Práctica 1
Patrones de Diseño

Ingeniería de Software Avanzada
Creado por: Francisco Rodríguez-Córdoba Lucena

Cuestiones

Q1: Consideremos los siguientes patrones de diseño: Adaptador, Decorador y Representante. Identifique las principales semejanzas y diferencias entre cada dos ellos (no es suficiente con definirlos, sino describir explícitamente similitudes y semejanzas concretas).

Para empezar podemos observar que estos tres patrones comparten el mismo tipo: Estructural. Esto quiere decir que buscan esxplicar como ensamblar objetos y clases en estructuras extendibles, que permiten añadir nuevas funcionalidades.

El patrón **Adaptador** modifica la interfaz de un objeto con el objetivo de que otro objeto pueda comprenderla, a diferencia del patrón **Decorador** el cuál mejora dicho objeto sin modificar la interfaz.

En cuanto al patrón **Decorador** y el patrón **Representante** podemos decir que su estructura es muy similar, los dos proponen que un objeto debe delegar parte de sus responsabilidades a otro.
Como diferencia, **Decorador** proporciona una interfaz mejorada mientras que **Representante** proporciona la misma interfaz.

Entre **Adaptador** y **Representante** como hemos dicho antes, el primero proporciona una interfaz diferente al objeto envuelto, mientras que el segundo proporciona la misma interfaz.

Q2: Consideremos los patrones de diseño de comportamiento Estrategia y Estado. Identifique las principales semejanzas y diferencias entre ellos.

La principal semejanza es que ambos son patrones de comportamiento.

Podemos considerar que el patrón **Estado** es una continuación del patrón **Estrategia**. Los dos patrones cambian el comportamiento del entorno delegando parte de las responsabilidades a otros objetos.

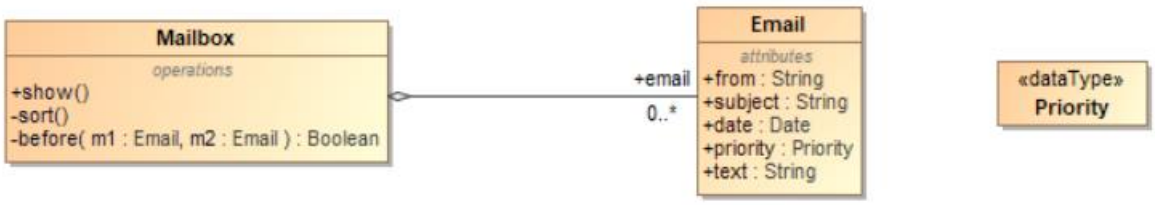
Se diferencian en que el patrón **Estrategia** hace dichos objetos independientes unos de otros, sin ninguna relación. En cambio, el patrón **Estado** no pone impedimentos a la hora de las dependencias.

Q3: Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos.

Ambos son patrones de comportamiento.

El patrón **Mediador** elimina de manera directa conexiones entre emisores y receptores, para que se conecten entre ellos a través de un objeto *mediador*. En cambio, el patrón **Observador** permite a los receptores decidir si recibir solicitudes o no.

Cliente de correo e-look



Para este tipo de ejercicio he elegido el patrón **Estrategia** ya que tomamos la clase *SortStrategy* como clase interfaz que hace algo específico pj el método *gBefore(Email m1, Email m2),*

```
public interface SortStrategy {
    boolean gBefore(Email e1,Email e2);

}
```

y extraemos ese método(el cuál devuelve **true** si el primer email es anterior al otro) para colocarlo en clases separadas que implementan dicha interfaz.

```java
public class sortByText implements sortStrategy{

    @Override
    public boolean gbefore(Email m1, Email m2) {
        // TODO Auto-generated method stub
        return false;
    }
```

```java
public class sortBySubject implements sortStrategy{

    @Override
    public boolean gbefore(Email m1, Email m2) {
        // TODO Auto-generated method stub
        return false;
    }
```

```java
public class sortByPriority implements sortStrategy{

    @Override
    public boolean gbefore(Email m1, Email m2) {
        // TODO Auto-generated method stub
        return false;
    }
```

```java
public class sortByFrom implements sortStrategy{

    @Override
    public boolean gbefore(Email m1, Email m2) {
        // TODO Auto-generated method stub
        return false;
    }
```

```java
public class sortByDate implements sortStrategy{

    @Override
    public boolean gbefore(Email m1, Email m2) {
        // TODO Auto-generated method stub
        return false;
    }
```

Esa sería la estructura del patrón **Estrategia** que hemos seguido para esta práctica, en resumen, con este patrón hemos conseguido realizar 5 tipos de ordenación de los e-mails.

Además en el caso de añadir un nuevo método de ordenación en un futuro, no sería necesario cambiar nada del código, si no, crear una nueva clase y que esta implemente los métodos de la interfaz.

Las clases principales son las de ***Email y MailBox***

```java
public class Email {
    public String from;
    public String subject;
    public LocalDate date;
    public Priority priority;
    public String text;

    public Email(String from,String subject,LocalDate date,Priority priority,String text) {
        this.from=from;
        this.subject=subject;
        this.date=date;
        this.priority=priority;
        this.text=text;
    }
}
```

```
public class Mailbox {
    private static sortStrategy strategy;
    public List<Email> email = new ArrayList<Email>();
    private void sort() {
         //for ( int i = 2; i <= email.size(), i++ )
         //for ( int j = email.size(); j >= i; j-- )
         //if ( before(email.at(j),email.at(j-1)) )
         // intercambiar los mensajes j y j-1
         //...
         }

    public void show() {


    }
    private boolean before(Email e1, Email e2) {
        return strategy.gbefore(e1, e2);
    }


}
```

En esta última clase se almacenan los e-mails, y se crean los métodos show() para visualizar los emails, y el sort() que los ordena, utilizando el método before(Email m1,Email m2)

URL Github:

https://github.com/Franrodriguezcordoba/e-look

En el siguiente repositorio he completado el algoritmo e-look:

https://github.com/Franrodriguezcordoba/e-lookComplete