IPS Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

Programação Avançada 2021-22

# Padrão de Desenho Observer

Bruno Silva, Patrícia Macedo

# Sumário 📝

- Padrão **Observer**
  - Enquadramento
  - Problema
  - Solução Proposta (pelo padrão)
  - Exemplo de Aplicação
    - Exercícios
  - Nota final
    - Java - *Observer* "descontinuado"

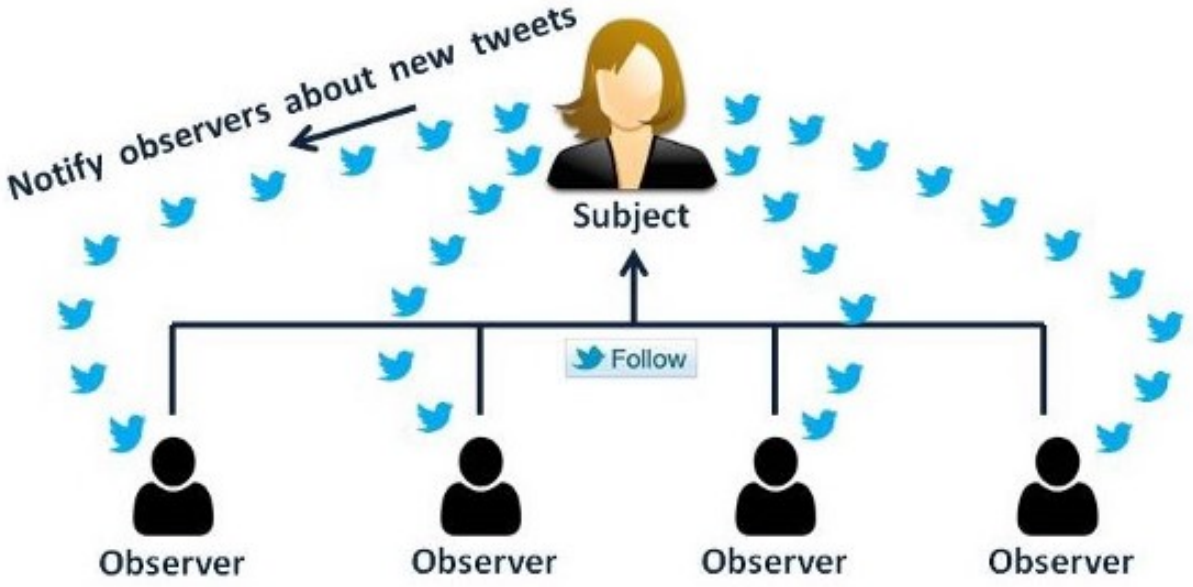
# Enquadramento 

Existem várias situações em que:

- Pretende-se assegurar que, quando um objecto muda de estado, um número de objectos dependentes é notificado automaticamente.
    - i.e., um objecto pretende notificar um conjunto de outros objetos.
- O que cada objeto faz sobre a notificação, depende do problema.

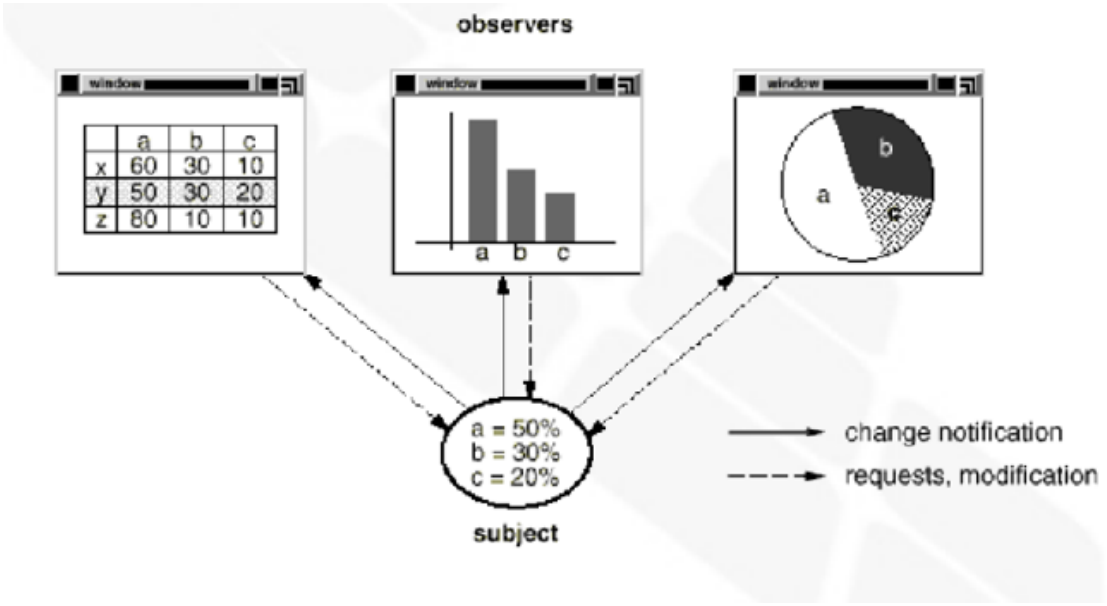
# Motivação 🤔

E.g., notificar um conjunto de susbcritores de uma plataforma que uma nova versão foi lançada.

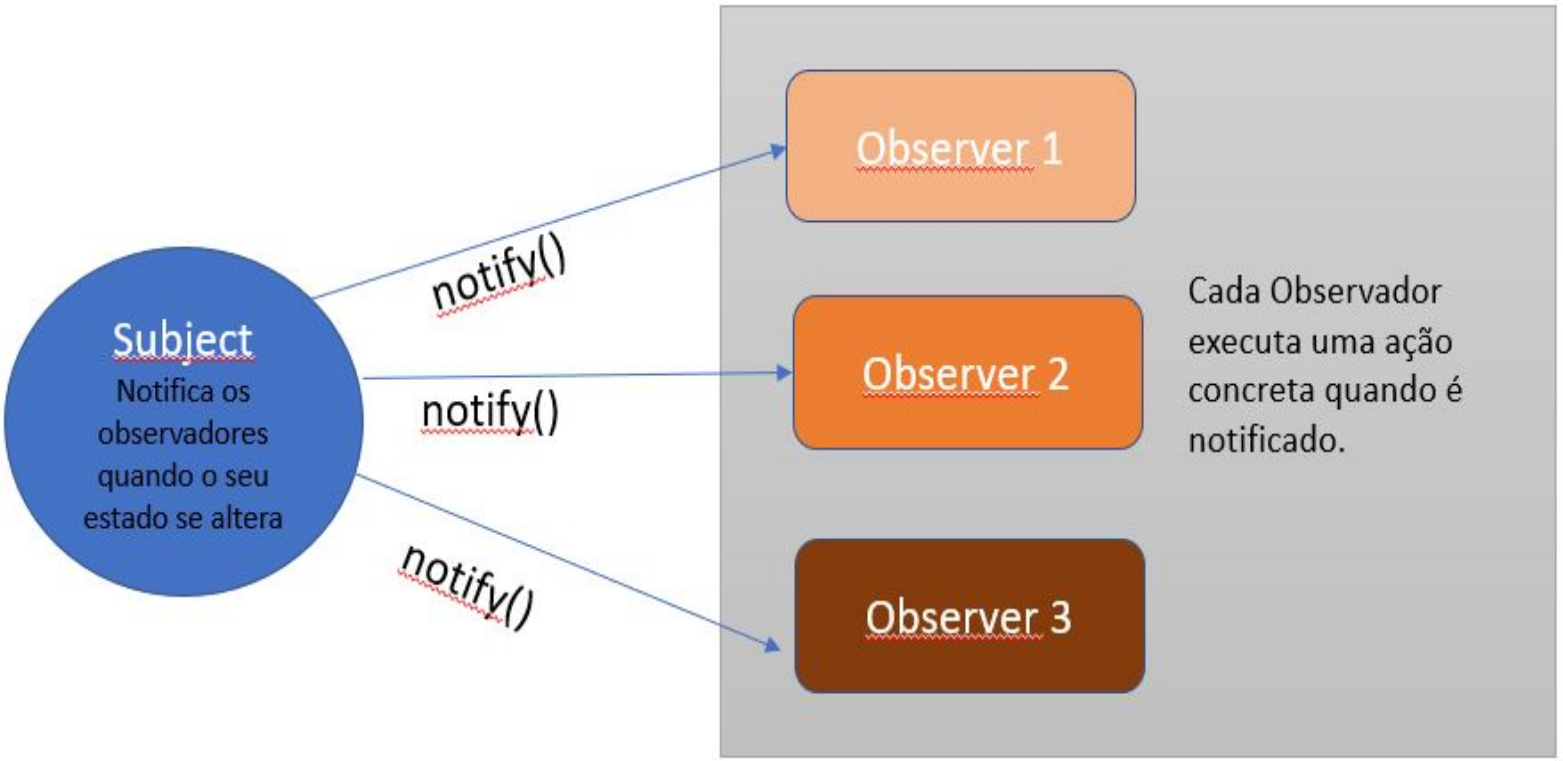


# Motivação 🤔

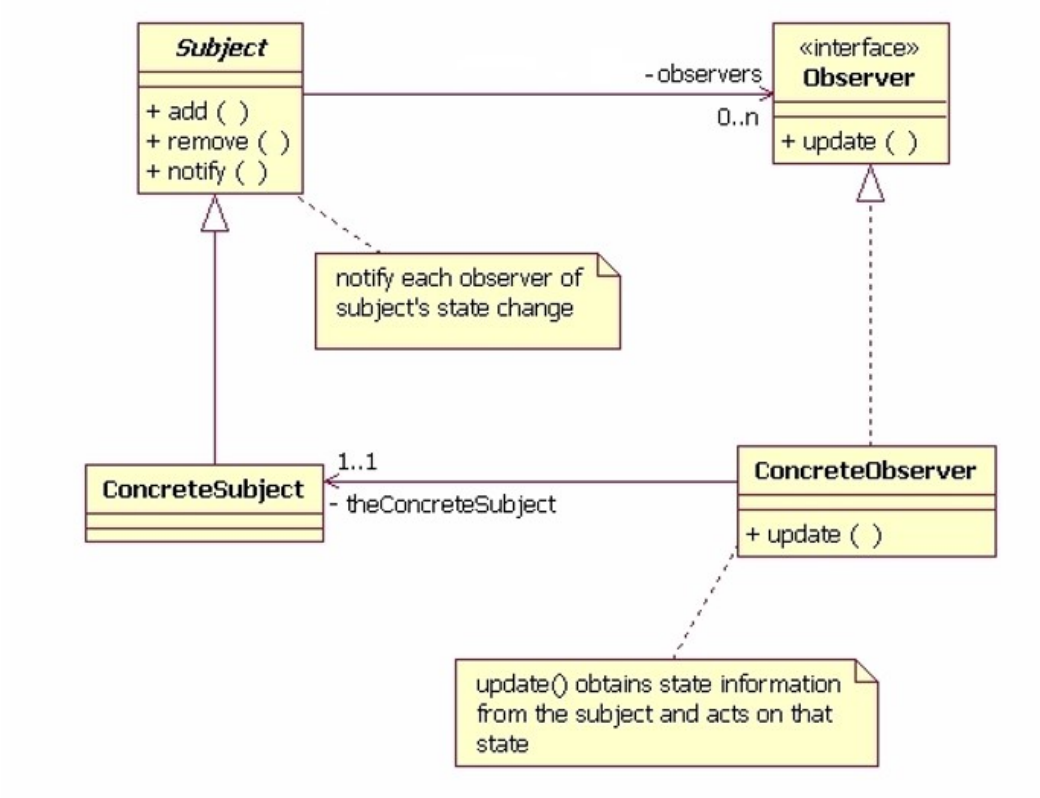
E.g., atualizar a visualização dos dados automaticamente cada vez que os dados são alterados.



# Solução Proposta | Padrão Observer



# Solução Proposta | Padrão Observer



# Participantes | Padrão Observer

- **Subject**
  - Classe *abstrata* que contém funcionalidade para adicionar/remover instâncias de `Observer`; Portanto:
    - Um *subject* conhece os seus *observadores*;
  - Sabe notificar os *observadores* através do método `notify()`
- **ConcreteSubject**
  - "Extende" de `Subject`;
  - Adiciona estado/comportamento adicional ao seu propósito;
  - Pode enviar notificação aos observadores, quando o seu estado se altera.

# Participantes | Padrão Observer

- **Observer**
  - Define uma *interface* que "obriga" à implementação do método `update()`; este será invocado pelo **ConcreteSubject** e representa a "notificação".
- **ConcreteObserver**
  - Implementa a interface **Observer**, para que receba as "notificações";
  - (opcional) Mantém uma referência para o objecto **ConcreteSubject**.

# Nota de implementação | Padrão Observer

- Um dos problemas que levanta a utilização do padrão Observer, é o facto de ele definir uma classe abstrata Subject em vez de uma interface.
- Como não existe em Java herança múltipla isso dificulta quando queremos tornar uma subclasse (de outra hierarquia) "observable".
- Por esta razão vamos implementar uma variante do padrão Observer onde é acrescentado uma interface Observable.

# Interface `Observable`

```
public interface Observable {
/**
*  Attach  observers to the subject.
* @param observers  to be attached
*/
public void addObservers(Observer... observers);
/**
*  Attach  observers to the subject.
* @param observer to be removed
*/
public void removeObservers(Observer observer);

/**
* notify all observer
* @param object, argument of update method
*/
public void notifyObservers(Object object);
}
```

# Classe *abstrata* `Subject`

```
public abstract class Subject implements Observable{
    private List<Observer> observerList;

    public Subject() {
        this.observerList = new ArrayList<>();
    }

    @Override
    public void addObservers(Observer... observers) {
        for (Observer obs : observers) {
            if (!observerList.contains(obs))
                this.observerList.add(obs);
        }
    }

    @Override
    public void removeObservers(Observer observer) {
        this.observerList.remove(observer);
    }

    @Override
    public void notifyObservers(Object obj) {
        for (Observer observer : this.observerList)
            observer.update(obj);

    }
}
```

# Interface `Observer`

```
public interface Observer {
    /**
    * When a observer is notified execute this method
    * @param obj - argument of the method
    */
    void update(Object obj);
}
```

# Exemplo de aplicação 🌎

- Temos uma classe típica que controla um "carrinho de compras" `ShoppingCart`, com as operações de adicionar e remover produtos.
- Com vista a desacoplar o modelo (ShoppingCart) das vistas sobre o mesmo, aplicou-se o padrão Observer.
  - **ConcreteSubject**: `ShoppingCart`
  - **ConcreteObserver**: `ShoppingCartCostView`

  💡 Cada vez que se faz uma alteração ao modelo (i.e., adiciona ou remove produtos), o valor total do carrinho de compras é atualizado e mostrado.

# Classe `ShoppingCart`

```
public class ShoppingCart extends Subject {

    private String name;
    private List<Product> products;

    public ShoppingCart(String name) {
        this.name = name;
        products = new ArrayList<>();
    }

    public void addProduct(Product p) {
        products.add(p);
        notifyObservers(this);
    }

    public void removerProduct(Product p) {
        products.remove(p);
        this.notifyObservers(this);
    }
```

# Classe `ShoppingCartCostView`

```
public class ShoppingCartTotalCostView implements Observer {

    @Override
    public void update(Object arg) {
        /* Invocado quando "notificado" */
        if(arg instanceof ShoppingCart) {
            ShoppingCart cart = (ShoppingCart)arg;
            String name = cart.getName();
            System.out.printf("(%s) total cost: %.2f € \n",
                 name,
                 cart.getTotal());
        }
    }

}
```

# *Context/Client*

É necessário adicionar explicitamente o "observador" ao "subject":

```
List<Product> productList= generateProductList();
ShoppingCart cart1 = new ShoppingCart("Bruno's Cart");
ShoppingCartTotalCostView costView = new ShoppingCartTotalCostView();

// add Observer to the Subject
cart1.addObservers(costView);

cart1.addProduct(productList.get(0));
cart1.addProduct(productList.get(1));
cart1.addProduct(productList.get(5));
cart1.removerProduct(productList.get(0));
```

Output:

```
(Bruno's Cart) total cost: 30.00 €
(Bruno's Cart) total cost: 380.00 €
(Bruno's Cart) total cost: 680.00 €
(Bruno's Cart) total cost: 650.00 €
```

# Exercícios ⌨️

Repositório de apoio à aula no GitHub:
https://github.com/estsetubal-pa-geral/ObserverPatternJava

1. Teste o programa fornecido e analise cuidadosamente as classes fornecidas;
2. Adicione uma nova classe que assume o papel de `ConcreteObserver` denominada `ShoppingCartListView` que tem como objetivo imprimir a lista de compras (ordenada por *id* do produto) com o seguinte formato:

   ```
   <shopping cart name>
   <ordem>: <nome> - <cost> euros
   ```

3. Faça as modificações necessárias no `main`, para adicionar ao `cart1` este novo *observador*.

# Exercícios ⌨️

4. Adicione uma nova classe que assume o papel de `ConcreteObserver` denominada `ShoppingCartAlert` - possui o atributo `maxValue` inicializado através de argumento do construtor. Cada vez que este observador é notificado, verifica se o último produto adicionado tem um valor superior ao `maxValue` e imprime uma mensagem com a seguinte configuração:

```
"ALERT!!! - The product <productName> has exceeded
    the maximum value <maxValue>
```

5. Faça as modificações necessárias no main, para adicionar ao `cart2` este novo *observador*.

# Nota final ⚠️
# Java - *Observer* "descontinuado"

O JAVA descontinuou a classe `Observable` e a Interface `Observer`.

- Em alternativa para resolver o problema que o Padrão Observer responde, propõe-se o uso do mecanismo de Listeners e mais especificamente `PropertyChangeListener`.
  - Leia mais em: https://www.baeldung.com/java-observer-pattern
- Em JavaFX o padrão Observer continua a ser implementado através das `ObservableList` e `ObservableMap`
  - Leia mais em: https://docs.oracle.com/javafx/2/collections/jfxpub-collections.htm

# Referências web 📖

- https://www.journaldev.com/1739/observer-design-pattern-in-java
- http://www.javaworld.com/jw-09-1998/jw-09-techniques.html
- https://refactoring.guru/design-patterns/observer