Práctica 1

Equipo: Los Peaky Blinders
Carlos Emilio Castañón Maldonado José Camilo García Ponce Dafne Bonilla Reyes

1 Menciona los principios de diseño esenciales del patrón Strategy y Observer. Menciona una desventaja de cada patrón

❖ Principios de diseño esenciales

- **Strategy**

  Strategy es un patrón de diseño de software que permite seleccionar el comportamiento de un algoritmo en tiempo de ejecución. Es decir, Strategy funciona a partir de la idea de definir un grupo de algoritmos, encapsular cada algoritmo, y hacer que los algoritmos sean intercambiables dentro de ese grupo, confiando en la composición en lugar de la herencia para la reutilización.

  En este patrón de diseño tendremos un contexto que se compone de una estrategia. En lugar de implementar un comportamiento, el contexto lo delega a la estrategia. El contexto sería la clase que requeriría cambiar comportamientos. Podemos cambiar el comportamiento dinámicamente. La estrategia se implementa como interfaz para que podamos cambiar el comportamiento sin afectar nuestro contexto.

- **Observer**

  El patrón de diseño Observer tiene como principio fundamental la relación entre un sujeto y un observador. A esta relación la podemos ver análogamente como una suscripción a una revista, periódico, etc. Es decir, el editor de la revista será el sujeto y el suscriptor el observador. De esta manera, si el observador está registrado como suscriptor recibirá nuevas ediciones y si se da de baja dejará de recibirlas. Es importante mencionar que el editor de la revista no sabe quién es el observador, ni como usa la revista, él solo se encarga de entregar la revista si el observador está suscrito.

  El patrón Observer define una dependencia de uno a muchos entre objetos, de modo que si un objeto cambia de estado, todos sus dependientes son notificados y actualizados automáticamente. Esta dependencia se da porque los propios observadores no tienen acceso a los datos.
  Dependen del sujeto para que les proporcione datos.

❖ Desventajas

- **Strategy**

  Debido a su estructura, el contexto y las clases normalmente se comunican a través de la interfaz especificada, lo que puede crear redundancias e ineficiencias en la comunicación interna, es decir, la interfaz puede acabar sobredimensionada, por lo que no siempre será utilizada de manera óptima, produciendo transferencias de datos innecesarias y dado que todas las clases deben implementar la misma interfaz, algunas de ellas tendrán que implementar métodos que no necesitan para su comportamiento.

- **Observer**

  Para ciertos observadores, las actualizaciones automáticas emitidas por el sujeto pueden ser irrelevantes, suponiendo una desventaja, especialmente cuando se tiene una gran cantidad de observadores registrados, ya que llevará mucho tiempo notificar a todos los observadores, malgastando tiempo de computación.

2 Uso del programa

➢ Compilar desde `src/`

```
javac *.java
```

➢ Ejecutar desde `src/`

```
java Practica1
```

➢ Generar la documentación desde `src/`

```
javadoc -d docs *.java
```

➢ **Explicación:**

Primero, es necesario compilar y ejecutar el programa. Una vez hecho esto, se generan los espectadores. Estos ya tienen un nombre predefinido, pero se elige aleatoriamente a quién apoyarán. A continuación, se realiza un orden aleatorio para la pelea. Existen 3 casos de distintos tamaños, uno pequeño, uno mediano y uno largo. Con todo esto listo, la pelea empezará. Al finalizar la pelea, los archivos de los espectadores se generarán en el directorio *src/*, sin embargo, en caso de que ocurra un error de *entrada/salida*, se mostrará el error en la terminal.

Si se quiere generar la documentación, esto sería con el comando dado arriba, y luego los archivos se generarán en el directorio llamado *docs*.