refactoring.guru

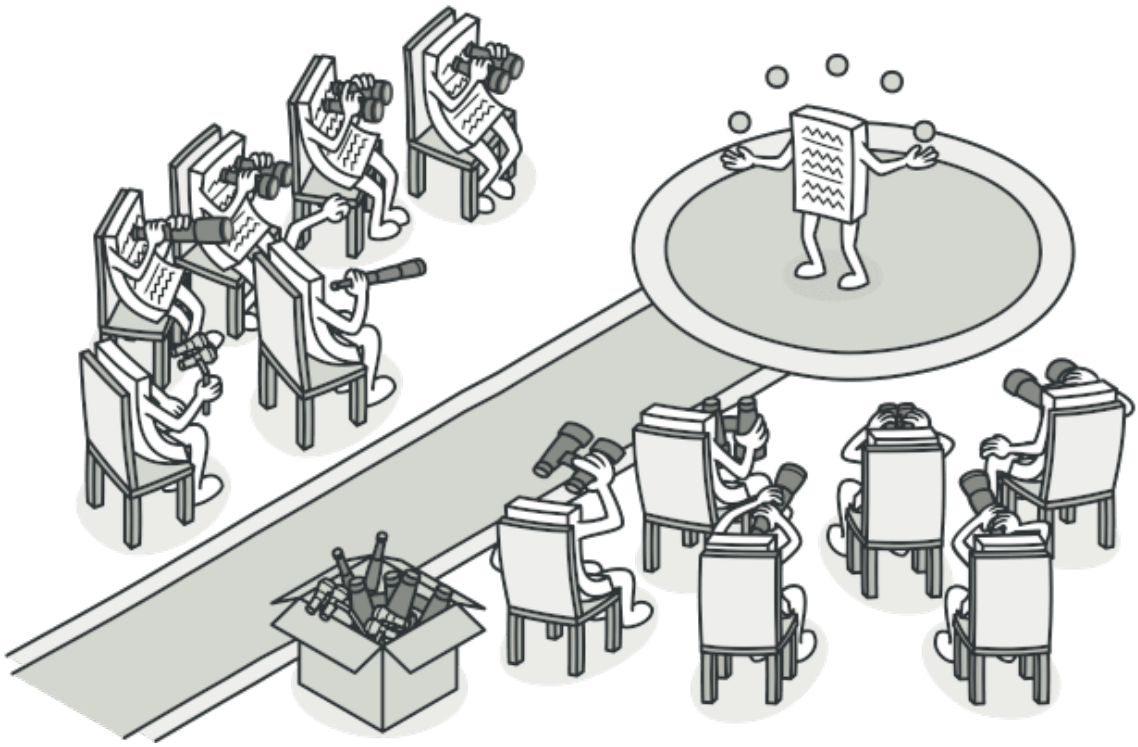
Observer

14–19 minutes

Também conhecido como: Observador, Assinante do evento, Event-Subscriber, Escutador, Listener

Propósito

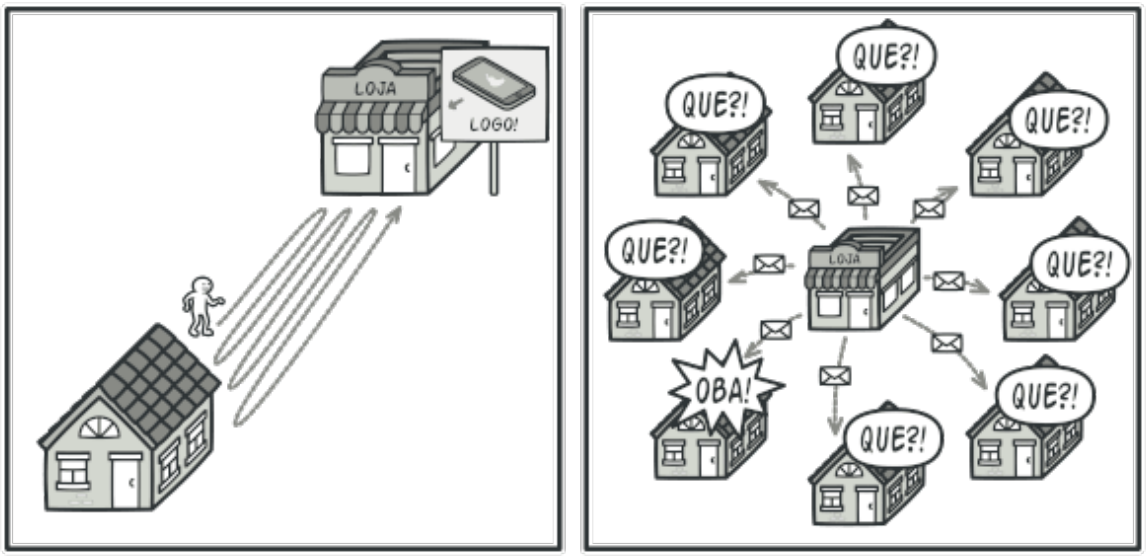
O **Observer** é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



Problema

Imagine que você tem dois tipos de objetos: um `Cliente` e uma `Loja`. O cliente está muito interessado em uma marca particular de um produto (digamos que seja um novo modelo de iPhone) que logo deverá estar disponível na loja.

O cliente pode visitar a loja todos os dias e checar a disponibilidade do produto. Mas enquanto o produto ainda está a caminho, a maioria desses visitas serão em vão.



Visitando a loja vs. enviando spam

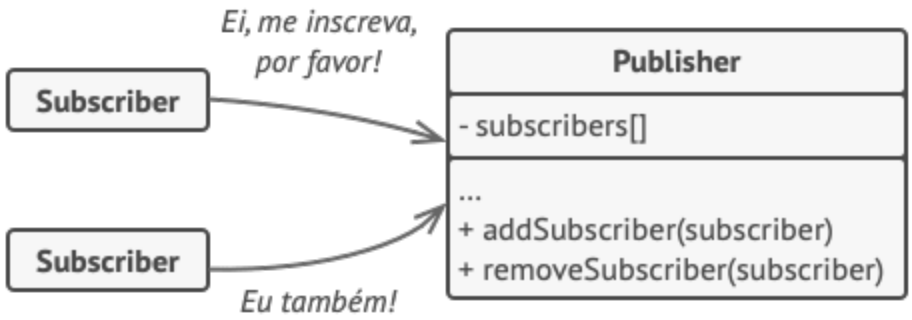
Por outro lado, a loja poderia mandar milhares de emails (que poderiam ser considerados como spam) para todos os clientes cada vez que um novo produto se torna disponível. Isso salvaria alguns clientes de incontáveis viagens até a loja. Porém, ao mesmo tempo, irritaria outros clientes que não estão interessados em novos produtos.

Parece que temos um conflito. Ou o cliente gasta tempo verificando a disponibilidade do produto ou a loja gasta recursos notificando os clientes errados.

Solução

O objeto que tem um estado interessante é quase sempre chamado de *sujeito*, mas já que ele também vai notificar outros objetos sobre as mudanças em seu estado, nós vamos chamá-lo de *publicador*. Todos os outros objetos que querem saber das mudanças do estado do publicador são chamados de *assinantes*.

O padrão Observer sugere que você adicione um mecanismo de assinatura para a classe publicadora para que objetos individuais possam assinar ou desassinar uma corrente de eventos vindo daquela publicadora. Nada tema! Nada é complicado como parece. Na verdade, esse mecanismo consiste em 1) um vetor para armazenar uma lista de referências aos objetos do assinante e 2) alguns métodos públicos que permitem adicionar assinantes e removê-los da lista.



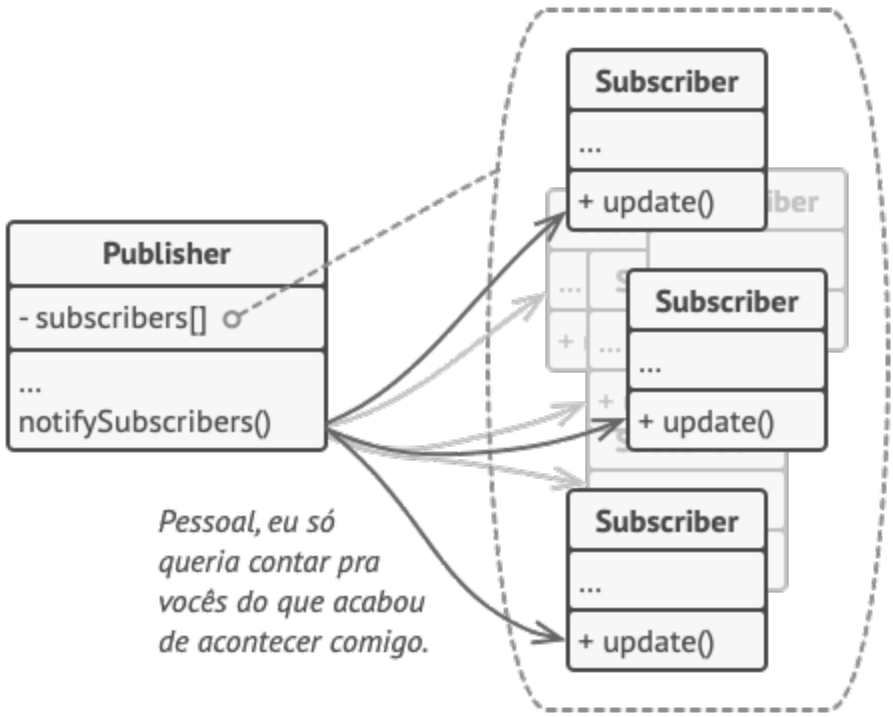
Um mecanismo de assinatura permite que objetos individuais inscrevam-se a notificações de eventos.

Agora, sempre que um evento importante acontece com a publicadora, ele passa para seus assinantes e chama um método específico de notificação em seus objetos.

Aplicações reais podem ter dúzias de diferentes classes assinantes que estão interessadas em acompanhar eventos da mesma classe publicadora. Você não iria querer acoplar a publicadora a todas essas classes. Além disso, você pode nem

estar ciente de algumas delas de antemão se a sua classe publicadora deve ser usada por outras pessoas.

É por isso que é crucial que todos os assinantes implementem a mesma interface e que a publicadora comunique-se com eles apenas através daquela interface. Essa interface deve declarar o método de notificação junto com um conjunto de parâmetros que a publicadora pode usar para passar alguns dados contextuais junto com a notificação.



A publicadora notifica os assinantes chamando um método específico de notificação em seus objetos.

Se a sua aplicação tem diferentes tipos de publicadoras e você quer garantir que seus assinantes são compatíveis com todas elas, você pode ir além e fazer todas as publicadoras seguirem a mesma interface. Essa interface precisa apenas descrever alguns métodos de inscrição. A interface permitirá assinantes observar o estado das publicadoras sem se acoplar a suas classes concretas.

Analogia com o mundo real

Assinaturas de revistas e jornais.

Se você assinar um jornal ou uma revista, você não vai mais precisar ir até a banca e ver se a próxima edição está disponível. Ao invés disso a publicadora manda novas edições diretamente para sua caixa de correio após a publicação ou até mesmo com antecedência.

A publicadora mantém uma lista de assinantes e sabe em quais revistas eles estão interessados. Os assinantes podem deixar essa lista a qualquer momento quando desejarem que a publicadora pare de enviar novas revistas para eles.

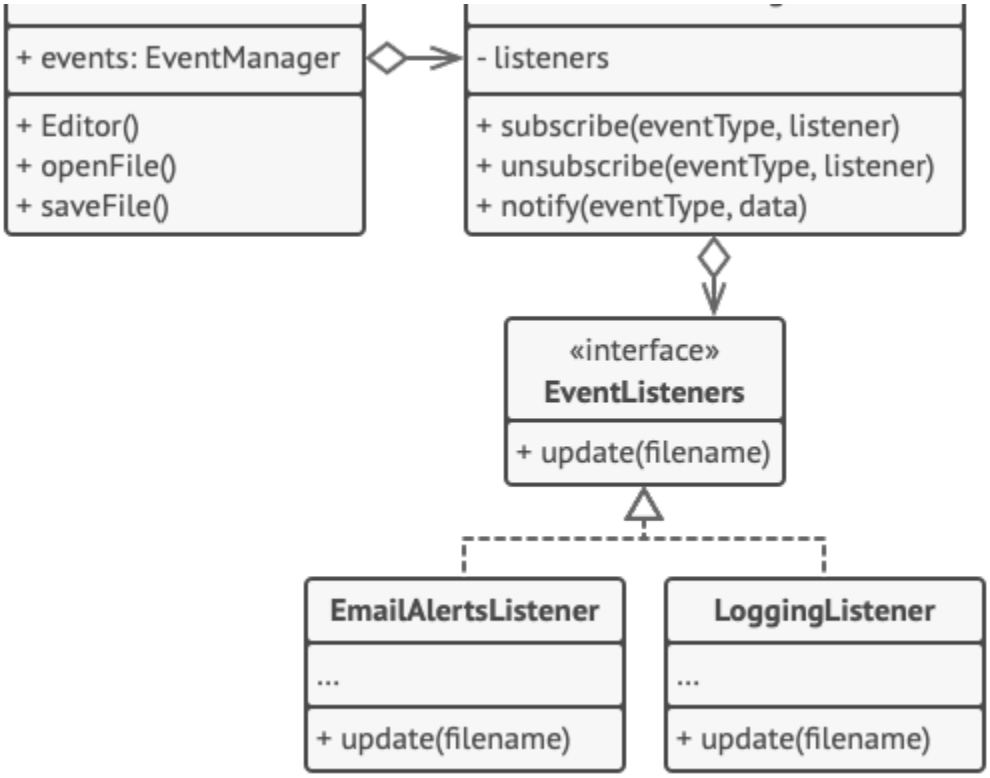
Estrutura

1. A **Publicadora** manda eventos de interesse para outros objetos. Esses eventos ocorrem quando a publicadora muda seu estado ou executa algum comportamento. As publicadoras contêm uma infraestrutura de inscrição que permite novos assinantes se juntar aos atuais assinantes ou deixar a lista.

2. Quando um novo evento acontece, a publicadora percorre a lista de assinantes e chama o método de notificação declarado na interface do assinante em cada objeto assinante.
3. A interface do **Assinante** declara a interface de notificação. Na maioria dos casos ela consiste de um único método `atualizar`. O método pode ter vários parâmetros que permite que a publicadora passe alguns detalhes do evento junto com a atualização.
4. **Assinantes Concretos** realizam algumas ações em resposta às notificações enviadas pela publicadora. Todas essas classes devem implementar a mesma interface para que a publicadora não fique acoplada à classes concretas.
5. Geralmente, assinantes precisam de alguma informação contextual para lidar com a atualização corretamente. Por esse motivo, as publicadoras quase sempre passam algum dado de contexto como argumentos do método de notificação. A publicadora pode passar a si mesmo como um argumento, permitindo que o assinante recupere quaisquer dados diretamente.
6. O **Cliente** cria a publicadora e os objetos assinantes separadamente e então registra os assinantes para as atualizações da publicadora.

Pseudocódigo

Neste exemplo o padrão **Observer** permite que um objeto editor de texto notifique outros objetos de serviço sobre mudanças em seu estado.



Notificando objetos sobre eventos que aconteceram com outros objetos.

A lista de assinantes é compilada dinamicamente: objetos podem começar ou parar de ouvir às notificações durante a execução do programa, dependendo do comportamento desejado pela sua aplicação.

Nesta implementação, a classe do editor não mantém a lista de assinatura por si mesmo. Ele delega este trabalho para um objeto ajudante especial devotado a fazer apenas isso. Você pode melhorar aquele objeto para servir como um enviador de eventos centralizado, permitindo que qualquer objeto aja como uma publicadora.

Adicionar novos assinantes ao programa não exige mudança nas classes publicadoras existentes, desde que elas trabalhem com todos os assinantes através da mesma interface.

```
// A classe publicadora base inclui o código de gerenciamento de
// inscrições e os métodos de notificação.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)

// O publicador concreto contém a verdadeira lógica de negócio
// que é de interesse para alguns assinantes. Nós podemos
// derivar essa classe a partir do publicador base, mas isso nem
// sempre é possível na vida real devido a possibilidade do
// publicador concreto já ser uma subclasse. Neste caso, você
// pode remendar a lógica de inscrição com a composição, como
// fizemos aqui.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Métodos da lógica de negócio podem notificar assinantes
```

```
    // acerca de mudanças.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)



// Aqui é a interface do assinante. Se sua linguagem de
// programação suporta tipos funcionais, você pode substituir
// toda a hierarquia do assinante por um conjunto de funções.
interface EventListener is
    method update(filename)

// Assinantes concretos reagem a atualizações emitidas pelo
// publicador a qual elas estão conectadas.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))
```

```
class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))


// Uma aplicação pode configurar publicadores e assinantes
// durante o tempo de execução.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

Aplicabilidade

Utilize o padrão Observer quando mudanças no estado de um objeto podem precisar mudar outros objetos, e o atual conjunto de objetos é desconhecido de antemão ou muda dinamicamente.

Você pode vivenciar esse problema quando trabalhando com classes de interface gráfica do usuário. Por exemplo, você criou classes de botões customizados, e você quer deixar os clientes colocar algum código customizado para seus botões para que ele ative sempre que usuário aperta um botão.

O padrão Observer permite que qualquer objeto que implemente a interface do assinante possa se inscrever para notificações de eventos em objetos da publicadora. Você pode adicionar o mecanismo de inscrição em seus botões, permitindo que o cliente coloque seu próprio código através de classes assinantes customizadas.

Utilize o padrão quando alguns objetos em sua aplicação devem observar outros, mas apenas por um tempo limitado ou em casos específicos.

A lista de inscrição é dinâmica, então assinantes podem entrar e sair da lista sempre que quiserem.

Como implementar

1. Olhe para sua lógica do negócio e tente quebrá-la em duas partes: a funcionalidade principal, independente de outros códigos, irá agir como publicadora; o resto será transformado em um conjunto de classes assinantes.
2. Declare a interface do assinante. No mínimo, ela deve declarar um único método `atualizar`.

3. Declare a interface da publicadora e descreva um par de métodos para adicionar um objeto assinante e removê-lo da lista. Lembre-se que publicadoras somente devem trabalhar com assinantes através da interface do assinante.

4. Decida onde colocar a lista atual de assinantes e a implementação dos métodos de inscrição. Geralmente este código se parece o mesmo para todos os tipos de publicadoras, então o lugar óbvio para colocá-lo é dentro de uma classe abstrata derivada diretamente da interface da publicadora. Publicadoras concretas estendem aquela classe, herdando o comportamento de inscrição.

   Contudo, se você está aplicando o padrão para uma hierarquia de classe já existente, considere uma abordagem baseada na composição: coloque a lógica da inscrição dentro de um objeto separado, e faça todos as publicadoras reais usá-la.

5. Crie as classes publicadoras concretas. A cada vez que algo importante acontece dentro de uma publicadora, ela deve notificar seus assinantes.

6. Implemente os métodos de notificação de atualização nas classes assinantes concretas. A maioria dos assinantes precisarão de dados contextuais sobre o evento. Eles podem ser passados como argumentos do método de notificação.

   Mas há outra opção. Ao receber uma notificação, o assinante pode recuperar os dados diretamente da notificação. Neste caso, a publicadora deve passar a si mesma através do método de atualização. A opção menos flexível é ligar uma publicadora ao assinante permanentemente através do construtor.

7. O cliente deve criar todas os assinantes necessários e registrá-los

com suas publicadoras apropriadas.

Prós e contras

- *Princípio aberto/fechado*. Você pode introduzir novas classes assinantes sem ter que mudar o código da publicadora (e vice versa se existe uma interface publicadora).
- Você pode estabelecer relações entre objetos durante a execução.
- Assinantes são notificados em ordem aleatória

Relações com outros padrões

- O Chain of Responsibility, Command, Mediator e Observer abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:
- O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
- O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
- O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
- O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.
- A diferença entre o Mediator e o Observer é bem obscura. Na maioria dos casos, você pode implementar qualquer um desses

padrões; mas às vezes você pode aplicar ambos simultaneamente. Vamos ver como podemos fazer isso.

O objetivo primário do *Mediator* é eliminar dependências múltiplas entre um conjunto de componentes do sistema. Ao invés disso, esses componentes se tornam dependentes de um único objeto mediador. O objetivo do *Observer* é estabelecer comunicações de uma via dinâmicas entre objetos, onde alguns deles agem como subordinados de outros.

Existe uma implementação popular do padrão Mediator que depende do *Observer*. O objeto mediador faz o papel de um publicador, e os componentes agem como assinantes que inscrevem-se ou removem a inscrição aos eventos do mediador. Quando o *Mediator* é implementado dessa forma, ele pode parecer muito similar ao *Observer*.

Quando você está confuso, lembre-se que você pode implementar o padrão Mediator de outras maneiras. Por exemplo, você pode ligar permanentemente todos os componentes ao mesmo objeto mediador. Essa implementação não se parece com o *Observer* mas ainda irá ser uma instância do padrão Mediator.

Agora imagine um programa onde todos os componentes se tornaram publicadores permitindo conexões dinâmicas entre si. Não haverá um objeto mediador centralizado, somente um conjunto distribuído de observadores.