ADP - Practica 1 - Solución

Análisis y diseño con patrones (Universitat Oberta de Catalunya)

Análisis y Diseño con Patrones

Práctica 1: Patrones de Análisis y Arquitectónicos

Una empresa de entretenimiento que ofrece un servicio de suscripción de contenidos audiovisuales bajo demanda nos ha pedido que le diseñemos una parte de un sistema software para gestionar las suscripciones de sus clientes.

Las suscripciones registran el usuario de la suscripción y el password. Cada suscripción permite establecer hasta 5 perfiles diferentes. Cada perfil, que registra su nombre, puede visualizar contenidos audiovisuales diferentes.

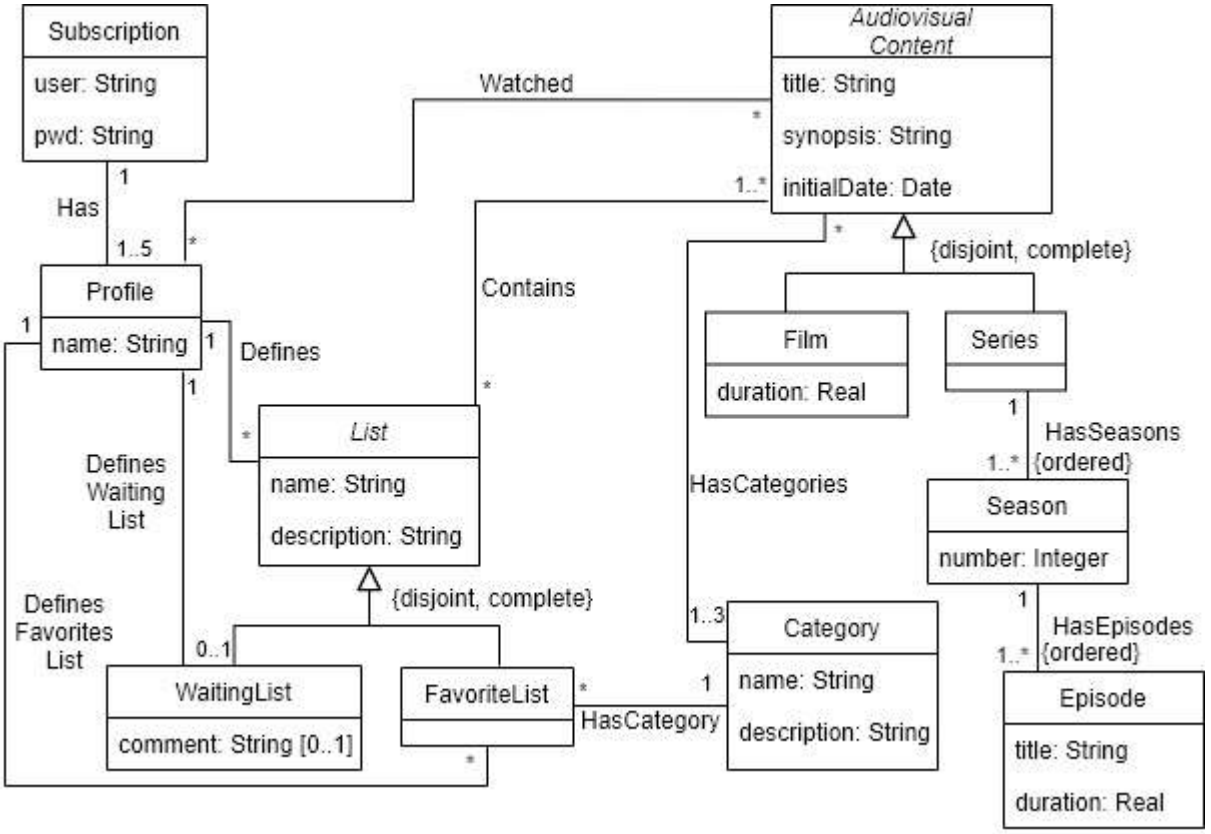
Los contenidos audiovisuales que ofrece la empresa pueden ser películas o series. Todos los contenidos registran su título, la sinopsis, la fecha de alta al sistema y sus categorías. Para las películas, adicionalmente, se registra la duración de la película. Para las series, se registran sus temporadas. Una temporada tiene un número dentro de la serie y, además, dispone de una serie de capítulos. Cada capítulo registra el título y su duración. Las categorías de los contenidos audiovisuales disponen del nombre de la categoría y de su descripción.

Cada perfil de una suscripción puede definir sus listas de contenidos audiovisuales. Las listas registran su nombre, la descripción de la lista y los contenidos audiovisuales que contiene la lista. Hay dos tipos de listas: las listas de contenidos pendientes de visualizar y las listas de contenidos favoritos por categoría. Las primeras pueden registrar un comentario y las segundas registrar la categoría de la lista. Un perfil puede definir como máximo una lista de contenidos pendientes. En cambio, puede definir las listas que quiera de contenidos favoritos por categoría.

Este sistema es una parte de un sistema más grande. Obviaremos muchos elementos que un sistema real tendría. La razón de esto no es otra que la de hacer una práctica didáctica, y que la dificultad y dedicación prevista sean las deseadas.

Disponemos ya de un análisis previo que podemos utilizar como punto de partida, pero que habrá que corregir y mejorar. A continuación disponéis del diagrama estático de análisis por esta parte del software:

Diagrama estático de análisis



Claves de les classes:

- *Subscription: user*
- *Profile: user (Subscription) + name*
- *Audiovisual Content: title*
- *Season: title (Series) + number*
- *Episode: title (Series) + number (Season) + title*
- *Category: name*
- *List: user (Subscription) + name (Profile) + name*

Restricciones de integridad:

- El número de las temporadas tienen valores consecutivos dentro de la serie y empieza por la temporada con número 1.
- Los atributos *duration* de una película y de un capítulo tienen que ser valores más grandes que 0.

- Las instancias de las asociaciones *DefinesWaitingList* y *DefinesFavoriteList* son un subconjunto de las instancias de la asociación *Defines*.
- Los contenidos audiovisuales de la lista de contenidos pendientes de visualizar de un perfil no han sido visualizados por ese perfil.
- Los contenidos audiovisuales de las listas de contenidos favoritos por categoría de un perfil han sido visualizados por ese perfil.
- La categoría de una lista de favoritos, tiene que ser una de las categorías de los contenidos audiovisuales de la lista de favoritos.

Nos han pedido diseñar una operación para eliminar un contenido audiovisual (*removeAudiovisualContent()*). A continuación disponéis del pseudocódigo de una posible implementación de este método:

```
public abstract class AudiovisualContent
{
  private String title;
  private String synopsis;
  private Date initialDate;
  protected Array<Category> categories;
  private Array<List> listsOfContents;


  public removeAudiovisualContent ()
  {
      if(typeof(this) == Film) {
            this.removeFilm();
      }
      if(typeof(this) == Series) {
            this.removeSeries();
      }
   }


public class Film extends AudiovisualContent
{
      private Integer duration;

      public removeFilm()
      {
            for each (Category c in categories) {
```

```
                c.removeAudiovisualContent(this);
            }
            this.destroy();
        }

}


public class Series extends AudiovisualContent
{
        private Array<Season> seasons;

        public removeSeries()
        {
            for each (Category c in categories) {
                c.removeAudiovisualContent(this);
            }
            for each (Season s in seasons) {
                Array<Episode> episodes = s.getEpisodes();
                for each (Episode e in episodes) {
                    e.destroy();
                }
                s.destroy();
            }
            this.destroy();
        }
}

public class Season
{
        private Integer number;
        private Array<Episode> episodes;

        public Array<Episode> getEpisodes()
        {
            return episodes;
        }

}

public class Episode
{
        private String title;
        private Integer duration;
}
```

```
public class Category
{
	private Array<AudiovisualContent> contents;

	public removeAudiovisualContent(AudiovisualContent ac)
	{

		/*Assumim que existeix una operació de List que elimina un element de
		la llista */

		contents.removeElement(ac);
	}

}
}
```

Ejercicio 1 (15%)

Revisad el pseudocódigo proporcionado. Nos preocupa que no se cumplan algunos de los principios de diseño. Se pide que, para cada principio de diseño de los que aparecen a los materiales, digáis si se cumple el principio, si no se cumple o si no aplica al pseudocódigo proporcionado. Justifica la respuesta.

Solución:

Bajo Acoplamiento: Se viola este principio puesto que, por ejemplo, la operación *removeSeries()* de la clase *Series* genera un acoplamiento entre la clase *Series* y la clase *Episode*. Este acoplamiento no existía ya que, según el diagrama de clases estas clases no estaban relacionadas.

Alta cohesión: Se viola este principio puesto que, por ejemplo, la operación *removeSeries()* de la clase *Series* tiene responsabilidades diferentes en la misma operación. Por ejemplo, esta operación tiene la responsabilidad de borrar los capítulos de la serie cuando esta responsabilidad no tendría que estar asignada a la serie. Tendría que estar asignada a la temporada de los capítulos.

Abierto-Cerrado: Se viola este principio puesto que, por ejemplo, si queremos añadir los documentales como un nuevo contenido audiovisual, la operación *removeAudiovisualContent()* habría que modificarla. Se tendría que poner otro condicional por el caso de los documentales. Una extensión al sistema implica una modificación de esta operación.

No-repetición: Se viola este principio puesto que, por ejemplo, las operaciones *removeFilm()* y *removeSeries()* tienen parte de su código duplicado.

Sustitución de Liskov: Se cumple este principio puesto que las películas y series se les puede aplicar la operación *removeAudiovisualContent()* sin ningún problema.

Segregación de interfaces: Se cumple. Todas las operaciones que están definidas en las clases se usan por sus clientes.

Inversión de dependencias: No aplica. Todas las clases que tenemos definidas tienen el mismo nivel de abstracción.

Ejercicio 2 (15%)

Se pide:

a) Proponéd una solución alternativa completa del pseudocódigo (solo las clases y operaciones que se vean afectadas), de la operación *removeAudiovisualContent()* de la clase *AudiovisualContent* que corrija la violación de los principios identificados al ejercicio 1.
b) Diagrama de clases con todas las operaciones de la solución que proponéis (solo hay que incluir las clases a las que añadís operaciones).

Solución:

a) Para resolver las violaciones indicadas al ejercicio 1 proponemos la siguiente solución:

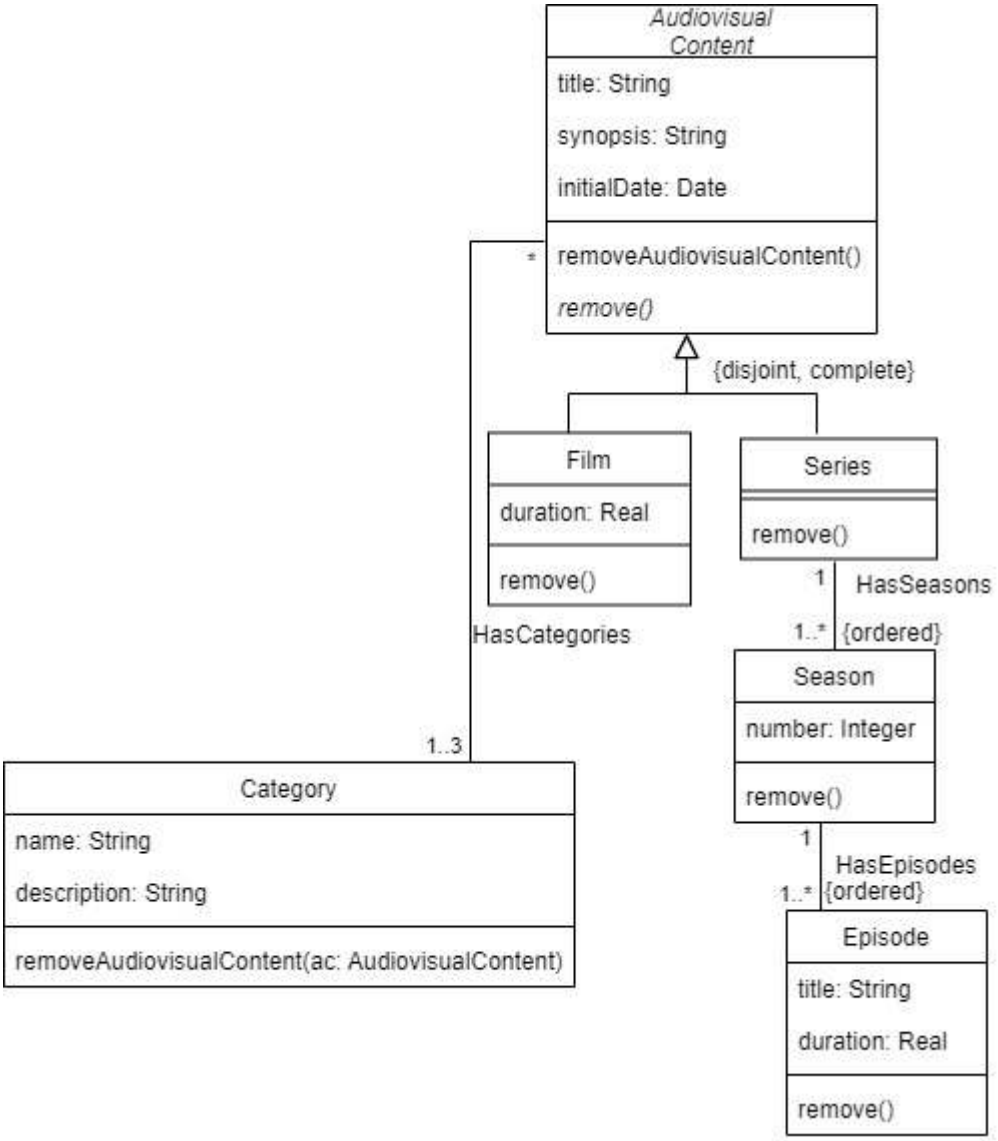
```
public abstract class AudiovisualContent
{
  private String title;
  private String synopsis;
  private Date initialDate;
  private Array<Category> categories;
  private Array<List> listsOfContents;


  public abstract remove()
  public removeAudiovisualContent ()
  {
            // Solucionamos la violación del principio de no-repetición

            for each (Category c in categories) {
```

```
                c.removeAudiovisualContent(this);
            }

            // Solucionamos la violación del principio Abierto-Cerrado

            this.remove();

            // Solucionamos la violación del principio de no-repetición

            this.destroy();
 }


public class Film extends AudiovisualContent
{
      private Integer duration;

      public remove()
      {

      /* Una solución alternativa sería poner la operación remove concreta en
       AudiovisualContent con el método vacío y no definir aquí la operación.
       Se hereda.*/

      }

}


public class Series extends AudiovisualContent
{
      private Array<Season> seasons;

      public remove()
      {
            for each (Season s in seasons) {

                  /* Solucionamos el problema del bajo acoplamiento. Series ahora
            no conoce los episodes */

                  s.remove();
            }

      }
}
```

```
public class Season
{
	private Integer number;
	private Array<Episode> episodes;

	public remove()
	{
		for each (Episode c in episodes) {
			c.remove();
		}
		this.destroy();

	}
}

public class Episode
{
	private String title;
	private Integer duration;

	public remove()
	{
		this.destroy();
	}

}
}
```

La clase Category queda igual que en el enunciado.

b) A continuación disponéis del fragmento del diagrama de clases con las operaciones que se han definido en el apartado anterior.



Ejercicio 3 (30%)

La empresa de entretenimiento para la cual estamos diseñando este sistema software se ha dado cuenta que solo está registrando si los perfiles de las suscripciones han visualizado un determinado contenido audiovisual, pero no sabe si este contenido se ha visualizado más de una vez. Es por eso que nos pide que modifiquemos el diagrama de clases para representar esta información. Además, la empresa nos pide que para cada visualización que realice un determinado perfil, este pueda de forma opcional valorar el contenido audiovisual con una nota entre 1 y 10. Esto quiere decir que es

posible que para un contenido audiovisual y un perfil determinado haya valoraciones diferentes (por ejemplo, el perfil puede valorar de forma diferente un determinado contenido si cuando lo ha visualizado su estado de humor era diferente).

Nos proponen buscar una solución que nos permita representar esta información en nuestro sistema. En concreto, se pide:

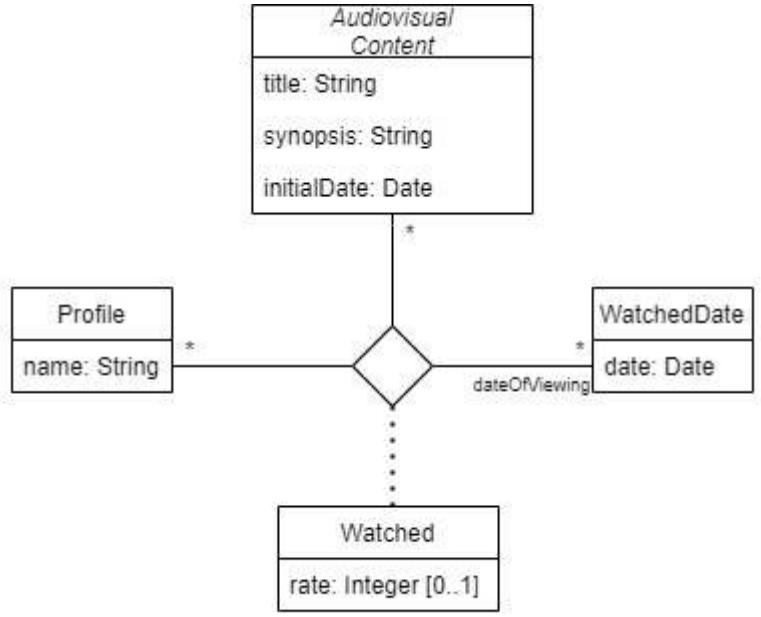
a) Describe brevemente cómo solucionarías este problema. En caso de que propongas la aplicación de un patrón, explica cual y justifica la respuesta.
b) Propón una solución detallada en forma de diagrama estático de análisis. Muestra solo las clases que cambien respecto al diagrama del enunciado y las restricciones de integridad que aparezcan.
c) Indica cómo sería el pseudocódigo resultante de la operación *periodRate(initialDate: Date, endDate: Date): Real* de la clase *AudiovisualContent* que devuelve la media de la valoración que han hecho de este contenido los diferentes perfiles que lo han visualizado (en las diferentes visualizaciones) durante el periodo indicado a los parámetros. Podéis suponer la existencia de la operación *insidePeriod(initialDate: Date, endDate:Date, date: Date): Boolean* que devuelve cierto, si la fecha *date* está dentro del periodo *[initialDate, endDate]* y falso, en caso contrario. Esta operación la podéis invocar desde donde la necesitáis.
d) Volviendo a examinar el diagrama estático de análisis nos encontramos con el problema de que no sabemos si las duraciones de las películas y de los capítulos de las series son en minutos, en segundos o en horas. De hecho, la empresa nos pide que se pueda indicar esta duración de alguna de las 3 formas (minutos, segundos u horas). Describe brevemente cómo solucionarías este problema. En caso de que propongas la aplicación de un patrón, explica cual, justificarlo y propón una solución detallada en forma de diagrama estático de análisis. Muestra solo las clases que cambien respecto al diagrama del enunciado.

Solución:

a) Según el diagrama estático de análisis del enunciado, solo se registran si se habían hecho visualizaciones de los contenidos audiovisuales para cada perfil sin tener en cuenta si la visualización se había producido más de una vez. Además, el diagrama estático de análisis del enunciado tampoco permite representar las valoraciones de cada visualización. Para poder representar esta información podemos utilizar el patrón Asociación Histórica.

Este patrón nos permite transformar la asociación binaria *Watched* en un asociación ternaria que permite registrar el histórico de las visualizaciones de un contenido audiovisual para un perfil determinado. Además la clase asociativa de la asociación ternaria nos permitirá representar en un atributo la valoración que hace el perfil de esta visualización.

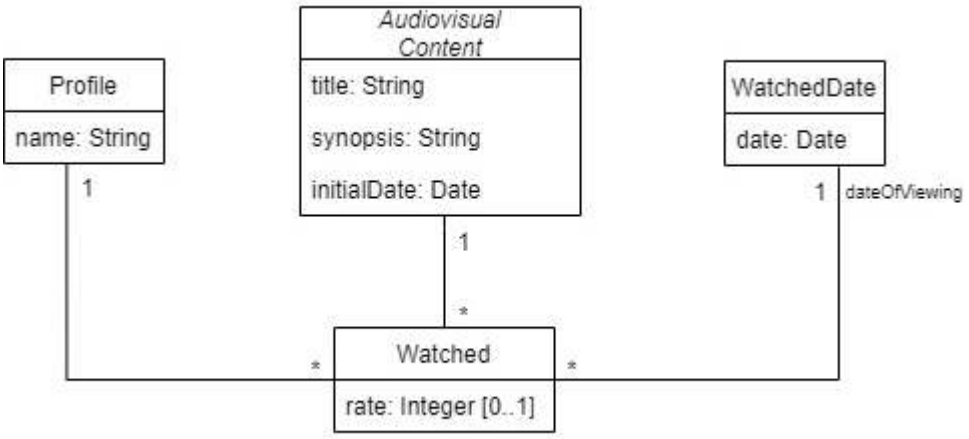
b) A continuación disponéis de la parte del diagrama estático de análisis que se ve modificado después de la aplicación del patrón Asociación Histórica.



Restricciones de integridad:

- La fecha de visualización tiene que ser posterior a la fecha de alta del contenido audiovisual en el sistema (*initialDate*).
- La valoración de una visualización tiene que ser un valor entre 1 y 10.

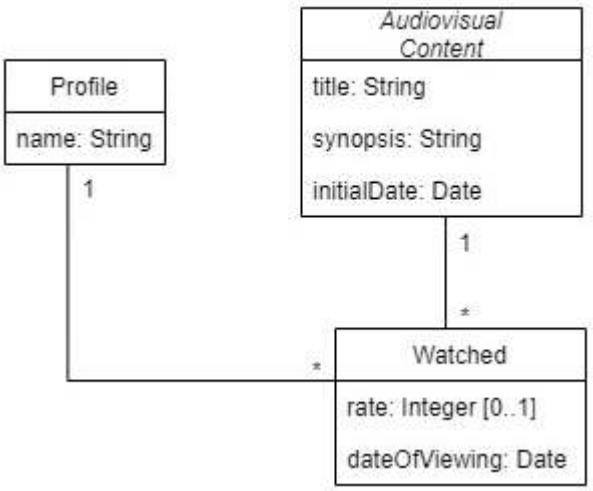
Los lenguajes de programación orientados a objetos actuales no permiten definir asociaciones ternarias, ni clases asociativas. Para poder trabajar con estos constructores, hay que aplicar una pequeña transformación al diagrama estático de análisis para obtener otro diagrama de clases que represente la misma semántica pero con constructores que los lenguajes de programación orientados a objetos permiten representar. Aplicando esta transformación obtendremos:



Restricciones de integridad (adicionales en esta solución):

- Clave de Watched: *Profile: user (Subscription) + name + title (Audiovisual Content) + date (Date)*

De hecho, como *Date* (fecha) se podría considerar como un tipo de datos, y los tipos de datos se pueden poner normalmente como atributos, esta solución la podemos transformar en:



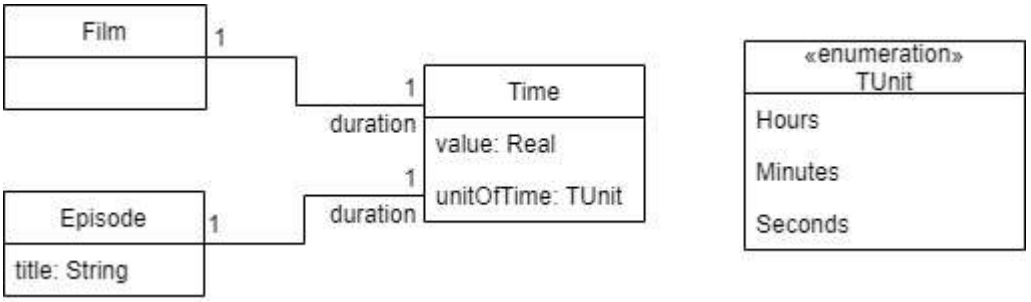
Este diagrama es el que utilizaremos para el apartado siguiente.

c)

```
public abstract class AudiovisualContent
```

```
{
  private String title;
  private String synopsis;
  private Date initialDate;
  private List<Watched> listOfWatched;


 public Real periodRate(initialDate: Date, endDate: Date)
  {
      Integer numberOfWatched = 0;
      Real totalRate = 0;

      for each (Watched w in listOfWatched) {
            if (insidePeriod(initialDate, endDate, w.getDate())) {
                  if (w.getRate() != null) {
                        totalRate = totalRate + w.getRate();
                        numberOfWatched ++;
                  }
            }
      }
      return totalRate/numberOfWatched;
  }
 }


public class Watched
{
      private Integer rate;
      private Date dateOfViewing;

      public Date getDate()
      {
         return dateOfViewing;
      }

      public Integer getRate()
      {
         return rate;
      }

}
```

d) Aplicaremos el patrón *quantity* para representar de forma más acurada la duración de los contenidos audiovisuales. A continuación disponéis del diagrama estático de análisis con la aplicación del patrón. Solo representamos las clases involucradas.



Ejercicio 4 (10%)

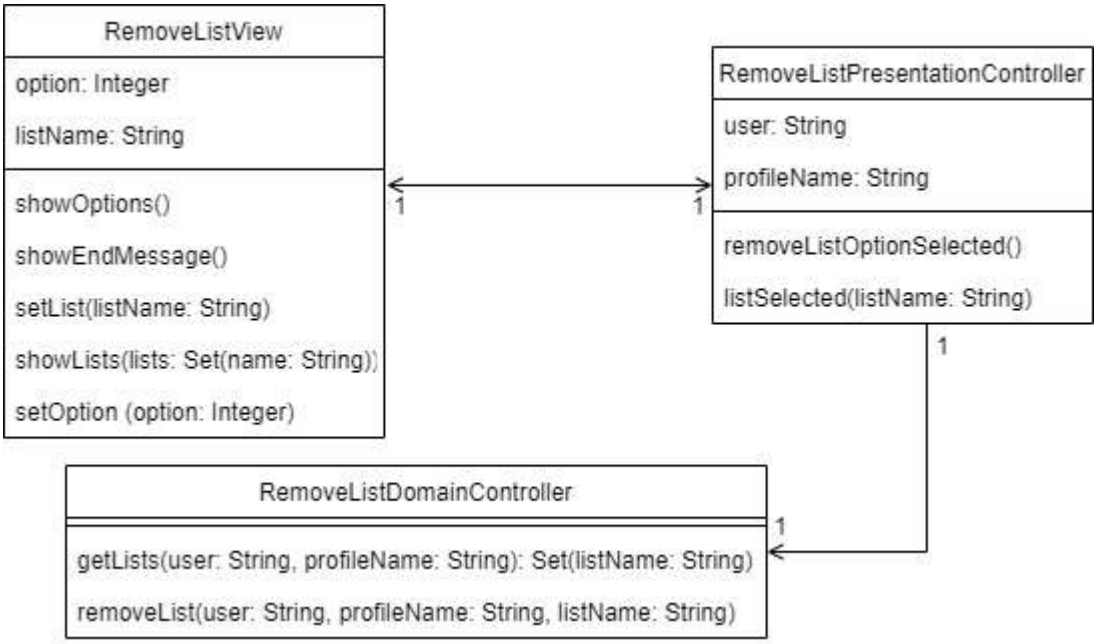
Queremos definir la capa de presentación de nuestro sistema y aplicar el patrón de arquitectura Modelo, Vista y Controlador (MVC) para el caso de uso *Eliminar una lista de un perfil*. Este caso de uso permite a un perfil, cuando está en la pantalla que muestra en un menú todas las funcionalidades que puede utilizar el perfil, seleccionar del menú la opción de eliminar una de sus listas. En respuesta, el sistema le mostrará todas las listas que tiene el perfil definidas. El usuario seleccionará la lista que quiere eliminar, el sistema eliminará la lista y le mostrará el mensaje “List successfully removed”.

Se pide:

a) Diagrama de clases del MVC para este caso de uso incluyendo las operaciones que sean necesarias (las operaciones tienen que contener los parámetros que necesiten y los tipos de retorno, si tienen). El diagrama de clases y las operaciones tienen que ser específicas para este caso de uso.
b) Describe brevemente qué hace cada operación.

Solución:

a) Aplicaremos el MVC a la capa de Presentación. La clase *RemoveListView* representa la vista del patrón, la clase *RemoveListPresentationController* representa el controlador y la clase *RemoveListDomainController* representa el controlador de la capa de dominio. El diagrama de clases del MVC correspondiente al caso de uso sería:



b) A continuación se describen todas las operaciones de las clases definidas en el diagrama de clases del apartado a). Asumimos que la clase *RemovePresentationController* dispone del identificador del perfil que se está utilizando en el caso de uso (se ha identificado el perfil cuando ha iniciado su sesión).

Operaciones de la clase RemoveListView:

- RemoveListView::showOptions() muestra las diferentes opciones correspondientes a los diferentes casos de uso (entre ellos la baja de la lista) a la pantalla.
- RemoveListView::setOption(option) registra el caso de uso seleccionado y lo notifica al controlador. Esta operación es opcional, solo si se quiere que la vista grabe el caso de uso seleccionado.
- RemoveListView::showLists(lists: Siete (name:String)) muestra en pantalla un desplegable con todas las listas del perfil.
- RemoveListView::setList(listName: String) registra la lista que se quiere dar de baja.

- RemoveListView::showEndMessage() muestra en pantalla el mensaje “List successfully removed”.

Operaciones de la clase RemoveListPresentationController:

- RemoveListPresentationController::removeListOptionSelected() detecta que el usuario ha escogido la opción para dar de baja una lista e invoca a la operación para obtener las listas del perfil.
- RemoveListPresentationController::listSelected(listName: String) detecta que el usuario ha seleccionado la lista a dar de baja e invoca a la operación que mujer de baja la lista.

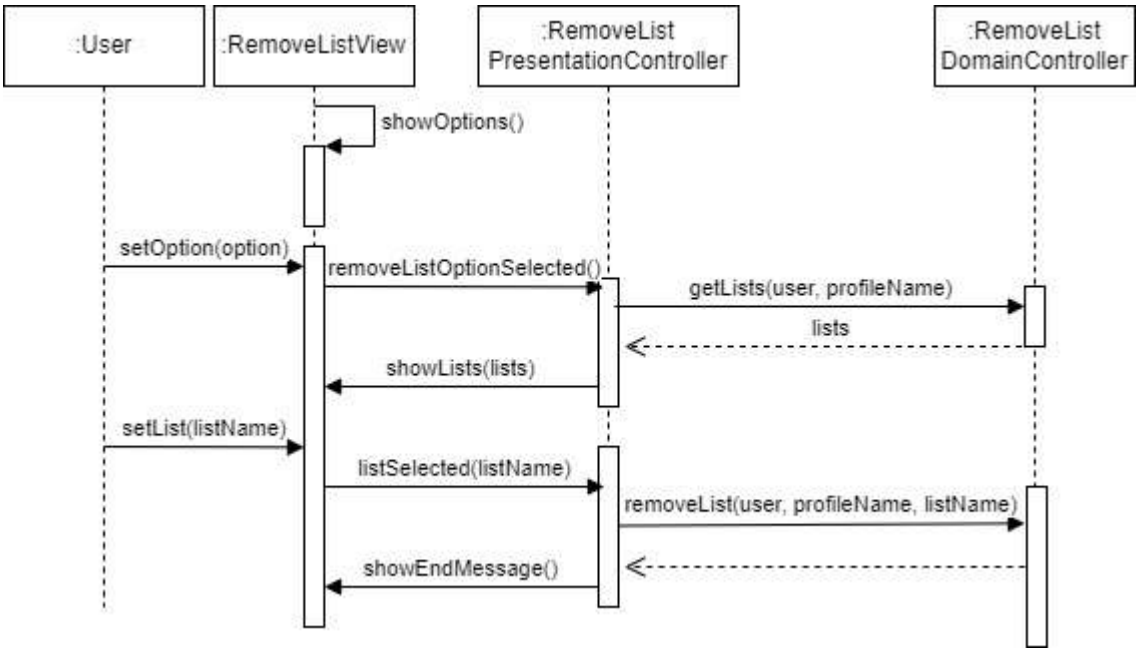
Operaciones de la clase RemoveListDomainController:

- RemoveListDomainController::getLists(user:String, profileName: String): Siete(listName: String) obtiene las listas que tiene el perfil.
- RemoveListDomainController::removeList(user:String, profileName: String, listName: String) da de baja lista del perfil.

Ejercicio 5 (10%)

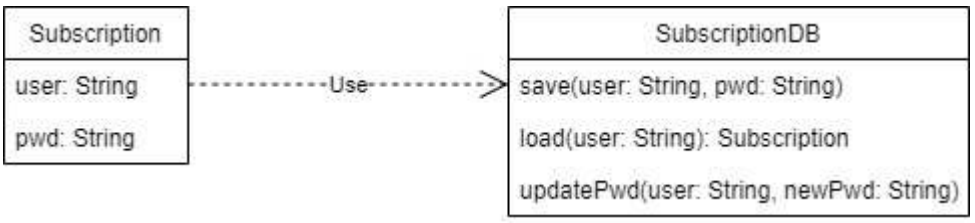
Propón el diagrama de secuencia o pseudocódigo del caso de uso *Eliminar una lista de un perfil* (del ejercicio anterior), desde que se muestra la pantalla para que el usuario del perfil pueda seleccionar del menú la opción de eliminar una de sus listas hasta que el sistema le muestra el mensaje “List successfully removed”.

Solución:



Ejercicio 6 (10%)

Seguimos diseñando la arquitectura del sistema y hemos decidido utilizar una arquitectura con 3 capas: presentación, dominio y servicios técnicos (en nuestro caso, capa de de datos). Hemos hecho un primer diseño de cómo se podrían comunicar las clases que ya tenemos definidas con la base de datos (BD). En concreto, para simplificar, nos centraremos en la clase *Subscription*. El resto de clases se podrían tratar de forma similar. El diseño que proponemos es el siguiente:

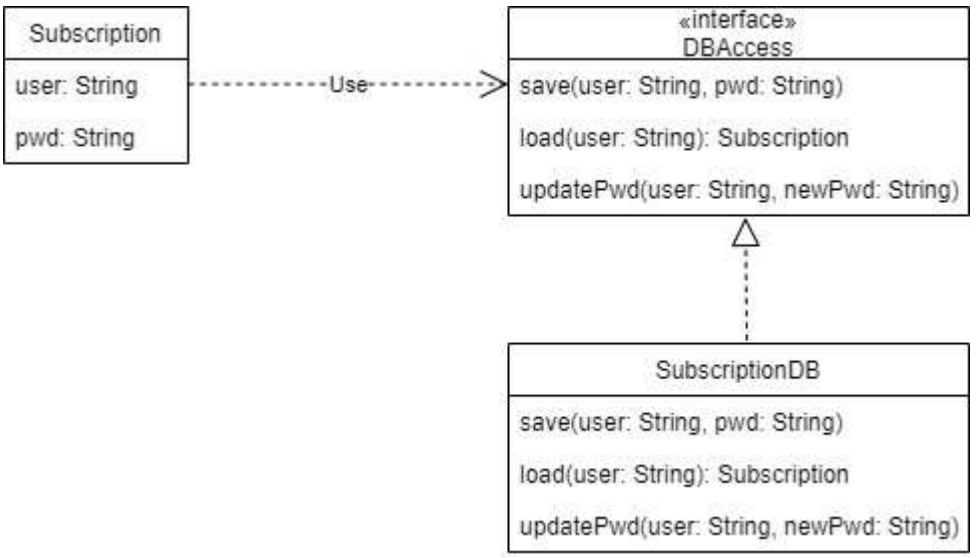


La clase *SubscriptionDB* es la encargada de interaccionar con la BD (y concretamente con la mesa *Subscription* de la BD). La operación *save* crea una fila con una nueva suscripción a la tabla *Subscription* de la BD, la operación *load* obtiene una fila, es decir, una suscripción a partir del identificador indicado en el parámetro de la operación y la operación *updatePwd* modifica el *password*, con el valor *newPwd* del parámetro, de la fila que tiene como clave la *user* que hay en el parámetro de la operación.

¿Qué principio de diseño se está violando? Justifica tu respuesta. Propón un diagrama de clases de análisis que solucione la violación del principio.

Solución:

El principio de diseño que se viola es el de Inversión de dependencias. Este principio nos dice que los módulos o clases de alto nivel no tendrían que depender de los de bajo nivel sino de una abstracción. Las abstracciones no tendrían que depender de los detalles. Los detalles tendrían que depender de las abstracciones. En nuestro caso, la clase Subscription depende de una clase (*SubscriptionDB*) de más bajo nivel que tiene detalles del acceso a la DB. El diagrama de clases que soluciona la violación de este principio es:



La nueva solución, la clase *Subscription* no depende de la clase *SubscriptionDB* sino que depende de una interfaz *DBAccess* (abstracción). Los detalles, es decir, el acceso a la DB están en la clase *SubscriptionDB* que solo depende de la interfaz *DBAccess* (abstracción).

Ejercicio 7 (10%)

Para acabar de diseñar la arquitectura en 3 capas del sistema, usando las tres capas clásicas de muchos sistemas de información:

1. Presentación
2. Dominio
3. Servicios técnicos (Datos)

Queremos reflexionar sobre a qué capa pertenece cada una de las clases que han ido surgiendo en el enunciado (y a la solución que has elaborado hasta ahora):

- Clases del diagrama de clases del ejercicio 2.
- Clases del diagrama de clases del ejercicio 4.
- Clases del diagrama de clases del ejercicio 6.

a) Indica qué clases (e interfaces, si hay) pertenecen a la capa de dominio. Justifica tu respuesta.
b) Indica qué clases (e interfaces, si hay) pertenecen a la capa de presentación. Justifica tu respuesta.
c) Indica qué clases (e interfaces, si hay) pertenecen a la capa de servicios técnicos (en nuestro caso, capa de datos). Justifica tu respuesta.

Solución:

a) Las clases de la capa de dominio es donde hay la lógica de negocio, donde trabajamos el problema que nuestro sistema resuelve.

   Esto incluye, naturalmente, las clases las instancias de las cuales representan objetos, documentos o valores del mundo real, como un contenido audiovisual, una película, etc. Por lo tanto, todas las clases del diagrama de clases del ejercicio 2 pertenecen a la capa de dominio.

   También incluye la clase que ofrece operaciones del nivel del dominio para ser usadas por la capa de presentación, el que denominamos controladores de dominio. En nuestro caso, *RemoveListDomainController* es un controlador de la capa de dominio que estaba definido en el diagrama de clases del ejercicio 4.

   Finalmente, la interfaz *DBAccess* (del diagrama de clases del ejercicio 6), que se ha añadido para satisfacer el principio de inversión de dependencias, también está definida a la capa de dominio.

b) Las clases de la capa de presentación son aquellas que gestionan la presentación de la información (vistas del MVC) y la recepción de los acontecimientos al sistema (controladores de presentación del MVC). En nuestro caso estas clases serían *RemoveListView* y *RemoveListPresentationController*.

c) Las clases de la capa de datos son aquellas que gestionan el acceso a la DB. En nuestro caso estas clases serían *SubscriptionDB* y el resto de clases correspondientes a las clases de la capa de dominio para acceder a la DB.