Padrão Decorator - Exemplo

1. Interface Base (Component)

```java
interface Notificacao {
    String enviar(String mensagem);
}
```

Propósito: Define o contrato comum que tanto o objeto base quanto os decoradores devem seguir. Garante que todos os componentes tenham o mesmo comportamento básico.

2. Implementação Concreta (ConcreteComponent)

```java
class NotificacaoSimples implements Notificacao {
    @Override
    public String enviar(String mensagem) {
        return "Notificação: " + mensagem;
    }
}
```

Propósito: É o objeto base que fornece a funcionalidade principal. Representa o comportamento "puro" sem decorações adicionais.

3. Decorator Base Abstrato

```java
abstract class NotificacaoDecorator implements Notificacao {
    protected Notificacao notificacao;

    public NotificacaoDecorator(Notificacao notificacao) {
        this.notificacao = notificacao;
    }

    @Override
    public String enviar(String mensagem) {
        return notificacao.enviar(mensagem);
    }
}
```

Elementos-chave:

- **Composição**: Mantém uma referência ao objeto `Notificacao` que está sendo decorado
- **Delegação**: Por padrão, delega a chamada para o objeto encapsulado
- **Herança**: Implementa a mesma interface, mantendo transparência
- **Proteção**: O atributo `protected` permite acesso pelas subclasses

4. Decoradores Concretos

EmailDecorator

```java
class EmailDecorator extends NotificacaoDecorator {
    public EmailDecorator(Notificacao notificacao) {
        super(notificacao);
    }

    @Override
    public String enviar(String mensagem) {
        return super.enviar(mensagem) + " + Email enviado";
    }
}
```

Funcionamento:

1. Chama `super.enviar(mensagem)` - executa o comportamento do objeto decorado
2. Adiciona sua própria funcionalidade: `" + Email enviado"`
3. Retorna o resultado combinado

Padrão Similar nos Outros Decoradores

- **SMSDecorator**: Adiciona `" + SMS enviado"`
- **SlackDecorator**: Adiciona `" + Slack notificado"`
- **CriptografiaDecorator**: Modifica a mensagem antes de passar adiante

5. Análise do Método Main

Construção Incremental

```java
Notificacao notificacao = new NotificacaoSimples();
notificacao = new EmailDecorator(notificacao);
notificacao = new SMSDecorator(notificacao);
```

O que acontece:

1. **Passo 1**: `notificacao` aponta para `NotificacaoSimples`
2. **Passo 2**: `notificacao` aponta para `EmailDecorator` que contém `NotificacaoSimples`
3. **Passo 3**: `notificacao` aponta para `SMSDecorator` que contém `EmailDecorator` que contém `NotificacaoSimples`

Estrutura em Camadas (Onion Pattern)

```
Unset
SMSDecorator
  └── EmailDecorator
      └── NotificacaoSimples
```

Fluxo de Execução

Quando `notificacao.enviar("mensagem")` é chamado:

1. **SMSDecorator.enviar()** é executado
2. Chama **EmailDecorator.enviar()** via `super.enviar()`
3. Que chama **NotificacaoSimples.enviar()** via `super.enviar()`
4. **NotificacaoSimples** retorna: `"Notificação: mensagem"`
5. **EmailDecorator** adiciona: `"Notificação: mensagem + Email enviado"`
6. **SMSDecorator** adiciona: `"Notificação: mensagem + Email enviado + SMS enviado"`

6. Configuração Dinâmica

```Java
public static void configurarNotificacao(String prioridade,
String mensagem) {
    Notificacao notificacao = new NotificacaoSimples();

    if (prioridade.equals("urgente")) {
        notificacao = new SMSDecorator(notificacao);
        notificacao = new EmailDecorator(notificacao);
        notificacao = new SlackDecorator(notificacao);
    } else {
        notificacao = new EmailDecorator(notificacao);
    }

    System.out.println("Prioridade " + prioridade + ": " +
notificacao.enviar(mensagem));
}
```

Demonstra:

- **Flexibilidade**: Diferentes combinações baseadas em condições
- **Runtime**: Decisões tomadas durante a execução
- **Reutilização**: Mesmo código base, comportamentos diferentes

7. Vantagens Demonstradas

Transparência

```Java
Notificacao n1 = new NotificacaoSimples();
Notificacao n2 = new EmailDecorator(new NotificacaoSimples());
// Ambos usam a mesma interface
```

Composição Flexível

```java
// Diferentes ordens produzem resultados diferentes
new EmailDecorator(new SMSDecorator(new NotificacaoSimples()));
new SMSDecorator(new EmailDecorator(new NotificacaoSimples()));
```

Extensibilidade

Para adicionar um novo tipo de notificação (ex: WhatsApp), basta criar:

```java
class WhatsAppDecorator extends NotificacaoDecorator {
    // implementação
}
```

8. Padrões de Design Aplicados

- **Single Responsibility**: Cada decorator tem uma responsabilidade específica
- **Open/Closed**: Aberto para extensão (novos decorators), fechado para modificação
- **Composition over Inheritance**: Usa composição para combinar comportamentos
- **Decorator Pattern**: Adiciona responsabilidades dinamicamente sem alterar a estrutura

Este exemplo demonstra como o padrão Decorator permite criar sistemas flexíveis onde comportamentos podem ser combinados de forma dinâmica e transparente.