MEMORIA

Proyecto Final ITV

ITC
Kevin, Jia, Iván

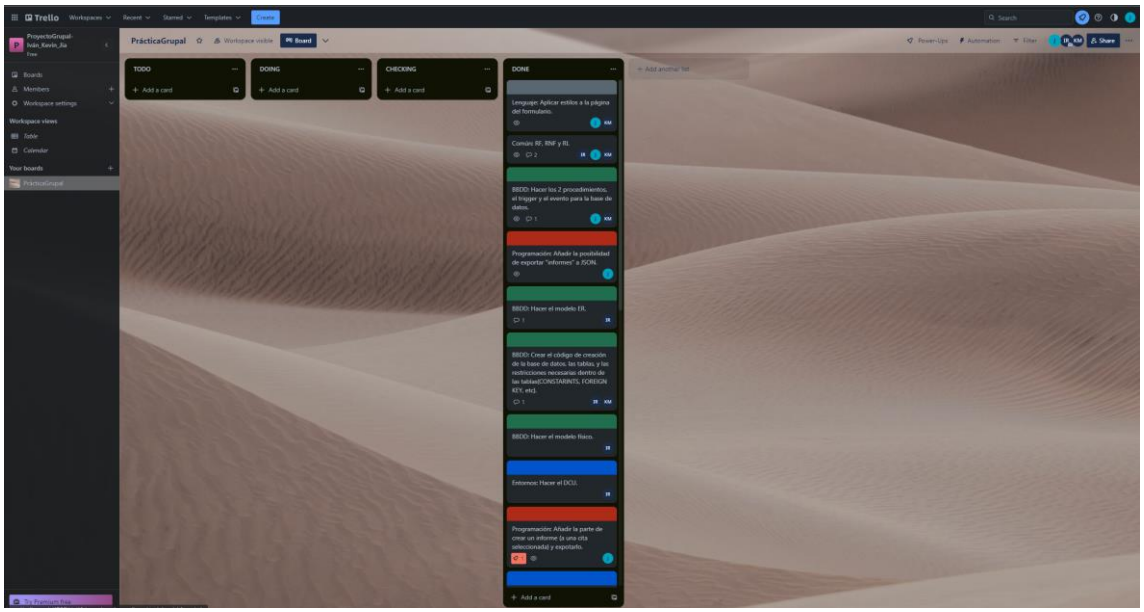
Tabla de contenido

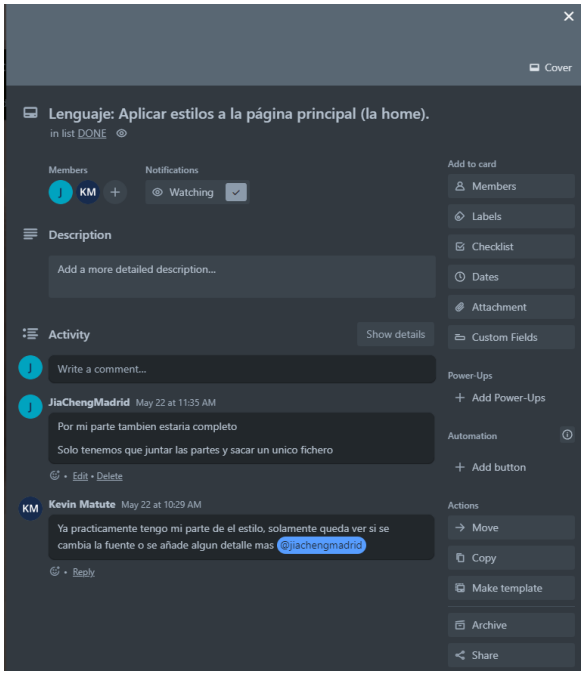
MEMORIA

COMUN

- planificación y gestión de tareas

Para este apartado se ha utilizado trello, github y git-flow. Trello ha servido para establecer las tareas a hacer de todo el proyecto y para repartirlas. Los tres del grupo estamos como integrantes del tablero del proyecto y cada uno se añade en las tareas que vaya a hacer y moverá las tarjetas cuando se haya avanzado hasta el siguiente apartado. Tenemos 4 apartados, todo, doing, checking y done. También se han ido poniendo comentarios en las tarjetas para dar información sobre la tarea al resto del grupo.





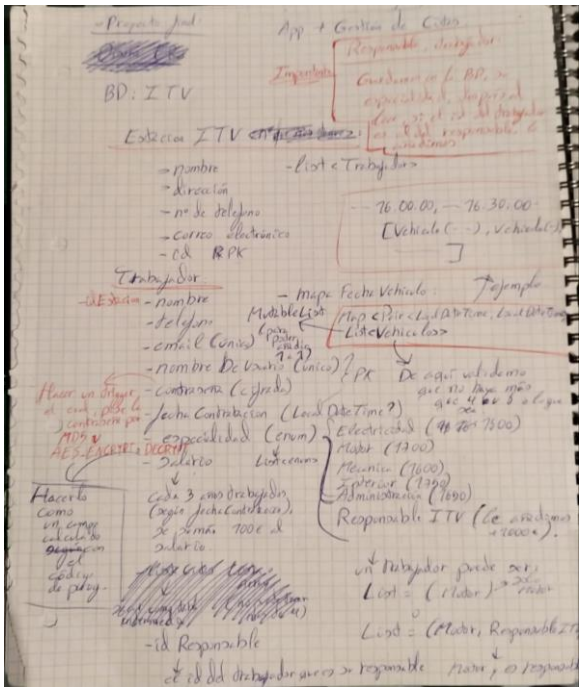
En el trello hemos estado colgando comentarios para mejorar la comunicación entre el grupo y notificar de las actualizaciones.

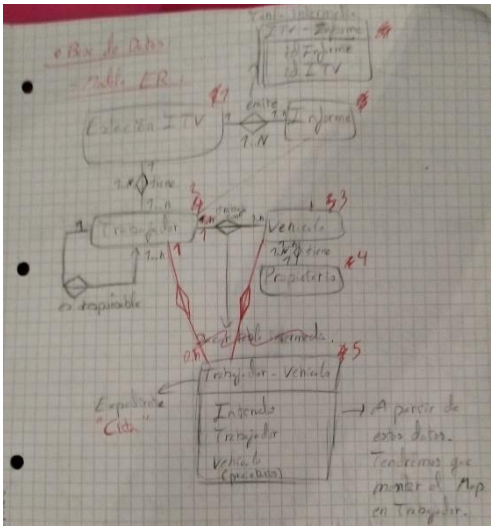
Hemos comenzado el proyecto con el diseño y creación de la pagina web, asi como el planteamiento de las relaciones en la base de datos. Tras concluir con esa parte, hemos seguido con el diseño de los distintos diagramas (DCU, Diagramas de secuencia, etc). Hemos implementado la base de datos y la hemos conectado a la parte de programación. Durante la creación del codigo, hemos seguido patrones de diseño MVVM y respetado los principios SOLID. Por último tras terminar la parte de desarrollo, hemos querido darle color a la interfaz gráfica del programa.

Etapa de análisis.

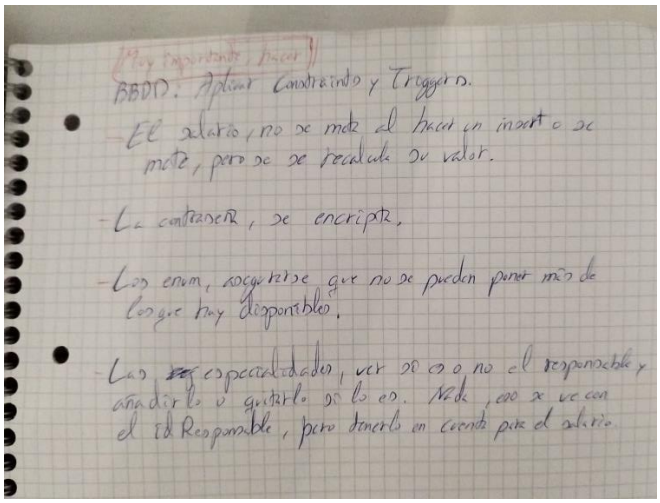
Durante la etapa de análisis, hicimos un diseño inicial de las tablas de bases de datos y clases de programación donde salieron ideas que fueron descartadas debido a la complejidad y que surgieron ideas más acertadas con el proyecto.

Tambien realizamos un diseño de las relaciones entre las tablas de la base de datos que se fueron puliendo hasta obtener el resultado final.

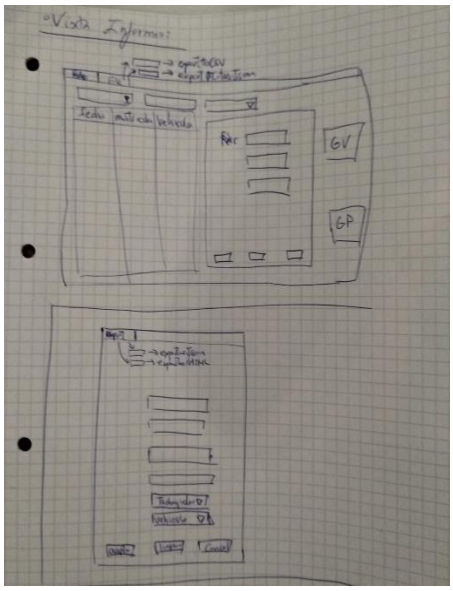




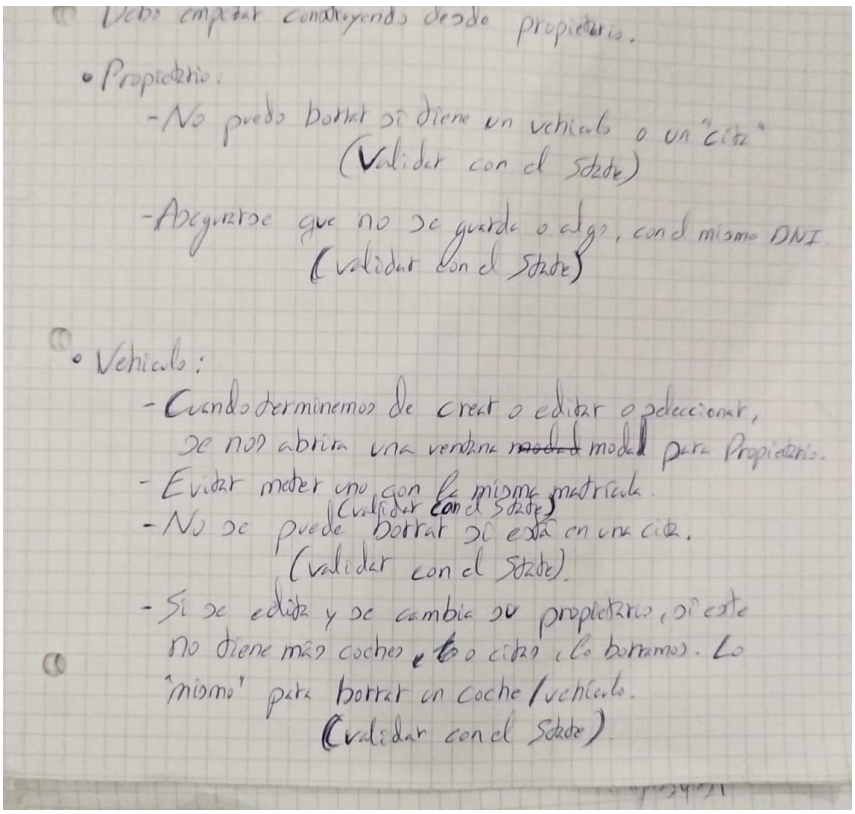
Se organizaron algunas de las restricciones para base de datos



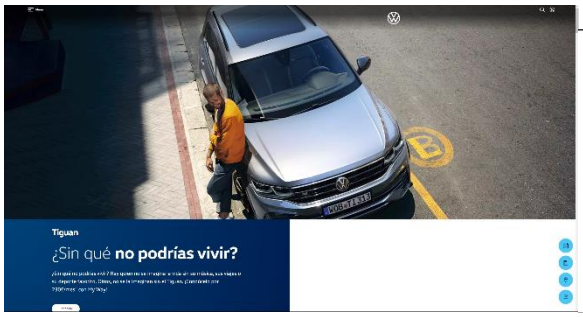
Se realizó un diseño inicial a papel de la interfaz grafica que mas tarde sería implementada con JavaFX (El diseño fue evolucionando durante la etapa de producción).

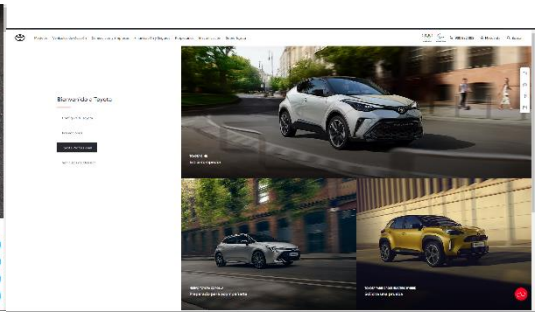


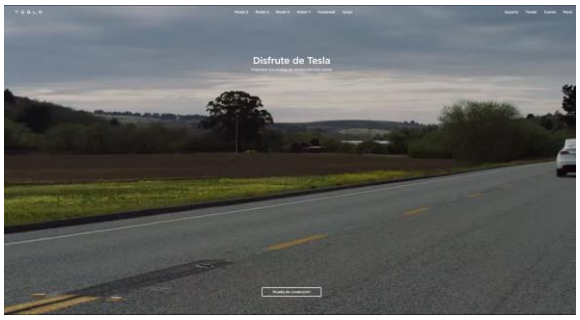
También se pensó como seria la funcionalidad de algunas posibles acciones del programa.

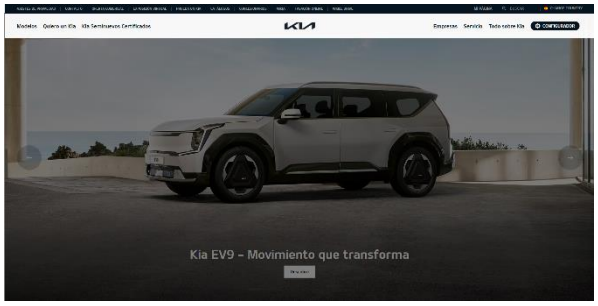


Para la planificación de la pagina web de lenguaje de marcas, nos inspiramos en diseños de paginas web de marcas conocidas, entre ellas Kia, Toyota, Tesla, Volkswagen, etc.









Para el esquema de colores, nos basamos en los colores que contiene el logo de ITV, con colores azulados y blancos. Estos colores finalmente fueron implementados tanto en la pagina web como en la aplicación de escritorio.

La división de las tareas ha sido:

BASE DE DATOS

- Modelo Entidad-Relación: Iván Ronco Cebadera

- Modelo Físico: Iván Ronco Cebadera

- Implementación de la base de datos: Ivan Ronco Cebadera, Kevin David Matute Obando

- Procedimientos y funciones: JiaCheng Zhang, Kevin Davind Matute Obando

LENGUAJE DE MARCAS

-Diseño del logo y eslogan: Kevin David Matute Obando

- Creación del sitio web: JiaCheng Zhang, Kevin David Matute Obando

-Creación del formulario: JiaCheng Zhang, Iván Ronco Cebadera

ENTORNOS

- Diagrama de Casos de Uso: Iván Ronco Cebadera

- Casos de Uso: Iván Ronco Cebadera, JiaCheng Zhang, Kevin David Matute Obando

- Diagrama de Secuencia: Iván Ronco Cebadera, JiaCheng Zhang, Kevin David Matute Obando

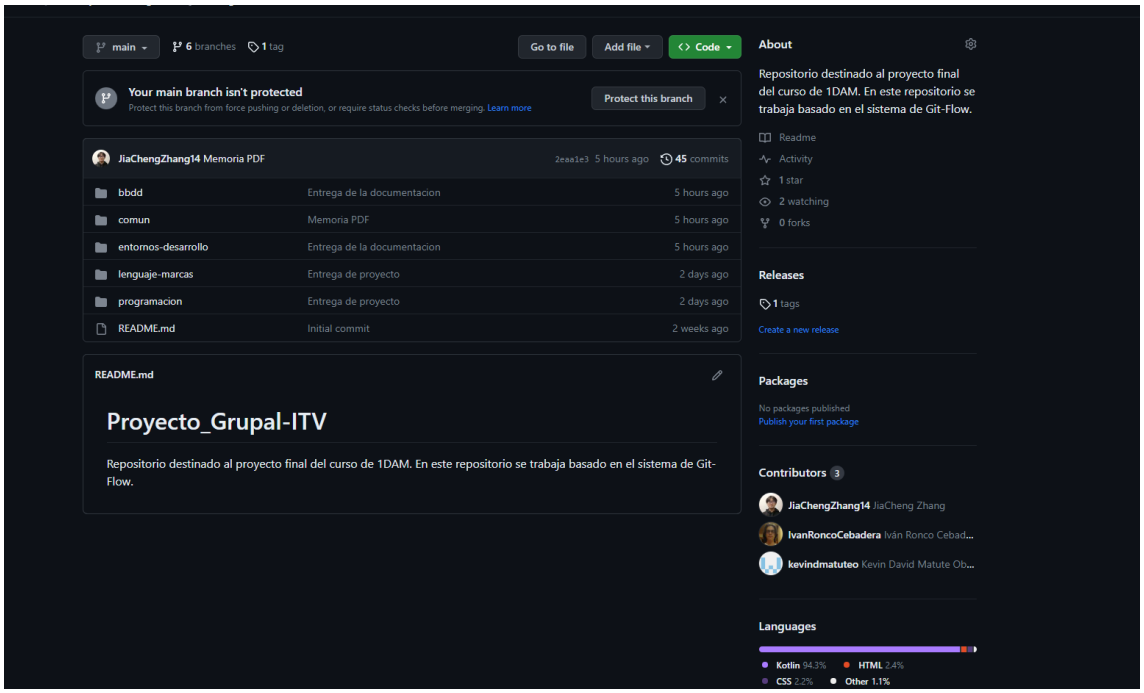
- Diagrama de Clases: Kevin David Matute Obando

- Testing: JiaCheng Zhang, Kevin David Matute Obando, Iván Ronco Cebadera

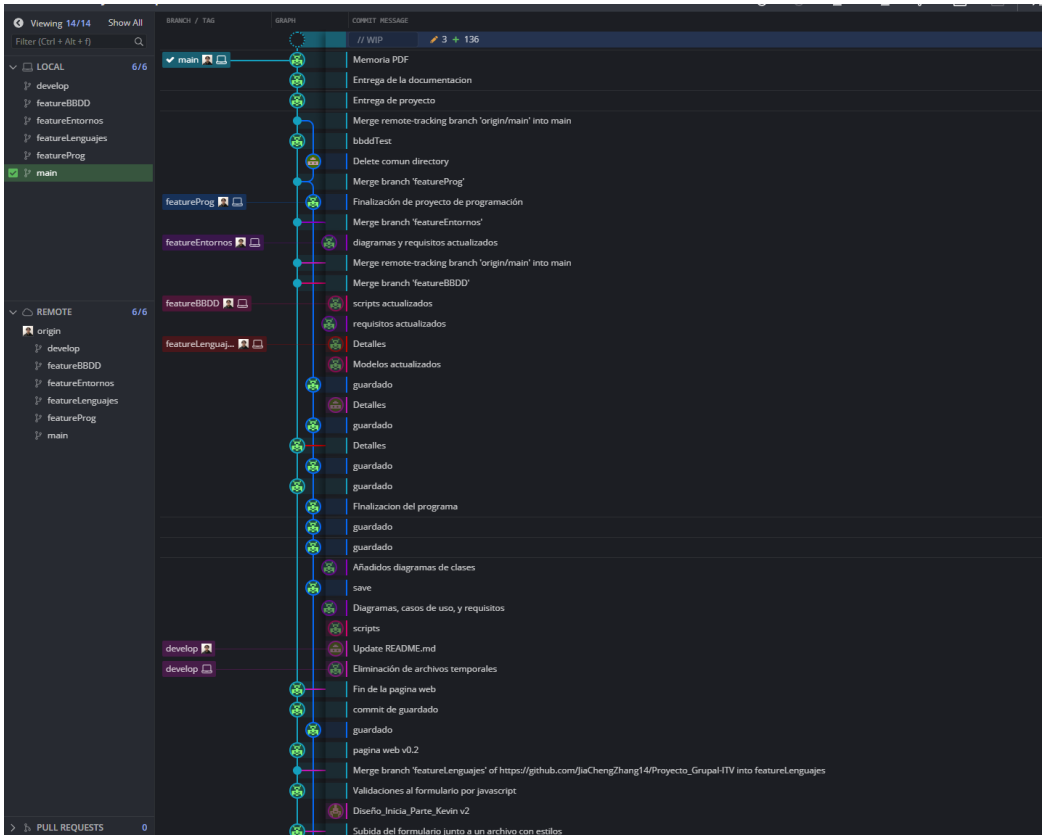
PROGRAMACION

- Creación de la app, back y front: JiaCheng Zhang, Kevin David Matute Obando, Iván Ronco Cebadera

Por otro lado, github y git flow se ha utilizado para como control de versiones. En github se ha creado un repositorio para el proyecto.



Este repositorio tiene varias ramas. Hay una rama feature para cada módulo. En estas se subieron actualizaciones de cada parte del trabajo. Después, cuando cada parte se completó, se mergearon las ramas en la rama main.



BASE DE DATOS

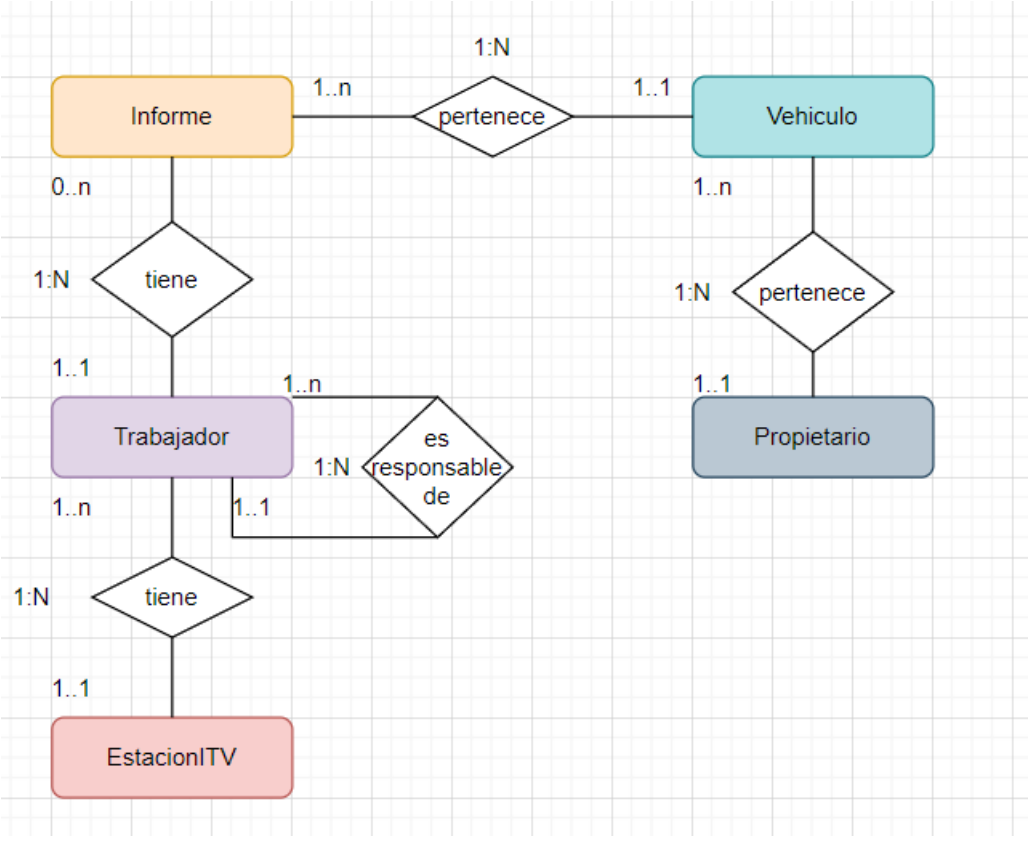
- Modelo Entidad-Relación

Es un primer planteamiento al modelo que se utilizará durante el proyecto, nos servirá como base para las tablas a crear de la base de datos y el tipo de relación que hay entre las entidades.

Las relaciones que hemos establecido son: una estación puede tener uno o varios trabajadores, mientras que un trabajador pertenece solo a una estación. Además, entre los trabajadores se encuentra un responsable de estos mismos. Un trabajador puede tener informes asociados, y cada informe pertenecerá solo a un trabajador. Un informe pertenecerá a un vehículo, el cual puede tener uno o varios informes. Además, el vehículo pertenece a un propietario, que puede uno o más vehículos.

Se ha hecho en la aplicación draw.io. Hemos asignado un color a cada entidad que se mantendrá en los siguientes diagramas. Los colores son los siguientes:

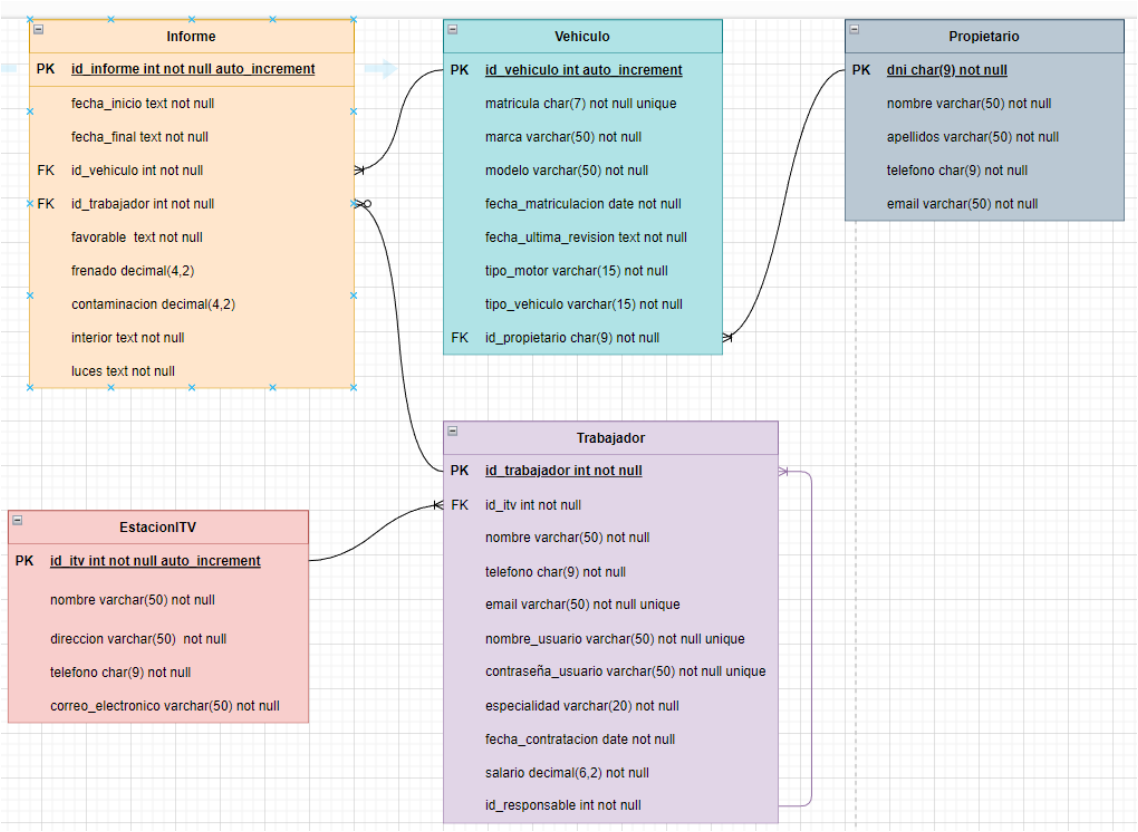
Estación: Rojo, Propietario: Gris, Informe: Amarillo, Vehículo: Azul, Trabajador: Violeta



- Modelo Físico

Este es el siguiente paso para el diseño, aquí se definen los atributos de cada tabla, sus claves primarias y las claves foráneas. La clave que encontramos en la tabla estaciónitv son: id_itv, auto incremental que actúa como clave primaria, las claves de la tabla trabajador son: id_trabajador, número que actúa como clave primaria, junto a id_itv, clave foránea que lo conecta con la tabla estacionitv. Las claves de tabla informe son: id_informe, autoincremental que actúa como clave primaria, id_trabajador, clave foránea que lo junta con la tabla trabajador y id_vehiculo, también clave foránea que lo conecta a la tabla vehículo. Las claves de la tabla vehículo son: id_vehiculo, autoincremental y clave primaria, junto a id_propietario, clave foránea que lo una a la

tabla propietario. Por último la clave de la tabla propietario es id_propietario, como clave primaria.

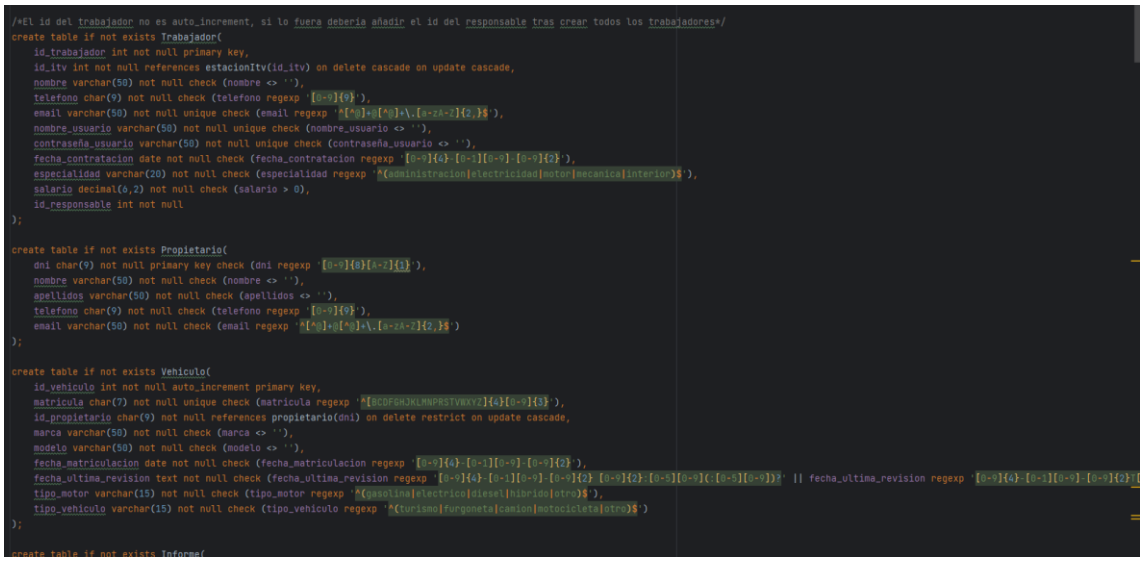


- Implementación de la base de datos

Las tablas se han implementado utilizando el lenguaje MySQL y la base de datos MariaDB, con los programas Data Grip y MySQL Workbench. Además, se ha utilizado Xampp para abrir el servidor. Se han incluido adecuadamente los tipos de cada atributo de las tablas y se han incluido las claves foráneas y las restricciones de los datos y de borrado necesarios para que el programa funcione correctamente, estos son:

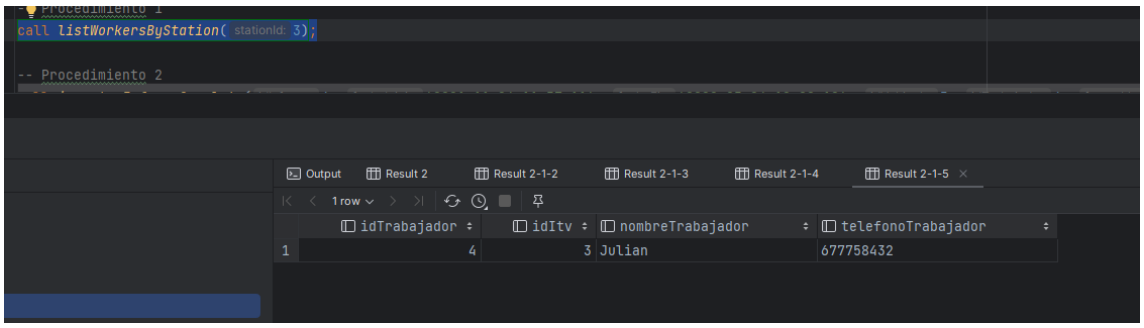
No se puede borrar un propietario si tiene un vehículo suyo guardado en la tabla vehículo. No se puede borra un trabajador si tiene un informe asociado a él. No se puede borrar un vehículo si tiene un informe asociado a él. A estas se les aplica la restricción “on delete restrict”.

Para restricciones de los datos de tablas se han utilizado tanto la función “check()” como el tipo y longitud de cada atributo. También se ha recurrido a expresiones regulares para campos como las fechas o las matrículas. Se ha comprobado que cada restricción funciona con “inserts” de prueba.

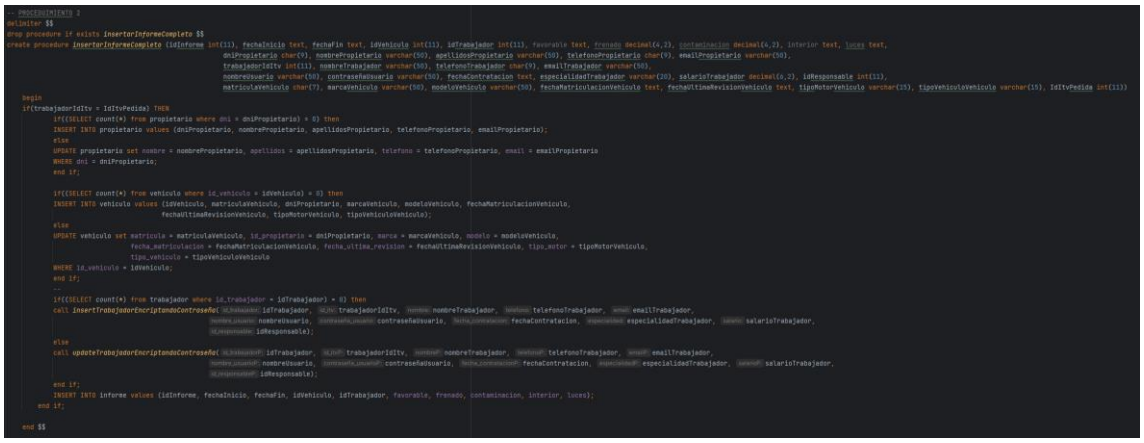


- Procedimientos y funciones

```
/*Procedimientos y funciones del apartado de Bases de Datos del Proyecto ITV*/

use empresaitv;

-- PROCEDIMIENTO 1
delimiter $$
drop procedure if exists listWorkersByStation $$
create procedure listWorkersByStation(stationId INT)
begin
    declare l_last_row_fetched int;
    declare idTrabajador, idItv int;
    declare nombreTrabajador varchar(50);
    declare telefonoTrabajador char(9);
    declare c1 cursor for select id_trabajador, id_itv, nombre, telefono from trabajador where id_itv = stationId;
    declare continue handler for not found set l_last_row_fetched = 1;
    set l_last_row_fetched = 0;
    open c1;
    repeat
        fetch c1 into idTrabajador, idItv, nombreTrabajador, telefonoTrabajador;
        select idTrabajador, idItv, nombreTrabajador, telefonoTrabajador;
    until l_last_row_fetched = 1 end repeat;
    close c1;
end ;$$
```

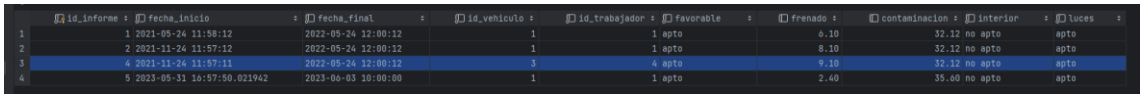


El primer procedimiento que se pide lista los trabajadores que pertenecen a una estación. Se pasa por parámetro el id de la estación, se seleccionarán y mostrarán los trabajadores que pertenezcan a esa estación, o sea que en los campos del trabajador coincida el id introducido por parámetro con el del propio trabajador. El número de nuestra estación es el nº3.



```
-- Procedimiento que inserta un trabajador a la tabla trabajador y encripta su contraseña
delimiter $$
drop procedure if exists insertTrabajadorEncriptandoContraseña; $$
create procedure insertTrabajadorEncriptandoContraseña
(id_trabajador int(11), id_itv int(11), nombre varchar(50), telefono char(9), email varchar(50), nombre_usuario varchar(50), contraseña_usuario varchar(50), fecha_contratacion date,
especialidad varchar(20), salario decimal(6, 2), id_responsable int(11))
begin
    insert into trabajador values
    (id_trabajador, id_itv, nombre, telefono, email, nombre_usuario, MD5(contraseña_usuario), fecha_contratacion, especialidad, salario, id_responsable);
end; $$

-- Procedimiento que actualiza la tabla trabajador y encripta su contraseña
delimiter $$
drop procedure if exists updateTrabajadorEncriptandoContraseña; $$
create procedure updateTrabajadorEncriptandoContraseña
(id_trabajadorP int(11), id_itvP int(11), nombreP varchar(50), telefonoP char(9), emailP varchar(50), nombre_usuarioP varchar(50), contraseña_usuarioP varchar(50), fecha_contratacionP date,
especialidadP varchar(20), salarioP decimal(6, 2), id_responsableP int(11))
begin
    update trabajador set id_itv = id_itvP, nombre = nombreP, telefono = telefonoP, email = emailP, nombre_usuario = nombre_usuarioP,
    contraseña_usuario = MD5(contraseña_usuarioP), fecha_contratacion = fecha_contratacionP, especialidad = especialidadP, salario = salarioP, id_responsable = id_responsableP
    where id_trabajador = id_trabajadorP;
end; $$
```

```
call insertarInformeCompleto( idInforme 4, fechaInicio '2021-11-24 11:57:11', fechaFin '2022-05-24 12:00:12', idVehiculo 3, idTrabajador 4, favorable 'apto', frenado 9.10, contaminacion 32.12, interior 'no apto', luces 'apto',
dniPropietario '334565563', nombrePropietario 'Pedro', apellidosPropietario 'Marquez', telefonoPropietario '655409907', emailPropietario 'aasdas@email.com',
trabajadorIdItv 3, nombreTrabajador 'Julian', telefonoTrabajador '677758432', emailTrabajador 'aasdas@email.com', nombreUsuario 'julian', contraseñaUsuario 'contrasenia', fechaContratacion '2012-05-24', especialidadTrabajador
matriculaVehiculo '5555222', marcaVehiculo 'Opel', modeloVehiculo ''Corsa', fechaMatriculacionVehiculo '2012-12-24', fechaUltimaRevisionVehiculo '2017-05-24 11:58:21', tipoMotorVehiculo 'diesel', tipoVehiculoVehiculo 'turismo
);
-- para el trigger
```

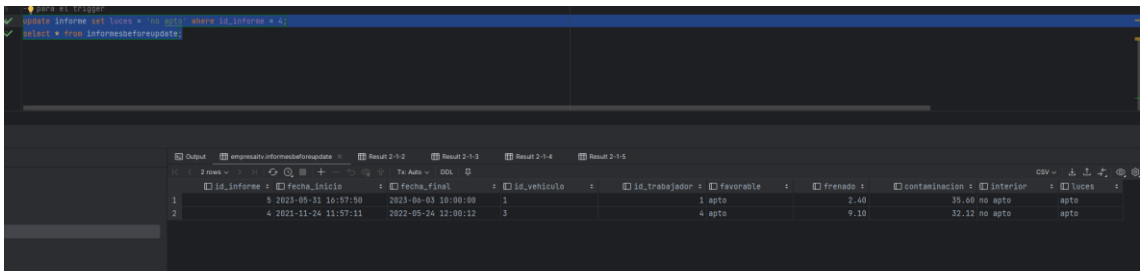


	id_informe	fecha_inicio	fecha_final	id_vehiculo	id_trabajador	favorable	frenado	contaminacion	interior	luces
1	1	2021-05-24 11:58:12	2022-05-24 12:00:12	1	1	apto	6.10	32.12	no apto	apto
2	2	2021-11-24 11:57:12	2022-05-24 12:00:12	1	1	apto	8.10	32.12	no apto	apto
3	4	2021-11-24 11:57:11	2022-05-24 12:00:12	3	4	apto	9.10	32.12	no apto	apto
4	5	2023-05-31 16:57:50.021942	2023-06-03 10:00:00	1	1	apto	2.40	35.60	no apto	apto

El siguiente procedimiento añade un informe a la base de datos. Para ello le entran todos los campos que pertenecen a las tablas vehículo, Trabajador, Informe y Propietario, ya que para que exista un informe deben existir también las demás entidades. Su funcionamiento consiste en comprobar si los datos pedidos por parámetro existen ya en las tablas, si es así actualiza los datos con un update, sino hace un insert para introducirlos.

Aparte tenemos dos funciones, una que hace un insert a la tabla trabajador con la contraseña cifrada, y otro que hace un update a la misma tabla también con la contraseña cifrada. El cifrado se realiza con la función MD5(). Estas funciones se llaman en la función grande a la hora de hacer el “insert” o “update” del trabajador.

```
-- Trigger que hace una guarda los datos antigüos de la tabla informe cada vez que se introduce un informe nuevo
delimiter $$
drop trigger if exists save_before_update $$
create trigger save_before_update after update on informe
    for each row
    begin
        insert into informesBeforeUpdate(id_informe, fecha_inicio, fecha_final, id_vehiculo, id_trabajador, favorable, frenado, contaminacion, interior, luces)
            values (old.id_informe, old.fecha_inicio, old.fecha_final, old.id_vehiculo, old.id_trabajador, old.favorable, old.frenado, old.contaminacion,
                    old.interior, old.luces) ;
    end ;$$

-- EVENTO que hace un borrado de la tabla informes cada dos meses
delimiter $$
DROP EVENT IF EXISTS borrado_informes_bimestral $$
CREATE EVENT borrado_informes_bimestral ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 2 month
    DO
    BEGIN
    DELETE FROM informe;
    END; $$

set global event_scheduler = on;
```



El trigger consiste en, cada vez que se actualiza un valor la tabla informe, coge los datos previos al cambio y los almacena con un insert en otra tabla que hemos creado específicamente para que funcione como almacén/copia de seguridad. Esta tabla tiene los mismos campos que la tabla informe.

```
-- EVENTO que hace un borrado de la tabla informes cada dos meses
delimiter $$
DROP EVENT IF EXISTS borrado_informes_bimestral $$
CREATE EVENT borrado_informes_bimestral ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 2 month
    DO
    BEGIN
    DELETE FROM informe;
    END; $$

set global event_scheduler = on;
```

Por último, el evento consiste en borrar los datos de la tabla informe cada dos meses. Por eso se pone "interval 2 month" para que cada dos meses se ejecute el evento y borre la tabla informe.

LENGUAJE DE MARCAS

-Diseño del logo y eslogan

Para la creación del logo hemos utilizado Canva utilizando las iniciales del nombre de la empresa (Inspeccionamos Tu Coche s.a.), hemos decidido el color azul por ser color de la ITV y el eslogan, Para vosotros conductores, se inspira en el de PlayStation ("Para vosotros jugadores"). El eslogan pretende acercarse a un público mayoritariamente joven. El logo lo conforman tres elementos, las iniciales, una lupa y un coche. La lupa y el coche se combinan de forma que parezca que se está inspeccionado el vehículo. Y las iniciales se han elegido para mostrar nuestro nombre de forma elegante y minimalista. Estos elementos se encuentran delante de un fondo azul, color que nos representa.

- Creación del sitio web

Hemos utilizado Html para la estructura. Esta está conformada por un header, el cual contiene la barra de menú, el main, que contiene los apartados desglosados que aparecen en el menú, y un footer, que contiene un address con los datos de contacto de la empresa y los derechos de propiedad. También se ha usado Css para aplicar el estilo utilizando clases e ids para dar estilo a cada componente de la página. Esta página sigue el esquema de colores Azul y Blanco del logo y presenta nuestra empresa, sus funciones y servicios.

-Creación del formulario

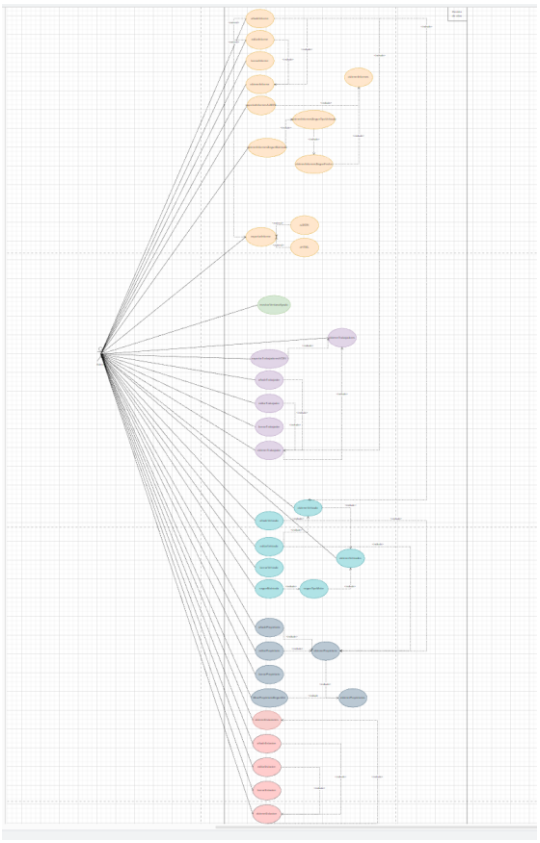
Para el formulario se ha usado html y css de la misma forma que en la página principal, además de utilizar java script para las restricciones y demás funcionalidades que debe tener el formulario. Al formulario se accede desde la parte superior-derecha de la página, justamente donde está imagen.

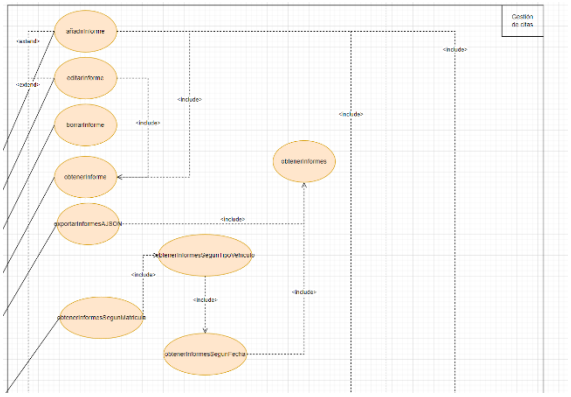
Cada campo del formulario esta validado, con una expresión regular, para que sea correcto. Hay tres botones: Enviar, Limpiar y Volver. Para enviar el formulario se deberá rellenar correctamente, si no saldrá un fallo. Si esta todo correcto se abrirá una ventana de correo para enviar un correo a la dirección que se ha especificado en el “mailto”. El botón de limpiar vaciará el formulario, en caso de que haya datos escritos. El botón volver nos devolverá a la página principal.

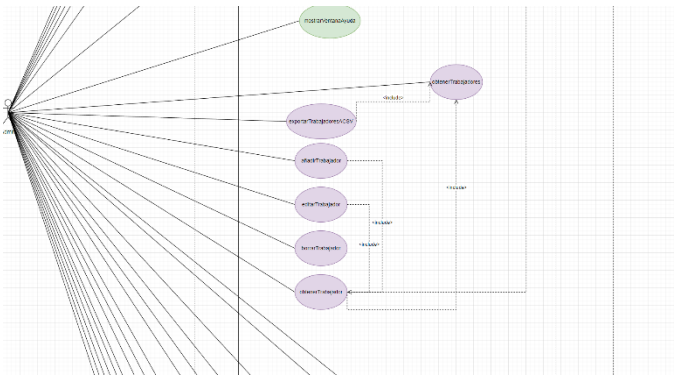
ENTORNOS

- Diagrama de Casos de Uso

En este diagrama empezamos definiendo que el único acto existente es el Admin, este es quien tendrá acceso a los casos de uso definidos. Tenemos los casos típicos de una gestión CRUD de las siguientes entidades: Estación, Propietario, vehículo, Trabajador e Informe. Durante la creación de este diagrama se ha aplicado <include> y <extend> donde se ha considerado necesario. Aparte de los casos típicos de CRUD se han añadido las funcionalidades del enunciado, como exportar trabajadores o informes a Json o Html, funciones para buscar según unos filtros, como buscas según una matrícula o una fecha. También el caso de uso de mostrar una ventana de ayuda. El caso de uso más destacado es el de añadir informe y que tiene dos <include>, uno para seleccionar el vehículo y otro para seleccionar los trabajadores







- Casos de Uso

Hemos hecho los casos de uso de los trece casos pedidos (añadir, editar y actualizar de vehículo, propietario, informe, trabajador y adicionalmente realizar una inspección de vehículo).

Todos los casos de uso se han estructurado de la misma forma. Se especifica el código del caso de uso (CU-01) después su nombre (añadirVehiculo), luego el requisito funcional que representa (RF-16; añadir un nuevo vehiculo), después los actores (Admin), y se termina definiendo la precondición, la postcondición y los posibles pasos para realizar la operación definida en el caso de uso

Código:	CU-01	Nombre:	añadirVehículo
RF:	RF-16; añadir un nuevo vehículo.	Actores:	admin

Precondicion:
Haber iniciado el programa, haber seleccionado previemente la opción de "Gestionar vehiculo" y haber seleccionado la opcion de "añadir vehiculo"

Postcondicion:
Si el proceso fue exitoso, se añadirá un nuevo vehículo en la BBDD, si no fue exitoso, se habrá mostrado el error que se haya producido.

Pasos:
1. Rellenar los campos del vehículo a crear. 2. Seleccionar la opción "aceptar" o "cancelar". 2.1. Si se selecciona "aceptar", se trata de crear el vehículo, si algo falla, no se crea y se devuelve el error. 2.2. Si se selecciona "cancelar" se pide confirmación para terminar la operación, si se niega, la operación no finaliza.

En este caso de uso, la precondición nos informa de que la aplicación debe estar en ejecución y que el usuario debe seleccionar la opción de gestionar vehículo.

La postcondición se divide en 2 casos; En caso de que el proceso sea correcto, se añadirá un nuevo vehículo en la base de datos; En caso de que sea incorrecto, se mostrará un mensaje de error.

Código:	CU-11	Nombre:	editarTrabajador
RF:	RF-34;editar un trabajador seleccionado.	Actores:	admin

Precondicion:
Haber iniciado el programa, haber seleccionado previamente la opcion de "Gestionar trabajador", haber seleccionado un trabajador, y haber seleccionado la opcion de "editar trabajador"

Postcondicion:
Si el proceso fue exitoso, se habrán aplicado cambios al trabajador en la BBDD, si no fue exitoso, se habrá mostrado el error que se haya producido.

Pasos:
1. Rellenar los campos del trabajador a editar. 2. Seleccionar la opción "aceptar" o "cancelar". 2.1. Si se selecciona "aceptar", se trata de editar el trabajador, si algo falla, no se edita y se devuelve el error. 2.2. Si se selecciona "cancelar" se pide confirmación para terminar la operación, si se niega, la operación no finaliza.

En el caso de uso 11, vemos que la precondición es similar a la del caso de uso anterior, requerimos que se inicie la aplicación, que el usuario seleccione un trabajador y que seleccione la opción de editar trabajador.

En cuanto a las postcondiciones, tambien se divide en 2 casos. Si sale bien, el trabajador editado se guardará en la base de datos y en caso contrario, se mostrará un mensaje de error específico.

Código:	CU-12	Nombre:	borrarTrabajador
RF:	RF-35;borrar un trabajador seleccionado.	Actores:	admin

Precondicion:
Haber iniciado el programa, haber seleccionado previamente la opcion de "Gestionar trabajador", haber seleccionado un trabajador y haber seleccionado la opción de "borrar trabajador"

Postcondicion:
Si el proceso fue exitoso, se habrán borrado los datos del trabajador seleccionado de la BBDD, si no fue exitoso, se habrá mostrado el error que se haya producido.

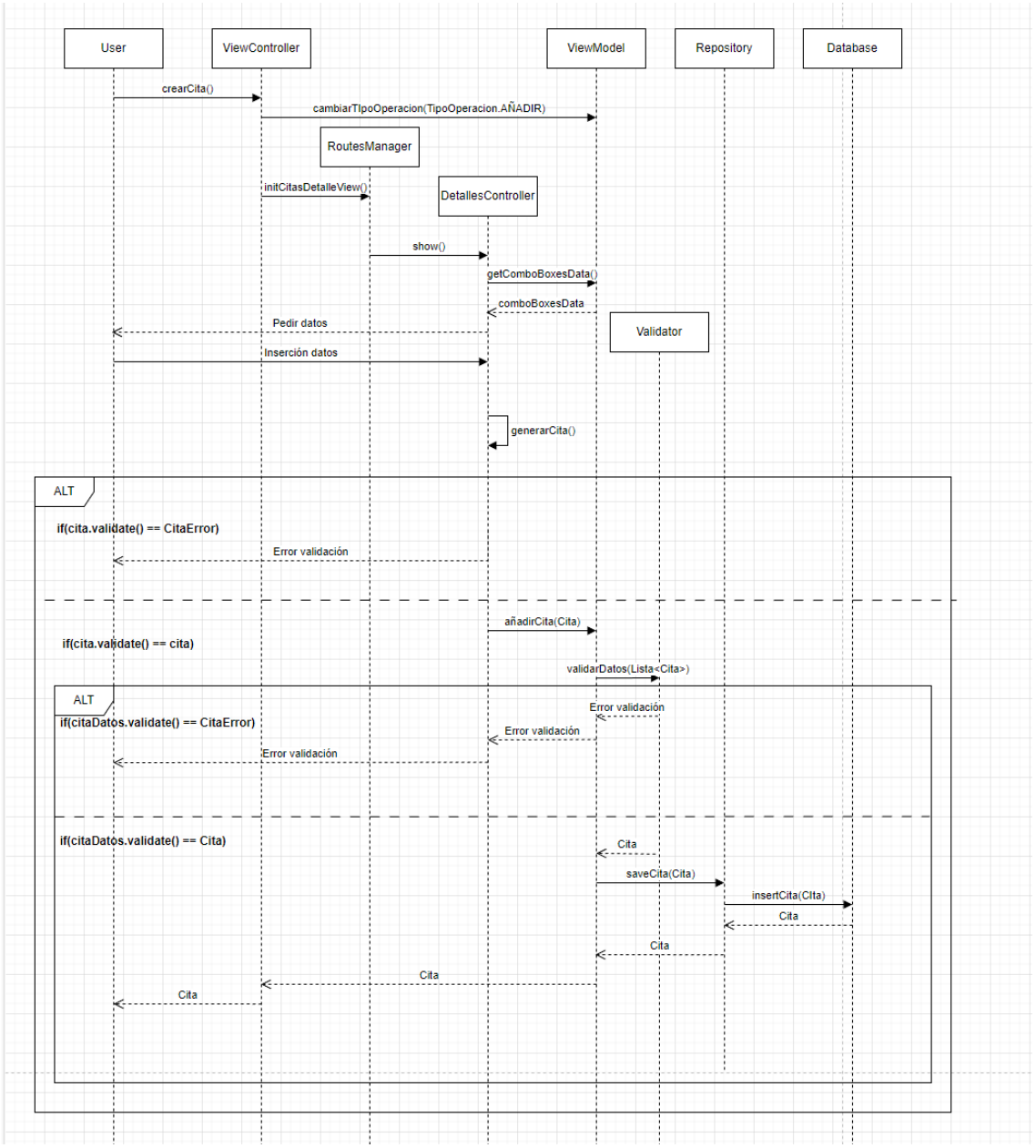
Pasos:
1. Seleccionar la opción "aceptar" o "cancelar" 1.1. Si se selecciona "aceptar", se trata de eliminar el trabajador, si algo falla, no se elimina y se devuelve el error. 1.2. Si se selecciona "cancelar" se pide confirmación para terminar la operación, si se niega, la operación no finaliza.

Para el caso de uso 12, la precondición es haber iniciado el programa, haber seleccionado un trabajador y seleccionar la opción de borrar.

Las postcondiciones se dividen en 2 partes, si el proceso sale bien, el trabajador se borra de la base de datos. Si sale mal, se mostrará el mensaje de error pertinente.

- Diagrama de Secuencia

Se han elaborado los diagramas de secuencia para los casos: Añadir Informe, Imprimir Informe y Actualizar Trabajador. En estos se muestran todas las clases que se involucran para la elaboración de cada uno de los tres sucesos. Mostramos el orden de llamadas a funciones que se elaboran durante los sucesos, mostramos visualmente de que clase sale la función y a que clase va. Señalizamos cuales son las posibles opciones con ciertas condiciones según los procesos. Por último, se enseñan los posibles resultados que pueden devolver las operaciones.



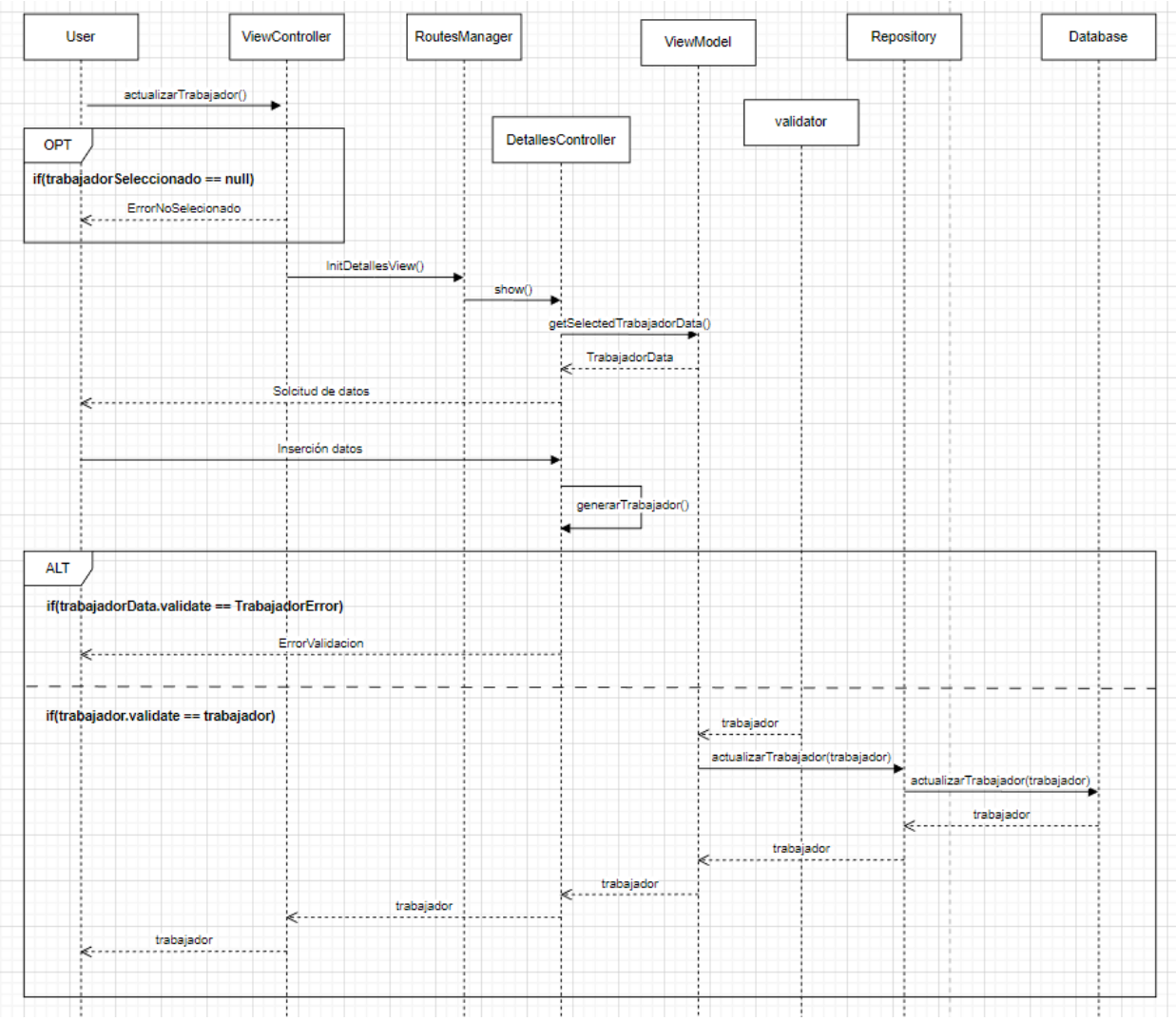
Una vez el usuario ha iniciado el proceso de crear una cita, el controlador de la ventana de gestión de citas cambia el tipo de operación a “AÑADIR”, modificando el ViewModel, después se llama al routes manager para iniciar la vista de detalles para poder añadir una nueva cita, el routes manager llama al controlador de la vista de detalles y mostrando la vista, se recogen todos los datos del ViewModel necesarios para mostrar los trabajadores y vehículos en los combo box de los datos del informe a crear.

Tras mostrar la ventana de detalles, el usuario puede introducir los datos de la cita. Una vez que se acepte la operación, dentro del controlador de la vista de detalles, se validaran los campos insertados y a su vez se creará el informe con los campos.

En caso de que los campos no se hayan validado correctamente, se mostrará un mensaje de error al usuario.

Si no se ha producido ningún error el programa creará la cita y se continuará con la ejecución del programa.

Una vez creada la cita, se llama al ViewModel para insertar en la base de datos la nueva cita, sin embargo, antes de realizar la inserción, se validará que cumpla con todas la restricciones necesarias en el programa, como que no se pueda crear más de 4 citas para el mismo trabajador en el mismo intervalo de tiempo. Si pasa las validaciones, la cita se inserta en la base de datos llamando al repositorio y este llamando al database. Finalmente, la cita insertada se devuelve al usuario, mostrándola en la tabla.

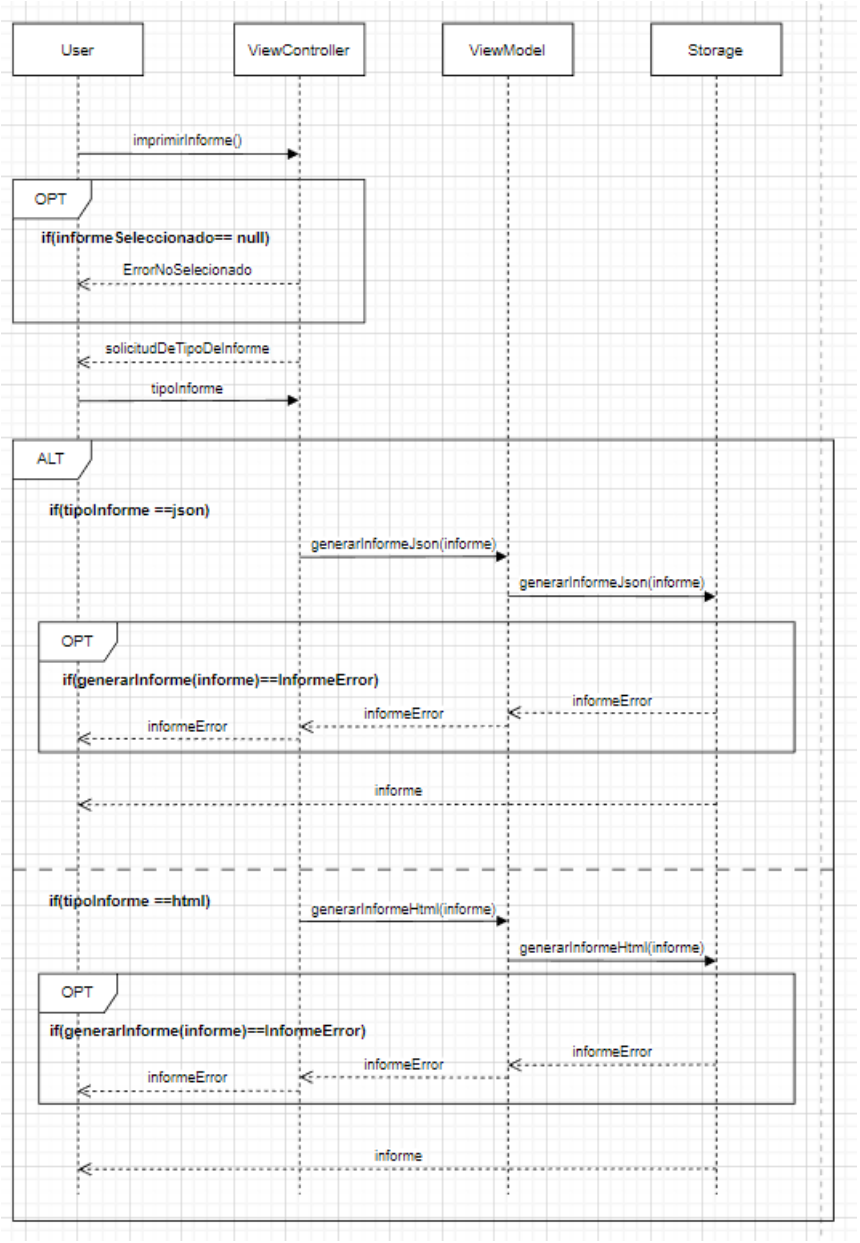


El usuario inicia la actualización del trabajador, el controlador de la vista inicia la acción validando primero si se ha seleccionado un trabajador, en caso de que no se haya seleccionado, se devolverá un mensaje de error.

Una vez terminado el proceso de validación, el RoutesManager inicia la vista de detalles y recupera los datos para poder mostrarlos en los campos de la vista.

Cuando el usuario haya rellenado los datos, se validarán los campos introducidos y se generará un trabajador que pasará por otro validador (Se validan que el email, nombre de usuario y contraseña de usuario sean únicos para el trabajador, aparte de comprobar que el resto de datos estén bien escritos). Si no se valida el trabajador, se devolverá un mensaje de error. Si el trabajador es validado correctamente, se llamará

al repositorio y se actualizará el trabajador en la base de datos. Finalmente se devuelve el trabajador ya actualizado mostrándolo en la tabla.



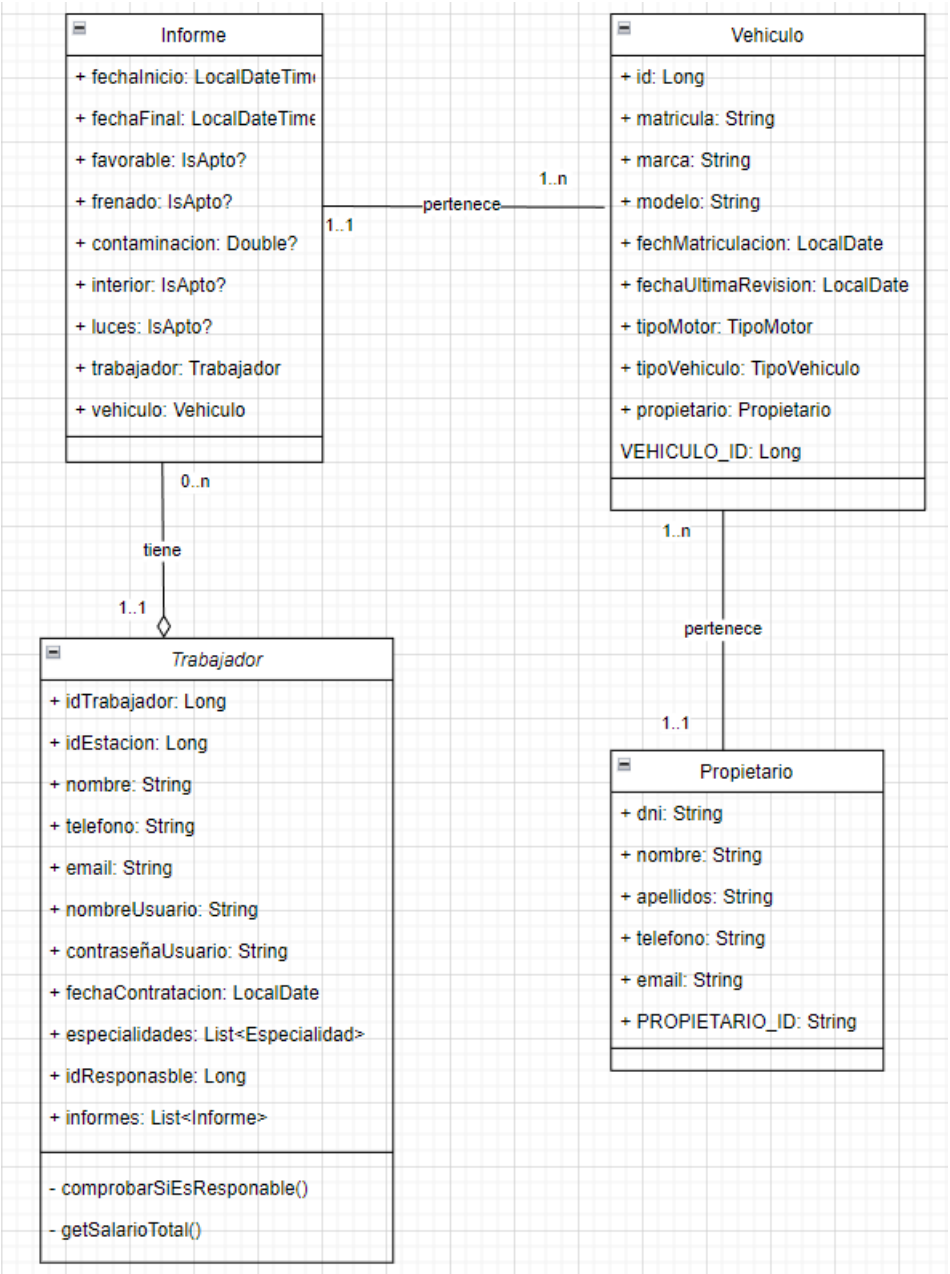
El usuario inicia la acción de imprimir un informe. Se valida que se haya seleccionado un informe, si el informe seleccionado es nulo, o no se ha seleccionado ningún informe, se mostrará un mensaje de error. Si se ha seleccionado un informe, entonces el usuario decidirá qué tipo de fichero quiere para el informe, existen 2 opciones.

Si el usuario elige JSON, el ViewModel llamará al Storage para generar el informe en JSON, si se produce algún error durante la generación, se nos devuelve un mensaje de error. En caso de que se haya generado bien, se generará un informe .json en la estructura de carpetas predefinida en el codigo (files/informes).

Si el usuario elige HTML, el ViewModel llamará al Storage para generar el informe en HTML, si se produce algún error durante la generación, se nos devuelve un mensaje de error. En caso de que se haya generado bien, se generará un informe .html en la estructura de carpetas predefinida en el codigo (files/informes).

- Diagrama de Clases

En este diagrama se definen las entidades utilizadas durante el apartado de programación junto a sus atributos y las funciones, en caso de que tengan. Además, se indica la visibilidad de las funciones y los atributos, + si es público, - si es privado.



Hemos planteado una estructura de clases que se relacionan de una forma similar a las tablas de la base de datos.

A un informe pertenecen uno o varios vehículos y un vehículo solo puede pertenecer a un informe.

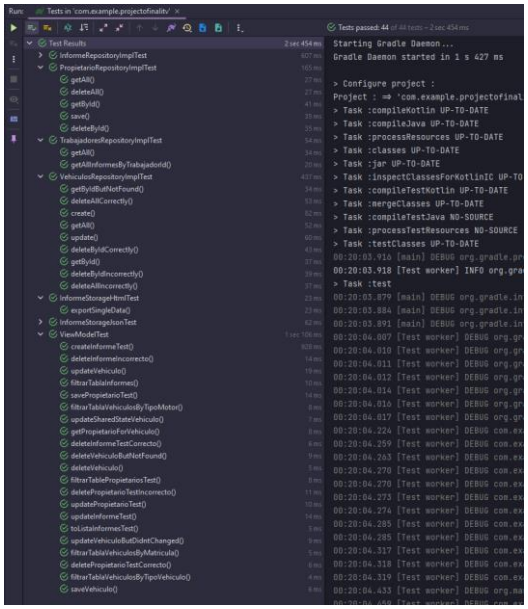
Un trabajador puede no tener informes o tener varios informes (no más de 4 en el mismo intervalo de tiempo en la misma fecha) y un informe solo lo puede tener un trabajador. El trabajador tiene una lista de informes, que puede estar vacía o contener informes.

Un vehículo pertenece a un solo propietario y un propietario puede tener uno o muchos vehículos.

Hemos querido realizar un diseño similar a las tablas de base de datos para que el proyecto sea uniforme y que, gracias a esa uniformidad, sea mas sencilla la parte de la implementación y la producción.

- Testing

En cuanto a los test unitarios hemos testeado los repositorios, los storage y el view model. En el viewModel hemos usado la librería de Mockito para poder mockear los repositorios y los storages. Cabe destacar que, al testear las funciones de exportar, tanto a Json como a Html, se ha comprobado que los fichero se crean correctamente.

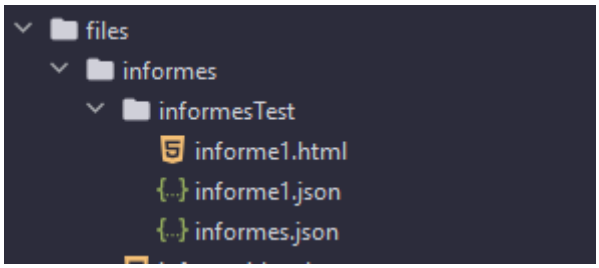


Tenemos 44 test que pasan limpios.

Durante le proceso de testeo, solo se ha aplicado el uso de la librería de mockito en el test del ViewModel.

```
private val configApp = ConfigApp()
private var filePath = configApp.APP_FILE_PATH +File.separator+"informes"+File.separator+ "informesTest"

private val singleFile = File( pathname: filePath + File.separator+"informe1.json")
private val multiFile = File( pathname: filePath + File.separator + "informes.json")
```

```
JiaCheng Zhang
@Test
fun exportSingleData() {
    println(singleFile)
    val res : Result<Informe, InformeError> = storageJson.exportSingleData(informe)
    assertEquals(Ok(informe), res)
    assertTrue(singleFile.exists())

}

JiaCheng Zhang
@Test
fun exportMultipleData() {
    val informes : List<Informe> = listOf(informe, informe2)
    val res : Result<List<Informe>, InformeError> = storageJson.exportMultipleData(informes)
    assertEquals(Ok(informes), res)
    assertTrue(multiFile.exists())
}

}
```

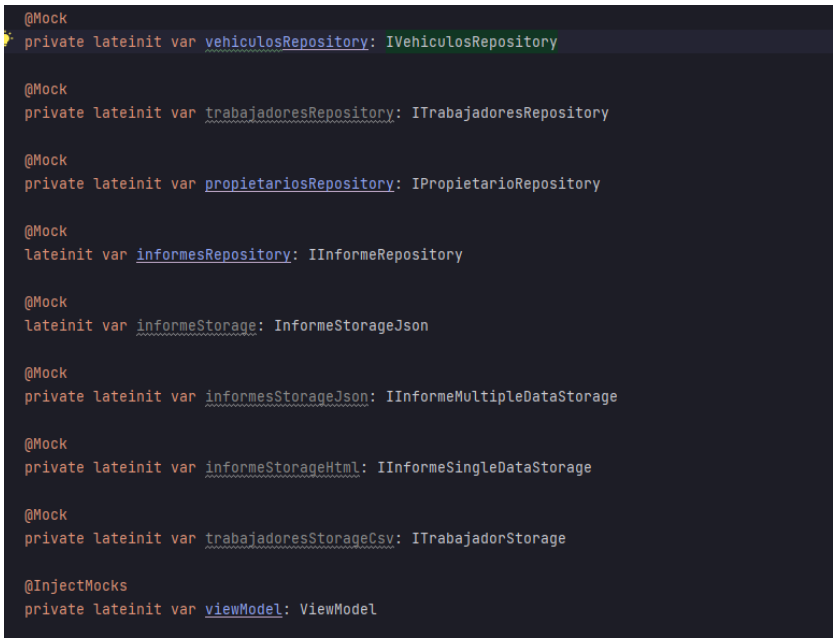


En el test del storage, se comprueba que las funciones nos den los resultados esperados y adicionalmente se comprueba que los ficheros se crean correctamente.

```
    JiaCheng Zhang
    @Test
    fun getAll() {
        val res : List<Trabajador> = repository.getAll()
        assertEquals(trabajadores[0], res[0])
        assertEquals(trabajadores[1], res[1])
    }

    JiaCheng Zhang
    @Test
    fun getAllInformesByTrabajadorId() {
        val res : List<Informe> = repository.getAllInformesByTrabajadorId( id: 1)
        val res2 : List<Informe> = repository.getAllInformesByTrabajadorId( id: 2)
        assertEquals(listOf(informe),res)
        assertEquals(listOf(informe2), res2)

    }
}
```

En el repositorio comprobamos que la conexión con la base de datos (una base de datos para test) se correcta y que las funciones nos devuelvan los resultados esperados.

```
JiaCheng Zhang
@Test
fun deletePropietarioTestCorrecto(){
    whenever(propietariosRepository.deleteById(propietarios[0].dni)).thenReturn( value: true)
    val res : Result<Boolean, PropietarioError> = viewModel.deletePropietario(propietarios[0].dni)
    assertEquals(Ok( value: true), res)
}

JiaCheng Zhang
@Test
fun deletePropietarioTestIncorrecto(){
    whenever(propietariosRepository.deleteById(propietarios[0].dni)).thenReturn( value: false)
    val res : Result<Boolean, PropietarioError> = viewModel.deletePropietario(propietarios[0].dni)
    assertEquals(res.getError()!!.message, actual: "El propietario de dni: ${propietarios[0].dni}, no fue encontrado")
}
```



En el test del ViewModel, aplicamos test con dobles usando Mockito Kotlin. Mockeamos tanto los repositorios como los storages, debido a que previamente estos han sido testeados y asi podemos simular los resultados de estos mismos.

-Arquitectura SOLID

Hemos aplicado los principios SOLID en distintos apartados del proyecto.

Principio de responsabilidad única. Antes de guardar o editar se llama al validador para que compruebe que todo es valido con respecto al objeto que se esté tratando de guardar o editar.

Principio Abierto/Cerrado: Este se ve reflejado en los repositorios, habiendo creado una primera interfaz llamada CrudRepository que implementa interfaces del repositorio, como IPropietarioRepository y que se termina implementando en la clase PropietarioRepositoryImpl.

Principio de sustitución de Liskov. En este ejercicio no se llega a aplicar debido a que no se aplica la herencia.

Principio de Segregación de Interfaces: Se puede apreciar en el mismo ejemplo que el propuesto para el principio de Abierto/Cerrado con la estructura de los repositorios.

Principio de inversión de dependencias: Cuando se inyecta los repositorios y los storages en el ViewModel se inyectan los tipos de las interfaces que implementan, no el tipo de la clase como tal.

-Mecanismos para proveer dependencias

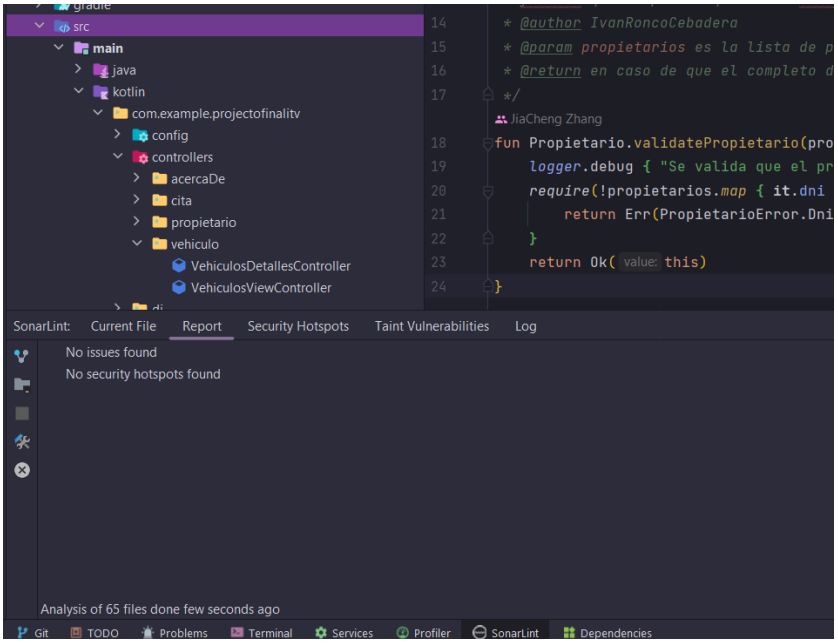
Para la inyección de dependencias hemos utilizado la librería de Koin sin anotaciones debido a que JavaFx no permite utilizar Koin con anotaciones. Se ha creado un fichero con nuestro modulo myModule, en donde se han creado singletons de los repositorios, el ConfigApp, los storages, el DatabaseManager y el ViewModel. Después inyectamos el ViewModel en los controladores de forma perezosa. En el main se inicializa el módulo que hemos creado.

PROGRAMACION

- Creacion de la app, back y front

Para la creación de la aplicación hemos utilizado IntelliJ y el lenguaje Kotlin para hacer el código de esta. Como base de datos hemos utilizado mariaDB. Para la interfaz hemos utilizado JavaFx y SceneBuider.

Hemos aplicado Railway Oriented Programming en el storage, en funciones del ViewModel que llaman a los storages o a los reporitorios, y en los validadores. Se le ha ido pasando regularmente el Sonarlint, mientras se avanzaba en el proyecto, para ir corrigiendo los errores que iban apareciendo.



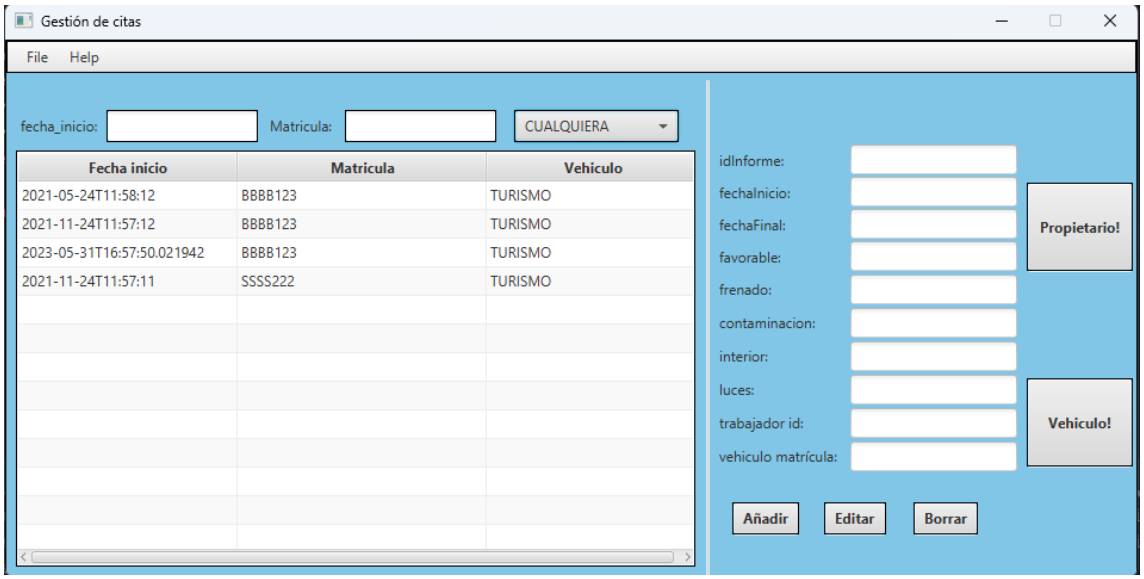
Pasamos el SonarLint sin ningún error.

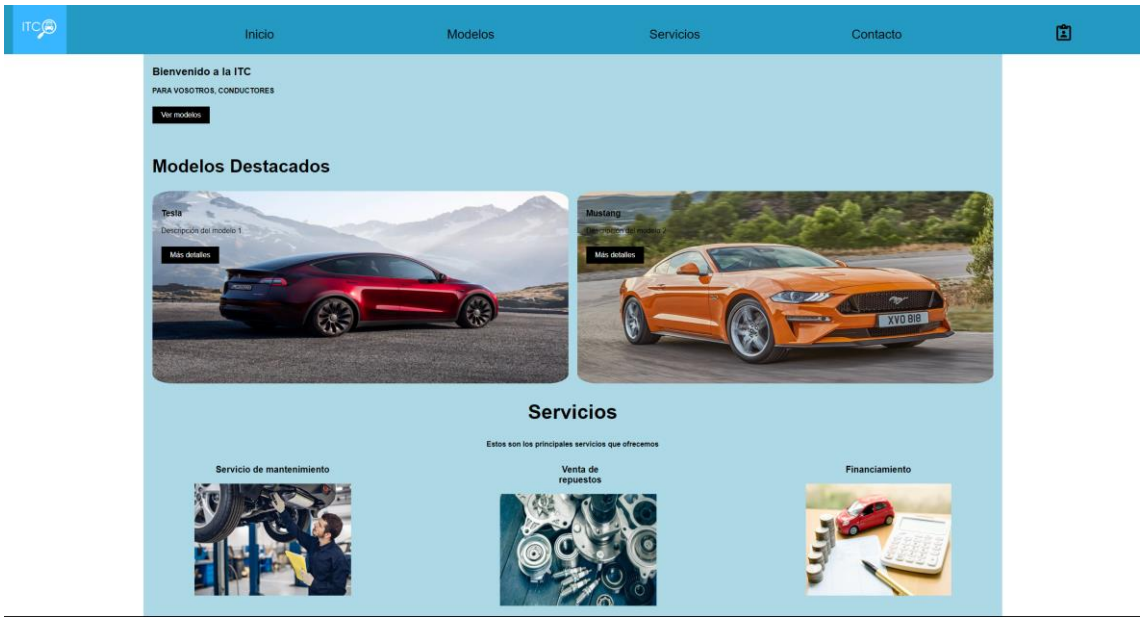
Hemos aplicado las mismas restricciones que aparecen en la creación de las tablas en el apartado de Base de Datos, con esto conseguimos una doble capa de seguridad, de forma que estamos comprobando y validando los datos, tanto en Base de Datos como en Programación y se controlan todos los errores que puedan suceder. Además, hemos añadido nuevas restricciones que solo están aplicada es el apartado de Programación estas son: Un trabajador no puede tener más de 4 citas en el mismo intervalo de tiempo, no puede haber más de 8 citas en el mismo intervalo de tiempo y un vehículo no puede tener la misma cita el mismo día.

Hemos añadido las opciones de poder exportar informes a HTML y JSON, y trabajadores a CSV. Cabe destacar que cuando se exportan todos los trabajadores o todos los informes, el fichero se sobrescribe o se crea en caso de que no existiera previamente. Por otro lado, mientras que cuando se exporta solo un informe, a JSON o HTML, hay un sistema que crea nuevos informes con un nombre distinto cada vez, de esta forma se tiene registro de todos los informes que se han creado.

En el apartado de la interfaz, hemos incluido una pantalla de acerca de, en esta se ha añadido el logo del proyecto, una breve descripción de las funcionalidades que tiene la empresa y enlaces a las respectivas páginas de GitHub de cada miembro del grupo.

La interfaz sigue el esquema de colores de la pagina web, este estilo se le ha aplicado directamente desde el SceneBuilder.





Estimación precio del desarrollo:

91 horas en total.

91horas entre 3 personas = 30.3 horas trabajadas por persona.

30.3 horas / 40 horas semanales = 76% de la semana trabajado por persona.

Con un sueldo base de aproximadamente 1600€/mes.

1600€ / 4 (4 semanas en un mes) = 400€/semana.

400€ * 0.76 * 3 personas = 912€.

Conclusión

Durante el desarrollo de este proyecto de programación, hemos tenido la oportunidad de sumergirnos en la experiencia de un "sprint" empresarial. Desde el primer día, nos vimos en la necesidad de organizarnos eficientemente y asignar las tareas de manera adecuada, con el objetivo de completar el proyecto en el tiempo establecido.

A medida que avanzaba el proyecto, el grupo pudo apreciar los beneficios del trabajo en equipo y la importancia de una buena organización. Al trabajar en conjunto, pudimos aprovechar las fortalezas individuales de cada miembro y colaborar de manera efectiva para alcanzar nuestros objetivos. La comunicación constante y la distribución adecuada de las responsabilidades nos permitieron mantenernos en el camino correcto y asegurar un progreso constante. Además, la experiencia de este proyecto nos permitió adquirir una comprensión más profunda del valor de la planificación y la gestión del tiempo.

En resumen, este proyecto de programación nos ha brindado un valioso primer acercamiento a la dinámica de un "sprint" empresarial. Hemos experimentado los beneficios del trabajo en equipo y la importancia de una buena organización. La planificación y la gestión efectiva del tiempo fueron fundamentales para el logro de nuestros objetivos. Estas lecciones aprendidas sin duda nos serán útiles en futuros proyectos, ya que hemos adquirido una base sólida para enfrentar desafíos similares con confianza y eficacia.