Situación Problema. Modelado de Streaming

Los Hackers

Junio 2022

Instituto Tecnológico y de Estudios Superiores de Monterrey

Departamento de Ciencias Computacionales/de Información

Programación Orientada a Objetos

**Indice** Página

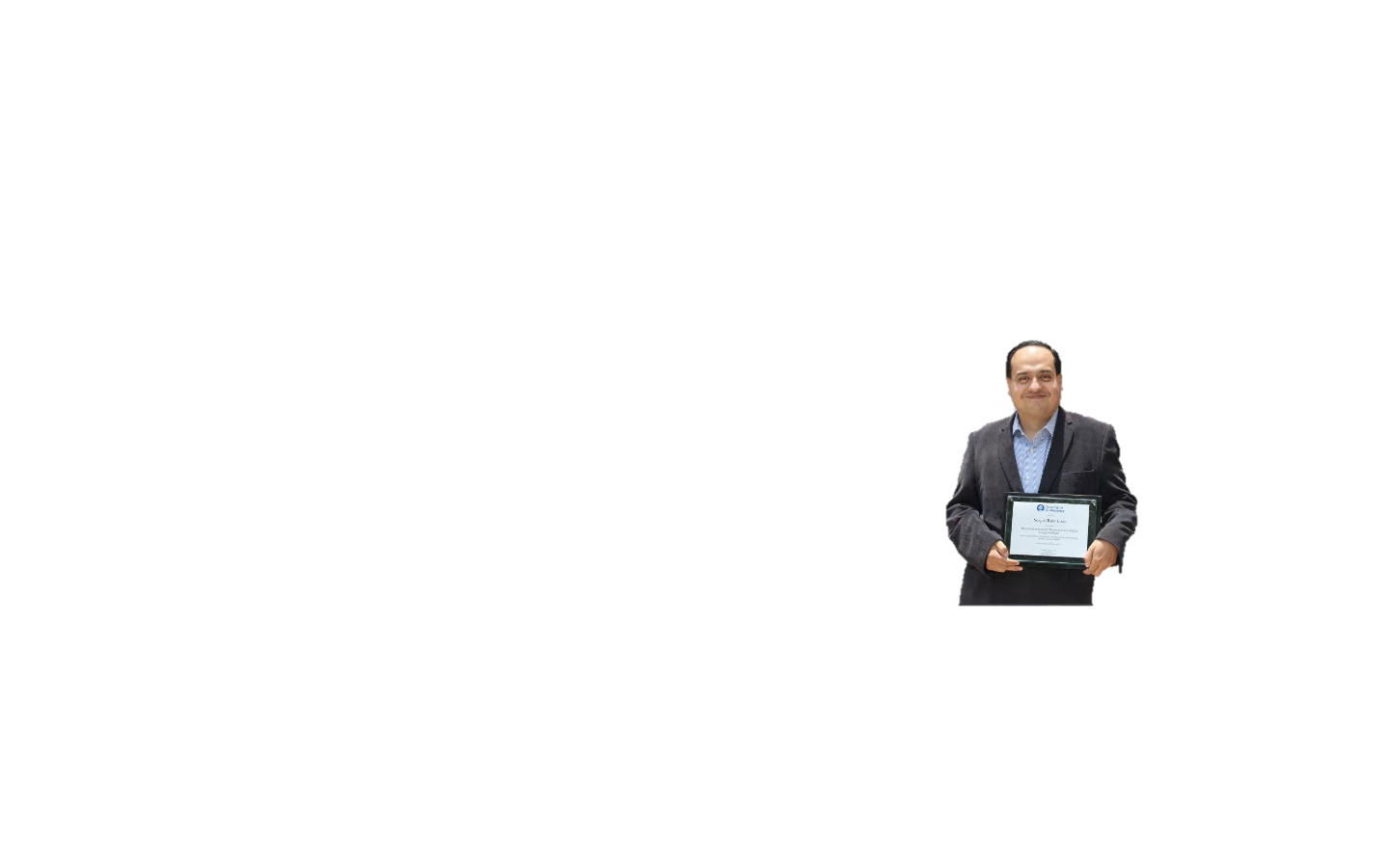
1. Introducción 3
2. Diagrama de clases UML 4 - 5
3. Ejemplo de ejecución 5 - 8
4. Argumentación 8 - 10
5. Conclusiones 10 - 11
6. Referencias 11

**Integrantes del equipo ‘Los Hackers’**

Ricardo Campos Luna                A01656898 @AntonioLaurance  
Moisés Adame Aguilar   A01660927 @MoisesAdame

Cristóbal Eleazar Meza Aranda A01661792 @eleazarmeza

**Enlace al repositorio del proyecto:** <https://github.com/AntonioLaurance/Modelado-de-Streaming>

****

**Profesor:** Sergio Ruiz Loza ([sergio.ruiz.loza@tec.mx](mailto:sergio.ruiz.loza@tec.mx)) (@sruizl)

**Introducción**

En este proyecto se modeló un servicio de Streaming a través de un programa mediante un paradigma de la programación imperativa que es el paradigma de la programación orientada a objectos (POO). Este mencionado paradigma tiene 4 técnicas que son:

* Polimorfismo
  + Consiste en el procesamiento diferentes de señales sintácticamente iguales dependiendo del contexto.
* Encapsulamiento
  + Consiste en el ocultamiento de variables, en la programación orientada a objetos se hace con los modificadores de acceso.
* Herencia
  + Es un tipo relación entre clases específicamente la relación es-un, en este tipo de relación hay una (herencia simple) o varias clases base (herencia múltiple) y una (herencia simple) o varias clases derivadas (herencia jerárquica). La clase derivada tiene todos los atributos y métodos de la clase padre.
* Abstracción
  + Es la capacidad de modelar clases y objetos a través de ejemplos de la vida real.

|  |  |  |  |
| --- | --- | --- | --- |
| Especificador de acceso del miembro de la clase base | Tipo de Herencia | | |
| Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | No accesible | No accesible | No accesible |

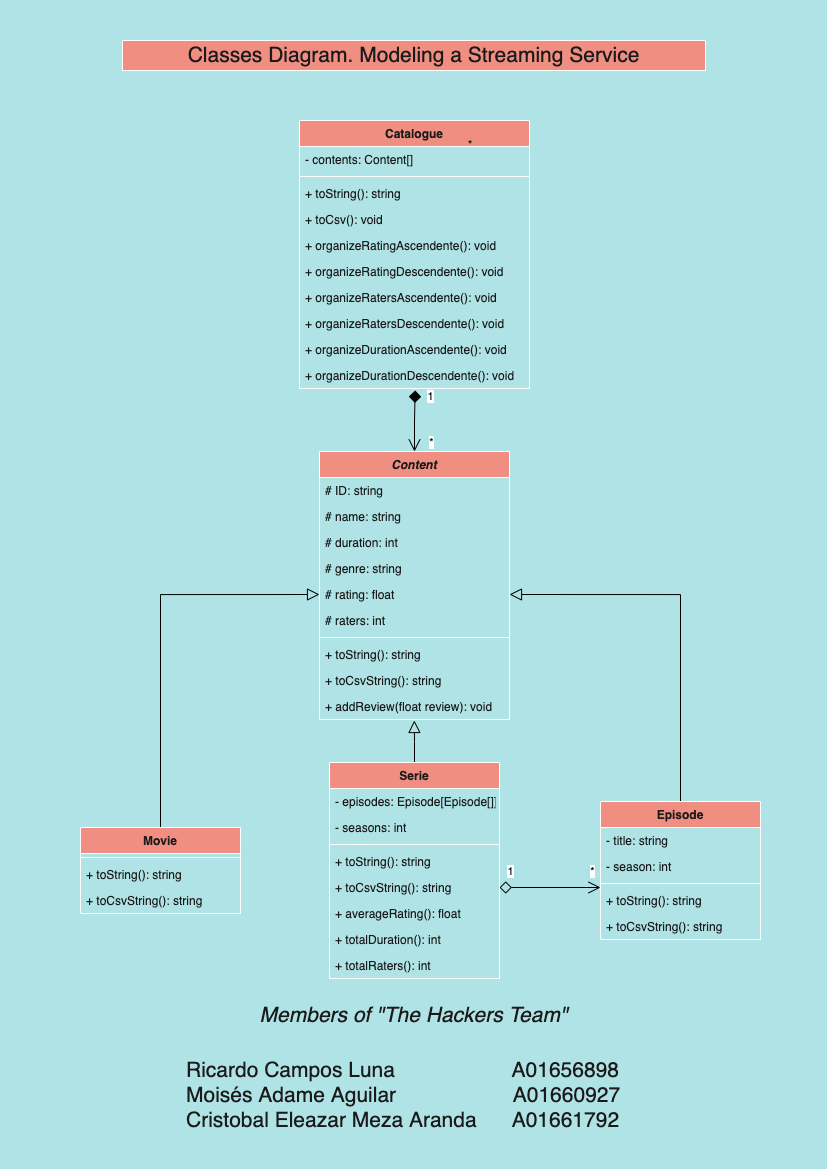
El concepto principal de la programación orientada a objetos (POO) es la *clase* que es un tipo de dato abstracto o un tipo de dato definido por el usuario (en este caso definido por el programador), que tiene atributos y métodos. Las clases pueden ser de 2 tipos que son:

* Concretas
  + Este tipo de clase puede ser instanciado
* Abstractas
  + Este tipo de clases no pueden ser instanciadas directamente, se forman con métodos virtuales puros generalmente se usan para heredar a clases derivadas concretas.

El lenguaje de programación usado para este proyecto es C++ y una de las mejoras con respecto a C es que permite usar este paradigma de la programación mencionado anteriormente. (Harsh Agarwal, 2022).

La situación problema a resolver es la modelación de un servicio de streaming, mediante la programación orientada a objetos (POO) con un programa extraiga datos de una base de datos relacional que muestre los videos en general con sus calificaciones y permita al usuario acceder y clasificar a los contenidos según los múltiples parámetros con los que se definen, ahora bien además de esto, el usuario es capaz de calificar los diferentes títulos, hecho que se guarda en la base de datos para su consulta *a posteriori.* Algo a mencionar en este proyecto fue que se utilizó herencia jerárquica.

**Diagrama de clases UML**



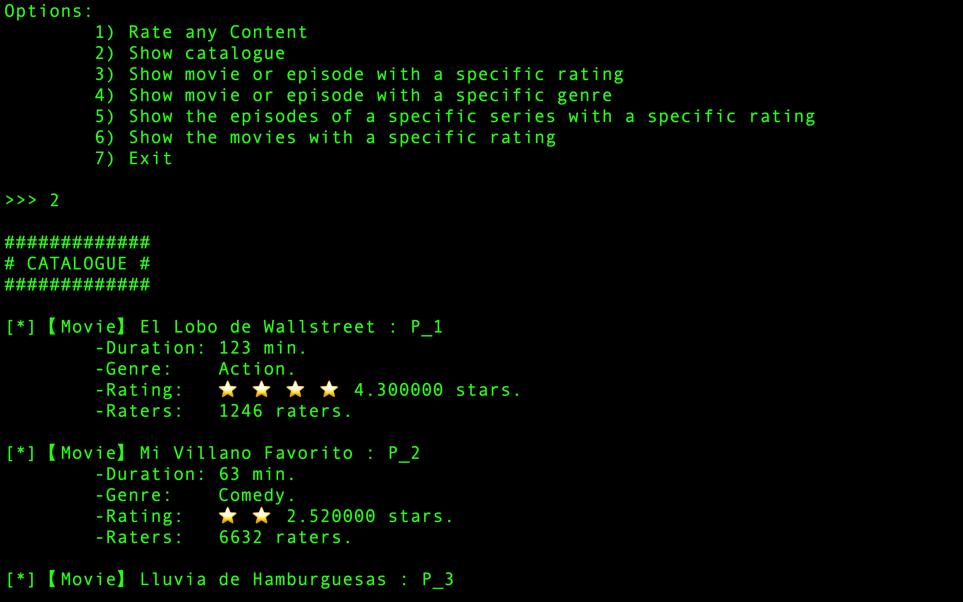
**Figura 1.** Diagrama de UML de clases

El diseño planteado en este proyecto resultó del análisis cuidadoso del problema que se tenía que resolver. En esencia, se planteó inicialmente una clase general capaz de modelar cualquier tipo de contenido a la cual se nombró Content y a la que se le brindaron atributos o datos miembro muy generales y aplicables para uno o un conjunto de contenidos (como en el caso de Serie, donde se hace uso de los atributos rating y raters respectivamente, para promediar los ratings de todos los episodios y para almacenar la cantidad total de raters. Por su parte, en cuanto a las funciones miembro o métodos se definieron tres esenciales, las cuales, de igual manera aplican para cualquier tipo objeto que pueda ser definido como un contenido, en este caso, toString() para imprimir los datos del contenido en la pantalla del usuario, toCsvString() para formatear cualquier objeto derivado de la clase Content para su almacenamiento en un archivo csv (coma separated values por sus siglas en Inglés) y finalmente, addReview() para calificar los contenidos (alterando tanto su rating como su número de raters). Algo a puntualizar de esta clase es que debido a que es muy general se definió como clase abstracta al establecer los métodos toString() y toCsvString() como funciones virtuales puras.

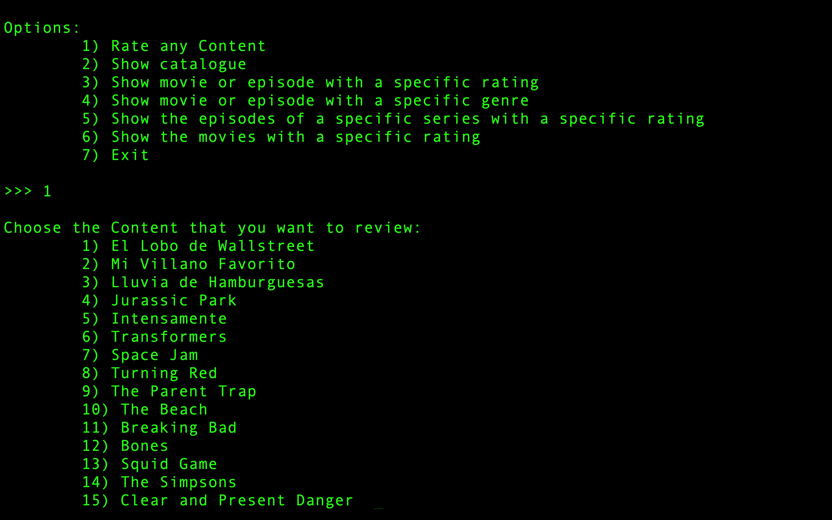
Por otro lado, se generaron tres clases derivadas de la clase base Content las cuales fueron Movie, Serie y Episode. Movie, conservó un diseño casi idéntico a Content, los únicos cambios realizados fueron en los métodos toString() y toCsvString() en los cuales debido al polimorfismo se presentaron características únicas que resultan visibles ya sea al imprimir en la pantalla una instancia de Movie o al almacenarlas en un csv, siendo esto último crucial a la hora de la instanciación no solo de los objetos de tipo Movie pero también del objeto Catalogue del cual se hablará más adelante. Por su parte, Episode se diseñó como una clase aparte a Serie con el fin de poder aplicar polimorfismo a cada uno de los episodios de una serie, los cuales, como es posible predecir, se pueden definir a sabiendas de su título, así como de la temporada a la que pertenecen, lo cual, nos obligó a añadir estos dos atributos al diseño de la clase Episode. Finalmente, con la intención de poder manipular y almacenar de forma real los episodios de una serie se definió la clase Serie, la cual, posee como atributos privados una matriz la cual se conforma por vectores de Episodios, de tal forma que una matriz funja como la serie, un vector como la temporada y cada elemento del vector como un Episodio de dicha temporada; adicionalmente, añadiendo el número de temporadas de la serie. Ahora bien, al hablar de las funciones miembro establecidas en esta clase primero, se definieron de forma obligatoria (debido a que son virtuales en Content) los métodos toString() y toCsvString(). Estos, hacen uso de los mismos métodos, pero de la clase Episode para así conglomerar todas las series en un mismo output ya sea al imprimir en la pantalla o al guardar los datos en un csv, lo cual, permite manejar todo como un conjunto (encapsulamiento) sin tener el riesgo de tener pérdidas de información. Finalmente, algo a remarcar es que la relación que existe entre Serie y Episode es de agregación ya que únicamente teniendo completos y bien definidos todos los episodios de una serie, podemos definir esta última.

Finalmente, el último objeto definido en este programa fue Catalogue, este, posee una relación de composición con cualquier objeto derivado de la clase Content, lo cual, se debe a que este lee un archivo csv y según la información que encuentre, instancia objetos de tipo Movie, Serie y Episode los cuales, almacena en el único atributo que definimos para Catalogue, siendo este, contents un vector de objetos derivados de la clase base Content. Además aunque no resultaba obligatorio se definieron las funciones miembro toString(), la cual, permite ver todas las películas, series y episodios de las series presentes en la base de datos así como la información clave de cada uno de estos objetos; además se definió la función toCsv(), la cual, se encarga de conglomerar toda la información de todos los objetos pertenecientes al catálogo y pasarlos a un archivo csv con el fin de actualizar los cambios (si es que se llegaron a hacer) en alguno de los múltiples datos de cada objeto (un ejemplo de esto, es cuando se recalcula el promedio de *rating* de una serie y se aumenta el número de *raters* así como el *rating* de una serie, película o episodio).

**Ejemplo de ejecución**



**Figura 2.** Impresión de todo el catálogo



**Figura 3.** Selección de Opción 1, calificar cualquier contenido

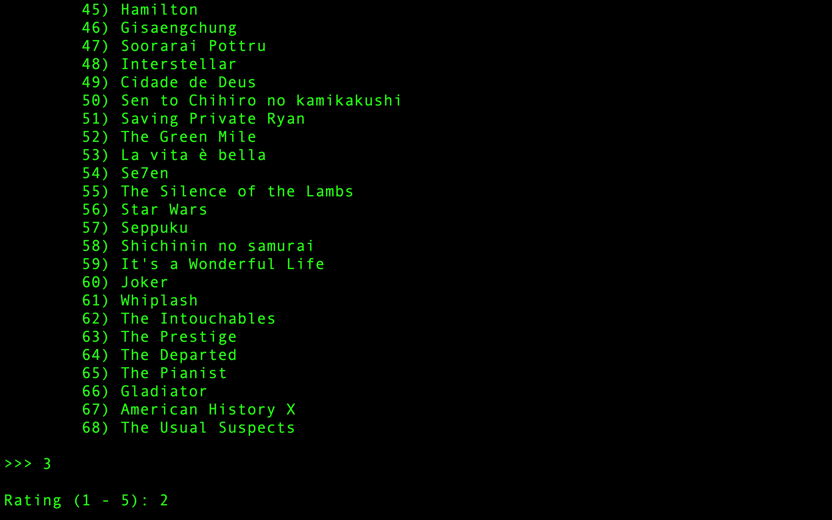
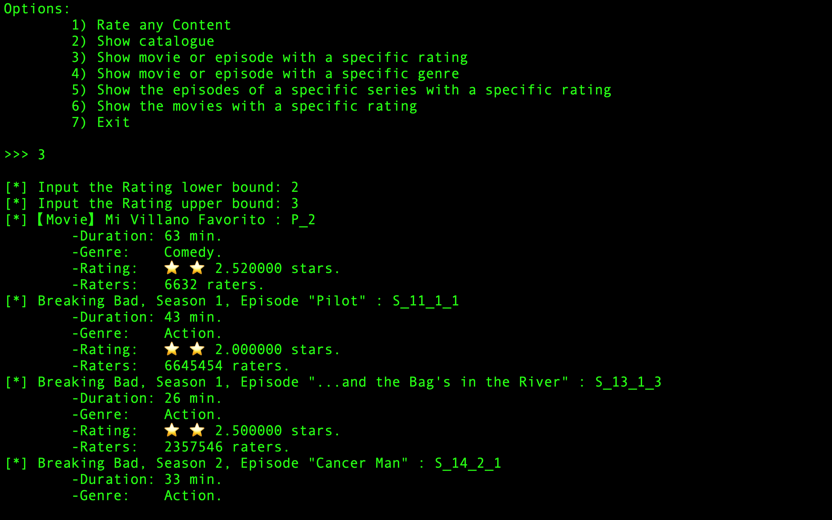
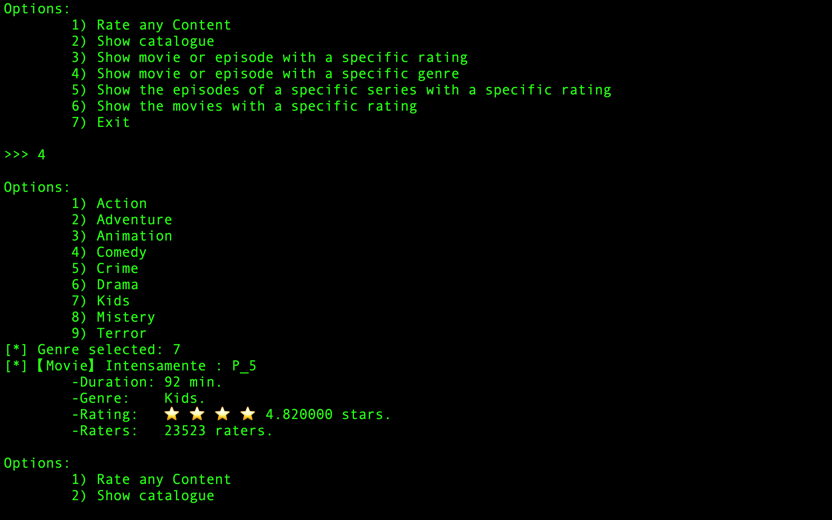


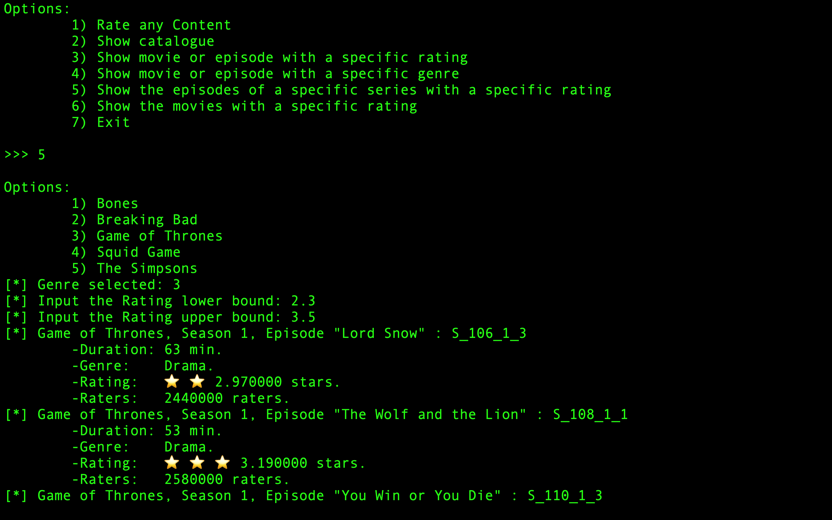
Figura 4. Asignación de la calificación 2 al contenido número 3



**Figura 5.** Impresión de contenidos con ratings mayores a 2 y menores a 3

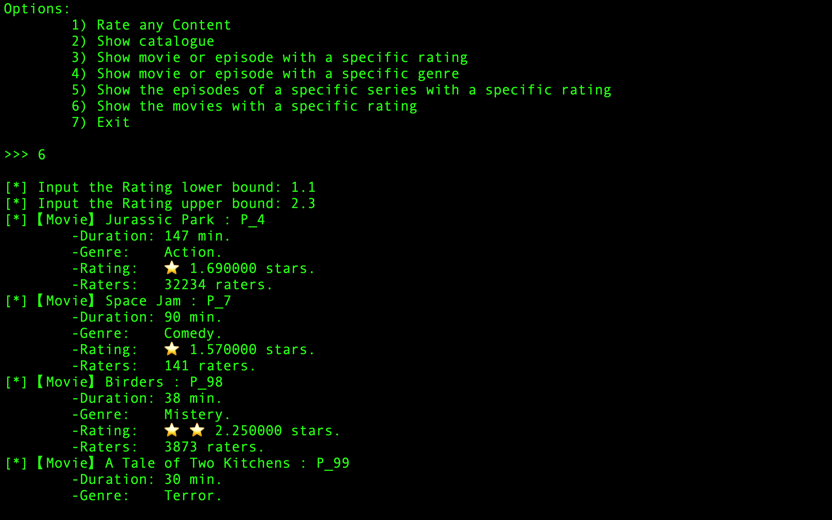


**Figura 6.** Impresión de todos los contenidos de género “Kids”

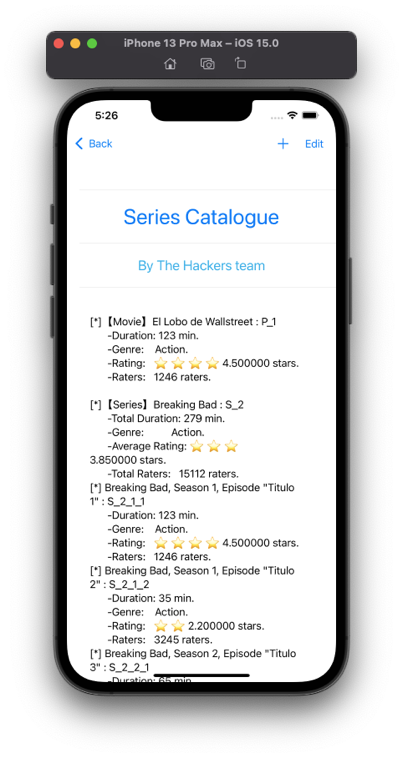
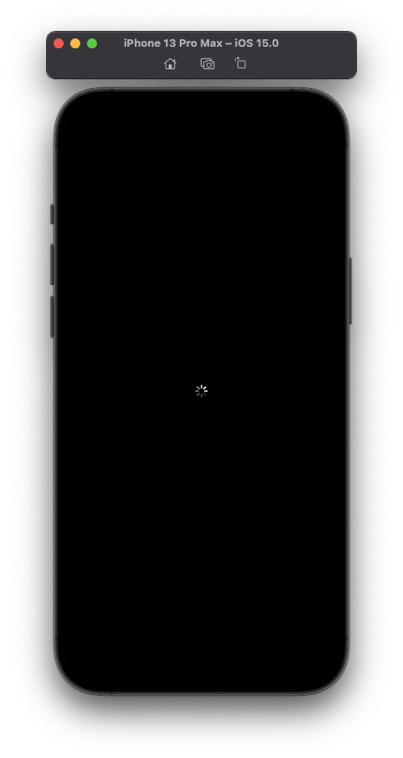


**Figura 7**. Impresión de todos los episodios de la serie “Game of

Thrones” con ratings mayores a 2.3 y menores a 3.5



**Figura 8.** Impresión de todas las películas con un rating mayor a 1.1 y menor a 2.3



**Figura 9.** Interfaz de Usuario implementada en un ambiente iOS

**Argumentación**

**Ricardo:**

1. La herencia de las clases se utilizó para aplicar el polimorfismo esto para procesar funciones como ya se mencionó toString() y toCsvString() para facilitar la escritura del código y ahorrarnos unas condicionales (*if’s*) y la verificación de tipo de dato (simplificar el código), también porque cumplen directamente la relación es-un (is-a)
2. Se declararon *protected* los atributos de la clase base de *Movie*, *Serie* y *Episode* para poder acceder a estos elementos directamente desde una de estas clases derivadas pero sin que nada fuera de ellas pueda acceder a estas variables directamente, esto debido a que es más rápido acceder a las variables directamente que mediante *getters y setters*.
3. En las clases derivadas de *Content* y en la clase *Catalogue* se definieron sus atributos mediante el modificador de acceso *private* esto para que nada fuera de ellas pueda acceder a sus atributos directamente esto con el propósito de definir funciones *getters* y *setters* para tener más control sobre la forma en que se accede a un atributo (por ejemplo a través de estos métodos para acceder podemos querer que el atributo no toque un conjunto específico de valores y a través de if’s en este tipo de funciones se puede lograr).
4. *Content* se hizo abstracto porque todo contenido tiene *raters*, *rating* y se le puede pedir la opinión al usuario si le gustó un tipo de contenido sin embargo no hay forma de implementar los métodos toString() y toCsvString() de manera general para Movie, Serie y Episode a pesar de que todas estas clases tengan estos métodos por lo tanto es útil definir estos 2 métodos mencionados anteriormente como virtuales puros.
5. Siempre que había composición entre clases como la clase Serie que tiene episodios y la clase Episode que está compuesta por algún tipo de contenido se usaron punteros esto ahorrarle un paso a la computadora en el procesamiento y aumentar la velocidad de nuestro programa.
6. Se utilizaron los archivos de cabecera .h y no los .hpp para respetar el estándar
7. No se definieron los métodos de cada clase en el archivo donde se definió la misma clase, en el archivo donde se definió la clase solamente se definieron los prototipos de los métodos esto para facilitar la lectura de que métodos y atributos tiene cada clase.
8. Se implemento el símbolo de *emojí* de estrella antes de mostrar el *rating* para dar un mayor apoyo visual a nuestro programa ya que muchos servicios de *streaming* muestran el rating de su contenido visualmente complementado con números.

**Moisés:**

1. Como se muestra en el diagrama UML se lograron identificar de forma correcta las clases a utilizar. Esta afirmación, lo sustento tomando en cuenta que durante la herencia no existieron ni métodos ni atributos que se dejaran vacíos. Además, se definieron el número de clases necesarias con las relaciones que necesitaban, lo cual, se puede ver reflejado en la facilidad de manipulación y cercanía al mundo real que estas clases emiten.
2. Como previamente se mencionó el concepto de herencia se utiliza de forma correcta, lo cual, podemos sustentar en que como ya hicimos mención, no sobran n se dejan sin uso atributos o métodos heredados por parte de la clase base (Content) a las clases derivadas (Movie, Serie, Episode).
3. Como es posible percibir a través del diagrama de clases se hizo uso de los tres principales modificadores de acceso (private, protected y public). El uso de estos fue correcto debido a varios puntos. Primero, debido a que se utilizó protected en los atributos de la clase base, los cuales, al ser heredados de forma pública siguen siendo accesibles (cosa que no hubiera sido así si estos atributos se hubieran definido como private). Segundo, debido a que las funciones getters, setters, constructoras, destructoras y adicionales se definieron como públicas, lo cual, resulta ser importante para la abstracción y el encapsulamiento, ya que permitimos únicamente el acceso, pero no la modificación de los atributos y además, permitimos que se pueda hacer uso de cualquier método perteneciente a los objetos sin temor de que este no sea accesible. Finalmente, se definieron como privados los atributos de las clases que no van a heredar, lo cual, permite guardar de forma segura la información Sin temer sobre su acceso (el cual no es motivo de preocupación debido a que no van a heredar).
4. A pesar de que no hubo sobrecarga de métodos, fue abundante la sobreescritura de estos, lo cual se dio básicamente en cualquier parte del programa. Un ejemplo de ello es cuando se dio específicamente en el contexto de imprimir datos en la pantalla, donde, para poder imprimir la información de las series, películas o episodios se hizo uso de la siguiente línea de texto (obj -> toString()), lo cual, por polimorfismo, involucró que los datos se imprimieran de una u otra manera según el tipo de objeto instanciado.
5. Habiendo mencionado lo anterior, es posible reconocer que el concepto de polimorfismo fue implementado de forma correcta. Otro ejemplo, es en el almacenamiento y la creación del archivo csv el cual, utilizando la función toCsvString() le daba un formato peculiar a cada objeto. Además, otro ejemplo de polimorfismo es el que se hizo en la sobrecarga del operador suma (+) el cual, al ser utilizado sobre dos objetos derivados de la clase base Content no realiza una suma aritmética, sino, más bien, regresa un objeto derivado de la clase Content cuyos atributos fueron modificados de tal manera que su rating, sus raters y su duración son la suma de los ratings, los raters y las duraciones de los objetos sobre los cuales se aplicó este operador.
6. Como fue mencionado en un inicio en la sección diagrama de UML, se definió a la función miembro toCsvString() perteneciente a la clase Content como una función pura virtual. Esto, por consiguiente, provocó que la clase Content se convirtiera en una clase abstracta, lo cual, fue implementado a propósito ya que es una clase muy general la cual, no tiene sentido instanciar, lo cual, por supuesto nunca se hizo ya que de lo contrario compilar y correr el programa hubiera sido una tarea imposible.
7. Como ya se hizo mención al hablar de polimorfismo, se sobrecargó el operador + en la clase Content de tal manera que al aplicarlo sobre dos objetos derivados de la clase base Content se regrese otro objeto de igual manera derivado de Content con atributos alterados de tal forma que sus ratings, *raters* y duración son la suma de estos atributos encada objeto. Además, su implementación fue útil a la hora de calcular los valores promedio de rating en las series así como los valores totales de *raters* y duración.
8. Finalmente, se utilizaron excepciones para controlar los inputs del usuario, si el tipo de dato era incorrecto, se pedía al usuario hacer su input hasta que este fuera correcto.

**Cristóbal:**

Se comenzó realizando la planeación del proyecto a través del diagrama UML, enunciando esencialmente las clases, identificando aquellas que fueran necesarias, para así definir los métodos y sus atributos, en los cuales se incorporara los temas relevantes del curso como Herencia, Polimorfismo y Sobrecarga de operadores.

Fue relevante que desde el comienzo se comprendiera que se crearía una clase Padre, la cual heredó a Movie, Serie y Episode, aún a pesar de que podía utilizarse una opción diferente, por ejemplo una en la que Serie y Episode estuvieran relacionadas dejando de lado a Movie.

Se utilizaron así mismo los usos básicos de una clase:

* + Main como receptor y “empleador” de todas ellas, bajo código escaso
  + Clases divididas en H y C.
  + H para declarar atributos y métodos (incluidos los heredables)
  + C para definir métodos

Tomando en cuenta así que en H se dividió cada clase de manera estándar, atributos no heredables en private, atributos heredables en protected, métodos en public, delcaración de Getters y Setters, así como destructores.

El programa culmina utilizando los métodos requeridos, mostrando series y películas con sus respectivos datos así como leyendo y alimentando la base CSV.

Como detalle adicional se buscó mejorar la presentación del programa a través de un APP de IOS, misma que vale mencionar es más que un emulador, puesto que está directamente en Xcode y consume código en directo de C++, convertido a través de un “Bridging” que traduce y envía la información de C++ a través de Objective C, para finalmente usar código de Swift y mostrarlo en la APP, dejándola prácticamente lista para lo que nos hubiera gustado que es pasar la base a *Firebase* y utilizar un JSON para alimentarla online.

**¿Cuándo pude fallar el programa?**

Según las múltiples pruebas realizadas el programa es incapaz de fallar debido a un input erróneo del usuario debido a que cada input fue validado en cuanto a dos aspectos; primero, tipo de dato, y segundo, rango de valor. Si el dato el cual se hizo input no cumplía alguna de las características establecidas de rango y de tipo de dato, este se le volvía a pedir al usuario. Ahora bien, el programa puede a llegar a fallar si se utiliza alguna base de datos mal llenada, es decir, que tenga los episodios de una misma serie en renglones no contiguos o que se pongan más o menos de los datos solicitados.

**Conclusión**

**Ricardo:** El paradigma de la programación orientada a objetos nos fue útil para evitar condicionales y verificación de tipo de variables todo eso gracias a su técnica de polimorfismo que para poder aplicarla reescribimos algunos métodos en algunas clases. Con este paradigma es más fácil escalar una aplicación para largas repeticiones de código ayuda a reducir el tiempo de desarrollo. Este paradigma lo considero especial porque es programación donde creas tus propios tipos de datos, en sí es creativo y los tipos de datos que creas depende del contexto en lo que lo vas a utilizar. El paradigma se puede encontrar especialmente en el desarrollo de interfaces gráficas de usuario (GUI por sus siglas en inglés) ya que los botones tienen un prototipo, los campos de texto también tienen un prototipo hasta incluso el texto tiene un prototipo y muchas funciones relacionadas.

**Moisés:** Para mi, la programación orientada a objetos resulta un paradigma de programación muy sencillo de entender y de llevar a cabo, considero que es la manera más fácil de plasmar el mundo que nos rodea en código y tiene una gran cantidad de ventajas entre las cuales, resalta lo que en algunos algoritmos denominan “divide y vencerás”, forma de programar que a través de la simplificación de un gran problema en muchos pocos es capaz de llevarnos a resultados fáciles de obtener. Por su parte, otra de las ventajas de esta forma de programación es la compartimentación (encapsulamiento) donde podemos resguardar bajo distintas capas de seguridad la información que es riesgosa o los métodos que queremos mantener en secreto. Finalmente, este reto me resultó muy entretenido y lleno de conocimientos, ya que finalmente pude aplicar los conceptos más rebuscados de este paradigma y sacarles mucho provecho.

**Cristóbal:** La programación por objetos se puede considerar que nace en C++, a través del uso de clases y diagramado en UML, donde hasta llegado a H se puede considerar el diseño o trabajo de un arquitecto de Software, donde posteriormente la creación del código en los métodos y su implementación corresponde a la parte de desarrollo.

Con esto, la programación por objetos es un paso importante ya que permite crear conjuntos de código que generen objetos parecidos, así como que estos puedan ser heredables y usados ya sea como se definieron por la clase padre o como los definan las clases hijo, siendo así más eficiente que la programación estructurada.

**Referencias**

Anónimo. (s.f). *Situación Problema*. Recuperado de <https://experiencia21.tec.mx/courses/228970/pages/situacion-problema>

Harsh Agarwal. (2022). Inheritance in C++. Recuperado de <https://www.geeksforgeeks.org/inheritance-in-c/>

Instituto Tecnológico y de Estudios Superiores de Monterrey. (s.f). *Programación Orientada a Objetos.* Recuperado de <https://samp.itesm.mx/Materias/VistaPreliminarMateria?clave=TC1030&lang=ES>