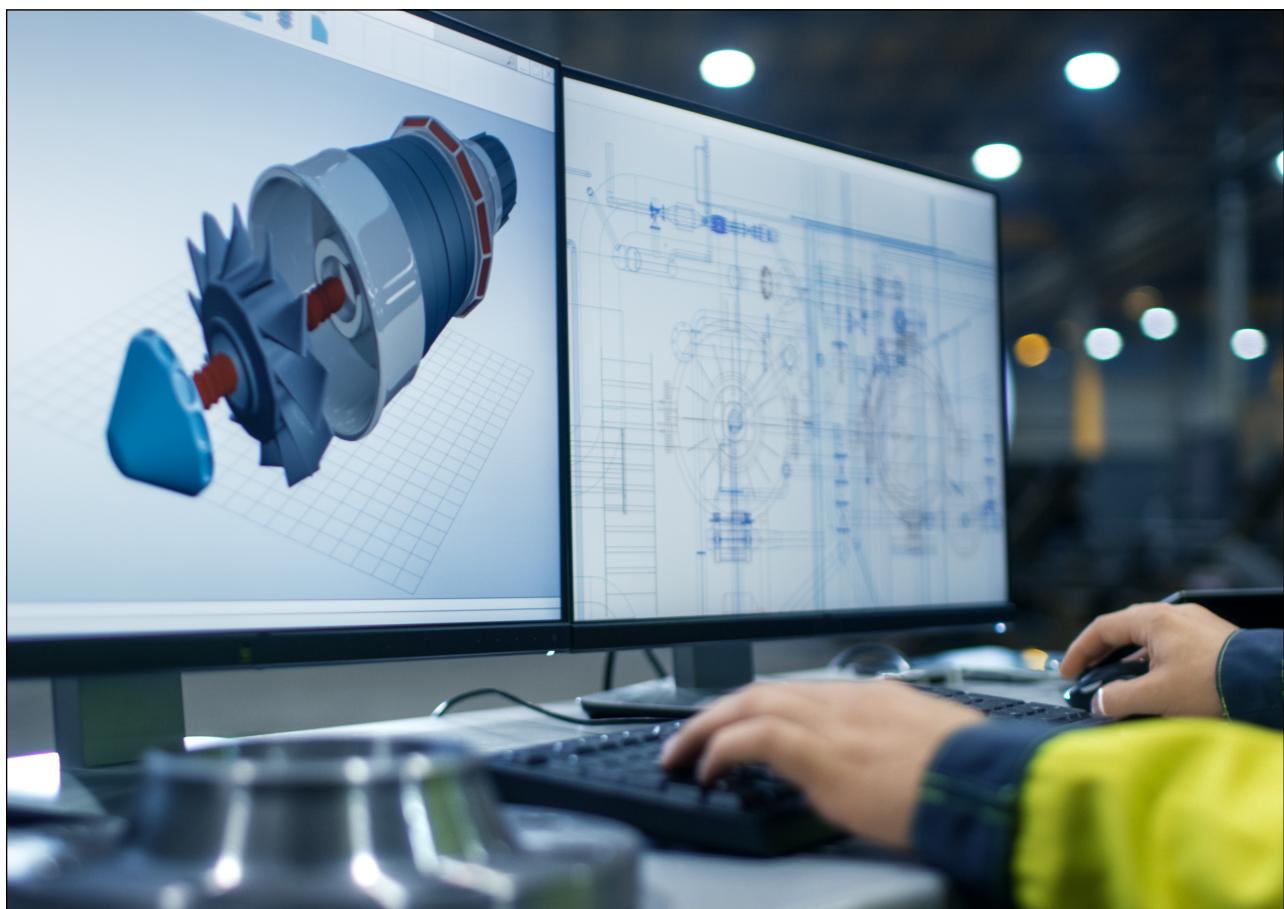


Modelado y Diseño del Software 2

Jesús Manuel Almendros Jiménez

Grado en Ingeniería Informática

Universidad de Almería



Plan de Actividades

Actividad 1. PROYECTO VAADIN

Actividad 2. GENERACIÓN DE CÓDIGO

Actividad 3. VISTAS DE LOS ACTORES

Actividad 4. VISTAS DE LAS LISTAS

Actividad 5. VISTAS DE OTROS CASOS DE USO

Actividad 6. ENSAMBLADO DE COMPONENTES ESTÁTICOS

Actividad 7. ENSAMBLADO DE COMPONENTES DINÁMICOS

Actividad 8. GENERACIÓN DE CÓDIGO ORM

Actividad 9. ACTUALIZACIÓN DE LOS CONSTRUCTORES

Actividad 10. DIAGRAMAS DE SECUENCIA

Actividad 11. IMPLEMENTACIÓN DE ORM COMPONENTS

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.

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.

Objetivo: Alta en el Equipo de Trabajo en el Aula Virtual

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*:

<https://www.eclipse.org/downloads/packages/release/2022-12/m3/eclipse-ide-enterprise-java-and-web-developers>

Es importante que sea la versión 2022.

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

- iv) Debemos tener Java instalado en nuestra máquina (*JAVA Development Kit 8*).

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

<https://nodejs.org/es>

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

- vi) Instalamos *EGit* desde *Eclipse Marketplace*.

- vii) Instalamos XAMPP:

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

Objetivo: Instalación del Software

Entrega del proyecto de MDS1.

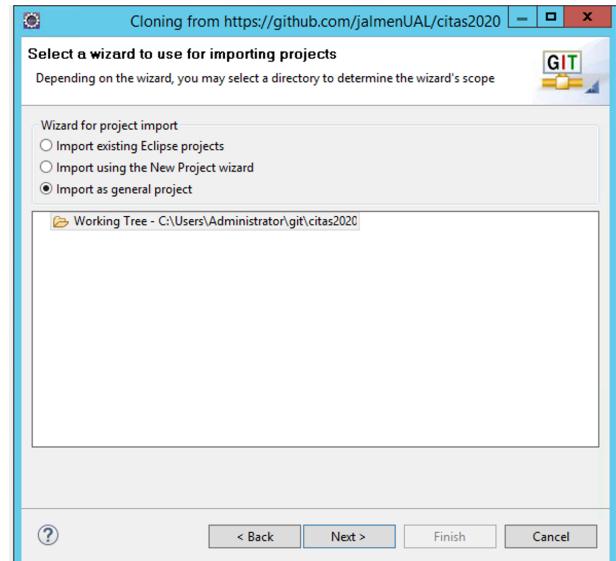
Se ha de enviar a través de la actividad creada a tal efecto la documentación del proyecto de MDS1 tal y como se envió en esa asignatura.

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

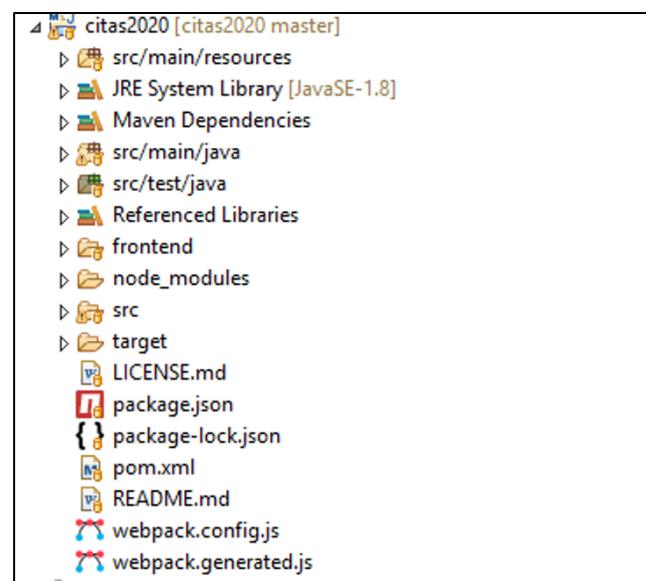
Entrega 1 (Aula Virtual): ZIP con la documentación de MDS1

Importado del Proyecto Ejemplo.

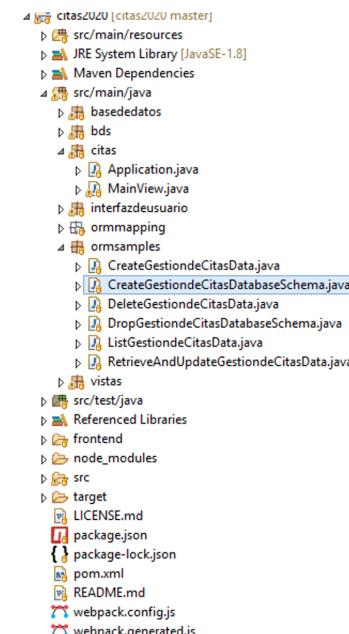
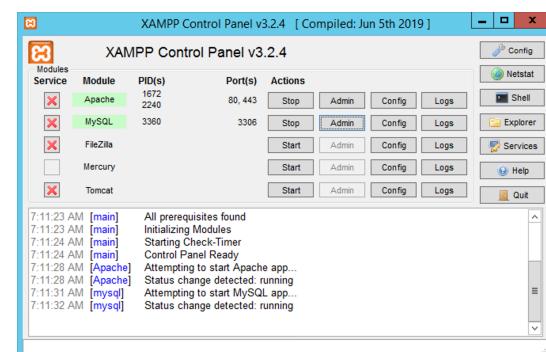
- i) Nos vamos a <https://github.com/jalmenUAL/citas2023>. Copiamos la dirección del código del proyecto.
- ii) Desde el menú principal de Eclipse, seleccionamos *File-> Import->Git->Projects From Git->Clone URI*
- iii) Pulsamos *siguiente* y elegimos un nombre para la carpeta (por defecto pondrá el nombre del proyecto) dentro de la carpeta Git local.
- iv) Pulsamos *siguiente*. Una vez leído le damos a *Cancelar* (ventana en el que aparece las tres opciones de Import)
- v) A continuación, de nuevo, desde el menú principal de Eclipse: *File->Import->Maven->Existing Maven Projects*
- vi) 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*, que tiene la estructura que aparece en la imagen.
- vii) 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. Ya tenemos importado el proyecto ejemplo.
- viii) Ahora hay que descargar del aula virtual el fichero VisualParadigm del proyecto. Abrídrlo para poder comparar el proyecto VP con el proyecto Eclipse. En la carpeta *citas* 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*. En la carpeta *interfazdeusuario* están las clases de la interfaz de usuario del proyecto de Visual Paradigm. En la carpeta *vistas* están las vistas que genera *Vaadin*. En la carpeta *basededatos* está la base de datos. Finalmente 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 para crear la base de datos y borrarla. También hay programas Java de ejemplo. En la carpeta *frontend/src* están las vistas Vaadin. Pinchad en cualquier *.ts* y veréis el diseño. Si se abre en el propio Eclipse podéis cambiarlo para verlo en Google Chrome yendo a *Preferencias->Vaadin* y cambiando el editor por defecto.



- ix) Para ejecutar el programa antes hay que arrancar el MySQL (*y Apache*) del XAMPP.
- x) Una vez arrancados, hay que crear la base de datos. Esto solo se hace la primera vez que se ejecuta la aplicación. Esto se hace ejecutando el programa Java *CreateDatabaseSchema* que aparece en la carpeta *ormsamples*. Una vez ejecutado este programa, ya tenéis la base de datos creada en MySQL. Ahora solo queda ejecutar el programa *Application.java* como *Run As->Java Application*. Al ejecutarlo, podéis abrir la aplicación en: <http://localhost:8080>. Podéis entrar como *admin* (sin password) o como *usuario* (sin password). En el aula virtual también se encuentra el zip de la documentación del proyecto.
- xi) Ahora hay que añadir 5 *citas* en modo *administrador*. Y en modo usuario hay que *posponer 2 de ellas y dar por finalizadas 2*.



Entrega 2 (Aula Virtual): Capturas de las ventanas de administrador y de usuario del proyecto citas (con los datos introducidos)

Creación del Proyecto Vaadin.

- i) El primer paso consiste en crear un proyecto *Vaadin 10+ (Vaadin Project (Maven))* en Eclipse: *File->New->Other->Vaadin->Vaadin 10+ Project* e introducir los datos del proyecto.
El *Group ID* es el nombre de la carpeta donde va a ir colocado el código, y el *Project Name* es el nombre del proyecto:
1) Group ID: *projetoMDS*

- 2) Project Name: apellidos de los miembros del grupo. Por ejemplo, *IribarneAlmendros*.

Esto es importante para que el profesor pueda importar el proyecto a Eclipse

- 3) Una vez creado 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/src* y *frontend/styles* donde van las vistas de las ventanas y los estilos de las mismas.

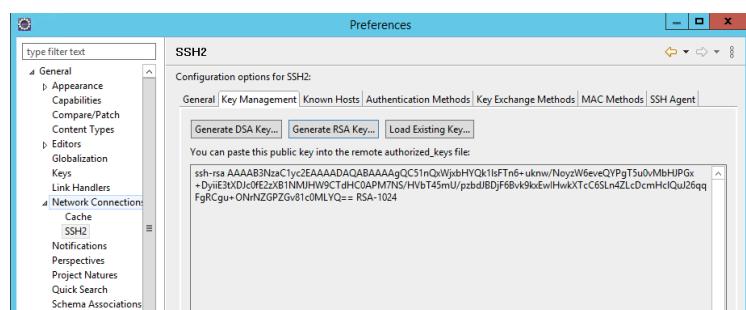
Estas son las únicas carpetas que vamos a modificar. El resto son gestionadas por Vaadin.

- 4) De momento solo tendremos disponible en *src/main/java* los tres programas por defecto que genera Vaadin: *MainView*, *Application* y *GreenService*. Borramos *GreenService* y quitamos el parámetro. El código de *Application.java* no hay que tocarlo de momento. El único que hay que modificar es el *MainView.java*. Este programa de momento tiene un trozo de código que escribe un saludo con tu nombre. Luego lo cambiaremos.

- ii) Ejecutad el programa *Application.java* (como no hay base de datos no hace falta arrancar el *MySQL del XAMPP*). Este proyecto lo vamos a utilizar para desarrollar la implementación. Conforme vayamos añadiendo el código iremos creando las carpetas correspondientes (al estilo del proyecto ejemplo).

- iii) Guardaremos nuestro proyecto en el *Github*. Primero debemos crear una *key ssh-rsa* en *Eclipse*. Y guardarla en cualquier carpeta. En *Windows->Preferences* de *Eclipse* como se muestra en la imagen.

En *settings del GitHub* debemos ir a *SSH and GPG keys* y crear una nueva *SSH key* y guardarla. Y copiar el código del *ssh-rsa* de *Eclipse* en esa key.



Importante: Parad la ejecución del proyecto Vaadin antes de hacer el siguiente paso.

- iv) 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*.

The screenshot shows the GitHub repository settings interface. On the left, there's a sidebar with options like General, Access (selected), Collaborators (highlighted in blue), Moderation options, Code and automation (Branches, Tags, Actions, Webhooks, Environments, Codespaces, Pages), and Security (Code security and analysis, Deploy keys). The main area is titled 'Who has access' and shows it's a 'PUBLIC REPOSITORY'. It says 'This repository is public and visible to anyone.' and 'DIRECT ACCESS'. It indicates '0 collaborators have access to this repository. Only you can contribute to this repository.' There's a 'Manage' button. Below this, a section titled 'Manage access' says 'You haven't invited any collaborators yet' and has a green 'Add people' button.

- v) Ahora vamos a compartir con nuestro compañero de equipo el proyecto.
- vi) A partir de ahora se van a hacer algunas entregas a través de Github. El enlace al repositorio de Github donde hay que enviarlo está 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.

Importante: Hay que tener en cuenta que la primera vez que se descarga el proyecto tu compañero ha de hacer Maven => Update Project. Update Project se ha de realizar cada que vez que se hacen cambios significativos en el proyecto tales como cambios en el pom, inclusión de librerías, etc. Es un recurso bastante habitual si el proyecto da error.

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 ORMPersistent 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*. 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.

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).

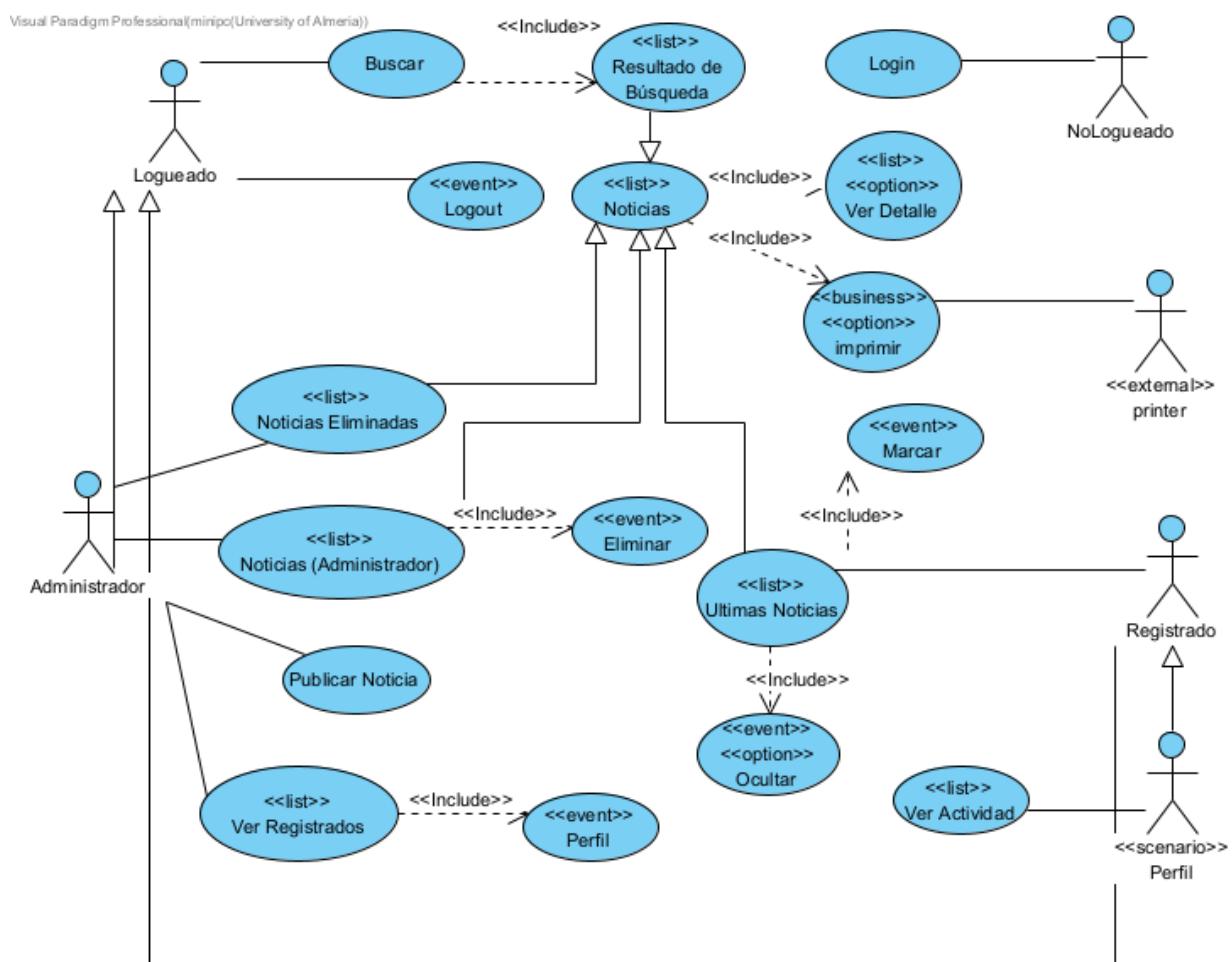
- i) Instalamos en *Visual Paradigm* el *plugin* que está disponible en el aula virtual. Al ejecutar el plugin hay que elegir una carpeta en la que se colocará el código generado.
- ii) A continuación creamos un paquete Eclipse que se llame *vistas* dentro de *src/main / java*.
- iii) Ahora hay que importar 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/src*.
- v) A continuación probamos que todo se ha generado de forma exitosa. Abrid cualquier *.ts* de la carpeta *frontend/src*. Debería estar vacío y el editor listo para editar la vista.

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

ACTIVIDAD 3: VISTAS DE LOS ACTORES

Vistas de los actores (No external).

Vamos a proceder a la edición de las vistas que tenemos en la carpeta *frontend/src*, añadiendo componentes gráficas. El plugin ha generado vistas para los actores y para los casos de uso. Sin embargo, no todos los actores y casos de uso tienen vista. Supongamos que tenemos el siguiente ejemplo.



En este caso, de entre los actores solo tendrían vista: *Logueado* y *NoLogueado*. La razón es que los demás heredan de otro actor: *Administrador*, *Registrado* heredan de *Logueado*, y *Perfil* hereda de *Logueado*. Tampoco tendría vista *Printer* porque es un *<<external>>*. De entre los casos de uso solo tendría vista *Buscar*, *Login*, *Noticias*, *Ver Detalle*, *Publicar Noticia*, *Ver Registrados* y *Ver Actividad*. El resto no tendrían vista porque heredan de otro caso de uso o porque son *<<event>>/<<option>>* o *<<business>>*. En particular, de entre las

<<list>> solo tendría vista *Noticias* que es la que está más arriba en la jerarquía de herencia.

¿Qué hacemos con los que heredan y no tienen vista? Pues debemos poner en la vista de la que heredan todos los componentes gráficos que necesitan. O sea, en cada vista debe aparecer la suma de todos los componentes gráficos que necesitan los actores o los casos de uso que heredan. Posteriormente, como veremos, reconstruiremos cada actor o caso de uso ocultando las componentes gráficas que no necesita. Las asociaciones entre actores y casos de uso y las relaciones de inclusión entre casos de uso dan lugar a vistas separadas. Luego se colocarán en su sitio correspondiente.

Diseño de los Actores (No external) que no heredan

En el ejemplo son *Logueado* y *NoLogueado*. Al ser actores, representan la ventana principal del usuario en cuestión y normalmente contiene los botones de acceso a la funcionalidad que aparece en el diagrama de casos de uso (la conectada directamente al actor). También puede hacerse con menú desplegable. Depende de lo que hayamos elegido en MDS1. Dado que tenemos que poner toda la funcionalidad *Logueado* y *NoLogueado* entonces ponemos botones o opciones de menú para toda la funcionalidad de los actores que heredan.

Por ejemplo, en *Logueado* el *Logout* es común a todos los usuarios logueados. Pero también añadimos un botón para *Noticias Eliminadas*, *Noticias de Administrador*, *Publicar Noticia* y *Ver Registrados*, que es de *Administrador*, para *Ver Actividad* que es de *Perfil* y para *Últimas Noticias* que es de *Registrado*. Después ocultaremos los botones para que quede en cada uno lo que corresponda.



Importante: Conforme se van creando las vistas hay que asignarle a cada componente al que se le va a dar comportamiento (botón, área de texto, etc) un id con un nombre significativo. También debe definirse el id para los Vertical Layout que luego van a contener elementos. El resto de Vertical o Horizontal Layouts no necesitan id. De hecho se recomienda fuertemente no dárselo. Así mismo hay que insertar el componente en el código de la vista pulsando un "+" que aparece en el navegador del Designer. 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. Debe de ponerse el estilo position:absolute en los vertical/vertical layouts mas externos siempre que la vista no este empotrada en otra.

Lo normal es que tengamos que dejar huecos en la vista para colocar componentes que añadiremos después y que son diseñados a parte. En el ejemplo, *Buscar* es un caso de uso separado que lo diseñaremos después y es por eso que hay un hueco a la izquierda donde va a ir colocado. Para ello debemos poner ahí un *Vertical Layout*.

Herencia de las vistas

Ahora hay que heredar de la vista correspondiente en cada clase Java de actor de las de MDS1. Se hereda de las clases .java que ha generado el plugin de MDS2. Por ejemplo,

```
public class Logueado extends VistaLogueado  
public class NoLogueado extends VistaNologueado
```

De esto modo estaremos indirectamente heredando en las clases *Administrador*, *Registrado* y *Perfil* de la clase *VistaLogueado*.

Código de los constructores

Ahora debemos crear un constructor para cada clase que hereda de este modo:

```
public class Logueado extends VistaLogueado{  
    public MainView MainView;  
    public Logueado(MainView MainView) {  
        this.MainView = MainView;  
    }  
  
public class NoLogueado extends VistaNologueado{  
    public MainView MainView;  
    public NoLogueado(MainView MainView) {  
        this.MainView = MainView;  
    }  
}
```

El constructor tiene un parámetro que es el *Main View*. De este modo, tenemos acceso desde el *Logueado* y desde el *NoLogueado* a la ventana principal de nuestra aplicación. A continuación hacemos lo mismo con el resto de actores. En el ejemplo, *Administrador*, *Registrado* y *Perfil*.

```
public class Administrador extends Logueado{  
    public Administrador (noticias.MainView MainView){  
        super(mainview);  
    }  
}
```

El constructor del *Administrador* llama al constructor del *Logueado* a través de *super*. Lo mismo tenemos que hacer con el resto.

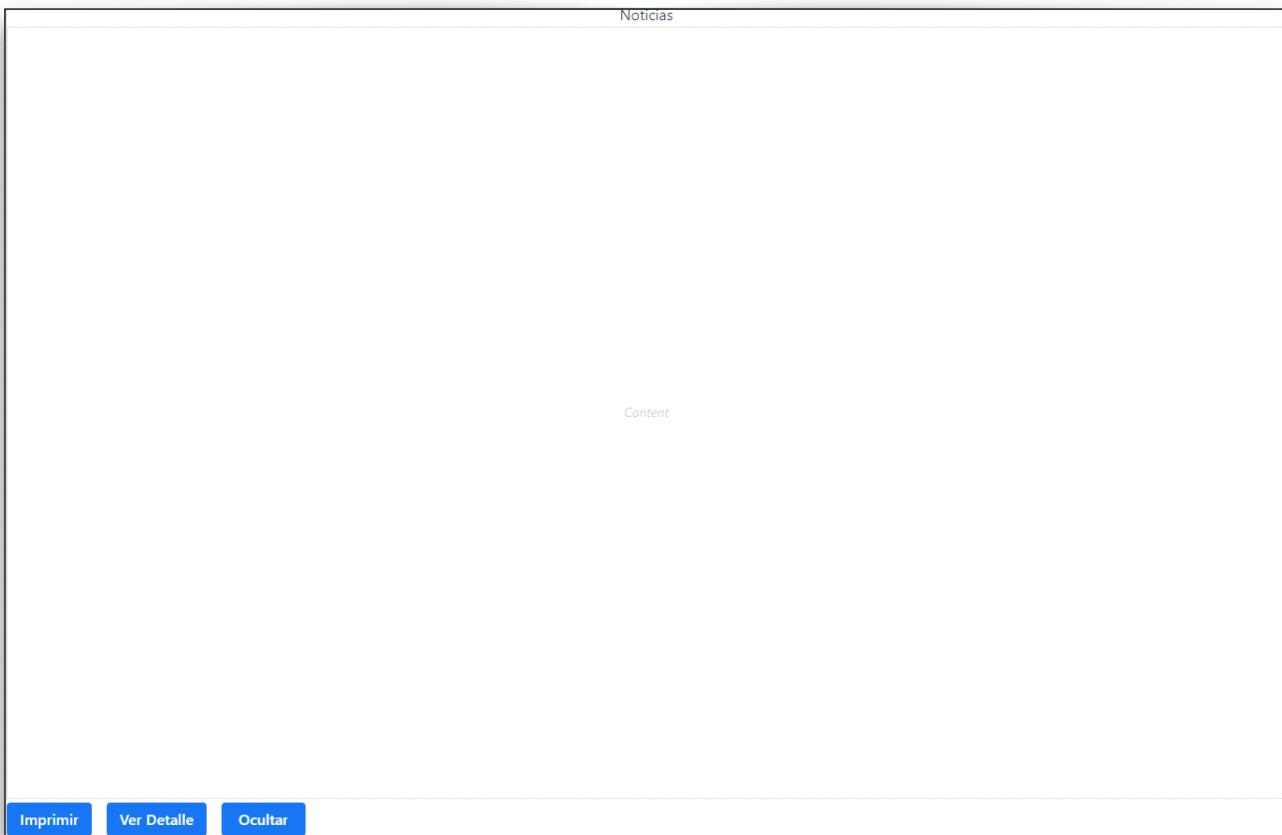
Ahora añadimos código en cada constructor para ocultar los componentes gráficos que no deben aparecer. En el ejemplo, *Ver Actividad* y *Ultimas Noticias* no deben estar en Administrador porque son de *Perfil* y *Registrado*, respectivamente.

```
public Administrador(noticias.MainView MainView) {  
    super(MainView);  
    this.getUltimasnoticias().setVisible(false);  
    this.getVeractividad().setVisible(false);  
}
```

Donde *this.getVerActividad()* y *this.getUltimasNoticias()* son los botones de la vista *Logueado*.

Entrega 5 (Github): Actores.

ACTIVIDAD 4: VISTA DE LAS LISTAS



Diseño 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.

En nuestro ejemplo, deberíamos diseñar una vista para *Noticias* como la de la imagen. Esta vista incluye toda la funcionalidad de *Noticias de Administrador*, *Últimas Noticias* y *Noticias Eliminadas*, o sea, *Imprimir* y *Ver Detalles* que son de *Noticias*, pero también *Ocultar* que es de *Últimas Noticias*. Nótese que la lista está vacía, y solo tiene un *Vertical Layout* con un título para colocar ahí las noticias. Luego se llenará con la vista de *Noticias item*.

Podría ocurrir que alguna o muchas de nuestras listas solo tengan un *Vertical Layout* y que no tenga ningún botón ni componente gráfica. Eso no quiere decir que no haya que crearla como el resto.

Herencia de las vistas

A continuación heredamos de las vistas como en el caso de los actores. Ahora hay que heredar de la vista correspondiente en cada clase Java de actor de las de MDS1. Se hereda

de las clases .java que ha generado el plugin de MDS2 y se añade el constructor. En este caso pasamos por parámetro el contenedor de las noticias, que en este caso es el Logueado. Por ejemplo,

```
public class Noticias extends VistaNoticias {  
    public Vector<Noticias_item> _item = new Vector<Noticias_item>();  
    public Logueado logueado;  
    public Noticias(Logueado logueado) {  
        super();  
        this.logueado = logueado;  
    }
```

Como antes estaremos indirectamente heredando en las clases *Últimas Noticias*, *Noticias Administrador* y *Noticias Eliminadas* de la clase *VistaNoticias*.

Constructoras de las listas

Procedemos a hacer lo mismo que antes pero con las listas, añadiendo constructores ocultando los elementos que no deben aparecer para obtener los casos de uso del diagrama. Por ejemplo,

```
public class NoticiasAdministrador extends Noticias {  
    public Administrador _administrador;  
    public NoticiasAdministrador(Administrador _administrador) {  
        super(_administrador);  
        this._administrador = _administrador;  
        this.getOcultar().setVisible(false);  
    }
```

Obsérvese que hemos aprovechado el *public Administrador _administrador;* que nos ha generado el plugin de MDS1.

Diseño de los items de las listas

Ahora diseñamos las vistas de los elementos que están incluidos en las listas, o sea, los items. Y hacemos lo mismo que antes. Ponemos toda la funcionalidad en el item que está mas arriba en la jerarquía.



Por ejemplo, la vista de *Noticias Item*, debe incluir al igual que en el caso de *Noticias* toda la funcionalidad que necesitan *Noticias de Administrador*, *Últimas Noticias* y *Noticias Eliminadas* que son *Marcar* y *Eliminar*. *Marcar* es de *Últimas Noticias* y *Eliminar* es de *Noticias Administrador*. El diseño es como la imagen.

Constructoras de los items de las listas

Hacemos lo mismo que hemos hecho con las listas, heredando de la vistas, añadiendo constructores y ocultando en los items los elementos que no deben aparecer. Por ejemplo:

```
public class Noticias_item extends VistaNoticiasitem {  
    public Noticias _noticias;  
    public Noticias_item(Noticias _noticias) {  
        super();  
        this._noticias = _noticias;  
    }  
}
```

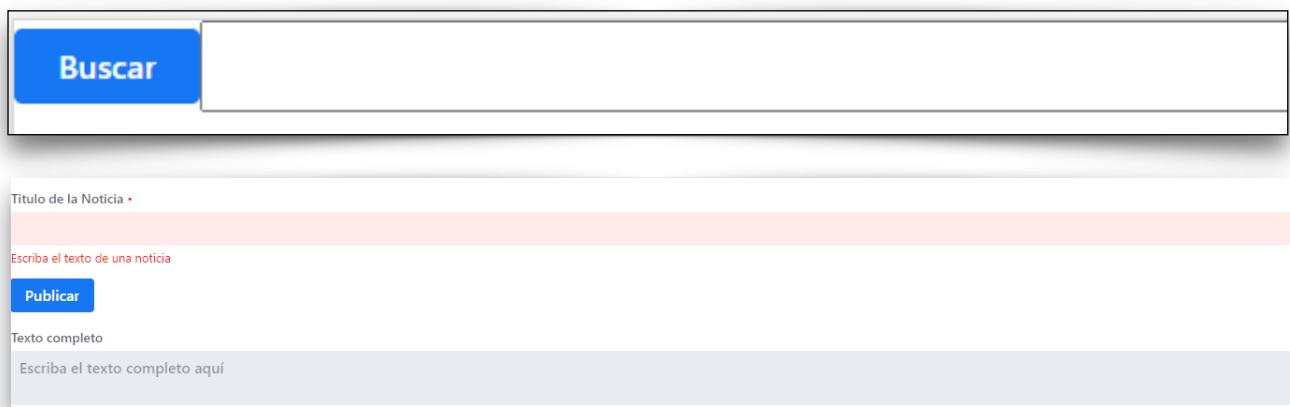
```
public class NoticiasAdministrador_item extends Noticias_item {  
    public NoticiasAdministrador_item(  
        NoticiasAdministrador padre) {  
        super(padre);  
        this.getMarcar().setVisible(false);  
    }  
}
```

Aquí el constructor tiene como parámetro el contenedor, o sea, *Noticias* en este caso. De nuevo reutilizamos el atributo que generó el plugin de MDS1.

Entrega 6 (Github): Listas.

ACTIVIDAD 5: VISTA 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. En el ejemplo, la vista del caso de uso *Buscar* es un botón *Buscar* y un área de texto. También tendríamos que hacer las vistas de *Detalle*, *Ver Actividad* o *Publicar Noticia*. Por ejemplo, *Publicar Noticia* podría ser como la imagen.



The screenshot shows a user interface for publishing a news item. At the top left is a blue button labeled "Buscar". Below it is a large, empty text area with a placeholder "Escriba el título de la Noticia". Underneath that is another text area with a placeholder "Escriba el texto de una noticia". A blue "Publicar" button is positioned above this second text area. Below the second text area is a smaller text area with a placeholder "Escriba el texto completo aquí".

Los casos de uso etiquetados con <<event>> y <<option>> dan lugar normalmente a botones como en el ejemplo *Marcar*, *Eliminar*, *Ocultar* o *Logout*. Pero también hay botones para lanzar desde la vista la correspondiente funcionalidad. Por ejemplo, *Imprimir*, *Ver Detalle*, *Noticias Eliminadas*, *Noticias de Administrador*, *Publicar Noticia*, *Ver Actividad* y *Últimas Noticias*. Estos últimos botones son especiales porque permiten la navegación por la aplicación. En el caso de casos de uso <<business>> estos han dado lugar a un método que luego implementaremos en código. Por último los actores <<external>> han dado lugar a clases Java normales en la exportación con Visual Paradigm.

Entrega 7 (Github): Otros casos de uso.

ACTIVIDAD 6: ENSAMBLADO DE COMPONENTES ESTÁTICOS

Ahora nos disponemos a ensamblar los componentes que hemos construido y a añadir 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, son estáticos, y que no cambian cuando ejecutamos la aplicación. Y por otro lado, b) aquellos componentes que son dinámicos y que se van modificando según se va ejecutando la aplicación.

Por ejemplo, el caso de uso *Buscar* es fijo y aparece siempre en la ventana de los usuarios logueados. Sin embargo, la vista *Publicar Noticia* en *Noticias* no está fijo sino que aparece cuando pulsamos el botón que lo abre. Otro ejemplo de componente dinámica es la lista de las noticias que va creciendo según se van añadiendo noticias.

Lo que hacemos ahora es añadir las componentes estáticas. Por ejemplo, *Buscar* hay que añadirlo a la vista de *Logueado* y se ha hecho aparte. Se ha de ensamblar para que quede como en la imagen. Para ello hay que escribir, por ejemplo, el siguiente código en el constructor que llama el método que nos ha generado el plugin de MDS1.

```
public class Logueado extends VistaLogueado {  
    public Buscar _buscar;  
    public MainView MainView;  
    public Logueado(MainView MainView) {  
        this.MainView = MainView;  
        _buscar = new Buscar(this);  
        Buscar();  
    }  
    public void Buscar() {  
        this.getBuscarLayout().as(VerticalLayout.class)  
            .add(_buscar);  
    }  
}
```

Cuando hicimos la vista de *Logueado* pusimos un *Vertical Layout* para colocar el caso de uso de *Buscar* y que hemos diseñado en la actividad anterior. Ahora es el momento añadirlo, creando el objeto y colocándolo en su sitio. Nótese que de este modo *Buscar* estará disponible en todos los usuarios por la herencia.

Entrega 8 (Github): Ensamblado de Componentes Estáticos

ACTIVIDAD 7: ENSAMBLADO DE COMPONENTES DINÁMICOS

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

Por ejemplo, en el *Administrador* podemos añadir *Noticias Eliminadas*, *Noticias de Administrador* y *Publicar Noticia* de la siguiente manera. Supongamos que el *Vertical Layout* del que dispone el *Administrador* se llama *contenido*, entonces haremos lo siguiente, por ejemplo, para *Noticias Eliminadas*, y lo mismo para el resto.

```
public class Administrador extends Logueado {  
    public NoticiasEliminadas _noticiasEliminadas;  
    public NoticiasAdministrador _noticiasAdministrador;  
    public PublicarNoticia _publicarNoticia;  
    public VerRegistrados _verRegistrados;  
    public Administrador(noticias.MainView MainView) {  
        super(MainView);  
        this.getUltimasnoticias().setVisible(false);  
        this.getVeractividad().setVisible(false);  
  
        this.getNoticiaseliminadas()  
            .addClickListener(event->NoticiasEliminadas());  
        this.getNoticiasadministrador()  
            .addClickListener(event->NoticiasAdministrador());  
        this.getPublicarnoticia()  
            .addClickListener(event->PublicarNoticia());  
        this.getVerregistrados()  
            .addClickListener(event->VerRegistrados());  
    }  
    public void NoticiasEliminadas() {  
        this.getContenido().as(VerticalLayout.class).removeAll();  
        _noticiasEliminadas = new NoticiasEliminadas(this);  
        this.getContenido().as(VerticalLayout.class)  
            .add(_noticiasEliminadas);  
    }  
    public void NoticiasAdministrador() {  
        this.getContenido().as(VerticalLayout.class).removeAll();  
        _noticiasAdministrador = new NoticiasAdministrador(this);  
        this.getContenido().as(VerticalLayout.class)  
            .add(_noticiasAdministrador);  
    }  
    public void PublicarNoticia() {  
        this.getContenido().as(VerticalLayout.class).removeAll();  
        _publicarNoticia = new PublicarNoticia(this);  
    }  
}
```

```

        this.getContenido().as(VerticalLayout.class)
            .add(_publicarNoticia);
    }
    public void VerRegistrados() {
        this.getContenido().as(VerticalLayout.class).removeAll();
        _verRegistrados = new VerRegistrados(this);
        this.getContenido().as(VerticalLayout.class)
            .add(_verRegistrados);
    }
}

```

Esto coloca la vista de *Noticias Eliminadas* en el área de contenido del *Administrador* cuando se pulsa el botón correspondiente. El método *NoticiasEliminadas()* lo tenemos disponible porque lo generó el plugin de MDS1. También tenemos el objeto correspondiente que lo generamos automáticamente. Lo mismo tendríamos que hacer con *Noticias de Administrador* y *Publicar Noticia*. Lo mismo podríamos hacer con *VerDetalle* e *Imprimir* que se abren cuando pulsamos el botón en la lista de *Noticias*.

```

public class Noticias extends VistaNoticias {
    public Vector<Noticias_item> _item =
        new Vector<Noticias_item>();
    public VerDetalle _verDetalle;
    public printer _printer;
    public Logueado logueado;
    public Noticias(Logueado logueado) {
        super();
        this.logueado = logueado;
        this.getVerdetalle().addClickListener(
            event->VerDetalle());
    }

    public void VerDetalle() {
        _verDetalle = new VerDetalle(this);
        logueado.getContenido().as(VerticalLayout.class)
            .remove(this);
        logueado.getContenido().as(VerticalLayout.class)
            .add(_verDetalle);
    }
}

```

Nótese a que al construirlo de esta manera, tenemos los botones de *Ver Detalle* e *Imprimir* disponibles también en *Noticias de Administrador*, *Últimas Noticias* y *Noticias Eliminadas*. Podríamos hacer lo mismo con el botón *Ocultar* en Últimas Noticias.

Un caso distinto, aunque parecido, es el *Publicar Noticia* que para ensamblarlo tenemos que hacer dos cosas. Por un lado, poner un evento como los anteriores para que desde *Administrador* se pueda abrir, pero a su vez cuando se pulsa en el botón Publicar Noticia debería cerrarse y volver a Administrador. Para ello, además de añadir el evento en

Administrador para abrir esta ventana, añadimos un evento en *Publicar Noticia* para cerrarlo. Nótese que el método *Publicar()* hay que añadirlo porque no es un <>event<> del diagrama de casos de uso y por tanto no sé añadió con el plugin de MDS1.

```
public class PublicarNoticia extends VistaPublicarnoticia {
    public Administrador _administrador;
    public PublicarNoticia(Administrador _administrador) {
        super();
        this._administrador = _administrador;
        this.getPublicar().addClickListener(event->Publicar());
    }

    public void Publicar() {
        this._administrador.getContenido()
            .as(VerticalLayout.class).remove(this);
    }
}
```

Como podemos ver, necesitamos cerrar la ventana que hemos abierto. Para ello, cuando se pulsa el botón de *Publicar Noticia* se recurre al padre (que es el *Administrador*) y quitamos el propio objeto (*this*).

Ahora vamos con el caso de los elementos (items) de las listas. Al contrario que pasa con, por ejemplo, *Ver Detalle* e *Imprimir*, necesitaremos las noticias con datos para verlos realmente funcionar pero eso no quiere decir que no podamos escribir un poco de código. Por ejemplo, en las noticias tenemos los items de la imagen. Podemos en este momento hacer que el botón *Eliminar* elimine el elemento de la lista.

```
public class NoticiasAdministrador_item extends Noticias_item {
    public NoticiasAdministrador_item(Noticias padre) {
        super(padre);
        this.getMarcar().setVisible(false);
        this.getEliminar().addClickListener(event->Eliminar());
    }

    public void Eliminar() {
        this._noticias.getList().as(VerticalLayout.class)
            .remove(this);
        this._noticias._item.remove(this);
    }
}
```

Donde *getList()* es el *Vertical Layout* donde se almacena cada *Noticia Item* y que se elimina de la lista al pulsar el botón eliminar. Nótese que estoy accediendo al *padre* y a la lista que contiene las noticias con *this._noticias.getList()*. Una vez hecho esto lo probamos añadiendo un elemento a la lista (con código) y pulsando *Eliminar*. Podríamos hacer también el botón *Marcar* del item de *Noticias*.

Nos queda por último escribir código en el *MainView* que también dinámico. El proyecto ejemplo de *Vaadin* tiene en el *MainView* un código de ejemplo. Este código de ejemplo lo

podemos borrar. También podemos eliminar *GreetService*. Ahora lo que vamos a hacer es crear el programa principal de nuestra aplicación que consiste en la creación del objeto *NoLogueado*. También debemos declarar objetos *Administrador*, *Registrado* y *Perfil* que se usarán para almacenar la sesión de cada tipo de actor.

```
public class MainView extends VerticalLayout {  
    public Administrador admin;  
    public Registrado registrado;  
    public Perfil perfil;  
    public NoLogueado nologueado;  
    public MainView() {  
        nologueado = new NoLogueado(this);  
        add(nologueado);  
    }  
}
```

Dado que hemos añadido *nologueado* al *Vertical Layout* del *Main View* nos aparecerá la vista del *nologueado* al arrancar la aplicación. En caso de que el *login* sea exitoso la propia ventana de *login* cambiará del *nologueado* al correspondiente usuario. Y lo mismo ocurre cuando se hace *logout* que pasamos del usuario que esté en ese momento al *nologueado*. Por ejemplo, el *logout* del *logueado* se implementa así:

```
public class Logueado extends VistaLogueado {  
    public Buscar _buscar;  
    public MainView MainView;  
    public Logueado(MainView MainView) {  
        this.MainView = MainView;  
        _buscar = new Buscar(this);  
        Buscar();  
        this.getLogout().addClickListener(event->Logout());  
    }  
    public void Logout() {  
        MainView.remove(this);  
        MainView.add(MainView.nologueado);  
    }  
}
```

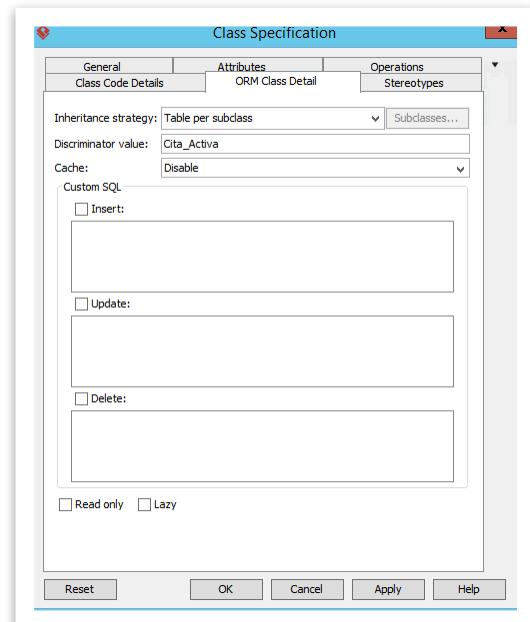
Entrega 9 (Github): [Ensamblado de Componentes Dinámicos](#)

ACTIVIDAD 8: GENERACIÓN DE CÓDIGO ORM

Preparación

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.

- i) A continuación comprobamos que los nombres de las *ORMPersistable* no contienen espacios en blanco, tildes ni eñes. Dado que ese nombre se va a usar como nombre de la clase es importante eliminarlos. 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.
- ii) También hay que comprobar que no falta ninguna de las *cardinalidades* (1, 0..1, *, 0..*, etc). 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, ponemos un nuevo atributo de tipo *Int* y lo usamos como clave de las asociaciones si hiciera falta.
- iv) 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*.
- v) Finalmente, 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 este más arriba y eliminar si las hubiera las claves de las tablas que heredan, actualizando las claves de las asociaciones si las hubiera. 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). Ver imagen.

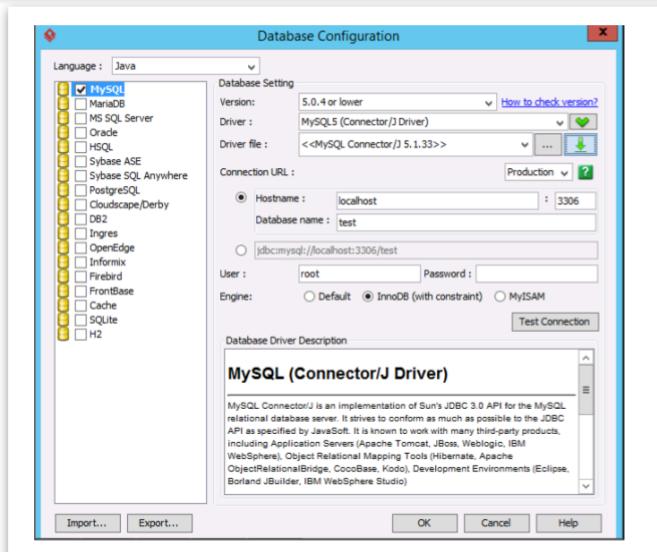
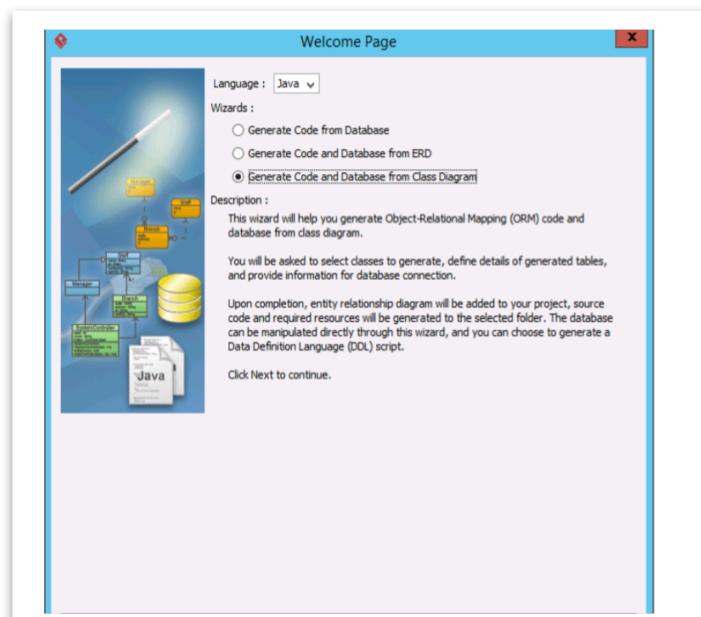
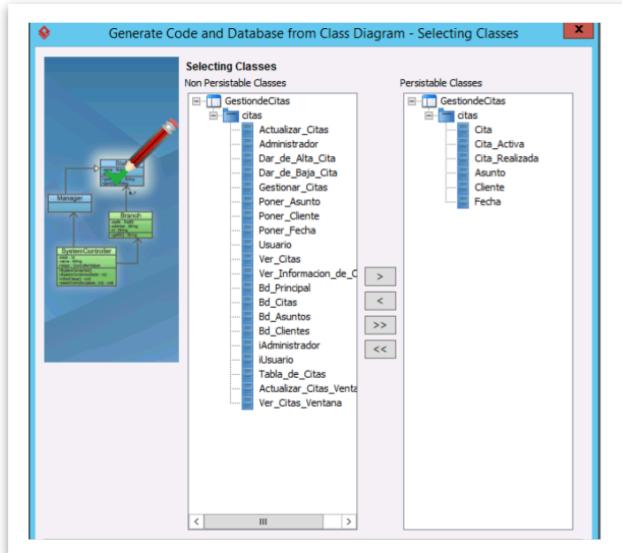
Arranque de MySQL

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

Generación de código ORM

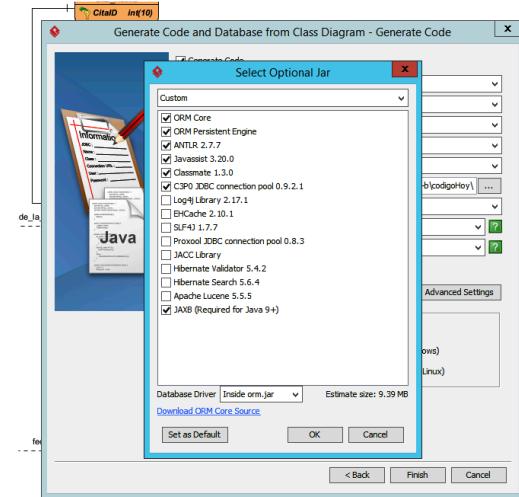
Ahora ya podemos generar el código ORM para la base de datos. Para ello:

- 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 iii) anterior. En las 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



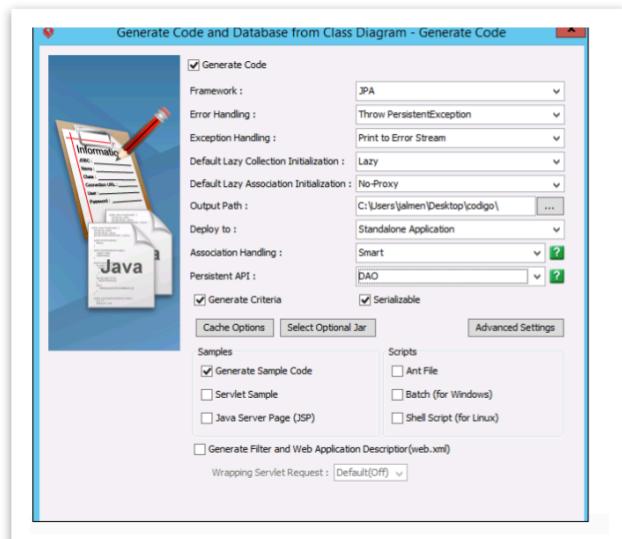
localhost 3306. Además clicamos en *InnoDB*. Debemos probar la conexión.

- 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*. En esta pantalla hay que eliminar *Log4j* en *Select Opcional Jar*. 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 donde hemos generado el código ORM. Debemos añadir este fichero *orm.jar* al build path de Eclipse.



Importante: Si se ha encontrado algún error en el proceso de generación de código hay que volver a generarla otra vez. Se recomienda borrar el diagrama entidad-relación cuando se vuelve a generar el código.

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 cree nuestra base de datos. Debemos tener arrancando el MySQL

de XAMPP. Esto nos habrá creado las tablas que las podemos ver yéndonos al administrador de MySQL. 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 añadir la siguiente línea de código en el programa *Application.java* de *Vaadin*:

```
public class Application extends SpringBootServletInitializer {  
    public static void main(String[] args) {  
        System.setProperty(  
            "spring.devtools.restart.enabled", "false");  
        SpringApplication.run(Application.class, args);  
    }  
}
```

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.

Actualización del Constructor de los actores

Por ejemplo, el constructor de Logueado que escribimos anteriormente podemos actualizarlo de la siguiente manera:

```
public class Logueado extends VistaLogueado{
    basededatos.Logueado logueado;
    public Logueado (MainView mainview,
                     basededatos.Logueado logueado){
        this.logueado = logueado;
    }
}
```

En el que añadimos un parámetro adicional al constructor de Logueado que es el objeto Logueado de la base de datos generado por ORM. Es importante guardarla en el objeto creado. Lo mismo podemos hacer con Administrador:

```
public class Administrador extends Logueado{
    public Administrador (MainView mainview,
                          basededatos.Administrador administrador){
        super(mainview,administrador);
    }
}
```

Y para Registrado:

```
public class Registrado extends Logueado{
    public Registrado (MainView mainview,
                       basededatos.Registrado registrado){
        super(mainview,registrado);
    }
}
```

En lugar de hacer un constructor para cada tipo de usuario.

El código del constructor debe coger los datos del objeto y rellenar la vista con sus datos. Por ejemplo, supongamos que en la vista del Logueado mostramos su nombre, su foto y el número de seguidores. En ese caso sería:

```
public class Logueado extends VistaLogueado{
    public Logueado (MainView mainview,
                     basededatos.Logueado logueado){

        this.getNombre().setText(logueado.getNombre());
        this.getFoto().setText(logueado.getFoto());
        this.getSeguidores().setText(logueado.getSeguidores());

    }
}
```

De momento si necesitamos pasar al constructor el valor del objeto creamos un objeto falso.

Actualización del Constructor de los items de las listas

Lo mismo podemos hacer con las listas y los items de las listas. Por ejemplo:

```
public class NoticiasItem extends VistaNoticiasItem{
basededatos.Noticia noticia;
public NoticiasItem (Noticias _noticias,
                     basededatos.Noticia noticia){
    this.noticia = noticia;
    this.getTexto().setText(noticia.getTexto());
}
}
```

De este modo, pasamos el objeto *Noticia* de la base de datos a la vista del item de *Noticias*.

Actualización del Constructor de otros casos de uso

Con el resto de casos de uso hacemos lo mismo, siempre y cuando el caso de uso muestre datos de la base de datos por pantalla.

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 un elemento de cada *ORMPersistable*
- E. Añadir o eliminar una relación entre dos elementos de *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, si se carga a un usuario tendríamos un atributo que almacena los usuarios que le siguen o 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, botones o cualquier componente o contenedor gráfico que necesite o envíe datos, de modo que se pueda cargar, crear o modificar datos de la base de datos. 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.

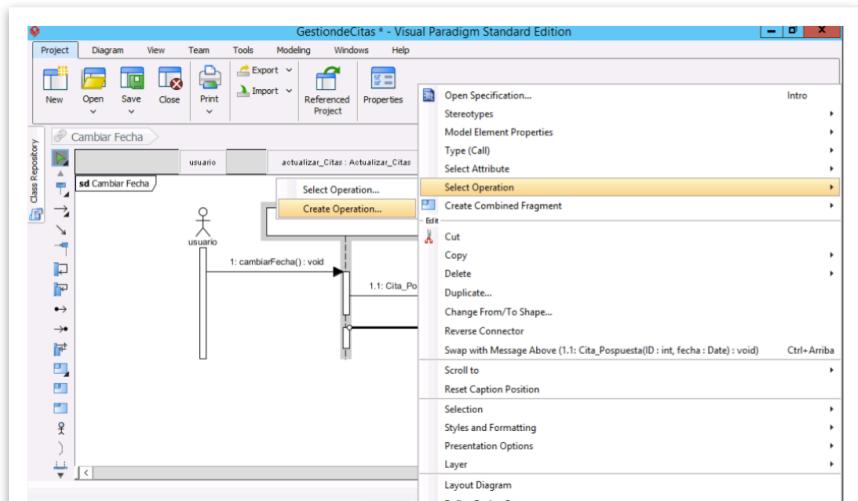
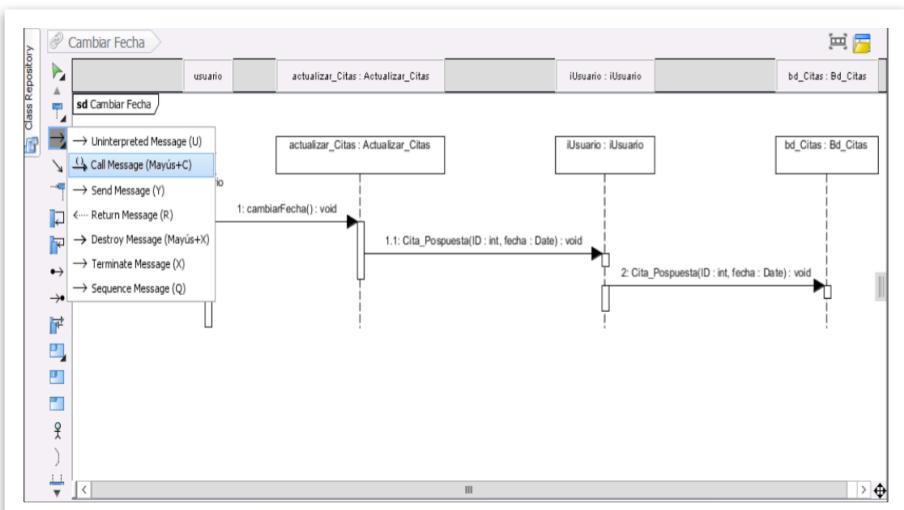
El diagrama de secuencia siempre incluye:

- El actor que pulsa el botón (o componente gráfico: layout, grid, etc), excepto sí se trata de cargar datos de un componente gráfico y el actor no interacciona con la interfaz.
- La clase Java del interfaz de usuario donde se encuentra el listener del botón o componente gráfico.
- El interfaz de la base de datos que corresponde a ese actor. En caso de no aparecer el actor en el diagrama (cargar datos), se incluye el actor que está visualizando la interfaz de usuario.
- Una o varias *ORMPersistable*. Al menos una. Son aquellas que participan en la ejecución.
- Los actores (*<>external<>*) involucrados, si los hubiera.

Creación de los diagramas de secuencia

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

ii) La primera flecha / operación del diagrama de secuencia es siempre una método que es invocado en la interfaz de usuario y, por tanto, es un método sin parámetros que representa la tarea a ejecutar: "guardar_datos()", "cargar_datos()", "borrar_usuario()", "buscar_usuario()" etc. La fecha (*Call message*) va desde la línea de tiempo del actor a la línea de tiempo del interfaz de



usuario donde está el listener. En caso de no aparecer el actor en el diagrama la flecha es recursiva sobre la línea de tiempo del interfaz de usuario (no estará el listener en el código). Este método ya estará dado de alta por el plugin si aparece como `<<event>>` en el diagrama de casos de uso. Si no es así lo añadiremos manualmente.

- iii) El resto de flechas/operaciones (también *Call message*) conectan la línea de tiempo del interfaz de usuario con la interfaz de la base de datos principal, o la línea de tiempo del interfaz de base datos principal con la línea de tiempo de las *ORMPersistable*.
- iv) Estas operaciones suelen tener uno o varios parámetros, excepto aquellas que cargan datos que no suelen tener parámetros. Los parámetros/valores de retorno puede ser IDs (de tipo entero) o bien *Booleanos*, *Enteros* o *Strings*. Si queremos pasar como parámetro o devolver una colección de elementos en alguna operación utilizaremos el List de Java (por ejemplo, `List<Usuario>`, `List<UsuarioRegistrado>`) o un vector (por ejemplo, `Usuario[]`, `UsuarioRegistrado[]`). El ORM recupera datos a través de un *ID* (entero) y devuelve los datos en una lista o un vector. Hay que pasar necesariamente todos los parámetros para realizar una interacción con la base de datos. Normalmente involucra pasar por parámetros los *IDs* de los objetos involucrados.

Implementación de las interfaces

Para llenar la base de datos principal con operaciones hay que implementar los interfaces. Esto se hace en Visual Paradigm, colocándose sobre la base de datos principal y con el botón derecho: Related Elements->Realice All Interfaces.

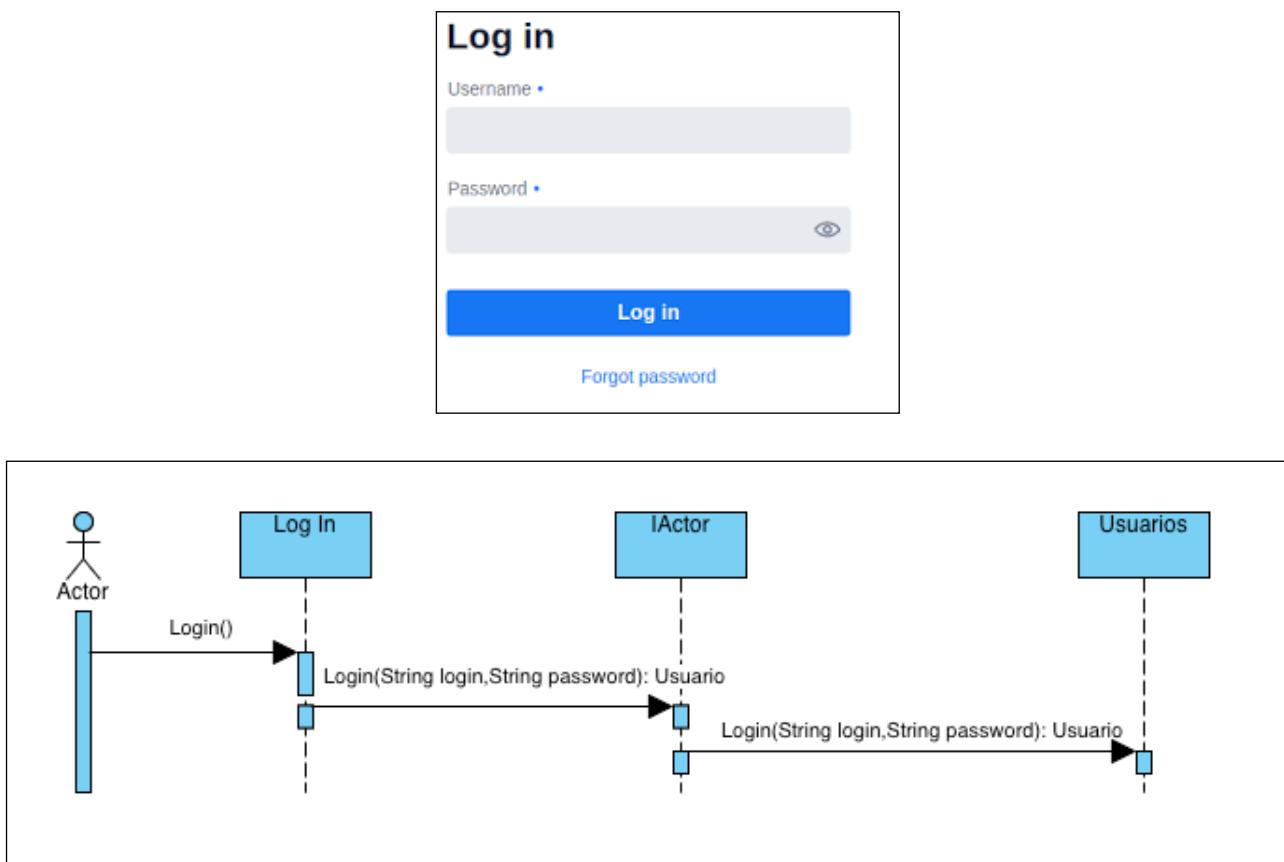
Exportar los métodos de los diagramas de secuencia

Lo siguiente que vamos a hacer es exportar el código de las bases de datos. Volvemos a ir a *Tools->Code->Instant Generator*, como hemos hecho en las interfaces de usuario, pero ahora en *Model Elements* seleccionamos las clases de la base de datos principal, los interfaces y los *ORMComponent*. No se eligen ni las clases de la interfaz de usuario (que ya exportamos anteriormente), ni las *ORM Persistable* (ya generamos con ellas el código ORM).

Entrega 12 (Aula Virtual): Documentación Parcial

Ejemplo 1

Diagrama de secuencia del botón de login



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) }}}
```

Podría ocurrir que el diagrama de secuencia contuviera un *alt* para distinguir casos. Por ejemplo, en el caso del Login podríamos usar varias ORMPersistable, la de Registrado y la de Administrador. En ese caso, el diagrama de secuencia es más complejo y el código también.

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@olson.com
Gunner	Karlsen	gunner@karlsen.com
Leland	Harris	leland@harris.com
Danielle	Watson	danielle@watson.com
Makena	Smith	makena@smith.com
Quinn	Hansson	quinn@hansson.com

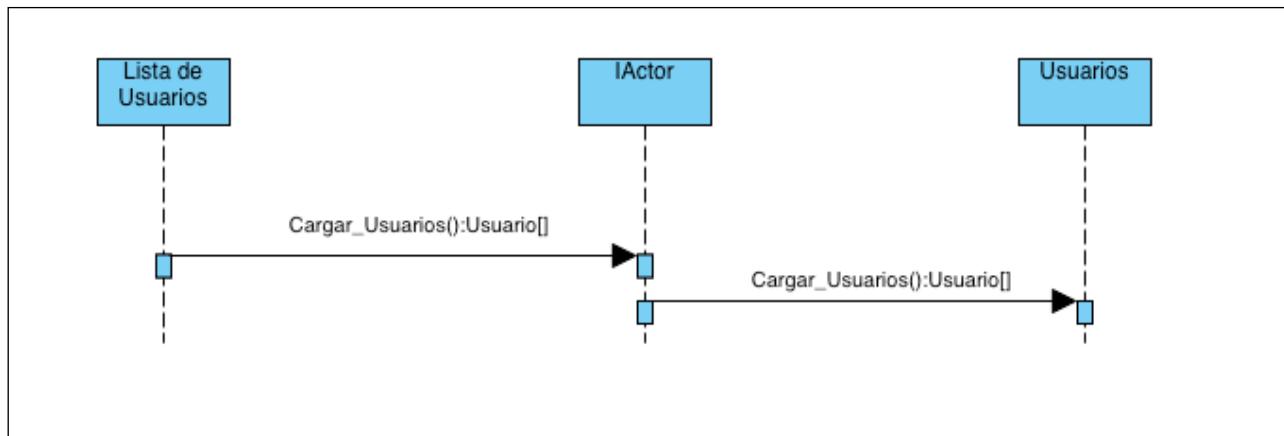
filter by name...xAdd new customer

Name: Email:

Email: Status:

Birthday:

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

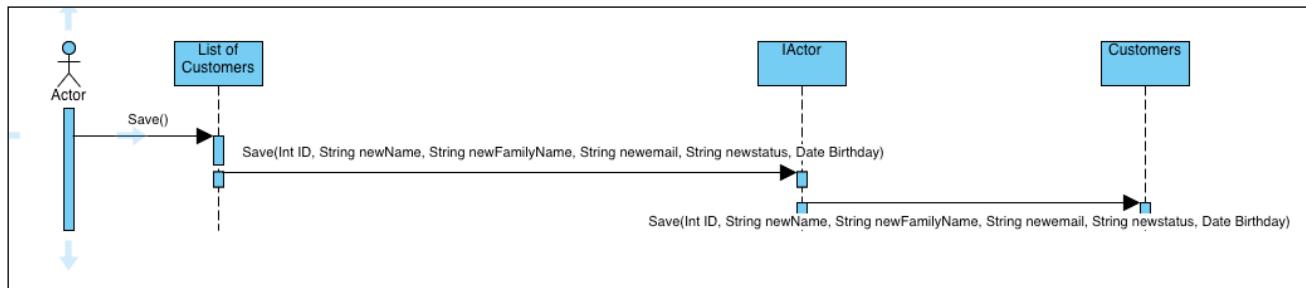
Nótese que cargamos los usuarios de la base de datos, creamos la vista de cada uno de ellos y los añadimos al contenido de la lista.

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



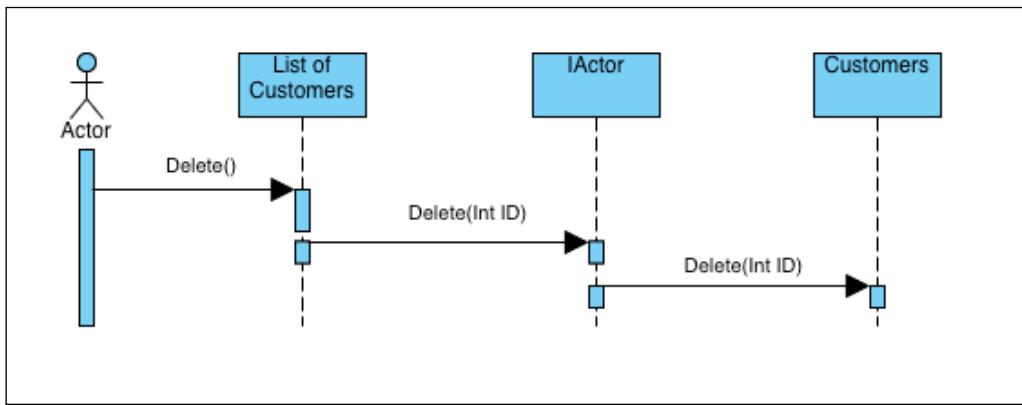
Este diagrama de secuencia se corresponde con este código:

```
public class ListOfCustomers {
    IActor iactor = new BD_Principal();
    public ListOfCustomers(Padre padre){
        this.getSaveButton().addClickListener(event -> Save())
    }
    public void Save(){
        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 void Save(String Name, String FamilyName,...){
        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());}}}
```

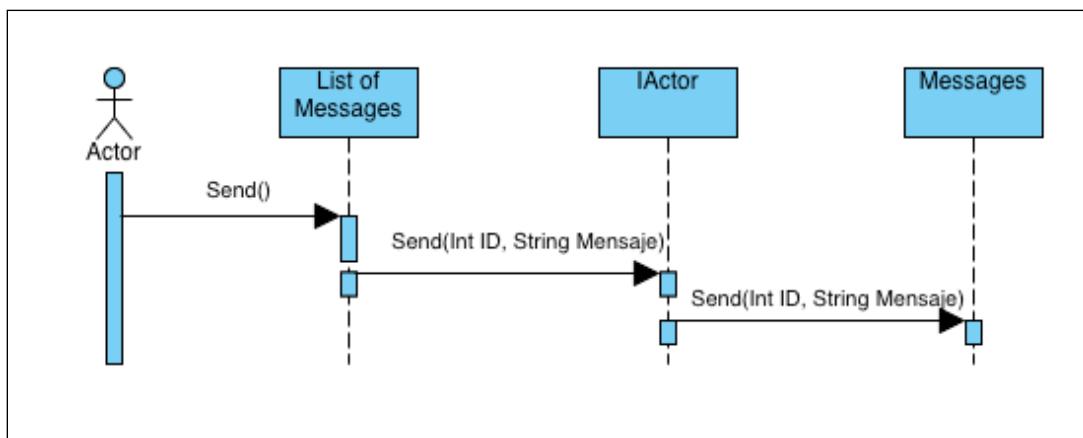
A su vez en la base de datos principal (que implementa el IActor):

```
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...logueado.getID(),
                    this.getMessage().getText());}}
```

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)}}
```

Ejemplo 4

Signup form

First name • Last name •

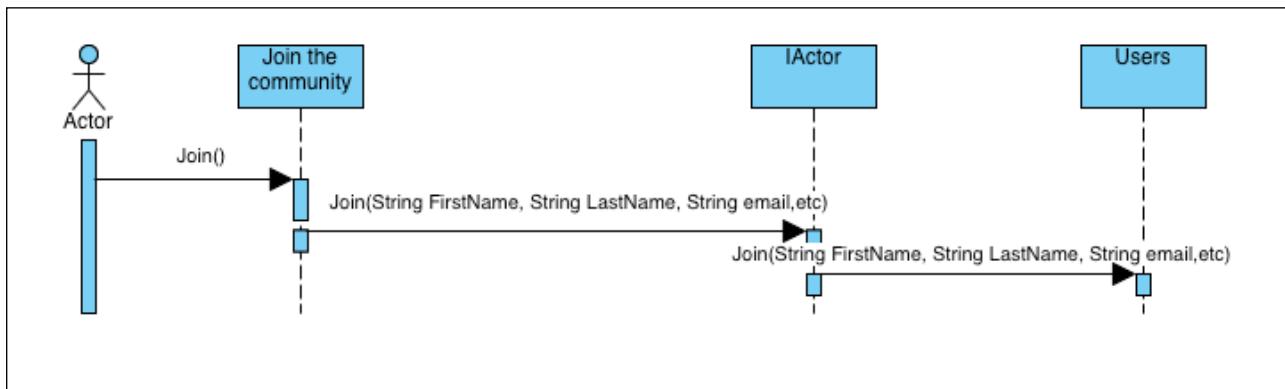
Email •

Password • Confirm password •

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 void Join(){
        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 void Join(String FirstName, String LastName,...){
        bd_users.Join(FirstName,LastName,...)}}}
```

Ejemplo 5

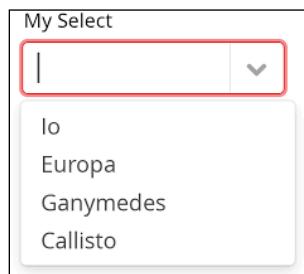
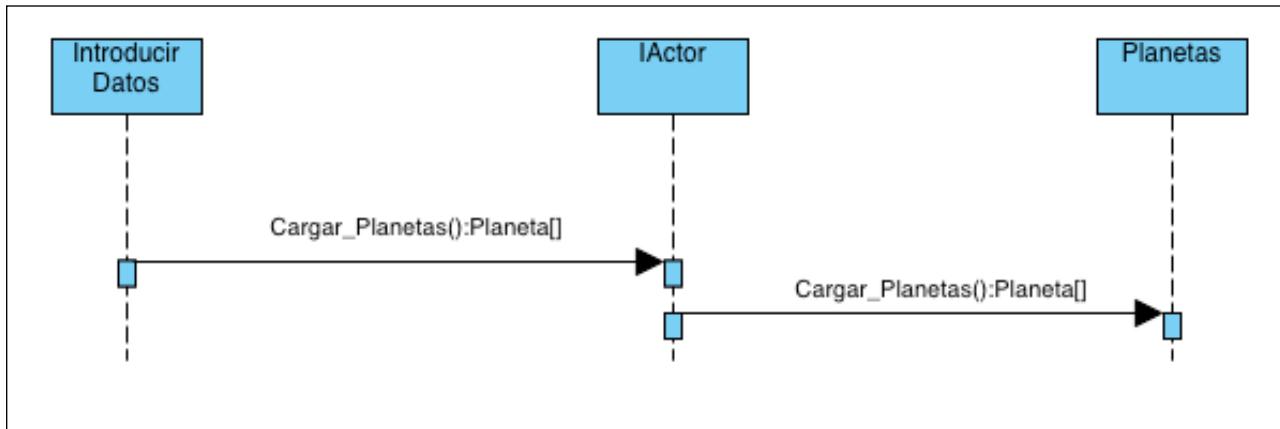


Diagrama de secuencia de cargar planetas



Este diagrama de secuencia se corresponde con este código:

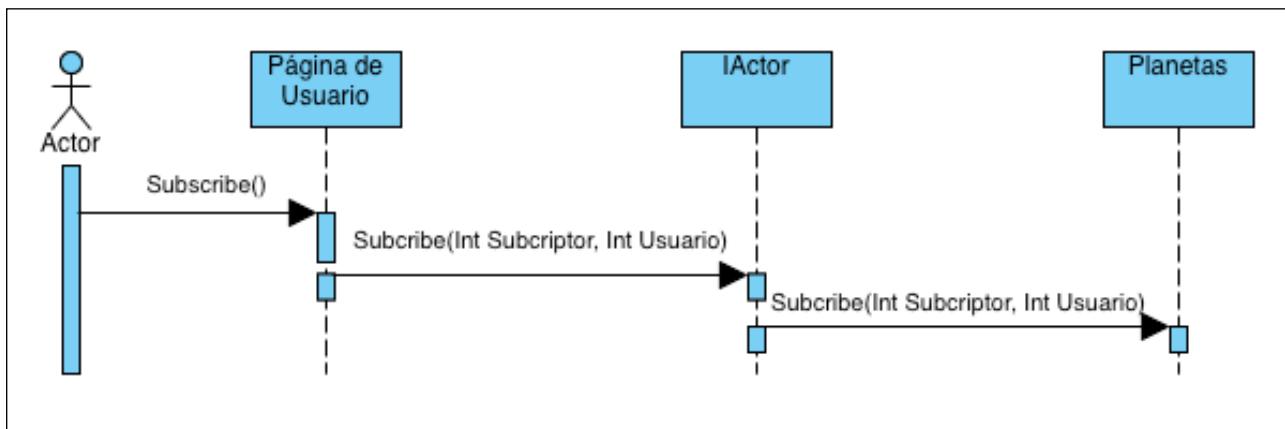
```
public class IntroducirDatos {
    IActor iactor = new BD_Principal();
    public Planeta[] planetas;
    public IntroducirDatos(Padre padre){
        planetas = Cargar_Planetas();
        this.getComboBox().setItems(planetas);}
    public Planeta[] Cargar_Planetas(){
        return iactor.Cargar_Planetas();}}}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

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:

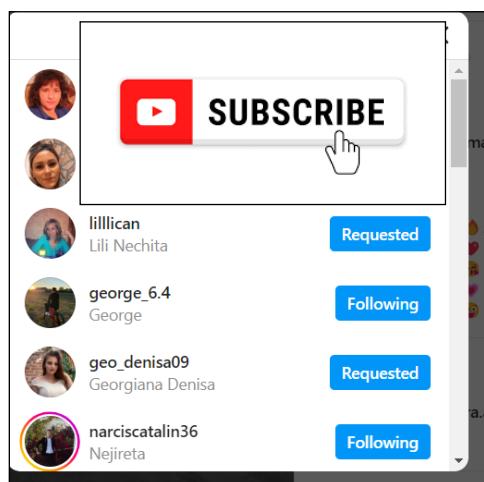
```
public class PaginaDeUsuario {
    IActor iactor = new BD_Principal();
    public PaginaDeUsuario(Padre padre){
        this.getSubscribeButton()
            .addClickListener(event -> Subscribe());
    }
    public void Subscribe(){
        iactor.Subscribe(padre...logueado.getID(),
           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

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)}}}
```

Ejemplo 7

No hace falta hacer diagrama de secuencia porque el usuario o el mensaje ya está cargado



Ejemplo 8

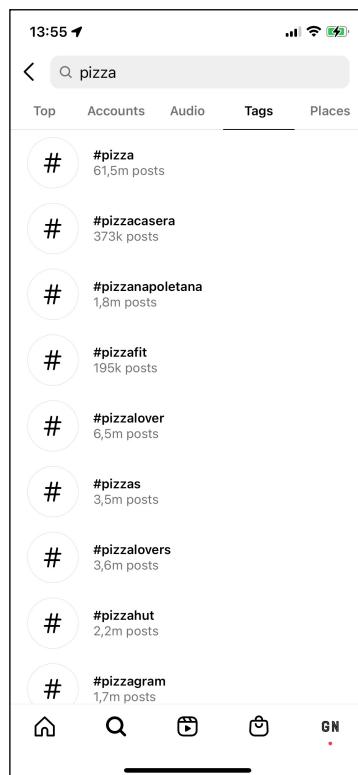
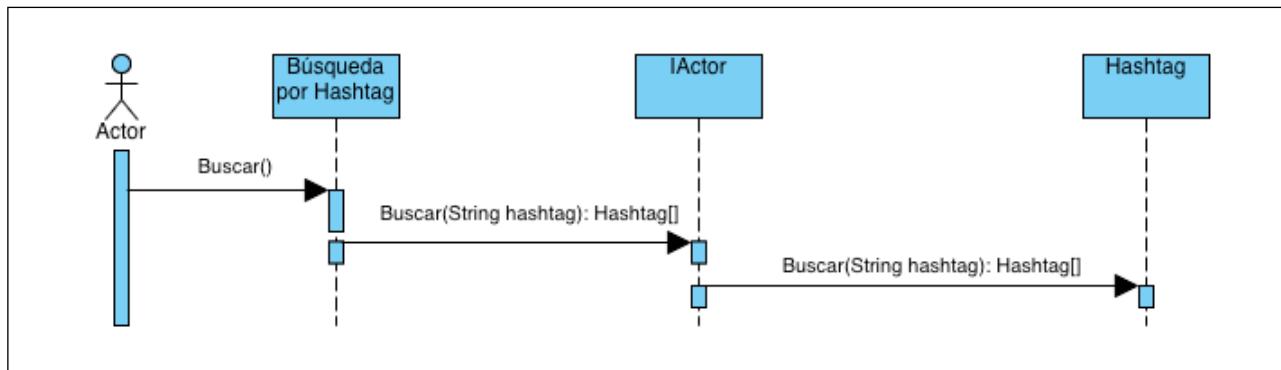


Diagrama de secuencia de buscar por hashtag



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().setOnClickListener(event ->{
            hashtags =Buscar();
            for (Int i=0;i++,i<hashtags.size){
                HashtagsItem hi = new HashtagsItem(this,hashtags[i]);
                this.getContenido().as(VerticalLayout.class)
```

```
        .add(hi);}}} }  
    public Hashtag[] Buscar(){  
        return iactor.Buscar(this.getHashtag().getText());}}}
```

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)} }
```

Ejemplo 9

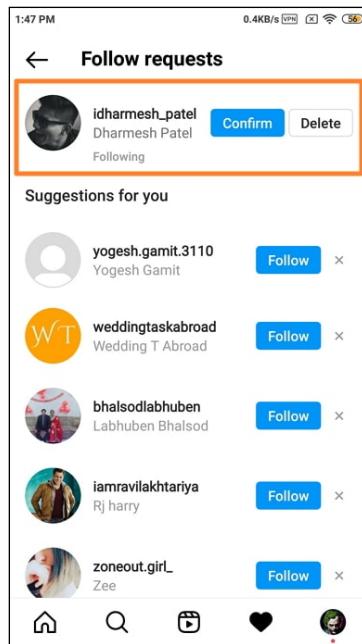
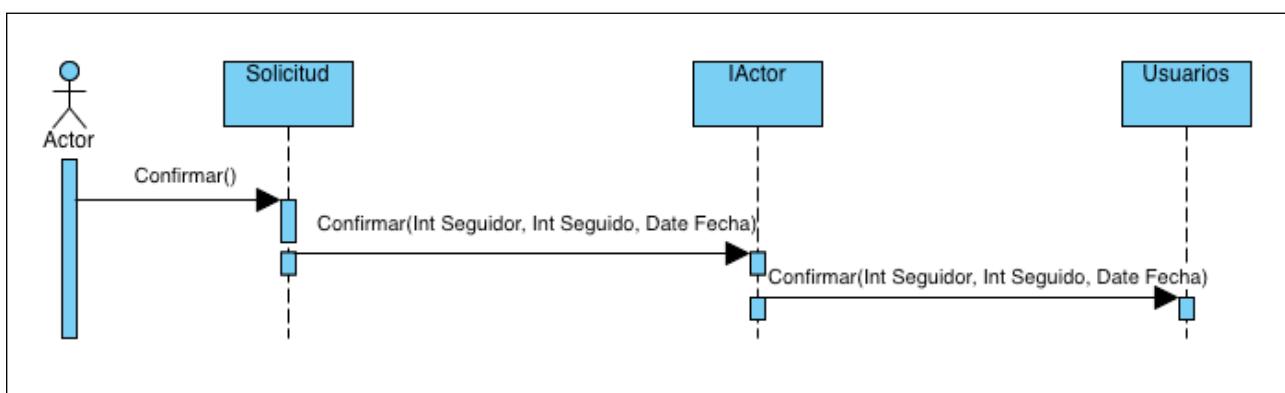


Diagrama de secuencia de confirmar seguimiento



Este diagrama de secuencia se corresponde con este código:

```
public class Solicitud {
    IActor iactor = new BD_Principal();
    public Solicitud(Padre padre){
        this.getConfirmarButton()
            .addClickListener(event -> Confirmar())
    }
    public void Confirmar(){
        iactor.Confirmar(padre..logueado.getID(),
        this.getUsuario().getId(),
        LocalDateTime.now());}}
```

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 Confirmar(Int Seguidor, Int Seguido, Date Fecha){
        bd_usuarios.Confirmar(Seguidor,Seguido,Fecha)} }
```

ACTIVIDAD 11: IMPLEMENTACIÓN DE ORM COMPONENTS

Ahora vamos a implementar los ORMCcomponents. Mostramos a continuación ejemplos diversos del tipo de operaciones que hay que implementar.

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){.....}}

```

Entrega 13 (Aula Virtual): Documentación Final