Behavioral - Observer Design Pattern (İkinci örneğe mutlaka bakılmalı)

One To Many ilişkilerde sıklıkla kullanılır.Bir nesnenin durumu değiştiğinde subscribe olan herkese bu bilgi gönderilir.Bir alışveriş sitesinde indirim olduğunu düşünelim, siteye tüm üye olan herkese bu bilgi gönderilir.Yada şöyle düşünelim yeni bir ürün çıkacak ve siz bu ürünü sürekli gidip mağazaya sormak mı istersiniz yoksa mağazanın size evet ürün geldi şeklinde bir mail atmasını mı istersiniz?

İlk olarak state class'ımızı create ediyorum;

```
//state object
//Bu değişmez bir class olmalıdır yanlışlıkla içeriği değiştirilememelidir
class Message {
   final String messageContent;
   public Message(String messageContent) {
      this.messageContent = messageContent;
   }
   public String getMessageContent() {
      return messageContent;
   }
}
```

Observer Interface;

```
//Observer
interface Observer {
   void update(Message content);
}
```

- Observer interface'i içerisinde update methodu yer alır.Bu interface'i inherit eden her subscriber update methodunu override etmek zorundadır.Update methodu Subject üzerinde bir değişiklik olduğu anda tüm inherit etmiş objelere bunu bildirmek için kullanılır

- Observer'ı inherit eden class'lar.Yani updateleri alacak olan class'lar

```
//Concrete Observer
class MessageSubscriberOne implements Observer {
   @Override
   public void update(Message content) {
      System.out.println("Message subscriber one : " + content.getMessageContent());
   }
}
class MessageSubscriberTwo implements Observer {
   @Override
   public void update(Message content) {
      System.out.println("Message subscriber two : " + content.getMessageContent());
   }
}
class MessageSubscriberThree implements Observer{
   @Override
   public void update(Message content) {
      System.out.println("Message subscriber three : " + content.getMessageContent());
   }
}
```

Subject

```
//Subject
interface Subject {
   void attach(Observer observer);
   void deAttach(Observer observer);
   void notifyUpdate(Message message);
}
```

Concrete Subject

```
//concrete subject
class MessagePublisher implements Subject {
   List<Observer> observerList = new ArrayList<>();
   @Override
   public void attach(Observer observer) {
      observerList.add(observer);
   }
   @Override
   public void deAttach(Observer observer) {
      observerList.remove(observer);
   }
   @Override
   public void notifyUpdate(Message message) {
      observerList.forEach(observer -> observer.update(message));
   }
}
```

Main class;

```
class Test{
   public static void main(String[] args) {
      MessageSubscriberOne msOne = new MessageSubscriberOne();
      MessageSubscriberTwo msTwo = new MessageSubscriberTwo();
      MessageSubscriberThree msThree = new MessageSubscriberThree();

      MessagePublisher publisher = new MessagePublisher();
      publisher.attach(msOne);
      publisher.attach(msTwo);
      publisher.attach(msThree);
      publisher.notifyUpdate(new Message("State changed"));
      publisher.deAttach(msTwo);
      publisher.notifyUpdate(new Message("State changed"));
   }
}
```

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
Message subscriber one : State changed
Message subscriber two : State changed
Message subscriber three : State changed
Message subscriber one : State changed
Message subscriber three : State changed
```

Behavioral - Observer Design Pattern (İkinci örnek)

Business Logic

```
class ProductManager {
   ArrayList<Observer> observerList = new ArrayList<>();
   public void UpdatePrice(){
      System.out.println("Product price changed");
      Notify();
   }
   void attach(Observer observer){
      observerList.add(observer);
   }
   void detach(Observer observer){
      observerList.remove(observer);
   }
   private void Notify(){
      for (Observer observer : observerList) {
         observer.Update();
      }
   }
}
```

Observer

```
abstract class Observer {
   abstract void Update();
}
class CustomerObserver extends Observer {
   @Override
   void Update() {
      System.out.println("Message to Customer : Product price changed");
   }
}
```

Main

```
class Test{
   public static void main(String[] args) {
      ProductManager pm = new ProductManager();
      pm.attach(new CustomerObserver());
      pm.UpdatePrice();
   }
}
```