2025-2 / Lorenzon : OO - Design Patterns - Observer

Design Patterns – Observer

O **Observer** é um também um dos padrões comportamentais do catálogo GoF (Gang of Four). Ele é usado quando precisamos **definir uma dependência um-para-muitos** entre objetos: quando o estado de um objeto muda, todos os seus dependentes (observadores) são notificados automaticamente.

Esse padrão é muito utilizado em sistemas que lidam com **eventos e notificações**, permitindo baixo acoplamento entre quem gera o evento (**Sujeito**) e quem reage ao evento (**Observadores**).

Quando usar o Observer

- Quando há necessidade de **notificação automática** de mudanças de estado;
- Quando se deseja manter **consistência entre objetos relacionados** sem acoplamento forte;
- Em sistemas de **eventos**, **interfaces gráficas** e **arquiteturas reativas**;
- Quando múltiplos objetos precisam reagir a mudanças em outro objeto, mas não queremos usar verificações constantes (polling).

Benefícios

- **Desacoplamento:** o sujeito não precisa conhecer detalhes dos observadores;
- **Flexibilidade:** novos observadores podem ser adicionados sem modificar o sujeito;
- **Reuso:** diferentes partes do sistema podem observar o mesmo evento;
- **Automação:** notificação é automática e transparente para o cliente.

O padrão Observer é um bom exemplo de uso de interfaces. Nosso exemplo tratará de um canal de notícias e tipos de assinantes.

Vamos criar uma interface Observador que indica quem quer ser observador, precisa ter esta ter implementado esta interface.

```
public interface Observador {

    void atualizar(String noticia);

}
```

Na sequencia criamos o repositório dos assinantes, denominado Observantes, quem mantém os assinantes e aciona a atualização aos Observadores.

```
/*public*/ class Observantes { //Repositorio dos assinantes, programação pra implementação
    //Associação 0..N
    private List< Observador > assinantes = new ArrayList<>();

    public void adicionarAssinante(Observador o) {
        assinantes.add(o);
    }

    public void removerAssinante(Observador o) {
        assinantes.remove(o);
    }

    public void publicarNoticia(String noticia) {
        System.out.println("Canal publicou: " + noticia);
        notificarAssinantes(noticia);
    }

    private void notificarAssinantes(String noticia) { //varredura d comunicação
        for (Observador o : assinantes) {
            o.atualizar(noticia);
        }
    }
}
```

Os assinantes, podem ser de vários tipos, pois podem ter várias formas de receber a notícia. Importante que, os assinantes sejam implementadores da interface Obervador.

```
/*public*/ class AssinanteEmail implements Observador {
    private String email;

    public AssinanteEmail(String email) {
        this.email = email;
    }

    @Override
    public void atualizar(String noticia) {
        System.out.println("Notificação enviada para " + email + ": " + noticia);
    }
}
```

```
/*public*/ class AssinanteApp implements Observador {
    private String nome;

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

    @Override
    public void atualizar(String noticia) {
        System.out.println("Notificação push para " + nome + ": " + noticia);
    }
}
```

Na aplicação, programações para a interface e não para a especialização, a classe concreta (implementação).

```
public class ObserverApp {

    public static void main(String[] args) {

        Observantes canal = new Observantes();

        //Programar para interface e não para implementação: Observador

        Observador assinante1 = new AssinanteEmail("joao@email.com");

        Observador assinante2 = new AssinanteApp("Maria");

        Observador assinante3 = new AssinanteApp("Carlos");


        canal.adicionarAssinante(assinante1);

        canal.adicionarAssinante(assinante2);

        canal.adicionarAssinante(assinante3);


        canal.publicarNoticia("Nova edição do jornal disponível!");

        canal.publicarNoticia("Promoção especial para assinantes!");

    }

}
```

O ponto de usar a **referência pela interface Observador** no seu ObserverApp é justamente um dos aspectos mais importantes do padrão **Observer** e, mais amplamente, da **programação orientada a abstrações**.

Você está dizendo que o que importa é o **contrato** (Observador), e não a forma concreta de implementação (AssinanteEmail, AssinanteApp). Isso dá liberdade para:

- Trocar a implementação sem mudar o código cliente (ObserverApp);
- Tratar todos os objetos que implementam Observador de maneira uniforme.

Imagine que amanhã você crie uma nova classe, no seu ObserverApp, não precisa mudar a lógica do canal, basta instanciar com a interface. O **código que notifica observadores não muda**, porque trabalha com a interface genérica.

Se o canal (Observantes) tivesse que conhecer cada classe concreta, você teria alto acoplamento e dificuldade para evoluir o sistema. Ao depender apenas da interface Observador, o canal não se importa se o assinante é App, Email, SMS ou qualquer outro. Isso cumpre o **princípio de inversão de dependência (D do SOLID)**: módulos de alto nível (o canal) não dependem de módulos de baixo nível (as classes concretas de assinantes), ambos dependem da abstração (Observador).

Sem interface: cada método teria que lidar com AssinanteEmail, AssinanteApp, etc. Isso gera código duplicado e rígido.

Com interface: todos são tratados como Observador. Basta chamar update(...), sem saber o "como" — cada classe cuida do seu próprio comportamento.

Considerações:

- **Sujeito (Subject):** mantém uma lista de observadores e os notifica em mudanças;
- **Observadores (Observador):** implementam a interface comum para reagir às notificações;
- **Notificação automática:** elimina necessidade de consultas repetitivas pelos observadores;
- **Baixo acoplamento:** o sujeito não precisa saber quem são os observadores, apenas que eles implementam a interface Observador;
- **Uso prático:** encontrado em sistemas de eventos, GUIs (ex.: Swing, JavaFX), bibliotecas reativas e notificações de sistemas distribuídos.