Informe patrones de diseño

PRÁCTICA 1

Nerea Martín Serrano | 3º Ingeniería de la Salud | 29/04/2022

Índice

URL a repositorio de GitHub donde está alojado el proyecto: https://github.com/nmartinse/pr1_patrones

Cuestiones

CUESTIÓN 1

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 principal semejanza es que los tres tienen un propósito estructural. Aunque cada uno soluciona un problema (estructural) distinto. El patrón adaptador trata de que se pueden reciclar las clases, el decorador evita jerarquías de clases complejas y el patón representante sirve para controlar el acceso a objetos.

Una similitud entre el patrón adaptador y el decorador modifican objetos ya existentes. La diferencia está en que cada uno modifica un objeto distinto. Por ejemplo, el patrón adaptador lo que hace es cambiar la interfaz de una clase (adapta una interfaz para poder reutilizar una clase existente). Por otro lado, el patrón decorador añade funciones a objetos mediante composición.

Una diferencia del patrón representante frente a los otros dos es que este añade un objeto, como el propio nombre índice añade un representante (añade un proxy).

CUESTIÓN 2

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

La principal similitud es el propósito de los dos es el comportamiento, es decir los dos van a modificar el comportamiento de los objetos.

El patrón estrategia sirve (como el propio nombre indica) para definir distintas estrategias para alcanzar un mismo fin y el patrón estado permite indicar en qué estado se encuentra el objeto en un determinado momento, y dependiendo de ese estado su comportamiento será distinto. Es decir, en el patrón estrategia siempre hay un mismo objetivo (aunque se llegue a él usando distintas estrategias) y el patrón estado el objetivo puede ser distinto dependiendo del estado en el que se encuentro el objeto.

Por ejemplo, imaginamos que vamos a programar un buffer. El patrón estrategia nos podría servir seguir un comportamiento fifo o lifo. El patrón estado nos serviría para llevar un control de si el buffer está lleno o no. Entonces, al añadir un objeto el patrón estrategia se encargaría de añadir un objeto por un método u otro, pero el patrón estado podría impedir que se añadiera un objeto dependiendo del estado (el objetico para el

patrón estrategia es siempre añadir el objeto, el objetivo del patrón estado puede ser evitar que se añada un objeto o no)

CUESTIÓN 3

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

Los dos se encargan de modular los mensajes entre objetos, los dos definen entre que objetos se van a mandar mensajes. El patrón mediador modula a quién le enviamos el mensaje y el patrón observador define una jerarquía entre los objetos que se mandan los mensajes.

El patrón mediador crea un objeto que define como los objeto interactuaran entre sí. El patrón observador no crea un objeto, si no que define a los objetos como Observadore y Observables.

Práctica 1. Cliente de correo e-look

Identifique un patrón de diseño que nos permita resolver la situación presentada, permitiendo incluso cambiar de un criterio de ordenación a otro mientras el usuario utiliza e-look, y de forma que sea fácilmente extensible (por ejemplo, para ordenar por otros criterios aparte de los indicados). Mostrar de forma esquemática la implementación en Java del patrón propuesto al problema descrito.

En esta práctica debemos de implementar un proyecto el Java para simular el programa e-look. Como muestra el enunciado, este programa en principio consta de dos clases: **Mailbox** y **Email**.

Se nos pide que usemos un patrón para dar solución a un problema, el problema de qué criterio seguir para ordenar los emails dentro del mail box. Bueno, pues el criterio que elija el cliente. Es decir, debemos de implementar varios criterios de ordenación y que el cliente pueda elegir el que quiera (incluso mientras esté usando el programa).

Al tener varias estrategias para hacer los mismo, la mejor elección es usar el patrón estrategia. Este nos permitirá definir los distintos criterios de ordenación en distintas clases (de esta forma disminuye la responsabilidad del método *sort()*). Este patrón permite cambiar de estrategia mientras se utiliza e-look. Además, si en un futuro se quiere añadir otro criterio de ordenación, no habría que cambiar nada de lo que ya este hecho, simplemente habría que crear una nueva clase que implemente el método *before()* con la nueva estrategia.

IMPLEMENTACIÓN EN JAVA

Este proyecto funcionará se la siguiente forma. El cliente guardará sus emails en el mail-box ordenándolos según el criterio que él elija, el cual podrá cambiar en cualquier momento.

En un principio tenemos las dos clases que, para implementarlas, vamos a seguir el esquema que nos presenta el enunciado:

- **Clase Email**: esta clase representa los emails. Consta de cinco atributos y un constructor.

```
1  import java.util.Date;
2
3  public class Email {
4      public String from, subject, text;
5      public Date date;
6      public Priority priority;
7
8      public Email(String f, String s, String t, Date d, Priority p) {
9          from =f; subject = s; text = t; date = d; priority = p;
10     }
11 }
```

- **Enum Piority**. La clase Email hace uso de un enumerados:

```
1
2  public enum Priority {
3
4  }
```

- **Clase Mailbox**. En esta clase se define el mail box, objeto que va a usar el programa e-look para almacenar los emails. Vamos a añadir un atributo (que se una instancia de la interfaz Criterio) que guarde el criterio que en ese momento se quiera seguir para ordenar los emails y otro atributo, un Array, como estructura de almacenamiento de los emails .

  Esta clase cuenta con tres métodos. Un primer método llamado *show()* que sirve para visualizar los emails guardados. El segundo método es *sort()*, que se dedica a ordenar los emails en el mail box. Este hace uso del último método, *before()*. El método *before()* va a hacer uso de la interfaz Criterio, la cual se explica posteriormente.

  Además, a esta clase debemos de añadirle un nuevo método que sirva para cambiar de estrategia cuando sea necesario.

```java
import java.util.ArrayList;

public class Mailbox {

    private Criterio criterio;
    private ArrayList<Email> email;

    public Mailbox() {
        email = new ArrayList<Email>();
    }

    public void show() {
    }

    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 criterio.before(m1, m2);
    }

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

Una vez tenemos esto vamos a aplicar el patrón Estrategia, creando interfaz que defina el método que tienen que implementar las distintas estrategias. Por cada criterio so de ordenación se va a crear una clase para cada uno de los criterios de ordenación.

- **Interfaz Criterio**: esta interfaz es la que se va a encargar de modular las distintas estrategias.

```java

public interface Criterio {
    boolean before(Email at, Email at2);
}
```

- Por último, tenemos las clases que implementan esta interfaz. Estas clases van a ser las distintas estrategias. Es decir vamos a crear tantas clases como criterios de ordenación tengamos actualmente. Si en un futuro aparece otro criterio simplemente creamos una nueva clase que implemente la interfaz Criterio.

  Las clases que implementen la interfaz tienen que desarrollar el método *before()*. El método *before()* es un método boolean al que se le pasan como parámetros de entradas dos emails. Devuelve *true*, si el primer email es anterior al otro (dependiendo de la clase, utilizará un criterio u otro).

  Estas clases implementadas en Java:

```java
public class text implements Criterio {
    public boolean before(Email mi, Email m2) {
        return false;
    }
}
```

```java
public class subject implements Criterio {
    public boolean before(Email mi, Email m2) {
        return false;
    }
}
```

```java
public class date implements Criterio {
    public boolean before(Email mi, Email m2) {
        return false;
    }
}
```

```java
public class from implements Criterio {
    public boolean before(Email mi, Email m2) {
        return false;
    }
}
```

```java
public class priority_criterio implements Criterio {
    public boolean before(Email mi, Email m2) {
        return false;
    }
}
```

Entonces, en el proyecto de Java, lo que pasará será que el cliente, para ordenar los emails invocará al método *sort()*, que a su vez este método invocará al método *before()*. El método *before()* invocará a una clase (que implemente la interfaz criterio) dependiendo de la estrategia elegida por el cliente. Esa clase se encargará de decidir entre dos emails cual va el primero y el método *sort()* se encargará de ordenarlos gracias a esto.