Entrega de situación problema

Modelado de servicio de streaming

TC1030: Programación Orientada a Objetos

Prof. Fabiola Uribe Plata

Miguel Arriaga A01028570

2 de diciembre de 2020

Contenido

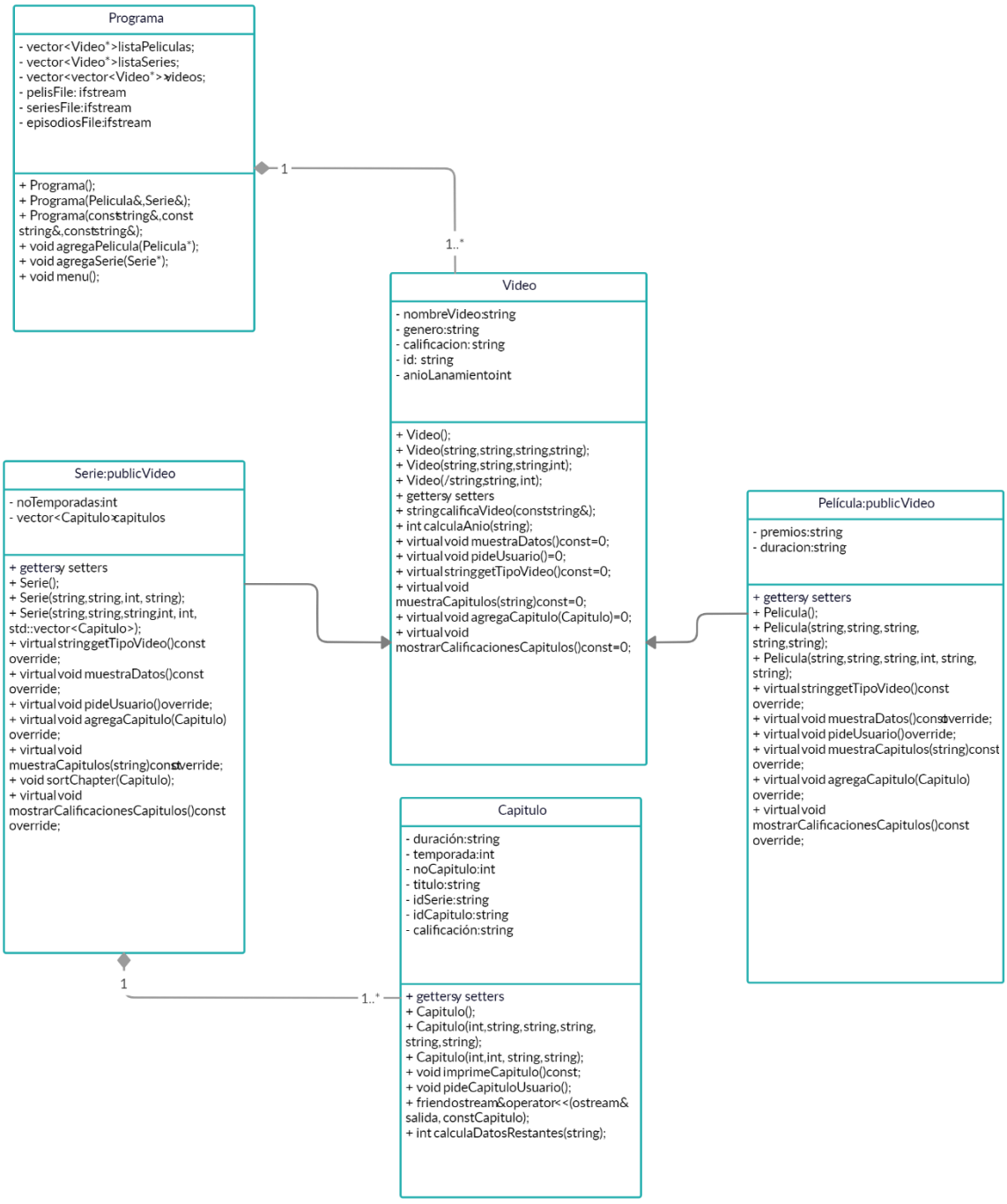
Introducción

La situación problema consiste en el modelado de un servicio de Streaming. “Los servicios de Streaming o retransmisión consisten en la distribución de datos desde un servidor a través de Internet para que los usuarios hagan uso de ellos sin esperar a que sean descargados totalmente. Es uno de los servicios de entretenimiento más utilizados en la actualidad”. Antes se utilizaban otros métodos para el entretenimiento como la televisión o los DvDs. Para esto modelar estas nuevas prácticas, se utilizó C++ y la Programación Orientada a Objetos.

Finalmente, se creó un programa con clases, herencia, polimorfismo, sobrecarga de operadores y manejo de excepciones, que puede leer desde archivos de texto, crear nuevas películas y series, ver los capítulos de cada serie y mostrar los videos a partir de ciertos filtros.

Diagrama de clases UML

Para facilitar la escritura del código, se creó un diagrama de clases UML, en dónde se especifican las clases, sus funciones y métodos. Además de la relación entre cada clase. Este diagrama se fue modificando conforme el programa se actualizaba.



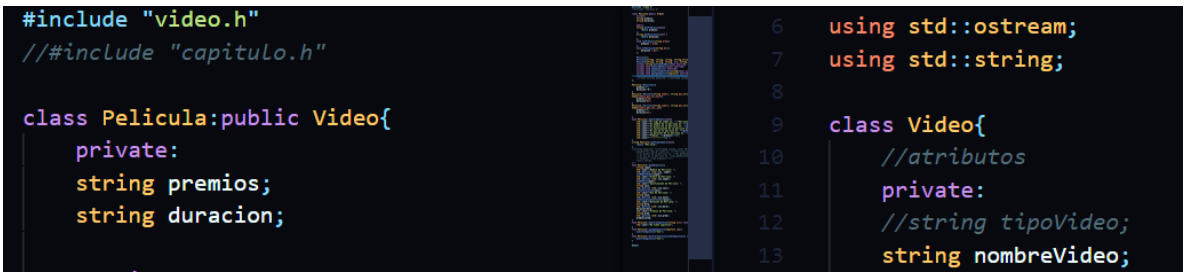
Ejemplo de ejecución

Al correr el programa, se le pregunta al usuario qué es lo que desea hacer y se le dan una serie de opciones. A partir de lo que el usuario vaya seleccionando, surgen nuevos menus con nuevas opciones. Esto se muestra a continuación, en una ejecución en la que el usuario selecciona la sección de “todos los videos” y luego selecciona la opción de verlos “por calificación”:

```
¿Qué deseas hacer?
(0) Todos los videos
(1) Sección películas
(2) Sección series
(3) Salir
0
(1) Orden predeteminado
(2) Por calificación
(3) Por género
(4) Regresar
2
Video :
El tipo de video es: Película
El nombre de la película es: Onward
El género de la película es: Accion
La calificación de la película es: 7.5
El año de la película es: 2020
La duración de la película es: 01:42
Premios:
------------
 Video :
El tipo de video es: Película
El nombre de la película es: Beauty and the Beast
El género de la película es: Accion
La calificación de la película es: 7.1
El año de la película es: 2017
La duración de la película es: 02:09
Premios:
------------
 Video :
El tipo de video es: Película
El nombre de la película es: The Platform
El género de la película es: Drama
La calificación de la película es: 7
El año de la película es: 2019
```

Herencia

La herencia es una de las funcionalidades más importantes de la Programación Orientada a Objetos, esta nos permite derivar clases se otras clases, de esta forma se puede reutilizar mucho código. En la situación problema, se utilizó herencia para las clases “película” y “serie”, que derivan de la clase base “video”. Esto se puede observar a continuación, en donde se ve que “película” es-un “video”:



```
#include "video.h"
//#include "capitulo.h"

class Pelicula:public Video{
    private:
    string premios;
    string duracion;
```

```
using std::ostream;
using std::string;

class Video{
    //atributos
    private:
    //string tipoVideo;
    string nombreVideo;
```

Polimorfismo

El polimorfismo aplica cuando una instancia de una clase, es decir, un objeto, se puede comportar de manera distinta. Esto ocurre cuando el objeto es una instancia de una clase derivada, ya que a su vez es una clase derivada y es una clase base.

En la situación problema, se utilizó el polimorfismo para que, en tiempo de ejecución, cada video se comporte ya sea como película o como serie. Para esto, se crearon varias clases virtuales puras en la clase video. De esta forma, en la clase programa, se crea un vector de apuntadores de Video, después de llenar este vector con series y videos de manera dinámica, las funciones virtuales (reescritas en las clases derivadas) se llevan a cabo de manera polimórfica.

Funciones virtual puras en clase Video:

```
//funciones virtual para polimorfismo
virtual void muestraDatos()const=0;
virtual void pideUsuario()=0;
virtual string getTipoVideo()const=0;
virtual void muestraCapitulos(string)const=0;
virtual void agregaCapitulo(Capitulo)=0;
virtual void mostrarCalificacionesCapitulos()const=0;
```

Ejemplo de implementación, aquí se lleva a cabo la función “muestraDatos()” de manera polimórfica, es decir, se comporta diferente para series y películas:

```
if (respuesta=="1"){
    for (int i = 0; i < videos[1].size(); i++)
    {
        std::cout<<"Video "<<i+1<<":\n";
        videos[1][i]->muestraDatos();
        //std::cout<<"Capitulos:\n";
        //videos[1][i]->muestraCapitulos();
    }
    std::cout<<"-------------\n ";
}
```

Al correrlo, se ve así:

```
------------
 Video :
El tipo de video es: Película
El nombre de la película es: Beauty and the Beast
El género de la película es: Accion
La calificación de la película es: 7.1
El año de la película es: 2017
La duración de la película es: 02:09
Premios:
------------
 Video :
El tipo de video es: Serie
El nombre de la serie es: Game of Thrones
El género de la serie es: Drama
La calificación de la serie es: Sin calificar
El año de la serie es: 2011
Número de temporadas: 8
Número de capítulos: 73
```

Sobrecarga de operadores

La sobrecarga de operadores es cuando se modifica la funcionalidad de alguno de los operadores existentes de C++, para realizar nuevas funciones o comportarse distinto.

En la situación problema, se utilizó la sobrecara de operadores con el fin de reducir el uso de std::cout y reutilizar código. Esta sobrecarga se hizo para el operador “<<” en la clase capítulo. De esta forma se puede imprimir un capitulo tan solo con el uso del operador.

Sobrecarga del operador “<<” en la clase capitulo:

```
//sobrecarga de operadores:
ostream &operator <<(ostream& salida, const Capitulo c1){
    salida<<"Nombre del capítulo: "<<c1.titulo<<"\n"
    <<"Temporada del capítulo: "<<c1.temporada<<"\n"
    <<"Número de capítulo: "<<c1.noCapitulo<<"\n"
    <<"Duración del capítulo: "<<c1.duracion<<"\n"
    <<"Calificacion del capítulo: "<<c1.calificacion<<"\n";
    return salida;
}
```

Implementación:

```
std::cout<<capitulos[i]<<std::endl; //sobrecaraga de operadores
```

Manejo de excepciones

El manejo de excepciones nos sirve para poder continuar un programa, a pesar de que existan ciertos errores. En la situación problema, se utilizó el manejo de excepciones para la opción que le permite al usuario mostrar los títulos de películas o capítulos con una cierta calificación (dada por el usuario). Ya que en la base de datos del programa no existen videos con todas las calificaciones posibles, se utilizó el manejo de excepciones para que el usuario solo introduzca valores de calificaciones posibles. Esto se realizó con un bloque try-catch, en el try se intenta mostrar los videos con esa calificación y en caso de fallar (catch), se muestra un mensaje de error al usuario y se le muestran las calificaciones que sí tienen títulos. Esto se hizo para tener una mejor interacción con el usuario y que no tenga que adivinar las calificaciones posibles.

Aquí se puede observar la implementación de esto en el programa, para las películas:

```
//manejo de excepciones
try{
    std::getline (std::cin,cali);
    for (int i = 0; i < videos[0].size(); i++){
        if (videos[0][i]->getCalificacion()==cali){
            videos[0][i]->muestraDatos();
            contador+=1;
        }
    };
    if (contador==0){
        throw std::out_of_range(cali + " inválido, no hay películas con esa calificación");
    }
}
catch (std::out_of_range &excp){
std::cout<<excp.what()<<std::endl;
vector<string> calificacionesTotales;
string actual;
    std::cout<<"Solo existen Películas con las siguientes calificaciones:"<<std::endl;
    for (int i = 0; i < videos[0].size(); i++){
        actual=videos[0][i]->getCalificacion();
        if(!(std::find(calificacionesTotales.begin(), calificacionesTotales.end(), actual)
            calificacionesTotales.push_back(actual);
        }
    }
    for (int i = 0; i < calificacionesTotales.size(); i++)
    {
        std::cout<<calificacionesTotales[i]<<". ";
    }

    std::cout<<std::endl;
}
```

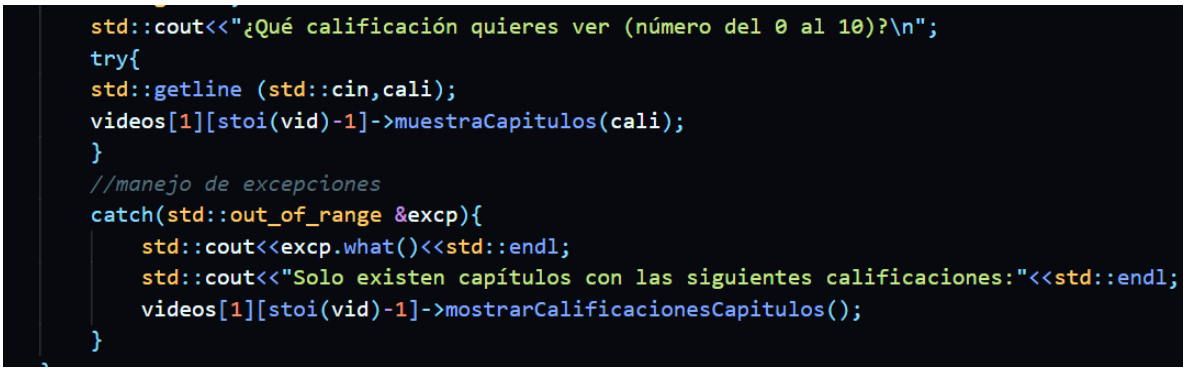
Aquí hubo una diferencia importante entre la excepción para las películas y para los capítulos. Ya que no se puede acceder directamente a las calificaciones de cada capitulo desde la clase “programa”,

para los capítulos se utilizó una función que lance la excepción en caso de que exista. Se crearon dos métodos virtuales, que se sobre escribieron en la clase serie. Uno de estos métodos intenta mostrar los capítulos y lanza una excepción en caso de que no haya capítulos con esa calificación. El otro método muestra las calificaciones que sí son válidas.

Estos métodos se muestran a continuación:

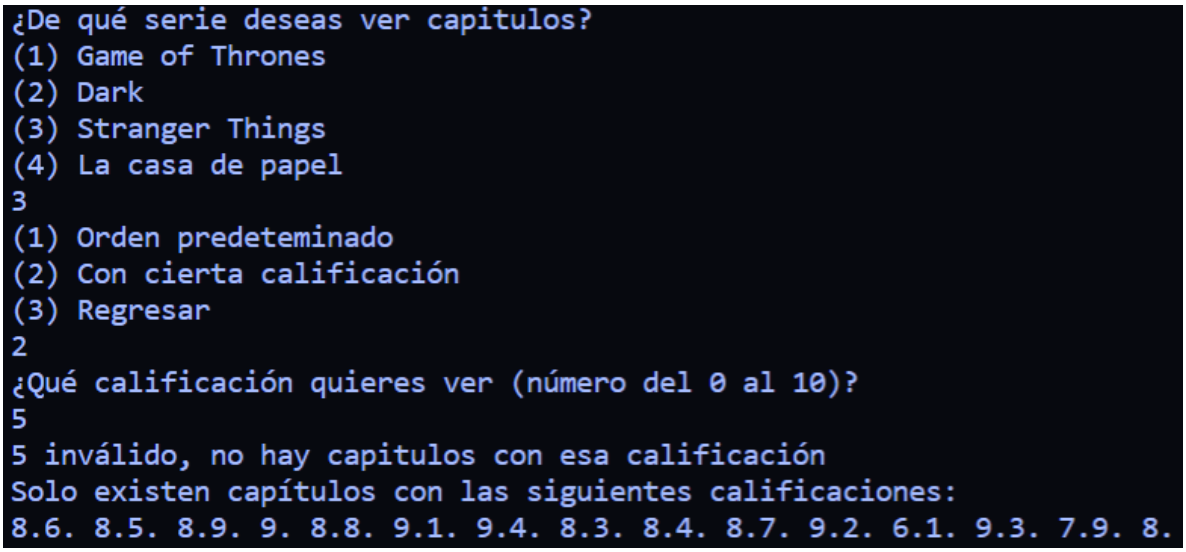
```
void Serie::muestraCapitulos(string cali)const{
    if (cali=="all"){
        for (int i = 0; i < capitulos.size(); i++){
            std::cout<<"Capitulo "<<i+1<<":\n";
            std::cout<<capitulos[i]<<std::endl; //sobrecaraga de operadores
            std::cout<<"------------\n";
            }
    }
    else{
        int cont=0;
         for (int i = 0; i < capitulos.size(); i++){
            if (capitulos[i].getCalificacion()==cali){
                std::cout<<"Capitulo:\n";
                std::cout<<capitulos[i]<<std::endl; //sobrecaraga de operadores
                std::cout<<"------------\n";
                cont+=1;
            }
        }
        //manejo de excepciones
        if (cont==0){
            throw std::out_of_range(cali + " inválido, no hay capitulos con esa calificación");
        }
    }
}
void::Serie::mostrarCalificacionesCapitulos()const{
    std::vector<string> calificacionesTotales;
     string actual;

            for (int i = 0; i < capitulos.size(); i++){
                actual=capitulos[i].getCalificacion();
                if(!(std::find(calificacionesTotales.begin(), calificacionesTotales.end(), actual) !=
```

Implementación en la clase Programa:



```
      std::cout<<"¿Qué calificación quieres ver (número del 0 al 10)?\n";
      try{
      std::getline (std::cin,cali);
      videos[1][stoi(vid)-1]->muestraCapitulos(cali);
      }
      //manejo de excepciones
      catch(std::out_of_range &excp){
          std::cout<<excp.what()<<std::endl;
          std::cout<<"Solo existen capítulos con las siguientes calificaciones:"<<std::endl;
          videos[1][stoi(vid)-1]->mostrarCalificacionesCapitulos();
      }
```

Al encontrarnos con una limitación al correr el programa, se ve así:

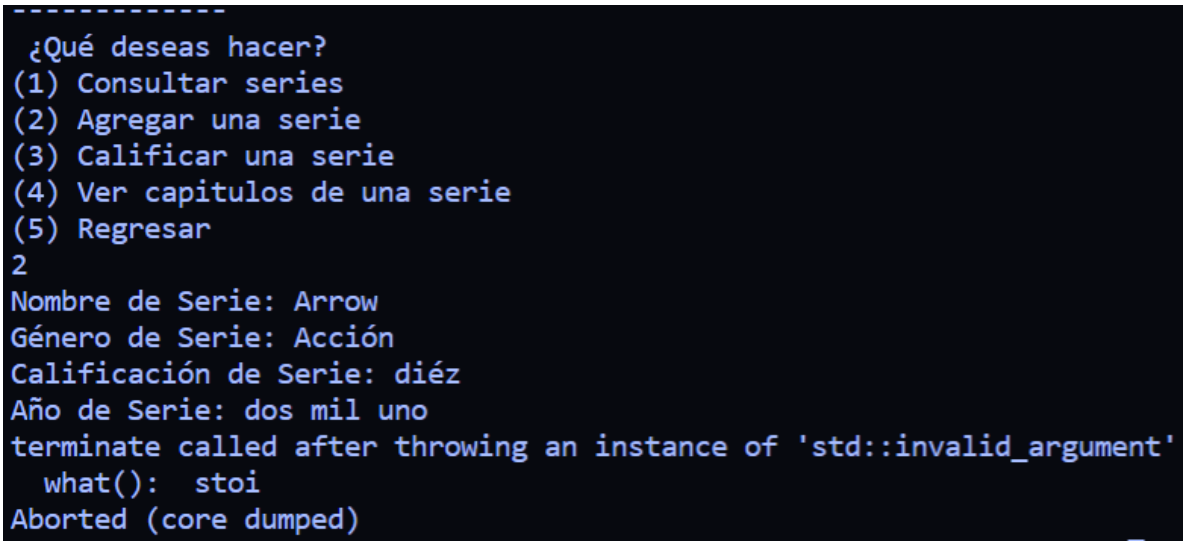


Aquí una mejora podría ser acomodar las calificaciones en un orden ascendiente.

Limitaciones

Este es el primer programa que escribo de este tipo y esta magnitud, es por esto y por falta de tiempo que el programa tiene algunas limitaciones. La más importante es que el programa no es capaz de validar todas las entradas del usuario, es decir, no se implementó el manejo de excepciones en todos los casos.

Esto se puede ver a continuación, cuando el programa crashea por un input inválido:



En este ejemplo, el programa crashea ya que el año de la serie debe de ser un número entero y aquí se introduce un string.

Otras limitaciones es que los filtros para mostrar videos son limitados, en una versión mejorada del programa, me gustaría implementar más filtros. Otra funcionalidad que me gustaría implementar es

que, al añadir series y películas nuevas, estas se escriban de manera automática en el archivo de texto de la base de datos.

Conclusión

A lo largo de esta situación problema, me di cuenta de lo eficiente que es la Programación Orientada a Objetos, solo puedo imaginar lo largo y complicado que sería este programa si se hubiera escrito sin Objetos. Una de las cosas que yo no entendía al principio era la funcionalidad y el uso del polimorfismo. Sin embargo, al implementarlo en esta situación problema, me di cuenta de lo útil que puede llegar a ser cuando se está tratando con objetos ligeramente diferentes. En general, creo que la Programación Orientada a Objetos actualmente es la mejor forma de programar. No solo es la más eficiente, sino que permite la identificación de problemas con facilidad, es óptima para la colaboración y mantiene la información delicada segura. De ahora en adelante, empezaré a utilizar esta práctica más comúnmente, al igual que la sobrecarga de operadores, polimorfismo y manejo de excepciones.