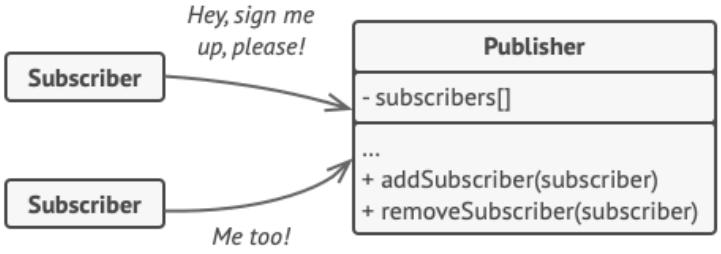
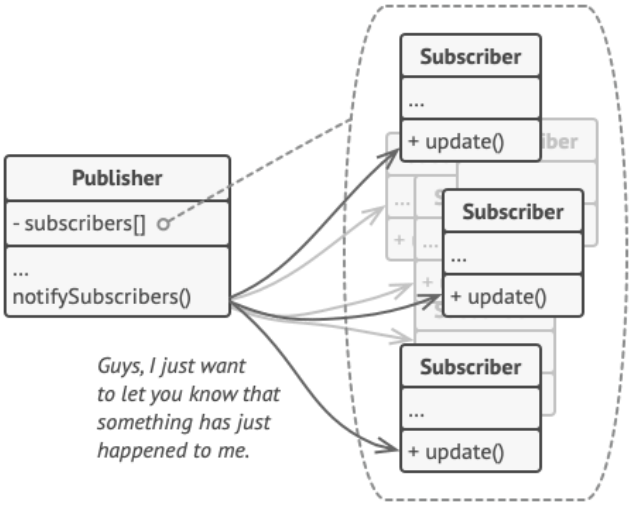
A linguagem escolhida foi Angular, ela possui alguns tipos de Design Patterns, como o Observer (Observador), o Strategy (Estratégia) e o Chain of Responsability (Cadeia de Responsabilidade). O qual iremos falar será o Observer, tanto por ser um Design Pattern conhecido, como por ter sido em partes já implementado no projeto de alguma forma.
Observer é um padrão comportamental (Design Patterns) que permite a definição de um mecanismo de assinatura em eventos para notificar outros objetos sobre os eventos que ocorreram no objeto que estão observando.
Observer pode ser explicado através de algo muito comum: ele é um aviso de disponibilidade.
Vamos imaginar a seguinte situação: você, cliente de uma loja, quer comprar um novo computador, você pode ir todos os dias até a loja para ver se já chegaram novas remessas com o computador que você quer, mas grande parte dessas idas a loja serão em vão, pois muito provavelmente o computador que você quer, estará na loja somente em um dos dias.
Uma opção para que você não fosse em vão a loja seria de que a loja enviasse um e-mail a todos os cliente comunicando a chegada de tal produto, o que seria péssimo do ponto de vista de usabilidade porque seria considerado um spam.
Para que você não necessite ir a loja todos os dias ver se o produto está disponível e nem que os e-mails da loja fossem considerados spam, a loja deveria manter um cadastro de todos os cliente que estão interessados no mesmo computador que você, e ela poderia fazer isso para cada produto em especifico. Assim quando os produtos mais requisitados chegassem à loja, ela avisaria através do e-mail para todos os interessados; seria poupado tempo de ir até a loja para todo mundo e também melhorando a qualidade de vida dos demais por não enviar spams.
Agora podemos explicar como isso funciona na prática:
Um objeto notificador, chamado de publisher possui uma lista de notificados (subscribers) e dois métodos, um para adicionar um novo subscriber e outro para remover algum já adicionado.

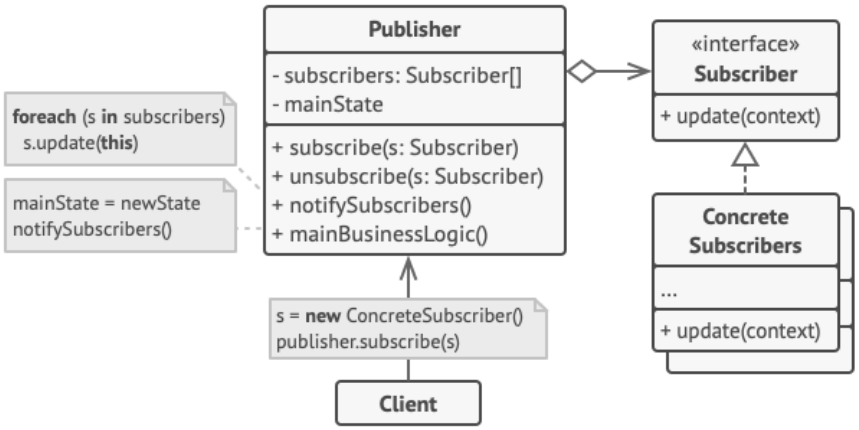


Porém ao acoplar o notificador a todos os inscritos seria criar um acoplamento de funcionalidade muito grande. Porque você precisaria implementar todas as formas de notificar diferentes classes de inscritos, por exemplo, alguns clientes querem receber um e-mail, outros um SMS. Por isso é importante que todos os observadores (subscribers) sigam uma mesma interface.

Assim, quando o objeto notificador chamar a sua função notifySubscribers, ele fará um loop por todos os inscritos, chamando o método update() de cada um deles, porque todos seguem e mesma interface:



Em uma sequência, o que acontecerá com o padrão será o seguinte:



a) O notificador (publisher) emite eventos de interesse para outros objetos. Estes eventos podem ocorrer durante uma alteração de estado ou durante outra modificação.

b) Quando um novo evento acontecer, o publisher itera sobre a lista de subscribers chamando o método responsável por atualizar seu contexto.

c) A interface do subscriber define um método update que será o método responsável por fazer a notificação para o subscriber.

d) Todo o subscriber concreto implementa a interface subscriber dessa forma desacoplamos o publisher de todos os subscribers.

e) O client cria o publisher e adiciona os subscribers a lista de objetos.

Exemplo de classe que utilizaria Observer:

```typescript
// Interface que todo o publisher deve seguir
interface Publisher {
    addSub(subscriber: Subscriber): void // Adiciona um subscriber
    removeSub(subscriber: Subscriber): void // Remove um subscriber
    notify(): void // Notifica todos os subscribers de um evento
}

// Um publisher que notifica os seus observadores sobre mudanças de estado
class ConcretePublisher implements Publisher {
    public state: number
    private subscribers: Subscriber[] = []

    public addSub (subscriber: Subscriber): void {
        const subExists = this.subscribers.include(subscriber)
        if (!subExists) this.subscribers.push(subscriber)
        console.log('Subscriber adicionado com sucesso')
    }

    public removeSub (subscriber: Subscriber): void {
        const index = this.subscribers.indexOf(subscriber)
        if (index === -1) return console.log('Subscriber não existe na lista')
        this.subscribers.splice(index, 1) // remove o subscriber
        console.log('Subscriber removido')
    }

    public notify (): void {
        console.log('Notificando observadores')
        for (const subscriber of this.subscribers) { subscriber.update(this) }
    }

     /**
    * Geralmente os publishers possuem um estado e uma regra de negócio
    * esta regra de negócio que emite a notificação quando seu estado é alterado
    */
    public logicaDeNegocio (): void {
        console.log('Publisher: Estou realizando uma regra de negócio')
        this.state = Math.floor(Math.random() * (10+1))
        console.log(`Publisher: Meu estado mudou para ${this.state}`)
        this.notify()
    }
}
```

```typescript
interface Subscriber {
  update (publisher: Publisher): void
}

class SubscriberA implements Subscriber {
  public update (publisher: Publisher): void {
    if (publisher instanceof ConcretePublisher && publisher.state < 3) console.log('SubscriberA reagiu ao evento emitido')
  }
}

class SubscriberB implements Subscriber {
  public update (publisher: Publisher): void {
    if (publisher instanceof ConcretePublisher && (publisher.state === 0 || publisher.state >= 2)) console.log('SubscriberB reagiu ao evento emitido')
  }
}

// Código do client
const publisher = new ConcretePublisher()

const observer1 = new SubscriberA()
publisher.addSub(observer1)

const observer2 = new SubscriberB()
publisher.addSub(observer2)

publisher.logicaDeNegocio()
publisher.logicaDeNegocio()

publisher.removeSub(observer2)

publisher.logicaDeNegocio()
```