PRÁCTICA 1

PATRONES DE DISEÑO

https://github.com/carlosbelop/Practica1.git

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).

Adaptador y Decorador.

- Semejanzas:
  1. Ambos permiten añadir funcionalidad a un objeto.
  2. Se pueden usar composición en los dos.
- Diferencias:
  1. El patrón adaptador se hace con objetivo de habilitar una clase ya existente concreta para que concuerde con la interfaz del cliente con posibilidad de añadir alguna funcionalidad mientras que el decorador no trata de adaptar una clase concreta para reutilizarla, si no que a partir de una más simple adecuada a la interfaz se van definiendo tantas funcionalidades como se quieran, mediante herencia y composición
  2. El patrón adaptador hace de intermediario entre la interfaz y el objeto. El patrón decorador no hace de intermediario de un objeto con la interfaz, porque el decorador base está en contacto directo con ella.

Adaptador y Representante.

- Semejanzas:
  1. Ambos hacen de intermediario entre dos clases.
- Diferencias:
  1. El objetivo del representante es facilitar el acceso a un objeto haciendo transparentes cosas indiferentes para el usuario. Por otro lado, el objetivo del adaptador es simplemente poder integrar ese objeto complejo en la interfaz requerida.
  2. El patrón adaptador esta relacionada directamente con una interfaz o clase abstracta y el representante no tiene por qué.
  3. El patrón Representante no contempla añadir funcionalidades como sí lo hace el adaptador.

Decorador y Representante

- Semejanzas:
  1. Delega las funciones en otras clases inferiores, (decoradores concretos en el caso de decorador).

- Diferencias:
  1. El patrón decorador añade funcionalidades a objetos y el representante engloba todas las funcionalidades de la forma más simplificada posible, sin variar el comportamiento final.

Q2

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

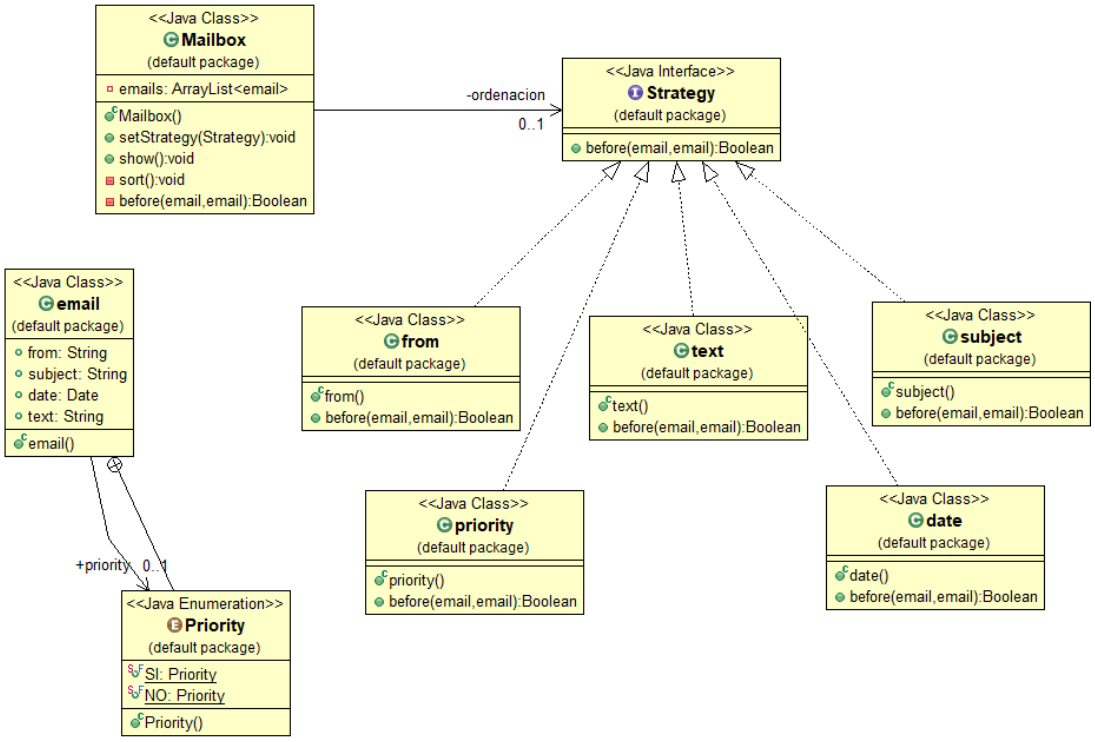
- Semejanzas:
  1. Ambos implementan una interfaz o clase abstracta para unificar en un tipo diferentes acciones.
  2. Las dos clases principales necesitan referenciar al tipo concreto de la interfaz.
  3. Varían su comportamiento dependiendo de acciones o elecciones previas.
- Diferencias:
  1. El patrón “estrategia” requiere de una previa elección de esa específica condición (estrategia) para realizar unas acciones concretas mientras que el patrón “estado” son unas acciones específicas las que desencadenan la condición (estado).
  2. En patrón “estrategia” por tanto, se debe incluir un método usable externamente que cambie la estrategia elegida mientras que en el patrón “estado” no tiene por qué.

Q3

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

- Semejanzas:
  1. Los dos patrones se encargan de gestionar la manera en la que clases interactúan entre ellas. Más específicamente, gestionan el método de notificar a otras clases si ha habido algún cambio de estado o evento en otra específica.
  2. Ambos disminuyen la complejidad y el acoplamiento cambiando de relaciones “muchos a muchos” a relaciones de “uno a muchos”.
- Diferencias:
  1. La clase mediador necesita conocer a todas las demás clases que tienen que ser notificadas sin embargo, el observador no necesita saber quiénes son sus observadores.
  2. El patrón mediador crea una clase adicional que gestiona todas estas relaciones entre el modelo mientras que el observador necesita implementar una referencia al estado del observable.

Práctica 1. Cliente de correo e-look



Una buena práctica sería aplicar el patrón de diseño “Estrategia” porque cambia el comportamiento de un objeto dependiendo de diferentes algoritmos. Estos algoritmos pueden ser diferentes clases independientes entre sí por lo que si quisiésemos añadir uno, no habría que modificar los demás.

Así podemos crear los métodos de ordenación ya propuestos (from, subject, date…), cada uno como un tipo distinto en una clase que hereda de una principal y así poder añadir en un futuro si fuese necesario.

El código implementado en java es una muestra estructural, por lo que los métodos y el programa de ordenación de correos no está completado.

Código

```
public class Mailbox {

    private Strategy ordenacion;
    private ArrayList<email> emails;

    public void setStrategy(Strategy x) {
        ordenacion=x;
    }

    public void show() {}

    private void sort() {
     //Aquí se usa el before(email m1, email m2);
    }

    private Boolean before (email m1, email m2) {

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

Clase Mailbox

En esta clase se incluye un atributo donde se almacenan todos los emails recibidos y un atributo el cual determina el método de ordenación aplicado en ese momento.

"Sort()" será el método que llame a "**before()**" (algoritmo de ordenación) y este variará con respecto a la estrategia establecida.

"Before()" será incluido en la interfaz e implementado en cada una de las estrategias.

```
public class email {

    public enum Priority{
        SI,NO;
        }

    public String from;
    public String subject;
    public Date date;
    public Priority priority;
    public String text;
}
```

Clase email

Una clase que simula todas las partes que componen a un e-mail.

```
public interface Strategy {

    public Boolean before(email m1, email m2);

}
```

Interfaz Strategy

Es la interfaz que unifica en un "tipo" todas las estrategias y les obliga a llevar el método de ordenación implementado de distinta manera.

<u>Estrategias:</u>

```java
public class text implements Strategy {

    public Boolean before(email m1, email m2) {
        return true;
    }

}
```

```java
public class subject implements Strategy {

    public Boolean before(email m1, email m2) {
        return true;
    }

}
```

```java
public class priority implements Strategy {

    public Boolean before(email m1, email m2) {
        return true;
    }

}
```

```java
public class from implements Strategy {

    public Boolean before(email m1, email m2) {
        return true;
    }

}
```

```java
public class date implements Strategy {

    public Boolean before(email m1, email m2) {
    return true;
    }

}
```

Todas muy similarmente implementadas, sólo varía el nombre (y la implementación de “before ()” si estuviese).

Conclusión

El patrón Estrategia encaja perfectamente con el problema planteado, permite implementar todos los métodos de ordenación sin cambiar apenas código y además reutilizando mucho lo que facilita su proceso.