Patrones de Diseño

Ingeniería del Software Avanzada

Jesús Aldana Martín

Cuestiones

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.

Pregunta 1:

Los tres patrones de estudio se clasifican entre los patrones estructurales, patrones que se refieren a relaciones entre clases y/u objetos. Solo con el nombre de los patrones ya tenemos una pista acerca de su funcionalidad. El patrón Representante es el que más se diferencia respecto a los otros dos ya que este te permite crear un sustituto para otro objeto ,es decir, una especie de 'proxy' que te da control de acceso al objeto original esto nos permite interferir en la solicitud al objeto original antes o después de una llamada. El siguiente patrón es el Adapter, patrón que nos permite encapsular una clase y traducirla a diferentes formatos, un ejemplo sería aplicar este patrón a una clase cuya estructura no sea compatible con una interfaz. Por último el patrón Decorador nos permite añadir funcionalidades a objetos colocándolos dentro de objetos encapsuladores que ya contienen estas funcionalidades. Como conclusión Adaptador proporciona una interfaz diferente al objeto envuelto, Representante proporciona la misma interfaz y Decorador entrega una interfaz mejorada.

Pregunta 2:

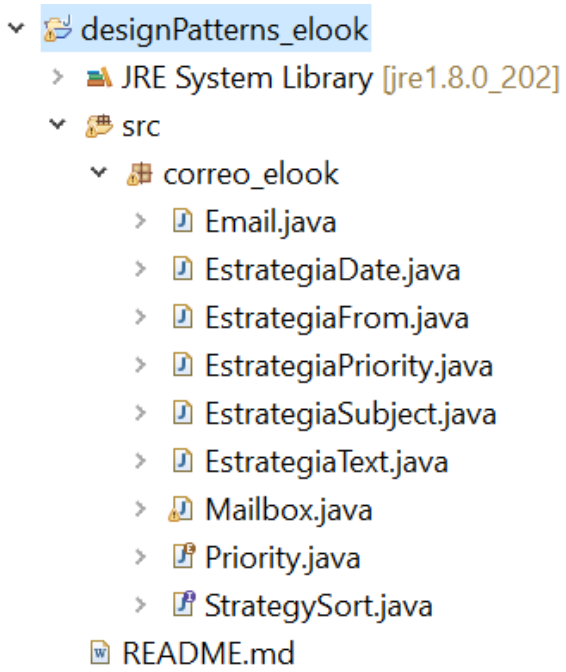
El patrón Estrategia nos permite definir una familia de algoritmos y colocarlos en clases separadas haciendo sus objetos intercambiables y el patrón Estado permite alterar su comportamiento cuando su estado interno cambia. Ambos patrones se categorizan dentro de los patrones de comportamiento y cambian el comportamiento de la clase principal delegando parte del trabajo a otros objetos. En estrategia cada estrategia es individual y no conoce el contexto por el cual son llamadas,sin embargo el patrón Estado no restringe las dependencias entre estados concretos permitiendo alterar el estado principal a voluntad. A efectos prácticos podría decirse que Estado es una extensión del patrón Estrategia.

Pregunta 3:

El patrón Observador permite definir un mecanismo de ‘suscripción’ para notificar a varios objetos cualquier evento que suceda en el objeto de observación, el patrón sugiere crear una matriz que almacena una lista de referencias a objetos suscriptores y varios métodos públicos que permiten añadir o eliminar suscriptores. El Mediador permite reducir dependencias ‘caóticas’ entre objetos restringiendo las comunicaciones entre los objetos forzándolos a colaborar a través de un objeto mediador. Ambos patrones se complementan pudiendo considerar al objeto mediador como el notificador del patrón Observador y los componentes actuando como suscriptores que cambian según los eventos del mediador. De esta forma el patrón Mediador es prácticamente igual al Observer. Esto es así debido a que la meta principal del patrón mediador es eliminar las dependencias mutuas entre un grupo de componentes del sistema volviéndose dependientes de un objeto mediador y el patrón observador establece conexiones dinámicas de un único sentido entre objetos.

Práctica 1. Cliente de correo e-look

Tal y como se nos presenta en el enunciado del problema necesitamos parametrizar el método sort() respecto al criterio que decidamos para ordenar el Mailbox y el patrón de diseño adecuado para ello es el estrategia. El patrón estrategia nos permite definir una familia de implementaciones y colocar cada una de ellas en una clase separada o estrategias.
Con la siguiente imagen podemos observar la estructura del proyecto, he implementado primero las clases Email y Mailbox, luego la interfaz StrategySort con el método before que implementarán cada una de las estrategias que se ven.



La clase Mailbox hace referencia a un objeto que implementa la interfaz.

```java
package correo_elook;

public class Mailbox {

    private StrategySort st;

    public Mailbox(StrategySort st) {
        this.st = st;
    }

    public void show() {
    }

    private void sort() {
        // TODO Auto-generated method stub
    }

    private boolean before(Email m1, Email m2) {

        return st.before(m1, m2);
    }
}
```

- Interfaz con el método before.

```java
package correo_elook;

public interface StrategySort {

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

- EstrategiaFrom, he utilizado compareTo() para realizar el método before, esta función viene en eclipse por defecto y es muy útil para comparar dos parámetros, si m1 es mayor que m2 devuelve 1, de ser iguales devuelve 0 y de ser m1 menor que m2 devolverá -1.

```java
package correo_elook;

public class EstrategiaFrom implements StrategySort {

    @Override
    public boolean before(Email m1, Email m2) {

        return m1.from.compareTo(m2.from) < 0;
    }
}
```

El resto de estrategias Date, Subject, Priority y Text se implementan de forma análoga ya que el método bubble sort que utilizamos es común la única diferencia son los parámetros por los cuales ordenaremos el Mailbox.

Github

https://github.com/jesusaldanamartin/designPatterns_eLook