DISEÑO SOFTWARE

PRÁCTICA DISEÑO

CURSO 2021-2022

Gráfico

Descripción generada automáticamente

Marcos Varela

Hugo Castro

Grupo 2.2

**INFORME EJERCICIO 1**

**Principios SOLID utilizados:**

**Principio de Sustitución de Liskov:** Cada una de las subclases pueden ser su clase base sin que cambie su comportamiento. Ejemplo de este principio en el código, en la hipotética creación de un objeto de tipo OR:

Composition or1 = new OR();  
*OR* or1 = new OR();

**Principio de Segregación de Interfaces**: En este caso solo hay una interfaz que incluye solo un método, con lo que no es necesaria la aplicación de este principio, pero de haber dos o más que definan comportamientos distintos, lo correcto sería implementarlo, como decimos, en varios interfaces y no en uno solo.

**Principio de Responsabilidad Única:** Todas las clases implementadas tienen una única responsabilidad. Cada criterio de filtrado lo realiza en base solo a ese criterio y la clase Ticket tiene operaciones triviales necesarias para trabajar sobre ella.

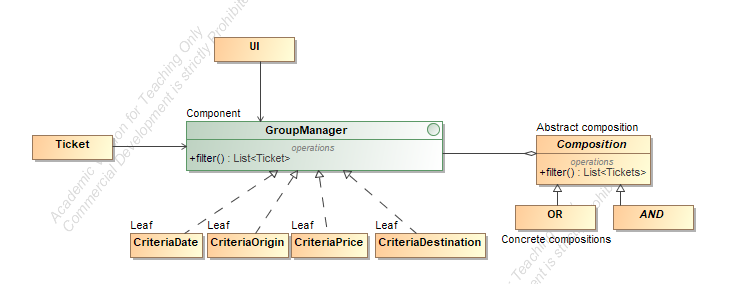
public class *CriteriaOrigin* implements *GroupManager* {  
  
 public *String* origin;  
  
 *List*<*Ticket*> filteredList = new ArrayList<>();  
  
 CriteriaOrigin(*String origin*){  
 this.origin = *origin*;  
 }  
  
 *@Override* public *List*<*Ticket*> filter(*List*<*Ticket*> *availableTickets*) {  
 int i;  
 for(i = 0; i < *availableTickets*.size(); i++){  
 if(*availableTickets*.get(i).getOrigin().equals(origin)){  
 if(!filteredList.contains(*availableTickets*.get(i))){  
 filteredList.add(*availableTickets*.get(i));  
 }  
 }  
 }  
 return filteredList;  
 }  
}

**Principio Abierto – Cerrado:** También hemos hecho hincapié en cumplirlo, pues cada criterio es una clase diferente, por lo que se puede ampliar, por ejemplo, añadiendo más criterios de filtrado, sin necesidad de modificar el código ya existente (ejemplo visible en el diagrama UML que se encuentra más abajo).

**Principio de Inversión de la Dependencia:** No hay suficientes niveles como para estudiar a fondo este principio, pero una evidencia sí es que no existen dependencias en las clases de alto nivel con respecto a las de bajo nivel.

**Patrones de Diseño utilizados:**

1. **Patrón Composición,** donde se puede ver el UML con el rol que juega cada clase más abajo. Hemos considerado la utilización de este patrón ya que pensamos el ejercicio como una composición, valga la redundancia. El OR estaría compuesto de dos criterios y un AND sería una composición de dos cláusulas OR. Diagrama y roles a continuación:

****

1. **Patrón Inmutable**. Un ticket ya expedido no debería cambiar, es decir, no se podría cambiar el origen, la fecha, el precio o el destino de una forma que no fuese expidiendo otro, por lo que hemos decidido la aplicación de este patrón sobre la clase Ticket. Para ello hemos utilizado técnicas como definir todos los atributos como privados y finales (se obtendrían mediante métodos get()).

public class *Ticket* {  
  
 private final *String* origin;  
 private final *String* destination;  
 private final *String* date;  
 private final int price;  
  
 Ticket(*String origin*, *String destination*, *String date*, int *price*){  
 this.origin = *origin*;  
 this.destination = *destination*;  
 this.date = *date*;  
 this.price = *price*;  
 }  
  
 public *String* getOrigin() {  
 return origin;  
 }  
  
 public *String* getDestination() {  
 return destination;  
 }  
  
 public *String* getDate() {  
 return date;  
 }  
  
 public int getPrice() {  
 return price;  
 }