imagine um sistema de **home theater** com várias classes:

```
// 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);
    }
}
```

//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.

6. Padrão Flyweight

1. Conceito

O Flyweight é um padrão estrutural que tem como objetivo economizar memória ao compartilhar objetos imutáveis que são muito repetidos dentro de um sistema.

Definição (GoF):

“Usa o compartilhamento para suportar eficientemente grandes quantidades de objetos de forma granular.”

Ou seja: em vez de instanciar vários objetos idênticos, criamos um só e o reutilizamos.

2. O Problema que o Flyweight Resolve

Programas que precisam lidar com milhares ou milhões de objetos iguais ou muito parecidos, como: letras em um editor de texto, tiles em jogos 2D e partículas de efeitos gráficos

Sem Flyweight: criam-se milhares de instâncias duplicadas, alto consumo de memória e baixo desempenho

Com Flyweight: objetos compartilham dados “intrínsecos” (imutáveis/repetidos), dados “extrínsecos” (variáveis/externos) são enviados como parâmetro

3. Estrutura

O padrão Flyweight é composto por quatro elementos principais que trabalham em conjunto para permitir o compartilhamento eficiente de objetos. O Flyweight Interface define os métodos que os objetos compartilháveis devem implementar, garantindo uma forma consistente de interação. O Concrete Flyweight representa a implementação concreta desses objetos compartilhados, armazenando os dados intrínsecos — ou seja, as informações imutáveis e comuns a várias instâncias. Para gerenciar o acesso e garantir que objetos repetidos não sejam criados várias vezes, utiliza-se a Flyweight Factory, responsável por fornecer instâncias existentes ou criar novas quando necessário. Por fim, o Client é o código que utiliza os objetos Flyweight, fornecendo os dados extrínsecos, que são variáveis e específicos do contexto, como posição, tamanho ou cor, de modo a personalizar cada uso sem duplicar os objetos compartilhados. Dessa forma, o Flyweight consegue reduzir o consumo de memória e melhorar a performance, especialmente em sistemas que lidam com grandes quantidades de objetos similares.

4. Exemplo em Java

Exemplo simples: sistema que renderiza milhares de árvores em um mapa — mas não duplica texturas ou dados iguais.

```
// Flyweight (dados intrínsecos, compartilhados)
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 + ")");
    }
}

// Flyweight Factory
class ArvoreFactory {
    private static Map<String, ArvoreTipo> tipos = new HashMap<>();

    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);
    }
}
```

```
// Client
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();
    }
}
```

Saída

```
Desenhando Ipê na posição (10, 20)
```

Desenhando Ipê na posição (50, 60)
Desenhando Ipê na posição (70, 80)

7. Padrão Proxy

1.**Conceito**

O Proxy (representante, intermediário) é usado quando se quer controlar o acesso a um objeto real. Ele se comporta como o objeto real, mas adiciona controle, cache, segurança ou otimização.

Definição (GoF): "Fornecer um substituto ou espaço reservado para outro objeto para controlar o acesso a ele."

2. O Problema que Resolve

Muitas vezes, é necessário adicionar funcionalidades de controle, otimização ou segurança a um objeto existente sem alterar o seu código. O Proxy permite:

1. **Adicionar Ações:** Executar algo antes ou depois da chamada ao objeto real.
2. **Controle o Acesso:** Verifica permissões ou execute *lazy loading* (carregamento lento).

3. Estrutura (Componentes)

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.

4. **Exemplo em Java**

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();
    }
}
// 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.