Práctica 04 - Pumalacticos

Integrantes

- 321671958 - *Cisneros Álvarez Danjiro*.
- 422083399 - *Teniente Ornelas Oscar Manuel*.
- 422109167 - *Tenorio Reyes Ihebel Luro*.

Este proyecto es una implementación de un sistema de catálogo para la tienda de renta de películas y discos "RockBuster".

Descripción

El sistema gestiona un catálogo de productos que incluye películas individuales, sagas de películas (que pueden contener otras sagas) y discos musicales de una tienda asociada, "Mixdown". La aplicación permite al usuario ver el catálogo completo, filtrar productos por género o precio máximo y ver los detalles de un producto específico.

Instrucciones de Ejecución

Con Docker

1. Desde la raíz, genera la imagen con docker

```
sudo docker build -t rock-buster .
```

2. Desde ahí mismo ejecuta con:

```
sudo docker run -it --rm rock-buster
```

> **Nota**
>
> Parece que según la distro no es necesario usar "sudo" pero como en el caso de uno de nuestros integrates (con LMDE6) si lo fue, decidimos agregarlo a la instrucción.

Con Javac y Java

Esta práctica fue hecha con Java 17 y se compila con los siguientes comandos e instruciones:

1. Compila todos los archivos `.java` desde la raíz del directorio `src`:

```
javac -d ./ src/**/*.java
```

2. Ejecuta la clase principal:

```
java com.pumalactivos.rock_buster_app.Rock_Buster_App
```

3. Sigue las instrucciones del menú interactivo en la consola.

Patrones de Diseño

Patrón Composite

El requisito principal era manejar una jerarquía de productos: las **películas** son objetos simples, mientras que las **sagas** son colecciones que pueden contener tanto películas como otras sagas. Esta estructura de "partes y todo" es exactamente el problema que el patrón Composite está diseñado para resolver.

Aplicación

1. Se creó una interfaz `Component`, que define el contrato común para todos los objetos en la jerarquía (tanto simples como compuestos). Esta interfaz declara métodos como `getNombre()`, `getPrecioRenta()`, etc.
2. La clase `Pelicula` implementa `Component` y actúa como la un objeto individual y no puede contener otros.
3. La clase `Saga` también implementa `Component` y actúa como el **"Composite"**. Mantiene una lista de `Component`s y delega las operaciones a sus hijos. Por ejemplo, el `getPrecioRenta()` suma los precios de sus componentes y aplica un descuento, permitiendo tratar a una saga compleja como si fuera un solo ítem.

Gracias a este patrón, el menú en `Rock_Buster_App` puede tratar a una película individual y a la saga completa de "Cosmere" de la misma manera, simplificando la lógica para mostrar el catálogo, filtrar y calcular precios.

Patrón Adapter

El segundo requisito era integrar los productos de "Mixdown", que son **discos**, a nuestro sistema. El sistema de Mixdown tiene su propia clase `Disco` con una interfaz incompatible

(por ejemplo, tiene `getArtista()` en lugar de `getDirector()`). Modificar nuestro sistema o la clase `Disco` para que fueran compatibles violaría el principio de Abierto/Cerrado.

Aplicación

1. Se creó una clase `Adapter` que funciona como un "traductor".
2. Esta clase implementa la interfaz que nuestro sistema espera: `Component`.
3. Internamente, la clase `Adapter` **envuelve** una instancia de la clase `Disco` (el "Adaptee").
4. Cuando el sistema cliente llama a un método de `Component` en el `Adapter` (por ejemplo, `getDirector()`), el adaptador redirige la llamada al método correspondiente en el objeto `Disco` (en este caso, `getArtista()`).

Este patrón nos permitió integrar la clase `Disco` en nuestro catálogo de `Component`s de forma transparente y sin modificar el código existente, logrando una solución limpia y elegante.