



Observer

O Observer é um padrão de projeto de software que define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente.

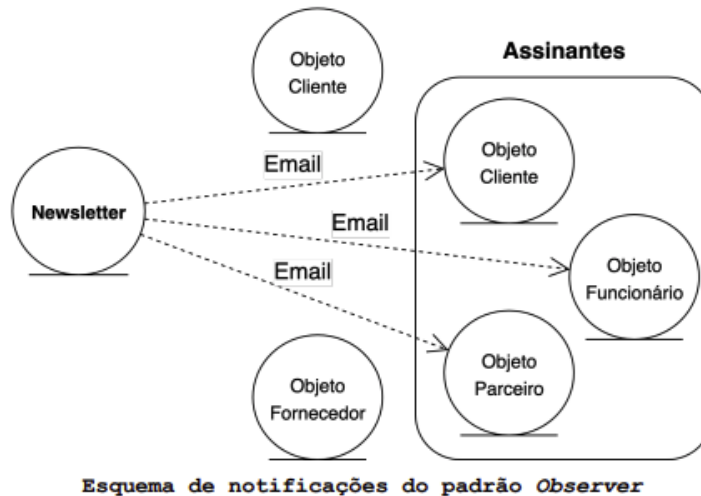
Motivação (Por que utilizar?)

Um sistema pode precisar manter a consistência entre objetos relacionados. Não é recomendado garantir tal consistência tendo como efeito colateral tornar as classes fortemente acopladas, pois isso reduz sua reutilização. Um objeto que se relaciona com outros objetos deve permitir que seus elementos sejam acessados sem que a sua estrutura interna seja exposta.

Para garantir que objetos dependentes entre si possam propagar suas mudanças de estado o padrão observer propõe que:

- Os observadores (observers) devem conhecer o objeto de interesse.
- O objeto de interesse (subject) deve notificar os observadores quando for atualizado.

Os objetos devem interligar-se entre si sem que se conheçam em tempo de compilação. Tomemos como exemplo a implementação de uma Newsletter para uma empresa (serviço de assinatura de emails) onde clientes, funcionários, parceiros e fornecedores podem se inscrever para receber emails de notícias sobre a empresa. A Newsletter é nosso objeto de interesse, portanto ela é nosso subject e os clientes, funcionários, parceiros e fornecedores são os observers.



A criação de um novo e-mail é uma mudança no estado de newsletter, tal e-mail deve ser enviado a todos os assinantes. No esquema acima temos uma instância de **Cliente** e outra de **Fornecedor** que ainda não são assinantes da newsletter e por isso não receberam e-mail, porém, tais objetos podem se tornar assinantes a qualquer momento. Da mesma forma qualquer assinante pode cancelar sua assinatura e deixar de receber emails.

A newsletter precisa saber como notificar todos os assinantes a respeito do novo email. Para que isso seja possível eles precisam implementar um método em comum. É possível garantir isso fazendo eles implementarem uma interface em comum.

```
export interface IObserver {
  update(message: string): string
  getName(): string
  getEmail(): string
}
```

Na interface acima apenas o método `update()` é parte do padrão observer, os métodos `getName()` e `getEmail()` fazem parte do contexto do problema.

Para viabilizar nosso exemplo considere a classe Email, que simula o envio de emails aos objetos Observers.

```
import { IObserver } from "../interface/observer.intarfece";

export class Email {
```

```

    public static sendMail(observer: IObserver, message: string) {
        console.log(`Sending email to ${observer.getName()} - ${observer.getEmail()}`)
        console.log(`Message: ${message}`)
    }
}

```

Ela recebe como parâmetro um objeto Observer e a mensagem que será enviada. Vamos agora a implementação dos observers.

```

import { IObserver } from '../interface/observer.intarfece'
import { ISubject } from '../interface/subject.interface'
import { Email } from './email';

export class Client implements IObserver {

    constructor(private name: string, private email: string, private subject: ISubject) {
        this.name = name
        this.email = email
        this.subject = subject
        this.subject.registerObserver(this)
    }

    update(message: string): void {
        Email.sendMail(this, message)
    }

    getName(): string {
        return this.name
    }

    getEmail(): string {
        return this.email
    }

}

```

```

import { IObserver } from '../interface/observer.intarfece'
import { ISubject } from '../interface/subject.interface'
import { Email } from './email';

export class Employee implements IObserver {

    constructor(private name: string, private email: string, private subject: ISubject) {
        this.name = name
        this.email = email
        this.subject = subject
        this.subject.registerObserver(this)
    }

}

```

```

    update(message: string): void {
        Email.sendMail(this, message)
    }
    getName(): string {
        return this.name
    }
    getEmail(): string {
        return this.email
    }
}

```

```

import { IObservable } from "../interface/observable.interface";
import { ISubject } from '../interface/subject.interface'

import { Email } from "../email";

export class Partner implements IObservable {

    constructor(private name: string, private email: string, private subject: ISubject) {
        this.name = name
        this.email = email
        this.subject = subject
        this.subject.registerObserver(this)
    }

    update(message: string): void {
        Email.sendMail(this, message)
    }
    getName(): string {
        return this.name
    }
    getEmail(): string {
        return this.email
    }
}

```

```

import { IObservable } from "../interface/observable.interface";
import { ISubject } from '../interface/subject.interface'

import { Email } from "../email";

export class Provider implements IObservable {

    constructor(private name: string, private email: string, private subject: ISubject) {
        this.name = name
        this.email = email
    }
}

```

```

    this.subject = subject
    this.subject.registerObserver(this)
}

update(message: string): void {
    Email.sendMail(this, message)
}

getName(): string {
    return this.name
}

getEmail(): string {
    return this.email
}
}

```

Em nosso exemplo as classes `Cliente`, `Funcionario`, `Parceiro` e `Fornecedor` são muito parecidas, poderiam inclusive ser subclasses de uma classe mais genérica para reduzir a duplicidade de código. Isso é proposital para simplificar o exemplo e focarmos no conceito do padrão `Observer`. Porém tais classes poderiam ser completamente distintas, a única coisa que precisam ter em comum é a implementação da interface `Observer`.

Agora a newsletter sabe que pode utilizar o método `update()` para notificar seus observers.

Da mesma forma que a newsletter precisa de garantias a respeito de seus observadores um observador precisa saber se um objeto é capaz de notificá-lo, ou seja, se o objeto é um subject.

Um subject deve ser capaz de:

- Adicionar observers a sua lista de objetos a serem notificados;
- Remover observers de sua lista de objetos a serem notificados;
- Notificar observers da sua lista de objetos a serem notificados.

É preciso criar um supertipo (interface) para os objetos observáveis para garantir aos observadores que suas necessidades serão supridas.

```
import { IObserver } from './observer.intarfece'

export interface ISubject {
  registerObserver(observer: IObserver): void
  removeObserver(observer: IObserver): void
  notifyObservers(): void
}
```

A newsletter é nosso objeto observável, portanto, ela implementa a interface Subject que será esperada pelos Observers.

```
import { IObserver } from '../interface/observer.intarfece';
import { ISubject } from '../interface/subject.interface'

export class Newsletter implements ISubject {

  private observers: IObserver[]
  private messages: string[]

  registerObserver(observer: IObserver): void {
    this.observers.push(observer)
  }

  removeObserver(observer: IObserver): void {
    const index = this.observers.indexOf(observer)
    this.observers.splice(index, 1)
  }

  /**
   * Notifica todos os observers sobre a nova mensagem
   */
  notifyObservers(): void {
    this.observers.forEach(observer => {
      observer.update(this.messages[this.messages.length - 1])
    })
  }

  addMessage(message: string): void {
    this.messages.push(message)
    this.notifyObservers()
  }
}
```

Deste modo os observers têm garantia de que irão se inscrever em um objeto capaz de notificá-los sempre que existir uma nova mensagem na Newsletter.

Vamos aos testes:

```
import { Newsletter } from './class/newsletter'
import { Employee } from './class/employee'
import { Client } from './class/client'
import { Partner } from './class/partner'
import { Provider } from './class/provider'

const newsletter = new Newsletter()

const employee1 = new Employee('Diego Almeida', 'diego.almeida@gmail.com', newsletter)
const client1 = new Client('Laura', 'laura.almeida@gmail.com', newsletter)
const provider1 = new Provider('Mayara Rocha', 'mayrocha@gmail.com', newsletter)
const partner1 = new Partner('Marcos Lopes', 'marcos.lopes@gmail.com', newsletter)

/**
 * Envia a primeira mensagem
 */
newsletter.addMessage('Vasco vence itano por 1 x 0 e retorna a elite do futebol brasileiro')

/**
 * Remove cliente da inscrição
 */
newsletter.removeObserver(client1)
newsletter.addMessage('Vasco demite técnico jorginho e inicia um novo planejamento')

/**
 * Adiciona o cliente novamente na inscrição
 */
newsletter.registerObserver(client1)
newsletter.addMessage('Vasco entra em contato com técnico português')
```

Saída:

```
Sending email to Diego Almeida - diego.almeida@gmail.com
Message: Vasco vence itano por 1 x 0 e retorna a elite do futebol brasileiro

Sending email to Laura - laura.almeida@gmail.com
Message: Vasco vence itano por 1 x 0 e retorna a elite do futebol brasileiro

Sending email to Mayara Rocha - mayrocha@gmail.com
Message: Vasco vence itano por 1 x 0 e retorna a elite do futebol brasileiro

Sending email to Marcos Lopes - marcos.lopes@gmail.com
Message: Vasco vence itano por 1 x 0 e retorna a elite do futebol brasileiro
```

```
Sending email to Diego Almeida - diego.almeida@gmail.com
Message: Vasco demite técnico jorginho e inicia um novo planejamento
```

```
Sending email to Mayara Rocha - mayrocha@gmail.com
Message: Vasco demite técnico jorginho e inicia um novo planejamento
```

```
Sending email to Marcos Lopes - marcos.lopes@gmail.com
Message: Vasco demite técnico jorginho e inicia um novo planejamento
```

```
Sending email to Diego Almeida - diego.almeida@gmail.com
Message: Vasco entra em contato com técnico português
```

```
Sending email to Mayara Rocha - mayrocha@gmail.com
Message: Vasco entra em contato com técnico português
```

```
Sending email to Marcos Lopes - marcos.lopes@gmail.com
Message: Vasco entra em contato com técnico português
```

```
Sending email to Laura - laura.almeida@gmail.com
Message: Vasco entra em contato com técnico português
```

Devido a utilização do padrão Observer o código passa a obedecer alguns bons princípios de programação orientada a objetos:

1. Programe para abstrações: Newsletter (subject) e Cliente, Funcionario, Parceiro e Fornecedor (Observers) usam interfaces. O subject monitora os objetos que implementam a interface Observer enquanto os observadores registram e são notificados pela interface Subject.
2. Mantenha objetos que se relacionam levemente ligados:
 - A única coisa que o subject sabe sobre os observers é que eles implementam a interface Observer.
 - Subjects e Observers podem ser reutilizados separadamente, um não depende do outro de forma concreta.
3. Open-closed principle:
 - Novos observadores podem ser adicionados a qualquer momento sem a necessidade de modificar o subject.
 - Alterações no Subject e Observer não afetarão um ao outro.

4. De prioridade a composição em relação à herança: O padrão observer utiliza a composição para compor, em tempo de execução, um subject com qualquer número de observers.

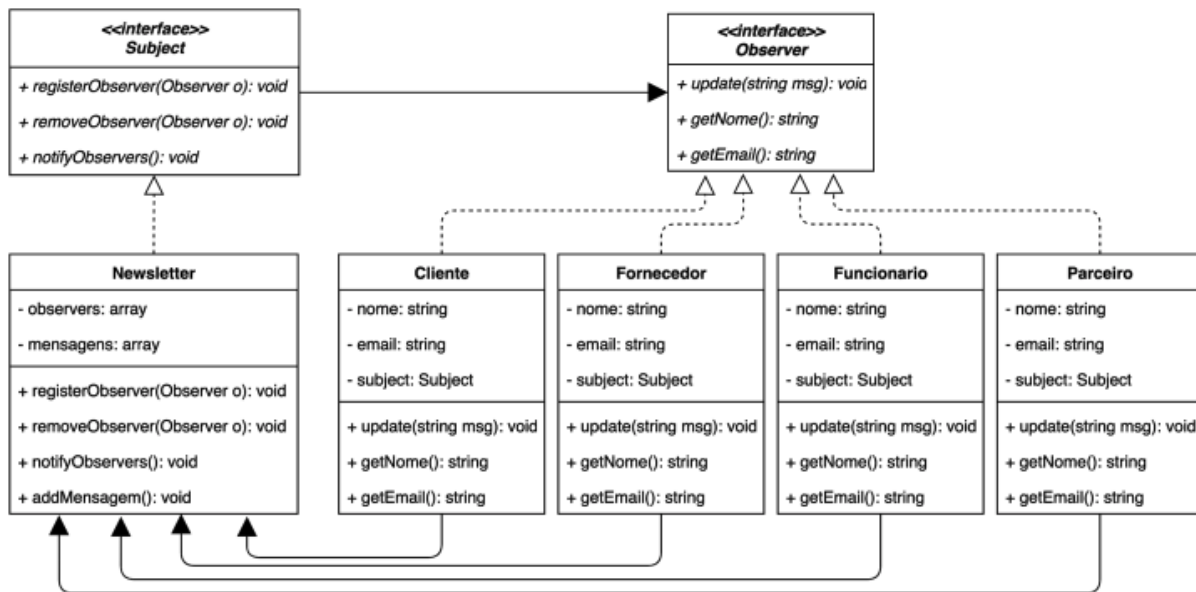


Diagrama de Classes do Exemplo

Aplicabilidade (Quando utilizar?)

- Quando uma abstração tem dois aspectos, um depende do outro, e é necessário que eles possam variar e serem reutilizados independentemente.
- Quando uma alteração em um objeto requer a alteração de outros, e não se conhece quantos objetos precisam ser alterados.
- Quando um objeto deve ser capaz de notificar outros objetos sem os conhecer, ou seja, tais objetos não podem ser fortemente acoplados.

Componentes

- **Subject:** Os objetos utilizam esta interface para se registrarem como observadores e para serem removidos.
- **Observer:** Define uma interface de atualização para objetos que devem ser notificados sobre alterações em um Subject.

- **AssuntoConcreto**: Sempre implementa a interface Subject além dos métodos para registrar e remover observers, o AssuntoConcreto implementa o método `notifyObservers()` que é utilizado para atualizar todos os observadores atuais sempre que o estado do **AssuntoConcreto** é alterado. Também pode ter métodos para definir e obter seu estado.
- **ObservadoresConcretos**: Podem ser qualquer classe que implemente a interface `Observer`. Cada observador se registra a um **AssuntoConcreto** para receber atualizações. Mantém uma referência a um objeto AssuntoConcreto (que é observado por ele). Tal referência serve para saber de onde vem as notificações e para poder se registrar e se remover.

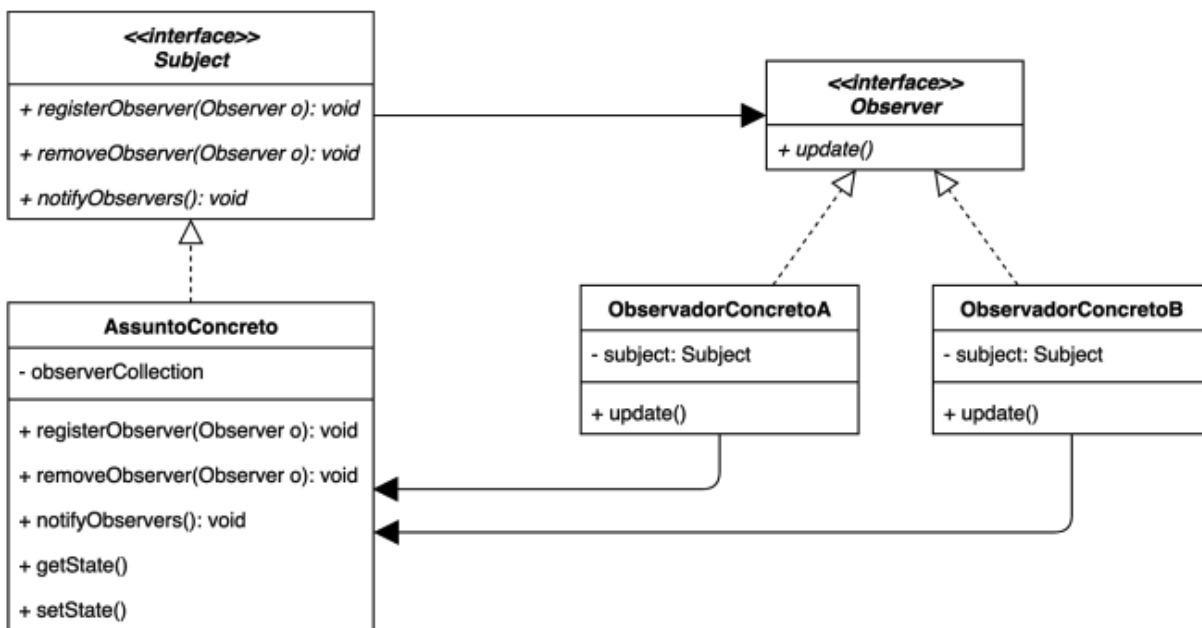


Diagrama de Classes

Consequências

- O padrão Observer permite variar assuntos (subject) e observadores (observers) de forma independente. É possível reutilizar assuntos sem reutilizar seus observadores e vice-versa. Também permite adicionar observadores sem modificar o assunto ou outros observadores.
- Acoplamento abstrato entre Assunto e Observador. Tudo que um assunto sabe é que ele possui uma lista de observadores, cada um em conformidade com a

interface Observer. O assunto não conhece a classe concreta de nenhum observador. Assim, o acoplamento entre assunto e seus observadores é abstrato e mínimo.

- Suporte para comunicação via broadcast. Ao contrário de uma solicitação comum, a notificação que um assunto envia não precisa especificar seu destinatário. A notificação é transmitida automaticamente para todos os objetos observadores que se inscreveram. O assunto não se importa com quantos objetos interessados existem, sua única responsabilidade é notificar seus observadores. Isso lhe dá a liberdade de adicionar e remover observadores a qualquer momento. Cabe ao observador manipular ou ignorar uma notificação.
- Pode causar atualizações inesperadas. Como os observadores não se conhecem, uma operação simples sobre o assunto pode causar uma cascata de atualizações em seus observadores e seus objetos dependentes. Além disso, critérios de dependência que não são bem gerenciados geralmente levam a atualizações desnecessárias, que podem ser difíceis de rastrear.