Patrones de diseño – IS Avanzada

Hugo Ávalos de Rorthais

https://github.com/hugoavalos01/Patrones1-elook.git

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).***

La semejanza principal es que los 3 son patrones estructurales y especifican la forma en que se relacionan unas clases con otras.

Por un lado, el patrón Decorador nos permite añadir funcionalidades extra a un objeto sin modificar su estructura mientras que el Adaptador crea una interfaz que permite que distintas clases e incompatibles, trabajen juntas para así usar sus funcionalidades. Ya, por último, el patrón Representante crea objetos para poder controlar el acceso al mismo.

En resumen, el Adaptador usa una interfaz diferente del objeto al que se refiere, el Representante la misma y el Decorador una mejorada.

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

Ambos patrones son patrones de comportamiento y controlan cómo interaccionan unas clases con otras. Además, tienen una estructura similar, una clase que controla el cliente, una interfaz que relaciona las clases y los métodos a los que se accede mediante dicha interfaz.

En cuanto a las diferencias, el patrón Estrategia encapsula algoritmos y permite variar dichos algoritmos haciéndolos independientes de los que usa el cliente, mientras que el Estado encapsula el estado del objeto y ayuda a la clase a exhibir diferentes comportamientos en un estado diferente.

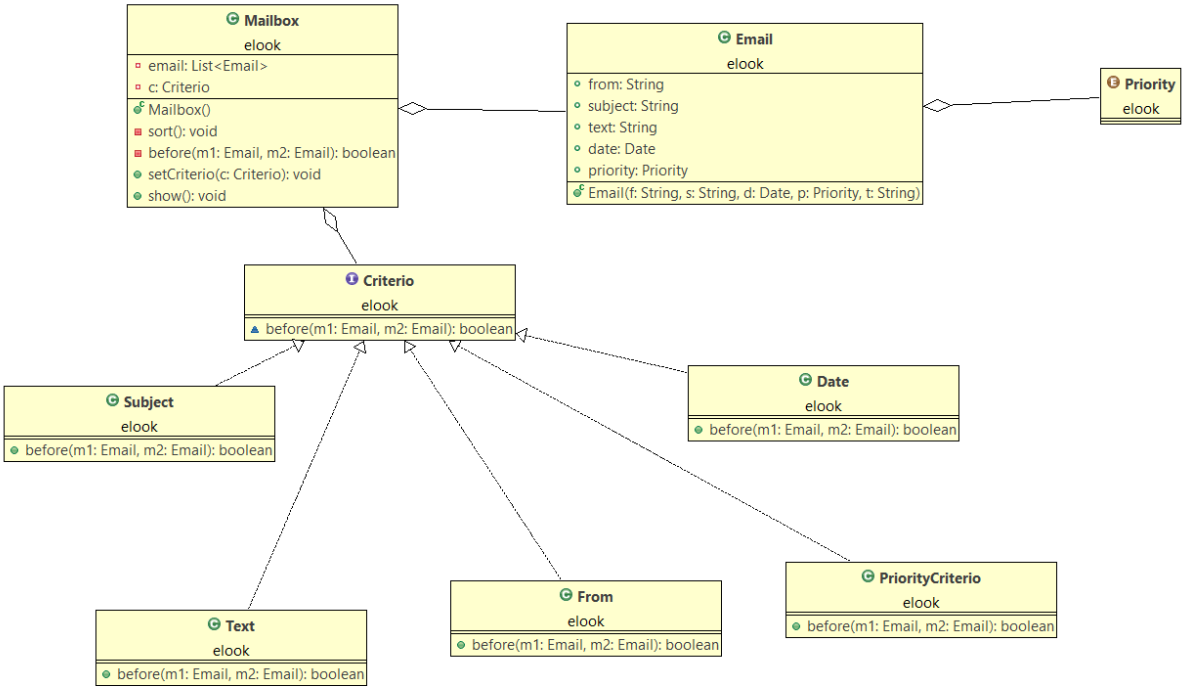
Q3. Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos.

Ambos patrones son patrones de comportamiento y son bastantes similares, estos conectan objetos e intercambian información entre ellos. En el patrón Mediador, un objeto encapsula cómo otro conjunto de objetos interactúan y se comunican entre sí. En el patrón Observador, los objetos son capaces de suscribirse a una serie de eventos que otro va a emitir, y estos serán avisados cuando el evento ocurra.

La principal diferencia es que el patrón Observador distribuye la comunicación introduciendo objetos emisores y otros receptores, es decir, es unidireccional. Mientras que, en el patrón Mediador, encapsula la comunicación entre objetos y esta es de manera bidireccional.

Cliente de correo e-Look

La mejor decisión sería aplicar el patrón Estrategia, de esta manera podemos definir los distintos criterios en clases distintas y si en el futuro queremos añadir un nuevo criterio, sólo tendríamos que crear una nueva clase e implementar el método before().



Implementación en Java (estructural)

Clase email: Simula un email con los siguientes atributos:

```java
public class Email {

    public String from, subject, text;
    public Date date;
    public Priority priority;

    public Email(String f, String s, Date d, Priority p, String t) {
        from = f;
        subject = s;
        date = d;
        priority = p;
        text = t;
    }

}
```

Clase Mailbox: Tenemos dos atributos, una lista “email” en el que se almacenan todos los emails recibidos, y un Criterio “c” el cual determinará en que orden se muestran los emails.

“Sort()” es el método que invoca a “before()”, y este último es el algoritmo que hará uso de la interfaz Criterio e invocará a otra clase para ordenar los mails de una forma u otra.

```
public class Mailbox {

    private List<Email> email;
    private Criterio c;

    public Mailbox() {
        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.get(j), email.get(j - 1))) {
                    // intercambiar los mensajes j y j-1
                }

    }

    private boolean before(Email m1, Email m2) {
        return c.before(m1, m2);
    }

    public void setCriterio(Criterio c) {
        this.c = c;
    }

    public void show() {

    }
}
```

Interfaz Criterio: Usada para definir los distintos tipos de criterios.

```
public interface Criterio {
    boolean before(Email m1, Email m2);
}
```

Criterios: Tenemos 5 clases diferentes que implementan la interfaz Criterio y solo habría que modificar el método before() en cada una de ellas.

```
public class PriorityCriterio implements Criterio {
    public boolean before(Email m1, Email m2) {
        return true;
    }

}
```

```
public class Subject implements Criterio {
    public boolean before(Email m1, Email m2) {
        return true;
    }

}
```

```
public class Text implements Criterio {
    public boolean before(Email m1, Email m2) {
        return true;
    }

}
```

```
public class From implements Criterio {
    public boolean before(Email m1, Email m2) {
        return true;
    }

}
```

```
public class Date implements Criterio {
    public boolean before(Email m1, Email m2) {
        return true;
    }

}
```