Padrões Estruturais de Design

Introdução

Os padrões estruturais focam em **como classes e objetos são compostos** para formar estruturas maiores, promovendo flexibilidade e eficiência. Abaixo está uma explicação detalhada dos principais padrões, com exemplos e aplicações:

1 Padrões Estruturais

1.1 Adapter (Adaptador)

- **Intenção**: Permitir que objetos com interfaces incompatíveis colaborem.
- **Estrutura**:
  - **Class Adapter**: Usa herança para adaptar uma classe existente a uma interface desejada.
  - **Object Adapter**: Usa delegação (composição) para envolver o objeto adaptado.
- **Exemplo**:

```
// Adaptando LegacyRectangle para a interface Shape
class OldRectangle implements Shape {
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(x1, y1, x2 - x1, y2 - y1);
    }
}
```

- **Quando usar**: Integrar bibliotecas legadas ou sistemas terceiros com interfaces incompatíveis.
- **Vantagens**: Baixo acoplamento; reutilização de código existente.
- **Relação com GRASP**: *Low Coupling* e *Pure Fabrication*.

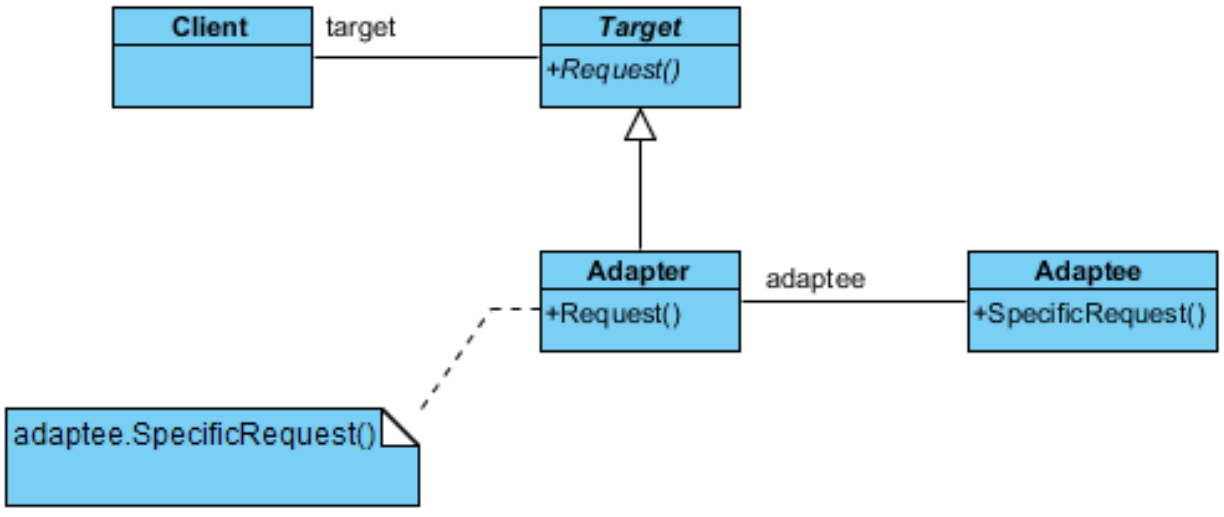

Figura 1: Diagrama UML do padrão Adapter

1.2 Bridge (Ponte)

- **Intenção**: Separar abstração de implementação, permitindo que ambas variem independentemente.
- **Estrutura**:
  - **Abstração** (ex.: `Machine`) delega operações para uma **Implementação** (ex.: `Press`, `Cutter`).
- **Exemplo**:

```
abstract class Machine {
    protected MachineImpl impl;
    public Machine(MachineImpl impl) { this.impl = impl; }
    abstract void start();
}
class Press extends Machine {
    public void start() { impl.startPress(); }
}
```

- **Quando usar**: Evitar explosão de classes ao combinar múltiplas dimensões (ex.: máquinas programáveis vs. não programáveis).
- **Vantagens**: Flexibilidade para adicionar novas implementações sem afetar clientes.

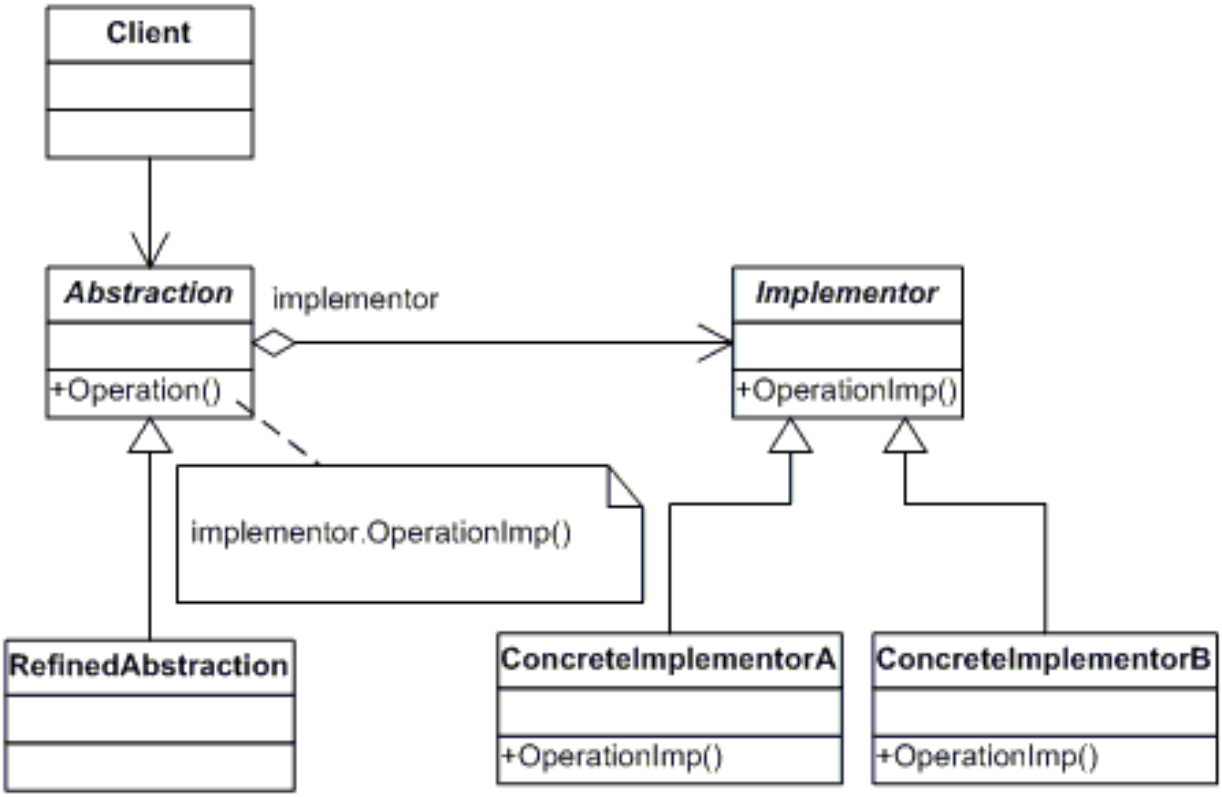

Figura 2: Diagrama UML do padrão Bridge

1.3 Composite (Composite)

- **Intenção**: Tratar objetos individuais e composições de objetos de forma uniforme.
- **Estrutura**:
  - **Component**: Interface comum para folhas (`File`) e composites (`Directory`).
  - **Composite**: Armazena componentes filhos e implementa operações recursivas.
- **Exemplo**:

```
interface Component { void traverse(); }
class Directory implements Component {
    private List<Component> children = new ArrayList<>();
```

```
    public void add(Component c) { children.add(c); }
    public void traverse() { children.forEach(Component::traverse); }
}
```

- **Quando usar**: Representar hierarquias parte-todo (ex.: interfaces gráficas, sistemas de arquivos).
- **Vantagens**: Código cliente simplificado; adição fácil de novos tipos.

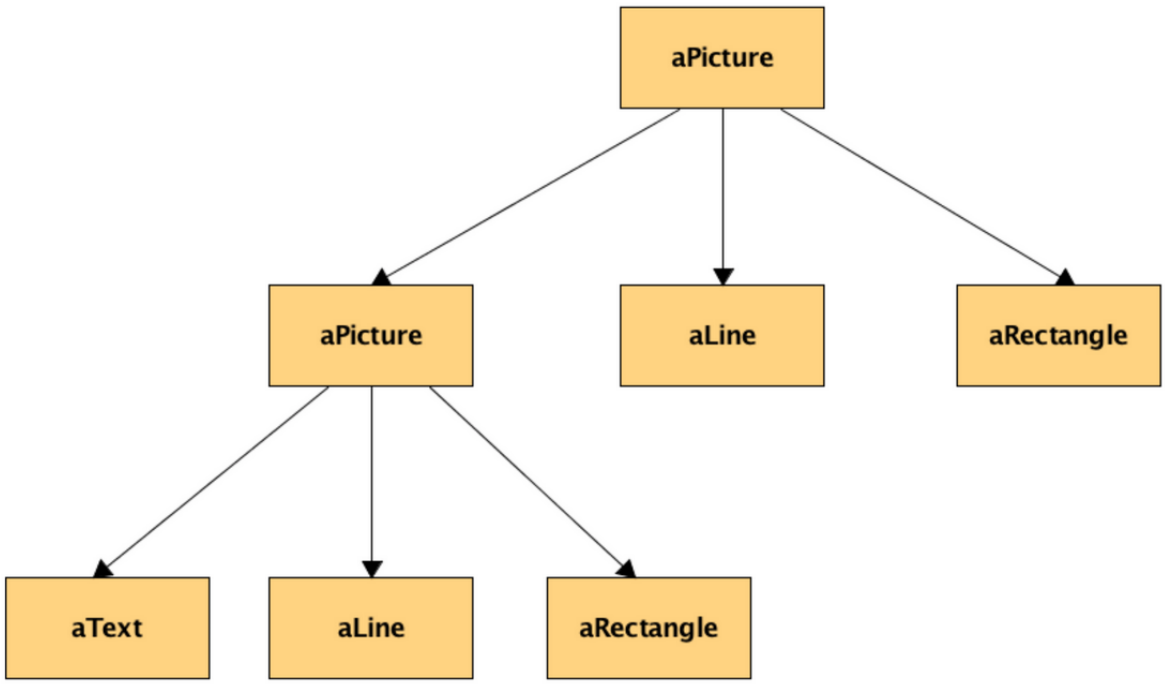

Figura 3: Diagrama UML do padrão Composite

1.4 Decorator (Decorador)

- **Intenção**: Adicionar responsabilidades a objetos dinamicamente.
- **Estrutura**:
  - **Component**: Define a interface comum.
  - **Decorator**: Mantém uma referência a um `Component` e adiciona funcionalidades.
- **Exemplo**:

```
interface Jogador { void joga(); }
class Tenista extends JogadorDecorator {
    public Tenista(Jogador j) { super(j); }
    @Override public void joga() { super.joga(); System.out.print("t nis"); }
}
```

- **Quando usar**: Estender comportamentos em tempo de execução sem herança múltipla.
- **Vantagens**: Flexibilidade; evita classes "inchadas".
- **Aplicações**: Java I/O (`BufferedReader`), GUIs com componentes scrolláveis.

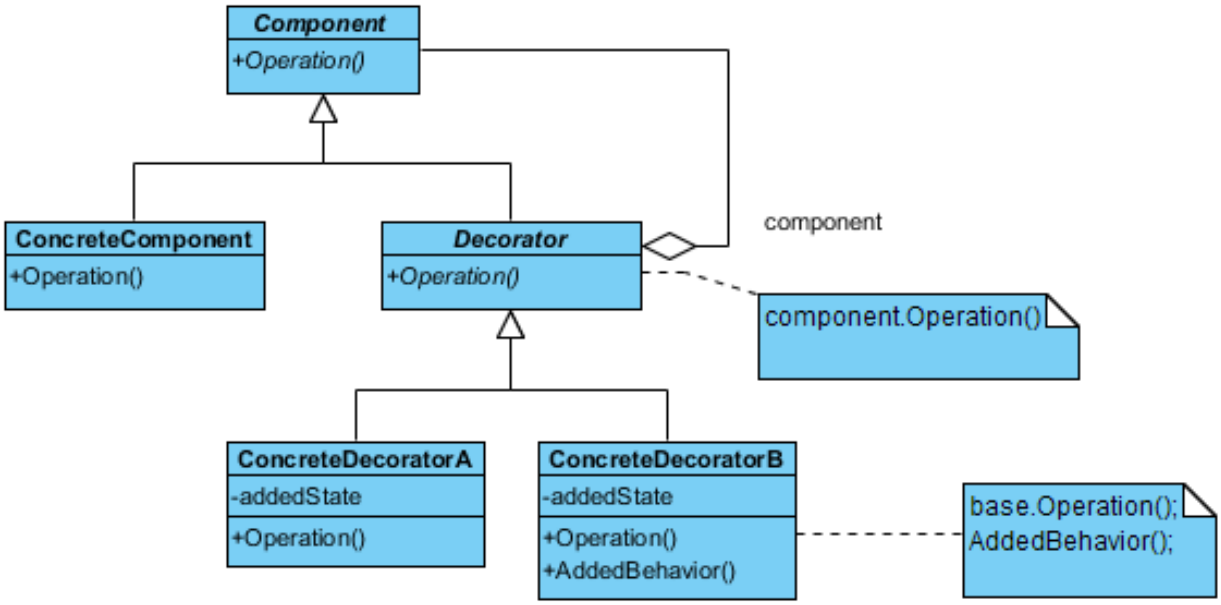

Figura 4: Diagrama UML do padrão Decorator

1.5 Façade (Fachada)

- **Intenção**: Fornecer uma interface unificada para um subsistema complexo.
- **Estrutura**:
  - **Façade** (ex.: `TravelFacade`) encapsula chamadas a múltiplos módulos (hotel, voo, tours).
- **Exemplo**:

```
class TravelFacade {
    public void planTrip(City city) {
        FlightBooker.reserveFlight(city);
        HotelBooker.bookHotel(city);
    }
}
```

- **Quando usar**: Simplificar interações com sistemas complexos.
- **Vantagens**: Isolamento de mudanças; interface amigável para clientes.

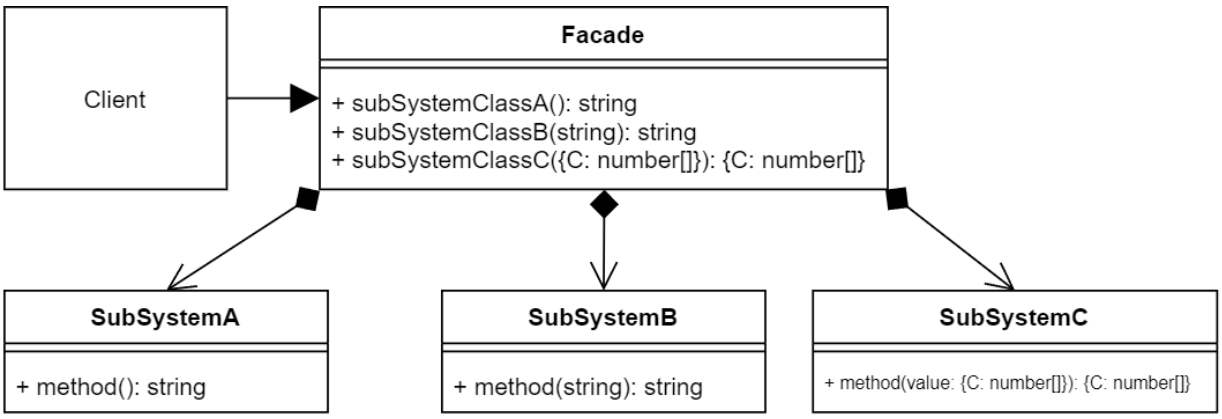

Figura 5: Diagrama UML do padrão Facade

1.6 Flyweight (Peso Mosca)

- **Intenção**: Compartilhar objetos para reduzir custos de memória.
- **Estrutura**:

  - **FlyweightFactory**: Gerencia objetos compartilhados (ex.: `IntegerCache` em Java).
  - **Flyweight**: Armazena estado intrínseco (compartilhado) e delega estado extrínseco.
- **Exemplo**:

```
// Cache de inteiros -128 a 127
Integer i1 = Integer.valueOf(42); // Reutiliza inst ncia existente
```

- **Quando usar**: Muitos objetos similares com estado duplicado.
- **Aplicações**: Caches de caracteres em editores de texto, imagens em navegadores.

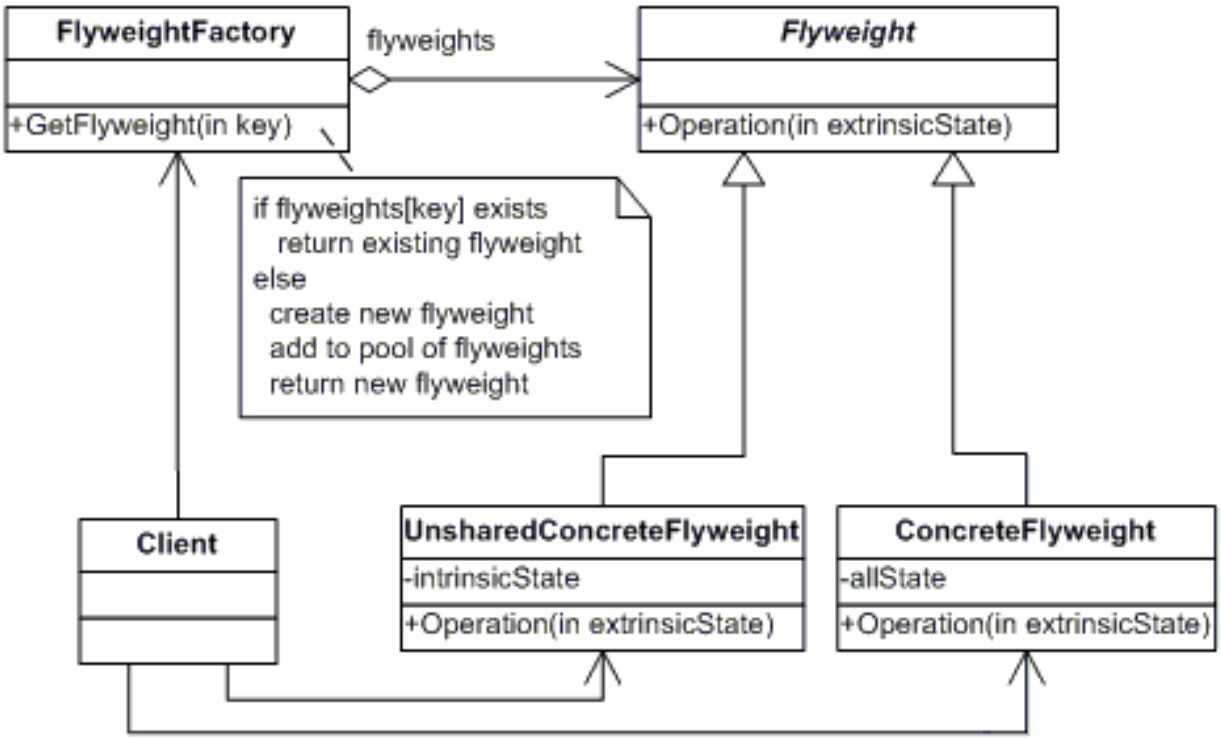

Figura 6: Diagrama UML do padrão Flyweight

1.7 Proxy (Proxy)

- **Intenção**: Controlar acesso a um objeto, adiando sua criação ou adicionando funcionalidades.
- **Estrutura**:
  - **Proxy** (ex.: `LazyProxy`) atua como substituto do objeto real (`RealSubject`).
- **Tipos**:
  - **Virtual Proxy**: Adia criação de objetos caros (ex.: lazy loading).
  - **Protection Proxy**: Controla acesso com permissões.
  - **Remote Proxy**: Encapsula chamadas remotas (ex.: RMI).
- **Exemplo**:

```
class ImageProxy implements Image {
    private RealImage realImage;
    public void display() {
        if (realImage == null) realImage = new RealImage("file.jpg");
        realImage.display();
    }
}
```

- **Vantagens**: Otimização de recursos; segurança; abstração de complexidade.

2 Relação com Outros Princípios

- **GRASP**:
  - *Low Coupling* (Adapter, Bridge).
  - *Pure Fabrication* (Proxy, Flyweight).
- **SOLID**:
  - *Open/Closed Principle* (Decorator permite extensão sem modificação).
  - *Interface Segregation* (Façade simplifica interfaces complexas).

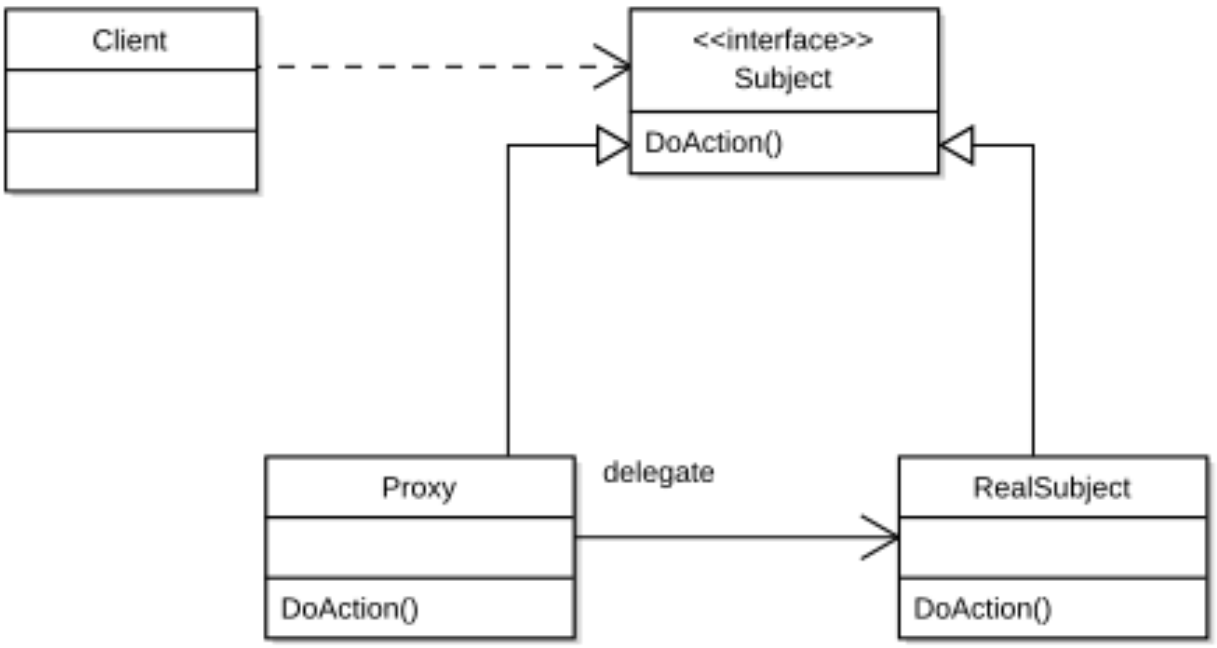

Figura 7: Diagrama UML do padrão Proxy

3 Conclusão

Os padrões estruturais resolvem problemas de **composição de objetos** e **integração de subsistemas**, promovendo código **flexível**, **eficiente** e **mantível**. Escolha o padrão conforme o contexto:

- Use **Adapter** para integrar interfaces incompatíveis.
- Aplique **Bridge** para separar abstrações de implementações.
- Opte por **Composite** para hierarquias parte-todo.
- Recorra a **Decorator** para adicionar funcionalidades dinâmicas.
- Simplifique com **Façade** para subsistemas complexos.
- Otimize com **Flyweight** para objetos compartilhados.
- Controle acesso com **Proxy** para recursos críticos.