**Martial Arts School**

**Gestión**

**MEMORIA PROYECTO**

**Abraham Garrido Rosillo.**

**DESARROLLO**

**APLICACIONES MULTIPLATAFORMA**

**1. Portada.**

**2. Índice de contenidos, pudiéndose incluir también de figuras y/o imágenes.**

**3. Introducción: con una extensión máxima de una página, donde se haga un resumen muy sintetizado de los apartados 5 a 7 (objetivos y finalidad, medios, planificación).**

Se pretende implementar una aplicación de escritorio para la gestión de los socios del club de artes marciales Martial Arts School. Se pretende una solución que permita el alta y modificación de los datos de los socios, así como informes de socios (activos, impagados, etc.), facilitando la gestión diaria del club. Para el desarrollo de la presente aplicación proponemos el clásico modelo de cinco fases o etapas.

El objetivo es crear una aplicación que de forma sencilla permita al gestor del club de artes marciales Martial Arts School almacenar de forma fiable y segura los datos personales (DNI, nombre, actividad que realiza, fecha de nacimiento, teléfono, etc.) de los socios así como la información relativa a pagos mensuales y grados de los alumnos. Pretendemos informatizar la gestión del club aportando una solución usable y ligera con el consiguiente ahorro de tiempo e incremento de productividad e imagen para el club MAS. Sustituiremos los registros actuales en papel por un programa de gestión adecuado con una solución ágil y sencilla de manejar modernizando en consonancia la gestión del club con poco esfuerzo de aprendizaje de los gestores del mismo.

Para el desarrollo de la aplicación hemos contado con un portátil ACER de gama media con Windows 10. Como lenguaje hemos optado por JAVA por ser el lenguaje principal aprendido en el ciclo en entorno de desarrollo Netbeans. Java dispone de recursos suficientes para una aplicación moderna como la que nos ocupa. La aplicación hará uso de un repositorio SQL para almacenar la información permanentemente.

Para el repositorio relacional MySQL será nuestra opción open source consiguiendo Atomicidad, Consistencia, Aislamiento y Durabilidad (ACID en inglés).

La planificación, análisis, desarrollo y mantenimiento serán realizados por mí de forma individual como es perentorio. La secretaria del club MAS en aquellas fases que requieran una persona diferente del desarrollador (pruebas).

Lo primero que haremos será planificar las fases de desarrollo de la aplicación asignando a cada una un tiempo orientativo:

La planificación en sí ha tenido un tiempo previsto de no más de una semana.

El análisis sí ha tenido un tiempo previsto de tres semanas.

El desarrollo no más de cuatro semanas.

Las pruebas no más de cuatro semanas.

Se previó por tanto la conclusión en tres meses (12 semanas) iniciando la fase de mantenimiento.

**4. Finalidad, objetivos y características del proyecto: especificar, con la mayor claridad y lo más detalladamente posible, las metas a alcanzar con el proyecto, la razón de ser del mismo; cómo funcionará la aplicación una vez terminada, partes de que consta, que se quiere conseguir con su implementación, entre otros. Indica los antecedentes del mismo.**

**Incluye aquí posibles estudios de mercado que hayas realizado y que permitan detectar las carencias en el mismo que se pretenden cubrir con el proyecto diseñado. Puedes especificar las fases o partes que componen el proyecto y su contenido, aunque también puedes hacerlo en el apartado de planificación del proyecto.**

Como gestor de club de artes marciales he podido constatar que en el mercado existen soluciones de gestión para centros deportivos incluso open source, pero ninguno de ellos específico para clubes de artes marciales y su problemática particular. Además, los existentes pueden generar rechazo por su excesiva complejidad por contener funcionalidades no necesarias para los clubes de artes marciales. Hay que tener en cuenta que el gestor medio de clubes de artes marciales suele tener un nivel bajo de destrezas informáticas. Es por ello que hemos puesto total énfasis en la sencillez y ligereza del programa, tanto a nivel visual como funcional y de manejo.

**5. Medios materiales usados: humanos, hardware, software, especificados con detalle. Habrá que indicar si finalmente se han respetado los materiales indicados en el anteproyecto o si la evolución del mismo ha precisado una nueva replanificación de éstos.**

Para el desarrollo de la aplicación contamos con un portátil ACER equipado con un microprocesador Intel i7 2.6Ghz, 8Gb de RAM y 1Tb de HDD. El sistema operativo es Windows 10.



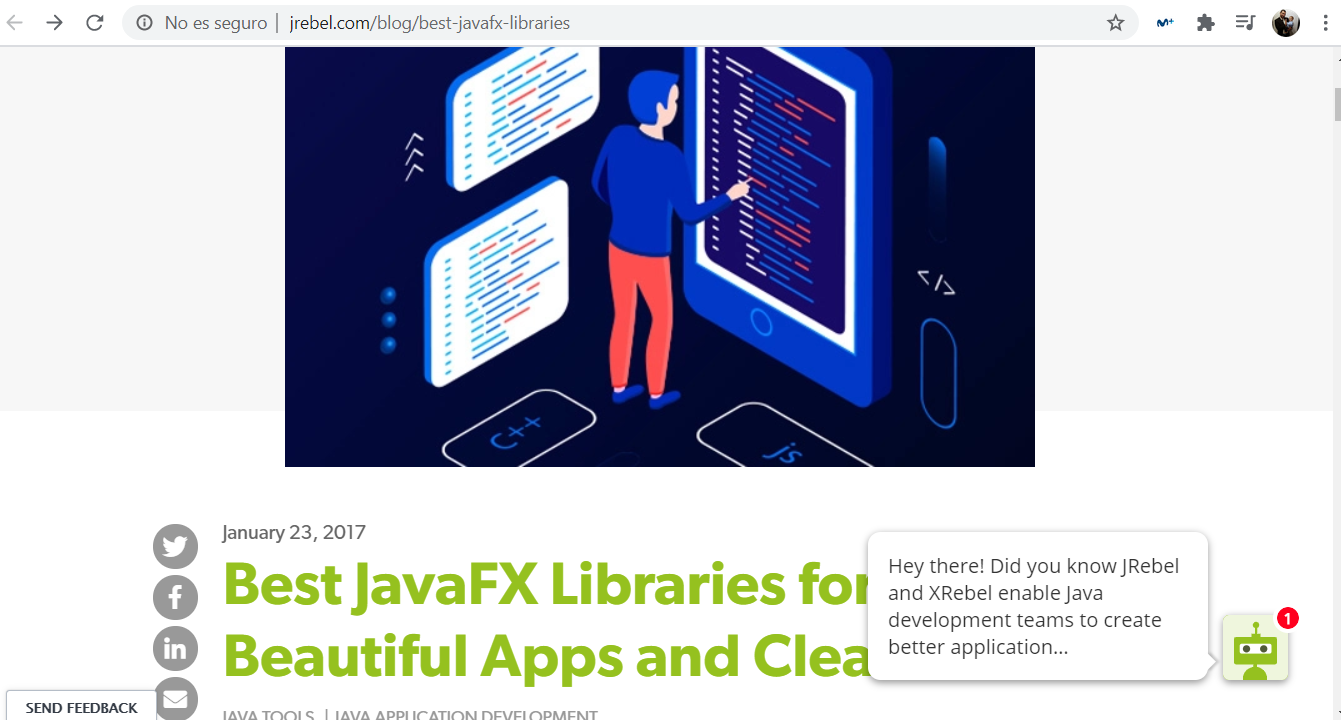
Imagen similar al portátil utilizado para el desarrollo. Fuente: PCComponentes.

En cuanto al lenguaje de desarrollo hemos optado por el popular JAVA por ser multiplataforma y portable, pero principalmente por ser de largo el lenguaje que más hemos utilizado en el ciclo formativo DAM. A lo largo de la titulación hemos visto otros lenguajes como C# en Visual Studio, donde hemos desarrollado pequeños programas con la base adquirida en Java. También hemos desarrollado en la asignatura de Sistemas de Gestion Empresarial módulos para Oddo en el leguaje Python. En ambos casos hemos podido comprobar que aprender un segundo o tercer lenguaje no es demasiado complejo con una buena base de aprendizaje de los conceptos clave de POO. A pesar de haber desarrollado en otros lenguajes creemos que la opción más lógica en nuestro caso es el lenguaje al que claramente más horas hemos dedicado y de ahí que la decisión haya sido sencilla de tomar. Java dispone de clases más que suficientes (GUI, SQL,etc.) para una aplicación moderna como la que nos ocupa.

En cuanto a los entornos de desarrollo, hemos trabajado en NetBeans y Eclipse principalmente, aunque hemos visto otros IDE como Visual Studio o Android Studio. Dado que desarrollamos una aplicación de escritorio en Java, quedan descartados Visual Studio y Android Studio. Quedan por tanto dos opciones (las más utilizadas durante el curso) de las que nos quedamos con Netbeans, siendo una opción puramente sentimental por haber sido el primer IDE aprendido en éste curso.

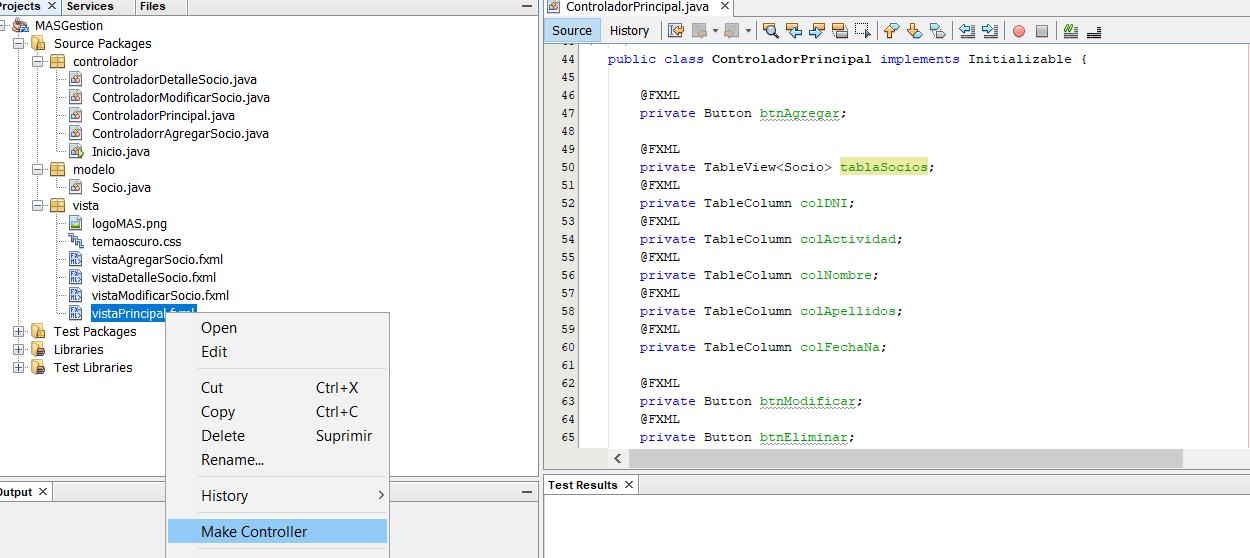
Hemos decidido utilizar la tecnología JavaFX en lugar de la librería SWING para dar un aspecto moderno a nuestra aplicación. A lo largo del ciclo hemos utilizado para el desarrollo de interfaces en Java principalmente SWING, habiendo visto JavaFX sólo superficialmente. Pese a ello creemos conveniente profundizar en JavaFX y desarrollar el proyecto haciendo uso de ella por varias razones entre las que destacan:

-Un aspecto visual más moderno. Los controles de JavaFX proporcionan una imagen más moderna con respecto a los de SWING para nuestras interfaces. JavaFX nos permite además personalizar mediante CSS el aspecto de la interfaz. En nuestro proyecto hemos creado un estilo oscuro y lo hemos aplicado a toda la interfaz, personalizando el aspecto de los controles estnadár de JavaFX para darle un aspecto acorde con la iconografía comercial del club. Aunque yo no he hecho uso de ésta posibilidad, es posible también usar librerías externas de controles como JFoenix, Afterburner.fx o JacpFX . En éste interesante artículo se nombran varias opciones para un aspecto visual refinado de nuestras aplicaciones en JavaFX.



<https://www.jrebel.com/blog/best-javafx-libraries>

- Uso del enfoque modelo-vista-controlador (model-view-controller) MVC. JavaFX simplifica y colabora en la aplicación del enfoque modelo-vista-controlador. JavaFX nos ayudará por tanto a conseguir un código acorde a las buenas prácticas de programación. Con JavaFX se genera un archivo XML que contiene la información necesaria de los layouts y controles de la interfaz. Para el diseño de las vistas (ventanas) emplearemos el JavaFX Scene Builder que, mediante drag and drop nos permitirá elaborar de forma sencilla las interfaces de nuestra aplicación generando los archivos XML de cada vista. En ése sentido es bastante similar al funcionamiento de Android Studio, IDE utilizado a lo largo de ciclo formativo, concretamente en la asignatura Programación Multimedia y Dispositivos Móviles. Indicaremos al JavaFX scene builder el archivo java que hará las veces de controlador de la vista y los métodos que serán disparados por los eventos. Una vez hecho esto usaremos la opción pertinente para que nos genere el código base del controlador automáticamente. De ésta forma se agiliza la programación y se evitan errores:



Opción para generar el controlador de la vista por el IDE.

Como se aprecia en la imagen, pulsando botón derecho sobre el archivo de la vista aparece la opción *make* *controller* que genera un esqueleto base para el desarrollo del controlador. El IDE nos genera las referencias para los controles y los métodos para los eventos para su manipulación y desarrollo a conveniencia.

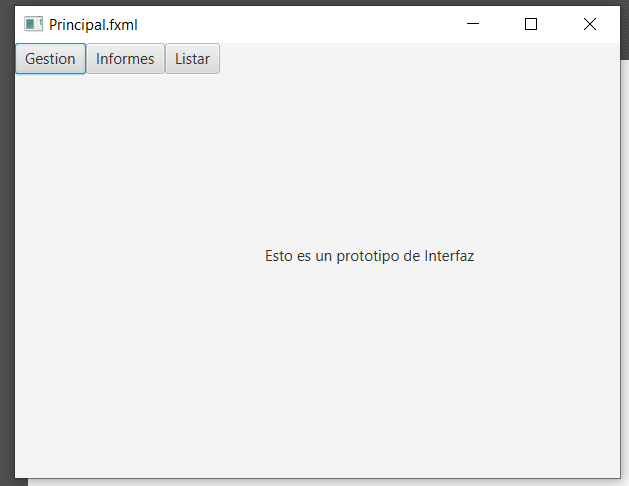
La aplicación hará uso de un repositorio SQL para almacenar la información de los usuarios del sistema así como la información característica de los socios que forman el club. El programa accederá a una base de datos SQL en la que existirán tablas para almacenar de forma fiable tanto los usuarios para autentificación con sus contraseñas como los datos de los socios. Así conseguiremos ACID.

**6. Planificación del proyecto: etapas de desarrollo del mismo, con el mayor nivel de detalle posible: especificación de requisitos, análisis, diseño, implementación, pruebas y explotación. Incluye aquí cuestiones relativas a ficheros de configuración, diagramas de clases, estructura del proyecto, wireframe, mockups, diagramas de uso entre otros.**

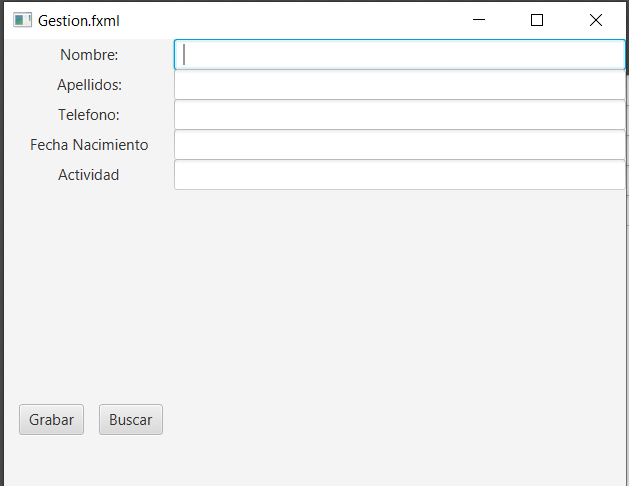
6.1.-ANALISIS. Aquí se trata de establecer el producto a desarrollar. Tenemos que establecer las funcionalidades deseadas mediante la toma de requisitos. Para ello, dentro de ésta fase elaboraremos **diagramas de flujo** para precisar el comportamiento del sistema de forma gráfica. Serán de gran utilidad los denominados diagramas UML: los diagramas de casos de uso, los diagramas de secuencia y colaboración, los diagramas de cambios de estado y los diagramas de actividad.

También se elaboraran **modelos de datos** que nos servirán para conocer las estructuras de datos y sus características haciendo uso de los modelos entidad-relación y formas normales. Se creará un **diccionario de datos** que reflejará todos los objetos utilizados en los diagramas y estructuras de datos mencionados anteriormente, en nuestro caso debemos incluir las clases propuestas en el diagrama.

Finalmente definimos las **interfaces de usuario** determinando la entrada y salida de datos dando paso a la siguiente fase. Sirvan como ejemplo los siquientes prototipos de ventanas a modo de bosquejo, a falta de concretar diseño final y posibles funcionalidades adicionales:



Prototipo de ventana principal de la aplicación elaborado en JavaFX Scene Builder



Prototipo de ventana de gestión elaborado en JavaFX Scene Builder

2.-DISEÑO. Éste será el momento de precisar todo el trabajo anterior teniendo en cuenta ya los recursos físicos y lógicos del entorno donde la aplicación va a ser ejecutada. En el **diseño externo** especificaremos la interfaz de usuario así como los formatos de entrada y salida de información. Tocaría ahora entrar en detalle sobre la interfaz.

La ventana principal está formada por :

-El logo del club en la parte superior.

- Una casilla de texto para búsquedas interactivas por texto.

-dos botones de selección para filtrar la lista de socios por activos y morosos.

-una tabla donde se mostraran al usuario un listado los datos principales de los socios.

-en la parte derecha encontramos los botones de generar PDF con los morosos, agregar nuevos socios, ver información completa de socio, modificar socio seleccionado y eliminar socio.

-por ultimo en la parte inferior aparece una etiqueta que servirá para dar feedback al usuario.



Ventana principal de la aplicación.

Como puede apreciarse hemos personalizado los controles JavaFX mediante el uso de tecnología CSS. Como comentamos anteriormente ésta fue una de las ventajas por las que optamos por JavaFX. Hemos creado para tal fin el archivo *temaoscuro.css* como comentaremos en el apartado de codificación.

La ventana “agregar nuevo socio” se despliega cuando el usuario pulsa el botón correspondiente en la “ventana principal” y está formada por:

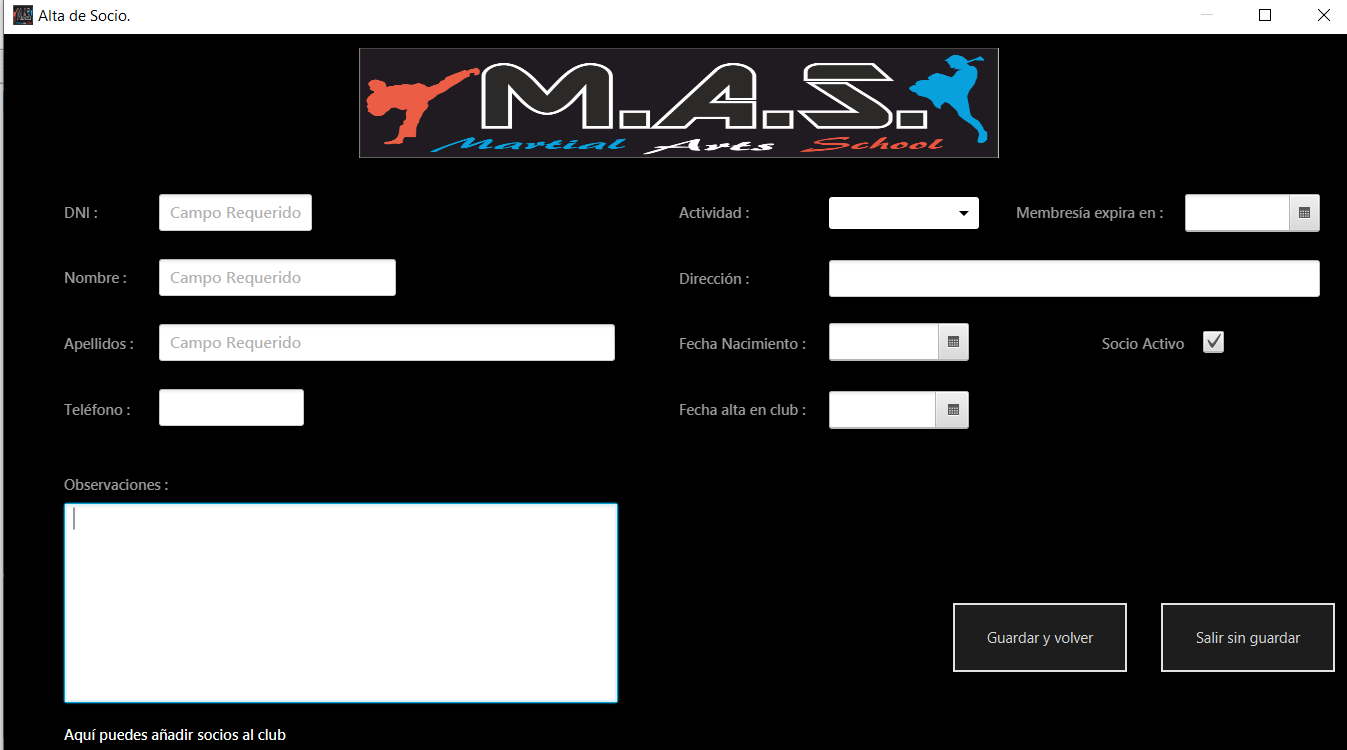
-El logo del club en la parte superior.

-Campos de texto para que el usuario introduzca los valores de los atributos que definen a un socio, junto a etiquetas identificativas con los nombres de los mismos.

Podemos identificar campos de texto sencillos, de gran extensión para las observaciones, datapickers para las fechas y una casilla tipo checkbox para el valor del atributo activo marcada por defecto. Cabe mencionar que cuando la casilla de socio activo se desmarca aparece un datapicker con la fecha de baja en el club inicialmente vacía.

-dos botones que permiten al usuario crear y almacenar al socio cuyos datos se han introducido

-por ultimo en la parte inferior aparece una etiqueta que servirá para dar feedback al usuario.



Ventana “alta de socio” con la casilla de socio activo marcada.



Ventana “alta de socio” con la casilla de socio activo desmarcada y el datapicker de fecha de baja.

La ventana “detalles de socio” se despliega cuando el usuario pulsa el botón correspondiente en la “ventana principal” y está formada por:

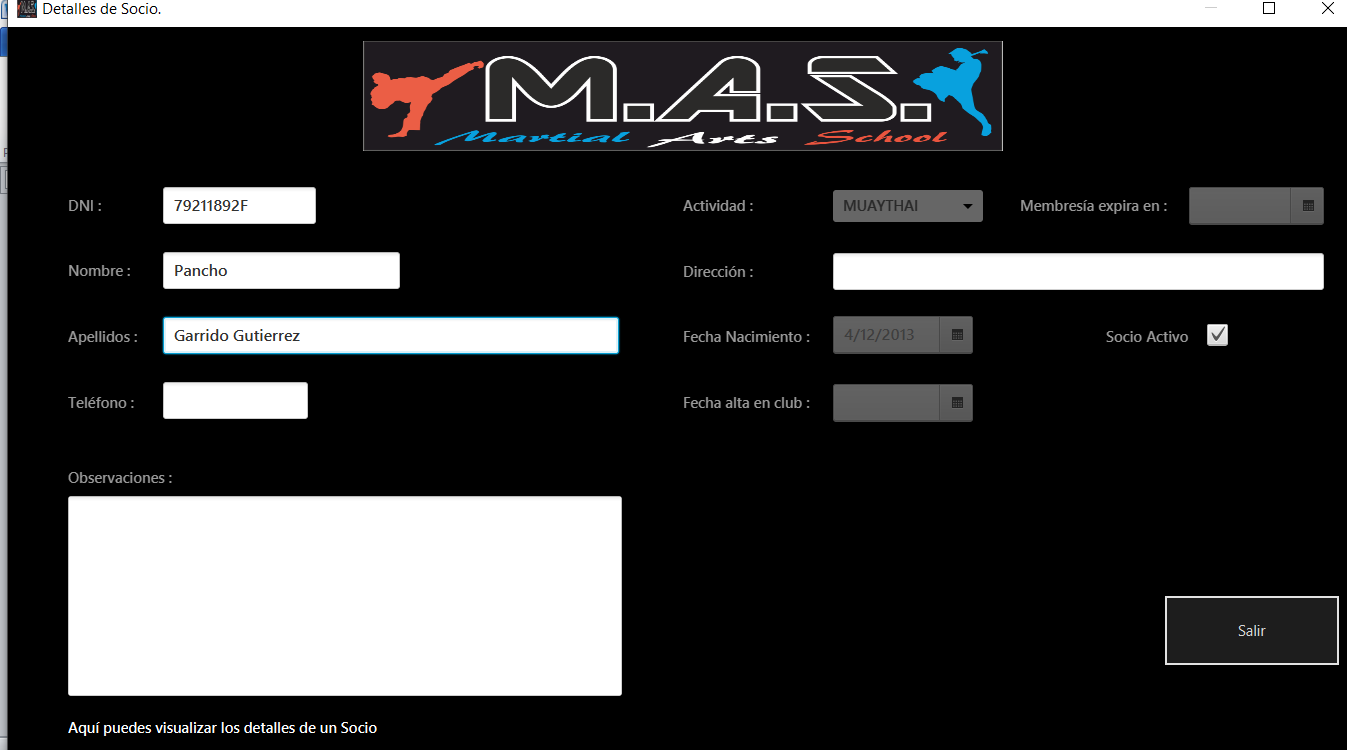
-El logo del club en la parte superior.

-Campos de texto que muestran los valores de los atributos del socio seleccionado en la pantalla principal, junto a etiquetas identificativas con los nombres de los mismos.

Podemos observar que los campos de texto sencillos, de gran extensión para las observaciones, los datapickers para las fechas y la casilla tipo checkbox se han modificado para que el usuario sólo pueda visualizar la información en ésta pantalla.

-un botón que permiten al usuario salir a la ventana principal una vez desee dar por finalizada la visualización de los detalles del socio seleccionado anteriormente.

-por ultimo en la parte inferior aparece una etiqueta que servirá para dar feedback al usuario.



Ventana “detalles de socio”.

La ventana “modificación de socio” se despliega cuando el usuario pulsa el botón correspondiente en la “ventana principal” y está formada por:

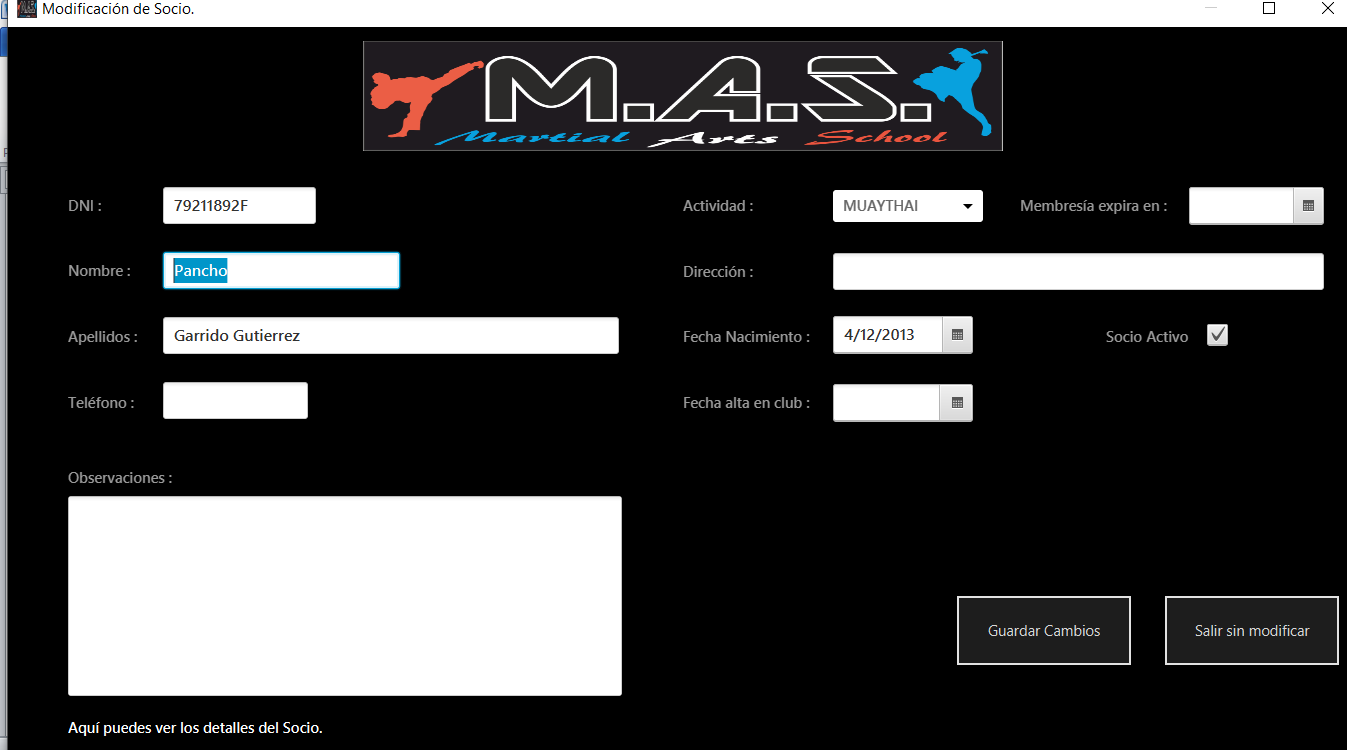
-El logo del club en la parte superior.

-Campos de texto para que el usuario visualice los valores actuales de los atributos que definen al socio seleccionado, junto a etiquetas identificativas con los nombres de los mismos.

Podemos identificar los mismos campos de texto sencillos, de gran extensión para las observaciones, datapickers para las fechas y la casilla tipo checkbox para el valor del atributo activo. Todos ellos permiten modificar el valor actual por otro nuevo llevando a cabo la modificación pertinente. Al igual que antes, lógicamente, cuando la casilla de socio activo se desmarca aparece un datapicker con la fecha de baja en el club.

-dos botones que en esta ventana permiten al usuario confirmar las modificaciones realizadas en los atributos del socio y salir sin guardar los cambios efectuados.

-por último, de nuevo en la parte inferior aparece una etiqueta que servirá para dar feedback al usuario.



Ventana “modificación de socio”.

Habría que hacer un diseño preliminar de la ventana principal de login y la interfaz propuesta cuando el usuario se haya logeado. También debemos especificar el formato deseado para el detalle de los listados.

En el **diseño de datos** estableceremos las estructuras de datos ahora ya teniendo en cuenta el Hardware y Software donde va a ser desplegada nuestra solución.

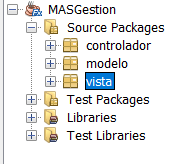
En el **diseño modular** dividiremos la aplicación en partes denominadas módulos. Definiremos preliminarmente el módulo de acceso o login, el módulo de gestión y el creación de listados.

En el **diseño procedimental** se establecerán las especificaciones para cada módulo descrito anteriormente y se escribirá el algoritmo necesario empleando técnicas de programación estructurada (seudocódigo, ordinogramas).

3.-CODIFICACION. Una vez tengamos el diseño de la aplicación teniendo claras las funcionalidades deseadas y habiéndose éstas descrito mediante los diagramas comentados, hay que traducir todo lo anterior al lenguaje de programación escogido. Habrá que elegir el entorno de desarrollo que deseamos así como otras tecnologías que queramos utilizar. Las primeras pruebas de funcionamiento tendrán lugar en esta fase. El sistema físico donde se podría instalar sería un ordenador personal de escritorio o portátil con conexión a internet.

Como ya comentamos anteriormente utilizamos el IDE NetBeans para implementar las funcionalidades con el lenguaje Java. Hacemos uso de JavaFX scene builder para las interfaces.

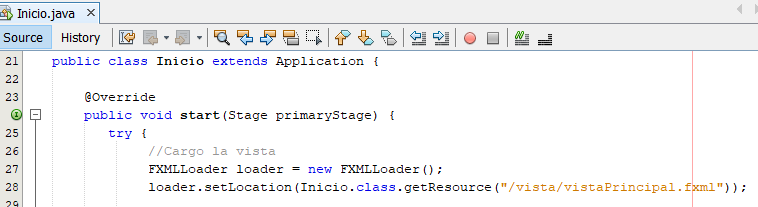
Para una correcta aplicación del enfoque modelo-vista-controlador creamos tres paquetes a los que denominador consecuentemente modelo, vista y controlador. Con ello mantendremos nuestros archivos correctamente clasificados y localizados dentro del proyecto:



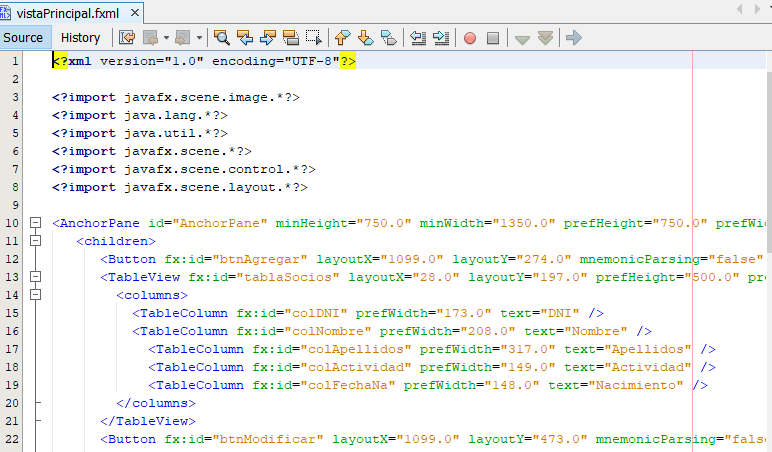
Paquetes para aplicar MVC.

**Paquete Controlador**

Nuestro proyecto JavaFX arranca con la clase *Inicio.java* dentro del paquete controlador que se encarga de cargar la vista *vistaPrincipal.fxml* que hemos diseñado anteriormente en JavaFX scene builder:



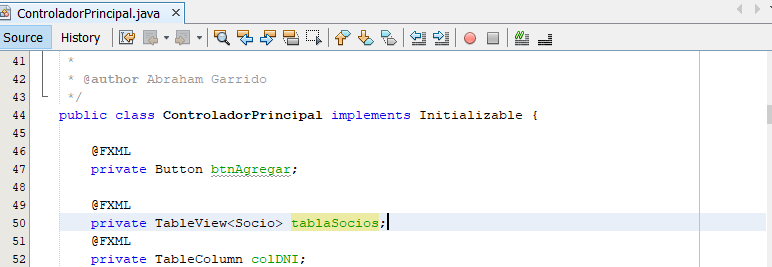
Carga de la vista principal de la aplicación.



Código xml generado por scene builder.

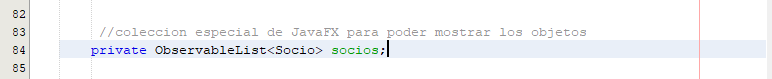
*ControladorPrincipal.java*

Una vez cargada la vista principal tenemos que acudir a la clase *ControladorPrincipal.java* para dar funcionalidad a la interfaz principal. En ella encontramos variables creadas por el asistente que referencian los controles que conforman la vista entre las que está *tablaSocios* y sus columnas:



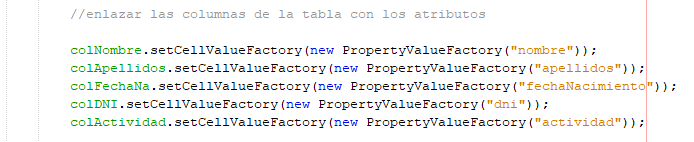
Clase *ControladorPrincipal*, parcialmente generada por JavaFX scene builder.

Definimos una variable de tipo *ObservableList* que denominamos socios y cuyo contenido se mostrará en el control *tablaSocios* :



Definición de la variable *socios* que nos permitirá almacenar los socios del club en tiempo de ejecución en un contenedor válido para su uso en el *TableView*.

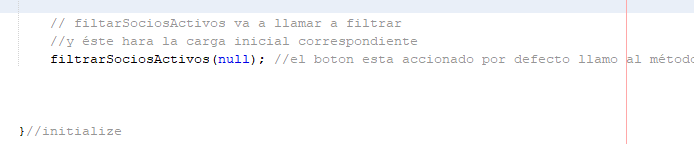
En el método sobreescrito *initialize* que se ejecuta a arrancar la vista indicaremos a las columnas que atributos les corresponde visualizar mediante el método *setCellValueFactory*:



Correspondencia de las TableColum con los atributos del modelo.

Esto lo hacemos de acuerdo con la clase *Socio* que supone el modelo de nuestra aplicación.

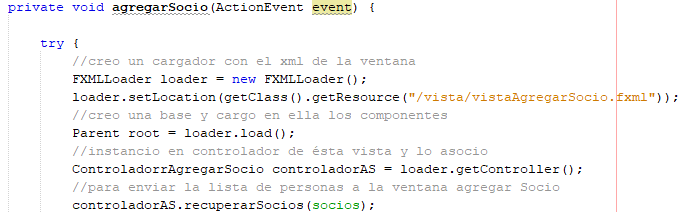
Al final de dicho método *initialize* se llama al método filtrarSociosActivos sin evento como argumento. Esto lo hacemos porque hemos dejado el togglebutton seleccionado por defecto para una experiencia más cómoda de usuario. Simulamos un click del usuario en el botón activos para que solo aparezcan los socios activos del club.



Llamada al método *filtrarSociosActivos* que simula la activación del botón por parte del usuario al inicializar la ventana en el método initialize.

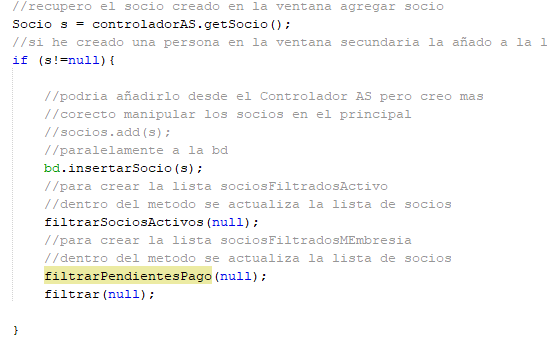
Dentro de la clase encontramos los siguientes métodos relevantes *agregarSocio, modificarSocio, eliminarSocio, mostrarDetalleSocio, filtrar, filtrarSociosActivos, filtrarPendientesPago y generarInformePDF*. Todos ellos recogen los eventos que genera el usuario en la ventana principal aportándoles la funcionalidad.

-public void agregarSocio(ActionEvent event). Éste método se ejecuta cuando el usuario acciona el botón “agregar nuevo socio”. Se encarga de cargar la ventana “agregar socio” (vista *vistaAgregarSocio*.*fxml*) y pasar al controlador de dicha ventana (*ControladorAgregarSocio.java*) de la misma la lista de socios.



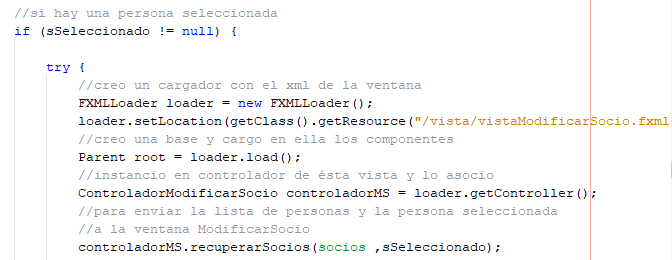
Fragmento relevante del método *agregarSocio* de la clase *ControladorPrincipal.java*

Posteriormente recoge el socio creado en la ventana “agregar socio” y lo añade a los socios del club. Después llama a filtrarSociosActivos y filtrarPendientes de Pago para que dentro de ellos se actualicen las listas de socios y sus filtradas sociosFiltradosActivo y sociosFiltradosMembresia. Terminamos pasando el flujo del programa al método *filtrar* que explicamos más adelante.



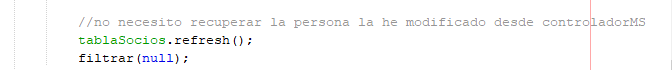
Fragmento relevante del método *agregarSocio* de la clase *ControladorPrincipal.java*

-public void modificarSocio(ActionEvent event). Éste método se ejecuta cuando el usuario acciona el botón “modificar socio”. Se encarga de cargar la ventana “modificar socio” (vista *vistaModificarSocio*.*fxml*) y pasar al controlador de dicha ventana (*ControladorModificarSocio.java*) la lista de socios y el socio previamente seleccionado el TableView tablaSocios de la vista principal.



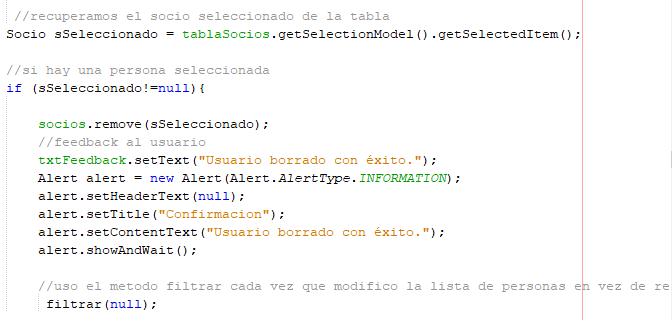
Fragmento relevante del método *modificarSocio* de la clase *ControladorPrincipal.java*

Posteriormente refresca el contenido de *tablaSocios* para que muestre las posibles modificaciones realizadas en el socio y pasa el flujo al método *filtrar.*



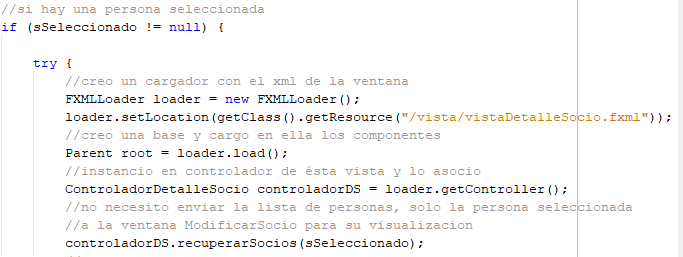
Fragmento relevante del método *modificarSocio* de la clase *ControladorPrincipal.java*

-public void eliminarSocio(ActionEvent event). ). Éste método se ejecuta cuando el usuario acciona el botón “eliminar socio”. Se encarga de eliminar al socio que se ha seleccionado previamente en la tabla y pasa el flujo al método *filtrar*.



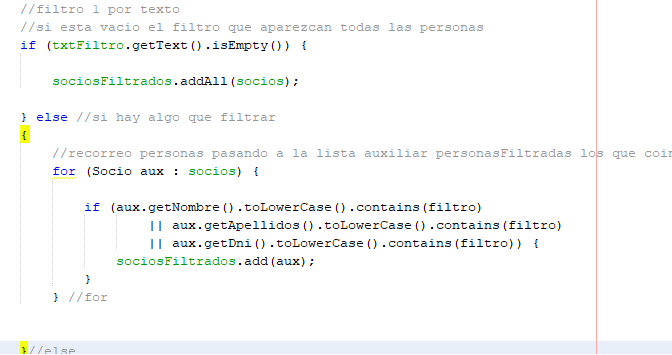
Fragmento relevante del método *eliminarSocio* de la clase *ControladorPrincipal.java*

-public void mostrarDetalleSocio (ActionEvent event). Éste método se ejecuta cuando el usuario acciona el botón “ver detalle socio”. Se encarga de cargar la ventana “detalles de socio” (vista *vistaDetalleSocio*.*fxml*) y pasar al controlador de dicha ventana (*ControladorDetalleSocio.java*) el socio previamente seleccionado el TableView *tablaSocios* de la vista principal. No es necesario llamar a filtrar o refrescar el contenido porque no se modifica en éste método.



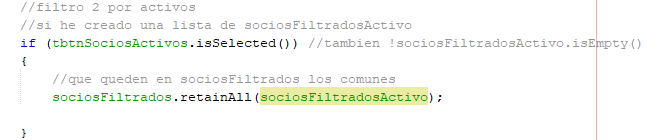
Fragmento relevante del método *mostrarDetalleSocio* de la clase *ControladorPrincipal.java*

-public void filtrar (ActionEvent event). Éste método es el que controla la visualización correcta del TableView *tablaSocios* después de cada evento que modifique su contenido de alguna forma, para ello actualiza la lista de socios desde la bd. Todos los métodos de evento lo llaman (con argumento null para el evento) al concluir su trabajo si modifican los datos de socios. Para ello modifica a conveniencia el contenido de la lista *sociosFiltrados*, que debe contener en todo momento los usuarios que visualiza el usuario en *tablaSocios* en tres pasos*.* En el paso 1, si la casilla de filtrado txtFiltro está vacía, la lista de socios completa socio pasara a formar la lista sociosFiltrados. Si contiene texto, solo aquellos socios que contengan los caracteres de la casilla en su DNI, Nombre o Apellidos formarán la lista sociosFiltrados.



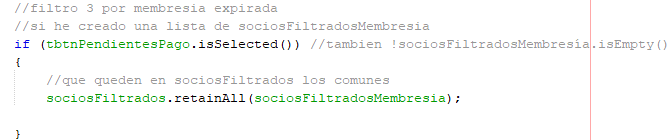
Primer filtrado por caracteres del método filtrar de la clase ControladorPrincipal.java.

Después de aplicar el primer filtro, pasamos al segundo que se aplicará si el botón “socios activos” está activo. Tomamos los elementos comunes de la lista *sociosFiltrados* (que ya ha sido modificada en el filtro 1 por caracteres) y la lista *sociosFiltradosActivos* (cuyo contenido ha sido fijado por el método de evento *filtrarSociosActivos*).



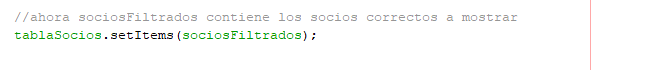
Segundo filtrado por socios activos del método *filtrar* de la clase *ControladorPrincipal*.*java*.

Por último, tras aplicar el segundo filtro, finalizamos con un tercer paso que se aplicará si el botón de “solo pendientes de pago” está activo. Tomamos los elementos comunes de la lista *sociosFiltrados* (que ya ha sido modificada en el filtro 1 por caracteres y el filtro 2 activos) y la lista *sociosFiltradosMembresia* (cuyo contenido ha sido fijado por el método de evento *filtrarPendientesPago*).



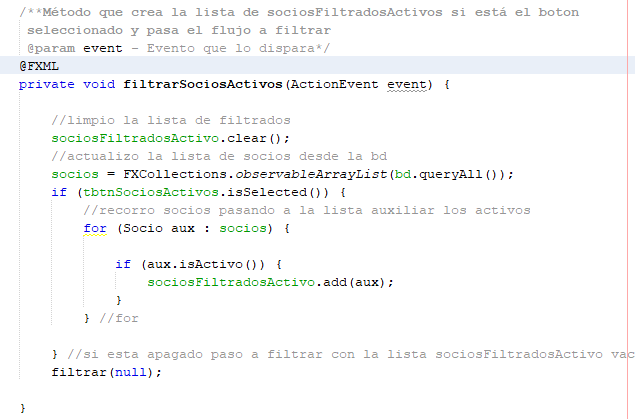
Tercer filtrado por socios con membresía expirada del método *filtrar* de la clase *ControladorPrincipal*.*java*.

Llegados a éste punto solo queda asignar a la tabla la lista que contiene los socios que ha de mostrar según los filtros que el usuario ha solicitado.



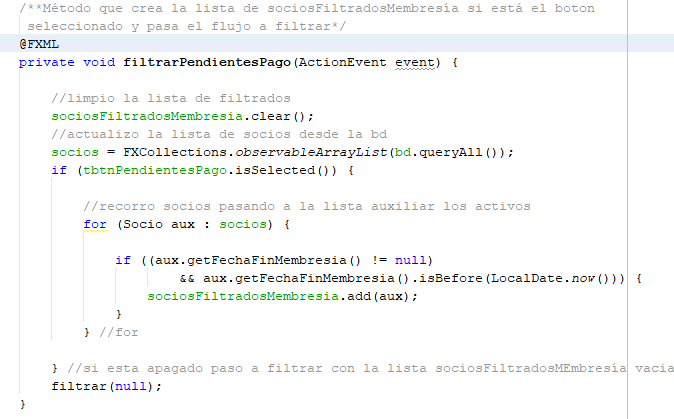
Finalmente asignamos los socios filtrados correctamente en la última parte del método *filtrar* de la clase *ControladorPrincipal*.*java*.

- public void filtrarSociosActivos (ActionEvent event). Éste método se encarga de formar la lista *sociosFiltradosActivo.* Si el botón está desmarcado la lista estará vacía. Luego pasa el flujo al método *filtrar*.



Método *filtrarSociosActivos* de la clase *ControladorPrincipal.java*

- public void filtrarPendientesPago (ActionEvent event). De forma análoga al anterior, do se encarga de formar la lista *sociosFiltradosMembresia.* Si el botón está desmarcado la lista estará vacía. Luego pasa el flujo al método *filtrar*.

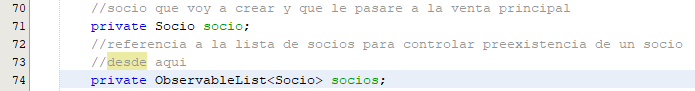


Método *filtrarPendientesPago* de la clase *ControladorPrincipal.java*

-public void generarInformePDF (ActionEvent event).

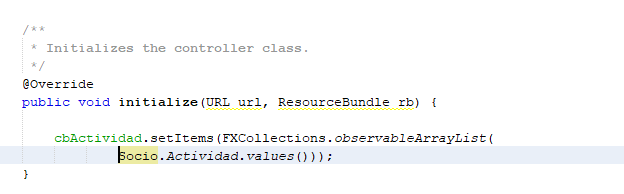
*ControladorAgregarSocio.java*

La clase *ControladorAgregarSocio.java* se encarga de la funcionalidad de la vista *vistaAgregarSocio.fxml*. Como ya hemos comentado en el apartado *ControladorPrincipal.java*, ésta ventana se inicia cuando el usuario pulsa el botón “agregar socio” en la pantalla principal de la aplicación. En la clase *ControladorAgregarSocio.java* encontramos las referencias a los controles y las cabeceras de los métodos que la opción *make controller* nos implementa automáticamente, así como otras variables y métodos que pasamos a describir en lo esencial. En cuanto a variables nos encontramos con una lista de objetos tipo Socio que denominados *socios*, de manera análoga a la lista de socios del *ControladorPrincipal.java*. Esto es así porque su misión será referenciar a la lista de socios con el mismo nombre, siendo en realidad referencias al mismo objeto ObservableList. Podremos controlar así si un socio que se pretende crear ya existe en el sistema, evitando por tanto duplicados. También observamos una variable tipo Socio denominada *socio*, en la que almacenaremos el socio que queremos crear y que como ya hemos descrito anteriormente recuperaremos desde el controlador principal mediante el método que se describe a continuación *recuperarSocio*.



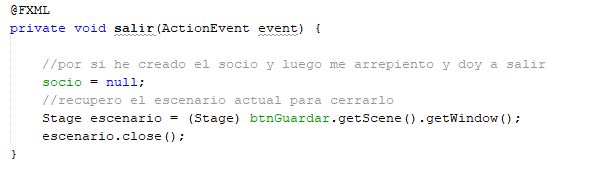
Variables para el trasvase entre ventanas de información de la clase *ControladorAgregarSocio.java*

En el método sobreescrito *initialize* que se ejecuta a arrancar la vista indicaremos al control *ChoiceBox* los posibles valores del enum Actividad que corresponde visualizar para su elección por parte del usuario.



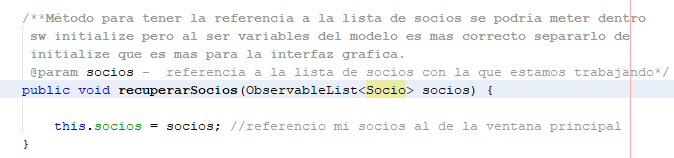
Método sobreescrito *initialize* de la clase *ControladorAgregarSocio.java*.

Dentro de la clase *ControladorAgregarSocio.java* encontramos los siguientes métodos relevantes *recuperarSocios, guardarSocio* y *mostrarFechaBaja.* Además nos encontramos con el método *salir* y los de acceso al atributo encapsulado *socio*. Éstos últimos tienen una funcionalidad evidente, con la salvedad de matizar que el botón de “salir” fuerza a la variable socio a valor *null*.



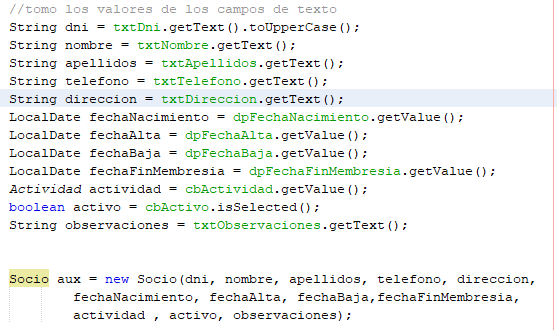
Método *salir* de la clase *ControladorAgregarSocio.java*.

- public void recuperarSocios (ActionEvent event). Éste método referencia la variable socios de la clase *ControladorAgregarSocio.java* a la que contiene el listado completo de socios de la clase *ControladorPrincipal.java*. Esto podría haberse hecho dentro del método *initialize* pero dado que éste último es más bien para la interfaz preferimos mantenerlo al margen.



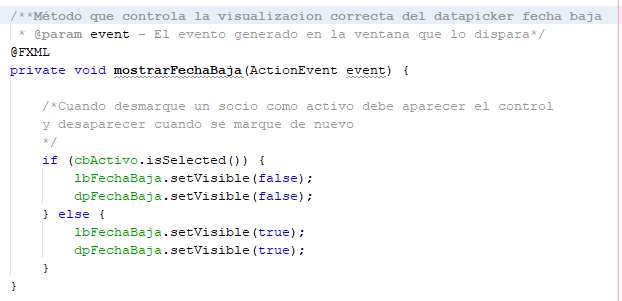
Método *recuperarSocios* de la clase *ControladorAgregarSocio.java*.

- public void guardarSocio (ActionEvent event). Éste método que recupera la información que ha introducido el usuario en las correspondientes casillas y construye un objeto tipo Socio. El método controla que no éste duplicado en el sistema y cierra la ventana “agregar socio”. Posteriormente se recupera desde la ventana principal para su almacenado como ya se explicó anteriormente.



Fragmento relevante de código del método *agregarSocio* de la clase *ControladorAgregarSocio.java*.

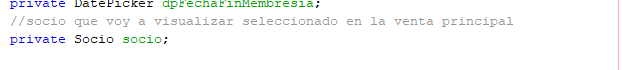
- public void mostrarFechaBaja (ActionEvent event). Éste método se encarga de que se visualice el control para la fecha de baja sólo cuando el socio no sea ya activo.



Método *mostrarFechaBaja* de la clase *ControladorAgregarSocio.java*.

*ControladorDetalleSocio.java*

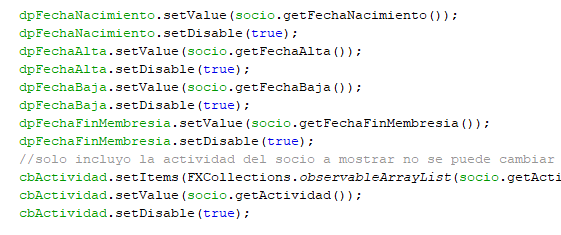
La clase *ControladorDetalleSocio.java* se encarga de la funcionalidad de la vista *vistaDetalleSocio.fxml*. Como ya hemos comentado en el apartado *ControladorPrincipal.java*, ésta ventana se inicia cuando el usuario pulsa el botón “ver detalle socio” en la pantalla principal de la aplicación, previa selección de un socio en la lista de socios. En la clase *ControladorDetalleSocio.java* encontramos las referencias a los controles y las cabeceras de los métodos que la opción *make controller* nos implementa automáticamente, así como una variable tipo Socio denominada *socio*, en la que almacenaremos el socio que el usuario había seleccionado en la ventana principal.



Variable socio para el trasvase entre ventanas de información de la clase ControladorDetalleSocio.java

Dentro de la clase *ControladorDetalleSocio.java* encontramos los métodos *recuperarSocio* y *salir* que simplemente cierra la ventana ya que en ésta vista no se modifica información alguna.

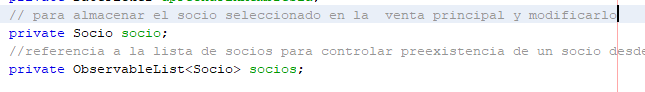
- public void recuperarSocio (ActionEvent event). Éste método nos sirve para pasar el socio seleccionado desde la ventana principal a la variable socio de la clase *ControladorDetalleSocio.java* y poder así mostrarlo por pantalla. En él se deshabilitan mediante código los controles pertinentes para que sea una autentica vista de visualización sin ser posible modificar nada.



Fragmento del método *recuperarSocio* de la clase *ControladorDetalleSocio.java* en el que se observa cómo se muestran los datos y se deshabilitan los controles.

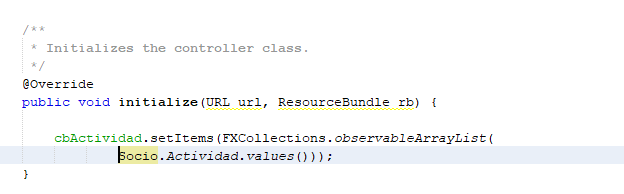
*ControladorModificarSocio.java*

La clase *ControladorModificarSocio.java* se encarga de la funcionalidad de la vista *vistaModificarSocio.fxml*. Como ya hemos comentado en el apartado *ControladorPrincipal.java*, ésta ventana se inicia cuando el usuario pulsa el botón “modificar socio” en la pantalla principal de la aplicación, previa selección de un socio en la lista de socios. En la clase *ControladorModificarSocio.java* encontramos las referencias a los controles y las cabeceras de los métodos que la opción *make controller* nos implementa automáticamente, así como otras variables y métodos que pasamos a describir en lo esencial. En cuanto a variables, al igual que en el anteriormente descrito *ControladorAgregarSocio.java*, nos encontramos con una lista de objetos tipo Socio que denominados *socios*. De manera análoga se referenciara a la lista de socios del *ControladorPrincipal.java*. De nuevo serán en realidad referencias al mismo objeto ObservableList. Controlaremos así que una modificación de un socio que se pretende efectuar no implique duplicar los socios existes en el sistema. También tenemos una variable tipo Socio denominada *socio*. En éste controlador *socio* nos servirá para obtener el socio seleccionado en la ventana principal que se desea modificar y efectuar las modificaciones deseadas por el usuario.



Variables para el trasvase entre ventanas de información de la clase *ControladorModificarSocio.java*

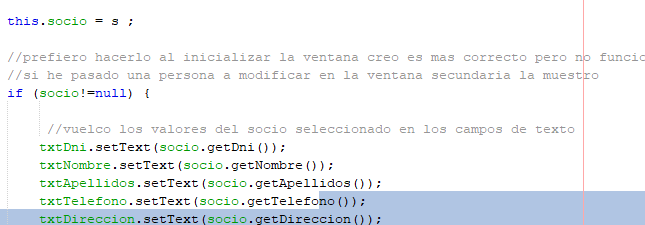
En el método sobreescrito *initialize* que se ejecuta a arrancar la vista indicaremos al control *ChoiceBox* los posibles valores del enum Actividad que corresponde visualizar para su modificación por parte del usuario.



Método sobreescrito *initialize* de la clase *ControladorModificarSocio.java*.

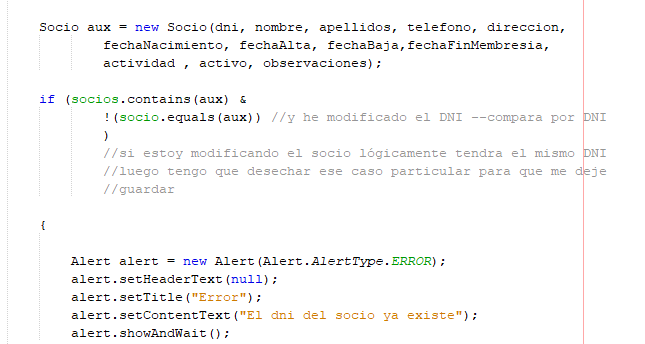
Dentro de la clase *ControladorModificarSocio.java* encontramos los siguientes métodos relevantes *recuperarSocios, guardarSocio, mostrarFechaBaja*. El método *salir* simplemente cierra la ventana.

- public void recuperarSocios (ActionEvent event). Éste método referencia la variable socios de la clase *ControladorAgregarSocio.java* a la que contiene el listado completo de socios de la clase *ControladorPrincipal.java*. Por otra parte recupera el socio seleccionado en la ventana principal y muestra sus valores por pantalla en el formulario, permitiendo su modificación por el usuario.

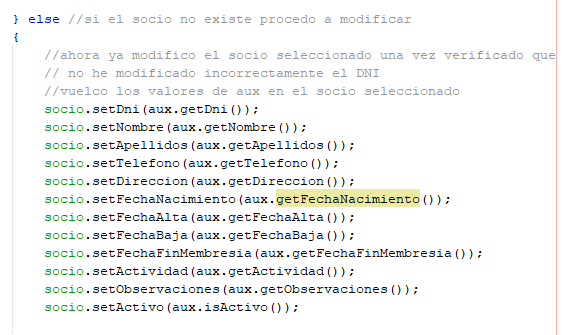


Fragmento del método *recuperarSocios* de la clase *ControladorModificarSocio.java*.

- public void guardarSocio (ActionEvent event). Éste método que recupera la información existente en las correspondientes casillas. Ésta información puede provenir de los valores que han sido volcadas desde el socio seleccionado o bien de alguna modificación por parte del usuario. En ambos casos construye un objeto tipo Socio. El método controla que no éste duplicado en el sistema y modifica directamente los valores del socio que se había seleccionado gracias a que le hemos pasado la referencia. Por último cierra la ventana “modificar socio”.



Fragmento relevante de código del método *agregarSocio* de la clase *ControladorModificarSocio.java*. Se crea un socio con los valores existentes en los controles y se comprueba que no exista el dni.



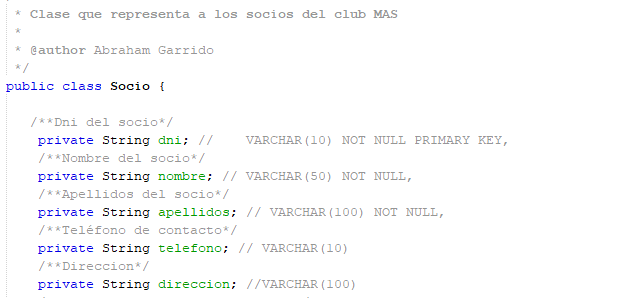
Fragmento relevante de código del método *agregarSocio* de la clase *ControladorModificarSocio.java*. Se vuelcan los valores tomados de los controles en el socio que se había seleccionado.

- public void mostrarFechaBaja (ActionEvent event). Comentado ya anteriormente en la clase *ControladorAgregarSocio.java*, éste método se encarga de que se visualice el control para la fecha de baja sólo cuando el socio no sea ya activo.

.

Paquete Modelo

Dentro del paquete modelo tenemos nuestra clase *Socio.java* que reproducimos parcialmente:



Comienzo de la clase Socio.java dentro del paquete modelo.

La clase *Socio.java* contiene los atributos que definen a un socio dentro del club deportivo Martial Arts School, según las especificaciones indicadas por el cliente y las funcionalidades solicitadas:

-Documento nacional de identidad del socio: Variable de tipo String denominada *dni.*

-Nombre del socio: Variable de tipo String denominada *nombre.*

-Apellidos del socio: Variable de tipo String denominada *apellidos.*

-Teléfono de contacto proporcionado: Variable de tipo String denominada *telefono.* Se almacena como cadena de caracteres porque no vamos a realizar cálculos con ella.

-Dirección completa del socio : Variable de tipo String denominada *direccion.*

-Fecha de nacimiento del socio: Variable de tipo LocalDate denominada *fechaNacimiento*. Optamos por éste tipo que nos ofrece java 8 para las fechas por su sencillez de gestión y operativa.

-Fecha de alta en club: Variable de tipo LocalDate denominada *fechaAlta.*

-Fecha de baja en club, cuando deja de estar activo independientemente de si tiene la membresía en vigor o no. Indica cuando un socio nos comunica que no va a seguir. Se puede estar pendiente de pago pero activo: Variable de tipo LocalDate denominada *fechaBaja.*

-Fecha fin de membresía, que nos indica cuando el socio debe abonar de nuevo su cuota: Variable de tipo LocalDate denominada *fechaFinMembresia*.

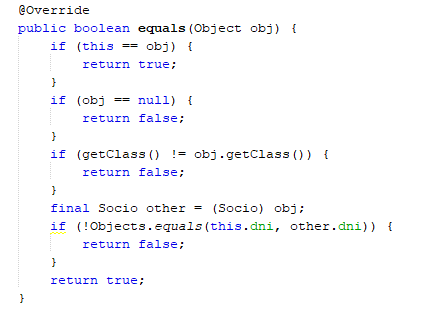
-Actividad del socio en el club de entre las que ofrece: Variable de tipo Actividad denominada *actividad*.

-Variable booleana para diferencia los socios activos de los que han causado baja. No indica si la membresía está en vigor. Se puede estar activo pero pendiente de pago: Variable tipo boolean denominada *activo*.

-Campo para observaciones adicionales sobre el socio: Variable tipo String denominada *observaciones*.

Dentro de la clase Socio.java encontramos además el tipo enum Actividad que recoge las actividades que los socios pueden realizar en el club con los posibles valores MUAYTHAI, TAEKWONDO, WINGCHUNG y YOGA.

Encontramos en ella también los métodos habituales para éste tipo de clase: contructores sobrecargados, getters y setters. Cabe destacar en este punto la sobreescritura del método *equals* de forma que considere iguales dos socios cuando su atributo *dni* coincida.



Método *equals* con la pertinente anotación *override* para poder comparar socios.

4.-PRUEBAS. Para que nuestra aplicación de gestión del club deportivo MAS tenga un nivel óptimo de calidad, es necesario que sea ampliamente probada. Una prueba es exitosa cuando descubre un fallo en nuestra aplicación y bajo esa premisa han de diseñarse. Elaboraremos pruebas bajo el enfoque de pruebas de caja blanca en las que se analiza la estructura interna del programa o bien de pruebas de caja negra en las que se tiene en cuenta únicamente las entradas y salidas del sistema sin tener en cuenta la estructura interna. Comenzaremos con las **pruebas de unidad** en las que testearemos los módulos en los que dividimos anteriormente nuestra aplicación. Comprobaremos el comportamiento por separado de los mismos y una vez concluidas con éxito pasaremos a las **pruebas de integración**. En ellas iremos comprobando la interconexión entre módulos. Para ello iremos integrando los mismos al programa de forma incremental para poder localizar y corregir errores los más fácilmente posible. La integración puede ser ascendente comenzando por los módulos de más bajo nivel integrándolos progresivamente en módulos mayores o descendente cuando comencemos por ventana principal y vayamos añadiendo módulos a medida que se van implementando. En la integración ascendente tenemos el inconveniente de no poder probar el programa hasta haber integrado todos los módulos mientras que en la descendente pueden surgir problemas cuando se requiere el procesamiento de algún módulo de nivel inferior para poder probar adecuadamente los de nivel superior. Teniendo en cuenta lo anterior, para nuestra aplicación optaremos por una integración descendente. Una vez probado el programa completo es el momento de realizar las **pruebas de sistema**. Bajo el paraguas de las pruebas de sistema se engloban una gran cantidad de pruebas a realizar como pueden ser las pruebas de aceptación por el cliente del programa completado, las pruebas de protección y seguridad, las de condiciones limite y recuperación, las de rendimiento… De éstas realizaremos las más adecuadas a nuestro proyecto teniendo en cuenta la relación esfuerzo/recompensa.

5.-EXPLOTACION Y MANTENIMIENTO. Llegados a ésta parte del desarrollo nos toca instalar nuestra app en el entorno final de trabajo y vigilar su adecuado comportamiento. También procede la entrega de toda la documentación generada en el proceso de desarrollo necesaria para la explotación de la misma. En el mantenimiento se culmina el ciclo de vida y en algunos casos se reinicia. En el mantenimiento correctivo se corregirá cualquier error no detectado anteriormente que haya surgido en el entorno de trabajo real, en el mantenimiento adaptativo se modificará el programa para adecuarlo a los cambios de entorno gráfico o lógico y en el mantenimiento perfectivo se tendrán en cuenta las propuestas de mejora de los usuarios.

**7. Conclusiones sobre el proyecto realizado y lo aprendido durante el desarrollo del mismo.**

**8. Trabajos futuros y/o posibles mejoras respecto a lo desarrollado y alcanzado.**

Funcion multiples socios han pagado desde la ventana principal.

Opcion desocultar contraseña en login.

**9. Referencias bibliográficas y/o webgrafía.**

En principio se trataba de poner en práctica los conocimientos adquiridos en el ciclo formativo DAM por lo que intentaremos ceñirme en todo momento al material aportado en las diferentes asignaturas y a las tecnologías que se han trabajado a lo largo del mismo. No creyendo que sea el momento de probar tecnologías nuevas y siendo más que suficiente lo aprendido en el ciclo para poder desarrollar una solución eficaz al problema de gestión planteado, la bibliografía ha intentado ajustarse básicamente a los apuntes de las diferentes asignaturas implicadas en el desarrollo de la aplicación. No obstante, como es lógico, hemos incluído en el proyecto cualesquiera aportaciones adicionales que han sido necesarias una vez que éstas se han producido.

Curso JavaFX discoduroderoer: <https://www.youtube.com/playlist?list=PLaxZkGlLWHGUWZxuadN3J7KKaICRlhz5->

Curso JavaFX code.makery:

<https://code.makery.ch/es/library/javafx-tutorial/>

Diseño interfaces JAVAFX con CSS:

<https://pixelduke.com/2012/10/23/jmetro-windows-8-controls-on-java/>

Librería de controles para JavaFX:

<https://www.jrebel.com/blog/best-javafx-libraries>

Curso de Java y MySQL de Códigos de Programacion:

<https://www.youtube.com/playlist?list=PL-Mlm_HYjCo_uketceEXL4CuoPf9t38FZ>

**10. Apéndices (opcional): incluye aspectos como código importante que se ha hecho referencia en la memoria, manual de usuario, manual de instalación, entre otros.**