

ITV AEA

Angel Maroto Chivite
Emma Fernández
Adrián Potenciano

Índice

Contenido

Índice	1
1. Introducción	2
1.1. Descripción del proyecto	2
1.2. Análisis económico y tecnológico	2
1.3. Objetivos	4
2. Especificación de requisitos	4
2.1. Requisitos funcionales	4
2.2. Requisitos no funcionales	6
2.3. Requisitos de información	8
3. Planificación y gestión de tareas	11
3.1. Filosofía GitFlow	11
3.2. Uso de Kanban	12
3.3. Tareas y responsabilidades del proyecto	13
4. Diseño de la base de datos	13
4.1. Modelo E/R	13
4.2. Modelo físico	15
5. Diseño de la página web	16
5.1. Logo y eslogan	16
5.2. Creación y especificaciones	16
6. Diagramas y diseño de aplicación de escritorio	17
6.1. Diagramas de casos de uso	17
6.2. Diagrama de clases	22
6.3. Diagrama de secuencias	23
7. Creación de aplicación de escritorio	29
7.1. Diseño, patrones y arquitectura	29
7.2. Arquitectura basada en principios SOLID	30
7.3. Uso de Railway Oriented Programming	31
7.4. Interfaz de escritorio	31
7.5. Pruebas del software	32
8. Detalles y conclusiones	33

1. Introducción

1.1. Descripción del proyecto

El proyecto consiste en desarrollar un sistema gestor informático de la Inspección Técnica de Vehículos (ITV).

Se implementará una infraestructura compuesta por una página web, un sistema centralizado de almacenamiento de información y una aplicación de escritorio para la gestión de citas. El objetivo principal es optimizar y automatizar el proceso de inspección de vehículos, facilitando la gestión de citas, el registro de datos de vehículos y propietarios, así como la generación de informes.

La fecha de exposición corresponde al 30 de mayo de 2023 desde las 08:30.

Realizado por 3 integrantes: Emma Fernández, Adrián Potenciano y Ángel Chivite.

1.2. Análisis económico y tecnológico

Teniendo en cuenta un salario bruto de 3 juniors (1600€) al haber trabajado 80h entre los 3 es decir 2 semanas el presupuesto necesario sería de 800€ sin tener en cuenta el salario de los superiores y utilizando licencias libres.

Área	Herramientas
Entornos	IntelliJ
Web	Html, Css, JavaScript
Base de datos	MariaDb, DataGrip
Programación	Kotlin
Interfaces	JavaFX, SceneBuilder
Dependencias	Mockito, JUnit, GSON, Koin, Logger, Results
Diagramas	Lucidchart, Draw.io

1. **Entornos - IntelliJ**: Se ha utilizado como entorno de desarrollo integrado (IDE) debido a su amplia gama de características y herramientas avanzadas que facilitan el desarrollo de aplicaciones.
2. **Web - HTML, CSS y JavaScript**: Estas tres tecnologías web son fundamentales para desarrollar la parte frontal de la aplicación siendo las más punteras del momento.

HTML proporciona la estructura básica de las páginas web, mientras que **CSS** se encarga del diseño y la apariencia visual.

JavaScript permite la interacción dinámica con los elementos de la página, brindando una experiencia de usuario más fluida y enriquecida.

3. Base de datos - **MariaDB** y **DataGrip**: Se ha optado por **MariaDB** como el sistema de gestión de bases de datos (SGBD) debido a su compatibilidad con **MySQL** y su enfoque en la escalabilidad y el rendimiento.

DataGrip, una herramienta de JetBrains, se ha utilizado como cliente de base de datos para administrar y consultar la base de datos de manera eficiente.

4. Programación - **Kotlin**: Se ha elegido Kotlin como lenguaje de programación principal debido a su compatibilidad con **Java** y su sintaxis más concisa y expresiva.
5. Interfaces - **JavaFX** y **SceneBuilder**: JavaFX, junto con la herramienta SceneBuilder, se ha utilizado para el desarrollo de las interfaces de usuario. JavaFX ofrece una amplia gama de componentes y controles gráficos, así como una arquitectura moderna para la creación de interfaces de usuario ricas y atractivas.

SceneBuilder facilita la creación y el diseño visual de las interfaces de forma intuitiva.

6. Dependencias - Mockito, JUnit, GSON, Koin, Logger, Results: Estas dependencias se han utilizado para mejorar el desarrollo de la aplicación.

Mockito y **JUnit** se utilizan para realizar pruebas unitarias y garantizar la calidad del código.

GSON se encarga de la serialización y deserialización de objetos JSON.

Koin es un framework de inyección de dependencias utilizado para facilitar la gestión de las dependencias en el proyecto.

Logger proporciona un registro de eventos y resultados relevantes para facilitar la depuración. Results es una biblioteca utilizada para estructurar y representar los resultados de las operaciones.

7. Diagramas - **Lucidchart** y Draw.io: Estas herramientas de diagramación se han utilizado para crear diagramas de casos de uso, diagramas de clases, diagramas de secuencia y otros diagramas necesarios para el diseño y la comprensión del sistema.

1.3. Objetivos

- Desarrollar una página web atractiva y funcional que promueva los servicios ofrecidos por la ITV y facilite la solicitud de citas.
- Diseñar un sistema gestor centralizado de almacenamiento de información que permita el registro y gestión eficiente de datos de relevantes de la ITV.

- Crear una aplicación de escritorio que simplifique la gestión de citas e inspecciones.
- Aprender la dinámica de trabajo en equipo y las dificultades que conllevan, y empleo de tecnologías que nos facilite la organización.

2. Especificación de requisitos

Al tener sistemas gestores independientes para cada modelo del proyecto, se logra una clara separación de responsabilidades.

Cada sistema se encarga de manejar y administrar un dominio específico, lo que facilita la comprensión y el mantenimiento del código, y sobre todo una mayor escalabilidad.

2.1. Requisitos funcionales

Sistema gestor de citas (mediante el actor <<Trabajador>>):

Requisito Funcional	Descripción
RF-01	Autenticación/Login de Trabajador
RF-02	Crear cita seleccionando un inspector y añadir datos del vehículo y su propietario
RF-03	Eliminar cita de la base de datos
RF-04	Actualizar cita de la base de datos
RF-05	Exportar cita + informe de la cita
RF-06	Exportar todas las citas + informes a JSON

Sistema gestor de trabajadores (mediante el actor <<Admin>>):

Requisito Funcional	Descripción
RF-07	Autenticación/Login de Admin
RF-08	Guardar trabajador en la base de datos
RF-09	Eliminar trabajador de la base de datos
RF-10	Actualizar trabajador en la base de datos
RF-11	Realizar trabajador del inspector
RF-12	Exportar todos los trabajadores a CSV

Sistema gestor de vehículos (mediante el actor <<Admin>>):

Requisito Funcional	Descripción
RF-13	Autenticación/Loggin de Admin
RF-14	Crear un vehículo
RF-15	Actualizar vehículo
RF-16	Eliminar vehículo de la base de datos

Sistema gestor de propietarios (mediante el actor <<Admin>>):

Requisito Funcional	Descripción
RF-17	Autenticación/Loggin de Admin
RF-18	Crear un propietario
RF-19	Actualizar propietario
RF-20	Eliminar propietario de la base de datos

Sistema gestor de estaciones (mediante el actor <<Super-Admin>>):

Requisito Funcional	Descripción
RF-21	Autenticación/Loggin de Super-Admin
RF-22	Guardar estación en la base de datos
RF-23	Actualizar estación en la base de datos
RF-24	Eliminar estación de la base de datos

2.2. Requisitos no funcionales

Sistema gestor de citas (mediante el actor <<Trabajador>>):

Requisito No Funcional	Descripción
RNF-01.1	La creación de una cita se realiza mediante un inspector con menos de 5 citas asociadas, y se deberán introducir los datos del vehículo y su propietario, generándose un informe el cual estará vacío con el atributo "no apto"
RNF-01.2	Una cita se podrá crear si hay menos de 8 citas creadas en el intervalo de 30 minutos
RNF-02	La búsqueda de la cita se podrá realizar mediante la matrícula del vehículo, tipo de vehículo y la fecha de la última revisión
RNF-03	Al actualizar una cita, se selecciona si es apto o no apto, asignando los datos del informe, y actualizando el informe que se encuentra vacío
RNF-04	Al exportar un informe de la cita, podremos elegir el formato de exportación a JSON o Markdown, tendrá los datos del propietario del vehículo, el vehículo, el trabajador que gestione la inspección, datos del informe de la inspección y datos de la cita
RNF-05	La base de datos está creada en MariaDB
RNF-06	La lógica de la aplicación está realizada con Kotlin

Sistema gestor de trabajadores (mediante el actor <<Admin>>):

Requisito No Funcional	Descripción
RNF-07	La búsqueda se realizará mediante el DNI
RNF-08	La base de datos está creada en MariaDB
RNF-09	La lógica de la aplicación está realizada con Kotlin

Sistema gestor de vehículos (mediante el actor <<Admin>>):

Requisito No Funcional	Descripción
RNF-10	La búsqueda se realizará mediante la matrícula
RNF-11	La base de datos está creada en MariaDB
RNF-12	La lógica de la aplicación está realizada con Kotlin

Sistema gestor de propietarios (mediante el actor <<Admin>>):

Requisito No Funcional	Descripción
RNF-13	La búsqueda se realizará mediante el DNI
RNF-14	La base de datos está creada en MariaDB
RNF-15	La lógica de la aplicación está realizada con Kotlin

Sistema gestor de estaciones (mediante el actor <<Super-Admin>>):

Requisito No Funcional	Descripción
RNF-16	La búsqueda se realizará mediante el ID
RNF-17	La base de datos está creada en MariaDB
RNF-18	La lógica de la aplicación está realizada con Kotlin

2.3. Requisitos de información

Sistema gestor de citas (mediante el actor <<Trabajador>>):

Requisito de Información	Descripción
RI-01	El ID será la clave primaria
RI-02	El estado corresponderá a “Apto” o “No apto”
RI-03	La fecha y hora tendrá formato: yyyy-MM-ddTHH:mm:ss.SSS
RI-04.1	Aptitud del frenado del vehículo, decimal formato: 0.00
RI-04.2	Aptitud de la contaminación del vehículo, decimal formato: 0.00
RI-04.3	La fecha del informe tendrá un formato: yyyy-MM-dd
RI-04.4	Aptitud del interior del vehículo, mediante apto o no apto
RI-04.5	Aptitud de luces del vehículo, mediante apto o no apto
RI-04.6	Aptitud general del vehículo, mediante apto o no apto
RI-05	Tendrá como referencia al usuario del trabajador que gestionará la cita
RI-06	Tendrá como referencia a la matrícula del vehículo que se realizará la inspección

Sistema gestor de trabajadores (mediante el actor <<Admin>>):

Requisito de Información	Descripción
RI-07	El trabajador como clave primaria será el nombre de usuario y será único
RI-08	El trabajador en el email será único
RI-09	El trabajador tendrá una contraseña cifrada
RI-10	El trabajador tendrá la fecha de contratación en formato yyyy-mm-dd
RI-11.1	Administración (1650€)
RI-11.2	Electricidad (1800€)
RI-11.3	Motor (1700€)
RI-11.4	Mecánica (1600€)
RI-11.5	Interior (1750€)
RI-11.6	Responsable (+1000€ sobre el salario de la especialidad que tenga)
RI-12	Cada trabajador no atenderá más de 4 citas por intervalo de 30 minutos
RI-13	El trabajador tendrá como referencia el ID de la estación en la que trabaja
RI-14	El trabajador tendrá un campo para saber si es el gerente de la estación o no

Sistema gestor de vehículos (mediante el actor <<Admin>>):

RI-15	La matrícula debe ser una cadena que favorezca la expresión "1111XXX"
RI-16	La marca debe ser una cadena de texto
RI-17	El modelo debe ser una cadena de texto
RI-18	La fecha de matriculación debe tener formato yyyy-mm-dd
RI-19	La fecha de revisión debe tener formato yyyy-mm-dd
RI-20	El tipo de motor podrá ser uno de los siguientes valores: Gasolina, Diésel, Híbrido o Eléctrico
RI-21	El tipo de vehículo podrá ser uno de los siguientes valores: Turismo, Furgoneta, Camión o Motocicleta

Sistema gestor de propietarios (mediante el actor <<Admin>>):

Requisito de Información	Descripción
RI-23	El DNI será una cadena de texto como clave primaria
RI-24	El nombre será una cadena de texto
RI-25	El apellido será una cadena de texto
RI-26	El correo será una cadena con el formato xxx@xxx.xx
RI-27	El teléfono será una cadena con el formato 666 666 666

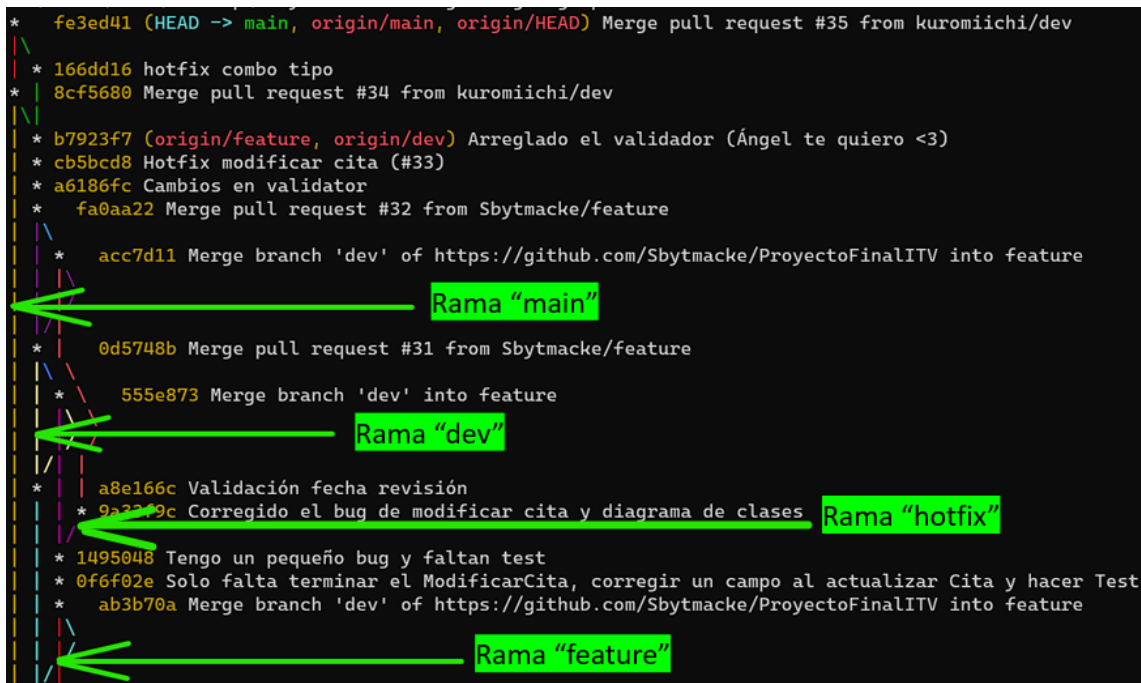
Sistema gestor de estaciones (mediante el actor <<Super-Admin>>):

Requisito de Información	Descripción
RI-28	El ID es una clave primaria
RI-29	El nombre es una cadena de texto
RI-30	La dirección es una cadena de texto
RI-31	El correo tiene el formato xxx@xxx.xx
RI-32	El teléfono es una cadena con el formato 666 666 666

3. Planificación y gestión de tareas

3.1. Filosofía GitFlow

Durante el desarrollo del proyecto, se adoptó la filosofía GitFlow como metodología de trabajo. Para conseguir un enfoque basado en la colaboración en equipo y el control de versiones.

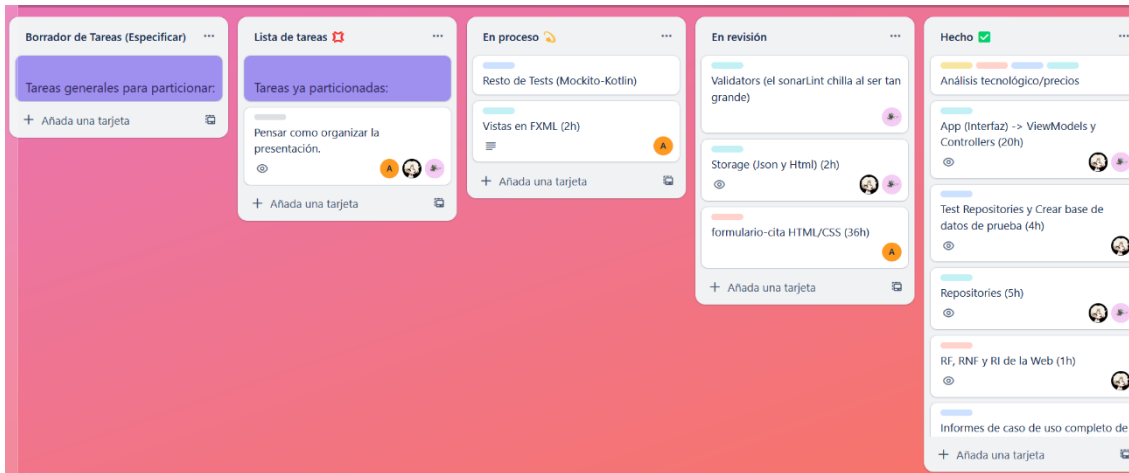


En nuestro caso, utilizamos las ramas de GitFlow de la siguiente manera:

- **Rama de producción:** La rama "main" se utilizó como la rama de producción final donde se fusionan las características de la rama de desarrollo "dev" cuando todo el funcionamiento o planteamiento es correcto.
- **Rama de desarrollo:** La rama "dev" se utilizó como la rama principal donde se integraban las nuevas. Las ramas de desarrollo de características individuales se crearon a partir de la rama "feature" y se fusionaron a esta rama de desarrollo común "dev" una vez completadas.
- **Rama de característica:** La rama "feature" para desarrollar nuevas funcionalidades o características del software de forma aislada y controlada, es decir de cada miembro del grupo. Una vez finalizado el trabajo fusionamos a "dev".
- **Rama de corrección de errores:** Las ramas de corrección "hotfix" se utilizaron para abordar problemas o errores puntuales. Estas ramas se han creado a partir de la rama correspondiente donde se encontrará el error, se solucionaron los problemas y luego se fusionaron a la rama del error.

3.2. Uso de Kanban

Para la gestión visual de tareas y el seguimiento del progreso del proyecto, implementamos un tablero Kanban mediante “**Trello**”. Utilizamos columnas representando los diferentes estados de una tarea, como:



- **Borrador de tareas:** Corresponde a las tareas globales, que posteriormente tendríamos que particionar y especificar en detalle.
- **Lista de tareas:** Corresponde a tareas por realizar, especificando en detalle que puntos hay que completar.
- **En proceso:** El trabajo que está actualmente completándose.
- **En revisión:** Una vez el trabajo esté realizado, siempre tendremos a otro miembro del equipo que revise el trabajo, para incorporarlo a la columna de “Hecho”.
- **Hecho:** Donde el trabajo se haya completado

A medida que avanzaba el proyecto, movíamos las tarjetas a través de las columnas para indicar su estado actual. Esto nos ayudó a visualizar de manera clara y concisa el flujo de trabajo, pudimos mantener un ritmo constante y evitar la sobrecarga de trabajo.

3.3. Tareas y responsabilidades del proyecto

Durante la fase de planificación, se asignaron tareas específicas a cada miembro del equipo de acuerdo con sus habilidades y conocimientos.

Para no dejar a todos los miembros sin entender los distintos aspectos que nos habíamos dividido, ideamos en el tablero Kanban el apartado de revisión, así podríamos entender todos sobre el trabajo realizado de otro compañero.

No nos centramos en designar distintos roles dentro del grupo, ya que partimos de un conocimiento muy similar.

4. Diseño de la base de datos

Partimos de un planteamiento donde el nexo es la cita, a través de una cita podremos conseguir información de toda la base de datos asociada a esa cita, es decir, las claves primarias del resto de entidades asociadas a la cita acabarán siendo claves ajenas de dicha cita.

4.1. Modelo E/R

Relaciones en el sistema gestor de las entidades "Citas":

- Con la entidad "Trabajador" mediante una relación "muchos a uno" (muchas citas son gestionadas por un trabajador).
- Con la entidad "Vehículo" mediante una relación "uno a uno" (cada cita está asociada a un único vehículo).
- Con la entidad "Informe" mediante una relación "uno a uno" (cada cita está asociada a un único informe).

Relaciones en el sistema gestor de las entidades "Trabajador":

- Con la entidad "Estación" mediante una relación "muchos a uno" (muchos trabajadores pueden pertenecer a una misma estación).
- Con la entidad "Cita" mediante una relación "uno a muchos" (un trabajador puede gestionar muchas citas).

Relaciones en el sistema gestor de las entidades "Vehículos":

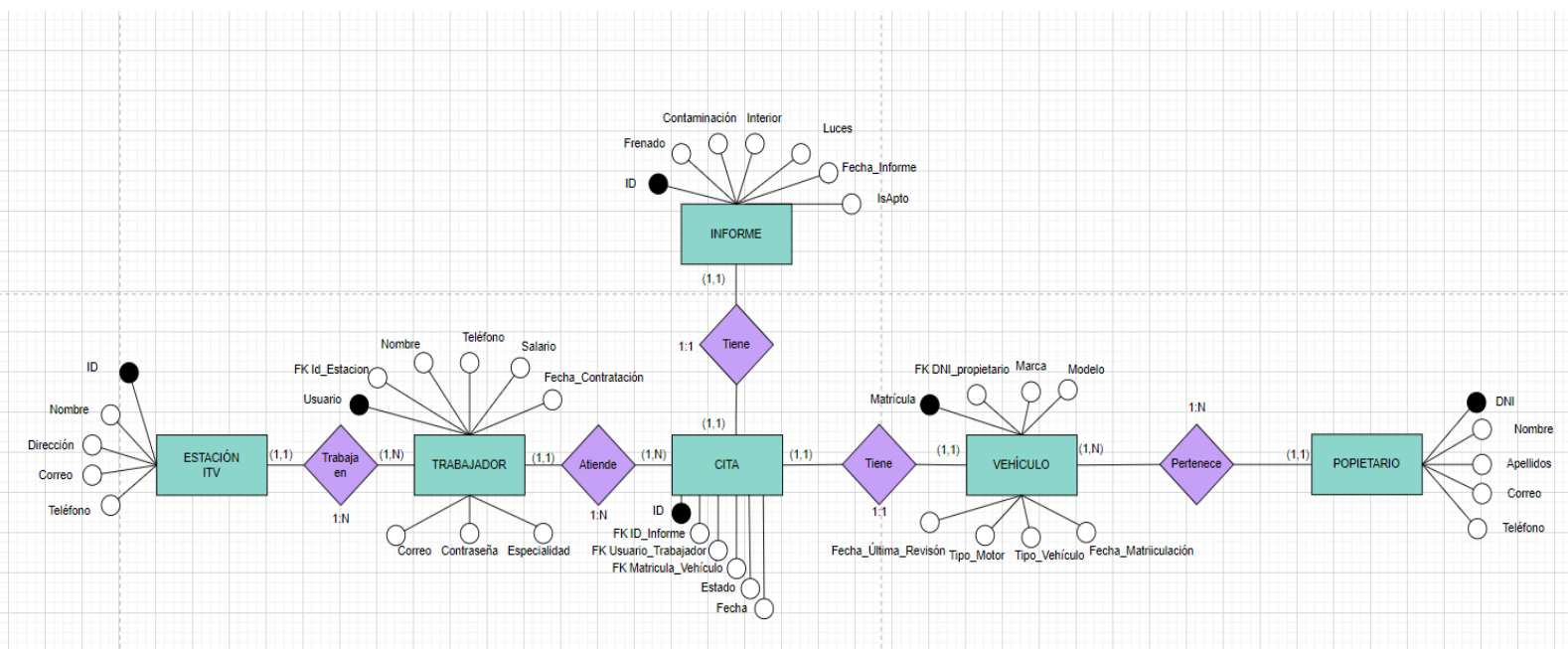
- Con la entidad "Propietario" mediante una relación "muchos a uno" (muchos vehículos pueden pertenecer a un mismo propietario).
- La entidad "Vehículo" está relacionada con la entidad "Cita" mediante una relación "uno a uno" (cada vehículo puede tener una única cita asociada).

Relaciones en el sistema gestor de las entidades "Propietarios":

- La entidad "Propietario" está relacionada con la entidad "Vehículo" mediante una relación "uno a muchos" (un propietario puede tener varios vehículos asociados).

Relaciones en el sistema gestor de las entidades "Estacion ITV":

- Con la entidad "Trabajador" mediante una relación "uno a muchos" (una estación puede tener varios trabajadores asociados).



4.2. Modelo físico

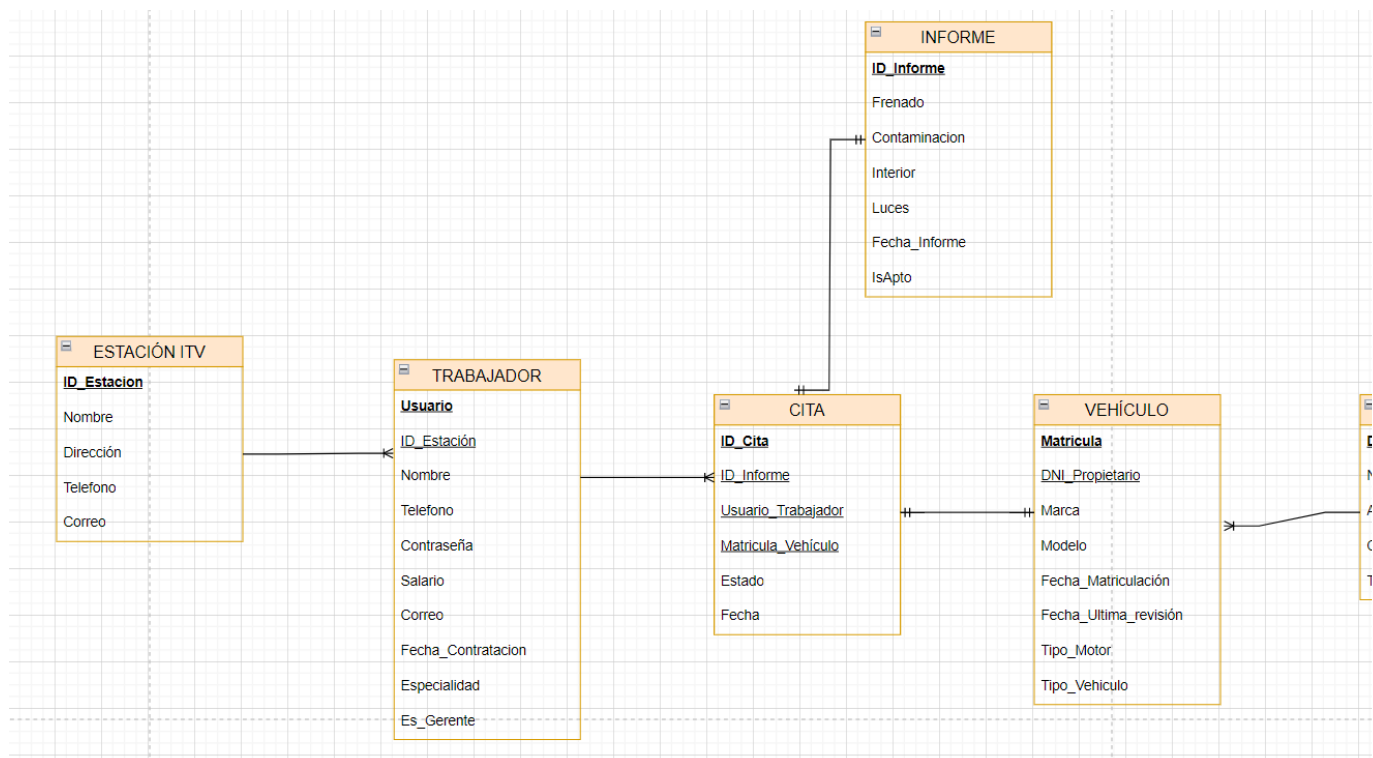
En nuestro modelo de base de datos, hemos tenido en cuenta la posibilidad de que las entidades existan incluso si no están asociadas a una estación en particular, esto nos permite mantener un registro completo de los datos, incluso en el caso de que una estación haya cerrado.

En cuanto a las **restricciones**, todas las entidades excepto el Trabajador no permiten nulabilidad en sus campos de claves foráneas, y es importante tener en cuenta el orden en el que se deben realizar las operaciones de agregación, modificación o eliminación en la base de datos para mantener la integridad de los datos.

Siguiendo un **flujo lógico**, recomendamos realizar las siguientes operaciones en este orden:

1. Disponer siempre en primer lugar de algún “Trabajador”, permitimos que su ID de estación sea nula en caso de cerrar alguna estación.
2. Agregar, modificar o eliminar registros en la entidad "Propietario".
3. A continuación, se pueden agregar, modificar o eliminar registros en la entidad "Vehículo".
4. Posteriormente, se deben gestionar las operaciones relacionadas con la entidad "Informe".
5. Por último, realizar las operaciones en la entidad "Cita".

Este orden garantiza que las restricciones y relaciones entre las entidades se mantengan consistentes y se eviten posibles problemas de integridad de datos.



5. Diseño de la página web

5.1. Logo y eslogan

Significado del logo: El logo de la AEA consiste en un diseño en color azul que corresponden a "Asociación Española Automovilística".

Significado del eslogan: El eslogan "Seguridad en cada inspección ITV, tu confianza ante todo". Destaca el enfoque prioritario en la seguridad de los vehículos y la importancia de generar confianza en los propietarios de estos.

5.2. Creación y especificaciones

La elección del diseño se basa en varios aspectos clave:

1. **Usabilidad:** El diseño se centra en brindar una experiencia de usuario intuitiva y agradable. Se utilizan elementos visuales como íconos y enlaces para facilitar la navegación y proporcionar acceso rápido a diferentes secciones de la página.
2. **Estilo:** El diseño refleja un estilo moderno y profesional.
3. **Estructura:** Se han utilizado contenedores y elementos de diseño para separar y agrupar visualmente diferentes secciones de la página, como el encabezado, el cuerpo y el banner inferior.
4. **Experiencia del usuario:** incluimos elementos interactivos, esto ayuda a captar la atención del usuario y proporcionar detalles relevantes de manera dinámica.
5. **Adaptabilidad:** El diseño ha sido implementado utilizando estilos CSS y scripts JavaScript, lo que permite una mayor flexibilidad y adaptabilidad a diferentes dispositivos y tamaños de pantalla.

En resumen, este diseño fue seleccionado para crear un formulario de registro atractivo, intuitivo y fácil de usar para los usuarios de la ITV. Se enfoca en proporcionar una experiencia positiva, reflejar la identidad de la marca y facilitar la interacción con la información relevante.

6. Diagramas y diseño de aplicación de escritorio

6.1. Diagramas de casos de uso

Diagrama de Casos de Uso - Sistema Gestor de Citas

1. Actores:
 - Trabajador: Representa al personal encargado de administrar las citas y brindar asistencia a los usuarios en la estación de ITV.
2. Disponemos de información en detalle sobre los casos de uso solicitados en la carpeta “Informe de Casos de Uso”



Diagrama de Casos de Uso - Sistema Gestor de Propietario

1. Actores:

- Admin: Representa al administrador encargado de gestionar los propietarios, hemos supuesto los clientes que tengan intención de realizar una cita son clientes potenciales y no deseamos realizar un cambio sustancial en ellos.

Respecto a la “modificación de un propietario por confusión”, al modificar una cita en el sistema Gestor de Citas, se puede modificar al propietario ahí mismo.

- ### 2. Disponemos de información en detalle sobre los casos de uso solicitados en la carpeta “Informe de Casos de Uso”

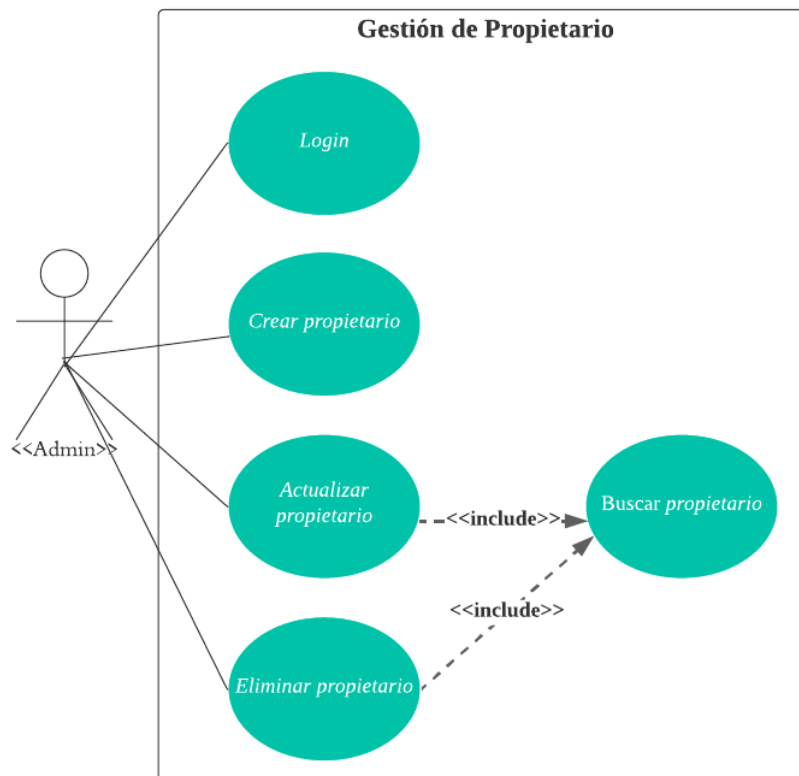


Diagrama de Casos de Uso - Sistema Gestor de Vehículo

1. Actores:

- Admin: Representa al administrador encargado de gestionar los vehículos, hemos supuesto los clientes que tengan intención de realizar una cita son clientes potenciales y no deseamos realizar un cambio sustancial en sus vehículos.

Respecto a la “modificación de un vehículo por confusión”, al modificar una cita en el sistema Gestor de Citas, se puede modificar al vehículo ahí mismo.

- ### 2. Disponemos de información en detalle sobre los casos de uso solicitados en la carpeta “Informe de Casos de Uso”

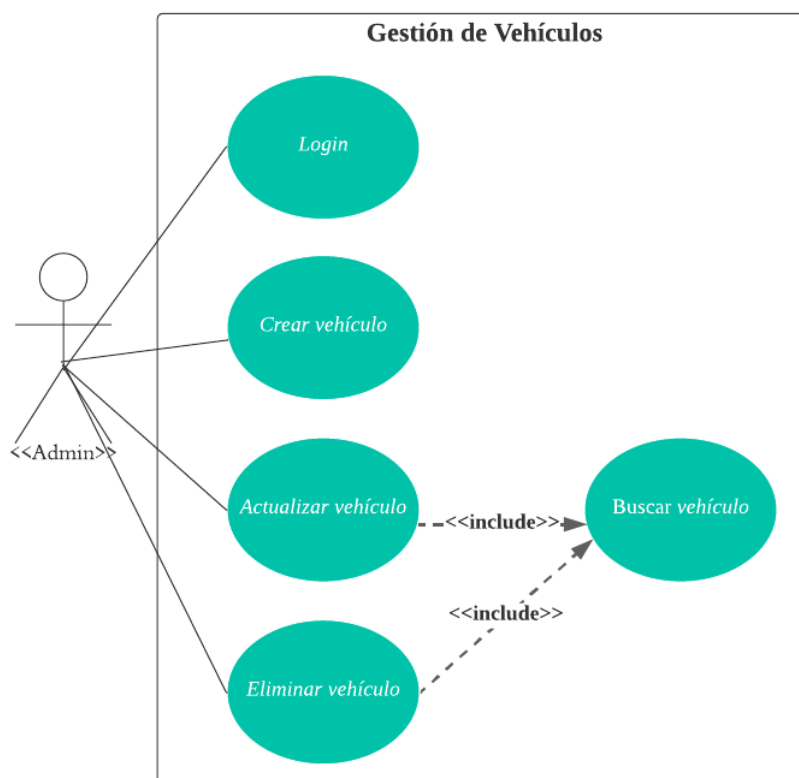


Diagrama de Casos de Uso - Sistema Gestor de Trabajador

1. Actores:
 - Admin: Representa al administrador encargado de gestionar los trabajadores.
2. Disponemos de información en detalle sobre los casos de uso solicitados en la carpeta “Informe de Casos de Uso”

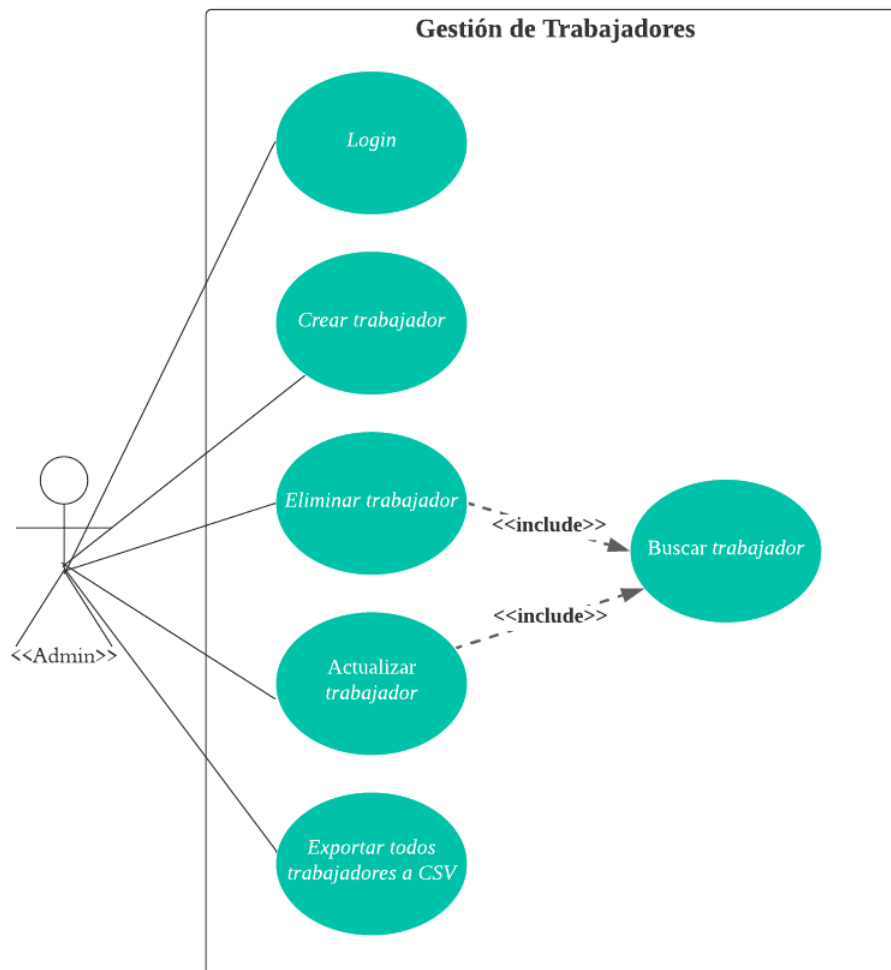
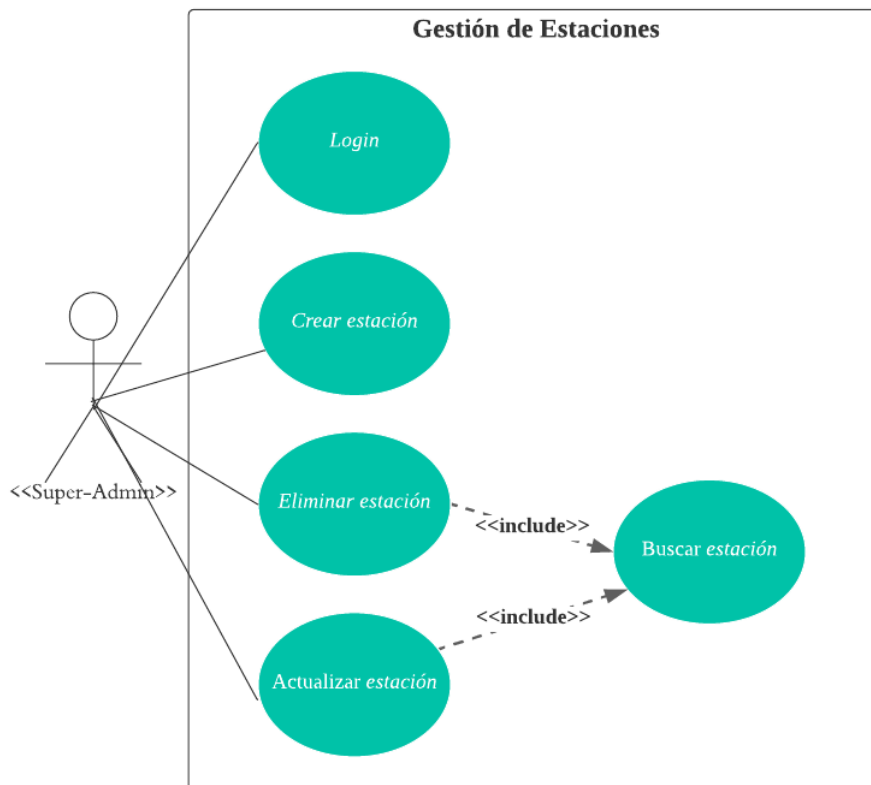


Diagrama de Casos de Uso - Sistema Gestor de Estación

1. Actores:
 - Super-Admin: Representa al super-administrador encargado de gestionar las estaciones.
2. Disponemos de información en detalle sobre los casos de uso solicitados en la carpeta “Informe de Casos de Uso”



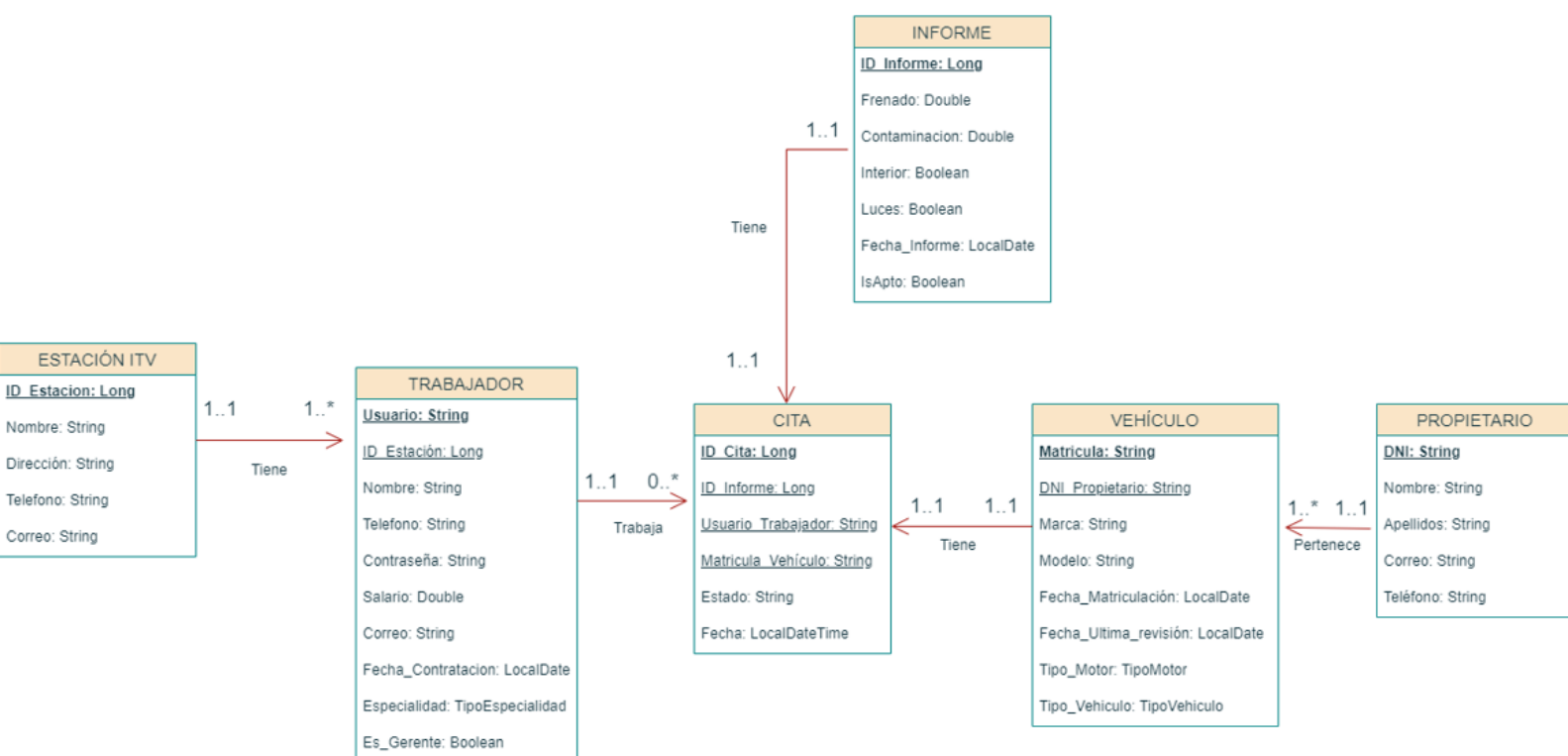
6.2. Diagrama de clases

La navegabilidad la hemos establecido a través de las claves foráneas en las clases. Estas relaciones nos permiten acceder a información relacionada y realizar consultas entre las entidades.

Por lo que la clase “Cita” en nuestro nexo para alcanzar cualquier información.

Navegabilidad:

- Desde la clase Cita, se puede acceder al informe, al Trabajador asignado y al Vehículo relacionados con la cita.
- Desde la clase Trabajador, se puede acceder a la Estación a la que pertenece.
- Desde la clase Vehículo, se puede acceder al Propietario del vehículo.



6.3. Diagrama de secuencias

Crear cita / Dar alta cita:

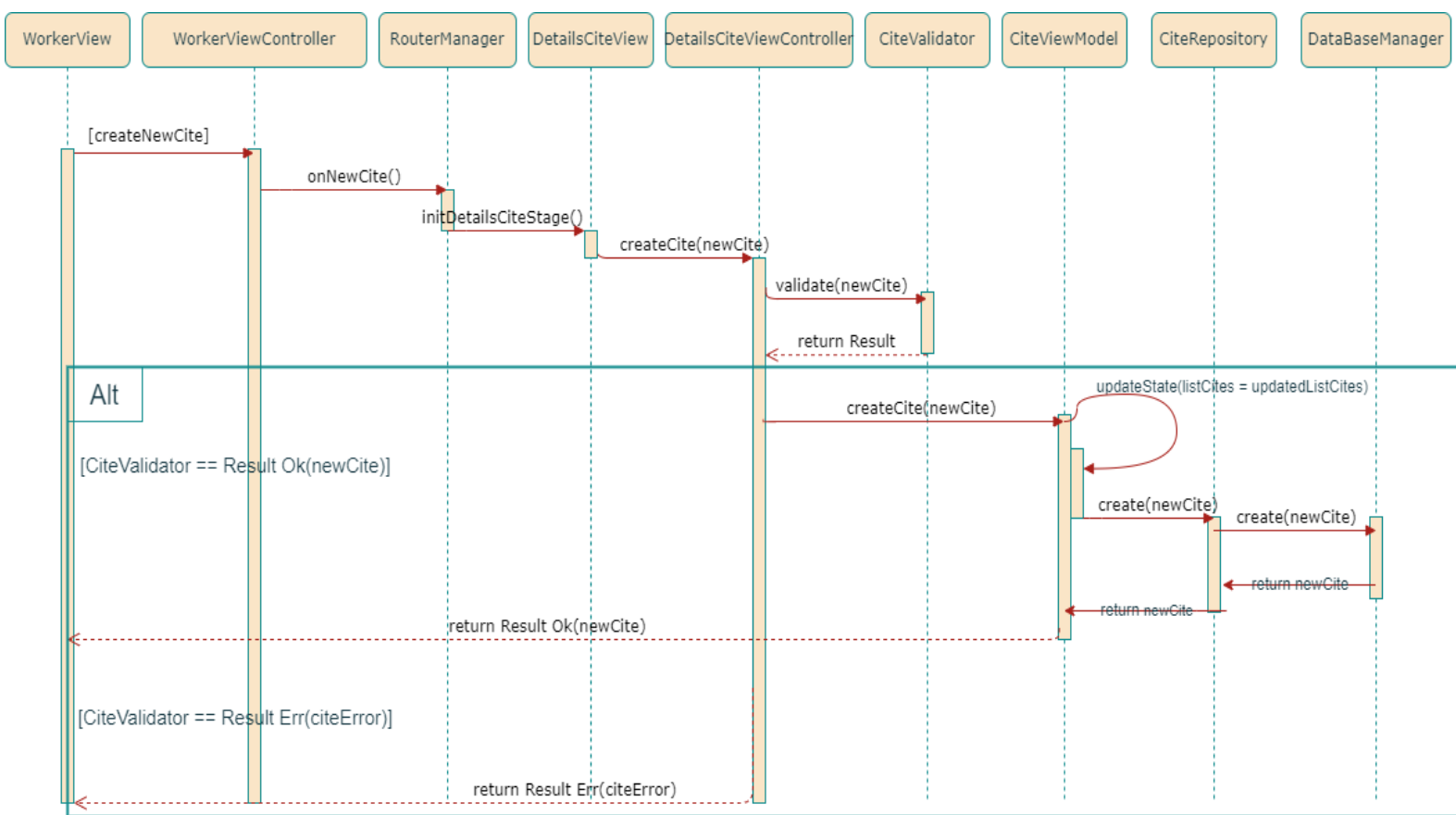
1. WorkerView y WorkerViewController:
 - WorkerView representa la interfaz de usuario que permite al usuario ingresar los detalles necesarios para crear una cita.
 - WorkerViewController es responsable de manejar la lógica y la interacción del usuario en la WorkerView. Actúa como el intermediario entre la vista y el ViewModel.
2. RouterManager:
 - RouterManager es responsable de gestionar la navegación entre las diferentes vistas de la aplicación. En el caso de crear una cita, puede ser utilizado para redirigir al usuario desde la vista de detalles del trabajador hacia la vista de creación de cita.
3. DetailsCitaView y DetailsCitaViewController:
 - DetailsCitaView representa la vista que muestra los detalles de una cita existente. Puede ser utilizada para que el usuario seleccione una cita existente y acceda a su información.
 - DetailsCitaViewController es responsable de manejar la lógica y la interacción del usuario en la DetailsCitaView.
4. CitaValidator:
 - CitaValidator se encarga de validar los datos ingresados por el usuario antes de crear una cita.
5. CitaViewModel:
 - CitaViewModel actúa como el intermediario entre la vista y el modelo de datos. Recibe los datos ingresados por el usuario desde la WorkerView y los procesa, utilizando el CitaValidator para validarlos.
 - Además de la validación, el CitaViewModel contiene la lógica de negocio adicional relacionada con la creación de citas, como la comunicación con el CitaRepository.

6. CitaRepository:

- CitaRepository es responsable de manejar las operaciones relacionadas con la creación y gestión de citas en el sistema. Interactúa con el DataManager para acceder y modificar los datos almacenados en la base de datos.

7. DataBaseManager:

- DataManager actúa como la capa de acceso a datos de la aplicación. Proporciona métodos y funcionalidades para realizar operaciones en la base de datos.



Actualizar trabajador:

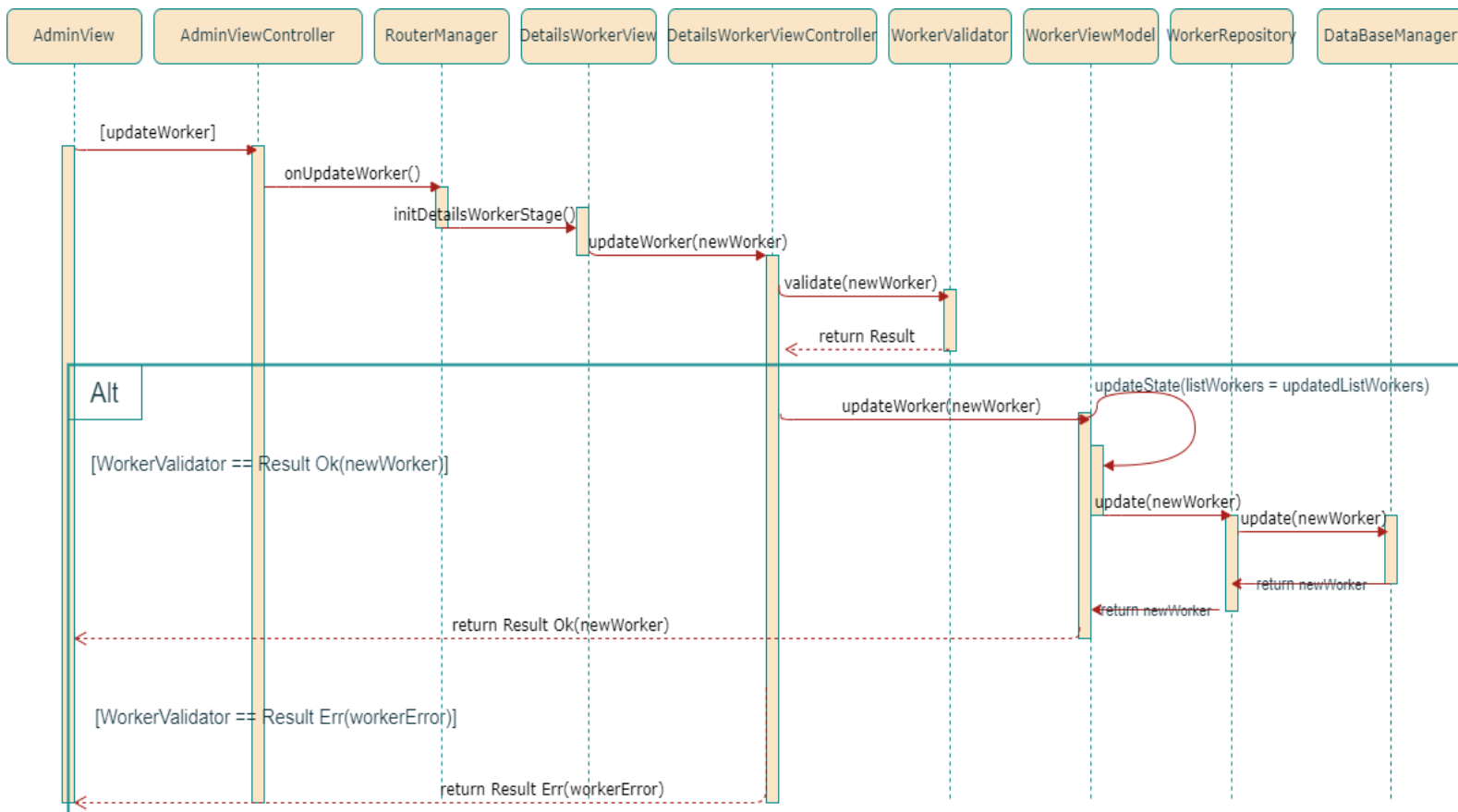
1. AdminView y AdminViewController:
 - AdminView representa la interfaz de usuario que permite al administrador ingresar los datos necesarios para actualizar un trabajador.
 - AdminViewController se encarga de manejar la lógica y la interacción del usuario en la AdminView. Actúa como intermediario entre la vista y el ViewModel.
2. RouterManager:
 - RouterManager es responsable de gestionar la navegación entre las diferentes vistas de la aplicación. En el caso de actualizar un trabajador, puede ser utilizado para redirigir al administrador desde la vista de detalles del trabajador hacia la vista de actualización de trabajador.
3. DetailsWorkerView y DetailsWorkerViewController:
 - DetailsWorkerView representa la vista que muestra los detalles de un trabajador existente. Es utilizado para que el administrador seleccione un trabajador existente y acceda a su información para realizar modificaciones.
 - DetailsWorkerViewController es responsable de manejar la lógica y la interacción del usuario en la DetailsWorkerView.
4. WorkerValidator:
 - WorkerValidator se encarga de validar los datos ingresados por el administrador antes de actualizar un trabajador.
5. WorkerViewModel:
 - WorkerViewModel actúa como el intermediario entre la vista y el modelo de datos. Recibe los datos ingresados por el administrador desde la AdminView y los procesa, utilizando el WorkerValidator para validarlos.
 - Además de la validación, el WorkerViewModel contiene la lógica de negocio adicional relacionada con la actualización de trabajadores, como la comunicación con el WorkerRepository.

6. WorkerRepository:

- WorkerRepository es responsable de manejar las operaciones relacionadas con la actualización y gestión de trabajadores en el sistema. Interactúa con el DataManager para acceder y modificar los datos almacenados en la base de datos.

7. DataBaseManager:

- DataManager actúa como la capa de acceso a datos de la aplicación. Proporciona métodos y funcionalidades para realizar operaciones en la base de datos.



Exportar cita:

Situación 1: Exportar a JSON

1. WorkerView y WorkerViewController:
 - Estos componentes representan la interfaz de usuario y su controlador asociado que permiten a los trabajadores interactuar con la aplicación y desencadenar la exportación de la cita a JSON.
2. RouterManager:
 - El RouterManager se encarga de la navegación entre las diferentes vistas de la aplicación. En este caso, puede ser utilizado para redirigir al trabajador desde la WorkerView a la vista de DecisionExportView.
3. DecisionExportView y DecisionExportViewController:
 - Estos componentes representan la interfaz de usuario y su controlador asociado que permiten al trabajador seleccionar la opción de exportar a JSON.
4. Storage:
 - El componente Storage se encarga de la gestión del almacenamiento de datos en la aplicación. En este caso, puede ser utilizado para guardar el archivo JSON generado con la cita exportada.

Situación 2: Exportar a Markdown

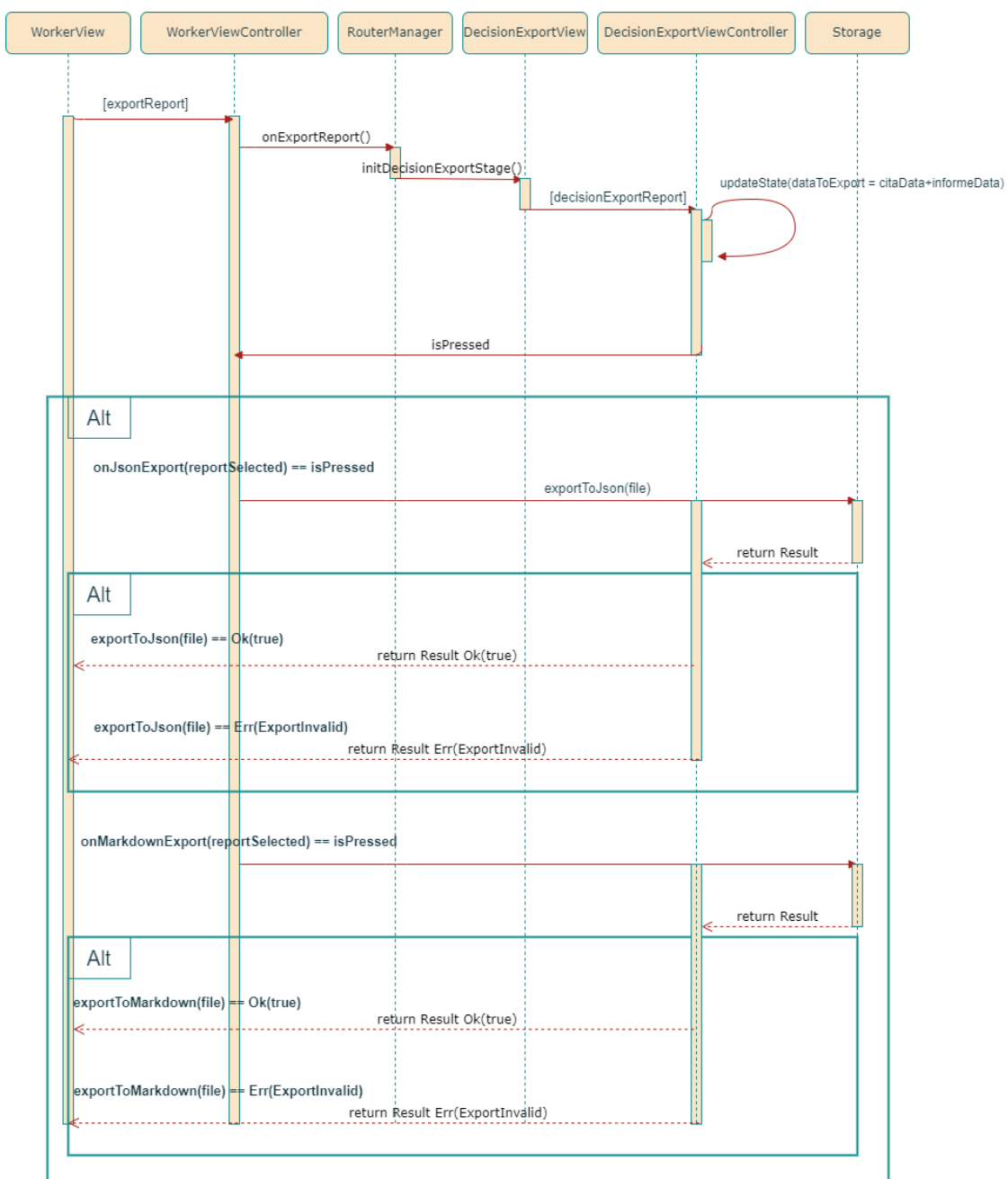
1. WorkerView y WorkerViewController:
 - Estos componentes representan la interfaz de usuario y su controlador asociado que permiten a los trabajadores interactuar con la aplicación y desencadenar la exportación de la cita a Markdown.
2. RouterManager:
 - El RouterManager se encarga de la navegación entre las diferentes vistas de la aplicación. En este caso, puede ser utilizado para redirigir al trabajador desde la WorkerView a la vista de DecisionExportView.

3. DecisionExportView y DecisionExportViewController:

- Estos componentes representan la interfaz de usuario y su controlador asociado que permiten al trabajador seleccionar la opción de exportar a Markdown.

4. Storage:

- El componente Storage se encarga de la gestión del almacenamiento de datos en la aplicación. En este caso, puede ser utilizado para guardar el archivo Markdown generado con la cita exportada.



7. Creación de aplicación de escritorio

7.1. Diseño, patrones y arquitectura

Para el diseño de la aplicación de escritorio basada en la arquitectura **MVVM**, se han seguido los siguientes principios y patrones:

Arquitectura de tres capas, la arquitectura de la aplicación se ha dividido en tres capas: interfaz de usuario, lógica de negocio y acceso a datos. Esta separación permite una clara división de responsabilidades y mejora la mantenibilidad y escalabilidad del sistema.

- Capa de interfaz de usuario (View): En esta capa se encuentra la interfaz gráfica de la aplicación, que muestra los datos al usuario y captura las interacciones.
- Capa de lógica de negocio (ViewModel): El ViewModel actúa como intermediario entre la vista y el modelo. Se encarga de exponer los datos y comandos necesarios para la interfaz de usuario. El ViewModel se comunica con el modelo para obtener y actualizar los datos necesarios.
- Capa de acceso a datos (Model): El modelo representa los datos y la lógica de negocio de la aplicación. Se encarga de acceder y manipular los datos, así como de realizar las operaciones relacionadas con la lógica de negocio.

A continuación, se describen algunos de los **patrones** utilizados:

1. Patrón Observer:

- Se ha aplicado el patrón Observer mediante el uso de SimpleObjectProperty, que actúa como el sujeto observable. Otros componentes pueden suscribirse a los cambios en el SimpleObjectProperty y ser notificados cuando ocurren modificaciones.

2. Patrón Singleton:

- Se ha buscado asegurar que exista una única instancia de esos objetos en toda la aplicación, para garantizar ahorro de recursos, acceso centralizado y control de dependencias.

En las clases: “DatabaseManager”, “Repositories”, “Storages” y “ViewModels”.

3. Patrón State:

- El patrón State se ha utilizado a través de la clase “CitaState” y “CrearModificarCitaState”, que se encarga de almacenar y pasar los datos de un controlador a otro en las distintas vistas. Este patrón permite gestionar el estado de la aplicación de manera eficiente y controlada.

4. Patrón Adapter:

- Mappers: Se ha utilizado el patrón Adapter al implementar mappers que se encargan de convertir objetos en formatos específicos, como JSON y Markdown, mediante el uso de Dto's (Data Transfer Object). Estos mappers actúan como adaptadores entre las estructuras de datos internas y los formatos externos.

Además de estos patrones, también se ha utilizado “RouterManager” para gestionar la navegación entre las diferentes vistas, “DataBaseManager” para manejar el acceso y manipulación de los datos en la base de datos, “AppConfig” para disponer de una configuración general de la aplicación mediante el archivo “properties” gracias a la carpeta de “resources”

7.2. Arquitectura basada en principios SOLID

Se han utilizado varios principios SOLID para mejorar la calidad del código y facilitar su extensibilidad y reutilización. Algunos de los patrones empleados son:

- Principio de **responsabilidad única** (SRP): hemos buscado que cada clase presente una responsabilidad y una única función en específico.
- Principio de **abierto/cerrado** (OCP): Las clases están abiertas para su extensión gracias a las implementaciones de las interfaces, y cerradas para su modificación.
- Principio de **sustitución de Liskov** (LSP): Las clases derivadas pueden ser utilizadas en lugar de las clases base sin afectar el funcionamiento del programa.
- Principio de **segregación de interfaces** (ISP): Las interfaces son específicas y contienen solo los métodos necesarios para su implementación a las distintas clases.
- Principio de **inversión de dependencia** (DIP): Las clases dependen de abstracciones, y nos ha facilitado esta tarea **Koin**, gracias a su automatización respecto a dependen de otras abstracciones.

7.3. Uso de Railway Oriented Programming

Hemos utilizado el patrón Railway Programming para manejar las validaciones y en el caso de ocasionar un error al exportar las citas.

1. Al realizar validaciones en tu sistema, al aplicar el patrón (ROP) en las validaciones, podemos tener un flujo de trabajo más estructurado, donde cada etapa se encarga de una validación específica de cada campo. Esto facilita la comprensión y el mantenimiento del código de validación, y por supuesto para el usuario detectar que está realizando incorrectamente.
2. Exportar citas con errores: al utilizar el patrón (ROP) en la exportación de citas con errores, podemos tener un control más preciso sobre los posibles problemas que pueden surgir durante el proceso de exportación y proporcionar una experiencia más informativa y útil al usuario.

En resumen, el patrón Railway Programming nos ha permitido una flexibilidad para manejar situaciones complejas de manera más elegante y escalable, sin dejar que la aplicación colapse.

7.4. Interfaz de escritorio

1. **Vista de Lista de Citas:** Esta vista muestra una lista de todas las citas registradas. Donde incluimos información muy detallada sobre la cita. Los usuarios pueden desplazarse por la lista y seleccionar una cita para editarla, exportarla a JSON o Markdown.

Disponemos de un filtro combinado de matrícula de vehículo, tipo de vehículo y fecha de la cita.

En la parte superior tenemos un menú, donde se podrá abrir una vista de “Acercade” sobre la información de la ITV, podremos exportar todas las citas a JSON, o crear una nueva cita.

2. **Vista de Acerca de:** se encuentra la información sobre la ITV.
3. **Vista de Creación de Cita:** En esta vista, los usuarios pueden ingresar los detalles necesarios para crear una nueva cita.
4. **Vista de Edición de Cita:** Esta vista permite a los usuarios modificar los detalles de una cita existente. Muestra los campos existentes de la cita y permite a los usuarios editar estos campos según sea necesario.


7.5. Pruebas del software

Hemos utilizado las bibliotecas de pruebas **Mockito** y **Junit**:
















JUnit nos permitió estructurar y ejecutar mis pruebas de manera organizada.

Mediante **Mockito** hemos podido crear objetos simulados, conocidos como "mocks".

Finalmente adquirimos en los apartados relevantes, es decir, "services" y "storages", 81% y 97% de cobertura respectivamente:

>  services	57% (24/42)	67% (42/62)	81% (310/380)
>  router	0% (0/14)	0% (0/32)	0% (0/136)
>  repositories	82% (46/56)	83% (92/110)	97% (716/734)

En el proyecto general nos aventuramos a realizar algunas pruebas a las funciones del "ViewModel" para intentar subir la cobertura general del proyecto pudiendo alcanzar el 46% de cobertura:

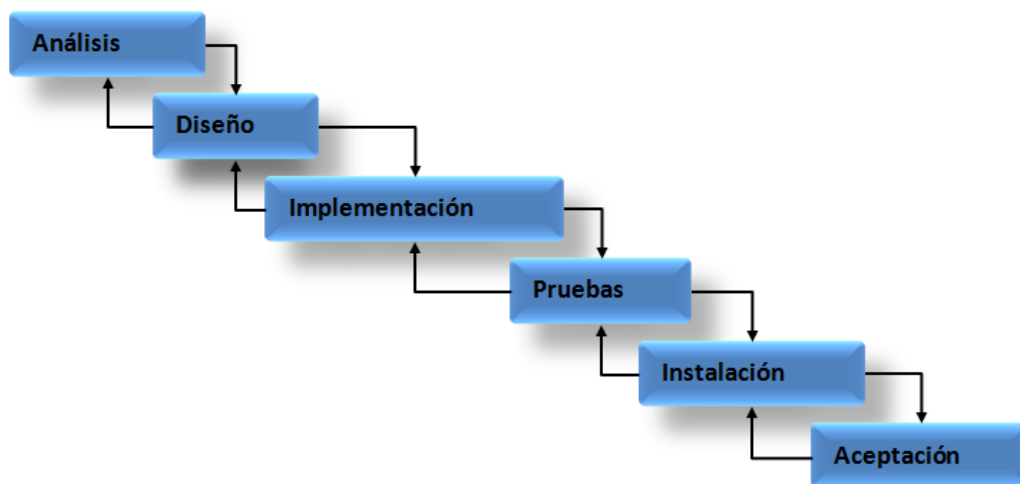
✓  proyectofinalitv	33% (124/3...	35% (230/652)	46% (1600/3452)
 AppMainKt	0% (0/1)	0% (0/1)	0% (0/1)
 AppMain	0% (0/2)	0% (0/3)	0% (0/6)
>  viewmodels	69% (18/26)	55% (32/58)	47% (276/584)
>  validators	0% (0/14)	0% (0/24)	0% (0/150)
>  services	57% (24/42)	67% (42/62)	81% (310/380)
>  router	0% (0/14)	0% (0/32)	0% (0/136)
>  repositories	82% (46/56)	83% (92/110)	97% (716/734)
>  models	100% (16/16)	100% (18/18)	100% (118/118)
>  mappers	100% (2/2)	100% (6/6)	70% (28/40)
>  errors	0% (0/40)	0% (0/40)	0% (0/40)
>  dto	100% (12/12)	100% (12/12)	100% (74/74)
>  di	0% (0/22)	0% (0/22)	0% (0/50)
>  controllers	0% (0/108)	0% (0/224)	0% (0/1046)
>  config	75% (6/8)	77% (28/36)	90% (78/86)

8. Detalles y conclusiones

Se ha utilizado **SonarLint** para detectar posibles “Code Smells” y tener una buena calidad en nuestro código y favorecer la filosofía “Clean Code”.

Durante el sprint, hemos adoptado un enfoque colaborativo, lo que nos ha permitido trabajar de manera eficiente y adaptarnos rápidamente a los cambios. La segmentación del trabajo ha sido fundamentales para lograr los objetivos establecidos en el tiempo asignado.

Finalmente, durante el desarrollo, seguimos rigurosamente las etapas del ciclo de vida en cascada, mediante una metodología secuencial y estructurada comenzando por el análisis de requisitos, seguido del diseño, la implementación, las pruebas y, finalmente, el despliegue. Cada etapa tuvo su propia dedicación de tiempo y recursos, y no se pasó a la siguiente fase hasta que la anterior se considerara completa y exitosa, en caso de éxito otro compañero continuaba con la secuencia de trabajo, mientras que otro repasaba el trabajo realizado teniendo en cuenta un apartado anterior del ciclo.



Ciclo de vida en cascada