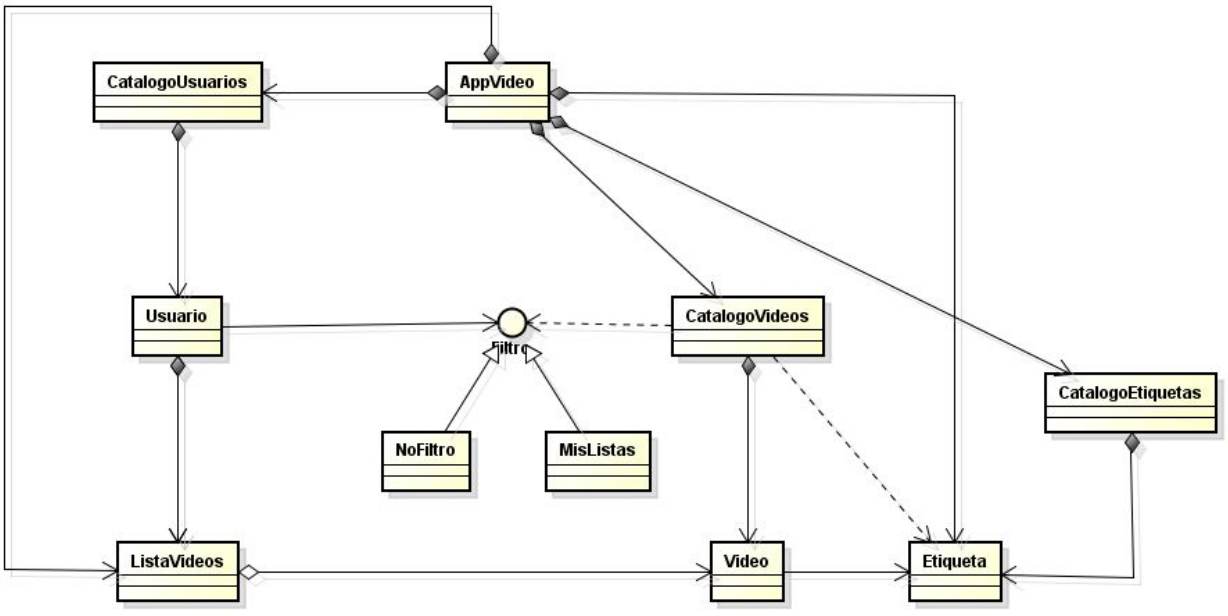
Memoria Aplicación AppVideo

Realizado por: José Miguel Sánchez Almagro josemiguel.sancheza@um.es y Francisco Hita Ruiz francisco.hita2@um.es

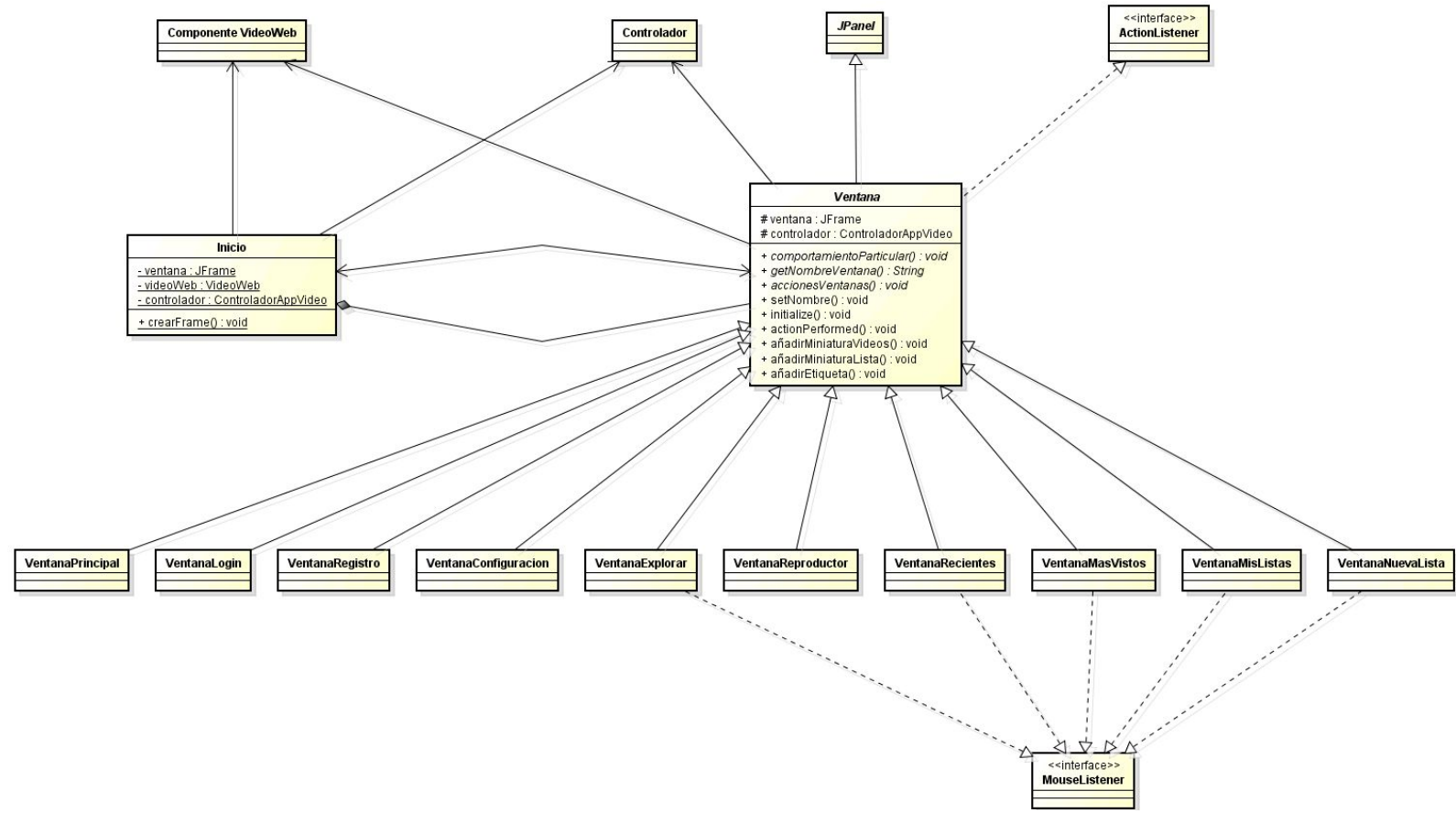
Índice

Diagrama de clases del dominio



Como se puede ver diagrama de clases apenas ha cambiado respecto al original proporcionado por los profesores, la única diferencia es que hemos tenido la necesidad de crear un catálogo. Esto es debido a que necesitamos tener un lugar donde se encuentren todas las etiquetas, para así cuando metemos nuevas etiquetas podemos comprobar si esa etiqueta ya existe en el catálogo y así no duplicarla.

Diagramas adicionales



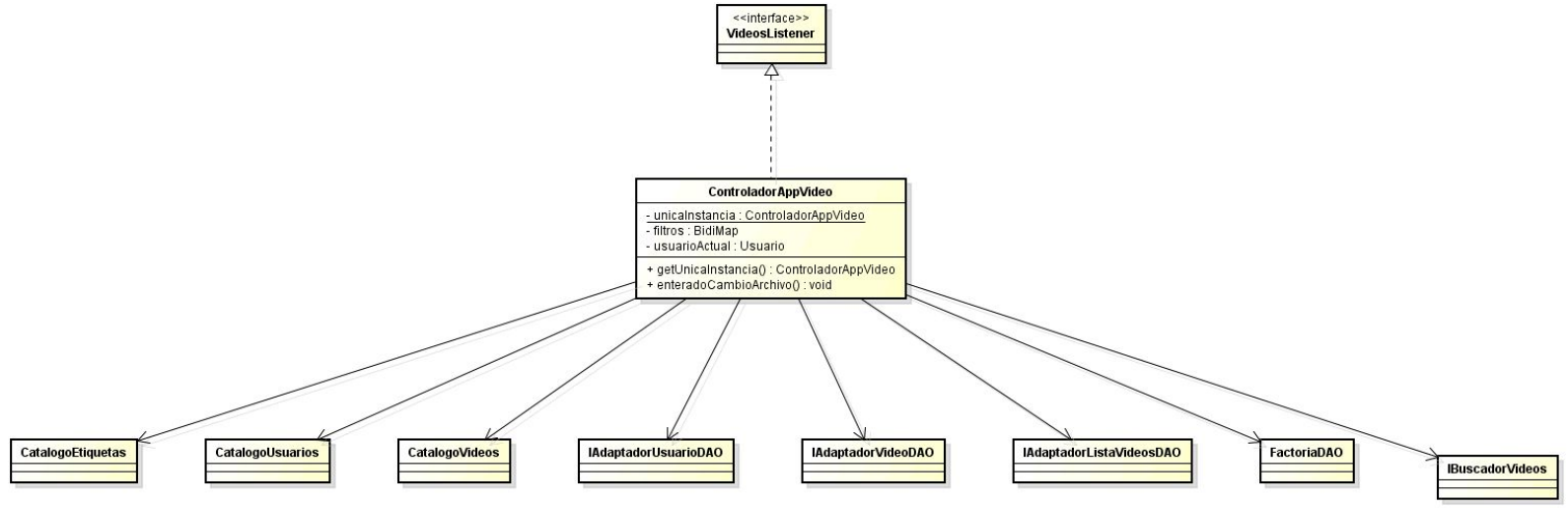
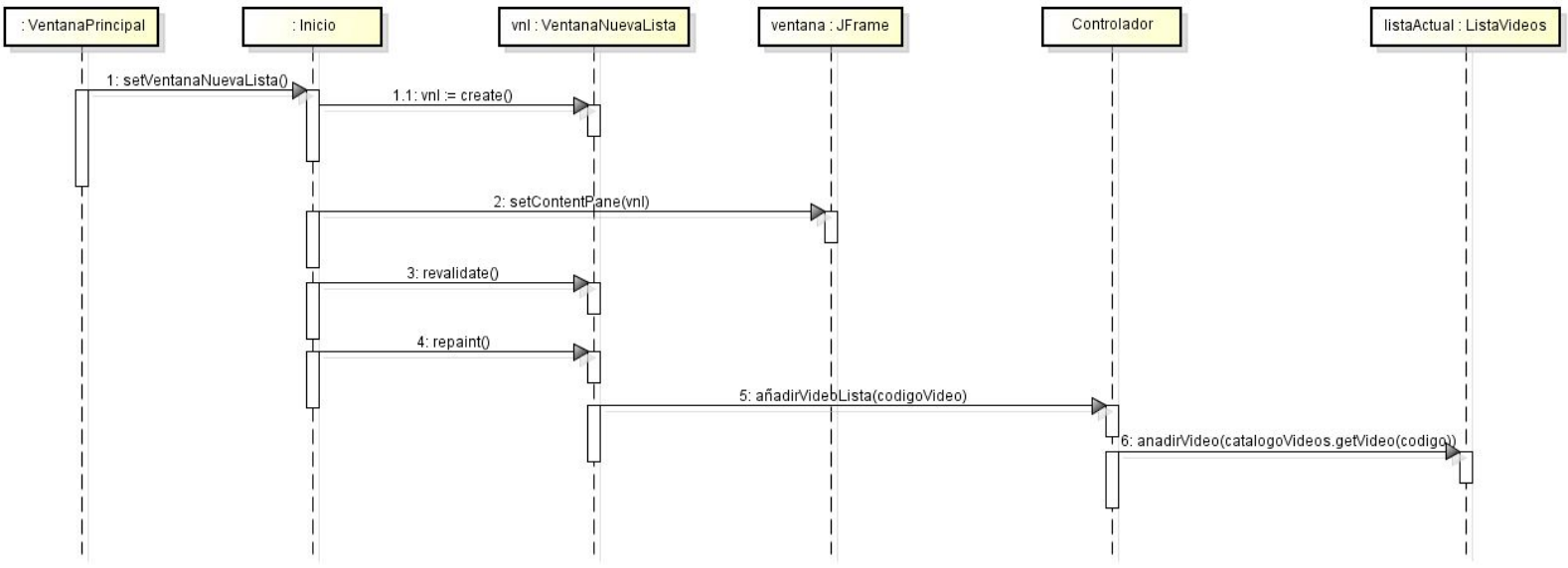


Diagrama de secuencia para la operación añadir un vídeo a una lista de vídeos del usuario



En este diagrama de secuencia podemos observar como desde la ventana principal al seleccionar el botón de nueva lista se llama a unas funciones estáticas de la clase Inicio que se encarga de cambiar el panel del frame por el panel que representa la clase VentanaNuevaLista. Ya una vez aquí tras seleccionar un vídeo y agregarlo a la lista, se llama al controlador indicando el código del vídeo el cual queremos introducir en la lista. Tras esto el controlador ejecuta el método, que añade un vídeo a una lista, sobre la lista actual, la cual es un atributo del controlador, que se va actualizando cuando se crea una nueva lista o cuando se busca dentro de una lista.

Explicación breve de la arquitectura de la aplicación

La arquitectura de nuestra aplicación se basa en la separación modelo-vista, es decir, que las clases del modelo no conocen directamente los objetos de la vista. Por lo tanto tenemos por un lado las clases del modelo, las cuales albergan la funcionalidad de la aplicación y las principales clases que la componen (Usuario, Video, ListaVideos…), por otro la vista, que alberga las distintas ventanas que muestran diversos elementos en pantalla para interactuar con nuestra aplicación, y por último el controlador que hace de nexo entre los dos anteriores, es decir, cuando realizamos interacciones con la vista, ésta delegará las acciones realizadas por el cliente al controlador, y este se encargará de delegar estas tareas en las diversas clases del modelo. Además de estas 3 capas, contamos con otra de

persistencia, la cual se encarga de almacenar los objetos en una base de datos. Para evitar acoplar nuestra aplicación a un servicio de persistencia concreto se usan una serie de adaptadores que hemos creado. Sin olvidar que para aliviarnos la tarea a los alumnos de tratar con la base de datos directamente, el profesorado nos entregó una API que es la que dialoga con la base de datos, y son nuestros adaptadores los que usan esta API.

Componentes utilizados

En este proyecto hemos usado varios componentes, varios de ellos destinados a manejar la persistencia, otro para generar un PDF (iText), otro que representa un botón que se ilumina (Luz), otro para reproducir videos de Youtube (VideoWeb), otro para mostrar un calendario (JCalendar) y por último el encargado de cargar videos desde ficheros xml (ComponenteCargadorVideos).

Los componentes usados para la persistencia en nuestro caso han sido los destinados a usar una base de datos H2, estos incluyen también el driver proporcionado por el profesorado, que nos abstrae de no tener que conocer en detalle el uso de una base de datos H2.

Respecto al componente iText, que sirve para generar ficheros PDF, usamos el que venía recomendado en el enunciado de la práctica. Este nos permite generar ficheros PDF de forma cómoda usando párrafos que contienen cadenas (String).

El componente botón de luz es un simple botón que se enciende al pulsarlo, y en esta práctica se usa para mostrar una ventana al usuario donde podrá elegir un fichero xml que contenga videos.

En torno al componente VideoWeb, este nos es entregado por el profesorado, y nos permite reproducir videos en nuestra aplicación a través de la url del video en cuestión. Además destacar una funcionalidad en particular, la cual nos permite obtener una pequeña miniatura de los videos para poder mostrar una pequeña vista previa de estos al usuario.

El componente JCalendar lo obtuvimos de un repositorio de internet, este nos permite mostrar un calendario desplegable, dando la opción al usuario de elegir una fecha, la cual no sirve de utilidad a la hora de realizar el registro de un nuevo usuario.

Por último el ComponenteCargadorVideos, del cual se nos entrega un esqueleto para completar siguiendo el modelo de delegación de eventos de Java, lo usamos para traducir los ficheros xml a objetos del dominio, y además de esto avisamos a las clases interesadas de los nuevos vídeos, que en nuestro caso es el controlado, y así este puede actualizar tanto el catálogo de vídeos como la base de datos.

Patrones de diseño utilizados

Hemos usado el patrón **Factoría abstracta** para la creación de los adaptadores DAO, ya que necesitamos crear una familia de productos, en nuestro caso adaptadores para usuarios, vídeos y listas de vídeos, por ello al usar este patrón si el día de mañana necesitaremos otros adaptadores para usar otro tipo distinto de base de datos, podríamos crear esos nuevos adaptadores con una factoría concreta que heredase de la factoría abstracta que hemos implementado.

Como ya hemos comentado, usamos el patrón **Adaptador** para la persistencia de las clases necesarias, en este caso Usuario, Video y ListaVideos. El patrón adaptador nos permite tener un mismo código que permite almacenar, recuperar y eliminar información de una base de datos sin importar qué tecnología concreta estemos usando, es decir, que si el día de mañana sustituimos la base de datos H2 por una SQL, solo habría que implementar los adaptadores necesarios para usar la base de datos SQL, estos implementarían las interfaces que tenemos hechas para cada clase, y por lo tanto nuestro código seguiría siendo funcional ya que usa los métodos estipulados en esas interfaces, permitiéndonos así olvidarnos de la implementación concreta de cada adaptador para un cierto tipo de base de datos.

También hacemos uso del patrón **Singleton** en distintas clases, por ejemplo en la Factoría abstracta, ya que solo necesitamos una y queremos que todos accedan a la misma instancia. También lo usamos en los catálogos, ya que es imprescindible que solo haya un catálogo de un tipo, y todo el que lo necesite acceda a una única instancia, no sería correcto que hubieses distintos catálogos, por ejemplo de vídeos, con distinto contenido, es por esto que se necesita el uso este patrón.

Se hace uso del patrón **Observer** en el componente cargador de vídeos, concretamente el modelo de delegación de eventos de Java. Gracias al uso de este podemos avisar a los interesados de cuando se introducen nuevos vídeos a través del parseo de un nuevo documento xml. En nuestro caso cuando en la aplicación usamos el botón Luz, se nos abre una ventana que nos permite explorar el sistema de ficheros en busca de un fichero xml, y por lo tanto éste, a través de cargador de vídeos, genera una nueva lista de vídeos, avisando al controlador de la existencia de nuevos vídeos, y ya éste se encargará de introducirlos al catálogo y a la base de datos. También resaltar que se ha usado este patrón en la vista, ya que todos los botones de ésta al ser clicados se producen eventos para avisar a los interesados.

También se hace uso del **método plantilla** por ejemplo en la clase FactoriaDAO, de la cual hereda la clase TDSFactoriaDAO, la cual ya tendrá que definir esos métodos. El uso de este patrón nos permite que cada factoría concreta implemente estos métodos según una especificación concreta, en este caso estos métodos devuelven una instancia de los adaptadores, por lo tanto según la factoría concreta de la que se tratase se devolverían unos adaptadores concretos u otros, pero todos seguirían una misma estructura. Otro

ejemplo de uso del método plantilla en nuestro proyecto sería en las ventanas, donde estás heredan de una clase Ventana abstracta, teniendo esta una serie de métodos abstractos, que ya cada ventana concreta implementa según sus necesidades.

Cómo hacemos uso de Swing para crear nuestra nuestra interfaz gráfica, hay que resaltar que Swing emplea el patrón **Composite** para sus componentes, es decir, que tiene componentes simples y compuestos. Los simples son botones, listas etc. Y los compuestos son contenedores, que más concretamente pueden ser ventanas o paneles, y estos contienen otros componentes tanto simples como compuestos. Este patrón es muy útil para permitirnos tener ventanas que puedan albergar paneles que a su vez puedan albergar uno o más paneles que también pueden contener más contenedores o componentes simples como un botón. También Swing usa el patrón **Bridge** en los componentes, ya que estos tienen un atributo “peer” que referencia a una implementación concreta, por ejemplo un botón puede tener distintas implementaciones, una de ellas podría ser en Windows. Al usar este patrón evitamos una explosión de clases y separamos los tipos de botón (abstracción) de su implementación.

También se hace uso de forma indirecta del patrón **Decorador** en la vista, por ejemplo cuando usamos un ScrollPanel. Este patrón nos permite extender la funcionalidad y aplicarla tanto sobre otro componentes, como sobre decoradores que decoran a otros componentes. Es decir, si tenemos un panel, este puede ser decorado con un Border, y así a la hora de ser dibujado llevará un borde de cierto tipo. Además ese panel que hemos mencionado podría ser decorado también por un Scroll, y por lo tanto el dibujarse tendría un borde y una barra para hacer scroll.

Tests unitarios implementados

Respecto a los tests, hemos realizado uno sobre el catálogo de vídeos, en el cual comprobamos que el método para eliminar un vídeo funciona correctamente.
También hemos hecho 5 tests del Controlador. El primero comprueba que el Controlador es informado cuando cambiamos el fichero xml desde el que se obtienen los vídeos.
El siguiente comprueba que el login de un usuario funciona correctamente. Luego comprobamos que se añaden correctamente los videos reproducidos por el usuario a la lista de recientes. También se testea el método que nos devuelve los 10 vídeos más vistos. Y por último se testea la búsqueda de vídeos. En alguno de estos test se puede apreciar como se abren varios xml seguidos, esto se hace para que si el test se hace varias veces podría coincidir que el último xml cargado es el que queremos abrir de nuevo, y entonces no se cargaría nada ya que se creería que no se ha modificado el valor y por lo tanto no tiene nada nuevo, por eso abrimos un fichero xml sin importancia antes de los necesarios.

Observaciones finales

Hemos dedicado aproximadamente, si sumamos las horas de ambos integrantes, unas 160 horas, las cuales han estado mayormente concentradas entre el 20 y el 4 de Julio.
Como conclusión final podemos decir que la realización de esta práctica, aunque un poco larga, nos parece interesante ya que nos acerca más a cómo se estructuran las aplicaciones que se desarrollan en grandes empresas con varias personas trabajando de forma paralela. Además de que nos hace ver la utilidad que tienen los patrones de diseño en la práctica, y como sin ellos podría tener que cambiar mucho código si en un futuro tuviéramos que ampliar la funcionalidad o cambiar el tipo de la base de datos etc.