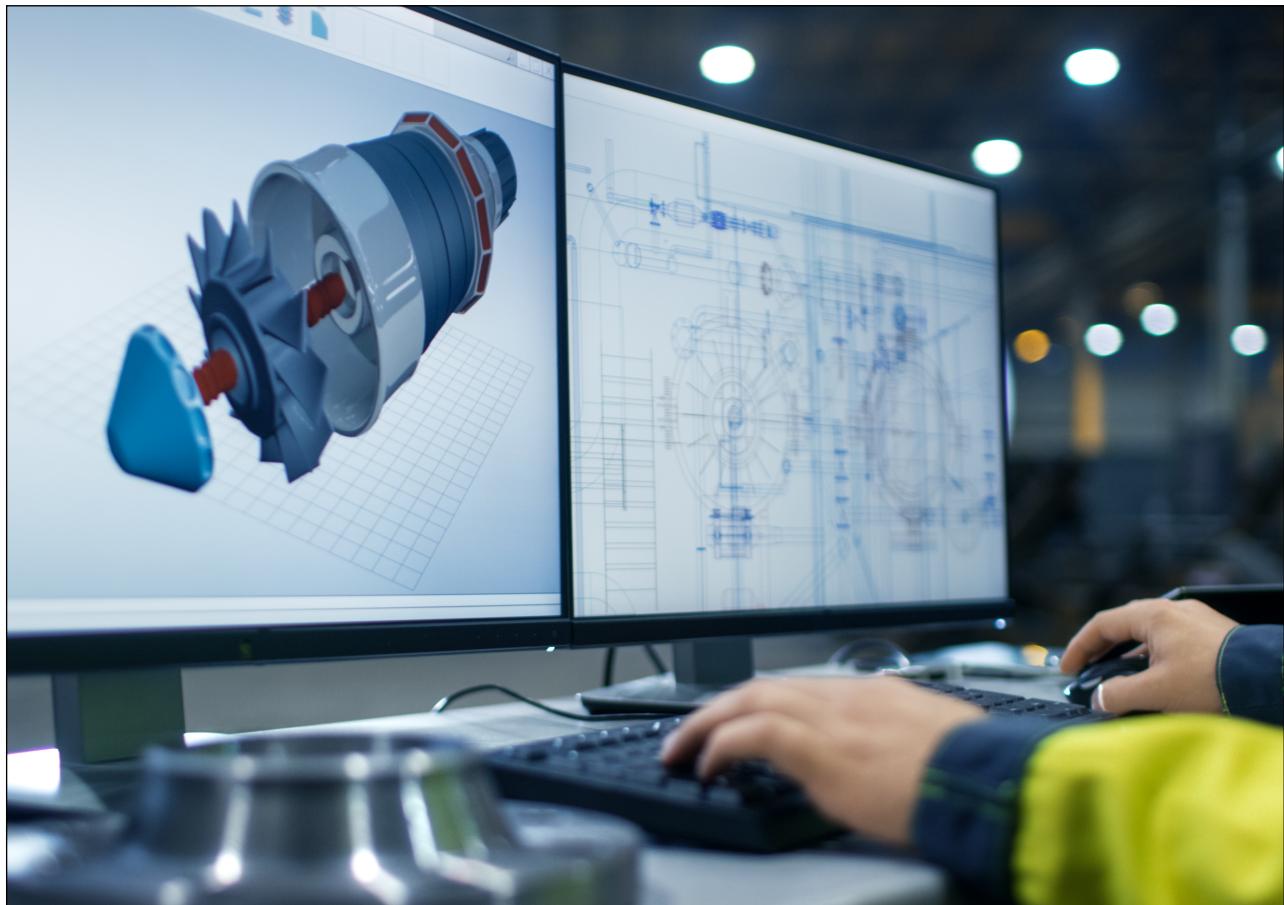


Modelado y Diseño del Software 2

Jesús Manuel Almendros Jiménez

Grado en Ingeniería Informática

Universidad de Almería



Índice

Índice	2
ACTIVIDAD 1: PROYECTO VAADIN	3
ACTIVIDAD 2: GENERACIÓN DE CÓDIGO	8
MODELO DE EJEMPLO	9
ACTIVIDAD 3: VISTAS DE LOS ACTORES	12
ACTIVIDAD 4: VISTAS DE LAS LISTAS	16
ACTIVIDAD 5: VISTAS DE OTROS CASOS DE USO	21
ACTIVIDAD 6: ENSAMBLADO DE COMPONENTES FIJOS	23
ACTIVIDAD 7: ENSAMBLADO DE COMPONENTES DINÁMICOS	25
ACTIVIDAD 8: GENERACIÓN DE CÓDIGO ORM	31
ACTIVIDAD 9: ACTUALIZACIÓN DE LOS CONSTRUCTORES	35
ACTIVIDAD 10: DIAGRAMAS DE SECUENCIA	38
ACTIVIDAD 11: IMPLEMENTACIÓN DE LOS COMPONENTS ORM	44
ACTIVIDAD 12: ACCESO A LOS COMPONENTES ORM	48
ANEXO: DIAGRAMAS DE SECUENCIA	50
ANEXO: UPLOAD	59

ACTIVIDAD 1: PROYECTO VAADIN

En esta actividad vamos a comenzar a trabajar en el Grupo de Trabajo primero creando los equipos de trabajo e instalando el software. A continuación revisaremos y enviaremos al profesor el proyecto diseñado en MDS1. Finalmente, tomaremos contacto con Vaadin, importando un proyecto existente y creando uno nuevo.

1. Alta en equipo de trabajo.

Se podrán crear equipos de trabajo de **uno o dos miembros**, preferiblemente con los mismos miembros que en MDS1. Aquellos que no hayan cursado MDS1 este año deben **preguntar al profesor como proceder**. Los equipos de trabajo se darán de alta a través de una hoja de inscripción disponible en el aula virtual.

2. Solicitud de la licencia Vaadin. Instalación del Software.

i) Se ha de solicitar la licencia de *Vaadin* a través del siguiente enlace:

<https://vaadin.com/student-program>

ii) Se ha de instalar el *Eclipse IDE for Enterprise Java and Web developers*.

iii) A continuación, desde desde el *Eclipse MarketPlace* se ha de instalar *Vaadin (Designer)*.

iv) Debemos tener Java instalado en nuestra máquina.

v) Debemos instalar *node.js LTS* en nuestra máquina:

<https://nodejs.org/es>

Una vez instalado *node.js* hay que reiniciar la máquina.

vi) Instalamos XAMPP:

<https://www.apachefriends.org/es/download.html>

3. Entrega del proyecto de MDS1.

Se ha de enviar a través de la actividad creada a tal efecto la documentación del proyecto de MDS1 tal y como se envió en esa asignatura.

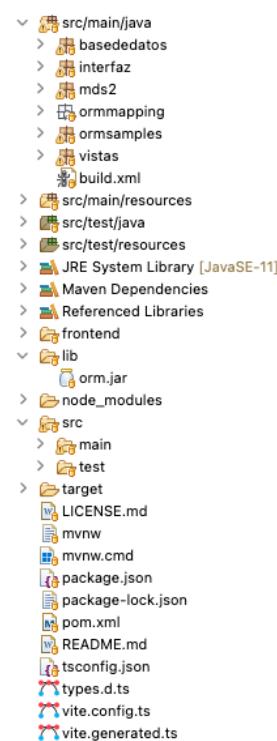
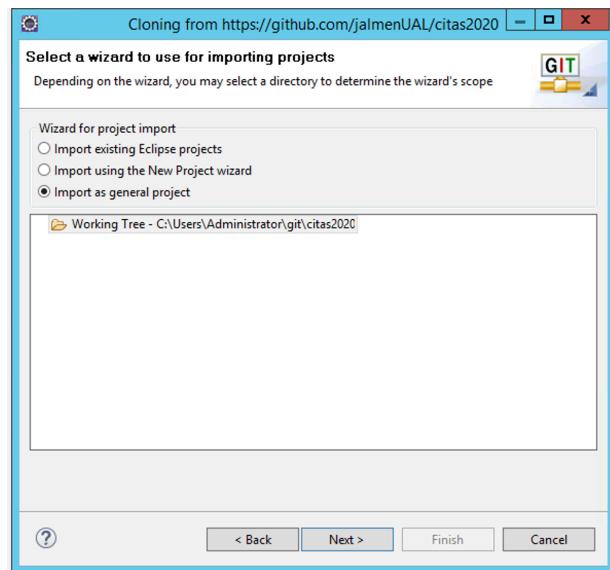
El proyecto de MDS1 hay que respetarlo, esto es, no se puede hacer ninguna modificación del mismo excepto si el profesor lo autoriza o lo recomienda.

Entrega 1 (Aula Virtual): ZIP con la documentación de MDS1

4. Importado del Ejemplo de Proyecto.

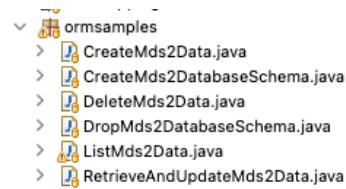
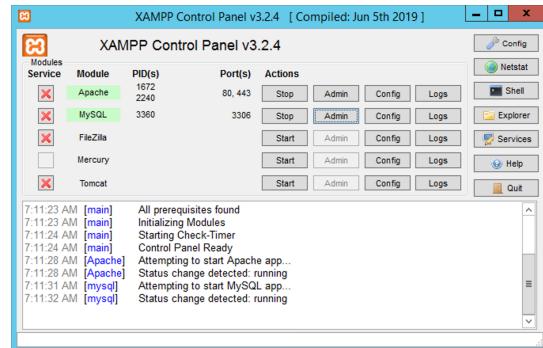
- i) Nos vamos a <https://github.com/jalmenUAL/ejemploMDSCompleto>
- ii) Copiamos la dirección del código del proyecto: git@github.com:jalmenUAL/ejemploMDSCompleto.git
- iii) Desde el menú principal de Eclipse, seleccionamos *File->Import->Git->Projects From Git->Clone URI->Next->Next*. Hay que elegir la subcarpeta de git donde se guarda el proyecto. Podéis dejar ejemploMDSCompleto.
- iv) Cuando lleguemos a la pantalla que se muestra (con las tres opciones de importado) damos a cancelar.
- v) A continuación, de nuevo, desde el menú principal de Eclipse: *File->Import->Maven->Existing Maven Projects* y buscamos la carpeta del Git local que nos ha creado anteriormente. Si todo ha ido bien nos habrá importado el proyecto importado como un proyecto *Vaadin*.
- vi) Veréis que nos da error en el código porque necesitáis añadir el *orm.jar* (está disponible en el aula virtual) al *build path* del proyecto. Para ello pulsamos en *Add External Jar* una vez situados en *ClassPath*. Ya tenemos importado el ejemplo de proyecto.
- vii) Ahora hay que descargar del aula virtual el fichero VisualParadigm del proyecto. Abridlo para poder comparar el proyecto VP con el proyecto Eclipse.
- viii) En el aula virtual también se encuentra el zip de la documentación del proyecto. Lo podéis descargar.

- A. En Eclipse, en la carpeta *mds2* está el programa principal (el que hay que ejecutar) que se llama *Application.java*. Además hay otro programa llamado *MainView.java*. Aunque el programa principal es *Application.java*, el código que se ejecuta está en *MainView.java*.
- B. En la carpeta *interfaz* están las clases de la interfaz de usuario del proyecto de Visual Paradigm (las generadas con el plugin de MDS1).
- C. En la carpeta *vistas* están las vistas java que



genera *Vaadin*.

- D. En la carpeta *basededatos* está la base de datos.
- E. En las carpetas *ormmapping* y *ormsamples* forman parte de la librería ORM. La carpeta *ormmapping* contiene ficheros de configuración del ORM. En *ormsamples* hay una serie de programas Java que sirven, entre otras cosas, para crear la base de datos y borrarla. También hay programas Java de ejemplo.
- F. Finalmente, en la carpeta *frontend/views* están las vistas ts de *Vaadin*.
- viii) Para ejecutar el programa antes hay que arrancar el MySQL del XAMPP. También necesitaremos Apache si queremos administrar la base de datos desde *phpMyAdmin*.
- ix) Debemos entrar en *phpMyAdmin* y dar de alta una base de datos que se llame "mds2". La base de datos ha de dejarse vacía.
- x) Ahora hay que crear el esquema de la base de datos "mds2". Esto se hace ejecutando el programa Java *Create...DatabaseSchema.java* que aparece en la carpeta *ormsamples*. Una vez ejecutado este programa, ya tenéis el esquema de "mds2" en MySQL. Podéis comprobarlo en *phpMyAdmin*.
- xi) Ahora solo queda ejecutar el programa *Application.java* como *Run As->Java Application*. Al ejecutarlo, podéis abrir la aplicación en: <http://localhost:8080>.
- xii) Una vez arrancado el proyecto ahora podemos echar un vistazo a los .ts de la carpeta *frontend/views*.



FAQ: Cuando creamos o descargamos un proyecto Vaadin, se ha de ejecutar el programa al menos una vez para que instale las librerías de Vaadin (las que están especificadas en el pom.xml). Si no se ejecutado el programa, no podremos ver las vistas, que necesita las librerías de Vaadin. Si compartimos el proyecto con un compañero/a, y lo tenemos en GitHub deberemos ejecutarlo al menos una vez.

FAQ: A veces es necesario (colocándose sobre el nombre del proyecto) hacer Maven->Update Project. Esto nos asegura que se ha actualizado el proyecto adecuadamente.

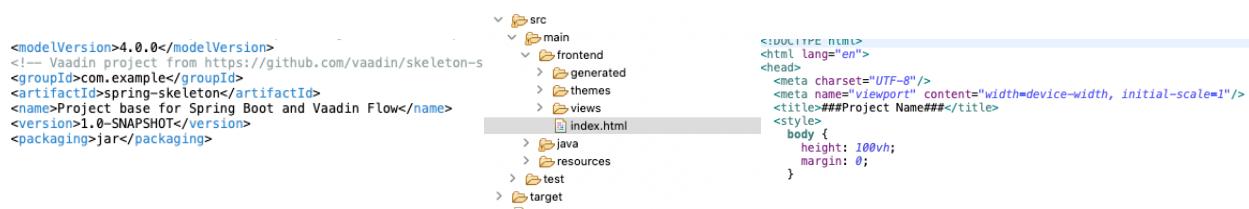
- xii) De momento no vamos a poder entrar en la aplicación en modo administrador, dado que no disponemos de credenciales. Si lo podemos hacer como usuario registrado en el botón "Registrarse". El usuario administrador se ha de dar de alta de forma manual en el *phpMyAdmin*. Para hacerlo hay que insertar en la tabla "registrado" (que sirve para todos los registrados de la aplicación) y luego especificar que es administrador, insertando en la tabla "administrador". La tabla "administrador" solo contiene los identificadores de los registrados que son administrador.
- xiii) Lo siguiente que vamos a hacer es:
- Registrar **dos** usuarios en la aplicación (en el botón "Registrarse")
 - Acceder con los dos usuarios y añadir dos elementos a la lista (dos elementos por cada usuario).

- C. Entrar como administrador y **tomar captura** de los cuatro elementos añadidos a la lista; y borrar un elemento de cada usuario registrado. Una vez borrados hay que **tomar captura** de la lista con los elementos restantes.

Entrega 2 (Aula Virtual): Capturas de las ventanas.

Creación del Proyecto Vaadin.

- En el aula virtual está disponible el esqueleto de un proyecto Vaadin. Tenemos que descargar este esqueleto e importarlo como proyecto Maven existente: *File->Import->Maven->Existing Maven Projects*.
- A continuación configuraremos este proyecto como sigue. Cambiamos el *name* en el



pom.xml y también el nombre en el *index.html*. Así mismo, hacemos *refactor* de la carpeta *org.vaadin.example* y poner el nombre que queramos (puede ser el mismo en todos lados). Es importante cambiar el nombre porque el profesor podrá importar el proyecto a su equipo.

- Una vez importado el proyecto, Eclipse generará una serie de carpetas dentro del proyecto. Entre las carpetas, las más importantes son: *src/main/java* donde se aloja todo el código Java de la aplicación; *frontend/views* y *frontend/styles* donde van los vistas de las ventanas y los estilos de las mismas. Estas son las únicas carpetas que vamos a modificar. El resto son gestionadas por *Vaadin*.
- De momento solo tendremos disponible en *src/main/java* los tres programas por defecto que genera Vaadin: *MainView*, *Application* y *GreenService*. El código de *Application.java* no hay que tocarlo (sólo cuando lo indique posteriormente). El único que hay que modificar normalmente es el *MainView.java*, que es el programa principal. Este programa de momento tiene un trozo de código que muestra un área de texto con un botón.
- Ahora hemos de ejecutar el programa *Application.java* (como no hay base de datos no hace falta arrancar el MySQL ni el Apache del XAMPP). En localhost:8080 debería mostrarse el área de texto con el botón.

FAQ: Los proyectos Vaadin se ejecutan haciendo uso de un servidor de aplicaciones. Las aplicaciones se arrancan por defecto en el puerto 8080. Si se desea cambiar, se puede hacer en el fichero *src/main/resources/applications.properties*. Si se intenta ejecutar dos veces la aplicación, va a dar error porque la segunda vez el puerto está ocupado. Por ello, se ha de parar la aplicación cada vez que se ejecuta, en el botón rojo y en las dos aspas negras que aparecen a su lado.

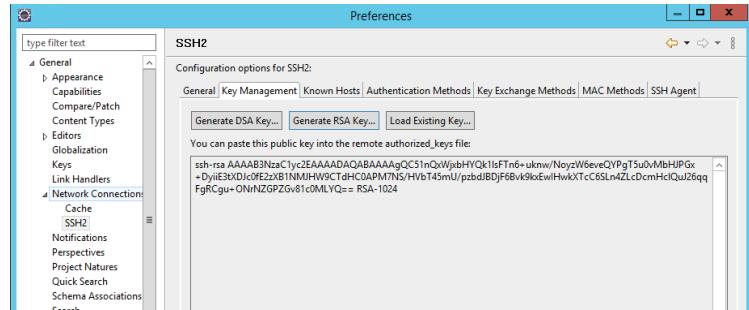
```

Application (4) [Java Application] /Users/jalmen/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.aarch64_23.0.1.v20241024-1700/jre/bin/java (27 nov 2024, 12:06:20) [pid: 31446]
2024-11-27 12:06:25.803 INFO 31446 --- [main] mds2.Application : Started Application in 4.626 seconds (JVM running for 5.09s)
2024-11-27 12:06:25.822 INFO 31446 --- [onPool-worker-1] c.v.b.devservr.AbstractDevServerRunner : Starting Vite
2024-11-27 12:06:26.203 INFO 31446 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost]./: Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-11-27 12:06:26.203 INFO 31446 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2024-11-27 12:06:26.205 INFO 31446 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
2024-11-27 12:06:26.259 INFO 31446 --- [nio-8080-exec-1] c.vaadin.flow.spring.SpringInstantiator : The number of beans implementing 'I18NProvider' is 0. Cannot
----- Starting Frontend compilation. -----
2024-11-27 12:06:26.795 INFO 31446 --- [onPool-worker-1] c.v.b.devservr.AbstractDevServerRunner : Running Vite to compile frontend resources. This may take a few minutes...
2024-11-27 12:06:27.941 INFO 31446 --- [v-server-output] c.v.b.devservr.DevServerOutputTracker : VITE v3.0.9 ready in 1113 ms
----- Frontend compiled successfully. -----
2024-11-27 12:06:27.942 INFO 31446 --- [v-server-output] c.v.b.devservr.DevServerOutputTracker :

```

Sin embargo, por defecto, si se hacen modificaciones en el código no hace falta parar la aplicación porque Vaadin recarga el programa de modo automático. Como veremos más adelante, esta recarga automática no es compatible con la librería orm que utilizamos, y tendremos que deshabilitar la recarga automática, pero mientras trabajemos sin base de datos se recargará de forma automática.

- v) Ahora guardaremos nuestro proyecto en el *Github*. Primero debemos generar una *RSA Key* en *Eclipse*, en *Preferences* de *Eclipse* como se muestra en la imagen. Y guardarla en cualquier carpeta. En *settings* del *GitHub* debemos ir a *SSH and GPG keys* y crear una nueva *SSH key* y copiar el código del *ssh-rsa* de *Eclipse* en esa key.



- vi) Ahora debemos crear un repositorio en nuestro *Github* y copiar el enlace *ssh* del repositorio. El repositorio se debe poner privado. A continuación nos colocamos con el ratón sobre el proyecto *Team->Share Project* y creamos un nombre para el repositorio. Pulsamos *Finish*. A continuación nos colocamos con el ratón sobre el proyecto *Team->Commit*. Ahí debemos copiar la dirección *ssh* del repositorio y seguir los pasos. No se necesita llenar los campos de autenticación porque se hace a través de la key. A partir de ese momento podemos hacer *Push* desde *Remote* (o desde el *Git Staging*).

FAQ: Es necesario para la ejecución de la aplicación para poder hacer Push.

- vii) Ahora vamos a compartir con nuestro compañero de equipo el proyecto, añadiéndolo como colaborador del proyecto *Github*.
- viii) A partir de ahora se van a hacer algunas entregas a través de **Github**. El enlace al repositorio de *Github* donde hay que enviarlo estará disponible en la actividad correspondiente del aula virtual. El repositorio que habéis creado es para vosotros, y no se envía. En cada entrega se hará una copia de lo que se pide y se colocará en el repositorio de la entrega. Esto no significa que el profesor pueda tener acceso en el algún momento al repositorio completo de *Github*.

Entrega 3 (*Github*): Proyecto Vaadin.

ACTIVIDAD 2: GENERACIÓN DE CÓDIGO

En esta actividad vamos a empezar a implementar en *Vaadin* la interfaz de usuario de la aplicación.

Generación de código del interfaz de usuario.

- i) Nos vamos a *Visual Paradigm* y exportamos las clases de la *interfaz de usuario*. Nos vamos a *Tools -> Code -> Instant Generator*. Aquí seleccionamos en *Model Elements* todas las clases que pertenezcan al interfaz de usuario. Se elige una carpeta de nuestra computadora donde generar el código (*output path*) y se genera el código. No se eligen las que forman parte de la base de datos (*bd_principal*, *iAdministrador*, etc, ni las *ORMPersistable* ni las *ORMComponent*).
- ii) A continuación se abre *Eclipse* y se importa el código de la carpeta que acabamos de generar en el proyecto *Vaadin*: *Import->General-> File System->(Selección de la carpeta elegida al generar el código)->Select All*. Este código se ha de alojar en la carpeta *src/main/java*. Debería aparecer en una carpeta/paquete dentro de esta carpeta. La carpeta tiene el mismo nombre que el paquete en el que estaba en *Visual Paradigm* (la que generó el plugin, o sea, *interfaz*). Veréis que nos da errores de compilación. Esto es debido a que aparecen en el código las componentes gráficas que introdujimos en MDS1. Debemos poner como comentarios el código de esas componentes gráficas. Esas componentes gráficas no las vamos a usar. Usaremos las componentes de *Vaadin* en su lugar. También dará error en las componentes de la base de datos que no hemos importado aún y que, por tanto, también pondremos entre comentarios.

Importante: Debe quedar todo el código sin errores de compilación. En otro caso no podremos ejecutar la aplicación Vaadin.

Generación de código para las vistas (ts y java).

- i) Instalamos en *Visual Paradigm* el *plugin de MDS2* que está disponible en el aula virtual. Al ejecutar el plugin hay que elegir una carpeta en la que se colocará el código generado.
- ii) A continuación creamos un paquete Eclipse que se llame *vistas* dentro de *src/main/java*.
- iii) Ahora hay que importar el código generado por el *plugin* a la carpeta *vistas*. Para ello nos colocamos con el ratón encima de la carpeta *vistas* y hacemos *Import->General-> File System->(Selección de la carpeta elegida al generar el código)->Select All*.
- iv) El código se habrá colocado en la carpeta *vistas* incluyendo los *.java* y *.ts*. Ahora movemos en *Eclipse* el código *.ts* de la carpeta *vistas* a la carpeta *frontend/views*.
- v) A continuación probamos que todo se ha generado de forma exitosa. Abrid cualquier *.ts* de la carpeta *frontend/views*. La vista debería estar vacía y el editor listo para editar la vista.

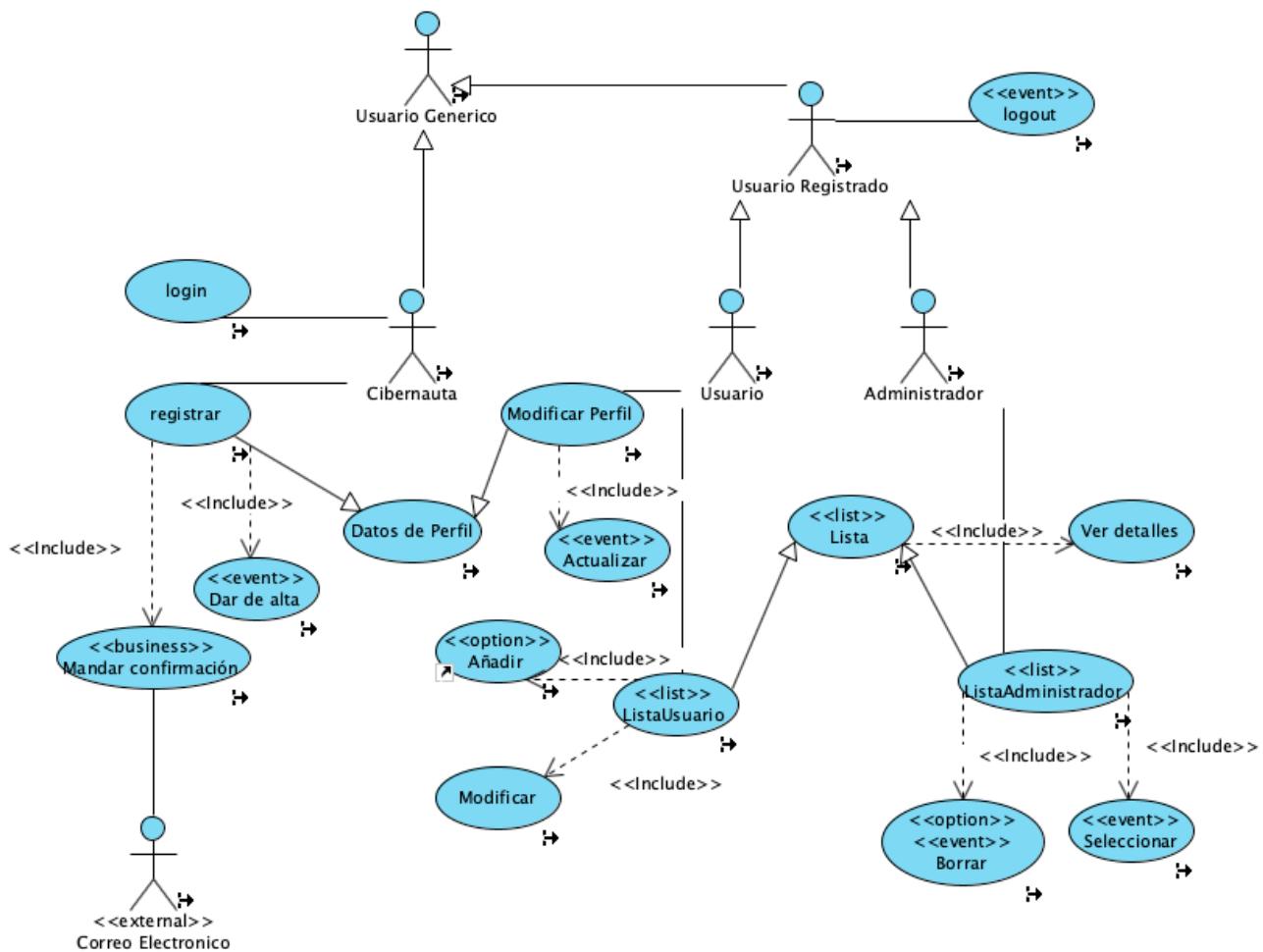
Entrega 4 (Github): Vistas, interfaz y frontend/views.

MODELO DE EJEMPLO

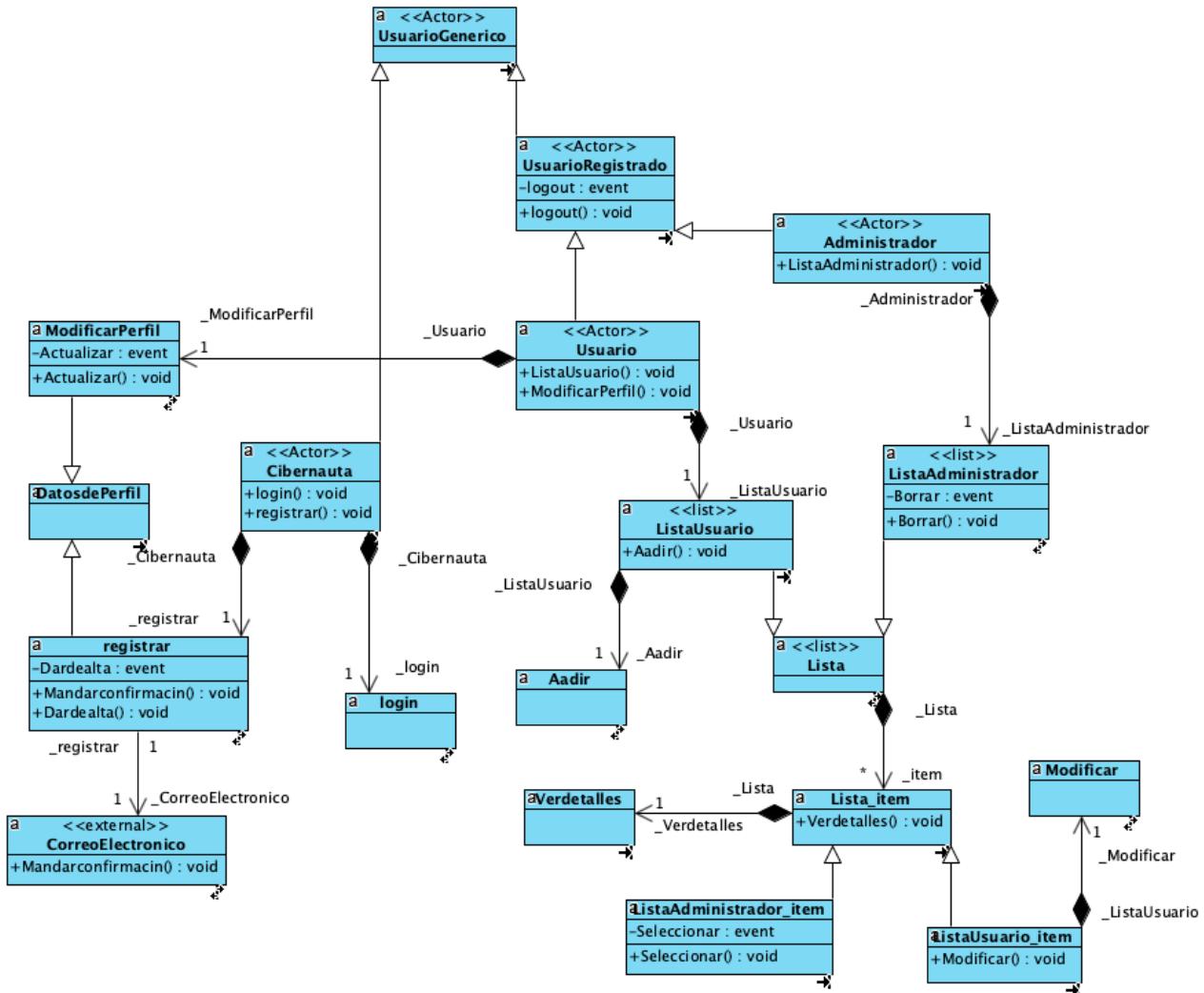
El ejemplo que hemos importado anteriormente nos va a servir de modelo para el desarrollo de nuestro proyecto. Este ejemplo sencillo incluye elementos que suelen ser comunes a todos los proyectos que se modelan en MDS1 y son luego desarrollados en MDS2.

Aunque no es objetivo de esta asignatura volver a describir los conceptos estudiados en MDS1, si haremos un breve resumen aquí con el objeto de entender cómo se van a desarrollar el resto de actividades de MDS2.

En el proyecto de MDS1 partimos principalmente de un modelo de casos de uso que describen la funcionalidad de la aplicación y, también, partimos de un modelo de clases de la interfaz que ha sido generado de forma automática por el *plugin* instalado en Visual Paradigm. Este modelo de clases tiene como objetivo proporcionar una implementación de la interfaz de usuario.



Además, partimos de un modelo de la base de datos, en la que se ha especificado los datos que se manejan como entidades y relaciones, y contenedores para los mismos. Así mismo se ha incluido un contenedor general, lo que llamamos BD principal e interfaces Java al mismo para poder acceder a los datos.



De momento nos vamos a centrar en los dos primeros modelos: el modelo de casos de uso y el modelo de clases.

En el ejemplo de proyecto, estos modelos son como se muestran en las figuras.

En esencia, se trata de una aplicación sencilla en la que se maneja una lista en la que los usuarios pueden escribir un texto. Esta lista es visible para los usuarios y el administrador, siendo ambos usuarios registrados. Sin embargo, existe otro tipo de usuario que no está registrado, que es el cibernauta. Todos los usuarios (registrados y cibernautas) son usuarios genéricos. Esto está especificado por la relación de herencia. Por otro lado, la lista en la que se escribe muestra todo lo que se ha escrito en la aplicación por todos los usuarios, pero aquellas cosas que escribe un usuario pueden ser sólo modificadas por él. El administrador no puede modificarlas pero sí las puede borrar. El cibernauta no puede

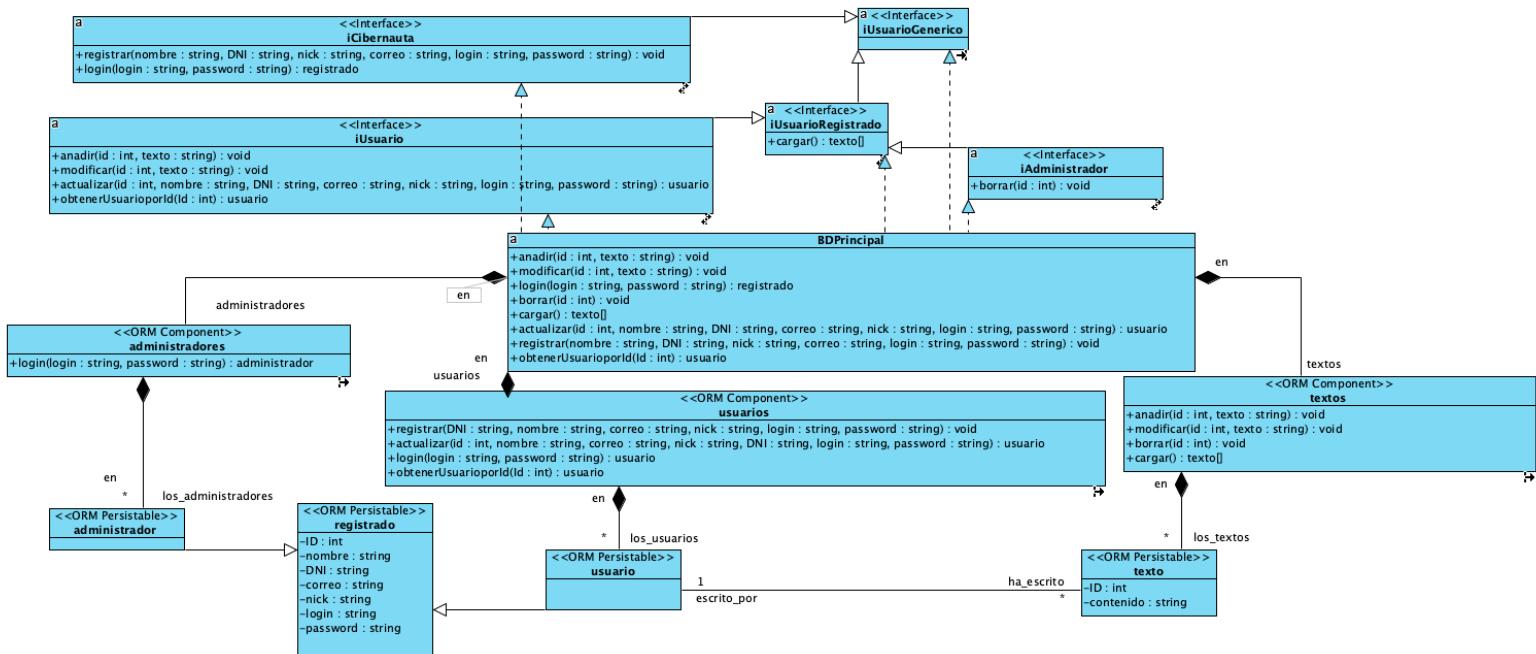
ver la lista. Para poder verla ha de registrarse y en ese momento pasa a ser usuario y puede hacer login. Al registrarse ha de llenar sus datos de perfil que luego puede modificar. Una vez registrado recibe un correo de confirmación de registro. Dado que cada uno (usuario y administrador) puede hacer cosas distintas en la lista hay dos tipos de listas: la lista de usuario y la lista de administrador y a su vez los ítem de la lista son distintos (unos se pueden modificar y otros no). Por tanto hay dos herencias más, unas que heredan de una lista genérica y otras que heredan del ítem de la lista. Finalmente, tanto el registro como la modificación del perfil son un caso particular de datos de perfil y por tanto, también hay herencia.

Como ejemplo que es pretenden cubrir los casos principales que se pueden dar en un diagrama de casos de uso: actores, herencia entre actores, casos de uso, herencia entre casos de uso, listas, herencia entre listas y finalmente los posibles estereotipos: event, option y business.

En vuestro proyecto es probable que tengáis muchas de estas características pero igual no todas. A lo largo de las actividades explicaré que hacer con cada una de ellas, incluido cuando que no se tengan.

Para completar el ejemplo de modelo, muestro el modelo de la base de datos del ejemplo. En esencia permite almacenar los usuarios y administradores como caso particular de registrados y los textos que aparecen en la lista junto a su propietario.

A la parte de la base de datos nos dedicaremos más adelante.



ACTIVIDAD 3: VISTAS DE LOS ACTORES

En esta actividad vamos a comenzar a hacer las vistas de la aplicación. O dicho de otro modo, vamos a editar los .ts que hemos generado con el plugin de MDS2. Sin embargo seguiremos un cierto orden por varias razones.

1. La primera es que es importante saber distinguir en el proyecto diferentes fases. Tener fases separadas con comienzo y fin nos va a permitir mejor organizar el trabajo.
2. La segunda razón es porque en cada fase se aplican criterios distintos.
3. La tercera razón es para que seamos capaces de crear componentes separados en cada fase que luego ensamblaremos.

Uno de los elementos principales en esta asignatura es la construcción incremental (a veces incluyendo refinamientos) y el ensamblado de componentes. No puede ser de otra manera para un proyecto de cierta envergadura.

1. Modelado de las vistas de los actores

Vamos a comenzar editando las vistas de los actores. Veremos que en la carpeta *frontend/views* hay ficheros .ts para algunos de los actores de nuestro diagrama de casos de uso. Sin embargo, pueden no aparecer todos. La razón es que solo hemos de hacer las vistas .ts para aquellos actores que no heredan de ningún otro actor.

[En el ejemplo de proyecto sólo hay un .ts para la vista del usuario genérico.](#)

¿Qué hacemos con los que heredan y no tienen vista? Pues debemos poner en la vista de la que heredan todos los eventos que necesitan. O sea, en cada vista debe aparecer la suma de todos los eventos que necesitan los actores que heredan. Esto incluye los casos de uso <<event>> conectados mediante inclusión al actor o los casos de uso sin <<event>> que necesitan un evento para dispararse.

[En el ejemplo de proyecto en la vista del usuario genérico deben aparecer todos los eventos del usuario registrado, usuario, administrador y cibernauta.](#)

Además de los eventos pueden aparecer textfields, labels, images, etc que formen parte de la presentación de la vista.

[En el ejemplo de proyecto, hay cuatro eventos: modificar perfil \(caso de uso del usuario\), registrarse \(caso de uso del cibernauta\), login \(caso de uso del cibernauta\) y logout \(caso de uso del usuario registrado\).](#)

Bienvenido al ejemplo de MDS2

Modificar Perfil

Registrarse

Login

Logout

En el ejemplo de proyecto se han utilizado botones para representar los <<event>> pero perfectamente se podría haber utilizado un menú Vaadin u otra componente gráfica que permita interactuar con la aplicación.

Esta vista no se usará tal cual sino que se harán versiones diferentes para cada actor con la funcionalidad que le corresponda.

En el ejemplo de proyecto esta vista .ts dará lugar a cuatro “vistas” diferentes: una que solo tiene el botón de logout (para el administrador), una que tiene registrarse y login (para el cibernauta) y otra que tiene logout y modificar perfil (para el usuario).

Las asociaciones entre actores y casos de uso que no necesiten un evento se harán posteriormente.

En el ejemplo de proyecto, lista administrador y lista usuario no tienen un evento en la vista dado que estas dos listas se mostrarán automáticamente al entrar en el perfil del administrador y del usuario, respectivamente.

FAQ: *Todas las vistas han de tener un layout externo que normalmente suele ser VerticalLayout. Esto no significa que podamos utilizar a veces HorizontalLayout directamente, pero es bastante raro que no se necesiten varias filas (una encima de otra) en una vista.*

FAQ: *Es bastante habitual que debamos poner otro VerticalLayout en la vista de los actores en la que aparezca el contenido del perfil del actor. Este VerticalLayout ahora mismo va a estar vacío, pero luego lo usaremos para mostrar otros casos de uso. Este VerticalLayout suele estar debajo del menú o de los botones.*

FAQ: *Como norma general, se debe ocupar todo el espacio de la ventana, tanto horizontalmente como verticalmente:*



Téngase en cuenta lo siguiente:

1. Conforme se van creando las vistas hay que asignarle a cada componente con comportamiento (botón, área de texto, etc) un *id* con un *nombre significativo*.
2. También debe definirse el *id* para los *VerticalLayout* que luego van a contener elementos. El resto de *VerticalLayout* o *HorizontalLayout* no necesitan *id*.
3. Así mismo hay que insertar el componente en el código de la vista .java pulsando un “+” que aparece en el navegador del *Designer*. Pulsando dos veces lo elimina.

- Finalmente, una vez hayamos finalizado la vista hay que generar los *get* y los *set* para cada componente gráfica en el código de la vista .java.

2. Herencia de las vistas de los actores

Ahora vamos a construir el interfaz a partir de las vistas de los actores que hemos creado anteriormente. Para ello, nos vamos a las clases de la carpeta *interfaz* de Eclipse y heredamos en todos los actores (que no heredan) de su vista correspondiente.

En el ejemplo de proyecto, sólo hay una herencia:

```
public class UsuarioGenerico extends VistaUsuariogenerico
```

De esto modo estaremos indirectamente heredando de la vista en cada uno de los actores que heredan de este clase.

3. Constructores de los actores

Ahora debemos crear un constructor los actores que no heredan, que tiene como parámetro un objeto de tipo *MainView*. Este objeto MainView hay que declararlo en la clase.

En el ejemplo de proyecto, sólo para usuario genérico.

```
public class UsuarioGenerico extends VistaUsuariogenerico {  
    MainView MainView;  
    UsuarioGenerico(MainView MainView) {  
        this.MainView = MainView;  
    }  
}
```

Como veremos, esto nos obliga a poner un constructor en el resto de actores. Pondremos el constructor y usaremos *super* para heredar el constructor del actor del que heredan.

```
public class Administrador extends UsuarioRegistrado {  
    Administrador(MainView MainView) {  
        super(MainView);  
    }  
}
```

El constructor tiene un parámetro que es el *MainView*. De este modo, tendremos acceso desde cualquier actor a la ventana principal de nuestra aplicación.

4. Ocultación de componentes

Ahora añadimos código en cada constructor para ocultar los componentes gráficos que no deben aparecer.

```
Administrador(MainView MainView) {  
    super(MainView);  
    this.getModificaperfil().setVisible(false);  
}
```

Entrega 5 (Github): Actores.

ACTIVIDAD 4: VISTAS DE LAS LISTAS

1. Modelado de las vistas de los items de las listas

En esta actividad vamos a modelar las listas y los ítems de las listas. Primero vamos a diseñar las vistas de los elementos que están incluidos en las listas, o sea, los items. Y hacemos lo mismo que con los actores, esto es, hacemos las vistas sólo para los ítems que no heredan de ningún otro ítem y en esa vista ponemos todos los eventos de los ítems que heredan. Además ponemos el resto de componentes gráficas que sean comunes: textfields, images, labels, etc.

En el ejemplo de proyecto tenemos *lista_item* del cual heredan *listalUsuario_item* y *listAdminstrador_item*. En *lista_item* debemos incluir los eventos que se disparan por parte del usuario (modificar) y del administrador (seleccionar) y además el único que se dispara en *lista_item* que es ver detalles.



FAQ: Normalmente en los items se suele usar un *HorizontalLayout* para colocar los elementos que aparecen en el ítem, pero debe ponerse un *VerticalLayout* más externo.

FAQ: El *Horizontal Layout* que contiene los ítems debe ocupar también todo el espacio horizontal y verticalmente.

FAQ: Se pueden utilizar varios *HorizontalLayout* y *VerticalLayout* para dar el formato adecuado al ítem, pero siempre es conveniente poner un *VerticalLayout* más externo. En todos los casos se debe ocupar todo el espacio horizontal y vertical.



2. Herencia, Constructores y ocultación en los items de las listas

Como hicimos con los actores ahora heredamos de esta vista en el .java de los ítems que no heredan. Además ponemos un constructor, que en este caso debe pasar como parámetro la lista en la que aparece el ítem. Finalmente, ocultamos en el ítem de cada lista los elementos que no aparecen.

En el ejemplo de proyecto, heredamos en *lista_item* de la vista correspondiente y pasamos por parámetro la *lista* a la que pertenece. Obsérvese que en el código que disponemos ya existe un atributo en la clase al que asignaremos el valor obtenido en el constructor.

```
public class Lista_item extends VistaLista_item {  
    public Lista _lista;  
    Lista_item(Lista lista) {  
        _lista = lista;  
    }  
}
```

El resto de ítems pasan por parámetro la lista a la que pertenecen. Como en el caso anterior, el resto de ítems hace uso del *super* en el constructor.

```
public class ListaUsuario_item extends Lista_item {  
    ListaUsuario_item(ListaUsuario lista) {  
        super(lista);  
    }  
}
```

FAQ: En el caso de que una lista se implemente con componente gráficos Vaadin distintos del *VerticalLayout*, como por ejemplo, un *ComboBox*, la vista de la lista mostrará el propio *ComboBox* y los ítems de la lista serán los ítems del *ComboBox*. Pero en este caso, no se tendrá vista. Para no llevar a confusión se recomienda anotarlo en el código de la vista.

Finalmente, procedemos a ocultar los elementos que no aparecen en cada ítem.

En el ejemplo de proyecto, en ítem de la lista de usuario no aparece seleccionar, y en el ítem de la lista de administrador no aparece modificar.

```
public class ListaUsuario_item extends Lista_item {  
    ListaUsuario_item(ListaUsuario lista) {  
        super(lista);  
        this.getSeleccionar().setVisible(false);  
    }  
}
```

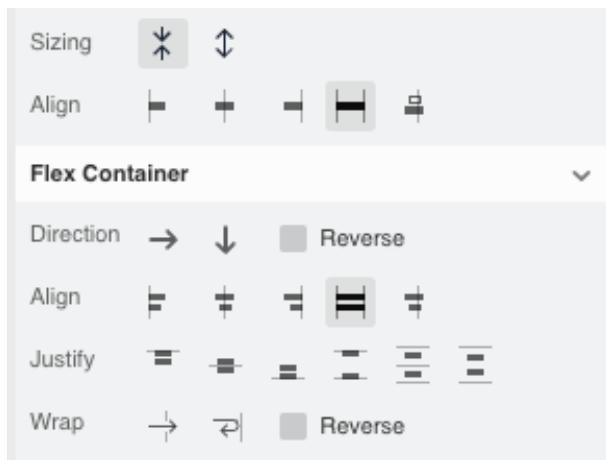
```
ListaAdministrador_item(ListaAdministrador lista) {  
    super(lista);  
    this.getModificar().setVisible(false);  
}
```

3. Modelado de las vistas de las listas

Procedemos de manera similar con las listas que tenemos en nuestra aplicación, o sea aquellas etiquetas con <<list>> en los casos de uso. Creamos vistas para todos los casos de uso <<list>> que no heredan.

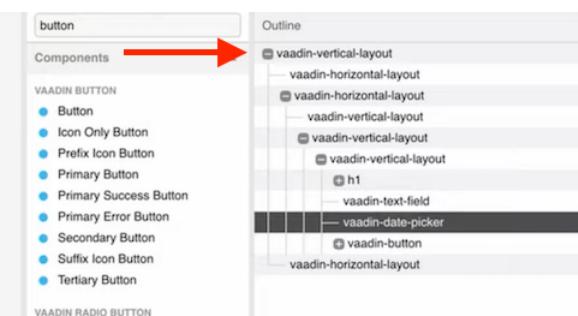
Como norma general, además de los textfields, images, labels que debemos poner en la lista, debemos poner un *VerticalLayout* para guardar los elementos de la lista. Y también aparecerán los eventos asociados a la lista, que son aquellos que están etiquetas como <<option>> en el caso de uso.

FAQ: Es bastante normal que los casos de uso de las listas solo tengan un *VerticalLayout*, que además está vacío. Este *VerticalLayout* suele utilizarse para poner los elementos de la lista, y que en esta fase no tenemos, dado que se crearán dinámicamente según vayamos ejecutando la aplicación.



FAQ: El layout en el que aparecen los elementos de la lista debe estar configurado como aparece en la imagen. Esto es, debe ocupar todo el espacio horizontalmente y además debe alinear los ítems. Pero no debe ocupar todo el espacio verticalmente.

FAQ: En el *VerticalLayout* más externo de la lista hay que poner en el style **position:absolute**. A diferencia de los actores que son las vistas principales de la aplicación, las listas son componentes



que aparecen dentro de otras vistas. Para que las listas ocupen todo el espacio cuando sean colocadas en su sitio deben tener **position:absolute** en su layout más externo.

FAQ: Ahora podemos probar como van a quedar las listas en el editor. Podemos arrastrar el ítem de la lista dentro del layout donde van. Esto lo hacemos para ver como va a quedar, pero una vez probado lo dejamos vacío.

NickTexto	Seleccionar	Modificar	Ver detalles
NickTexto	Seleccionar	Modificar	Ver detalles
NickTexto	Seleccionar	Modificar	Ver detalles

4. Herencia, Constructores y ocultación en las listas

A continuación heredamos de las vistas de las listas que no heredan como antes.

```
public class Lista extends VistaLista {  
}
```

Ahora añadimos los constructores de las listas. Aquí debemos hacer una distinción de casos:

1. La lista está incluida en otro caso de uso o actor. En este caso se pasa por parámetro un objeto del caso de uso o actor que la incluye.
2. La lista no está incluida en otro caso de uso ni ningún actor. En este caso, no hace falta poner parámetro al constructor. Esto significa que no hace falta poner el constructor, dado que se usa el que hay por defecto.

```
public class Lista extends VistaLista {  
    //no hace falta poner el constructor  
}
```

```
public class ListaUsuario extends Lista {  
    public Usuario _usuario;  
    ListaUsuario(Usuario usuario) {  
        _usuario = usuario;  
    }
```

Obsérvese que como antes, ya hay un atributo en la clase para almacenar el parámetro que pasamos.

Finalmente ocultamos aquellos elementos que no deben aparecer en cada lista.

En el ejemplo de proyecto, hay un <>option></> asociado a la lista de administrador para borrar elementos de la lista y por tanto lo debemos ocultar en la lista de usuarios que no tienen esta funcionalidad.

```
ListaUsuario(Usuario usuario) {  
    _usuario = usuario;  
    this.getBorrar().setVisible(false);  
}
```

FAQ: Podría ocurrir que una lista (genérica o no) estuviese incluida en varios casos de uso o asociada a varios actores. En ese caso hay que poner un constructor por cada caso de uso o actor que la contiene.

Entrega 6 (Github): Listas.

ACTIVIDAD 5: VISTAS DE OTROS CASOS DE USO

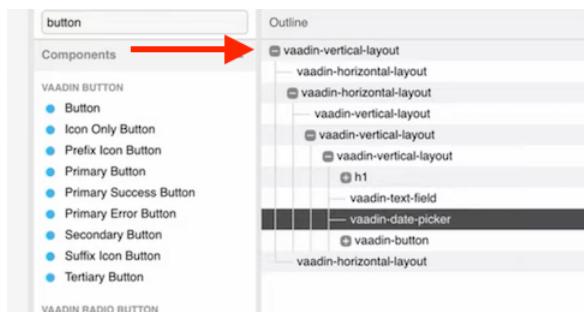
El resto de casos de uso se hacen exactamente igual. Esto es, diseñando las vistas para cada caso de uso que no hereda de otro e incluyendo toda la funcionalidad en el mismo. Posteriormente, se añaden los constructores y se ocultan componentes.

Como en casos anteriores, si un caso de uso está incluido en un actor o incluido en otro caso de uso hay que añadir un constructor con un parámetro. Si no es así, ese constructor no es necesario.

FAQ: *El VerticalLayout más externo debe ocupar todo el espacio horizontalmente y verticalmente.*



Además debe tener en el style `position:absolute`.



En el ejemplo de proyecto, los casos de uso que quedan son login, datos de perfil, añadir, modificar y ver detalles.

FAQ: *Si el caso de uso que estamos diseñando contiene a otro entonces deberíamos poner un VerticalLayout donde va colocado el caso de uso incluido, aunque de momento este vacío.*

```
public class login extends VistaLogin {  
    public Cibernauta _cibernauta;  
    login(Cibernauta cibernauta) {  
        _cibernauta = cibernauta;  
    }  
}
```

```
public class DatosdePerfil extends VistaDatosdeperfil {  
    DatosdePerfil() {}  
}
```

```
public class Aadir extends VistaAadir {  
    public ListaUsuario _listaUsuario;  
    Aadir(ListaUsuario listausuario) {  
        _listaUsuario = listausuario;  
    }  
}
```

```
public class Modificar extends VistaModificar {  
    public ListaUsuario_item _listaUsuario;  
    Modificar(ListaUsuario_item ListaUsuario_item) {  
        _listaUsuario = ListaUsuario_item;  
    }  
}
```

```
public class Verdetalles extends VistaVerdetalles {  
    public Lista_item _lista;  
    Verdetalles(Lista_item lista, texto t) {  
        _lista = lista;  
    }  
}
```

Los casos de uso que están incluidos en las listas pero que no llevan <>option<> están incluidos en los ítems de las listas. Sin embargo los que llevan <>option<> están incluidos en las listas.

No hay que olvidar ocultar los componentes gráficos en cada caso de uso.

```
public class ModificarPerfil extends DatosdePerfil {  
    public Usuario _usuario;  
    ModificarPerfil(Usuario usuario) {  
        this.getDardealta().setVisible(false);  
    }  
}
```

```
public class registrar extends DatosdePerfil {  
    public Cibernauta _cibernauta;  
    registrar(Cibernauta cibernauta) {  
        _cibernauta = cibernauta;  
    }  
}
```

Entrega 7 (Github): Otros casos de uso.

ACTIVIDAD 6: ENSAMBLADO DE COMPONENTES FIJOS

Ahora nos disponemos a ensamblar los componentes que hemos construido y a lanzar eventos en la aplicación que permitan la navegación. De momento lo que tenemos es un montón de componentes y ahora queremos integrarlos para ver cómo van a quedar de verdad.

Podemos distinguir dos tipos de ensamblado:

- a) Aquellos componentes que están fijos, y que no depende de los eventos (botones, opciones de menú, etc) de la aplicación.
- b) Aquellos componentes que no son fijos, y que aparecen cuando ocurren eventos.

Dicho de otro modo, los fijos son aquellos que se *crean y añaden en el constructor* y los no fijos se *crean y añaden en un ClickListener*.

En esta actividad vamos a ensamblar los fijos. El modo de ensamblarlos es siempre el mismo: tomamos el método que tenemos en la clase con tal fin, lo llamamos en el constructor y en el código del método lo ensamblamos.

En el proyecto ejemplo, tenemos que ensamblar de forma fija las dos listas, la de usuario y la de administrador en el actor del usuario y del administrador, respectivamente.

```
Usuario(MainView MainView) {
    super(MainView);
    ListaUsuario();
}

public void ListaUsuario() {
    _listaUsuario = new ListaUsuario(this);
    this.getContenido().as(VerticalLayout.class).add(_listaUsuario);
}
```

```
Administrador(MainView MainView) {
    super(MainView);
    ListaAdministrador();
}

public void ListaAdministrador() {
    _listaAdministrador = new ListaAdministrador(this);
    this.getContenido().as(VerticalLayout.class).add(_listaAdministrador);
}
```

Para ensamblarlo en el método, debemos crear el objeto que queremos ensamblar con el new, y luego colocarlo donde va.

FAQ: *Podría ocurrir que no tuviésemos donde ponerlo. Es probable que se nos haya olvidado poner un vertical layout en el contenedor para ponerlo.*

FAQ: *Debemos hacer add en el vertical u horizontal layout donde va colocado. En el caso del vertical layout no es posible hacer add directamente, hay que llamar a as(VerticalLayout.class).*

Entrega 8 (Github): Ensamblado de Componentes Fijos

ACTIVIDAD 7: ENSAMBLADO DE COMPONENTES DINÁMICOS

Ahora vamos a ensamblar los componentes dinámicos. Al contrario que los fijos no se añaden en el constructor, sino que normalmente hay un evento que los ensambla.

En este caso pueden ocurrir dos situaciones.

1. Que dispongamos de un método en la clase para ensamblarlo. En ese caso, debemos añadir un *clickListener* al evento que lo ensambla y llamar en el *clickListener* a ese método. En el método hacemos el ensamblado como antes.
2. Que no dispongamos de un método en la que clase para ensamblarlo. En ese caso el ensamblado lo hacemos directamente en el *clickListener*.

FAQ: *El hecho de que tengamos o no el método depende de que lo hayamos hecho al modelar en diagrama de casos de uso. Normalmente en el diagrama de casos de uso no se modela todos los eventos para no complicar en exceso el modelo pero es posible que si hayamos añadido a la vista una componente gráfica un evento para lanzar el caso de uso. Dado que el plugin no ha generado un método en ese caso, debemos hacer el ensamblado directamente en el clickListener.*

1. Ensamblado de las vistas no lista

Lo primero que vamos a hacer es ensamblar aquellos caso de uso que no son listas y que son dinámicos. Todos los botones que nos llevan a un caso de uso no lista han de disponer de su evento que muestra la vista del caso de uso.

En el ejemplo de proyecto, la ventana de login se ensambla cuando se pulsa el botón de login.

```
public class Cibernauta extends UsuarioGenerico {  
    public login _login;  
    public Cibernauta(MainView MainView) {  
        super(MainView);  
        this.getLogin().addClickListener(event -> login());  
    }  
  
    public void login() {  
        _login = new login(this);  
        MainView.removeAll();  
        MainView.add(_login);  
    }  
}
```

Obsérvese que utilizamos el atributo de clase que tenemos para crear el objeto y ensamblarlo.

En el ejemplo de proyecto, tenemos un botón cancelar al registrarnos que no aparece en el diagrama de casos de uso.

```
public class registrar extends DatosdePerfil {  
    public Cibernauta _cibernauta;  
  
    registrar(Cibernauta cibernauta) {  
        _cibernauta = cibernauta;  
        this.getCancelar().addClickListener(event -> {  
            this._cibernauta.MainView.removeAll();  
            this._cibernauta.MainView.add(this._cibernauta);  
        });  
  
    }  
}
```

FAQ: Cuando se abre una nueva vista, creamos el objeto de la nueva vista y lo añadimos donde corresponda. Para añadir esta vista, normalmente hacemos `removeAll()` en donde va colocada. De este modo quitamos lo que haya y ponemos la nueva vista.

2. Ensamblado de las listas

Aunque no tengamos los ítems de las listas esto no significa que no se puedan ensamblar. Su ensamblado de momento sirve para que se cambie de una vista a otra, aunque de momento no haya elementos. Podemos distinguir dos casos:

1. Eventos que modifican la lista, esto insertan, modifican o borran en la lista
2. Eventos que no modifican la lista

En el primer caso, la lista hay que recargarla. Esto es, una vez modificada debería mostrar los elementos de la lista con la modificación. Para ello, debemos cargar el actor, o dicho de otro modo, debemos crear un nuevo objeto actor y mostrar la información actualizada. Si es necesario se debe reconstruir la vista actual.

En el ejemplo de proyecto, al añadir un elemento a la lista de usuarios, debemos cargar al usuario. Al cargar el usuario mostrará la lista actualizada. En este caso basta con colocar al usuario en el `MainView`.

```
public class Aadir extends VistaAadir {  
    public ListaUsuario _listaUsuario;  
    Aadir(ListaUsuario listausuario) {  
        _listaUsuario = listausuario;  
        this.getBotonAadir().addClickListener(event -> {  
            Usuario nu = new Usuario(_listaUsuario._usuario.MainView);  
            _listaUsuario._usuario.MainView.removeAll();  
        });  
    }  
}
```

```
_listaUsuario._usuario.MainView.add(nu));});
```

Lo mismo ocurre con modificar que vuelve a cargar la lista del usuario.

```
public class Modificar extends VistaModificar {
    public ListaUsuario_item _listaUsuario;
    Modificar(ListaUsuario_item ListaUsuario_item) {
        _listaUsuario = ListaUsuario_item;
        this.getBotonModificar().addClickListener(event -> {
            ListaUsuario lu = (ListaUsuario) _listaUsuario._lista;
            Usuario nu = new Usuario(lu._usuario.MainView);
            lu._usuario.MainView.removeAll();
            lu._usuario.MainView.add(nu);}}}
```

En el segundo caso, la lista no hay que recargarla, y podemos hacerlo como en el caso de casos de uso no lista.

En el ejemplo de proyecto, el botón cancelar no necesita recargar la lista por que no se ha modificado.

```
this.getBotonCancelar().addClickListener(event -> {
    ListaUsuario lu = (ListaUsuario) _listaUsuario._lista;
    lu._usuario.getContenido().as(VerticalLayout.class).removeAll();
    lu._usuario.getContenido().as(VerticalLayout.class).add(lu);}})
```

3. Ensamblado de los ítem de las listas

Debemos ensamblar también los ítem de las listas. Para ensamblarlos podemos añadir de forma manual en el código elementos en la lista y darle funcionalidad a los eventos.

En el ejemplo de proyecto le damos funcionalidad al botón de modificar un elemento de la lista:

```
public void Modificar() {
    ListaUsuario lu = (ListaUsuario) this._lista;
    lu._usuario.getContenido().as(VerticalLayout.class)
        .removeAll();
    Modificar m = new Modificar(this);
    lu._usuario.getContenido()
        .as(VerticalLayout.class).add(m);
}
```

O al botón de seleccionar, en el que ponemos en color rojo los elementos seleccionados.

```

public class ListaAdministrador_item extends Lista_item {
    Boolean seleccionado = false;
    ListaAdministrador_item(ListaAdministrador lista) {
        super(lista, texto);
        this.getSeleccionar().addClickListener
            (event -> Seleccionar());
    }
    public void Seleccionar() {
        if (!seleccionado) {
            seleccionado = true;
            this.getStyle().set("color", "red");
            this.getSeleccionar().setText("Quitar");
            la.getBorrar().setEnabled(true);
            la._item.add(this);
        } else {
            seleccionado = false;
            this.getStyle().set("color", "black");
            this.getSeleccionar().setText("Seleccionar");
            la.getBorrar().setEnabled(false);
            la._item.remove(this);}}}
}

```

FAQ: El código generado por el Instant Generator nos proporciona un vector para cada lista, este vector nos puede ser de utilidad para guardar los elementos de la lista. En el caso anterior, el vector nos permite almacenar los elementos seleccionados para luego poder borrar.

```

public class Lista extends VistaLista {
    public Vector<Lista_item> _item = new Vector<Lista_item>();
    Lista() {
    }
}

```

FAQ: El interfaz debería permitir hacer sólo aquellas cosas que tienen sentido. Por ejemplo, en el código anterior el botón de Borrar es habilitado o deshabilitado dependiendo de si se ha seleccionado al menos un elemento. Además, cuando se selecciona un elemento, se cambia el texto del botón para poder quitarlo. Dicho de otro modo, el botón seleccionar sirve tanto para seleccionar como para quitar. Para que se marque los elementos seleccionados se ha cambiado el color del texto a rojo.

3. Funcionalidad Común

A veces varios casos de uso o actores heredan de un mismo caso de uso o actor, y ese caso de uso o actor tiene una funcionalidad incluida que es común a todos los casos de uso o actores que heredan. Eso ocurre, por ejemplo, en los actores que incluyen habitualmente el caso de uso login. Ese login se comporta distinto en cada actor particular. Es distinto, porque el login abre la vista (el perfil) del actor que hace login, que suele ser distinto para cada actor.

Para tratar este caso, se puede optar por la siguiente solución. En el código del caso de uso incluido se hace distinción de casos sobre el actor o caso de uso que está lanzando la funcionalidad.

En el ejemplo de proyecto, para dar funcionalidad a login hacemos lo siguiente.

```
this.getBotonlogin().addClickListener(event -> {
    this._cibernauta.MainView.removeAll();
    if (this.getLogin().getValue() == "usuario") {
        Usuario u = new Usuario(this._cibernauta.MainView);
        this._cibernauta.MainView.add(u);
    }
    else if (this.getLogin().getValue() == "admin") {
        Administrador a = new Administrador(this._cibernauta.MainView);
        this._cibernauta.MainView.add(a);
    }
    else {
        Notification.show("Este usuario no existe");
    }
});
```

Como vemos hacemos distinción de casos (de momento, simplemente con el login) para ir a la vista de usuario o ir a la vista de administrador.

En el ejemplo de proyecto, hay una funcionalidad incluida en los ítems de las listas de usuario y administrador, que es ver los detalles de la misma. De nuevo, hacemos distinción de casos.

```
public void Verdetalles() {
if (_lista instanceof ListaUsuario) {
    ListaUsuario lu = (ListaUsuario) _lista;
    _verdetalles = new Verdetalles(this);
    lu._usuario.getContenido().as(VerticalLayout.class).removeAll();
    lu._usuario.getContenido().as(VerticalLayout.class).add(_verdetalles);
}
else {
    ListaAdministrador lu = (ListaAdministrador) _lista;
    _verdetalles = new Verdetalles(this);
    lu._administrador.getContenido().as(VerticalLayout.class).removeAll();
    lu._administrador.getContenido().as(VerticalLayout.class).add(_verdetalles);
}
}
```

```
this.getVolver().addClickListener(event -> {
if (_lista instanceof ListaUsuario_item) {
    ListaUsuario lu = (ListaUsuario) _lista._lista;
    lu._usuario.getContenido().as(VerticalLayout.class).removeAll();
    lu._usuario.getContenido().as(VerticalLayout.class).add(lu);
} else {
    ListaAdministrador lu = (ListaAdministrador) _lista._lista;
    lu._administrador.getContenido().as(VerticalLayout.class).removeAll();
    lu._administrador.getContenido().as(VerticalLayout.class).add(lu);
}
});
```

4. Ensamblado del MainView

Nos queda, por último, escribir código en el *MainView* que también es dinámico. La estructura de proyecto *Vaadin* que importamos tiene en el *MainView* un código de ejemplo. Este código de ejemplo lo podemos borrar. En su lugar colocamos simplemente, la primera vista que aparece en la aplicación, que normalmente suele ser el usuario no registrado.

En el ejemplo de proyecto, sería como sigue.

```
public class MainView extends VerticalLayout {  
  
    public MainView(@Autowired GreetService service) {  
        Cibernauta cb = new Cibernauta(this);  
        add(cb);  
    }  
}
```

Al final de esta fase, deberíamos tener código en todos los métodos que ha generado el plugin excepto en aquellos que proceden de <> que los haremos más adelante. Además cada evento que aparezca en una ventana debe tener un ClickListener. Y finalmente todas las vistas deberían estar ensambladas.

Entrega 9 (Github): Ensamblado de Componentes Dinámico

ACTIVIDAD 8: GENERACIÓN DE CÓDIGO ORM

En esta actividad vamos a crear nuestra base de datos, el esquema de la misma y vamos a generar de forma automática una librería Java para hacer operaciones con la base de datos.

1. Preparación

- i) Primero nos vamos a ir a *Visual Paradigm* y abrimos el diagrama de clases de la base de datos. Abrimos el *Model Explorer* y comprobamos que hay una carpeta que contiene las clases *ORMPersistable*. Si no la hay, la creamos y movemos con el ratón las clases *ORMPersistable* a esa carpeta. *Visual Paradigm* generará las clases *ORMPersistable* en una carpeta con este nombre y luego, en *Eclipse*, el nombre de esta carpeta será el nombre del paquete en el que están las clases.
- ii) A continuación comprobamos que los nombres (nombre de la clase y nombre de atributos) en las *ORMPersistable* no contienen espacios en blanco, tildes ni eñes. Dado que esos nombres se va a usar como nombres de las clases y de los atributos es importante modificarlos. Hay que hacer lo mismo con los roles de las asociaciones que hay entre las *ORMPersistable*: quitar los espacios en blanco, las eñes y las tildes.
- iii) También hay que comprobar que no falta ninguna de las *cardinalidades* (1 , $0..1$, $*$, $0..*$ etc) entre las *ORMPersistable*. Hay que repasar las *cardinalidades* 1 y $1..*$. Cuando se genere la base de datos estas *cardinalidades* van a incluir un *NOT NULL* o una *clave externa*.
- iii) En MDS1 se han elegido claves para cada *ORMPersistable* y claves para cada asociación. Desafortunadamente, ORM no permite claves que no sean valores enteros. Por tanto, tenemos que elegir claves para que sean enteras (*Int*). Si no hay ningún valor que sea entero que pueda servir de clave, luego generaremos un nuevo atributo de tipo *Int* y lo usamos como clave de las asociaciones si hiciera falta.

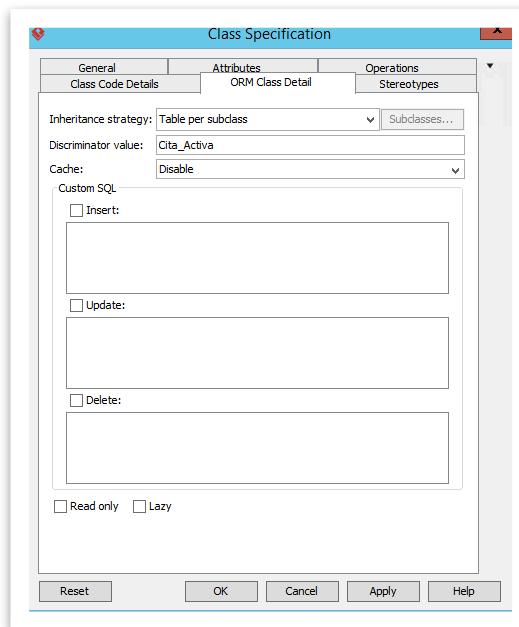
FAQ: *A veces ocurre que accidentalmente hemos borrado alguna relación (asociación o herencia) y aunque ésta no aparece en el diagrama, sí está almacenada en el modelo. En este caso debemos comprobar que no hay relaciones extra en el Open Specification de cada clase en Visual Paradigm.*

- iv) A continuación, debemos preparar las *ORMPersistable* que heredan de otras. El ORM va a usar la misma clave para todas las *ORMPersistable* que heredan de una. La clave estará como atributo de tipo entero en la *ORMPersistable* que está más arriba, y las *ORMPersistable* que heredan la tendrán como clave externa. Por tanto, debemos dejar una única clave de tipo *Int* en la *ORMPersistable* que esté mas arriba y eliminar si las hubiera las claves de las tablas que heredan, actualizando las claves de las asociaciones si las hubiera.

- v) Finalmente hay que habilitar en cada *ORMPersistable* que hereda la opción *Table per SubClass*. Esto se hace colocándose sobre ella en el diagrama de clases, pulsando *Open Specification* y buscando en el menú superior *ORM Class Detail* (esta opción normalmente suele estar oculta en la flechita de la esquina superior derecha). Ver imagen.

2. Arranque de MySQL

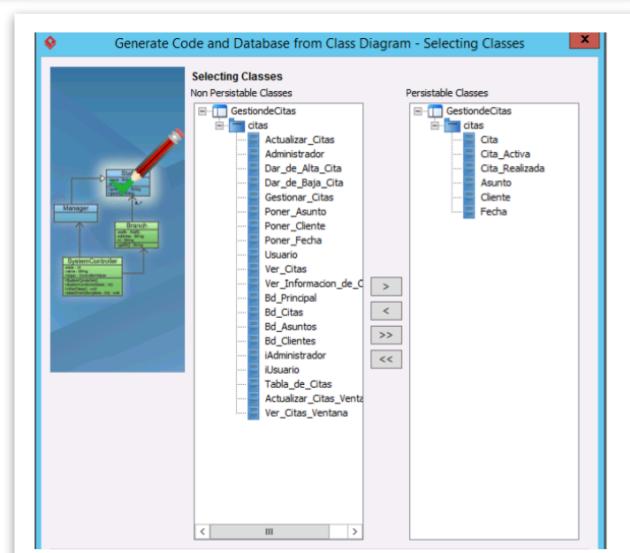
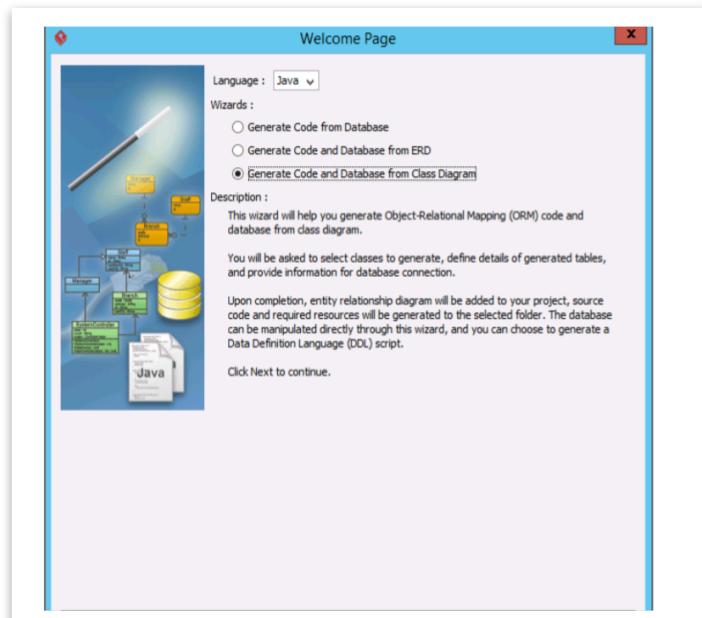
Ahora lo que vamos a hacer es arrancar el *MySQL* y *Apache* de XAMPP e irnos al *PhpMyAdmin* para crear una base de datos nueva que es donde vamos a guardar los datos de nuestro proyecto.



Generación de código ORM

Ahora ya podemos generar el código ORM para la base de datos. Para ello:

- Nos vamos en Visual Paradigm a *Tools -> Hibernate -> Wizard*. Seleccionamos a continuación *Generate Code and Database from Class Diagram* y después nos aparecerá una ventana en la que elegimos del modelo los *ORMPersistable* (normalmente también aparecen los *ORMComponent* pero hay que eliminarlos de la lista).
- En la siguiente pantalla, podremos elegir la clave de cada una de las *ORMPersistable*. Aquí elegimos las claves enteras que pusimos en el paso anterior. Si alguna *ORMPersistable* no tuviese, la generamos automáticamente. En la siguiente ventana podemos comprobar los elementos que va a generar para cada tabla.
- A continuación podemos comprobar la configuración de la base de datos, descargando el driver de *MySQL*, y poniendo el nombre de la base de datos

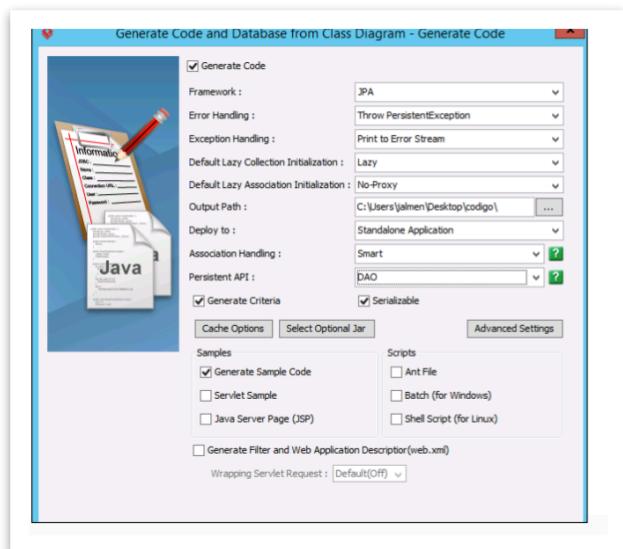


que hemos creado antes. La base de datos debería estar en localhost 3306. Además clicamos en *InnoDB*. Debemos probar la conexión.

- iv) En la siguiente pantalla lo que tenemos que hacer es seleccionar el *Framework JPA*, seleccionar una carpeta de destino para que guarde el código (*Output Path*), y seleccionar *Smart*, *DAO*, *Generate Criteria* y *Generate Sample Code*. Pulsamos para generar el código.

Además de la carpeta con el código, Visual Paradigm habrá generado un *diagrama entidad-relación*. Es importante comprobar en este diagrama que las tablas aparecen conectadas entre ellas. Aunque la generación de código sea exitosa, el hecho de que algunas tablas no aparezcan conectadas con otras indica que la generación de código no es correcta y hay que revisar el diagrama de clases.

- v) El siguiente paso, consiste en importar a Eclipse todo el código generado. Debemos importar la carpeta con el código así como la carpeta *ormsamples* y *ormmapping*. Estas carpetas deben estar a la misma altura en el src del proyecto (ninguna dentro de otra, ni dentro de otra carpeta). Es muy probable que encontremos errores en Eclipse. Esto puede ser debido a que aún no hemos añadido el fichero *orm.jar*. Este fichero se encuentra en la carpeta *lib* donde hemos generado el código ORM. Debemos añadir este fichero *orm.jar* al classpath del build path de Eclipse.



FAQ: Si se ha encontrado algún error en el proceso de generación de código hay que volver a generarlo otra vez. Se recomienda borrar el diagrama entidad-relación cuando se vuelve a generar el código. Los ids que se hayan generado automáticamente no se borran, por tanto, hay que elegirlos en lugar de volver a generarlos.

3. Creación del Esquema y configuración de Vaadin

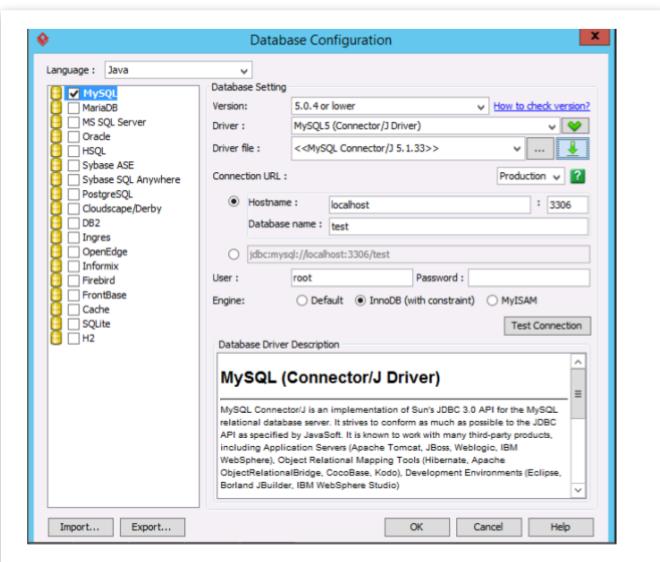
Aunque aún no hemos acabado de diseñar nuestra aplicación (faltan los diagramas de secuencia que se harán en la siguiente actividad), podemos ya probar el código ORM.

- i) Nos vamos a la carpeta *ormsamples*. Ahí veréis que hay una serie de programas ejemplo que nos ha generado el *Visual Paradigm*. Hay uno concretamente que se puede ejecutar directamente como aplicación Java. Se suele llamar así: *CreateDatabaseSchema*. Lo ejecutamos para que el esquema de la base de datos. Debemos tener arrancando el MySQL de XAMPP. Esto nos habrá creado las tablas que las podemos ver yéndonos al *PhpMyAdmin* de MySQL.

FAQ: Veréis que en la carpeta *ormsamples* hay también un programa para borrar la base de datos y programas de ejemplo para crear o consultar la base de datos. Estos programas nos pueden servir

para probar que se añaden datos a la base de datos y que se consultan. No es necesario en este momento ejecutarlos.

- ii) Hay que configurar *Vaadin* para acepte ORM. Desafortunadamente, ORM crea conflictos con la recarga automática de *Vaadin* cuando se hacen modificaciones del código *Vaadin*. Es por ello que hay que modificar el pom.xml del proyecto poner entre comentarios la recarga automática. Buscad "devtools" en el pom.xml y ponedlo entre <!-- y -->.



```
<!--<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>-->
```

Entrega 10 (Github): Código ORM

ACTIVIDAD 9: ACTUALIZACIÓN DE LOS CONSTRUCTORES

Una vez hemos generado el código ORM ahora podemos actualizar los constructores de modo que podamos construir vistas con datos. Para ello pasaremos los objetos *ORMPersistable* por parámetro al constructor, aunque no en todos los casos. Como normal general:

1. A las vistas de **actores** se les pasa el objeto *ORMPersistable* que almacena sus datos.
2. A las vistas de los **ítems de las listas** se les pasa el objeto *ORMPersistable* que almacena el ítem.
3. **Cualquier vista que muestre datos de la base de datos.**

1. Actores

Debemos declarar en la clase un atributo del objeto *ORMPersistable*, pasarlo por parámetro y asignarlo, igual que hicimos con el *MainView*.

```
public basededatos.usuario u;
Usuario(MainView MainView, basededatos.usuario u) {
    super(MainView);
    this.u = u;
}
```

Puede ocurrir que algún actor no necesite el objeto *ORMPersistable*. Esto ocurre a veces con el administrador de la aplicación, que en la base de datos no tiene relaciones con otros datos.

En el ejemplo de proyecto, hemos pasado a la vista de usuario, el objeto usuario de la base de datos. No es necesario pasar el objeto administrador a la vista del administrador porque los datos del administrador no se usan en la vista del administrador. Sin embargo, necesitamos el objeto del usuario en la vista de usuario para poder saber cuáles son sus aportaciones a la lista.

FAQ: Cuando empecemos a introducir estos parámetros, veremos que aparecen errores de compilación, dado que hemos utilizado estas vistas en otras partes del código y hemos modificado su constructor. De momento, podemos pasar *NULL* en las llamadas al constructor. Luego lo reemplazaremos por el valor correcto.

2. Ítems

En el segundo caso, se ha de pasar el objeto *ORMPersistable* que almacena los datos del ítem, y además el código del constructor debe coger los datos del objeto y llenar los componentes gráficos.

En el proyecto ejemplo, pasamos el *ORMPersistable* texto, y rellenamos la vista del ítem con los datos del ítem: Nick y Texto. Como vemos a partir de este objeto podemos coger también el nick de quien ha escrito el texto: *texto.getEscrito_por().getNick()*.

```
public texto t;
Lista_item(Lista lista, texto texto) {
    _lista = lista;
    t = texto;
    this.getNick().setText(texto.getEscrito_por().getNick());
    this.getTexto().setText(texto.getContenido());
}
```

Si hay herencia en las listas, pasaremos el objeto *ORMPersistable* en todas, pero debemos llenar los datos en la clase más arriba, de este modo todos los ítems de las subclases lista llenarán el objeto cuando se creen.

```
public class ListaUsuario_item extends Lista_item {
    ListaUsuario_item(ListaUsuario lista, texto texto) {
        super(lista, texto);
    }
}
```

3. Vistas que muestran datos

En el tercer caso, y una vez pasado el objeto a los actores, podemos llenar el resto de vistas que muestran datos. Al pasar el objeto al actor, y gracias al paso del parámetro vista al constructor, ahora tenemos acceso a ese objeto en el resto de vistas.

En el proyecto ejemplo, podemos mostrar los datos de registro en la vista Modificar Perfil para que puedan ser modificados. Accedemos a los datos a través del objeto *ORMPersistable* "u" de la vista de Usuario.

```
public class ModificarPerfil extends DatosdePerfil {

    public Usuario _usuario;

    ModificarPerfil(Usuario usuario) {
        this._usuario = usuario;
        this.getNombre().setValue(_usuario.u.getNombre());
        this.getDni().setValue(_usuario.u.getDNI());
        this.getCorreo().setValue(_usuario.u.getCorreo());
        this.getLogin().setValue(_usuario.u.getLogin());
        this.getPassword().setValue(_usuario.u.getPassword());
        this.getNick().setValue(_usuario.u.getNick());
    }
}
```

Si hubiese herencia entre vistas que muestran datos podemos hacer el paso del objeto ORMPersistable en la clase de más arriba.

Al final de esta fase, deberíamos tener todos los constructores actualizados con el nuevo objeto y el código que captura los datos del objeto y asigna los datos a los componentes gráficos.

Entrega 11 (Github): Constructores

ACTIVIDAD 10: DIAGRAMAS DE SECUENCIA

En esta actividad vamos a crear diagramas de secuencia que nos permitirán describir la interacción con la base de datos a través de métodos. Esto nos permitirá por otro lado poblar el diagrama de clases de la base de datos con métodos.

Tenemos, en términos generales, que crear un diagrama de secuencia para:

- A. Crear un elemento de cada *ORMPersistable*
- B. Cargar todos los elementos de cada *ORMPersistable*
- C. Buscar un elemento (o elementos) de cada *ORMPersistable*
- D. Borrar o modificar un elemento de cada *ORMPersistable*
- E. Añadir o eliminar una relación entre cualesquiera dos elementos *ORMPersistable*

El que sea necesario o no crear el diagrama de secuencia dependerá de nuestra aplicación y el número de cada tipo también dependerá de la misma. Dependerá principalmente de las interacciones que hagamos con la base de datos para crear elementos, carga datos, buscar datos, borrar elementos o actualizar relaciones.

Por ejemplo, supongamos que nuestra aplicación almacena usuarios de una red social que pueden hacer publicaciones en la red social y tener relaciones de seguimiento. En este caso, tendríamos que crear usuario (en la ventana de registro), crear publicación (cuando publica un post el usuario), buscar usuario (si queremos tener un buscador de usuarios), buscar publicación por fecha o por hashtag, borrar un usuario (cuando se da de baja) o añadir o borrar una relación de seguimiento. Por último, si se quiere mostrar la lista de usuarios en alguna ventana, habría que cargar los elementos de la tabla de usuarios.

Tal y como esta diseñado ORM si se carga un elemento se dispone de un atributo (objeto simple o Set) dentro del objeto que lo representa que almacena los elementos que están relacionados con él.

Por ejemplo, en el ejemplo la red social si se carga a un usuario tendríamos un atributo de tipo Set que almacena los usuarios que le siguen y otro atributo de tipo Set las publicaciones que ha hecho. Por tanto, no es necesario cargar los seguidores de un usuario o cargar las publicaciones de un usuario.

Dicho de otro modo, hay que hacer un diagrama de secuencia para cada uno de los componentes gráficos de la interfaz de usuario que interactúen con la base de datos, esto es, eventos o contenedores que necesiten o envíen datos, de modo que se pueda cargar, crear, modificar o borrar datos de la base de datos, lo que se suele llamar *operaciones CRUD (Create-Retrieve-Update-Delete)*.

Sólo hará falta hacer un diagrama de secuencia de carga de datos *a menos que no ya estén cargados de manera indirecta* porque estén almacenados en uno de los atributos de un objeto previamente cargado.

Normas generales

El diagrama de secuencia siempre incluye:

- a. CASO A) El **actor** cuando se dispara un evento y se dispone de método (generado por el plugin de MDS1) en la vista. El actor que se elige es el que está visualizando esa vista. Además el diagrama de secuencia incluye la propia **vista** (la que contiene el *clickListener* del evento).
- b. CASO B) Cuando no hay método, y es un evento o una carga de datos, la **vista** desde la que se lanza la interacción (puede haber *clickListener* o no).
- c. El **interfaz de la base de datos** del actor que está visualizando esa vista.
- d. **Una o varias ORMPersistable**. Al menos una. Son aquellas que participan en la ejecución.
- e. **Los actores (<<external>>) involucrados**, si los hubiera.

Herencia

La herencia también es considerada a la hora de describir las interacciones con los diagramas de secuencia. En lugar de repetir el mismo diagrama de secuencia para una vista que hereda de otra, es suficiente con hacer un diagrama de secuencia para la vista de la que se hereda. Dicho de otro modo, hacemos como en la futura implementación, que solo implementamos código en la clase de la que se hereda y luego asumimos que heredamos el comportamiento en la clase que hereda.

Vistas compartidas

Podría ocurrir que una misma vista sea utilizada por diferentes actores. En ese caso, sí hay que repetir la vista para cada uno de los actores.

Actores External

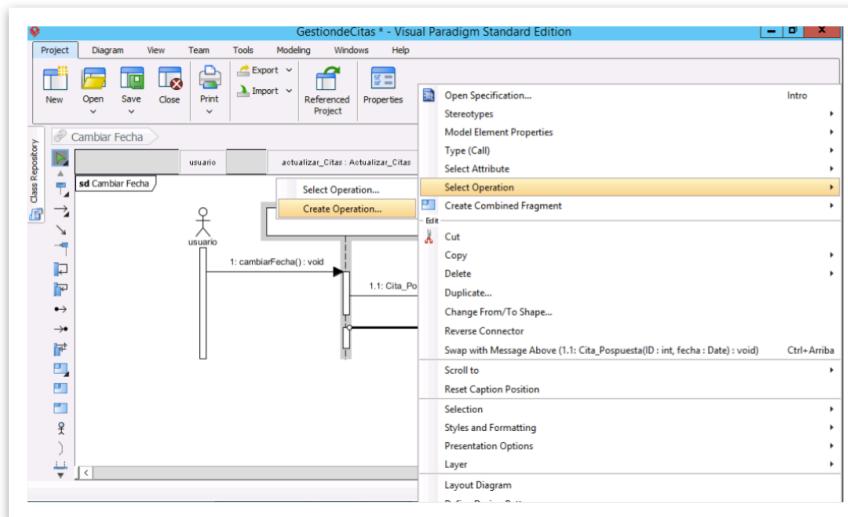
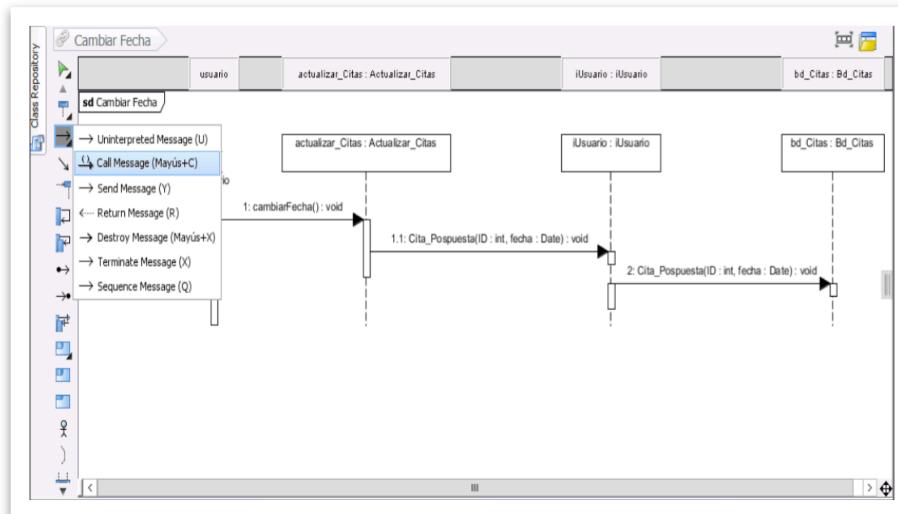
Si la vista interactúa con un actor *<<external>>*, es el momento de especificarlo mediante el diagrama de secuencia.

Creación de los diagramas de secuencia

- i) Hay que **arrastrar** los actores y la vistas desde el *Model Explorer* para **añadirlas al diagrama**. Las flechas de interacción son siempre *Call Message*, y cada método hay que añadirlo con *Create Operation*, excepto cuando ya existe, que hay que seleccionarlo. Ha de comprobarse que conforme se van añadiendo los métodos nuevos van apareciendo en el diagrama de clases (de la interfaz de usuario y de la base de datos).

ii) En el CASO A) la primera flecha / operación del diagrama de secuencia va del actor a la vista y es siempre el método que es invocado en la vista por el *clickListener*. Por tanto, el método será “login()”, “añadir_usuario()”, “borrar_usuario()”, “buscar_usuario()” etc, que fue generado por el plugin de MDS1. La fecha (*Call message*) va desde la línea de tiempo del actor a la línea de tiempo de la vista donde está el *clickListener*.

iii) La segunda flecha del CASO A) y la primera del CASO B) van de la vista al interfaz de la base de datos. Estas operaciones son nuevas, y por tanto hay que crearlas con el *Create Operation*. Deberá dársele un nombre y añadir parámetros. Los parámetros se



utilizan para pasarle al interfaz los datos que añadir/recuperar/modificar/borrar de la base de datos. Por ejemplo, *login(String nick, String password)*, *añadir_usuario(String nombre, String edad, String nick, String password)*. Pero pueden no tener parámetros. Normalmente esto ocurre cuando se quieren cargar todos los datos de un *ORMPersistable*: *cargar_usuarios()*.

iv) Todas deben tener tipo de retorno: void, Boolean, Int, etc. e incluso objetos *ORMPersistable* de la base de datos, o vectores de objetos *ORMPersistable*. Normalmente, void se reserva para las operaciones de actualización, mientras que las operaciones de

creación suelen devolver el objeto creado, o las carga suelen devolver un vector con los elementos cargados.

v) Muchas operaciones necesitan acceder a datos que ya existen en la base de datos. En ese caso deben pasar por parámetro el **id** que asigna el ORM. Los casos más habituales son *borrar(Int Id)* o *seguir(Int IdSeguidor,Int IdSeguido)*. Como vemos, pasamos los ids de los elementos que queremos borrar o que queremos relacionar.

vi) El resto de flechas/operaciones (también *Call message*) conectan la línea de tiempo de la vista con la interfaz de la base de datos, o la línea de tiempo de la interfaz de la base de datos principal con la línea de tiempo de las *ORMPersistable*. En este caso también hay que pasar parámetros, y los ids correspondientes. Puede ocurrir que no tengan parámetros porque se dispare como respuesta a un método sin parámetros a su vez.

vii) Como se comentó al principio se puede interactuar con varias *ORMPersistable* en un mismo diagrama de secuencia. Normalmente ocurre con operaciones complejas que crean/recuperan/actualizan/borran elementos en varias tablas.

viii) La interacción con los actores <>external></> consiste en flechas que van desde la vista hasta el actor <>external></>, que hay que arrastrar también como en el caso del resto de actores. En este caso, el método ya ha sido creado por el plugin de MDS1 y es suficiente con seleccionarlo.

Carga de Actores

En el caso de diagramas de secuencia que modifican datos de un actor hay que añadir un método para cargar el actor. Este método puede llamarse *obtenerActorById(int Id):actor* y al cual se le pasa el id del actor, devolviendo el object actor correspondiente. Podría ocurrir que ya cargásemos al actor en el método principal, en cuyo caso no es necesario.

En el proyecto ejemplo, esto ocurre cuando se añade o modifica un elemento en la lista de usuarios. En el caso de modificar perfil no es necesario porque se carga directamente al actualizar.

Implementación de las interfaces

Una vez hechos todos los diagramas de secuencia, vamos a poblar la BDPrincipal que está actualmente vacía con métodos. Estos métodos provendrán de que la BDPrincipal implementa todos los interfaces. Esto lo hace Visual Paradigm de forma automática, colocándonos sobre la BDPrincipal y pulsando con el botón derecho: *Related Elements->Realice All Interfaces*.

Exportar los métodos de los diagramas de secuencia

Finalmente vamos a exportar el código de las bases de datos. Volvemos a ir a *Tools->Code->Instant Generator*, como hemos hecho para la interface de usuario, pero ahora en *Model*

Elements seleccionamos las clases de la BDPrincipal, los interfaces de la misma y los *ORMComponent*.

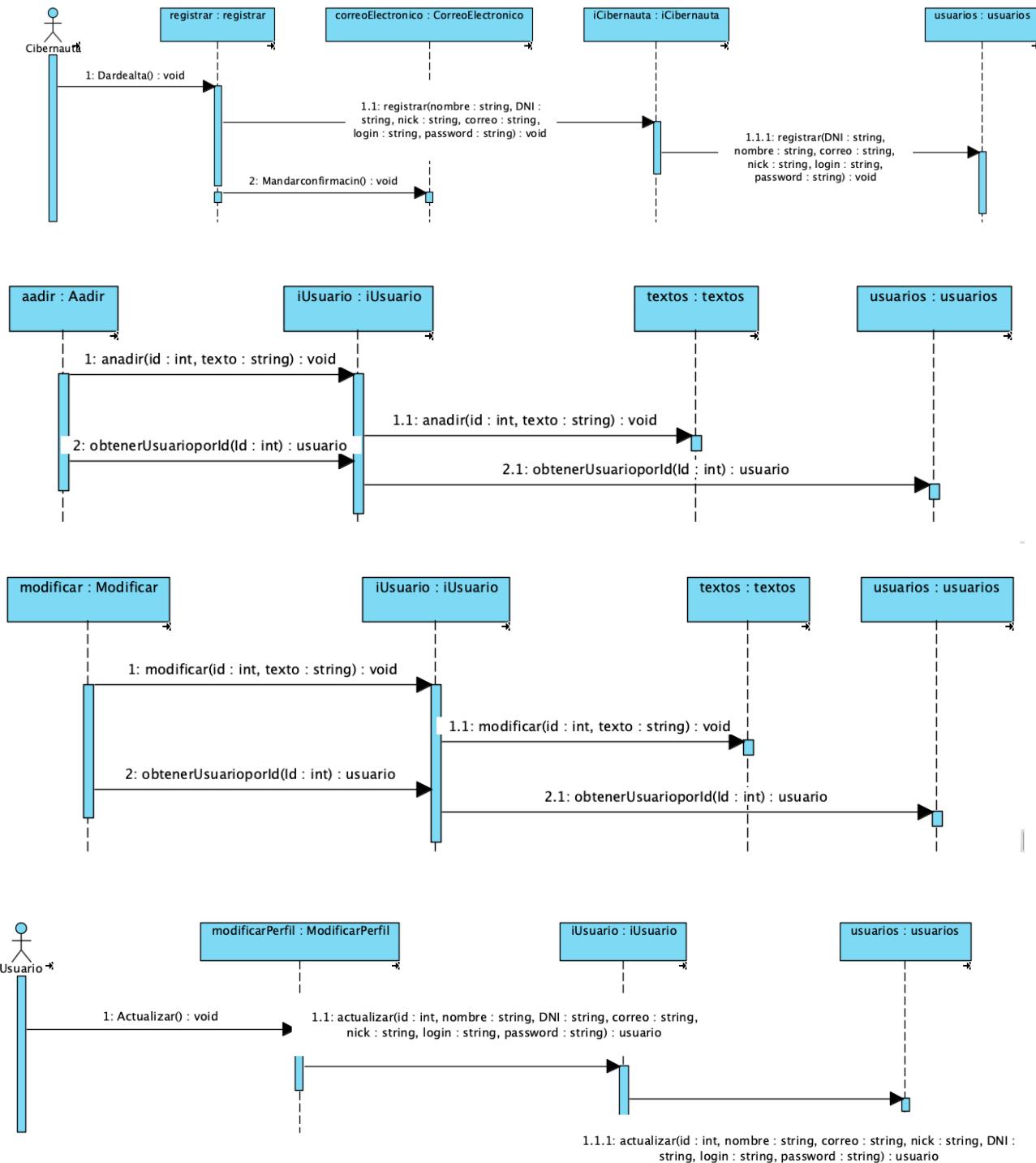
Entrega 12 (Aula Virtual): Documentación Parcial

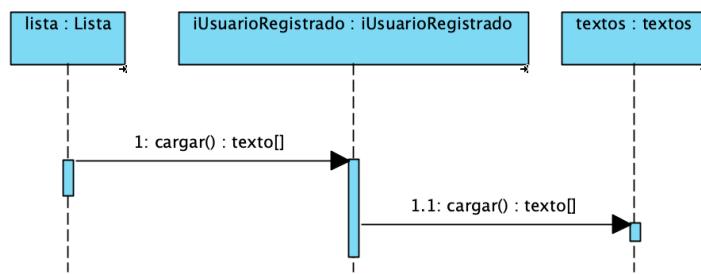
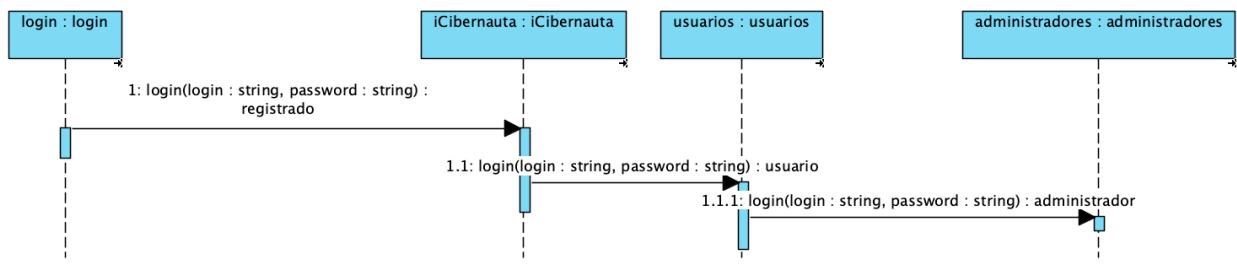
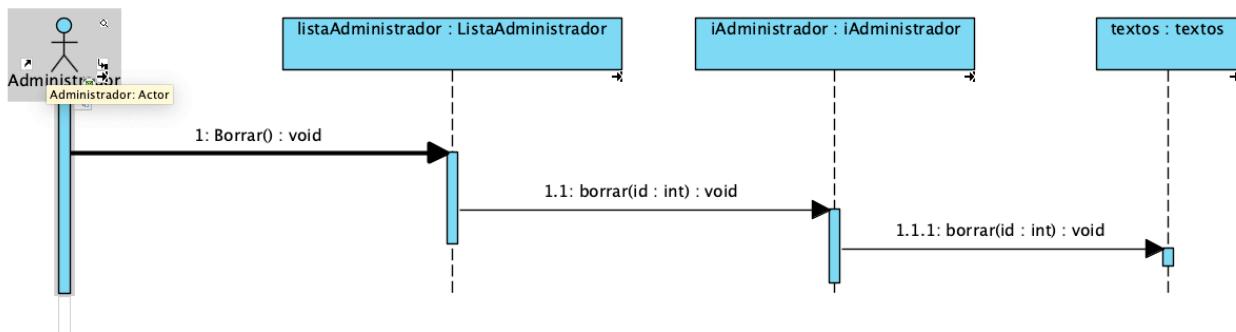
FAQ: En el anexo de esta guía aparecen más ejemplos.

FAQ: Para que los diagramas en la documentación respeten el tamaño original:

<https://knowhow.visual-paradigm.com/reporting/publisher-blurry-image/>

En el ejemplo de proyecto quedarían como sigue:





ACTIVIDAD 11: IMPLEMENTACIÓN DE LOS COMPONENTS ORM

Ahora vamos a implementar los ORMCcomponents. Mostramos a continuación ejemplos diversos del tipo de operaciones que hay que implementar. Obsérvese que en las operaciones de modificación hay que hacer *ProyectoPersistentManager.instance().disposePersistentManager();* mientras en el resto no hay que hacerlo.

Registrarse

```
public Logueado registro(String login, String password)
    throws PersistentException{
PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
Logueado logueado = null;
try {
    logueado = LogueadoDAO.createLogueado();
    logueado.setLogin(login);
    logueado.setPassword(password);
    LogueadoDAO.save(logueado);
    t.commit();
}
catch (Exception e) {
    t.rollback();
}
ProyectoPersistentManager.instance().disposePersistentManager();
return logueado;
}
```

Login

```
public Logueado login(String login, String password)
    throws PersistentException{
Logueado logueado = null;
PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
try {
    logueado = LogueadoDAO.loadLogueadoByQuery(
        "Login = '" +login+ "' AND Password = '"
            + password + "'", null);
    t.commit();
} catch (Exception e) {
    t.rollback();
}
```

```
}

return logueado;
}
```

Cargar Usuarios

```
public List cargarUsuarios()
    throws PersistentException {
List<Logueado> usuarios = null;
PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
try {
    usuarios = LogueadoDAO.queryLogueado(null, null);
    t.commit();
} catch (Exception e) {
    t.rollback();
}
return usuarios;
}
```

Eliminar Usuario

```
public void eliminar(int Id)
    throws PersistentException {
Logueado logueado = null;
PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
try {
    logueado = LogueadoDAO.getLogueadoByORMID(Id);
    LogueadoDAO.deleteAndDissociate(logueado);
    t.commit();
} catch (Exception e) {
    t.rollback();
}
ProyectoPersistentManager.instance().disposePersistentManager();
}
```

El borrado eliminar los usuarios y toda su información en cascada. Para ello tiene que tener asociaciones a su información.

Seguir a usuario

```
public void seguirUsuario(int seguidor, int seguido)
    throws PersistentException {
```

```

PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
Logueado logueado = null;
try {
    logueado = LogueadoDAO.getLogueadoByORMID(seguidor);
    usuario_seguido=LogueadoDAO.getLogueadoByORMID(seguido);
    usuario_seguido.seguidores.add(logueado);
    LogueadoDAO.save();
    t.commit();
} catch (Exception e) {
    t.rollback();
}
ProyectoPersistentManager.instance().disposePersistentManager();
}

```

Buscar hashtags

```

public List buscarListaHashtagCoincidenteItem(String hashtag)
throws PersistentException {
PersistentTransaction t = ProyectoPersistentManager.instance()
    .getSession().beginTransaction();
List<Hashtag> hashtags = null;
try {
    hashtags = HashtagDAO.queryHashtag(
        "Hashtag LIKE '%" + hashtag + "%'", null);
    t.commit();
} catch(Exception e) {
    t.rollback();
}
return hashtags;
}

```

Implementación de los Business

Por último tenemos que implementar los casos de uso etiquetados con <<Business>>. Por ejemplo:

```

public class RecuperarContrasena extends VistaRecuperarContrasena
{
public Serviciodecorreo serviciodecorreo = new Serviciodecorreo();
public void enviar_password(String email) {
    serviciodecorreo.enviar_password(email);
}

```

```

public class Serviciodecorreo{

```

```
public enviar_password(String email)
    {Notification.show("Correo Enviado")}
```

Entrega 13 (Aula Virtual): Código ORM

ACTIVIDAD 12: ACCESO A LOS COMPONENTES ORM

Nos queda por último hacer uso de los componentes ORM en el interfaz de usuario. En esencia lo que hay que hacer es usar los métodos que hemos especificado en los diagramas de secuencia para crear, actualizar, buscar o borrar los elementos de la base de datos y que la aplicación sea completamente funcional.

El código que hemos construido progresivamente está preparado para ello, de modo que con pocas modificaciones podamos finalizar la implementación.

[En el ejemplo de proyecto, podemos primero actualizar login, dar de alta y actualizar perfil.](#)

```
this.getBotonlogin().addClickListener(event -> {

    registrado r =
    this._cibernauta._iCibernauta.login(this.getLogin().getValue(),
                                         this.getPassword().getValue());
    this._cibernauta.MainView.removeAll();
    if (r instanceof usuario) {
        Usuario u = new Usuario(this._cibernauta.MainView, (basededatos.usuario)
        r);
        this._cibernauta.MainView.add(u);
    }
    else if (r instanceof administrador) {
        Administrador a = new Administrador(this._cibernauta.MainView);
        this._cibernauta.MainView.add(a);
    }
    else {
        this._cibernauta.MainView.add(this._cibernauta);
        Notification.show("Este usuario no existe");
    }});
}
```

```
public void Dardealta() {
    this._cibernauta._iCibernauta.registrar(this.getDni().getValue(),
                                              this.getNombre().getValue(), this.getCorreo().getValue(),
                                              this.getNick().getValue(), this.getLogin().getValue(),
                                              this.getPassword().getValue());
    this._cibernauta.MainView.removeAll();
    this._cibernauta.MainView.add(this._cibernauta);
}
```

```
public void Actualizar() {
    usuario u = this._usuario._iUsuario.actualizar(_usuario.u.getID(),
                                                   this.getNombre().getValue(), this.getDni().getValue(),
                                                   this.getCorreo().getValue(), this.getNick().getValue(),
                                                   this.getLogin().getValue(), this.getPassword().getValue());
    this._usuario = new Usuario(this._usuario.MainView, u);
    this._usuario.MainView.removeAll();
```

```
    this._usuario.MainView.add(this._usuario);}}
```

A continuación podemos actualizar Añadir y Modificar de la lista de usuarios y Borrar en la lista de Administradores:

```
this.getBotonAñadir().addClickListener(event -> {
    _listaUsuario._usuario._iUsuario.anadir(_listaUsuario._usuario.u.getID(),
    this.getTexto().getValue());
    usuario nubd =
    _listaUsuario._usuario._iUsuario.obtenerUsuarioporId(_listaUsuario._usuario.u.getID());
    Usuario nu = new Usuario(_listaUsuario._usuario.MainView, nubd);
    _listaUsuario._usuario.MainView.removeAll();
    _listaUsuario._usuario.MainView.add(nu));
```

```
this.getBotonModificar().addClickListener(event -> {
    ListaUsuario lu = (ListaUsuario) _listaUsuario._lista;
    lu._usuario._iUsuario.modificar(_listaUsuario.t.getID(),
    this.getTexto().getValue());
    usuario nubd =
    lu._usuario._iUsuario.obtenerUsuarioporId(lu._usuario.u.getID());
    Usuario nu = new Usuario(lu._usuario.MainView, nubd);
    lu._usuario.MainView.removeAll();
    lu._usuario.MainView.add(nu);
});
```

```
public void Borrar(){
for (int i = 0; i < _item.size(); i++) {
    _administrador._iAdministrador.borrar(_item.get(i).t.getID());
}
this.getBorrar().setEnabled(false);
_administrador.MainView.removeAll();
_administrador = new Administrador(_administrador.MainView);
_administrador.MainView.add(_administrador);
}
```

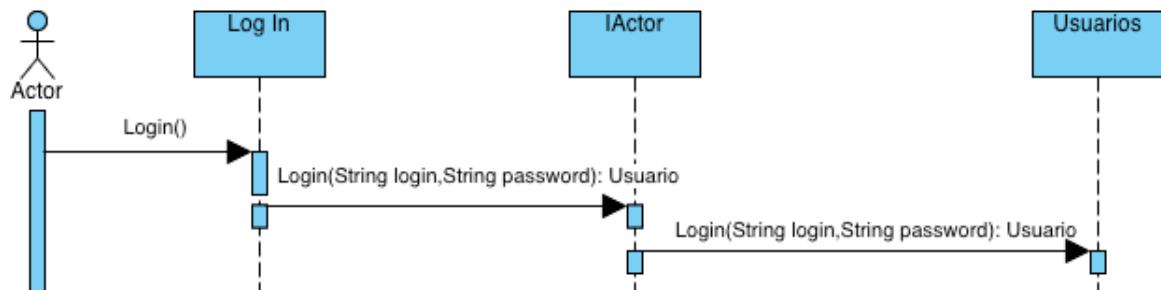
Entrega 14 (Aula Virtual): Documentación Final

ANEXO: DIAGRAMAS DE SECUENCIA

Ejemplo 1

Diagrama de secuencia del botón de login

The screenshot shows a login form titled "Log in". It contains two text input fields: "Username •" and "Password •", both with placeholder text. Below the password field is an "eye" icon for password visibility. A large blue "Log in" button is centered at the bottom. At the bottom right, there is a link "Forgot password".



Este diagrama de secuencia se corresponde con este código:

```
public class LogIn {
    IActor iactor = new BD_Principal();
    public Usuario usuario;
    public LogIn(Padre padre){
        this.getLoginButton().addClickListener(event -> Login())
    }
    public Usuario Login(){
        usuario = iactor.Login(this.getLogin().getText(),
            this.getPassword().getText());
        padre.MainView.as(VerticalLayout.class)
            .removeAll();
        padre.MainView.logueado = new
            Logueado(actor.MainView,usuario);
        padre.MainView.as(VerticalLayout.class).add(
    }
```

```

        actor.MainView.logueado);
}

```

Nótese que hemos declarado un atributo de tipo usuario para guardar el usuario que cargamos desde la base de datos. A su vez en la base de datos principal (que implementa el IActor):

```

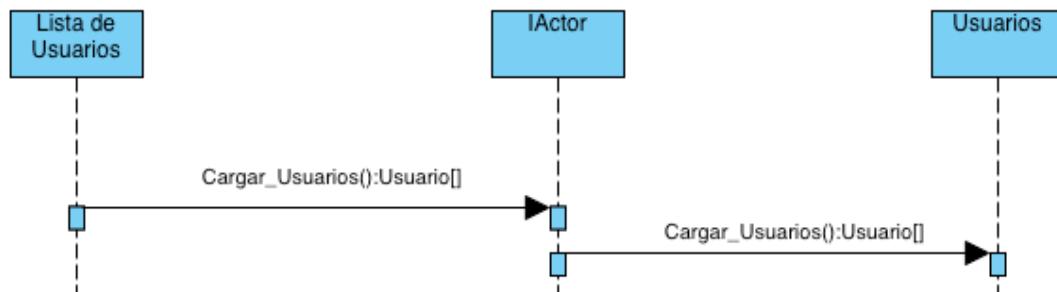
public class BD_Principal implements IActor{
public Usuarios bd_usuarios = new Usuarios();
public Usuario Login(String login, String password){
    return bd_usuarios.Login(login, password)}}

```

Ejemplo 2

First Name	Last Name	Email
Bernard	Nilsen	bernard@nilsen.com
Jaydan	Jackson	jaydan@jackson.com
Solomon	Olsen	solomon@olsen.com
Elvis	Olsen	elvis@olsen.com
Rene	Carlsson	rene@carlsson.com
Remington	Andersson	remington@andersson.com
Ann	Andersson	ann@andersson.com
Lara	Martin	lara@martin.com
Jamar	Olsson	jamar@olsson.com
Gunner	Karlsen	gunner@karlsen.com
Leland	Harris	leland@harris.com
Danielle	Watson	danielle@watson.com
Makenna	Smith	makenna@smith.com
Quinn	Hansson	quinn@hansson.com

Diagrama de secuencia de cargar usuarios en la lista



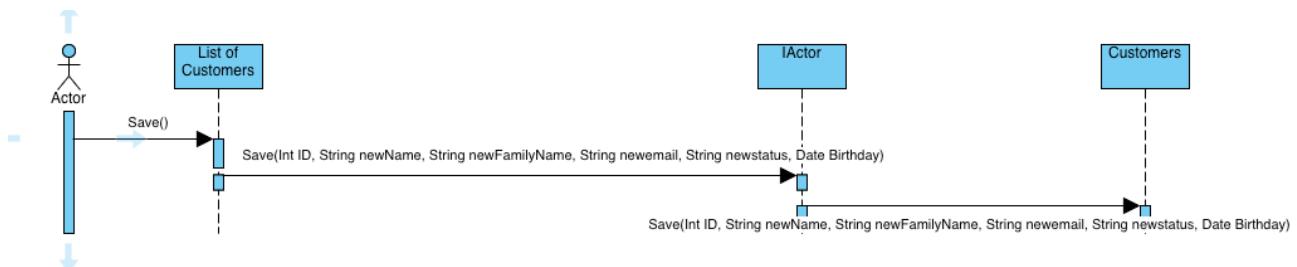
Este diagrama de secuencia se corresponde con este código:

```
public class ListadeUsuarios {  
    IActor iactor = new BD_Principal();  
    public Usuario[] usuarios;  
    public ListadeUsuarios(Padre padre){  
        Cargar_Usuarios();}  
    public void Cargar_Usuarios(){  
        Usuarios = iactor.Cargar_Usuarios();  
        for (Int i=0;i++,i<usuarios.size){  
            UsuariosItem ui = new UsuariosItem(this,usuarios[i]);  
            this.getLista().as(VerticalLayout.class).add(ui);}}}
```

Nótese que cargamos los usuarios de la base de datos, creamos la vista de cada uno de ellos y los añadimos al contenido de la lista. A su vez en la base de datos principal (que implementa el IActor):

```
public class BD_Principal extends IActor{  
    public Usuarios bd_usuarios = new Usuarios();  
    public Usuario[] Cargar_Usuarios(){  
        return bd_usuarios.Cargar_Usuarios();}}
```

Diagrama de secuencia de save



Este diagrama de secuencia se corresponde con este código:

```
public class ListOfCustomers {  
    IActor iactor = new BD_Principal();  
    public ListOfCustomers(Padre padre){  
        this.getSaveButton().addClickListener(event -> Save())}  
    public actor Save(){  
        return iactor.Save(this.getNewName().getText(),  
                           this.getNewFamilyName().getText(),...);}}
```

A su vez en la base de datos principal (que implementa el IActor):

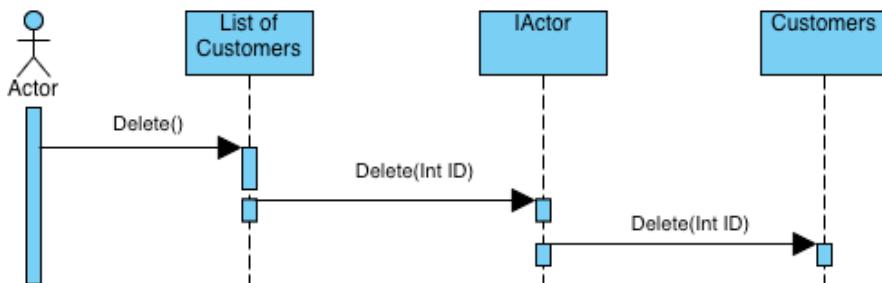
```
public class BD_Principal implements IActor{
```

```

public Customers bd_customers = new Customers();
public actor Save(String Name, String FamilyName,...){
    return bd_customers.Save(Name,FamilyName,...)}}

```

Diagrama de secuencia de delete



Este diagrama de secuencia se corresponde con este código:

```

public class ListOfCustomers {
    IActor iactor = new BD_Principal();
    public ListOfCustomers(Padre padre){
        this.getDeleteButton()
            .addClickListener(event -> Delete())
    }
    public void Delete(){
        iactor.Delete(selected_user.getID());}}

```

A su vez en la base de datos principal (que implementa el IActor):

```

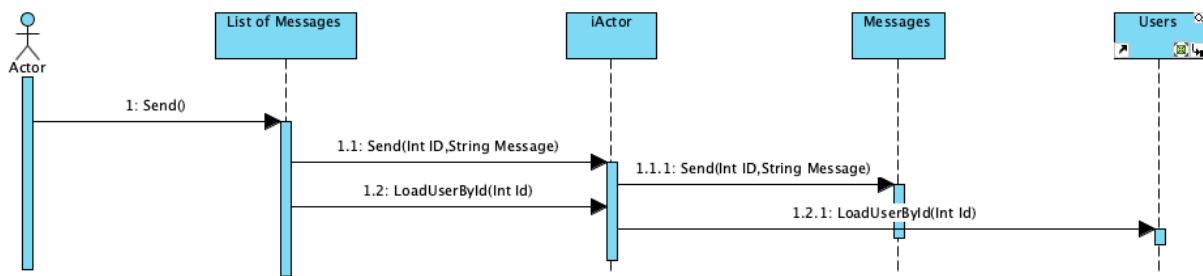
public class BD_Principal implements IActor{
    public Customers bd_customers = new Customers();
    public void Delete(Int Id){
        bd_customers.Delete(Id)}}

```

Ejemplo 3



Diagrama de secuencia de send mensaje



Este diagrama de secuencia se corresponde con este código:

```
public class ListOfMessages {
    IActor iactor = new BD_Principal();
    public ListOfMessages(Padre padre){
        this.getSendButton().addClickListener(event -> Send())
    }
    public void Send(){
        iactor.Send(padre.usuario.getID(),
                    logueado.getID(),this.getMessage().getText());
        actor a = iactor.LoadUserById(
                    padre.logueado.getID())}}}
```

A su vez en la base de datos principal (que implementa el IActor):

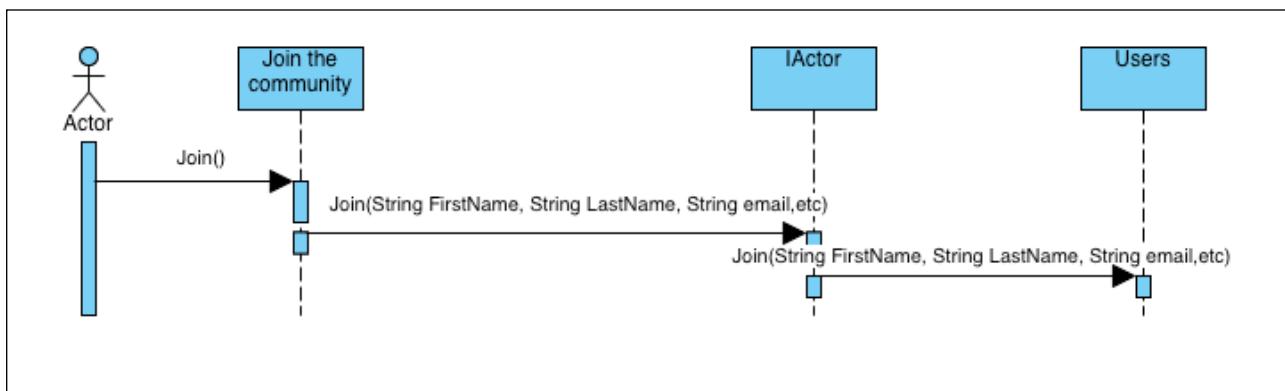
```
public class BD_Principal implements IActor{
    Messages bd_messages = new Messages();
    public void Send(Int ID, String Message){
        bd_messages.Send(ID,Message)}}
    public actor LoadUserById(Int ID){
        return bd_users.LoadUserById(ID)}}}
```

Ejemplo 4

Signup form

First name *	Last name *
Email *	
Password *	Confirm password *
<input type="checkbox"/> Allow Marketing Emails?	
Join the community	

Diagrama de secuencia de join the community



Este diagrama de secuencia se corresponde con este código:

```
public class JoinTheCommunity {  
    IActor iactor = new BD_Principal();  
    public JoinTheCommunity(Padre padre){  
        this.getJoinButton().addClickListener(event -> Join())  
    }  
    public actor Join(){  
        actor = iactor.Join(this.getFirstName().getText(),  
                            this.getLastName().getText(),...);}}}
```

A su vez en la base de datos principal (que implementa el IActor):

```
public class BD_Principal implements IActor{  
    Users bd_users = new Users();  
    public actor Join(String FirstName, String LastName,...){  
        return bd_users.Join(FirstName,LastName,...)} }
```

Ejemplo 5

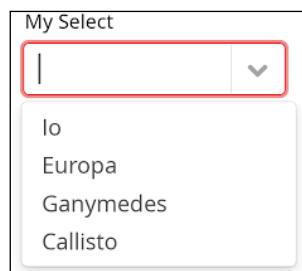
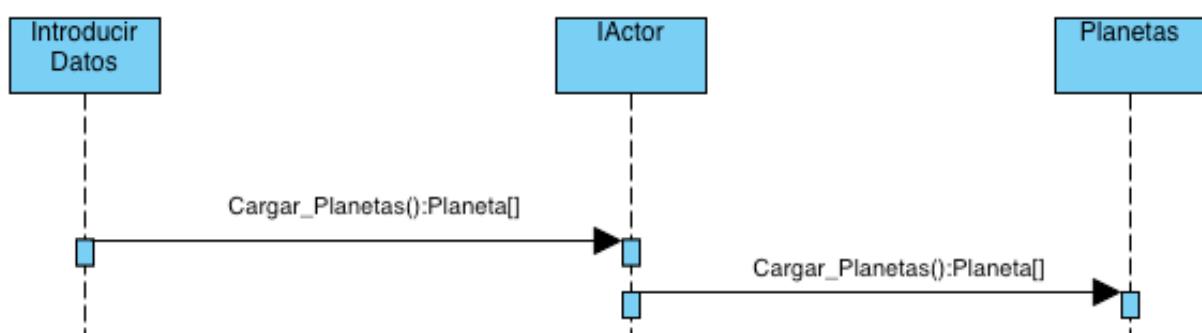


Diagrama de secuencia de cargar planetas



Este diagrama de secuencia se corresponde con este código:

```
public class IntroducirDatos {
    IActor iactor = new BD_Principal();
    public Planeta[] planetas;
    public IntroducirDatos(Padre padre){
        Cargar_Planetas();}}
    public Planeta[] Cargar_Planetas(){
        planetas = iactor.Cargar_Planetas();
        this.getComboBox().setItems(planetas);}}
```

A su vez en la base de datos principal (que implementa el IActor):

```
public class BD_Principal implements IActor{
    Planetas bd_planetas = new Planetas();
    public Planeta[] Cargar_Planetas(){
        return bd_planetas.Cargar_Planetas();}}
```

Ejemplo 6

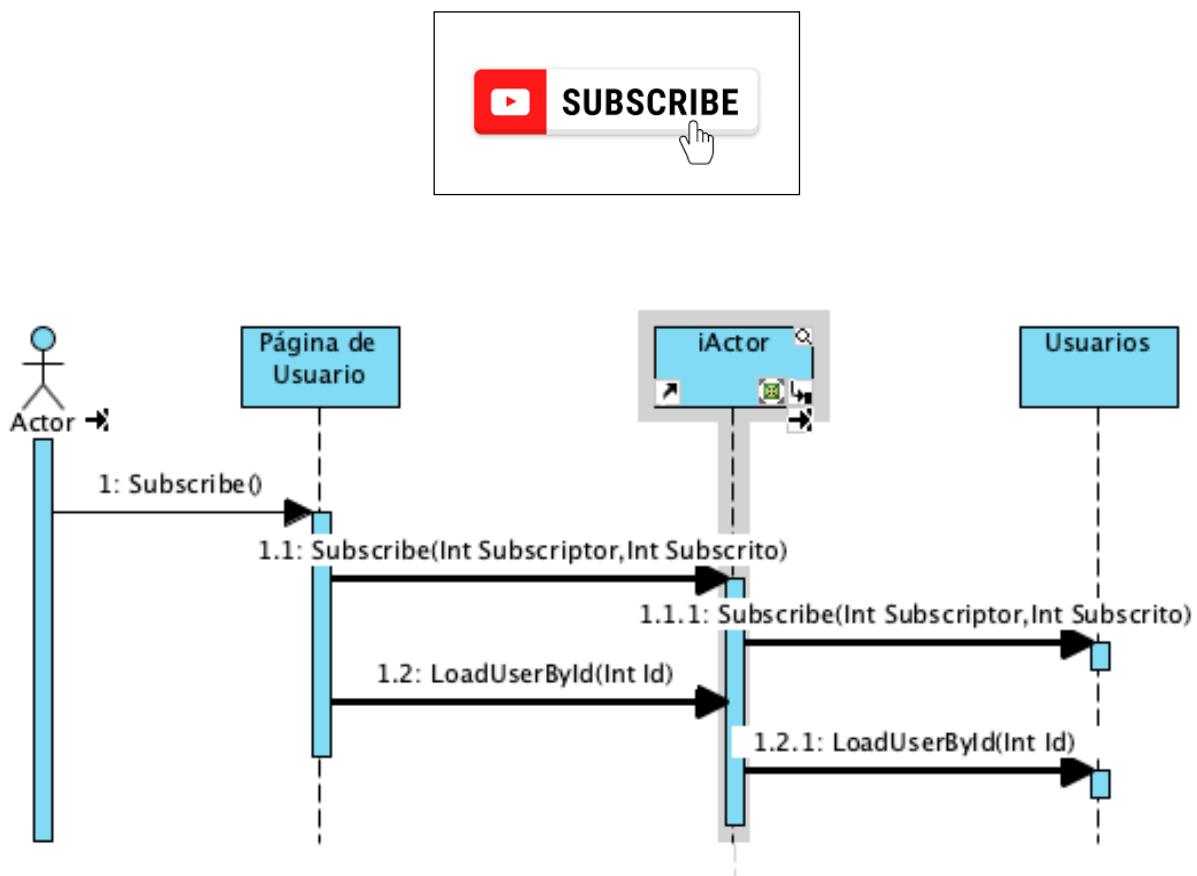


Diagrama de secuencia de subscribe

Este diagrama de secuencia se corresponde con este código:

```

public class PaginaDeUsuario {
    IActor iactor = new BD_Principal();
    public PaginaDeUsuario(Padre padre){
        this.getSubscribeButton()
            .addClickListener(event -> Subscribe())
    }
    public void Subscribe(){
        iactor.Subscribe(padre...logueado.getID(),
                        usuario_visitado.getID());
        actor = iactor.LoadUserById(padre...logueado.getID());}}}

```

A su vez en la base de datos principal (que implementa el IActor):

```

public class BD_Principal implements IActor{
    Usuarios bd_usuarios = new Usuarios();
    public void Subscribe(Int Subscriptor, Int Subscrito){
        bd_usuarios.Subscribe(Subscriptor,Subscrito)}}}
    public actor LoadUserById(Int Usuario){
        return bd_usuarios.LoadUserById(Usuario)}}}

```

Ejemplo 7

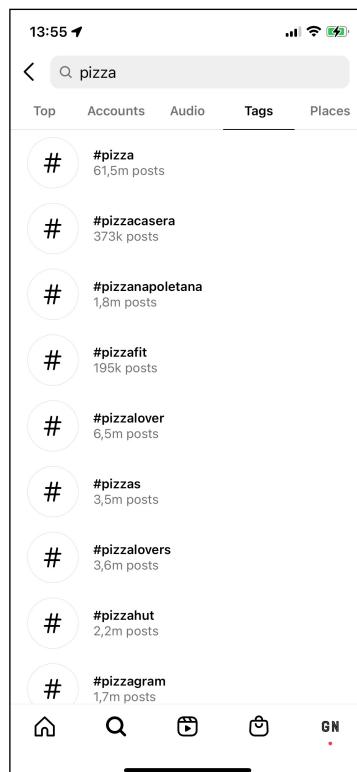
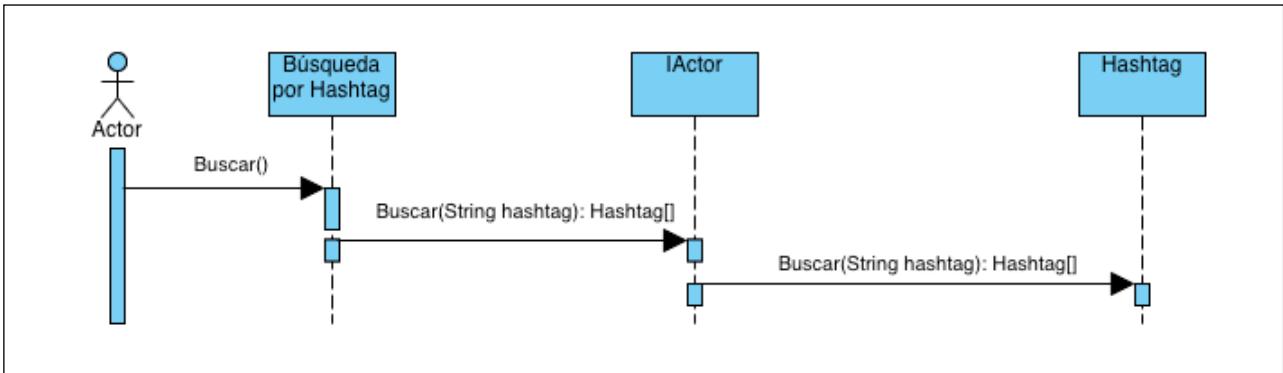


Diagrama de secuencia de buscar por hashtag

Este diagrama de secuencia se corresponde con este código:



```

public class BusquedaPorHashtag {
    IActor iactor = new BD_Principal();
    public Hashtag[] hashtags;
    public BusquedaPorHashtag(Padre padre){
        this.getBuscarButton().addClickListener(event ->{
            hashtags =Buscar();
        })
    }
    public Hashtag[] Buscar(){
        return iactor.Buscar(this.getHashtag().getText());
        for (Int i=0;i++,i<hashtags.size){
            HashtagsItem hi =
                new HashtagsItem(this,hashtags[i]);
            this.getContenido().as(VerticalLayout.class)
                .add(hi);}}}}}

```

A su vez en la base de datos principal (que implementa el IActor):

```

public class BD_Principal implements IActor{
    Hashtags bd_hashtags = new Hashtags();
    public Hashtag[] Buscar(String hashtag){
        return bd_hashtags.Buscar(hashtag)}}

```

ANEXO: UPLOAD

```
public class MainView extends VerticalLayout {
    private static final String IMAGE_PATH = "src/main/resources/META-INF/resources/uploads/";
    private static final String UPLOAD_DIR = "src/main/resources/META-INF/resources/uploads/";
    public MainView() {
        createUploadDirectory();
        FileBuffer buffer = new FileBuffer();
        Upload upload = new Upload(buffer);
        upload.setAcceptedFileTypes("image/jpeg", "image/png",
"image/gif");
        add(upload);
        upload.addSucceededListener(event -> {
            File uploadedFile = buffer.getFileData().getFile();
            try {
                Path destinationPath = Paths.get(UPLOAD_DIR +
event.getFileName());
                Files.move(uploadedFile.toPath(), destinationPath);
                Notification.show("Image uploaded successfully!");
                Image img =
createImageFromFile(IMAGE_PATH+event.getFileName());
                add(img);
            } catch (IOException e) {
                Notification.show("Error saving the image: " +
e.getMessage(), 5000,
Notification.Position.MIDDLE);
            }
        });
        upload.addFailedListener(event -> {
            Notification.show("Image upload failed: " +
event.getReason().getMessage(), 5000,
Notification.Position.MIDDLE);
        });
        upload.addFileRejectedListener(event -> {
            Notification.show("File rejected: " +
event.getErrorMessage(), 5000,
Notification.Position.MIDDLE);
        });
    }

    private void createUploadDirectory() {
```

```

Path uploadDirPath = Paths.get(UPLOAD_DIR);
if (!Files.exists(uploadDirPath)) {
    try {
        Files.createDirectories(uploadDirPath);
    } catch (IOException e) {
        throw new RuntimeException("Could not create upload
directory: " + e.getMessage(), e);
    }
}

private Image createImageFromFile(String filePath) {
    File file = new File(filePath);
    if (file.exists()) {
        StreamResource resource = new StreamResource(file.getName(),
() -> {
    try {
        return new FileInputStream(file);
    } catch (FileNotFoundException e) {
        Notification.show("Error: " + e.getMessage(), 5000,
Notification.Position.MIDDLE);
        return null;
    }
});
    Image image = new Image(resource, "Image not found");
    image.setMaxWidth("500px");
    return image;
    } else {
        Notification.show("File not found: " + filePath, 5000,
Notification.Position.MIDDLE);
        return new Image();
    }
}

```