Práctica 1- Patrones de diseño

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).

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

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

Q1

El patrón **Adaptador** consiste en la implementación de una clase que permita reutilizar clases antiguas adaptándolas al funcionamiento de nuevas clases implementadas.

El patrón **decorador** permite extender comportamientos de forma dinámica, es óptimo para convertir una clase previamente desarrollada en una con un comportamiento diferente sin la necesidad de crear una clase nueva o una herencia de esta.

El patrón **representante** proporciona un nivel adicional de indirección para proporcionar acceso distribuido, controlado o inteligente.

Los tres patrones de alguna forma hacen de mediadores entre dos clases u objetos diferentes, tanto en el patrón **adaptador** como en el patrón **decorador** se pretende obtener una nueva funcionalidad sobre una clase antigua. La diferencia principal entre ambos reside en la forma de hacerlo, mientras en el adaptador se encarga de traspasar las funcionalidades de una clase nueva a una clase antigua, el decorador simplemente extiende el comportamiento de la clase sobre la que se usa.

El patrón **representante** sigue en la línea de los dos patrones previamente explicados pues comunica una clase con otra utilizando el principio de delegación, pero a diferencia de sus comparados este no busca transformar directamente el comportamiento de ninguna clase.

Q2

Ambos buscan aportar diversas formas de funcionamiento a una clase, la diferencia entre ellos radica en el que el **patrón estado** otorga diferentes funcionalidades en función de las circunstancias en las que se encuentre una clase, en cambio, el **patrón estrategia** buscara el desarrollo de nuevas formas de trabajar, pero mediante la elección de estas.

Por ejemplificar esto, en el funcionamiento de un semáforo el **patrón estado** determinaría a que luz va a pasar el semáforo en función de la luz que ya este encendida y el tiempo que esta lleve en funcionamiento, el **patrón estrategia** en cambio permitirá alternar entre modo noche o modo día donde el semáforo, en modo noche, tarde dos minutos mas en ponerse en verde para peatones que en el modo día.

Q3

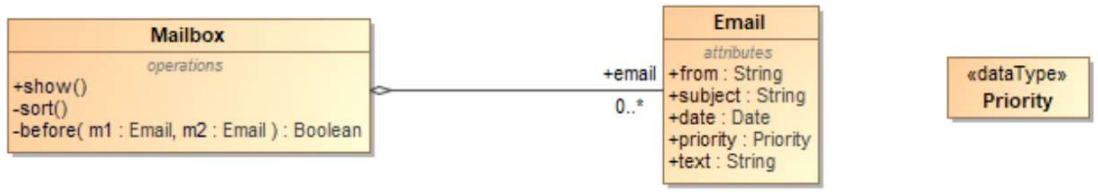
El **patrón mediador** se encarga de resolver el problema múltiples conexiones entre objetos de la misma clase, define un objeto encargado de encapsular como interactúan un conjunto de objetos, evitando que estos deban tener conocimiento entre ellos.

El **patrón observador** se utiliza cuando aparece la necesidad de notificar cambios de estado a algún objeto.

Ambos patrones se encargan de comunicar una serie de objetos entre ellos, la diferencia principal radica en que el patrón mediador comunica una serie de objetos haciendo de intermediario para el evitar el tedio de hacer que todos los objetos tengan una referencia a los demás, mientras que el patrón observador comunica los cambios producidos en un objeto a el resto de los que pertenezcan a esta "red de comunicación".

Práctica 1- Cliente de correo

En esta práctica se nos pide que se implemente de forma esquemática en java un cliente de correo, que almacene una serie de Emails y tenga la función "show" para mostrar los mismos y la función "sort" para ordenarlos siguiendo un criterio. El objetivo de la práctica es identificar un patrón de diseño que permita implementar y alternar nuevos criterios de ordenación para el método "sort"



Se ha decidido que el mejor patrón para esta circunstancia es el **patrón estrategia** pues mediante la implementación de una interfaz podremos desarrollar distintos funcionamientos para el método "before". A continuación, se ira explicando clase por clase el esquema desarrollado:

Clase Email

```java
import java.util.Date;

public class Email {
    private String from, subject, text;
    private Date date;
    private Priority priority;

    public Email(String from1, String subject1,String text1,Date date1,Priority priority1) {
        from=from1;
        subject=subject1;
        text=text1;
        date=date1;
        priority=priority1;

    }
    public String toString() {
        return "[From: "+this.from+", Subject: "+this.subject+", Text: "+this.text+", Date: "
                +this.date.toString()+", Priority: "+this.priority+"]";
    }
}
```

Objeto que posteriormente será almacenado en nuestro “Mailbox”, cuenta con un método “toString” para la visualización por pantalla de este. La clase “Priority” es un enum de java que consta de tres niveles de prioridad.

Interfaz IEstrategia

```java
public interface IEstrategia {
    boolean before(Email e1, Email e2);

}
```

Interfaz que define el método before, que posteriormente será utilizado en “sort” para la ordenación del Mailbox

Clase BeforeFrom

```java
public class beforeFrom implements IEstrategia {
    @Override
    public boolean before(Email e1, Email e2) {
        return true;
    }
}
```

Clase que implementa la interfaz y le da un funcionamiento siguiendo el criterio de ordenación del atributo “from” de la clase “Email”. (No ha sido desarrollado pues es meramente el esquema estructural del proyecto).

*Existirá una clase que implemente la interfaz IEstrategia por cada uno de los criterios que se quiera seguir, esto hace muy cómoda la implementación de nuevos criterios de ordenación o la modificación de estos. (No se muestran las clases que sigan otros criterios pues sin su implantación todos son idénticos exceptuando el nombre de la clase.

Clase Mailbox

```
import java.util.ArrayList;

public class Mailbox {
    ArrayList<Email> Mails;
    private IEstrategia Orden;

    public Mailbox(IEstrategia Orden1) {
        Mails = new ArrayList<Email>();
        Orden=Orden1;

    }

    public void setOrden(IEstrategia orden) {
        Orden = orden;
    }

    public void add(Email a1) {
        this.Mails.add(a1);
    }
    public void show() {
        this.sort();
        Iterator<Email> iterator = Mails.iterator();
        while(iterator.hasNext()){
            String elemento = iterator.next().toString();
            System.out.println(elemento);
        }
    }

    private void sort() {
         for ( int i = 2; i <= Mails.size()-1; i++ ) {
             for ( int j = Mails.size()-1; j >= i; j-- ) {
                 if ( Orden.before(Mails.get(j),Mails.get(j-1)) ) {
                     // interccambiar los mensajes j y j-1


                 }
             }
         }
    }

}
```

La clase Mailbox cuenta con un ArrayList donde ira almacenando los emails que le manden y un atributo orden, que determinara el tipo de “before” que usa y por tanto el criterio de ordenación que sigue.

Tiene un método setOrden que permitirá alternar entre las distintas estrategias que hayan sido implementadas, un método add para añadir Emails, y un método show que mostrará por pantalla todos los Emails después de haberlos ordenado usando el “sort”.

Sort usa el método buble para ordenar los correos según el criterio del método before de la interfaz, este método before será uno u otro dependiendo de a estrategia que tenga en el momento de hacer el show el Mailbox

Main Mailbox

```
import java.util.Date;

public class MainMailbox {

    public static void main(String[] args) {
        Mailbox mb=new Mailbox(new beforeFrom());
        Date fecha1 = new Date(116, 5, 1);
        Date fecha2 = new Date(116, 5, 2);
        Date fecha3 = new Date(116, 5, 3);
        Email a1=new Email("Pepe1","Saludo1","¿Que tal?1",fecha1,Priority.Prioridad1);
        Email a2=new Email("Pepe2","Saludo2","¿Que tal?2",fecha2,Priority.Prioridad2);
        Email a3=new Email("Pepe3","Saludo3","¿Que tal?3",fecha3,Priority.Prioridad3);
        mb.add(a1);
        mb.add(a2);
        mb.add(a3);
        mb.show();
        mb.setOrden(new beforeSubject());
    }

}
```

Clase en la que se prueba el funcionamiento de un Mailbox, se crea primeramente un MailBox que sigue la estrategia “beforeFrom”, para posteriormente añadirle 3 Emails y que sean mostrados ejecutando el método “show”. Posteriormente se cambia la estrategia de ordenación a otra para comprobar que esto no de fallo.

La salida por pantalla es:

```
[From: Pepe1, Subject: Saludo1, Text: ¿Que tal?1, Date: Wed Jun 01 00:00:00 CEST 2016, Priority: Prioridad1]
[From: Pepe2, Subject: Saludo2, Text: ¿Que tal?2, Date: Thu Jun 02 00:00:00 CEST 2016, Priority: Prioridad2]
[From: Pepe3, Subject: Saludo3, Text: ¿Que tal?3, Date: Fri Jun 03 00:00:00 CEST 2016, Priority: Prioridad3]
```

Conclusión

El patrón estrategia se adapta perfectamente a este esquema y permite la modificación y la implementación de los criterios de ordenación de una forma extremadamente cómoda en la que no es necesario modificar el código base.

Enlace del proyecto en GitHub:
https://github.com/alexpascualm/PruebaD.Pattern-Strategy