Tecnológico de Monterrey

Modelado de servicio de streaming

Raúl Correa Ocañas

A01722401

Programación orientada a objetos (Gpo 303)

Campus Monterrey

16 de junio de 2023

Evidencia 2: Situación Problema

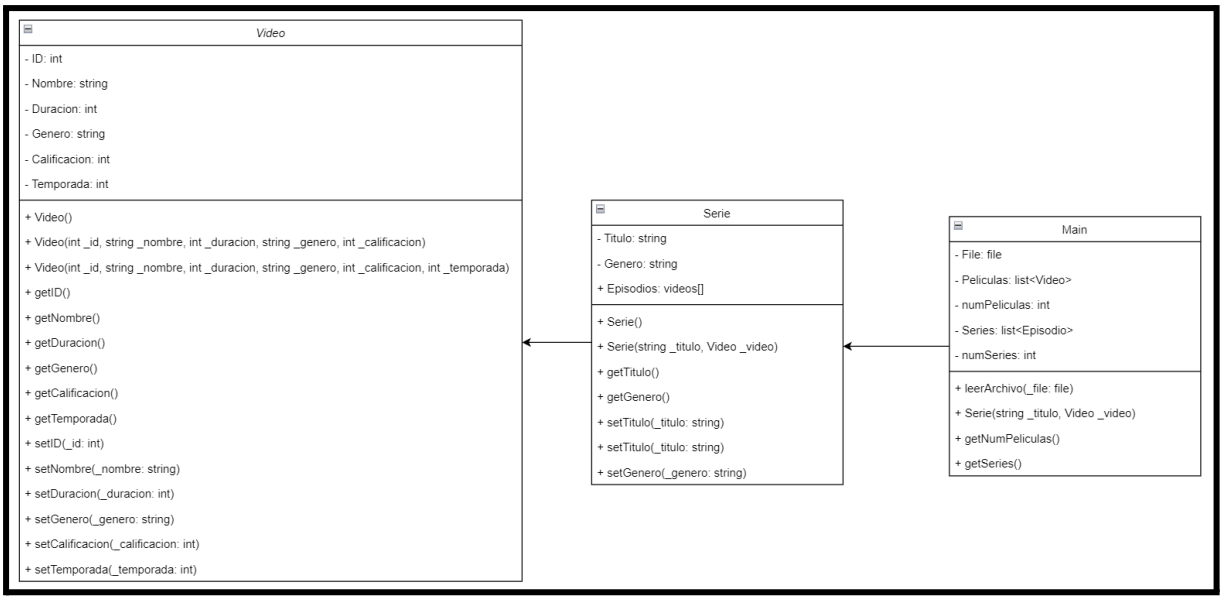
Indice

Introducción

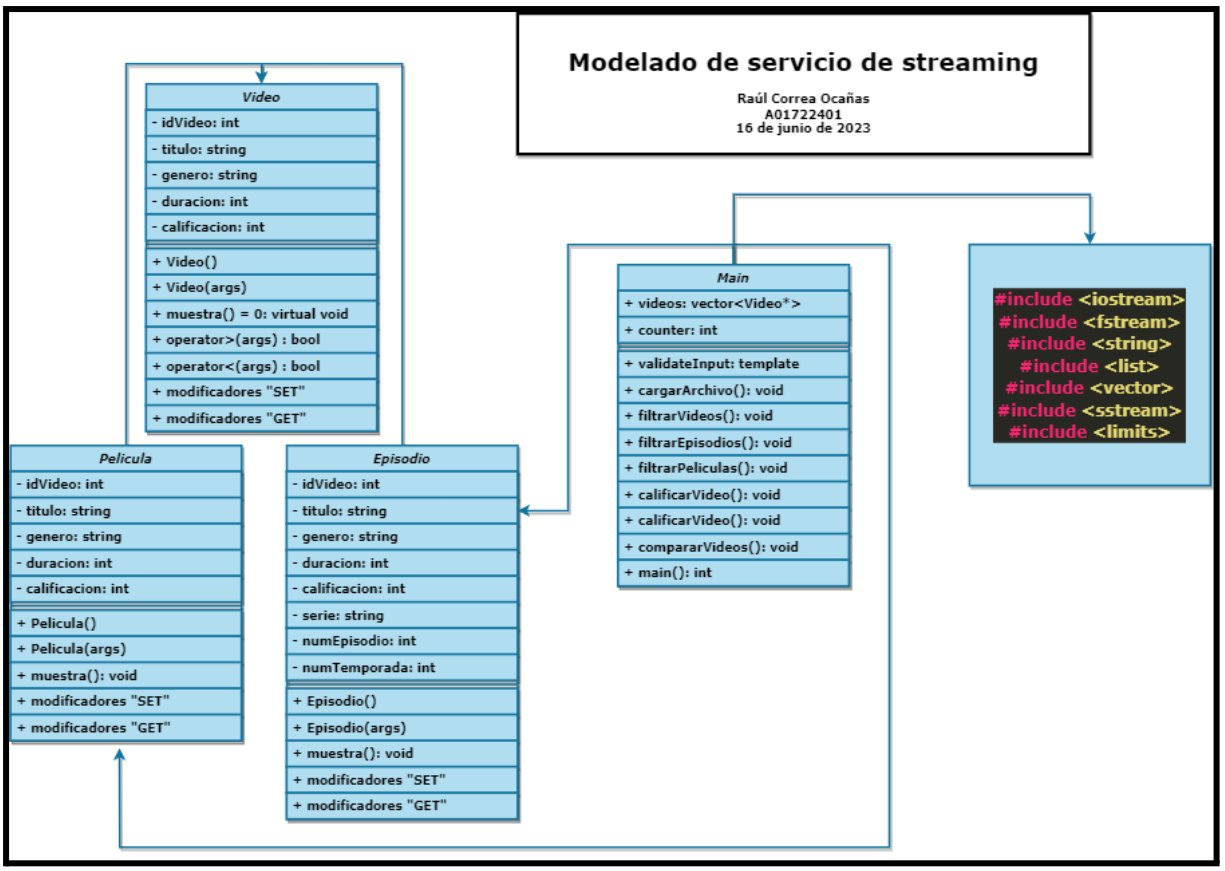
Estos últimos años, el desarrollo de sistemas para el streaming de películas, series, podcasts, videos, y más ha crecido exponencialmente. Muchas de estas plataformas se manejan en su funcionamiento básico a través de una arquitectura basada en objetos. Este concepto es ampliamente utilizado, y es mejor conocido como la Programación Orientada a Objetos. El construir código bajo este esquema proporciona ventajas al manipular datos que tienen características que pueden ser percibidas como objetos, y a la vez planear la interacción que se tiene entre estos para llevar a cabo un funcionamiento, en este caso para visualizar las opciones de series y películas en una base de datos. Los cuatro pilares que sostienen el desarrollo de sistemas basados en la POO son el *encapsulamiento, polimorfismo, herencia,* y *abstracción.* Por lo tanto, se busca diseñar un programa en C + + que pueda replicar a cierto alcance el funcionamiento de estos softwares, y aplicar los conceptos de la programación orientada a objetos en un ejemplo de la industria del servicio de streaming.

Diseño y planeación del Programa

Primer Modelo:



Segundo Modelo:



Ejemplos de Ejecución

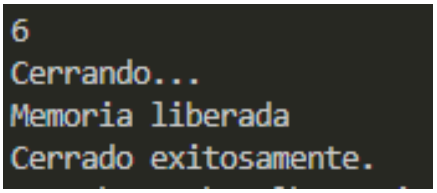
```
1
Cargando archivo...
Archivo abierto exitosamente
Videos agregado exitosamente
Total de videos: 320
1. Cargar archivo de datos
2. Filtrar videos (calificacion / genero)
3. Filtrar episodios de una serie. (calificacion)
4. Filtrar peliculas. (calificacion)
5. Calificar un video
6. Salir
```

```
ID: 308
Titulo: Trivia
Genero: Comedia
Duracion: 22
Calificacion: 4
Serie: The_Office
Numero de episodio: 11
Calificacion: 4
Serie: The_Office
Numero de episodio: 26
Numero de temporada: 6

Total de videos: 125
1. Cargar archivo de datos
2. Filtrar videos (calificacion / genero)
3. Filtrar episodios de una serie. (calificacion)
4. Filtrar peliculas. (calificacion)
5. Calificar un video
6. Salir
```

```
Filtrando...
Introduzca el nombre de la serie
Breaking_Bad
Introduzca la calificacion
1
ID: 6
Titulo: Abiquiu
Genero: Drama
Duracion: 47
Calificacion: 1
Serie: Breaking_Bad
Numero de episodio: 11
Numero de temporada: 3

Total de episodios: 1
1. Cargar archivo de datos
2. Filtrar videos (calificacion / genero)
3. Filtrar episodios de una serie. (calificacion)
4. Filtrar peliculas. (calificacion)
5. Calificar un video
6. Salir
```

```
4
Filtrando...
Introduzca la calificacion
1
Total de peliculas: 0
1. Cargar archivo de datos
2. Filtrar videos (calificacion / genero)
3. Filtrar episodios de una serie. (calificacion)
4. Filtrar peliculas. (calificacion)
5. Calificar un video
6. Salir
```

```
hola
10
No se encontro el video, verifica nuevamente los parametros.
1. Cargar archivo de datos
2. Filtrar videos (calificacion / genero)
3. Filtrar episodios de una serie. (calificacion)
4. Filtrar peliculas. (calificacion)
5. Calificar un video
6. Salir
```



Argumentación

a) (10 pts) Se identifican de manera correcta las clases a utilizar

Observando las necesidades del usuario al visualizar la base de datos, uno primero debe comprender los datos que se pueden mostrar, como son encapsulados estos, y cómo se relacionan estos unos entre otros. Al diseñar el programa, es evidente que dos objetos están presentes: *Película* y *Episodio*. Estos dos objetos son similares en algunos aspectos, como el hecho de tener los atributos de un ID, nombre, género, duración y calificación. Sin embargo, un episodio es un poco más complejo ya que también debe incluir información sobre la temporada, número de episodios y la serie a la que pertenece. Por lo tanto, es útil diseñar un tercer objeto abstracto como base para que estas dos puedan heredar una mayoría de los atributos y métodos más importantes. Esta clase se denominó como *Video,* y es la clase madre de ambas.

b) (12 pts) Se emplea de manera correcta el concepto de Herencia

Se puede observar como la herencia es implementada en el código desde la inicialización de los atributos de *Película* y *Episodio* respecto a *Vídeo.* Al definir el archivo header de éstas, se menciona el comando #include “video.h”. Desde aquí se aplica el fundamento de la herencia, ya que el objeto base tiene que existir en el entorno donde se define una clase hija. Consecuentemente, al crear la estructura de la clase, se inicializa de la forma class Pelicula : public Video {}. Esta estructura es precisamente lo que definirá a ésta clase como una clase hija de Video, y de esta forma se heredan los atributos y métodos previamente definidos en la clase madre.

c) (10 pts) Se emplea de manera correcta los modificadores de acceso

En cuanto al encapsulamiento y los diferentes tipos de acceso a los atributos de cada uno de estas clases, se decide utilizar la keyword *private* sobre *protected*. Se tomó esta decisión ya que el funcionamiento del programa no es del todo distinto, con la unica diferencia de que las clases hijas tienen acceso directo a los atributos de la clase madre. Sin embargo, por el esquema de programación que decidí seguir, es más legible para mi el desarrollo del código utilizando únicamente datos privados para las clases, y acceder estos a través de los getters y setters. En cuanto al código main.cpp, se hace una excepción y se decide tener algunas variables globales, debido a que para los propósitos de este programa, no perjudica

el funcionamiento ni la privacidad de los datos el tener variables listas con los apuntadores de las películas y episodios de la base de datos.

d) (12 pts) Se emplea de manera correcta la sobrecarga y sobreescritura de métodos

La clase abstracta Video contiene un método puro y virtual con el nombre de *muestra().* Este método tiene como fin mostrar aquellos datos más relevantes de la clase. Sin embargo, los datos mostrados para *Película* tienen que ser diferentes, o al menos no iguales, a los datos mostrados para un *Episodio*. Por lo tanto, se aplica la sobrecarga y sobreescritura del método *muestra(),* con el fin de adaptarlo a lo que cada clase necesite. De esta forma, se define un solo método con dos diferentes funcionamientos según la clase que llame a este método.

e) (12 pts) Se emplea de manera correcta el concepto de Polimorfismo

El uso del polimorfismo es fundamental para el funcionamiento del código, ya que al leer el archivo con los datos del texto de ejemplo, estos pueden tener objetos ya sean de tipo *Episodio* o *Pelicula*. Por lo tanto, se creó una lista de punteros de tipo *Video*, en donde se guardaron las direcciones de memoria de variables dinámicas creadas al leer una línea del archivo de prueba. De esta forma, se pudo facilitar la forma de filtrar los archivos de estos tipos sin tener que determinar su clase. Aplicando el polimorfismo, se pudo hacer ejecución del código video -> muestra(), que mostraba la información relevante según el tipo de dato para cada uno de los punteros en la lista. Esta tarea hubiera sido mucho más complicada sin la implementación de los punteros y el polimorfismo.

f) (12 pts) Se emplea de manera correcta el concepto de Clases Abstractas

Una clase abstracta consiste en una clase que sirve como modelo para otras clases hijas. Entrando en mayor detalle técnico, la clase debe tener al menos una función de tipo puro y virtual. La clase *Video* es la clase madre de Película y Episodio, y específicamente contiene un método puro virtual con el nombre de *muestra()*. Esto significa que el método no puede ser utilizado directamente al hacer una instancia de *Video*, sino que solo se podrá utilizar para las clases hijas que hayan sobrescrito la definición de este método en su propia clase. Ambas clases hijas implementan su propia definición de *muestra().*

g) (12 pts) Se sobrecarga al menos un operador en el conjunto de clases propuestas

Al momento de diseñar el código, no era evidente el que se podía hacer una sobrecarga del operador de << para imprimir todos los datos relevantes de cada clase. Por lo tanto, eso se logró utilizando una función virtual pura llamada *muestra().* Para mostrar el uso de sobrecarga de operadores, se sobrecargaron < y >, ya que pensé que sería bueno saber cómo se compara la calificación de un video

a comparación de otro. En el código se crearon ambos métodos sobrecargando los operadores, y por lo tanto el diseño de menú en el software contiene 7 opciones, y no 6. Esta sobrecarga solo es necesaria para la clase *Video*, ya que con los punteros solo es necesario comparar los atributos de calificación.

h) (opcional) Se utiliza de manera correcta el uso de excepciones tanto predefinidas como definidas por el programador: NA

Casos de Error

A lo largo del proyecto, buscó evitar que el usuario rompiera el código al evitar guardar un tipo de dato incorrecto en una variable de otro tipo. Un ejemplo muy prominente de esto es cuando se le pregunta al usuario que calificación busca en la plataforma, en donde accidentalmente se puede ingresar un carácter y por ende haciendo que el programa se cicle. Este ciclo no se detendrá hasta que se acabe la memoria alocada, entonces es crucial prevenir este tipo de errores. Se implementó un template en donde cada vez que se ingrese un tipo de dato que no era el esperado, se vuelve a pedir al usuario que ingrese un tipo de dato que sí corresponda al de la variable en donde se guardará. De esta forma, se evitan ciclos infinitos y se optimiza el funcionamiento del programa.

Conclusión

Este proyecto ha sido una oportunidad valiosa para combinar una gran variedad de los conceptos que conforman a C + +, la programación orientada a objetos, y las diferentes estrategias para trabajar con datos en los lenguajes centrados en este tipo de arquitectura. Considero que el código hace un buen trabajo para modelar la situación problema, considerando que el uso de punteros facilita mucho la manipulación de los datos y el acceso a aquellos atributos más importantes de los objetos. Así mismo, considero que el programa era capaz y no requería del todo la sobrecarga de operadores, pero aún así es muy interesante aprender como se puede adaptar el lenguaje de programación a las necesidades del software y del programador.