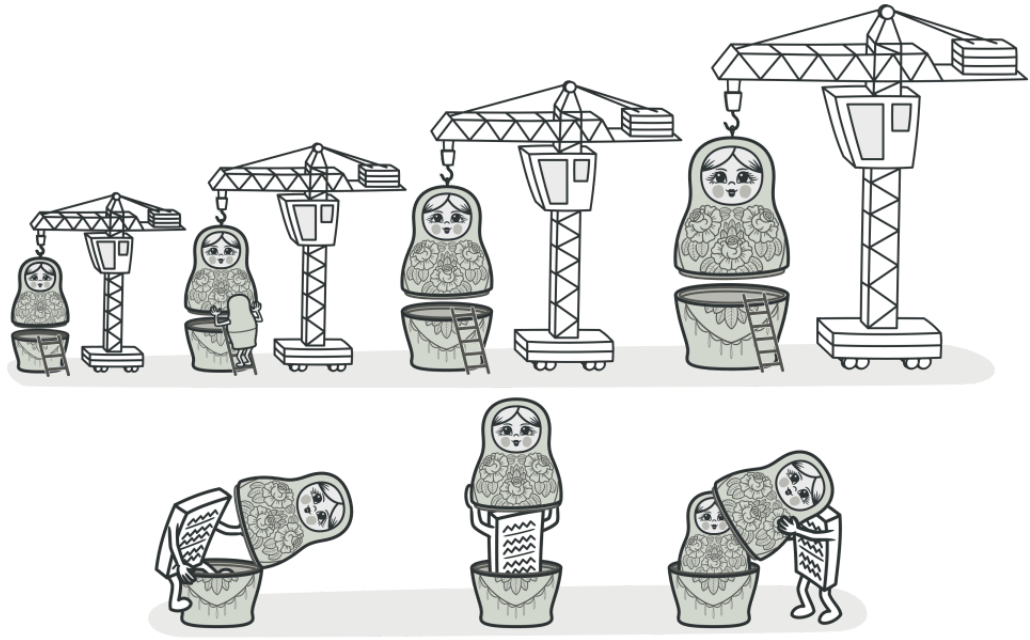
TÓPICO 12 - DECORATOR

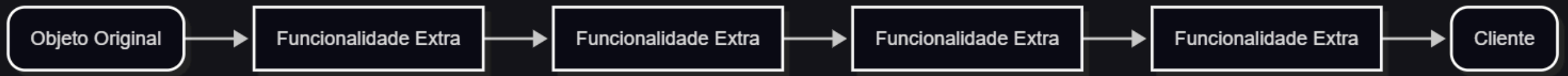
Design Patterns - Professor Ramon Venson - SATC 2026.1

Motivação

Quando funcionalidades extras são opcionais, a herança pode gerar uma explosão de subclasses.

O padrão Decorator resolve isso com composição em camadas **sem alterar a interface do cliente**.



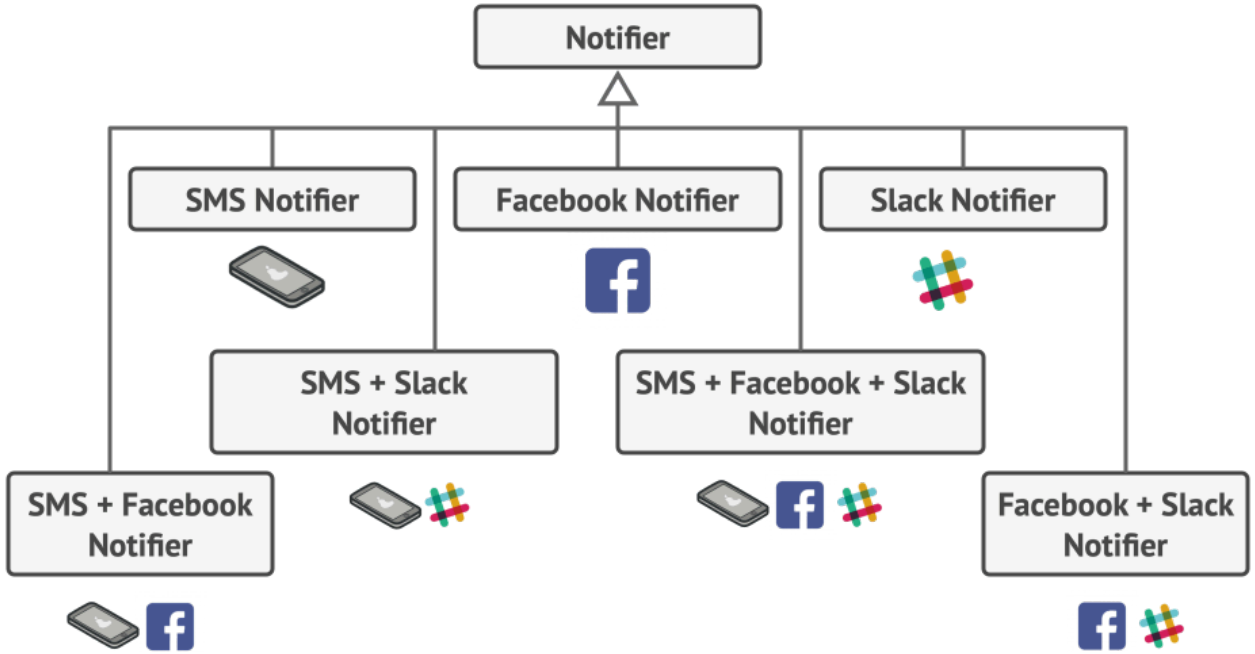

Objeto Original
Funcionalidade Extra
Funcionalidade Extra
Funcionalidade Extra
Funcionalidade Extra
Cliente

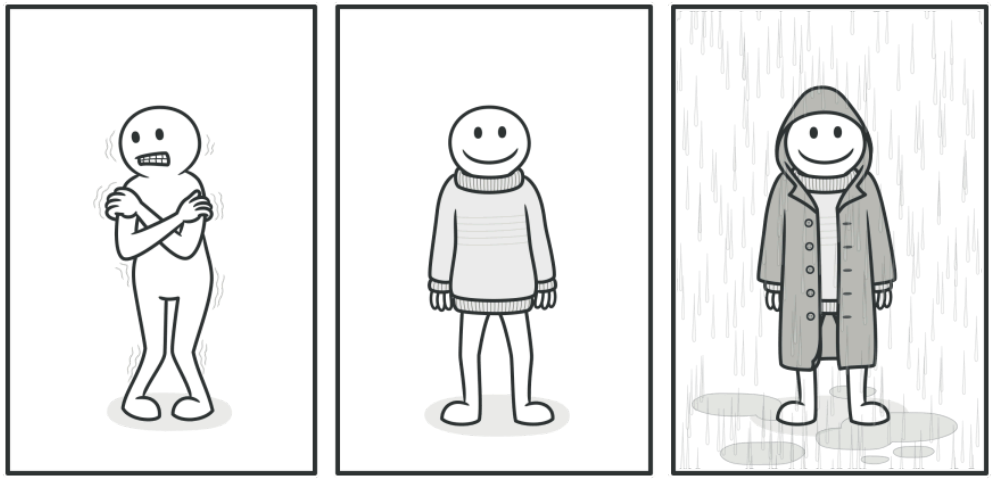

Pedido
Seguro Extra
Frete Expresso
Papel de Presente
Desconto Especial
Cliente

Problema

Imagine um sistema de biblioteca que apenas notifica via e-mail.

O que acontece quando queremos adicionar notificações por SMS ou Slack?



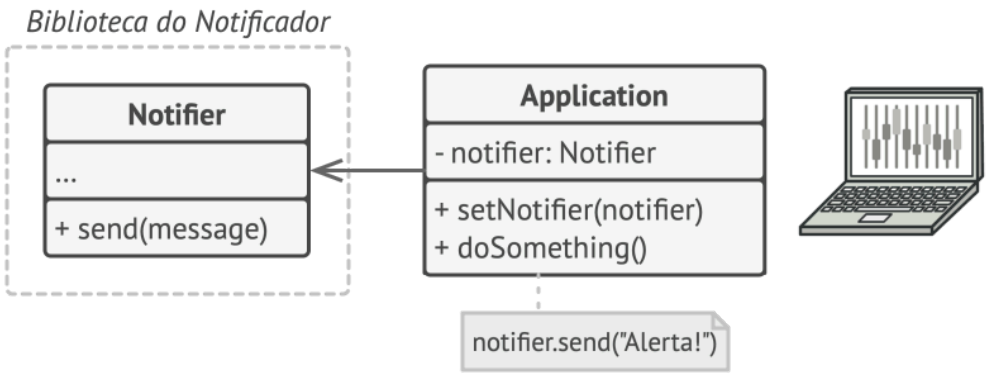


Conceito principal

- **Decorator** é um padrão estrutural
- Ele preserva a interface base
- Novos comportamentos envolvem o objeto original
- Cada camada delega e adiciona algo extra
- O cliente continua usando o mesmo contrato

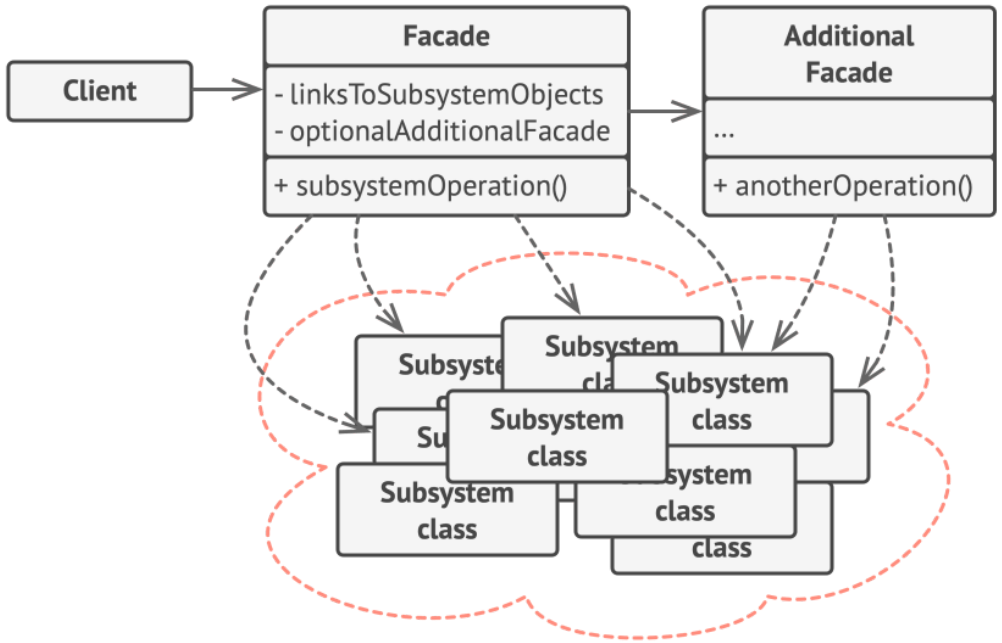
Como funciona

- Existe um componente base
- O decorador também implementa o mesmo contrato
- Ele guarda outro componente internamente
- Recebe a chamada, delega e acrescenta comportamento
- Várias camadas podem ser empilhadas



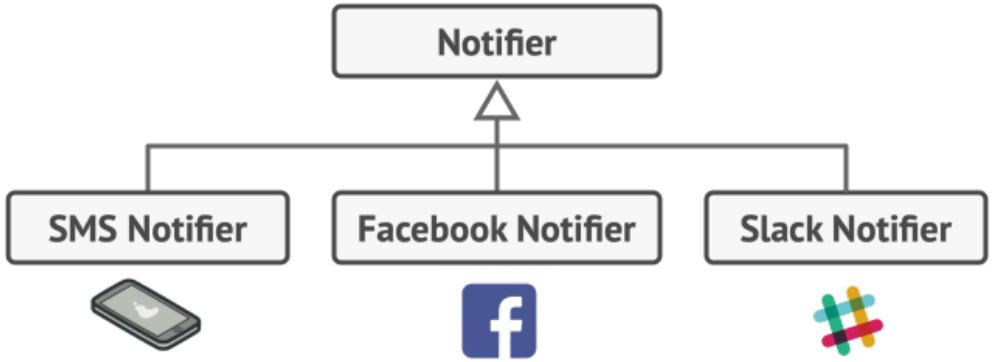
Estrutura / Componentes

- **Component** define a interface comum
- **ConcreteComponent** entrega o comportamento base
- **Decorator Base** encapsula outro componente
- **ConcreteDecorator** adiciona nova responsabilidade
- Cliente monta a pilha conforme a necessidade



Exemplo conceitual

- Sistema de notificações para uma universidade
- E-mail é o comportamento mínimo
- SMS pode ser uma camada opcional
- Slack pode ser outra camada opcional
- O cliente combina os canais sem novas subclasses



Código: contrato e componente base

```
interface Notificador {
    void enviar(String mensagem);
}

class EmailNotificador implements Notificador {
    public void enviar(String mensagem) {
        System.out.println("E-mail enviado: " + mensagem);
    }
}
```

Código: decorador base

```
abstract class NotificadorDecorator implements Notificador {
    protected Notificador wrappee;

    public NotificadorDecorator(Notificador wrappee) {
        this.wrappee = wrappee;
    }

    public void enviar(String mensagem) {
        wrappee.enviar(mensagem);
    }
}
```

Código: decorador concreto

```
class SmsDecorator extends NotificadorDecorator {
    public SmsDecorator(Notificador wrappee) {
        super(wrappee);
    }

    public void enviar(String mensagem) {
        super.enviar(mensagem);
        System.out.println("SMS enviado: " + mensagem);
    }
}
```

Quando usar

- Há responsabilidades opcionais
- As combinações variam por contexto
- Herança geraria muitas subclasses
- O comportamento deve mudar em execução
- O cliente deve manter o mesmo contrato

Quando não usar

- A funcionalidade extra é fixa para todos
- A ordem das camadas é muito sensível
- O número de decoradores ficaria confuso
- Uma composição mais simples já basta
- O comportamento deveria estar no componente base

Vantagens

- Evita explosão de subclasses
- Favorece composição sobre herança
- Permite combinar responsabilidades livremente
- Mantém o cliente desacoplado das variações
- Facilita extensão sem alterar classes prontas

Desvantagens

- Aumenta o número de classes pequenas
- Pode dificultar rastrear a execução
- Ordem dos decoradores pode importar
- Configuração inicial pode ficar verbosa
- Pilhas profundas exigem mais cuidado