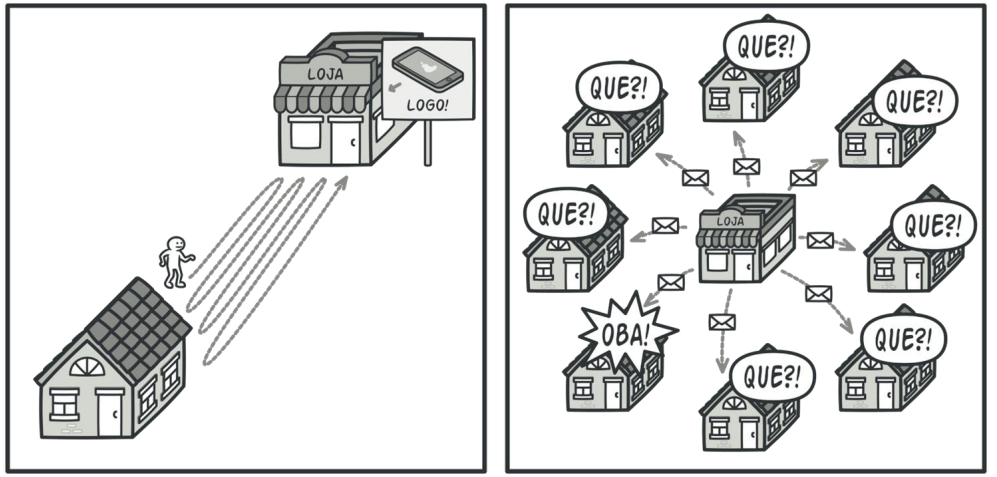
Observer

Padrões de Projeto
Comportamentais I

Prof. Me Jefferson Passerini

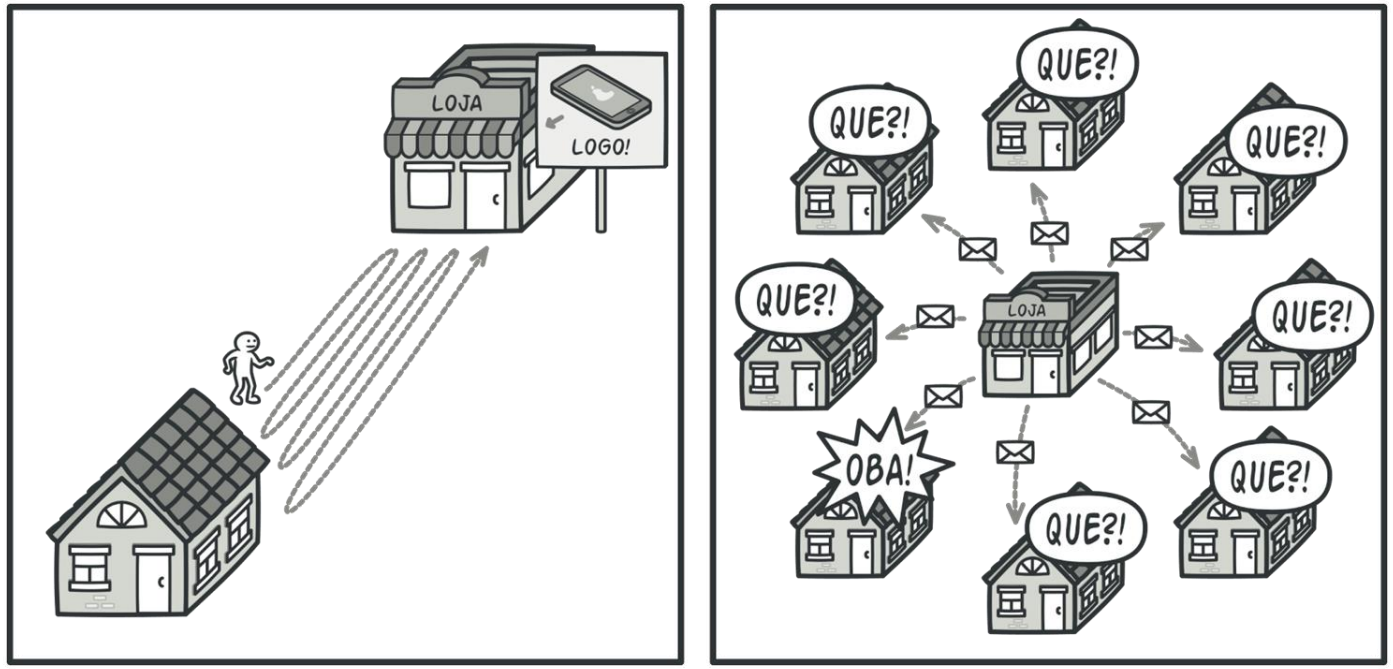
O ***Observer*** é um padrão de projeto de software que define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda seu estado, todos seus dependentes são notificados e atualizados automaticamente.

- **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 dessas visitas serão em vão.
- Por outro lado, a loja poderia mandar milhares de e-mails (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.

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

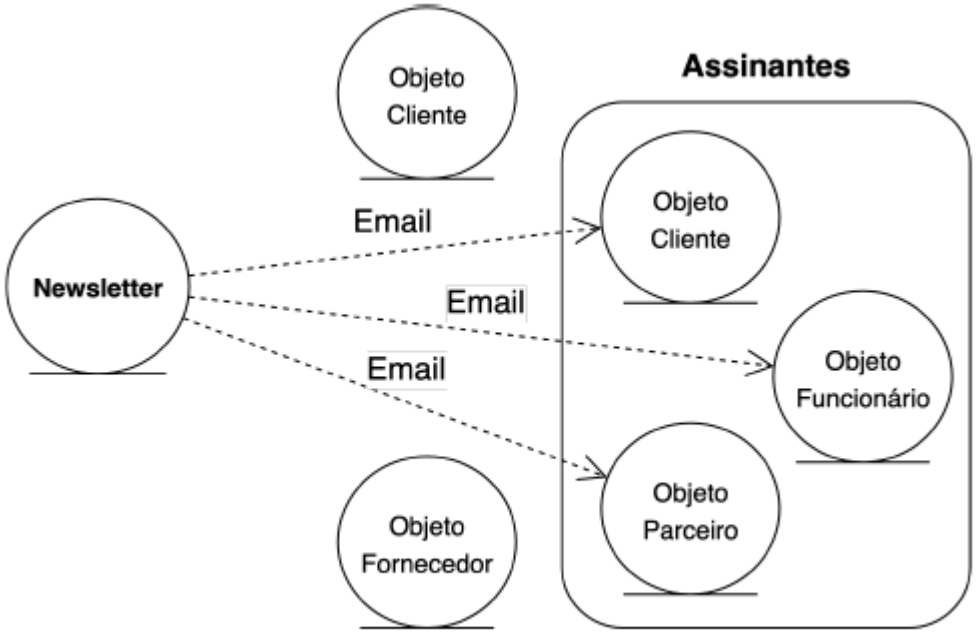


• Motivação

- 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.

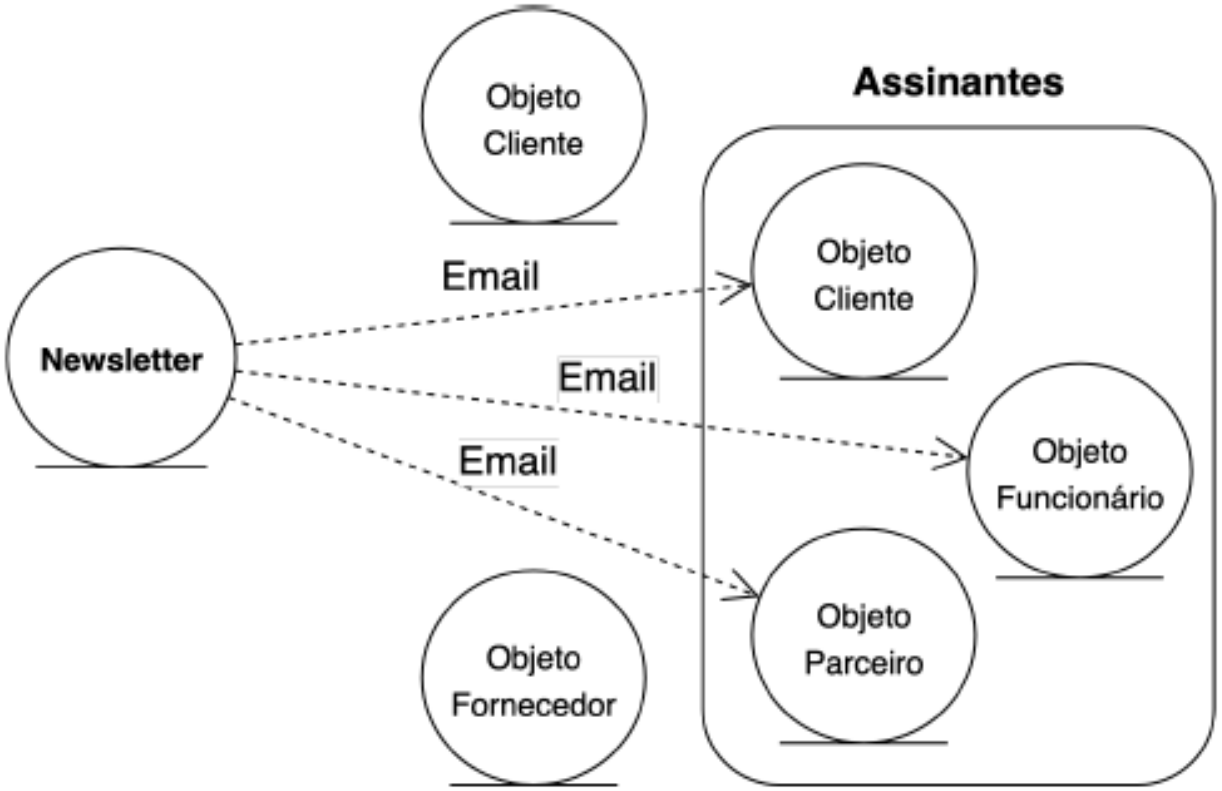
- Um objeto que se relaciona com outros objetos deve permitir que seus elementos sejam acessados sem que a sua estrutura interna seja exposta.



Esquema de notificações do padrão *Observer*

• Motivação

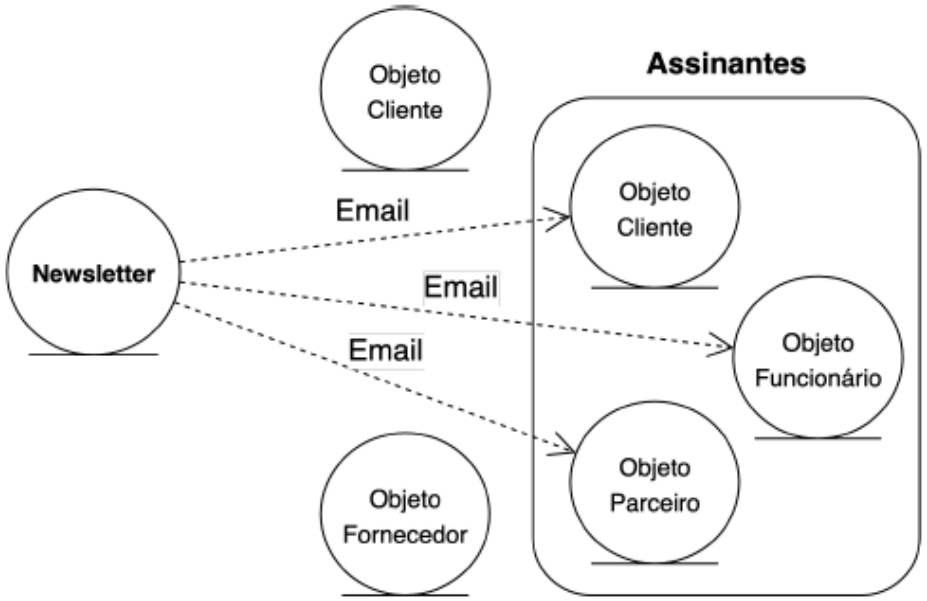
- Para garantir que objetos dependentes entre si possam propagar suas
- mudanças de estado o padrão ***Observer*** propõe que:
  - Os observadores (**observers**) devem conhecer o objeto de interesse.
  - O objeto de interesse (**subject**) deve notificar os observadores quando for atualizado.



Esquema de notificações do padrão *Observer*

- **Motivação**

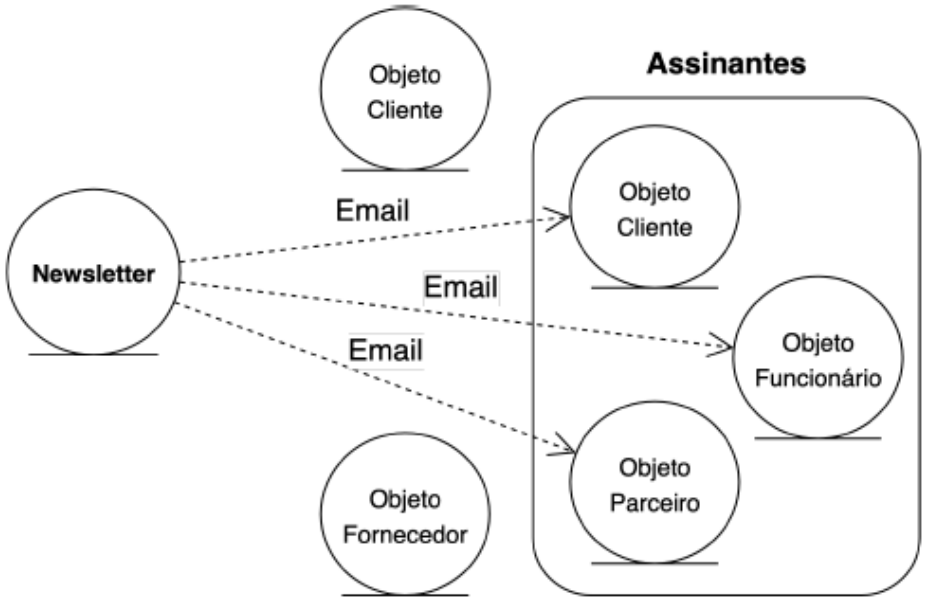
- Os objetos devem interligar-se entre si sem que se conheçam em tempo de compilação.
- Tomemos como exemplo a implementação de uma Newsletter para uma empresa (serviço de assinatura de emails) onde clientes, funcionários, parceiros e fornecedores podem se inscrever para receber emails de notícias sobre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



Esquema de notificações do padrão *Observer*

- **Motivação**

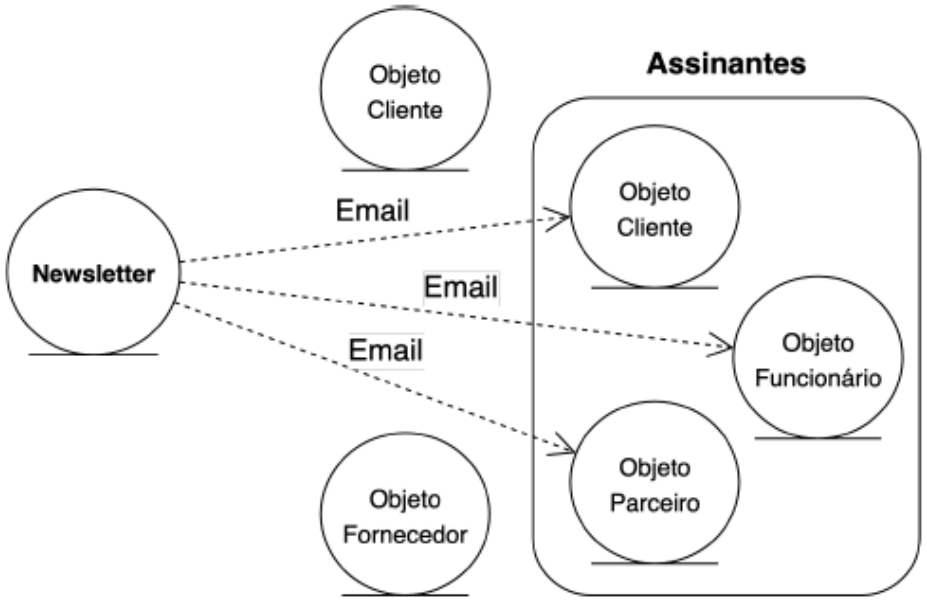
- A criação de um novo e-mail é uma mudança no estado de newsletter;
- Tal e-mail deve ser enviado a todos os assinantes.
- No esquema temos uma instância de Cliente e outra de Fornecedor que ainda não são assinantes da newsletter e por isso não receberam e-mail, porém, tais objetos podem se tornar assinantes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



Esquema de notificações do padrão *Observer*

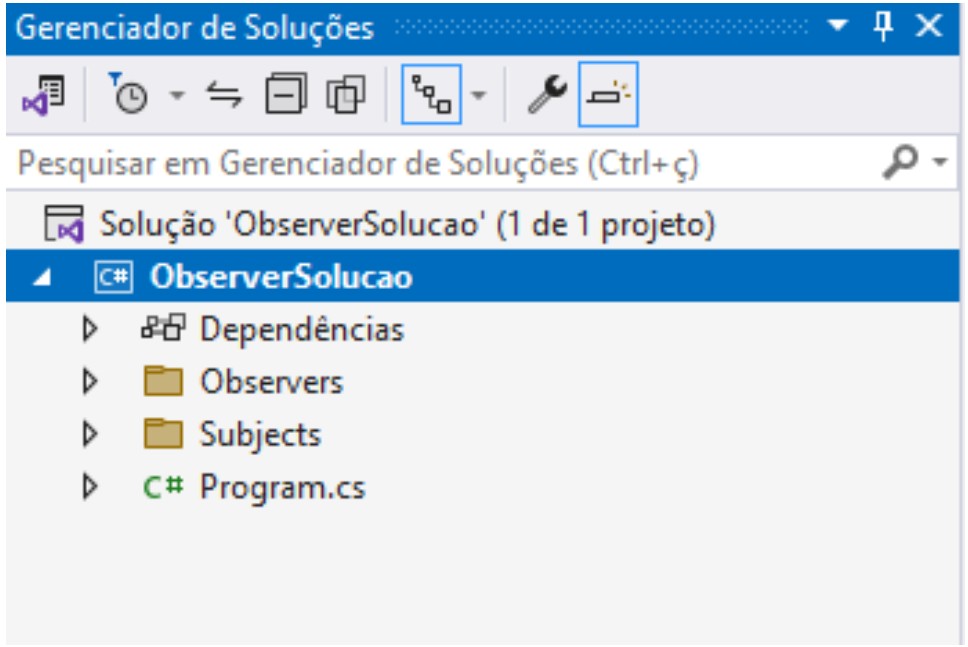
- **Solução**

- 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.

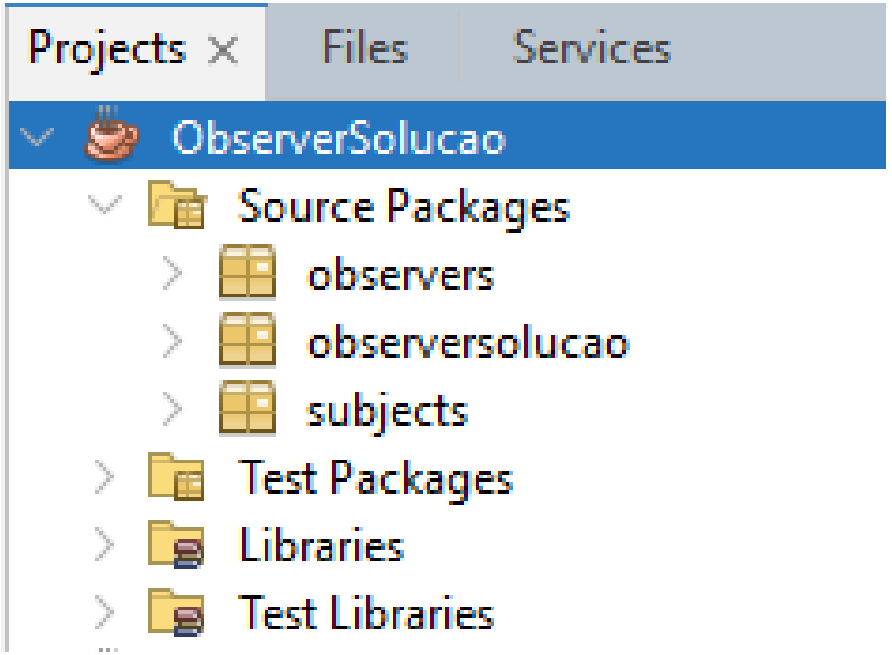


Esquema de notificações do padrão *Observer*

- **C#**
  - Crie as estrutura de pastas



- **Java**
  - Crie as estrutura de pacotes



- **C#**
  - Crie a interface Observer no pacote observers

```
namespace ObserverSolucao.Observers
{
    13 referências
    public interface Observer
    {
        5 referências
        public void update(string mensagem);

        5 referências
        public string getNome();

        5 referências
        public string getEmail();

    }
}
```

- **Java**
  - Crie a interface Observer no pacote observers

```
...4 lines
package observers;

/**...4 lines */
public interface Observer {

    public void update(String mensagem);

    public String getNome();

    public String getEmail();

}
```

- Apenas o método update() é parte do padrão Observer, os métodos getNome e getEmail fazem parte do contexto do problema.

- **C#**
- Crie a interface Subject no pacote subjects

```
namespace ObserverSolucao.Subjects
{
    1 referência
    public interface Subject
    {
        7 referências
        public void registerObserver(Observer observer);
        2 referências
        public void removeObserver(Observer observer);
        2 referências
        public void notifyObserver();
        4 referências
        public void addMessage(string message);
    }
}
```

- **Java**
- Crie a interface Subject no pacote subjects

```
package subjects;

import observers.Observer;

/**...4 lines */
public interface Subject {

    public void registerObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObserver();
    public void addMessage(String message);

}
```

- **C#**
- Crie a classe Email no pacote subjects

```
namespace ObserverSolucao.Subjects
{
    4 referências
    public class Email
    {
        4 referências
        public static void enviarEmail(Observer observer, string mensagem)
        {
            Console.WriteLine("-----------------------------------------------------------------------------");
            Console.WriteLine("Email enviado para: " + observer.getNome() + " - " + observer.getEmail());
            Console.WriteLine("Mensagem :" + mensagem);
        }
    }
}
```

- **Java**
- Crie a classe Email no pacote subjects

```
package subjects;

import observers.Observer;

/**...4 lines */
public class Email {

    public static void enviarEmail(Observer observer, String mensagem){
        System.out.println("-----------------------------------------------------------------------------");
        System.out.println("Email enviado para: " + observer.getNome() + " - " + observer.getEmail());
        System.out.println("Mensagem :" + mensagem);
    }

}
```

- **Solução**
- 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.

- **C#**
  - Crie a classe Newsletter no pacote subjects

```
namespace ObserverSolucao.Subjects
{
    3 referências
    public class Newsletter : Subject {
        private List<Observer>? observers;
        private List<string>? emails;

        1 referência
        public Newsletter()
        {
            this.observers = new List<Observer>();
            this.emails = new List<string>();
        }
        4 referências
        public void addMessage(string message)
        {
            this.emails.Add(message);
            this.notifyObserver();
        }
        2 referências
        public void notifyObserver()
        {
            foreach (Observer observer in observers)
            {
                observer.update(emails.Last());
            }
        }
        7 referências
        public void registerObserver(Observer observer)
        {
            this.observers.Add(observer);
        }
        2 referências
        public void removeObserver(Observer observer)
        {
            for (int i = 0; i < observers.Count; i++)
            {
                Observer observerReg = observers[i];
                if (observerReg.Equals(observer))
                    observers.Remove(observerReg);
            }
        }
    }
}
```

- **Java**
  - Crie a classe Newsletter no pacote subjects

```
package subjects;

import java.util.ArrayList;
import java.util.List;
import observers.Observer;

/**...4 lines */
public class Newsletter implements Subject {

    private List<Observer> observers;
    private List<String> emails;

    public Newsletter() {
        this.observers = new ArrayList<>();
        this.emails = new ArrayList<>();
    }
    @Override
    public void registerObserver(Observer observer) {
        this.observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        if (!observers.isEmpty())
            observers.removeIf(x -> x.equals(observer));
    }
    @Override
    public void notifyObserver() {
        for(Observer observer : observers){
            if(!emails.isEmpty())
                observer.update(emails.get(emails.size()-1));
        }
    }
    @Override
    public void addMessage(String message) {
        this.emails.add(message);
        this.notifyObserver();
    }

}
```

- **Solução**
- 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.

Padrões de Projetos Comportamentais – Observer

C# -> Crie as classes de observers (Cliente, Fornecedor, Funcionario e Parceiro

```
namespace ObserverSolucao.Observers
{
    4 referências
    public class Cliente : Observer
    {
        private string nome;
        private string email;

        1 referência
        public Cliente(string nome, string email)
        {
            this.nome = nome;
            this.email = email;
        }
        0 referências
        public override bool Equals(object? obj)
        {
            return obj is Cliente cliente &&
                   nome == cliente.nome &&
                   email == cliente.email;
        }
        2 referências
        public string getEmail()
        {
            return this.email;
        }
        2 referências
        public string getNome()
        {
            return this.nome;
        }
        2 referências
        public void update(string mensagem)
        {
```

```
namespace ObserverSolucao.Observers
{
    4 referências
    public class Fornecedor : Observer
    {
        private string nome;
        private string email;

        1 referência
        public Fornecedor(string nome, string email)
        {
            this.nome = nome;
            this.email = email;
        }
        0 referências
        public override bool Equals(object? obj)
        {
            return obj is Fornecedor fornecedor &&
                   nome == fornecedor.nome &&
                   email == fornecedor.email;
        }
        2 referências
        public string getEmail()
        {
            return this.email;
        }
        2 referências
        public string getNome()
        {
            return this.nome;
        }
        2 referências
        public void update(string mensagem)
        {
            Email.enviarEmail(this, mensagem);
        }
    }
```

Padrões de Projetos Comportamentais – Observer

C# -> Crie as classes de observers (Cliente, Fornecedor, Funcionario e Parceiro

```
namespace ObserverSolucao.Observers
{
    6 referências
    internal class Funcionario : Observer
    {
        private string nome;
        private string email;

        2 referências
        public Funcionario(string nome, string email)
        {
            this.nome = nome;
            this.email = email;
        }
        0 referências
        public override bool Equals(object? obj)
        {
            return obj is Funcionario funcionario &&
                   nome == funcionario.nome &&
                   email == funcionario.email;
        }
        2 referências
        public string getEmail()
        {
            return this.email;
        }
        2 referências
        public string getNome()
        {
            return this.nome;
        }
        2 referências
        public void update(string mensagem)
        {
            Email.enviarEmail(this, mensagem);
        }
    }
}
```

```
namespace ObserverSolucao.Observers
{
    4 referências
    public class Parceiro : Observer
    {
        private string nome;
        private string email;

        1 referência
        public Parceiro(string nome, string email)
        {
            this.nome = nome;
            this.email = email;
        }
        0 referências
        public override bool Equals(object? obj)
        {
            return obj is Parceiro parceiro &&
                   nome == parceiro.nome &&
                   email == parceiro.email;
        }
        2 referências
        public string getEmail()
        {
            return this.email;
        }
        2 referências
        public string getNome()
        {
            return this.nome;
        }
        2 referências
        public void update(string mensagem)
        {
            Email.enviarEmail(this, mensagem);
        }
    }
}
```

Padrões de Projetos Comportamentais – Observer

JAVA -> Crie as classes de observers (Cliente, Fornecedor, Funcionario e Parceiro

```java
package observers;

import java.util.Objects;
import subjects.Email;

/**...4 lines */
public class Cliente implements Observer {

    private String nome;
    private String email;

    public Cliente(String nome, String email) {
        this.nome = nome;
        this.email = email;
    }
    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...16 lin

    @Override
    public void update(String mensagem) {
        Email.enviarEmail(this, mensagem);
    }
    @Override
    public String getNome() {
        return this.nome;
    }
    @Override
    public String getEmail() {
        return this.email;
    }
}
```

```java
package observers;

import java.util.Objects;
import subjects.Email;

/**...4 lines */
public class Fornecedor implements Observer {

    private String nome;
    private String email;

    public Fornecedor(String nome, String email) {
        this.nome = nome;
        this.email = email;
    }
    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...16 lines

    @Override
    public void update(String mensagem) {
        Email.enviarEmail(this, mensagem);
    }
    @Override
    public String getNome() {
        return this.nome;
    }
    @Override
    public String getEmail() {
        return this.email;
    }
}
```

Padrões de Projetos Comportamentais – Observer

JAVA -> Crie as classes de observers (Cliente, Fornecedor, Funcionario e Parceiro

```java
package observers;

import java.util.Objects;
import subjects.Email;

/**...4 lines */
public class Funcionario implements Observer {

    private String nome;
    private String email;

    public Funcionario(String nome, String email) {
        this.nome = nome;
        this.email = email;
    }

    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...16 lines

    @Override
    public void update(String mensagem) {
        Email.enviarEmail(this, mensagem);
    }
    @Override
    public String getNome() {
        return this.nome;
    }
    @Override
    public String getEmail() {
        return this.email;
    }
}
```

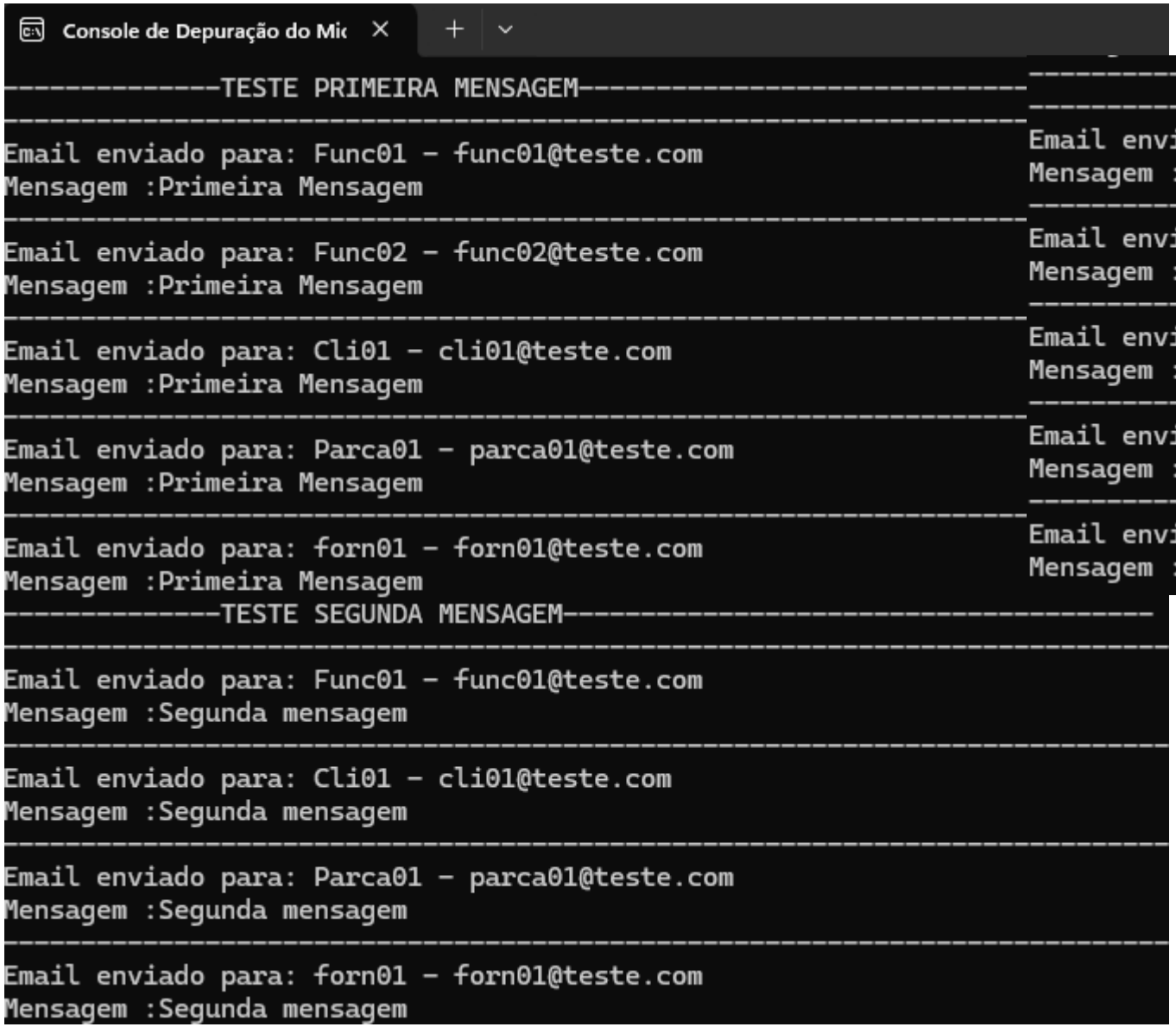
```java
package observers;

import java.util.Objects;
import subjects.Email;

/**...4 lines */
public class Parceiro implements Observer {

    private String nome;
    private String email;

    public Parceiro(String nome, String email) {
        this.nome = nome;
        this.email = email;
    }

    @Override
    public int hashCode() {...6 lines }

    @Override
    public boolean equals(Object obj) {...16 lin

    @Override
    public void update(String mensagem) {
        Email.enviarEmail(this, mensagem);
    }
    @Override
    public String getNome() {
        return this.nome;
    }
    @Override
    public String getEmail() {
        return this.email;
    }
}
```

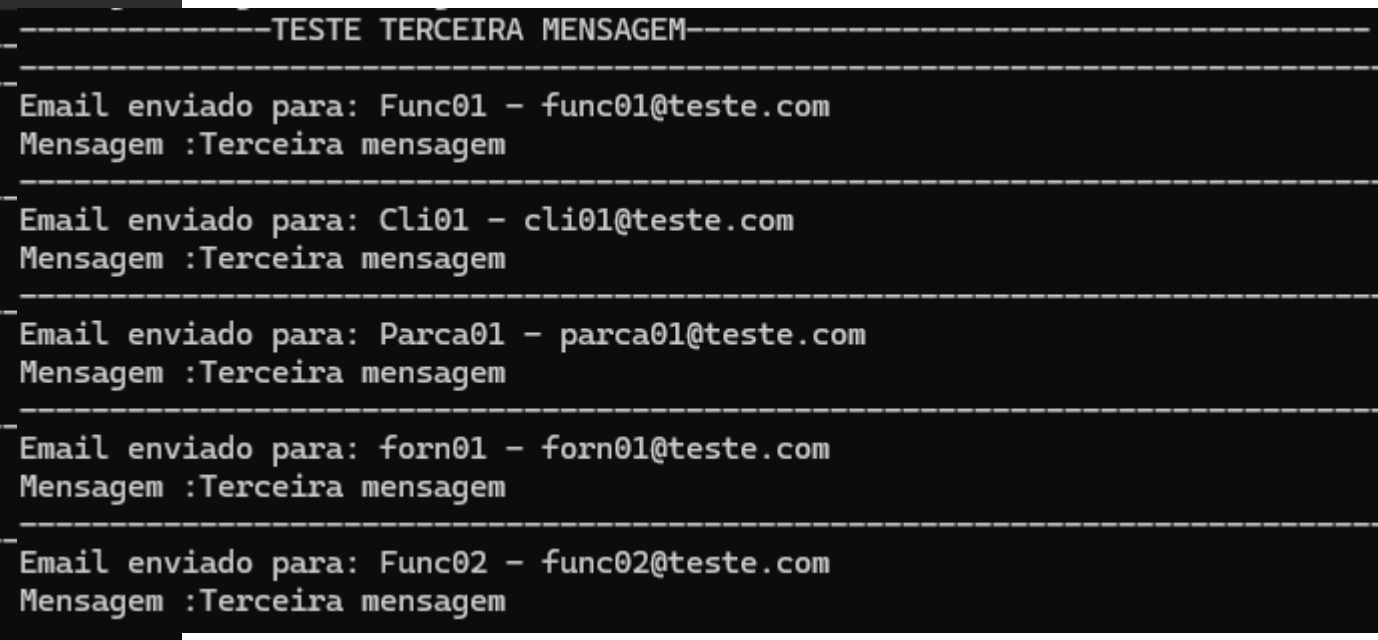
Padrões de Projetos Comportamentais – Observer

- **JAVA - Crie código para teste no main**

```java
package observersolucao;

import observers.Cliente;
import observers.Fornecedor;
import observers.Funcionario;
import observers.Parceiro;
import subjects.Newsletter;

/**...4 lines */
public class ObserverSolucao {

    /**...3 lines */
    public static void main(String[] args) {
        //criação Newsletter (subject)
        Newsletter newsletter = new Newsletter();
        //criação funcionarios
        Funcionario funcionario1 = new Funcionario("Func01", "func01@teste.com");
        newsletter.registerObserver(funcionario1);
        Funcionario funcionario2 = new Funcionario("Func02", "func02@teste.com");
        newsletter.registerObserver(funcionario2);
        //cricao cliente
        Cliente cliente = new Cliente("Cli01", "cli01@teste.com");
        newsletter.registerObserver(cliente);
        //criacao parceiro
        Parceiro parceiro = new Parceiro("Parca01", "parca01@teste.com");
        newsletter.registerObserver(parceiro);
        //criacao fornecedor
        Fornecedor fornecedor = new Fornecedor("forn01", "forn01@teste.com");
        newsletter.registerObserver(fornecedor);

        //envio primeira mensagem
        System.out.println("--------------TESTE PRIMEIRA MENSAGEM-------------------------------------");
        newsletter.addMessage("Primeira Mensagem");

        newsletter.removeObserver(funcionario2);
        System.out.println("--------------TESTE SEGUNDA MENSAGEM-------------------------------------");
        newsletter.addMessage("Segunda mensagem");

        newsletter.registerObserver(funcionario2);
        System.out.println("--------------TESTE TERCEIRA MENSAGEM-------------------------------------");
        newsletter.addMessage("Terceira mensagem");
    }
}
```

- **Resultado**


Console de Depuração do Mic
----------TESTE PRIMEIRA MENSAGEM----------
Email enviado para: Func01 - func01@teste.com
Mensagem :Primeira Mensagem
Email enviado para: Func02 - func02@teste.com
Mensagem :Primeira Mensagem
Email enviado para: Cli01 - cli01@teste.com
Mensagem :Primeira Mensagem
Email enviado para: Parca01 - parca01@teste.com
Mensagem :Primeira Mensagem
Email enviado para: forn01 - forn01@teste.com
Mensagem :Primeira Mensagem
----------TESTE SEGUNDA MENSAGEM----------
Email enviado para: Func01 - func01@teste.com
Mensagem :Segunda mensagem
Email enviado para: Cli01 - cli01@teste.com
Mensagem :Segunda mensagem
Email enviado para: Parca01 - parca01@teste.com
Mensagem :Segunda mensagem
Email enviado para: forn01 - forn01@teste.com
Mensagem :Segunda mensagem


----------TESTE TERCEIRA MENSAGEM----------
Email enviado para: Func01 - func01@teste.com
Mensagem :Terceira mensagem
Email enviado para: Cli01 - cli01@teste.com
Mensagem :Terceira mensagem
Email enviado para: Parca01 - parca01@teste.com
Mensagem :Terceira mensagem
Email enviado para: forn01 - forn01@teste.com
Mensagem :Terceira mensagem
Email enviado para: Func02 - func02@teste.com
Mensagem :Terceira mensagem

- Devido a utilização do padrão Observer o código passa a obedecer alguns bons princípios de programação orientada a objetos:
  1. **Programe para abastrações:** Newsletter (subject) e Cliente, Fornecedor, Parceiro e Funcionario (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. **Dê prioridade à composição em relação a herança:** o padrão Observer utiliza a composição para compor, em tempo de execução, um subject com qualquer número de observers.

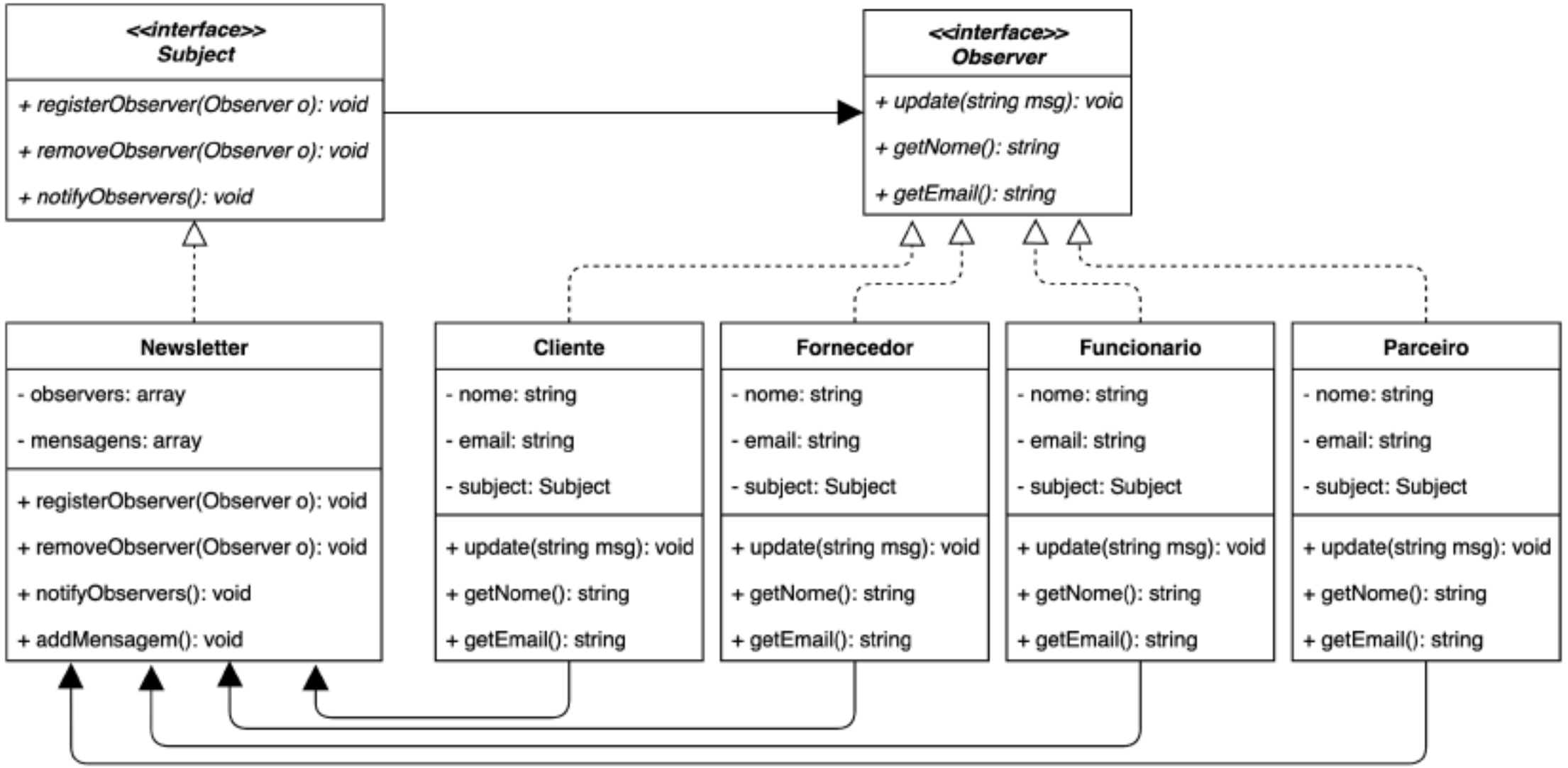


Diagrama de Classes do Exemplo

- Aplicabilidade:

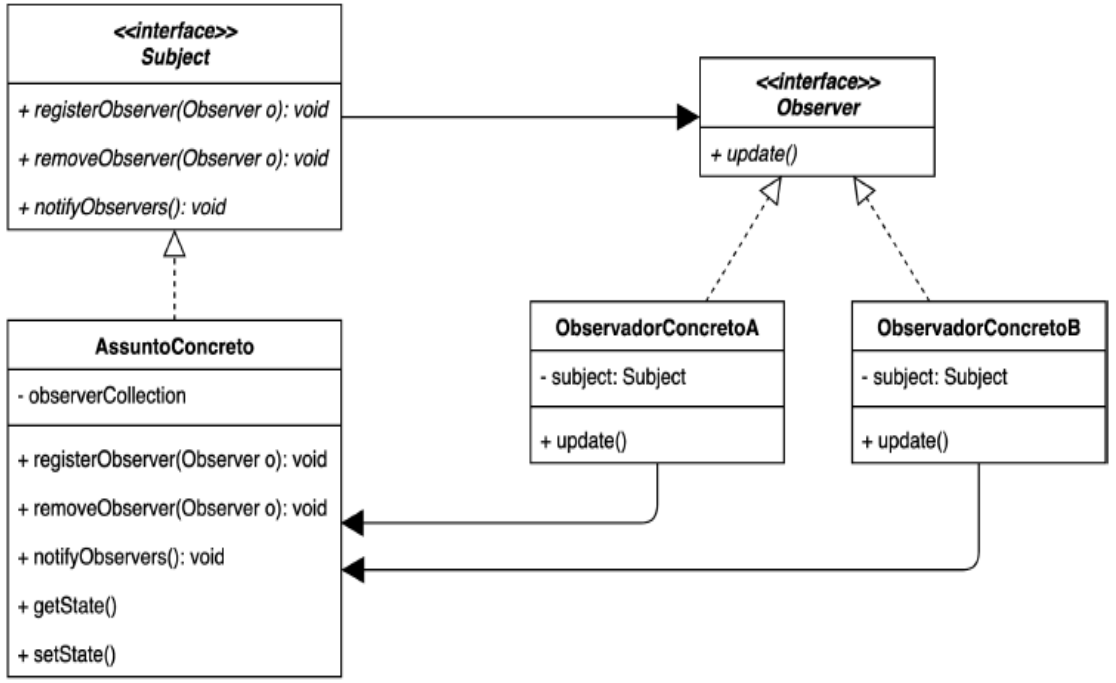
1. Quando uma abstração tem dois aspectos, um depende do outro, e é necessário que eles possam variar e serem reutilizados independentemente.

2. Quando uma alteração em um objeto requer a alteração de outros, e não se conhece quantos objetos precisam ser alterados.

3. Quando um objeto deve ser capaz de notificar outros objetos sem os conhecer, ou seja, tais objetos não podem ser fortemente acoplados.

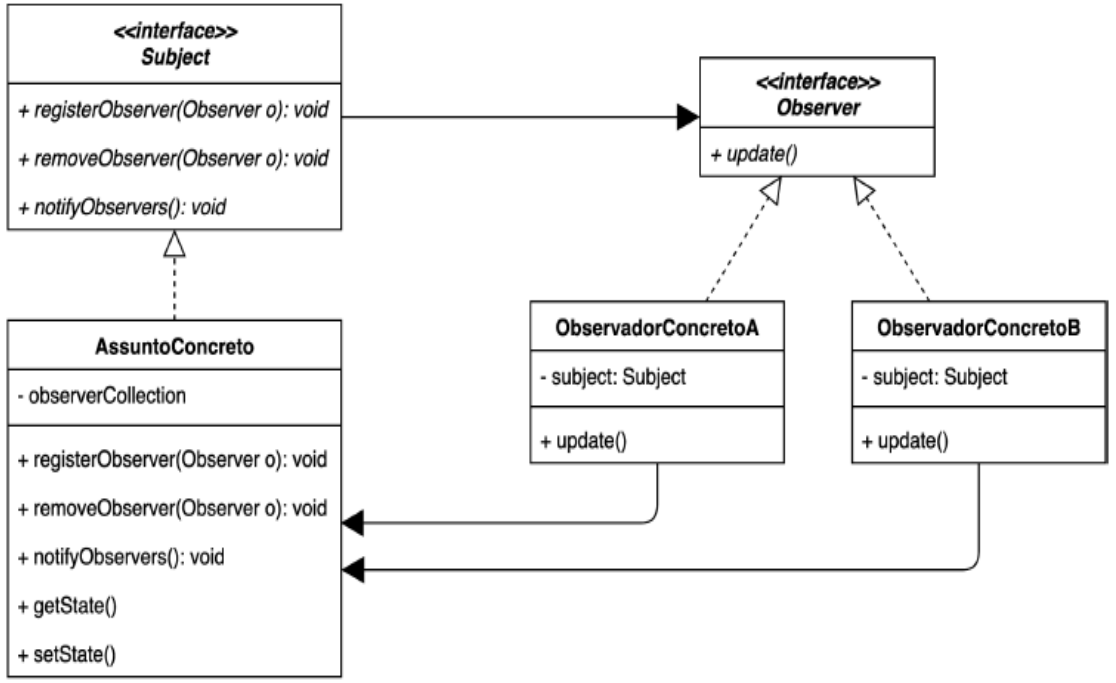
- Componentes:

  - **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 notifyOberservers() que é utilizados para atualizar todos os observadores atuais sempre o que o estado do AssuntoConcreto é alterado. Também pode ter métodos para definir e obter seu estado.



- Componentes:

  - **ObservadoresConcretos:** podem ser qualquer classe que implemente a interface Observer. Cada observer 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.



- 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.

- Consequências:

  - 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.

- Consequências:
  - 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.