Instituto Tecnológico y de Estudios Superiores de Monterrey

TC1030. Programación orientada a objetos

Grupo 570

E2. Proyecto Integrador

Alumno:

Icker Villalón Lozoya A01613306

Docente:

León Felipe Guevara Chávez

24 de Julio del 2024, Monterrey, Nuevo León

Índice

Introducción (planteamiento del problema).

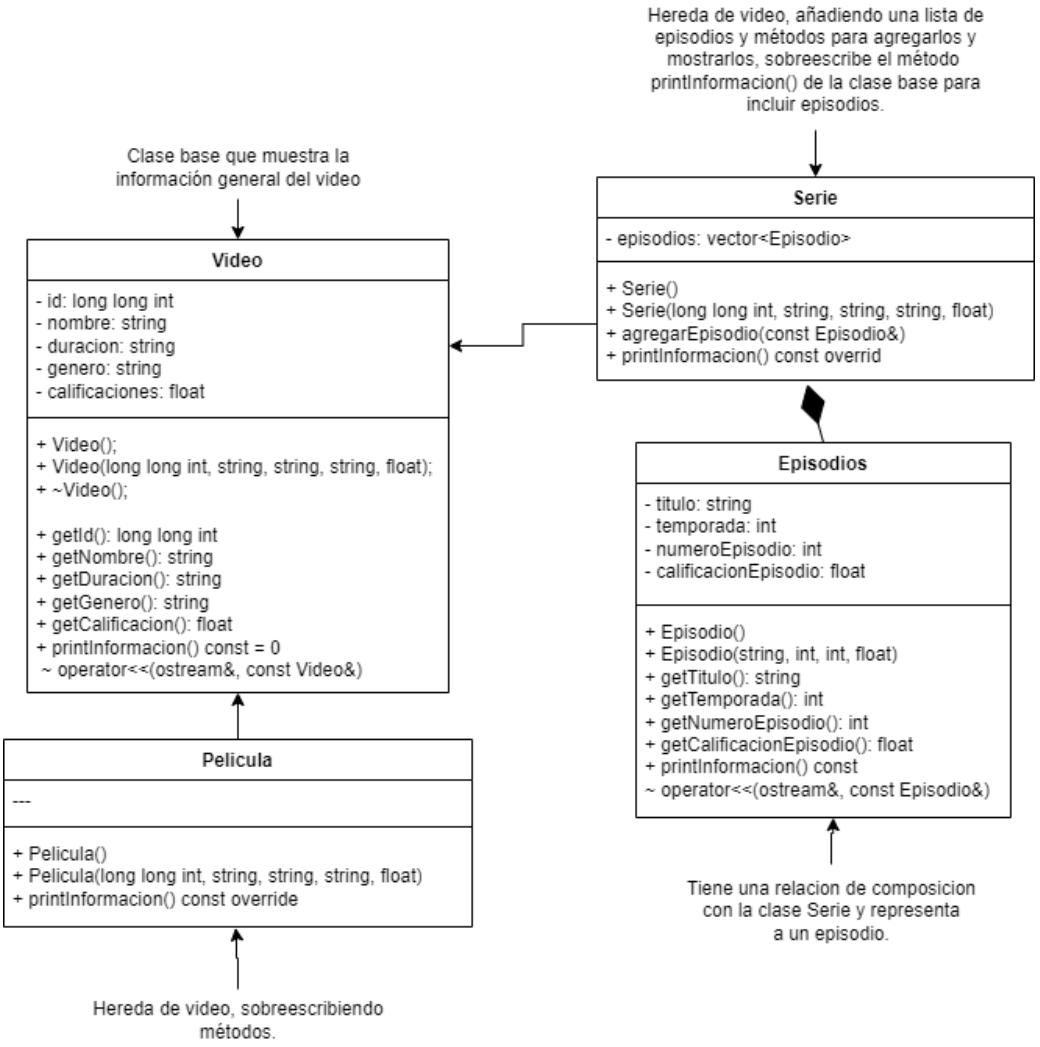
En los últimos años, los servicios de streaming de video bajo demanda, como Netflix, Disney+, y DC Universe, han proliferado. Algunos de estos servicios se especializan en ofrecer un amplio volumen de videos a sus usuarios, mientras que otros se enfocan en mostrar únicamente contenido de su propia marca. Una versión limitada para apoyar a un futuro proveedor de este tipo de servicios es la que se describe a continuación:

Se desea trabajar con dos tipos de videos: películas y series. Todo video tiene un ID, un nombre, una duración y un género (drama, acción, misterio). Las series tienen episodios, y cada episodio tiene un título y una temporada a la que pertenece. Nos interesa conocer la calificación promedio que ha recibido cada uno de los videos. Esta calificación está en una escala de 1 a 5, donde 5 es la mejor calificación.

El sistema debe ser capaz de:

- ➔ Mostrar los videos en general con sus calificaciones.
  - ◆ Mostrar los episodios de una determinada serie con sus calificaciones.
  - ◆ Mostrar las películas con sus calificaciones.

Diagrama de clases UML.



Argumentación del diseño.

El diseño de las clases presentadas sigue principios de programación orientada a objetos como la herencia y la composición para lograr una estructura modular y reutilizable. Aquí se describe el razonamiento detrás del diseño:

1. Herencia:
    a. Se utiliza herencia para modelar una jerarquía donde **Video** actúa como clase base que encapsula las características mencionadas en el planteamiento del problema. **Pelicula** y **Serie** heredan de **Video**, lo cual permite que ambas clases reutilicen los atributos y métodos de **Video**, promoviendo la reutilización del código y reduciendo la redundancia.
    b. **Pelicula** y **Serie** sobrescriben el método printInformacion() de la clase base para proporcionar una implementación específica para cada tipo de video.
2. Composición:
    a. **Serie** tiene una relación de composición con **Episodio**, lo que implica que una **Serie** está compuesta por varios episodios. Esta relación es modelada mediante un vector de objetos **Episodio** dentro de la clase **Serie**. La composición es adecuada aquí ya que un episodio no puede existir independientemente sin una serie.

Objetos Involucrados y sus Características y Comportamientos.

1. Video:
    a. Características (Atributos):
        i. id: Identificador único del video.
        ii. nombre: Nombre del video.
        iii. duracion: Duración del video en formato de texto.
        iv. genero: Género del video.
        v. calificacion: Calificación del video.
    b. Comportamientos (Métodos):
        i. Constructor por defecto y sobrecargado.
        ii. Métodos getter para cada atributo.
        iii. Método virtual puro printInformacion() para imprimir la información del video.
        iv. Sobrecarga del operador << para facilitar la impresión de la información del video.
2. Pelicula:
    a. Características (Atributos):
        i. Hereda todos los atributos de Video.
    b. Comportamientos (Métodos):
        i. Constructor por defecto y sobrecargado.
        ii. Sobrescribe el método printInformacion() para proporcionar una implementación específica para las películas.
3. Serie:
    a. Características (Atributos):
        i. Hereda todos los atributos de Video.
        ii. episodios: Vector de objetos Episodio que representa los episodios de la serie.

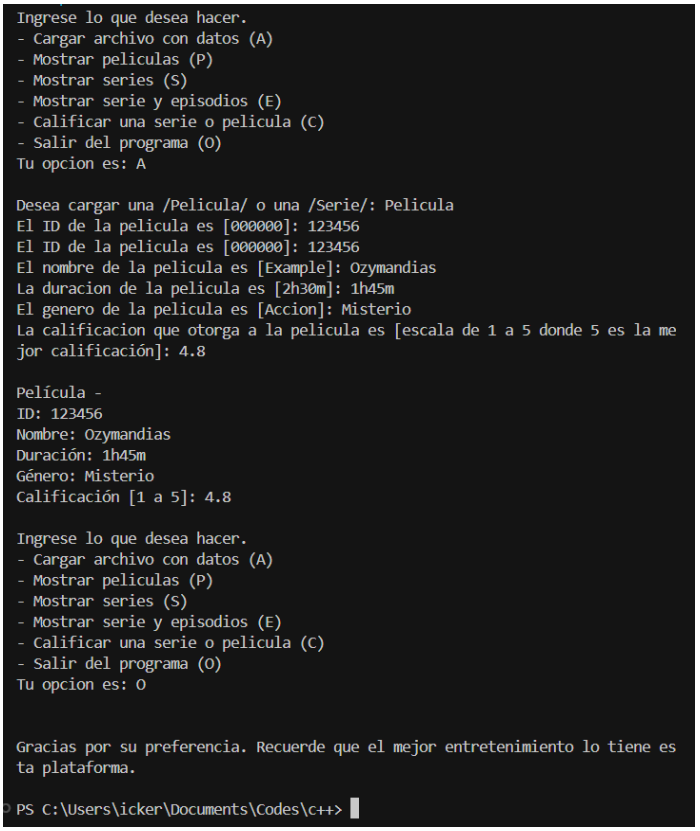
b. Comportamientos (Métodos):
    i. Constructor por defecto y sobrecargado.
    ii. Método agregarEpisodio() para añadir episodios a la serie.
    iii. Sobrescribe el método printInformacion() para proporcionar una implementación específica para las series, incluyendo la información de los episodios.

4. Episodio:
    a. Características (Atributos):
        i. titulo: Título del episodio.
        ii. temporada: Número de la temporada a la que pertenece el episodio.
        iii. numeroEpisodio: Número del episodio dentro de la temporada.
        iv. calificacionEpisodio: Calificación del episodio.
    b. Comportamientos (Métodos):
        i. Constructor por defecto y sobrecargado.
        ii. Métodos getter para cada atributo.
        iii. Método printInformacion() para imprimir la información del episodio.
        iv. Sobrecarga del operador << para facilitar la impresión de la información del episodio.

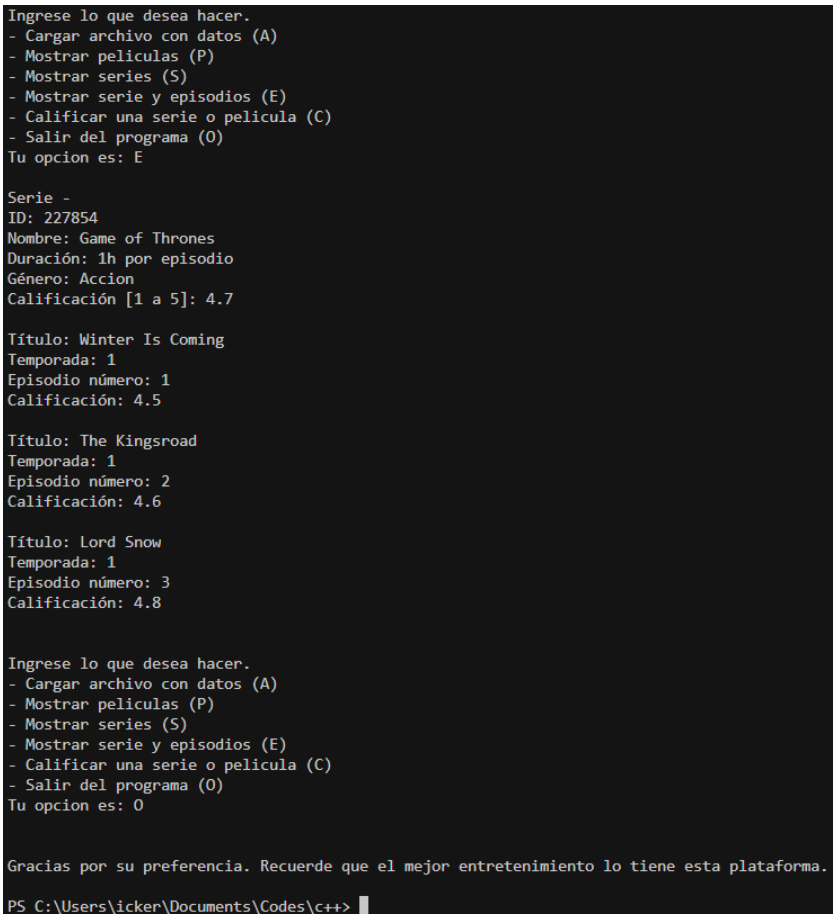
Este diseño modular permite una fácil expansión y mantenimiento del código, facilitando la adición de nuevos tipos de videos o funcionalidades adicionales sin necesidad de modificar el código existente de manera significativa.

Ejemplo de ejecución.

Ejemplo utilizando “Cargar archivo con datos (A)” y luego “Salir del programa (O)”:



Ejemplo utilizando “Mostrar serie y episodios (E)” y luego “Salir del programa (O)”:



Argumentación de las partes del proyecto.

a) Se identifican de manera correcta las clases a utilizar

- Las clases Video, Pelicula, Serie, y Episodio son claramente identificadas y utilizadas en el diseño. Video actúa como la clase base que define las propiedades generales descritas en el planteamiento del problema, mientras que Pelicula y Serie heredan de Video para definir tipos específicos de videos. Episodio es una clase independiente que representa los episodios de una serie. Esta solución permite una estructura clara y modular, facilitando la extensión del sistema con nuevos tipos de videos si es necesario, sin afectar las clases existentes.

b) Se emplea de manera correcta el concepto de Herencia

- Pelicula y Serie heredan de Video, reutilizando sus atributos y métodos. La herencia permite compartir el comportamiento común entre diferentes tipos de videos, reduciendo la redundancia y promoviendo la reutilización del código. La herencia es adecuada aquí porque Pelicula y Serie son tipos específicos de Video, según se menciona en el planteamiento del problema, y comparten muchas propiedades comunes. La herencia facilita la gestión de estas propiedades comunes en una clase base (Video).

c) Se emplea de manera correcta los modificadores de acceso

- Se utilizan los modificadores private para los atributos de las clases, protegiendo los datos y asegurando que solo se puedan acceder a ellos a través de métodos públicos (public). Esto sigue el principio de encapsulación. La encapsulación mejora la seguridad y la integridad de

los datos, permitiendo un control más fino sobre cómo se accede y modifica la información en las clases. Esto es esencial para mantener un código limpio y manejable.

d) Se emplea de manera correcta la sobrecarga y sobreescritura de métodos

- ➢ El método printInformacion() es sobrescrito en Pelicula y Serie para proporcionar implementaciones específicas para cada clase. La sobrescritura permite que cada subclase (Pelicula, Serie) tenga su propia versión del método printInformacion(), adaptada a sus necesidades específicas.

e) Se emplea de manera correcta el concepto de Polimorfismo

- ➢ El método virtual puro printInformacion() en la clase Video permite que las subclases (Pelicula y Serie) implementen su propia versión del método. Esto permite el uso de polimorfismo, donde se puede llamar a printInformacion() en un objeto Video y la implementación correcta se ejecutará dependiendo del tipo real del objeto. El polimorfismo proporciona flexibilidad y extensibilidad, permitiendo que el código maneje diferentes tipos de videos de manera uniforme. Esto es útil en situaciones donde se maneja una colección de objetos Video y se necesita ejecutar el mismo método en todos ellos, sin preocuparse del tipo específico de cada objeto.

f) Se emplea de manera correcta el concepto de Clases Abstractas

- ➢ Video es una clase abstracta porque contiene el método virtual puro printInformacion(). Esto significa que no se pueden crear instancias directas de Video, y las subclases deben proporcionar una implementación para el método virtual. Las clases abstractas permiten definir una interfaz común para todas las subclases, asegurando que cada subclase implemente métodos específicos. Esto es útil para crear una jerarquía de clases clara y organizada.

g) Se sobrecarga al menos un operador en el conjunto de clases propuestas

- ➢ El operador << se sobrecarga para las clases Video y Episodio, permitiendo la impresión directa de objetos de estas clases utilizando cout. La sobrecarga de operadores proporciona una manera intuitiva y conveniente de imprimir la información de los objetos, mejorando la legibilidad del código y facilitando la depuración y presentación de datos.

Casos que harían que deje de funcionar.

- ➢ Entrada de caracteres no válidos: Por ejemplo, si el usuario ingresa algo que no es un número donde se espera un número (por ejemplo, al ingresar el ID de un video).

Conclusión personal.

Este proyecto integrador ha sido una valiosa experiencia para aplicar conceptos de programación orientada a objetos (POO) en C++. Utilizando herencia, encapsulación, polimorfismo y composición, se logró crear un sistema modular, reutilizable y fácil de mantener para gestionar videos. La herencia permitió reutilizar código y extender el sistema sin modificar su estructura, mientras que la composición facilitó la organización lógica de los datos. La validación de entradas fue crucial para asegurar la robustez del sistema. En resumen, este proyecto consolidó mi comprensión de POO y destacó la importancia de buenas prácticas de diseño en el desarrollo de software.

Referencias.

- Karunakar, V. (2024, 18 abril). Object Oriented Programming in C++. GeeksforGeeks. Recuperado 22 de julio de 2024, de https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/
- Agarwal, H. (2024, 5 julio). Inheritance in C++. GeeksforGeeks. Recuperado 22 de julio de 2024, de https://www.geeksforgeeks.org/inheritance-in-c/
- Agarwal, H. (2023, 16 noviembre). C++ Polymorphism. GeeksforGeeks. Recuperado 22 de julio de 2024, de https://www.geeksforgeeks.org/cpp-polymorphism/
- GeeksforGeeks. (2024, 9 julio). Operator Overloading in C++. Recuperado 22 de julio de 2024, de https://www.geeksforgeeks.org/operator-overloading-cpp/