UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Val&Go: Aplicación de Movilidad Urbana para Valencia

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Pablo Gallardo Mosteo

**Tutor**: Germán Moltó Martínez

2018/2019

# Resumen

Este proyecto trata sobre la elaboración de una aplicación móvil para la plataforma Android. La aplicación consiste en un agregador de servicios de movilidad urbanos, es decir, junta varios servicios de alquiler de vehículos geolocalizables en una misma aplicación. Estos servicios proporcionan vehículos de alquiler a los usuarios, pudiéndose alquilar por minutos. La aplicación objeto de este trabajo junta todos los servicios existentes en la ciudad de Valencia, haciendo visible todos los vehículos en el mismo mapa y pudiéndose comparar parámetros como precio, distancia o carga del vehículo.

**Palabras clave:** movilidad, Valencia, motosharing, carsharing, Android.

# Abstract

This project is about the developement of a mobile app for Android. This app consists of an aggregator of urban mobility services, meaning that it brings together several rental services of geolocated vehicles in the same app. These services provide rental vehicles, being able to rent a vehicle in a minute basis. This app brings together all the services that are available in the city of Valencia, making all vehicles visible on the same map and being able to compare parameters such as price, distance or vehicle charge.

**Keywords :** mobility, Valencia, motosharing, carsharing, Android.

# Tabla de contenidos

# Índice de ilustraciones

# 1. Introducción

Las nuevas tecnologías han invadido nuestra sociedad por completo. Prácticamente la totalidad de la población posee algún aparato electrónico, ya sea un teléfono móvil, una tableta, o un ordenador. La sociedad cambia, y con ella cambiamos nosotros, nuestras ciudades, y la forma en la que vivimos y nos relacionamos.

La movilidad o el transporte de personas ha sido siempre un tema muy importante en la sociedad moderna, desde la invención del automóvil, hasta los últimos sistemas de transporte urbanos. En los últimos años han aparecido nuevos modelos de movilidad, siendo uno de los más relevante el llamado *sharing*, que consiste en el uso de vehículos compartidos ofrecidos por una empresa determinada. Dicho sistema tiene numerosas ventajas: el usuario utiliza el vehículo que se encuentre más cercano en ese momento, lo alquila, lo deja en su lugar de destino, y solo paga por el tiempo que lo ha estado utilizando. Posteriormente otro usuario puede utilizar el mismo vehículo con la misma finalidad.

Los servicios que actualmente utilizan el modelo de *sharing* utilizan aplicaciones móviles propias para comunicarse con sus clientes. Estas aplicaciones contienen información en tiempo real de la posición, autonomía, y estado de cada vehículo de su flota. El usuario, previamente registrado, puede seleccionar el vehículo que mejor se ajuste a sus necesidades y reservarlo al momento para su posterior utilización. Es aquí donde se presenta un problema, debido a que existen múltiples plataformas o empresas que ofrecen este tipo de servicios, puede ser complicado para el usuario gestionar tantas aplicaciones y tantos vehículos que, por otra parte, comparten el mismo espacio urbano.

El objetivo de este proyecto es el desarrollo de **Val&Go**, una aplicación dirigida al sistema operativo Andoird que une e integra todos los servicios *sharing* de la **ciudad de Valencia**.

## 1.1 Motivación

La realización de este proyecto surge a partir de la problemática expuesta anteriormente. La ciudad de Valencia dispone de diversos servicios de movilidad compartida, la mayoría de motocicletas eléctricas. Cada servicio dispone de una aplicación independiente, sus propios servidores y sus propias APIs para interactuar con el usuario. Si por ejemplo un usuario utiliza varios de estos servicios, le será complicado deducir que vehículo se encuentra más cerca, cual le puede salir más barato, o cuál de ellos tiene más autonomía para su trayecto. Val&Go pretende resolver este

problema, proporcionando información conjunta y en tiempo real de todas las plataformas.

A parte de esto, asignaturas como **Integración e Interoperabilidad** (de la rama Ingeniería del Software) y **Desarrollo de Aplicaciones para dispositivos móviles** también han contribuido o motivado la realización de esta aplicación. Muchos de los conocimientos adquiridos de estas asignaturas han sido de gran ayuda.

También cabe destacar que en un principio esta aplicación se empezó a diseñar para uso propio. Yo, el autor, necesitaba algo parecido, y al no encontrarlo, decidí realizarlo por mi cuenta. Por otra parte, me parecía interesante diseñarla en el lenguaje que más he aprendido durante la carrera (Java), y conocer más el sistema operativo de Google, profundizando en conceptos mucho más avanzados. Por tanto el aprendizaje de nuevas tecnologías también es un motivo importante para la realización de este proyecto.

## 1.2 Objetivos

El principal objetivo es la realización de una buena aplicación, que sea sencilla, fácil de utilizar e intuitiva. Para ello se prestará especial atención a los mínimos detalles estéticos y a la experiencia que el usuario tiene utilizándola. Es necesario que no se le haya de explicar al usuario como utilizarla, sino que aprenda a utilizarla al instante. Esto se puede lograr utilizando elementos reconocibles de Android. En este sistema operativo muchos de los elementos, imágenes, e iconos están estandarizados para que el usuario los reconozca al instante y sepa lo que hace cada uno. Este objetivo se conseguiría utilizando librerías y estilos de Android, utilizando una distribución de elementos y colores acordes a las guías de estilo propuestas por Google (Material design).

Otro objetivo importante es que la aplicación sea estética a la vista. Este punto es importante, pues una aplicación bonita atrae consciente o inconscientemente a más usuarios, y da la sensación de un producto más acabado y más serio. Se van a cuidar mucho los detalles y el diseño de los elementos y la interfaz. Android es un sistema operativo para dispositivos muy diferentes, con diversos tamaños de pantalla y densidad de píxeles. Es necesario por ello utilizar las herramientas de diseño necesarias para mostrar la interfaz con la mayor resolución y fluidez posible.

Un objetivo no menos importante es la buena estructuración del código fuente. Las aplicaciones para Android se escriben en Java, un lenguaje orientado a objetos. Es por ello que el objetivo es utilizar los patrones de diseño necesarios para modularizar y estructurar el código de forma limpia y entendible. El patrón de arquitectura de diseño que se va a utilizar en este proyecto es el MVP o Modelo-Vista-Presentador, que consiste en la separación y estructuración del código de tal forma que se separan las diferentes lógicas de la aplicación. También se pretenden utilizar patrones de diseño como *Singleton*, *Abstract Factory,* o *Wrapper.*

# 2. Estado del arte

## 2.1 Desarrollo de aplicaciones Android

Android es el sistema operativo para móviles más popular en la actualidad. Esto hace que la mayoría de desarrolladores se decante por esta plataforma, ya que aproximadamente más del 80% de los *smartphones* montan este sistema actualmente. Cada vez más fabricantes se decantan con el sistema operativo de Google, y esto hace que el desarrollo se estandarice y sea más fácil para los programadores. Desarrollar una aplicación para Android significa que la va a poder utilizar más del 80% de la población, lo que es una buena noticia.

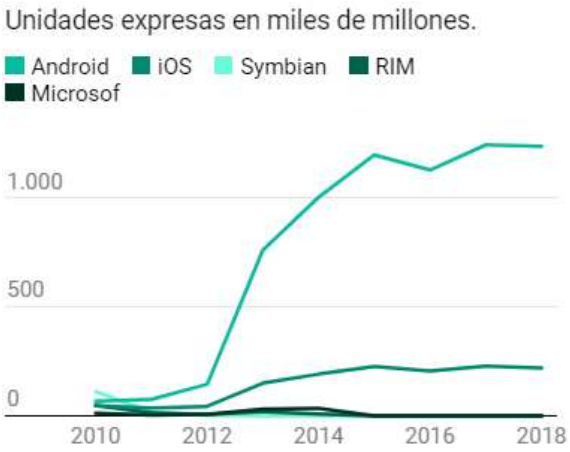


*Ilustración 1: Sistemas operativos para móviles más usados*

Android domina la cuota de mercado. Como se puede observar en la gráfica, su tendencia ha sido creciente a lo largo de la historia, y todo parece indicar que lo seguirá siendo.

Debido a la gran cantidad de desarrollo en esta plataforma, es normal que aparezcan distintas técnicas para desarrollar y distintos tipos de aplicaciones. Históricamente el desarrollo en Android se hace en el lenguaje de programación **Java**. Este es un lenguaje perfecto para la plataforma, ya que es orientado a objetos y altamente tipado. En 2013 el equipo de JetBrains diseño un lenguaje llamado **Kotlin**, de tipado estático, muy simple, y que corre en la máquina virtual de Java. En los últimos años, concretamente a partir de 2017, se ha utilizado para el desarrollo en Android, hasta el punto de convertirse en un lenguaje cooficial junto a Java.

Java y Kotlin constituyen el desarrollo nativo de Android, pero también es posible realizar una aplicación en cualquier otro lenguaje que se ajuste más a nuestras

necesidades. Esto se hace a través de los llamados *frameworks*, que son un conjunto de programas y bibliotecas que nos facilitan el desarrollo software. A continuación se exponen brevemente los *frameworks* más relevantes:

### Xamarin

Xamarin ofrece un conjunto de herramientas para desarrollar aplicaciones en el lenguaje de programación **C#**. Ha sido usado por más de 1 millón de usuarios. Gracias a este *framework* los desarrolladores disponen de todas las ventajas y la potencia de Microsoft Visual Studio, a parte de la posibilidad de poder desarrollar en un lenguaje de programación distinto a Java.

### Ionic

Ionic es un *framework* de código abierto y gratuito. Ofrece un amplio catálogo de componentes, gestos y herramientas. Te permite crear apps tanto para Android como para IOS. Ionic se centra en la optimización y el buen funcionamiento. Se utiliza para el desarrollo de aplicaciones híbridas, es decir, son una combinación de tecnologías web como HTML, CSS y JavaScript, que no son ni aplicaciones móviles verdaderamente nativas ni tampoco están basadas en Web. Por tanto es posible utilizar toda la potencia de las tecnologías web (JS, CSS) para el desarrollo de aplicaciones móviles.

### React native

Este *framework* está basado en Node y NPM, por lo que se desarrolla completamente de Javascript. La parte interesante es que nuestro desarrollo en este lenguaje sirve tanto para la plataforma Android como para IOS. Lo único que se necesita son conocimientos de Javascript. El desarrollador solo se tiene que centrar de la lógica de negocio y de la maquetación.

## 2.2 Aplicaciones similares

Actualmente existen diversos agregadores de movilidad en el mercado. No todos tienen el mismo objetivo, unos se centran en servicios de transporte público, como autobuses, metro… Otros sirven para contratar servicios privados, por ejemplo servicios de taxi, Uber o Cabify. En menor cantidad se encuentran las aplicaciones de *sharing* o vehículos compartidos. Es en este último apartado en el que se encuentra la aplicación que se desarrolla en este proyecto.

A continuación se muestran algunas de estas aplicaciones, en su mayoría agregadores de servicios.

### 2.2.1 Free2Move

Free2Move es la aplicación más completa de servicios de movilidad. Es un agregador de servicios de c*arsharing, motosharing,* y bicicletas. Ofrece este servicio en múltiples ciudades europeas y estadounidenses, como por ejemplo París, Lisboa, o Washington DC. Es posible consultar tanto la posición de los vehículos como su carga y distintos parámetros. A parte también son proveedores de servicios *carsharing,* y disponen de una flota de vehículos eléctricos solo disponibles en algunas ciudades europeas.

La compañía que está detrás de esta aplicación es una compañía grande, concretamente el Grupo PSA. Es por esto que se trata de un producto muy acabado, seguramente desarrollado por un grupo grande de personas.

Entre los aspectos más destacados de esta aplicación se encuentran:

- Ofrece la posibilidad de reservar el vehículo a través de su misma plataforma, es decir, sin redirigirte a la aplicación original. Una vez reservado es posible iniciar el viaje y detenerlo cuando se precise.
- Ofrece descuentos en algunas tarifas debido a la colaboración con los proveedores disponibles.
- También es posible alquilar servicios VTC a través de su plataforma.

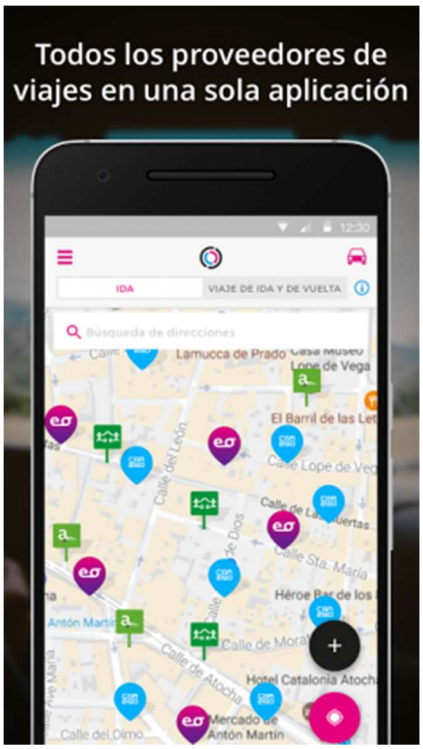


*Ilustración 3: Publicidad de Free2Move*

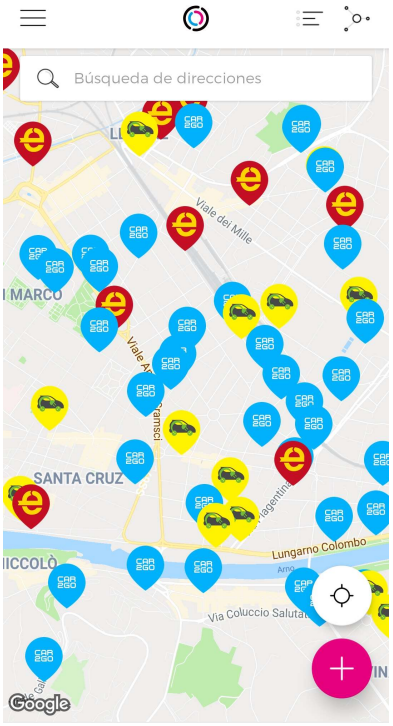


*Ilustración 2: Captura de la aplicación Free2Move*

### 2.2.2 Urbi

Se trata de un agregador de *carsharing* muy completo. Esta disponible para la plataforma Android y tiene mas de 100.000 descargas. Ofrece información de muchas ciudades alrededor del mundo, como Madrid, Roma o San Francisco. Ofrece una gran cantidad de servicios de movilidad, incluyendo el transporte público, pero tiene una interfaz poco amigable y produce fallos muy a menudo.

Estos son las características más destacadas:

- Muestra servicios de *sharing* tanto de coches como de motocicletas y también proveedores de bicicletas.
- Ofrece la posibilidad de llamar a un taxi o contratar servicios de VTC.
- En alguna ciudad tiene integrados los servicios de transporte público, permitiendo visualizar la hora a la que pasa un autobús o un tren.
- Algunos servicios se pueden contratar a través de su plataforma sin necesidad de intermediarios.

A pesar de que tiene muchas funcionalidades, al igual que Free2Move, no se centra en la ciudad de Valencia, sino que se centra en otras ciudades más grandes. Es en este aspecto donde se diferencian de la aplicación de este proyecto. Val&Go ofrece la información en tiempo real de todos los servicios de sharing de la ciudad de Valencia, y se adapta a las necesidades de los usuarios de esta ciudad en concreto.

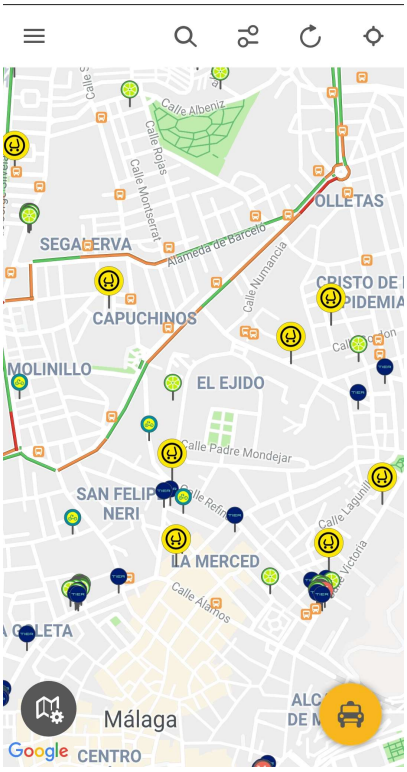


*Ilustración 4: Captura de la aplicación Urbi*

## 2.3 *Sharing* en Valencia

Como ya se ha expuesto en apartados anteriores, el *sharing* o *carsharing* es un sistema de uso temporal de vehículos. Se trata de un modelo de alquiler en el que el usuario contrata el alquiler de un vehículo, normalmente por minutos o horas. Este vehículo suele encontrarse ya en la vía urbana y es localizable a través de una aplicación móvil.

Actualmente en la ciudad de Valencia hay disponibles varios servicios de carsharing, en su totalidad de alquiler de motocicletas eléctricas. En un futuro se espera que empiecen a operar en la ciudad servicios de *sharing* de coches eléctricos. De momento existen seis servicios de motos eléctricas: **Muving, Blinkee, Yego, Ecooltra, Acciona y Molo**. Val&Go une la totalidad de ellos, añadiendo además el ya popular servicio de bicicletas **Valenbisi.** A continuación se expone una breve descripción de estos servicios.

### Muving

Se trata de una empresa española con sede en Cádiz que ofrece servicios de motos de alquiler en un total de 9 ciudades españolas. Es la empresa pionera de este tipo de servicios, ya que lleva operando desde 2017, y la primera que empezó a operar en Valencia. Ofrece motos eléctricas de una potencia equivalente a 125cc, por lo que es necesario un permiso de motocicleta. Ofrece sus servicios a 0,25 céntimos el minuto, lo que lo convierte en uno de los servicios más caros, pero también más rápidos.

### Blinkee

Empresa polaca que solo opera en ciudades de Polonia y en la ciudad de Valencia. Ofrece servicios de motocicletas y de bicicletas, aunque estas últimas solo en ciudades polacas. Sus motos son de una potencia equivalente a 49cc, y no superan los 50km/h. Esto es una ventaja, ya que pueden ser utilizadas con el permiso de ciclomotor (AM). El aspecto más característico de esta marca es que ofrecen sus servicios por solo 0,18 céntimos el minuto, cantidad que se reduce a 0,16 si se posee un carnet de estudiante. Es la opción más económica de la ciudad y uno de los servicios de más calidad.

### Yego

Se trata de una empresa catalana que ofrece motocicletas con un aspecto de *Vespa.* De momento solo ofrecen servicios en Barcelona y Valencia. Sus motos también están limitadas a 50 km/h.

### Ecooltra

Se trata de una empresa grande que opera en 6 ciudades de Europa, incluyendo Roma, Madrid o Barcelona. Es uno de los servicios más caros, cobrando 0,26 céntimos el minuto. Sus motos son más potentes, llegando a velocidades de 80 km/h.

### Acciona Mobility

Se trata de una filial del gigante eléctrico español Acciona. Ofrecen motos eléctricas por minutos. Estos vehículos ofrecen dos modos de conducción distintos, uno que está limitado a 50 km/h y otro que puede llegar a 80. El precio del trayecto es de 0,27 euros el minuto, lo que lo convierte en el servicio más caro de la ciudad.

### Molo

Se trata de una joven empresa Valenciana que ofrece alquiler de motos con un tiempo ilimitado, es decir, se paga una suscripción mensual asequible y es posible utilizar sus vehículos tantas veces como se quiera sin limitación de tiempo. Esto lo convierte en un servicio muy interesante. Además, los vehículos son muy potentes, llegando a velocidades de 70-80 km/h.


*Ilustración 5: Vehículos de algunas plataformas disponibles en Valencia*

# 3. Análisis del problema

Como todo proyecto de ingeniería de software una de las primeras fases siempre es el análisis del problema. En nuestro caso el problema a resolver es la creación de una aplicación que permita al usuario visualizar información en tiempo real, procedente de diferentes servicios web. La aplicación tiene que proporcionar funcionalidades útiles y sencillas de entender. En este apartado se muestran diferentes aspectos importantes del análisis del problema: la especificación de requisitos, diagrama de casos de uso, diagrama UML, y finalmente se analizarán los riesgos de la realización de esta aplicación.

## 3.1 Especificación de requisitos

En este apartado se van a especificar los requerimientos del software. Este apartado tiene como objetivo formalizar las funcionalidades de la aplicación, y hacer un esquema inicial de como va a funcionar. De esta manera es más sencillo el desarrollo y es más fácil saber si se han cumplido los objetivos al finalizar el proyecto. A continuación, se detallan los **Requisitos Funcionales (RF)** especificados:

- Visualizar mapa: El usuario podrá visualizar el mapa con todos los vehículos mostrados como marcadores, y podrá elegir el que más se ajuste a sus necesidades.

- Abrir información vehículo: El usuario podrá presionar un vehículo determinado y se mostrará automáticamente la información referente a este, la carga de la batería, la distancia andando, el precio del trayecto, o la información de la matrícula.

- Reservar vehículo: Existirá un botón llamado “Reservar” que redireccionará al usuario a la aplicación oficial del servicio de s*haring* pertinente. En esta aplicación podrá registrarse, o reservar el vehículo si ya lo está.

- Ocultar proveedor: Si se desea se podrá ocultar un proveedor determinado y desaparecerán automáticamente los vehículos de dicho proveedor del mapa. Esto se mantendrá una vez se cierre la aplicación y se vuelva abrir, siempre mostrándose los servicios preferentes del usuario.

- Mostrar proveedor: Si se ha ocultado un proveedor anteriormente, el usuario podrá volver a mostrarlo, mostrándose así en el mapa todos los vehículos del servicio disponibles en el momento.

- Ver lista vehículos cercanos: El usuario podrá acceder a una lista con los vehículos más cercanos a su alcance. Esta lista se mostrará en una *Activity* distinta, y se tratará de una lista reciclada de la API de Android.

- Abrir Google Maps: Una vez se haya seleccionado un vehículo, el usuario tendrá la opción de abrir Google Maps con un marcador en la posición exacta del vehículo.

Los requisitos funcionales sirven para saber los procesos que puede llevar a cabo el usuario, pero también es importante saber que requisitos específicos va a poder cumplir la aplicación. Para esto se utilizarán los **Requisitos No Funcionales (RNF)**. A continuación se detallan algunos:

- Menos de 5 segundos en iniciar: La aplicación debe tardar menos de 5 segundos en iniciar. Aunque esto depende del tipo de móvil que se posea, en promedio debe ser inferior a esta marca. Móviles de alta gama deben poder iniciarla en menos de 2 segundos.

- Conexión a internet: La aplicación necesita un mínimo de conexión a internet para funcionar, ya que su función depende de la conexión con distintos servicios web y distintas APIs.

- Usable: Debe ser muy fácil de utilizar, y lo más intuitiva posible. No debe hacer falta un tutorial para que el usuario sepa cómo utilizarla.

- Resoluciones de pantalla: La aplicación debe visualizarse de forma correcta para distintos tamaños de pantalla. Este es un requisito indispensable ya que Android se utiliza en teléfonos muy diversos con distintas densidades de píxeles. Las imágenes y elementos en pantalla deben ser flexibles a estos cambios. Si no se cumple este requisito se producen fenómenos como imágenes borrosas o pixeladas, o elementos excesivamente grandes o pequeños en pantalla.

- Versión de Android: Es indispensable que la aplicación funciones para el mayor número de versiones de Android posible, por lo menos para las que más abundan y son más recientes. La API del sistema brinda herramientas para conseguir esta retrocompatibilidad.

## 3.2 Diagrama de casos de uso

Debido a que Val&Go se trata de una aplicación simple y a que se puede utilizar sin necesidad de registro, el diagrama de casos de uso se limita a mostrar los requisitos funcionales de la aplicación de una forma gráfica. A continuación se muestra el diagrama:

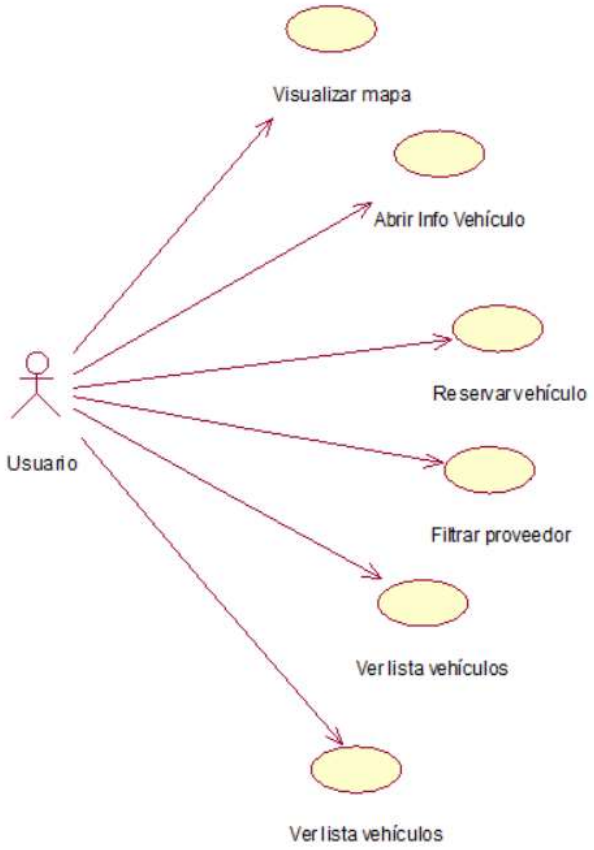


*Ilustración 6: Diagrama de casos de uso*

Como se puede observar solo existe un tipo de usuario, ya que no es necesario el registro. Si que es necesario el registro en los diferentes proveedores, pero esto ya se sale del uso de esta aplicación, por lo que no sería correcto mostrarlo en los diagramas.

## 3.3 Análisis legal

Para llevar a cabo este proyecto es un requisito indispensable conseguir la interoperabilidad con varios servicios web, concretamente con las APIs de cada proveedor. Estas APIs son las que proporcionan los datos de cada vehículo de su plataforma. Es por esto que han sido necesarias labores de **Ingeniería Inversa** para recabar información. Esto último se expone en apartados posteriores de este proyecto.

La ingeniería inversa es una técnica o un proceso de ingeniería de software que tiene como objetivo obtener información a partir de un producto software ya construido. Se trata de una labor compleja, ya que es necesario comprender el funcionamiento de un sistema software que ha sido desarrollado por otras personas.

El objetivo de Val&Go es unir diferentes servicios que geolocalizan vehículos en tiempo real. Estos servicios utilizan **APIs con la arquitectura REST**, a las que se realizan peticiones y devuelven los datos de los vehículos. Para conseguir estos datos y averiguar de dónde se obtienen es necesario realizar la ingeniería inversa sobre las aplicaciones de estos servicios.

Estas técnicas puede dar la sensación de que no están dentro del marco legal, o que se tratan de técnicas de *hacking*. Pero esto no es cierto, ya que en España la ingeniería inversa es legal siempre que lo que se quiera es conseguir la interoperabilidad. Según el **Artículo 100.3** de la Ley de Propiedad Intelectual: *"El usuario legítimo de la copia de un programa estará facultado para observar, estudiar o verificar su funcionamiento, sin autorización previa del titular, con el fin de determinar las ideas y principios implícitos en cualquier elemento del programa, siempre que lo haga durante cualquiera de las operaciones de carga, visualización, ejecución, transmisión o almacenamiento del programa que tiene derecho a hacer"*. Esto quiere decir que es posible hacer un análisis de la copia de un programa, y que la persona que posee dicha copia posee legítimamente ese derecho. En el artículo 100.5 se deja claro que todo esto es legal siempre que se busque la interoperabilidad con un programa independiente.

En el capítulo *Ingeniería Inversa* de este proyecto se hablará más a fondo sobre este tema.

## 3.4 Análisis de riesgos

Depender de tantos servicios web diferentes añade una problemática importante: no es posible controlar estos servicios ya que pertenecen a distintas organizaciones. Es por ello que si algún servicio se cae no se puede hacer nada por recuperarlo. Por tanto la persistencia de los servicios que se visualizan en la app es incontrolable, y solo depende del mantenimiento de los distintos proveedores (que por otro lado harán todo lo posible por mantenerlos). La parte positiva es que este es un riesgo conocido, y es posible tomar medidas para prevenirlo. La única medida que se puede tomar para reducir este riesgo es avisar al usuario de que el error es externo y no depende de la propia aplicación. La parte positiva es que el servicio original tampoco estaría disponible, lo que no cabrearía al usuario.

Otro riesgo importante de utilizar tantos servicios web es que con la falta de conexión a internet la aplicación sería completamente inútil. También es cierto que cualquier aplicación actual requiere de internet para funcionar completamente, pero Val&Go funciona con información en tiempo real, lo que la hace más vulnerable a este tipo de situaciones. La solución más obvia a este tipo de problema sería informar al usuario de su falta de conexión, y este podría hacer las gestiones necesarias para reestablecerla.

# 4. Diseño

La fase de diseño es una de las más importantes en todo desarrollo de software. En este apartado se va a explicar la fase de diseño arquitectónico del software, es decir, el patrón arquitectónico utilizado y un diagrama UML que muestra la distribución de las clases y sus relaciones. Otro subapartado de este punto va a ser el diseño de la interfaz. Se van a explicar todos los procesos seguidos para el diseño de la interfaz de la aplicación, todos sus componentes, imágenes, animaciones…

## 4.1 Arquitectura del sistema

Para la realización de este producto software se ha elegido un patrón arquitectónico personalizado, a partir del patrón arquitectónico existente **Modelo-Vista-Presentador (MVP).** Esto quiere decir que no se ciñe completamente a lo que dicta el patrón, sino que hay una serie de cambios respecto a este, como puede ser una división distinta de las clases en paquetes.

MVP es el patrón arquitectónico más utilizado en Android. Está dirigido a un lenguaje orientado a objetos, como es Java. Deriva del patrón Modela-Vista-Controlador (MVC), y es utilizado en su mayoría para construir interfaces de usuario. Toda lógica de presentación es colocada en el *presentador*, y este asume una función de intermediario. A continuación se explica el patrón por cada componente:

- El modelo es la capa que gestiona los datos. Es lo que denominaríamos como lógica de negocio.
- El presentador se sitúa entre el modelo y la vista, y permite conectar la interfaz con los datos.
- La vista se encarga de mostrar los datos al usuario e interaccionar con él. Es aquí donde irían las clases de *Views y Fragments*.

### 4.1.1 Estructura de paquetes

Por tanto la distribución de las clases se hará dependiendo de la función que desempeñan, si sirven para mostrar un elemento en pantalla, para recuperar información de las APIs, o para hacer diferentes cálculos. Val&Go consta de varios paquetes de clases distintos:

- **Adapter**: A este paquete pertenecen aquellas clases que se utilizan como adaptador para listas reciclables de Android. Crear un adaptador es un requisito indispensable para este tipo de listas.

- **Main**: Contiene las clases principales, concretamente las *activities*, que son las clases que dan soporte a las distintas pantallas o escenarios de la aplicación. También contiene otras clases importantes.

- **Util**: Contiene clases con métodos de utilidad, que no dependen de ningún contexto y se utilizan para cálculos simples que son comunes en varias partes del software.

- **View**: Contiene clases referentes a vistas de la interfaz, es decir, a elementos gráficos. Por ejemplo, en este paquete está la clase que controla la animación de ruta cuando seleccionamos un vehículo.

- **Webservices**: En este paquete se encuentran todas las clases que se utilizan para hacer llamadas a los diferentes servicios web. Se encuentran todas las clases que conectan con las APIs de las plataformas de s*haring,* así como la API de direcciones y rutas.

Como se puede observar cada paquete contiene clases que realizan funciones diferentes, pero que aun así están conectadas entre si y se referencian unas a otras. A continuación se muestra un diagrama de paquetes realizado con la herramienta *SimpleUML*. Este diagrama muestra las relaciones entre distintos paquetes:

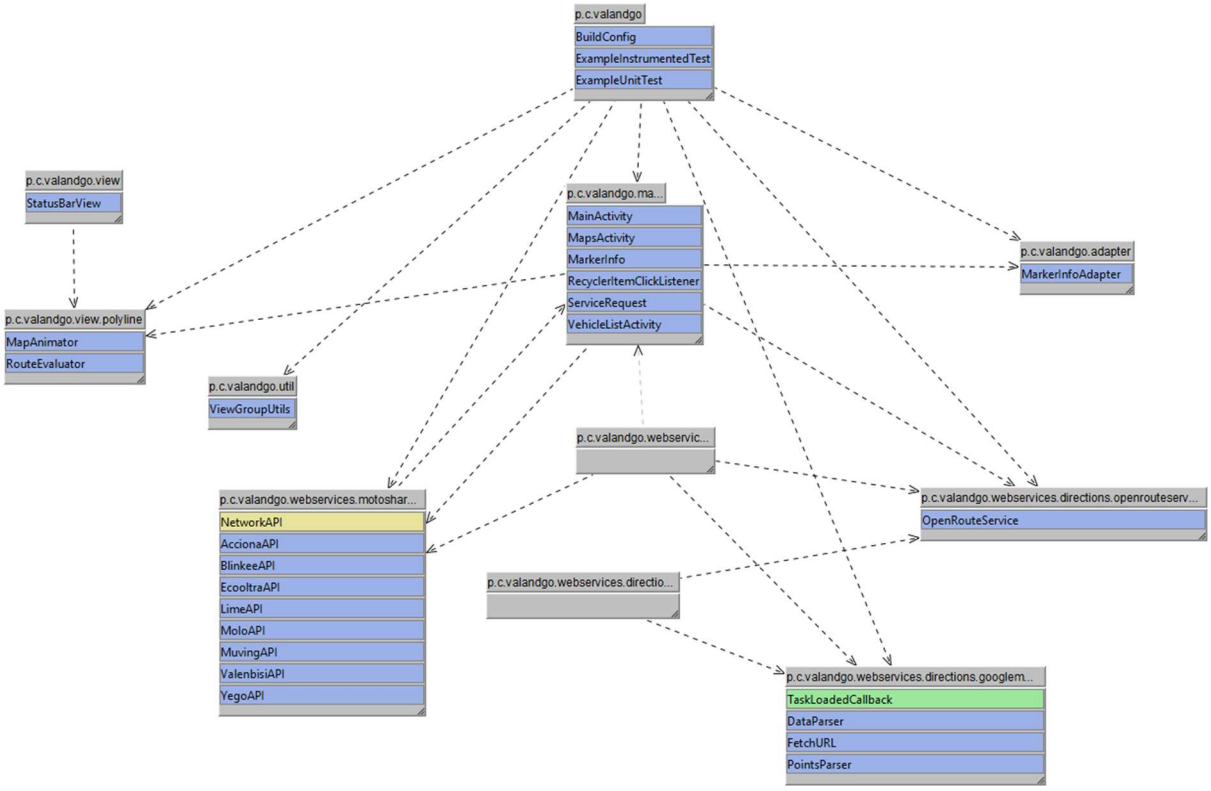


*Ilustración 7: Diagrama de paquetes*

### 4.1.2 Diagrama de clases UML

Para mostrar de forma correcta las distintas clases y las relaciones entre ellas se ha diseñado un diagrama de clases UML. Este diagrama muestra las clases del proyecto (*activities*, utilidades, POJO), sin mostrar sus atributos ni métodos para mayor simplicidad. También muestra las diferentes relaciones a nivel de instancia (asociación) y las relaciones jerárquicas de herencia.

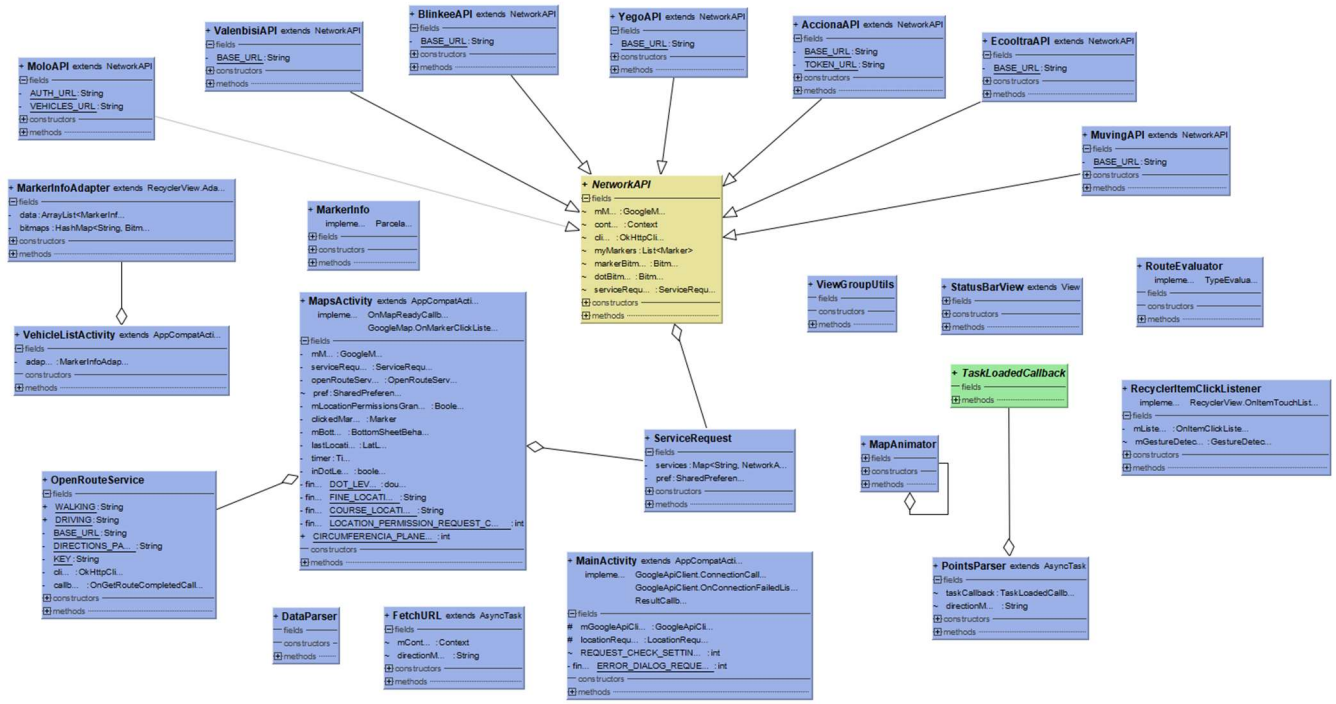


*Ilustración 8: Diagrama de clases UML*

En el diagrama UML de la imagen se puede observar como existen múltiples relaciones de herencia. Estas relaciones hacen referencia a las clases que manejan la conexión con las diferentes APIs. Todas ellas heredan de una superclase abstracta (*NetworkAPI*), que implementa varios métodos y define el método *call()* que deberá implementar cada subclase según las circunstancias. *NetworkAPI* es instanciada por *ServiceRequest*, que a su vez se comunica con la clase de la actividad principal *MapsActivity*. También hay diversas relaciones asociativas entre otras clases. Esto se debe a que unas poseen atributos de instancias de otras clases.

Se observa que existen clases aisladas sin ninguna relación. Esto no significa que no se relacionen con otras clases en absoluto, simplemente no poseen un atributo con la instancia de otra clase.

En el diagrama no se muestran las relaciones con clases de la API de Android, aunque estas son múltiples. Esto se debe a la complejidad que tomaría el diagrama, que quedaría ilegible.

### 4.1.2 Esquema de red

El esquema de red muestra como se conecta la aplicación con los diferentes *webservices*. Se trata de un esquema muy simple que representa visualmente los componentes que conforman la red, en este caso el intercambio de información entre la aplicación y las APIs de las diferentes plataformas. A continuación se muestra un diagrama de red que brinda un panorama amplio de las comunicaciones entre las clases del programa y los servicios de red pertinentes:

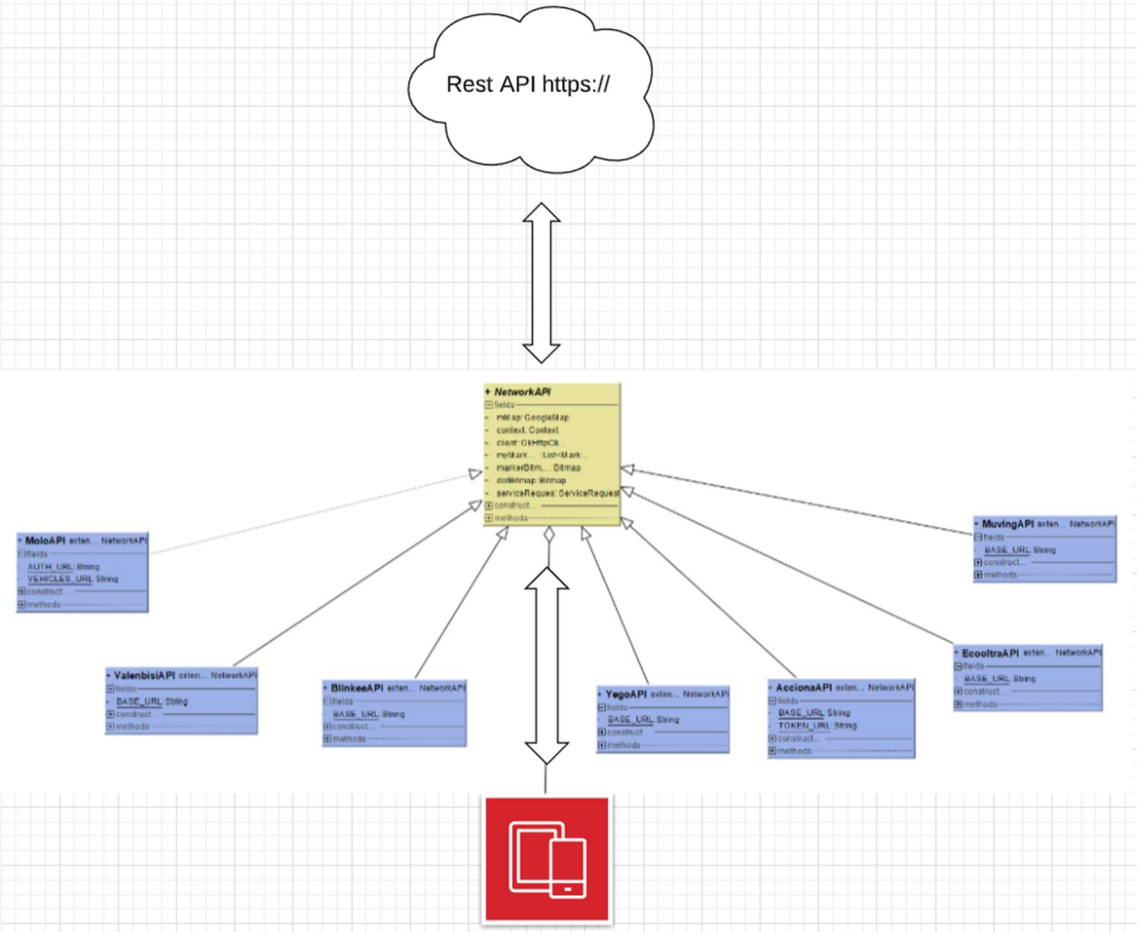


*Ilustración 9: Diagrama de red simplificado*

Como se observa la clase abstracta *NetworkAPI* es la encargada de comunicase directamente con los servicios REST. Lo hace mediante peticiones HTTP utilizando la librería para Android ***OkHttp.*** Es una librería que proporciona herramientas eficientes para la comunicación en red.

## 4.2 Diseño gráfico

En este apartado se va a mostrar como se ha diseñado la interfaz de la aplicación, así como sus distintos componentes, iconos, imágenes y animaciones. En primero lugar se van a mostrar los diseños iniciales o *mockups*. Para la realización de estos bocetos se ha utilizado la aplicación web ***ninjamock.*** Esta aplicación permite el diseño de bocetos de diferentes estados de la aplicación, y pone a disposición del diseñador un amplio catálogo de componentes, como botones, texto, menús, o cuadros de diálogo.

La finalidad de los *mockups* es crear un prototipo inicial de la aplicación, y a partir de este prototipo se realiza el desarrollo del diseño, de una forma más guiada y más simple.

Los diferentes prototipos se han diseñado de forma acorde a las diferentes funciones que puede realizar el usuario en la aplicación. Cada diseño muestra una faceta distinta de la app, mostrándose todos los componentes que aparecen cuando el usuario realiza una operación. También muestra las distintas actividades.

Para el diseño de la interfaz se han seguido los principios de Gestalt, que son una serie de principios de diseño que se centran en la percepción que el usuario tiene de los elementos. Los principales son:

- **Ley de la continuidad**: si varios elementos parecen colocados formando un flujo hacia una misma dirección se pueden percibir como un todo.
- **Ley de la proximidad**: los elementos próximos se perciben como si formaran parte de la misma unidad.
- **Ley de la similitud**: los elementos parecidos se perciben como si tuvieran la misma forma.
- **Ley de cierre**: si una línea describe una forma cerrada tiende a verse como una forma cerrada y no como una línea.

En el siguiente apartado se muestran los diseños o *mockups* que se han realizado de las diferentes pantallas y situaciones que nos encontramos en la aplicación.

### 4.2.1 Mockups

Para la realización de este tipo de diseños se ha utilizada una herramienta online antes ya mencionada. Dicha herramienta ofrece la base de una pantalla de smartphone, y sobre ella se pueden realizar los distintos diseños mediante imágenes, formas, e iconos.

En primer lugar se ha diseñado el *mockup* de la primera pantalla que ve el usuario. Lo primero que pasa al iniciar la aplicación, justo después de la *splash screen*, es que se inicia la actividad del mapa. Posteriormente y en muy pocas milésimas se hacen las llamadas pertinentes a los servicios web, y se dibujan los distintos marcadores de todos los vehículos sobre el mapa. A su vez se muestra la herramienta de filtrado en la parte inferior izquierda. Se trata de un botón con el que podremos acceder posteriormente al filtrado de los diferentes servicios. A continuación se muestra el *mockup* diseñado.

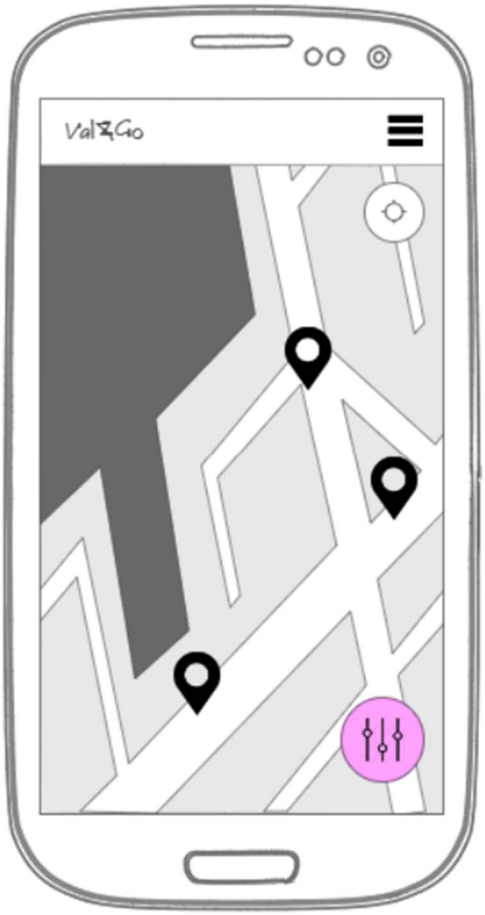

*Ilustración 10: Mockup de la pantalla de mapa*

Como podemos observar en el diseño anterior también aparece un segundo botón flotante, o **Floating Action Button (FAB),** que en este caso sirve para geolocalizar nuestra posición. Si el usuario lo pulsa comienza una animación de movimiento de la cámara que termina en la posición actual del usuario mostrándose como un icono azul (al estilo Google Maps).

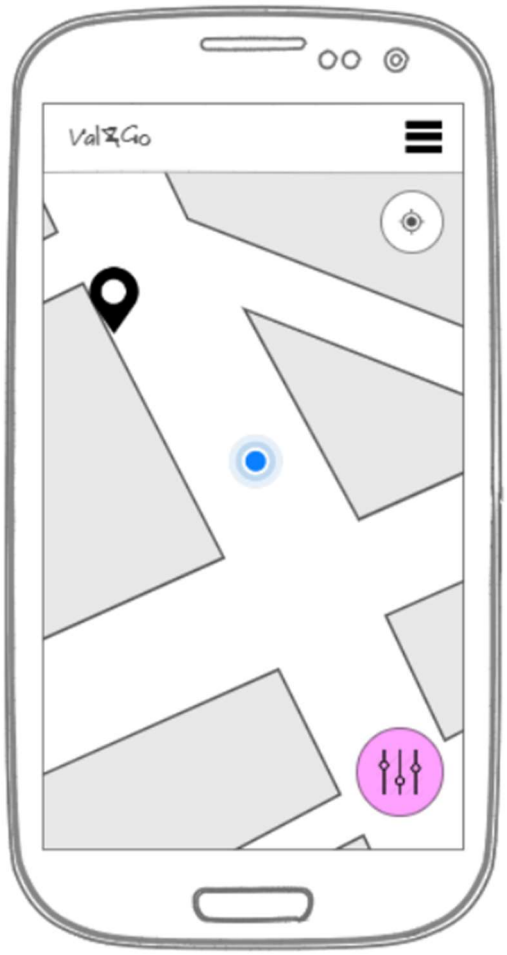


*Ilustración 11: Mockup de la localización GPS*

Como podemos deducir, es importante que el usuario tenga activados los servicios de geolocalización GPS, ya que muchas de las funcionalidades de la aplicación, aunque no todas, dependen de este servicio. El hecho de que el icono sea parecido al de la aplicación Google Maps es muy importante, ya que esto facilita que el usuario identifique instantáneamente que se trata de su ubicación, ya que ya esta familiarizado con este tipo de interfaz.

El siguiente paso natural del usuario será localizar un vehículo, normalmente un vehículo cercano y de la plataforma de su preferencia. Una vez localizado el vehículo, el usuario procederá a presionarlo. En este momento pasan varias cosas. La primera es que se abrirá un cuadro de diálogo o de información desde la parte inferior de la pantalla. Esto es lo que se conoce como *BottomSheet*, una librería de la API de diseño de

Andoird. Este cuadro va a contener toda la información del vehículo que esta internamente almacenada en un objeto de la case **MarkerInfo**. Se trata de la carga del vehículo, la cantidad de cascos, la distancia en minutos, el precio que nos puede costar por minuto, e información sobre su matrícula o nombre. Esto último es muy importante para que el usuario pueda identificar el vehículo con más facilidad una vez se acerca a él.

Lo siguiente que sucede al presionar un marcador del mapa es que se hace una llamada a la API gratuita **OpenRouteService.** Esta plataforma ofrece un servicio de direcciones y rutas. Esto quiere decir que proporcionando dos puntos geográficos del globo nos puede devolver la distancia más corta que hay entre ellos. Esto puede servir, como es obvio, para dibujar una ruta sobre el mapa entre el punto actual del usuario y el vehículo que este ha seleccionado. Una vez obtenido el resultado de la llamada se dibuja una línea animada sobre el mapa y se hace zoom hacia esta ruta. A continuación se muestra el *mockup* descrito.


*Ilustración 12: Mockup de la animación de ruta y el cuadro de información*

En el cuadro informativo inferior se muestra el logo de la plataforma correspondiente en la esquina superior izquierda. En la parte derecha aparece una

imagen real del vehículo seleccionado. Esto ayuda mucho al usuario a localizar con más rapidez y certeza el vehículo. En la parte inferior del logo de la plataforma aparecerán los distintos parámetros; por simplicidad no se han representado todos. Por último, en la parte inferior, aparecen dos botones. El primero redirige a la aplicación original del servicio para poder hacer la reserva del vehículo. El segundo redirige al usuario a la aplicación Google Maps, donde aparecerá un marcador en el punto exacto donde se encuentra el vehículo. El usuario tiene la opción de deslizar este cuadro de diálogo hacia abajo para descartarlo. Cualquier gesto sobre el mapa también descarta el cuadro y desmarca el vehículo seleccionado.

Otra funcionalidad importante de esta aplicación es la posibilidad de filtrar los vehículos por proveedor. Esto permite que ocultemos todos los vehículos de un proveedor que no utilizamos o no nos interesa. Esta tarea se hace a través de la interfaz de una forma sencilla y fluida, respetando los principios de diseño de Android. En la parte inferior derecha de la pantalla se muestra un botón flotante que, al presionarlo, abrirá un menú desplegable de botones flotantes. Cada botón desplegado corresponde a una compañía diferente, con su propio logotipo y color corporativo. Al presionar uno de estos botones automáticamente se ocultan los vehículos de este servicio y el botón pasa a un estado deseleccionado. Si queremos volver a mostrar los vehículos solo tenemos que volver a presionar el botón.

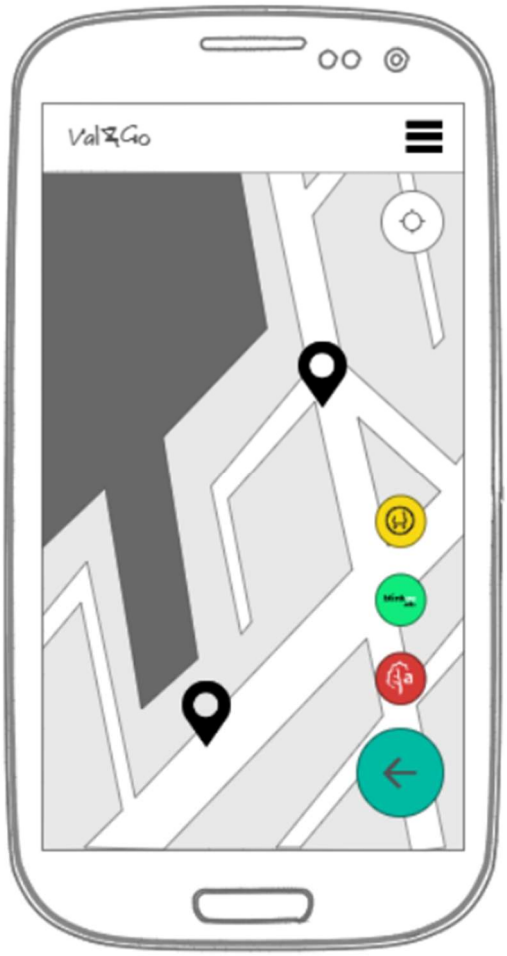


*Ilustración 13: Mockup del menú de filtrado*

El usuario también tiene la opción de visualizar una lista con los vehículos más cercanos a su posición. Esto le puede facilitar la elección del vehículo que mas se ajuste a sus necesidades. Esta lista se encuentra en una actividad diferente a la cual se accede a través del botón de la barra de herramientas superior. Una vez en la actividad se muestra la lista, con los vehículos ordenados de más cercanos a más lejanos. Concretamente se muestran los 20 vehículos más cercanos.

Cada elemento de la lista contiene el logo de la plataforma, la matrícula del vehículo en grande, la carga del vehículo y su cercanía en metros (igual que en el cuadro de diálogo anteriormente mostrado). El usuario tiene la opción de presionar cualquier elemento de la lista, en cuyo caso se volverá automáticamente a la actividad principal del mapa y se enfocará el marcador correspondiente al vehículo seleccionado. La lista se trata de una **RecyclerView** de la API de Android. El principio básico de esta lista es que se reciclan los elementos al hacer s*croll,* es decir, solo existen los elementos mostrados en pantalla y se van reutilizando para dar la sensación de que existen más elementos.

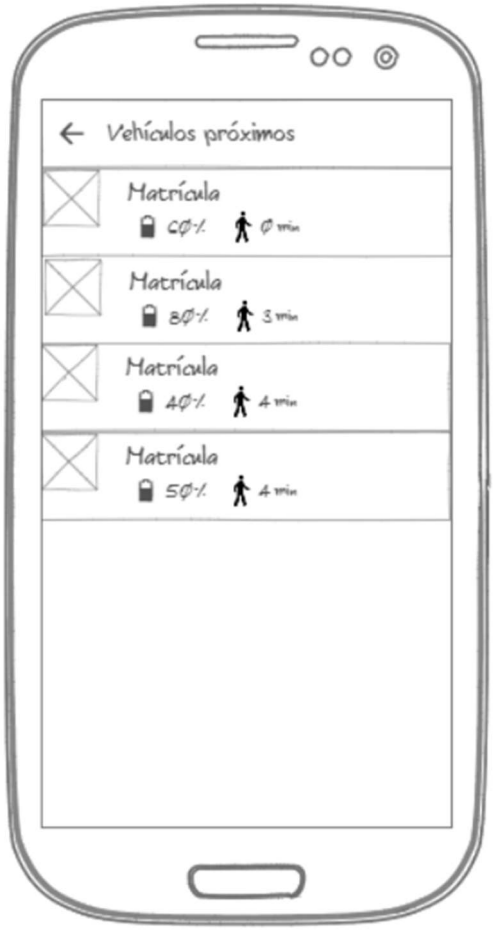


*Ilustración 14: Mockup de la lista de vehículos cercanos*

### 4.2.2 Diseño de componentes

En este apartado se va explicar a grandes rasgos como ha sido el proceso de diseño gráfico de componentes de la aplicación, así como diseño de logotipos, imágenes, y animaciones. Este apartado dista de las prácticas referentes a la ingeniería de software; aun así, es un proceso necesario y muy importante en el desarrollo de una aplicación móvil. Además, el diseño gráfico ha sido un proceso que ha ocupado mucho tiempo de este proyecto, ya que uno de los objetivos principales de este trabajo es que la aplicación sea gráficamente agradable y profesional. De nada sirve crear un buen producto software si no atrae estéticamente al usuario.

Val&Go está enfocada al público en general, pero sobre todo al público que más utiliza el trasporte público o que esta familiarizado con las nuevas formas de movilidad. Es por esto que el diseño intenta ser simple, fresco y fluido. Se han utilizado colores agradables, y se ha intentado seguir una coherencia cromática en el diseño de todos los componentes.

#### Imágenes y logotipos

Vamos a empezar con el diseño del logotipo principal de la aplicación. A pesar de haber sido lo último en ser diseñado, es lo primero que ve el usuario al instalar la aplicación. Se ha realizado con la herramienta online **Canva.** Sigue una línea de diseño muy sencilla y agradable. Se compone del nombre de la aplicación con una fuente vistosa y el color principal de toda la app, el verde. Además, se ha añadido un elemento de conexión de nodos, que representa la comunicación y la movilidad de las personas.


*Ilustración 15: Logotipo de la aplicación*

Como se puede observar el logotipo destaca por su simplicidad y la sensación de fluidez y dinamismo. Este logotipo se muestra tanto en el logotipo del lanzador de aplicaciones, como en la *splash screen*, que es la pantalla que se muestra justo al inicio de la aplicación y justo antes de la carga de la primera actividad.

El diseño de los logotipos de las distintas plataformas también ha sido importante. A pesar de que son logotipos ya diseñados, ha sido necesario ajustarlos al tamaño deseado y al color necesario. A parte, Android se ejecuta en una inmensa cantidad de dispositivos, todos con pantallas distintas y diferentes densidades y resoluciones. Esto hace que diseñar una imagen sea más complicado de lo que parece, puesto que ha de ser diseñada en muchos tamaños distintos para acoplarse a resoluciones distintas. Afortunadamente Android y Android Studio ofrecen diferentes mecanismos para facilitas estas peculiaridades.

Los elementos visuales de la interfaz se almacenan por convenio en una carpeta llamada *drawable.* En dicha carpeta se almacenan todas las imágenes y elementos que forman parte de la interfaz gráfica. Para respetar todas las resoluciones de pantalla, una imagen de mapa de bits o *bitmap* se tiene que depositar en varias carpetas diferentes.

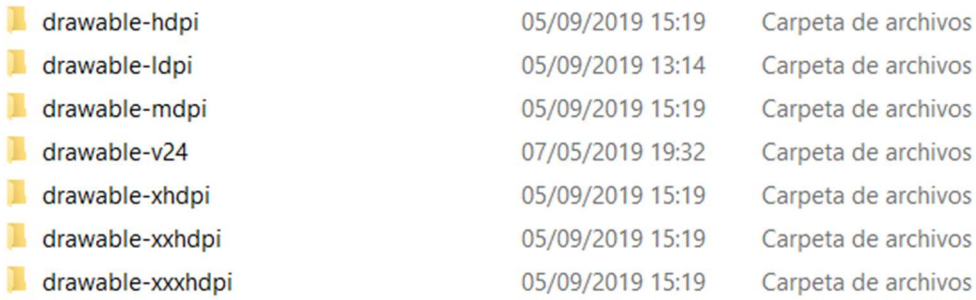


*Ilustración 16: Estructura de ficheros de los recursos drawable*

Como se observa en la figura existen hasta 7 carpetas diferentes para resoluciones de pantalla diversas. A continuación se muestra una tabla con la clasificación de las diferentes densidades de pantalla.

| **Calificador de densidad** | **Descripción** |
|---|---|
| ldpi | Recursos para pantallas de densidad baja (*ldpi*) (120 dpi) |
| mdpi | Recursos para pantallas de densidad media (*mdpi*) (~160 dpi; esta es la densidad de referencia) |
| hdpi | Recursos para pantallas de densidad alta (*hdpi*) (~240 dpi) |
| xhdpi | Recursos para pantallas de densidad muy alta (*xhdpi*) (~320 dpi) |
| xxhdpi | Recursos para pantallas de densidad muy, muy alta (*xxhdpi*) (~480 dpi) |
| xxxhdpi | Recursos para usos de densidad extremadamente alta (*xxxhdpi*) (~640 dpi) |

Por tanto, una imagen depositada en la carpeta xxxhdpi debe tener el tamaño ideal para mostrarse de forma correcta en pantallas de densidad 640 dpi, ya que es la imagen que se mostrará cuando el sistema detecte este tipo de densidad.

Las imágenes de esta aplicación estás redimensionadas con la herramienta online **Android Asset Studio.** Esta herramienta genera todas las imágenes en todos los tamaños necesarios a partir de la imagen original.

A continuación se muestra el logotipo de Muving en diferentes tamaños:


*Ilustración 17: Logotipo escalado a distintos tamaños*

## Elementos de diseño XML

A parte de las imágenes y los *bitmaps* convencionales, existen otras herramientas en Android para producir elementos gráficos, así como formas, componentes e iconos. En esta plataforma se utiliza el lenguaje XML para el diseño de algunos recursos gráficos (*drawables*). Por ejemplo, el diseño de los marcadores del mapa se ha hecho utilizando la etiqueta <*shape*> de la convención XML de Android. Esta etiqueta produce una figura geométrica a la que posteriormente se le puede añadir un bitmap. El lenguaje de etiquetas permite gestionar distintos parámetros, como el tamaño del elemento, la forma, y el grosor de los bordes.

Para el la representación del marcador de los diferentes vehículos se ha utilizado la forma de un cuadrado, girado 45 grados, y con 3 bordes redondeados al 100%. Esto da la sensación de un pin en el mapa. A continuación se muestra el resultado de esta figura:

*Ilustración 18: Diseño del marcador de Acciona Mobility*

Como se observa el resultado de crear una figura geométrica de este modo es un marcador con el logo de la plataforma en el medio, que a demás no es necesario reescalar para distintas densidades, puesto que se trata de un recurso escrito en XML.

```xml
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:height="34dp"
        android:width="34dp"
        android:gravity="center"
        tools:ignore="UnusedAttribute">
        <rotate
            android:fromDegrees="45"
            android:toDegrees="45"
            android:pivotX="50%"
            android:pivotY="50%" >
            <shape android:shape="rectangle">

                <corners
                    android:bottomRightRadius="5dp"
                    android:radius="40dp" />

                <gradient
                    android:angle="180"
                    android:endColor="@color/acciona"
                    android:startColor="@color/acciona"
                    android:type="linear" />

            </shape>
        </rotate>
    </item>
    <item
        android:drawable="@drawable/ic_acciona_logo"
        android:gravity="center"
        android:height="30dp"
        android:width="30dp"
        tools:ignore="UnusedAttribute" />

</layer-list>
```

*Ilustración 19: Ejemplo de código del diseño*

En el código anterior se muestra como se construye el marcador. Se compone de una *<layer-list>,* que es una lista de elementos que se superponen entre sí. El primer elemento corresponde a la figura geométrica, en este caso el cuadrado retocado. El segundo corresponde a la imagen del logotipo de la plataforma. Se ha realizado el mismo proceso para todas las plataformas disponibles.

Otros elementos importantes que funcionan con el lenguaje de etiquetas XML son los iconos vectoriales. Estos se importan desde archivos .svg, que son archivos vectoriales escalables. Android Studio convierte estos archivos en archivos XML con la etiqueta principal <vector>. Estos elementos son escalables y por tanto tampoco es necesario crearlos para diferentes densidades de pantalla.

### Animaciones

Las animaciones son una parte muy importante de esta aplicación, ya que la convierten en una aplicación más dinámica, y es identificada por el usuario como una aplicación profesional y que ha costado esfuerzo desarrollar. Las animaciones de Android son un nivel bastante avanzado de la plataforma, y requieren un conocimiento amplio de como funcionan los eventos y los dibujados de los elementos en pantalla.

En el caso de Val&Go se han implementado distintos tipos de animaciones: la animación de ruta entre la ubicación del usuario y el vehículo, la animación de rebote cuando se selecciona un servicio para ser desactivado, o la animación de el botón reservar que nos recuerda que debemos pulsarlo y que es el elemento más remarcable en pantalla. Puesto que las animaciones requieren de grabación de video para ser mostradas, no es posible ofrecer una demostración de las animaciones implementadas en esta memoria, pero a continuación se muestra una imagen que puede ilustrar un poco la idea.

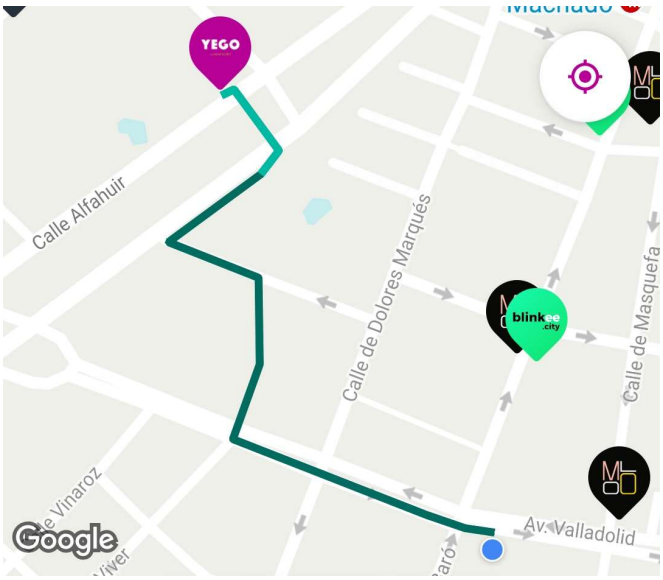


*Ilustración 20: Captura de la animación de ruta*

Como se observa en la ilustración anterior, la ruta entre la ubicación y el marcador del vehículo esta siendo dibujada. Este proceso es continuo, la línea se redibuja para dar la sensación de que está avanzando por el camino que tiene que seguir el usuario.

Estas animaciones se consiguen gracias a diferentes librerías de Android. La más relevante es ***ObjectAnimator***, que permite animar cualquier elemento de la pantalla representado por un objeto. Los tipos de animaciones son de movimiento, es decir, mover un objeto del punto x al punto y, de zoom, es decir, agrandar o empequeñecer un elemento de la pantalla, o de rotación, que consiste en rotar un elemento. A estos movimientos se les pasa el tiempo en el que tienen que producirse, normalmente en milisegundos.

La clase ***AnimationSet*** permite sincronizar diferentes animaciones, ya sea del mismo elemento o de elementos distintos. Esto es importante, ya que se pueden producir dos animaciones distintas en el mismo lapso de tiempo, por ejemplo, que un elemento se mueva de posición y al mismo tiempo se haga más grande.

# 5. Tecnología utilizada

En este capítulo de la memoria se va a explicar cuáles han sido las tecnologías utilizadas para el desarrollo del software. Se van a exponer todos los entornos, lenguajes y tecnologías que se han utilizado, así como sus principales características. Cabe destacar que para el desarrollo del proyecto se ha utilizado el sistema operativo Windows 10, y todas las herramientas mencionadas han sido instaladas en este sistema.

## 5.1 Android

Android es un sistema operativo para móviles desarrollado por Google y basado en el kernel de Linux. Desde sus inicios fue diseñado para dispositivos con pantalla táctil, como móviles y tabletas; pero en los últimos años han aparecido nuevos dispositivos inteligentes como televisores, relojes y automóviles.

Android es el sistema operativo para smartphones más utilizado del mundo con más del 80% de cuota de mercado, muy por encima de su competidor de Apple IOS.

Entre sus principales características destacan:

- La plataforma es adaptable a diferentes resoluciones de pantalla.
- Android soporta las siguientes tecnologías de conectividad: GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, HSDPA, HSPA+, NFC y WiMAX, GPRS, UMTS y HSDPA+.
- El desarrollo de sus aplicaciones es en Java, aunque posee una maquina virtual propia llamada *Dalvik* y basada en la máquina virtual de Java.
- Android tiene soporte para cámaras de fotos, de vídeo, pantallas táctiles, GPS, acelerómetros, giroscopios, aceleración por GPU 2D…
- Incluye la tienda Google Play, con uno de los mayores catálogos de aplicaciones del mundo

También es interesante hablar sobre las aplicaciones y su funcionamiento, y de como funciona su ciclo de vida. Las aplicaciones son un conjunto de actividades o a*ctivities,* un tipo de clase que esta ligado a una pantalla de la aplicación y que pasa por un ciclo de eventos. Al iniciar la actividad principal uno de los primeros métodos en ejecutarse es el método *onCreate()*. Es en este método por tanto donde debemos introducir las primeras operaciones de inicialización. Seguidamente se ejecutan más métodos, como el *onStart()*. Si la aplicación es minimizada pasa a un estado de pausa y pasa a segundo plano. En este momento se ejecuta el método *onPause()*, y es en él

donde debemos guardar la información que esté en pantalla para no perderla. Si cerramos la aplicación incluso de la multitarea, es decir, la matamos, se ejecutará el método *onDestroy()*. En la imagen siguiente se muestra el ciclo de vida de las actividades.

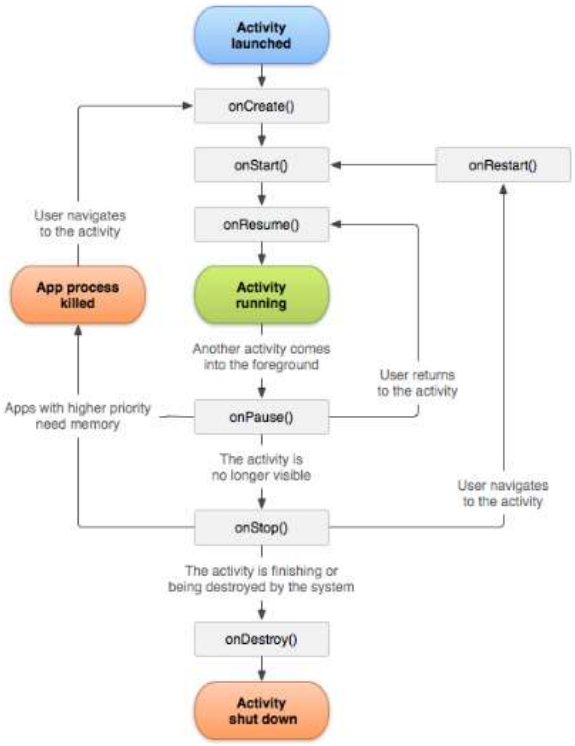


*Ilustración 21: Ciclo de vida de una actividad*

## 5.2 Android Studio

Android Studio es el IDE o entorno de desarrollo oficial para la plataforma Android. Esta basado en IntelliJ, de la empresa JetBrains, y se puede utilizar de forma gratuita. Estas son sus principales características:

- Renderizado en tiempo real
- Soporte para construcción basada en Gradle
- Refactorización específica para Android
- Editor de diseño grafico que permite manipular elementos arrastrándolo y soltándolos.
- Soporte para Android Wear y Android TV.

## 5.3 Java

Se trata de un lenguaje de programación orientado objetos, desarrollado por la empresa **Sun Microsystems**. Tiene una sintaxis basada en C, aunque se trata de un lenguaje de más alto nivel. Las aplicaciones Java se compilan a *bytecode* y posteriormente son ejecutadas en la máquina virtual de Java (JVM), sin importar cual sea la arquitectura del equipo. El lenguaje se diseñó con unos principios primordiales, que se exponen a continuación:

- Java es un lenguaje con un paradigma orientado a objectos.
- Debe permitir la ejecución de una misma aplicación en múltiples equipos.
- Incluye por defecto soporte para trabajo en red
- Java debe ser fácil de usar y tomar lo mejor de otros lenguajes orientados a objetos como C++.

En el caso de desarrollo en Android, además de todas las librerías y APIs de Java también están disponibles múltiples librerías específicas de Android, escritas también en Java. Se podría decir que la API de Android complementa a la API de Java.

## 5.4 XML

XML es un metalenguaje que permite definir lenguajes de marcas para almacenar o transmitir información. Por tanto, una de las ventajas de este lenguaje es que es extensible, es decir, se pueden crear nuevas etiquetas a nuestro gusto. En el caso de Android, XML se utiliza como herramienta principal para los elementos gráficos, sobre todo para definir imágenes, figuras, iconos, y animaciones. También se utiliza para toda la estructuración de ventanas, los denominados *layouts*. Por tanto los elementos de las ventanas se distribuyen mediante etiquetas XML propias, como se haría por ejemplo para construir una página web con HTML. Este lenguaje también se utiliza para la definición de recursos como *Strings,* colores, dimensiones, o estilos; es decir, valores de todo tipo. A continuación se muestra un ejemplo de como se utiliza XML para la construcción de elementos en Android.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="176dp"
    android:background="@drawable/gradient_bar"
    android:gravity="bottom"
    android:orientation="vertical"
    android:paddingLeft="16dp"
    android:paddingTop="16dp"
    android:paddingRight="16dp"
    android:paddingBottom="16dp"
    android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:contentDescription="Val&Go"
        android:paddingTop="8dp"
        app:srcCompat="@mipmap/ic_launcher_round" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="8dp"
        android:text="Val&Go"
        android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Val&Go" />

</LinearLayout>
```

*Ilustración 22: Código XML de la estructura de una ventana*

Se puede observar como las distintas etiquetas se anidan y forman parte del mismo elemento. Se observan algunas como *ImageView,* que se trata de un contenedor donde aparecerá una imagen, o *TextView*, que se trata de un simple campo de texto.

## 5.5 API de Google Maps

Una de las tecnologías más importantes para este proyecto es la API de Google Maps. Esta API nos ofrece muchos servicios, algunos gratuitos y otros de pago. El principal y el más importante es el que dibuja el mapa sobre la aplicación y contiene toda la información de calles y lugares de interés. A parte de este elemento también se proporciona una librería muy amplia de clases que sirven para manejar el mapa y operar sobre él, como por ejemplo dibujar un marcador o hacer una animación a otra localización.

Otro de los servicios que ofrece esta API es la geolocalización, que permite averiguar la ubicación del usuario en tiempo real y dibujarla sobre el mapa. Estos servicios ya mencionados son gratuitos para la plataforma Android.

Los servicios de direcciones y rutas no son gratuitos, es por esto que para este proyecto se ha optado por elegir un servicio llamado **OpenRouteService,** que ofrece una API de direcciones completamente gratuita, pero con una limitación de peticiones mensual. Este servicio se ha utilizado principalmente para el dibujado de ruta entre el usuario y el vehículo seleccionado.

# 6. Implementación

En este apartado se van a explicar los procesos que se han seguido para implementar la aplicación. En primer lugar se explicarán a grandes rasgos los trabajos de Ingeniería Inversa realizados para recabar información sobre los servicios web de la plataforma. Seguidamente se explicarán los procesos de desarrollo de la aplicación con la plataforma Android Studio, y se expondrán partes del código de interés. Por último se mostrará el resultado final de la aplicación y se explicará su funcionamiento.

## 6.1 Ingeniería inversa

Como ya se ha explicado en el análisis legal (apartado 3.3), la ingeniería inversa es una práctica legal en España. Según el artículo 100 de **la Ley de Propiedad Intelectual** el usuario tiene el derecho a entender el funcionamiento de un producto software como propietario legítimo de una copia de este, siempre que el objetivo sea conseguir la interoperabilidad. Es esto último precisamente lo que se intenta conseguir en este proyecto.

El proceso seguido para el desempeño de esta técnica ha sido, para todas las plataformas y servicios, la inspección del código de las aplicaciones originales y la comprensión de su funcionamiento. A partir de este estudio, que por otra parte es bastante costoso y tedioso, se ha conseguido recabar la información necesaria (tipo de conexión, *URLs*, protocolos) para poder establecer una conexión con los servicios web originales.

### 6.1.1 ApkTool

Se trata de una herramienta creada por desarrolladores poco conocidos que consigue decodificar los recursos de la aplicación y devolverlos a su forma original exacta. Facilita el trabajo creando una estructura de archivos exacta a la de un proyecto hecho con Android Studio. Estas son sus características principales:

- Desensambla recursos originales a su forma casi original, incluyendo imágenes, XML, y archivos binarios .dex.
- Recompila recursos decodificados otra vez en un archivo APK.
- Ayuda en tareas repetitivas.

Esta herramienta está disponible para Windows, y se trata de una aplicación de línea de comandos. A pesar de su gran utilidad en los procesos de ingeniería inversa, no consigue extraer el código Java original, y solo transforma los archivos binarios en un lenguaje intermedio y complejo llamado *Smali*. Por tanto, solo con esta herramienta es muy difícil hacer un análisis completo del código.

### 6.1.2 Dex2Jar

Se trata de una herramienta que, a partir de un archivo de aplicación APK consigue traducir de manera precisa los binarios *.dex* en archivos *.class* de Java, empaquetados en archivos JAR. Este archivo contiene todo el código fuente de la aplicación en Java. Al igual que ApkTool también se trata de una aplicación de línea de comandos para Windows.

El archivo JAR obtenido es descomprimido con herramientas específicas. El resultado de esto es la obtención del código fuente, que posteriormente se puede inspeccionar con herramientas de búsqueda avanzada como las que tiene Eclipse.

### 6.1.3 Resultados obtenidos

A partir de estas herramientas podemos inspeccionar el código original e intentar comprender su funcionamiento. Esta es una tarea tediosa, pues la mayoría de códigos están minimizados y los nombres de clases y atributos son letras del alfabeto. Esto se traduce en que leer el código sea muy complicado, y no se puedan seguir con facilidad las distintas referencias a clases u objetos.

Otra dificultad al llevar a cabo estas tareas es que hoy en día la mayoría de desarrolladores optan por la utilización de *frameworks* y librerías que les permiten diseñar la aplicación en diferentes lenguajes de programación. Este ha sido el caso de algunas aplicaciones estudiadas, muchas de ellas diseñadas en Javascript o en C Sharp. Esto significa que el código es todavía más complicado de leer y seguir.

El modelo de red de la mayoría de plataformas es casi siempre el mismo. Normalmente se trata de APIs REST, esto quiere decir que se establece la conexión mediante el protocolo HTTP con peticiones GET. En la URL se almacenan los distintos parámetros necesarios, como por ejemplo la ciudad que queremos que nos devuelva. Posteriormente el servidor devuelve en la mayoría de casos un objeto JSON con toda la información filtrada de los vehículos. Se trata normalmente de un array de elementos en el que cada elemento es un objeto que representa un vehículo. Cada objeto contiene atributos como un id, la carga, la posición en coordenadas, o el estado del vehículo.

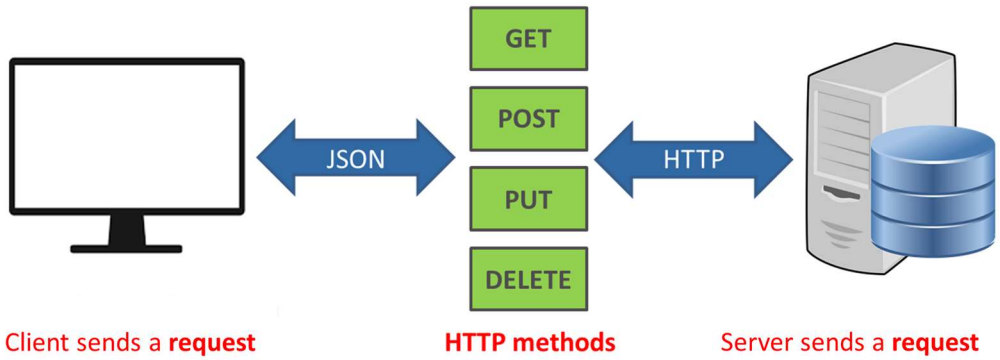


*Ilustración 23: Arquitectura de red de una API*

Afortunadamente, insistiendo mucho e invirtiendo tiempo se ha podido conseguir la interoperabilidad con todas las plataformas. Algunas han sido más complicadas que otras, pero al fin y al cabo se ha conseguido uno de los objetivos más importantes del desarrollo de esta aplicación.

## 6.2 Implementación del código

La escritura e implementación del código de la aplicación también ha sido una de las partes más costosas del proyecto. Se ha invertido mucho tiempo en esta parte, y el resultado de esto es un total de 25 clases Java creadas, algunas de ellas con mas de 1.000 líneas de código. En el apartado *Diseño* de este proyecto ya se ha explicado la estructura de las clases y su diseño, por lo que en este apartado solo se explicarán algunos detalles del código y de algunas implementaciones interesantes.

### 6.2.1 MainActivity

Es la primera actividad que se ejecuta al iniciar la aplicación. No posee ninguna interfaz asociada, puesto que su único objetivo es comprobar que están activos los servicios de internet, GPS, y los servicios de Google Play. Este último servicio se utilizará posteriormente para geolocalizar al usuario. Si el usuario no tiene los servicios activados se le preguntará si quiere activarlos y si se niega en algunos casos no se puede utilizar la aplicación, por lo que esta se cerrará. A pesar de esto son servicios que todo el mundo suele tener activados y que no suelen dar problemas. Una vez se han realizado todas estas comprobaciones se lanza un **Intent**. Se trata de un servicio que posee Android para lanzar otra actividad y que informa al sistema de ello. Si se desea se pueden guardar parámetros en el objeto Intent, pasándose de una actividad a otra objetos simples.

### 6.2.2 MapsActivity

Se trata de la clase principal de la aplicación y es la clase manejadora de la pantalla del mapa. Contiene mas de 1.000 líneas de código, 26 métodos, y 15 atributos. Se trata por tanto de la clase más extensa y donde se implementan todas las funcionalidades principales de la aplicación.

En el método ***onCreate()***, que se ejecuta al crearse la actividad, está localizada toda la inicialización de elementos visuales, componentes, y también la inicialización de las variables principales. Una vez el mapa está disponible se ejecuta el método *onMapReady(GoogleMap)*, el cual pasa el mapa como un objeto y se pueden hacer operaciones con él. Las operaciones básicas que se pueden hacer son mover la cámara de sitio, añadir un marcador al mapa, o dibujar figuras geométricas como líneas o rectángulos.

Existen múltiples métodos que se encargan de ratificar que todos los servicios de ubicación tienen permiso para ejecutarse. A partir de Android 6.0 la aplicación ha de preguntar explícitamente al usuario si permite estos tipos de servicios. A continuación se muestra el cuadro de diálogo típico utilizado:

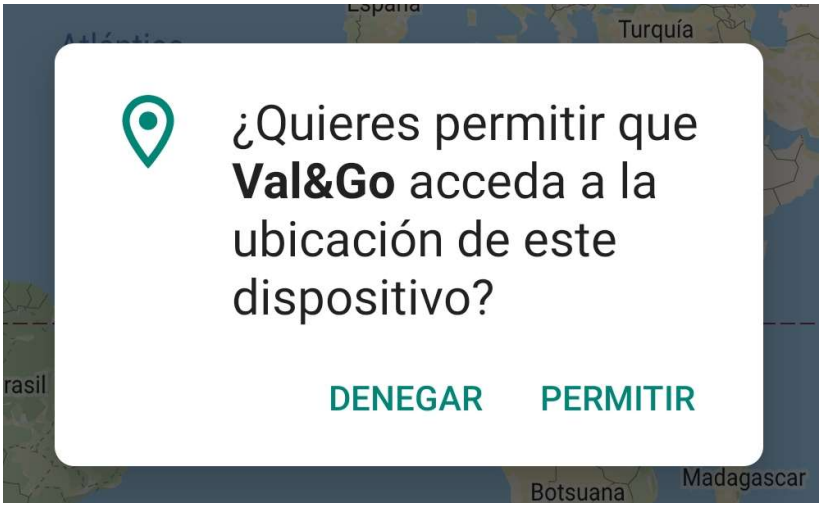


*Ilustración 24: Cuadro de diálogo del permiso GPS*

Una vez resueltos estos permisos el mapa se ejecuta con normalidad, los vehículos se dibujan sobre él y aparece un punto azul con nuestra posición.

Otro método destacado es el que se ejecuta cuando un marcador es presionado. Este método se llama *onMarkerClick()*. En este método se implementan varias cosas. La primera es que el marcador se cambia de color para que dé la sensación de que está seleccionado. La segunda es que se rellena el cuadro de dialogo inferior con la información de el vehículo y los iconos necesarios. La tercera es que se ejecuta el dibujado de la ruta, haciendo primero una llamada a la API de direcciones y rutas y

ejecutando después un método que dibuja y anima líneas desde la posición del usuario hasta la del vehículo. A continuación se muestra una parte de este método.

```
@Override
public boolean onMarkerClick(Marker marker) {
    if (marker.getTag() != null) {
        //Desmarca el ultimo marker clickado
        if (clickedMarker != null) {
            String comp = ((MarkerInfo) clickedMarker.getTag()).getCompany();
            clickedMarker.setIcon(BitmapDescriptorFactory.fromBitmap(createClickedIcon( m: null, comp)));
            clickedMarker.setZIndex(0);
            MapAnimator.getInstance().resetPolylines();
        }

        drawRoute();

        //Actualiza el nuevo marker
        clickedMarker = marker;
        clickedMarker.setIcon(BitmapDescriptorFactory.fromBitmap(createClickedIcon(clickedMarker,  company: null)));
        clickedMarker.setZIndex(1);

        MarkerInfo m = (MarkerInfo) clickedMarker.getTag();

        ImageView logo = findViewById(R.id.imageViewLogo);
        ImageView vehicle = findViewById(R.id.imageViewVehicle);

        int idLogo = getResId( resName: "ic_" + m.getCompany().toLowerCase() + "_logo_large", R.drawable.class);
        int idVehicle = getResId( resName: "ic_" + m.getCompany().toLowerCase() + "_moto", R.drawable.class);

        logo.setImageResource(idLogo);
        if(!m.getCompany().toLowerCase().equals("valenbisi")) {
            vehicle.setImageResource(idVehicle);
        }
```

*Ilustración 25: Trozo del método onMarkerClick()*

Se puede apreciar en la imagen que antes de nada lo primero es desmarcar el marcador anterior, que queda almacenado en el atributo de clase *clickedMarker,* y se sustituye por el nuevo marcador *clickado*. También se cambia su icono por uno de otro color para que haga el efecto de que está seleccionado. Posteriormente se llama al método *drawRoute(),* que inicia la comunicación con el servicio web y posteriormente dibujará la ruta en el mapa. También se observa como se recupera el objeto *MarkerInfo* del marcador. Se trata de un objeto simple que almacena toda la información del vehículo que se ha recogido de la API. Mediante el método *setTag()* de la clase *Marker* se le puede asociar al marcador cualquier objeto que descienda de *Object.* Posteriormente se utilizarán los *getters* y *setters* de la clase *MarkerInfo*, para recuperar toda la información que ha de ser mostrada en pantalla.

Otro método destacable de esta clase es el método llamado *getNearMarkersInfo()*, que tiene como objetivo crear una lista de los 20 vehículos mas cercanos a la posición del usuario. Esta lista se utiliza para pasársela a la actividad de vehículos cercanos, accesible a través del botón de la barra superior. Para la realización de esta tarea se ha utilizado un algoritmo lo más eficiente posible. Los objetos que hacen referencia a los marcadores están almacenados en una lista de la clase ServiceRequest, la cual está instanciada y es un atributo de MapsActivity. Se recorre esta lista y se calcula la distancia al usuario de cada marcador. A partir de aquí los vehículos mas cercanos se van almacenando en una lista por orden utilizando un

algoritmo de ordenación. Para almacenar el resultado se utiliza un ArrayList que ha sido modificado para que solo admita un máximo de 20 elementos, sobrescribiendo el método *add()*. Mientras se recorre la lista con todos los marcadores del mapa se van almacenando en la lista resultado dependiendo de su distancia al usuario. Si el marcador tiene una distancia mayor que el primer elemento de la lista se pasará a introducirlo en el segundo, y así sucesivamente hasta llegar a los 20 elementos. Si el marcador actual está más lejos que el último elemento de la lista (si ya está llena), se descartará puesto que ya existen 20 marcadores más cercanos. A continuación se muestran tanto la *ArrayList* modificada "en línea", como el algoritmo de ordenación diseñado.

```
ArrayList<MarkerInfo> mIList = new ArrayList<MarkerInfo>(){

    @Override
    public boolean add(MarkerInfo markerInfo) {

        if(size() >= 30){
            return false;
        }

        return super.add(markerInfo);
    }

    @Override
    public void add(int index, MarkerInfo element) {

        if(size() >= 30){
            remove( index: size() - 1);
        }

        super.add(index, element);
    }


};
```

*Ilustración 26: Código del ArrayList limitado a 20 elementos*

```
for(Marker m:enabledMarkers) {

    dist = distanceToLocation(m);
    mI = (MarkerInfo) m.getTag();
    mI.setDistanceToLocation(dist);

    if(mIList.isEmpty()) {
        mIList.add(mI);
        continue;
    }

    for(int i = mIList.size() - 1; i >= 0; i--){

        if(dist < mIList.get(i).getDistanceToLocation()){

            if(i == 0 || dist > mIList.get(i-1).getDistanceToLocation()) {
                mIList.add(i, mI);
            }

        } else {
            mIList.add(mI);
        }

    }

}
```

*Ilustración 27: Algoritmo utilizado para la ordenación de los Markers*

### 6.2.3 NetworkAPI

Se trata de una clase abstracta de la cual heredan tantas clases como plataformas de *sharing* hay. Contiene el método abstracto *call()* no implementado, ya que es este método el que implementarán las subclases para hacer peticiones a las distintas APIs. Contiene una lista llamada *myMarkers*, que es común a todas las subclases y en ella se almacenan los marcadores de las distintas plataformas. También contiene el objeto *client* de la clase OkHttpClient. Este objeto es el encargado de hacer peticiones HTTP a los diferentes servicios web.

Por otra parte están las clases *MuvingAPI, BlinkeeAPI, YegoAPI, EcooltraAPI, AccionaAPI*, y *MoloAPI.* Todas ellas extienden de la clase *NetworkAPI*, y corresponden a los diferentes servicios disponibles. Cada clase implementa el método *call()*. Existen implementaciones muy diversas puesto que cada servicio web funciona de una manera distinta y tiene distintos modos de intercambiar la información. No se va a entrar en más detalle de la implementación de estas llamadas, pero básicamente se hacen peticiones HTTP con una URL y devuelve un objeto JSON. Este objeto JSON es recorrido y la información de cada vehículo es almacenada en las diferentes listas de cada clase. Como ya hemos explicado antes es posible acceder a estas listas desde la clase principal para hacer distintas operaciones de recorrido sobre los marcadores y su información. A continuación se muestra un fragmento de la clase donde se pueden observar los métodos mencionados anteriormente.

```
public abstract class NetworkAPI {

    GoogleMap mMap;
    Context context;
    OkHttpClient client;
    List<Marker> myMarkers;
    Bitmap markerBitmap;
    Bitmap dotBitmap;
    ServiceRequest serviceRequest;

    NetworkAPI(GoogleMap mMap, Context context, OkHttpClient client, ServiceRequest serviceRequest) {
        this.mMap = mMap;
        this.context = context;
        this.client = client;
        this.serviceRequest = serviceRequest;
        this.myMarkers = new ArrayList<>();
    }

    public abstract void call();

    public List<Marker> getMarkers() { return myMarkers; }

    public void setMarkers(List<Marker> myMarkers) { this.myMarkers = myMarkers; }

    public Bitmap getMarkerBitmap() { return markerBitmap; }

    public Bitmap getDotBitmap() { return dotBitmap; }
```

*Ilustración 28: Trozo de la clase NetworkAPI*

### 6.2.4 MarkerInfo

Esta es la clase que representa cada marcador o vehículo con toda su información almacenada en atributos. Se trata de un **POJO**, es decir, un ***Plain Old Java Object.*** Este tipo de clases se caracterizan por contener un constructor, atributos, *getters* y *setters*, y a parte no heredan de ninguna otra clase. Es por ello que es un tipo de clase muy simple cuyo único objetivo es almacenar información, en este caso la información de cada vehículo que aparecerá en el mapa.

Los atributos que contiene esta clase son *id, lat, lon, info, charge, adress, zipcode, city, helmets, y company,* que almacenan respectivamente el id del vehículo, su latitud, su longitud, información como la matrícula, carga, dirección, código postal, ciudad, número de cascos, y compañía a la que pertenecen. Ademas de estos atributos incluye todos sus *getters* y *setters,* y la implementación del método toString(). Se trata de un objeto "serializable", ya que se intercambia por mensaje entre actividades distintas.

Los objetos de esta clase se añaden como etiqueta a los objetos *Marker* que representan los vehículos en el mapa. Esto quiere decir que estas dos clases van cogidas de la mano y que se puede recuperar un objeto *MarkerInfo* teniendo un objeto *Marker*. A continuación se muestran los atributos y el constructor de la clase.

```
public class MarkerInfo implements Parcelable {
    private String id;
    private double lat;
    private double lon;
    private String info;
    private int charge;
    private String address;
    private String zipCode;
    private String city;
    private int helmets;
    private int range;
    private int totalBikes;
    private int availableBikes;
    private int availableStands;
    private String company;
    private double distanceToLocation;

    public MarkerInfo(String id, double lat, double lon, String info, int charge, String address, String zipCode, String city, int helmets,
        this.id = id;
        this.lat = lat;
        this.lon = lon;
        this.info = info;
        this.charge = charge;
        this.address = address;
        this.zipCode = zipCode;
        this.city = city;
        this.helmets = helmets;
        this.range = range;
        this.totalBikes = totalBikes;
        this.availableBikes = availableBikes;
        this.availableStands = availableStands;
        this.company = company;
        this.distanceToLocation = 0;
    }
```

*Ilustración 29: Atributos y constructor de la clase MarkerInfo*

## 6.3 Resultado final

En este apartado se va a exponer el resultado final de la aplicación acabada, analizando su funcionamiento final y su apariencia mediante imágenes.

La primera pantalla que se muestra al iniciar la aplicación es la pantalla de carga, donde aparece el logotipo de la aplicación y el color principal de fondo. Si el dispositivo utilizado para iniciar la aplicación tiene mucha potencia esta pantalla no durará más de 1 segundo, pues desaparece cuando la primera actividad se carga con éxito. A continuación se muestra una captura de dicha pantalla.


*Ilustración 30: Splash Screen de la aplicación*

Una vez la actividad principal se ha cargado aparece el mapa. Posteriormente se dibujan los marcadores sobre el mapa, se localiza al usuario, y se centra la cámara en la posición de este. También están a nuestra disposición las herramientas de filtrado y la barra de herramientas superior.

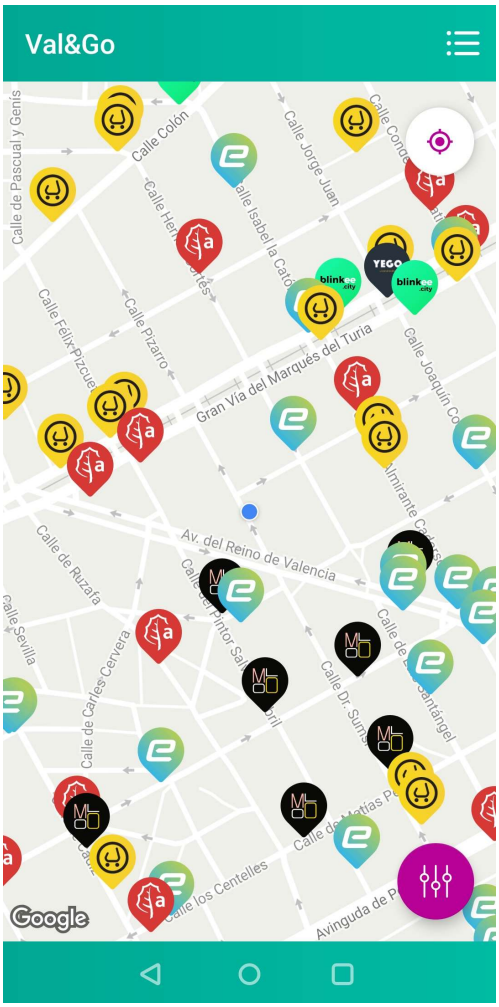


*Ilustración 31: Mapa con marcadores y usuario geolocalizado*

Como se observa en la imagen anterior aparecen múltiples tipos de marcadores, cada uno con un logotipo diferente y un color distinto. El usuario tiene la opción de elegir cualquiera de estos vehículos, siendo los más interesantes los vehículos más cercanos o con más autonomía.

También se pueden observar dos botones flotantes sobre el mapa. El de la parte superior derecha sirve para centrar la cámara en nuestra posición. Se utiliza si nos hemos desplazado muy lejos de nuestra posición y queremos volver de forma rápida sin tener que buscar el icono azul. El botón de la parte inferior derecha se utiliza para desplegar las herramientas de filtrado, que aparecerán superpuestas sobre el mapa. El botón de la barra de herramientas sirve para entrar en la actividad de listado de vehículos cercanos.

Por otro lado, podemos ver como la barra de navegación inferior se tiñe del color de la aplicación. Esto es meramente estético, pero mejora la congruencia de colores y de estilo. La barra de notificaciones, que no aparece en la imagen, también se tiñe del mismo color.

La herramienta de filtrado consta de una serie de botones desplegados con una animación fluida que contienen el logotipo y el color de la compañía a filtrar. A continuación se muestra una imagen del menú desplegado.

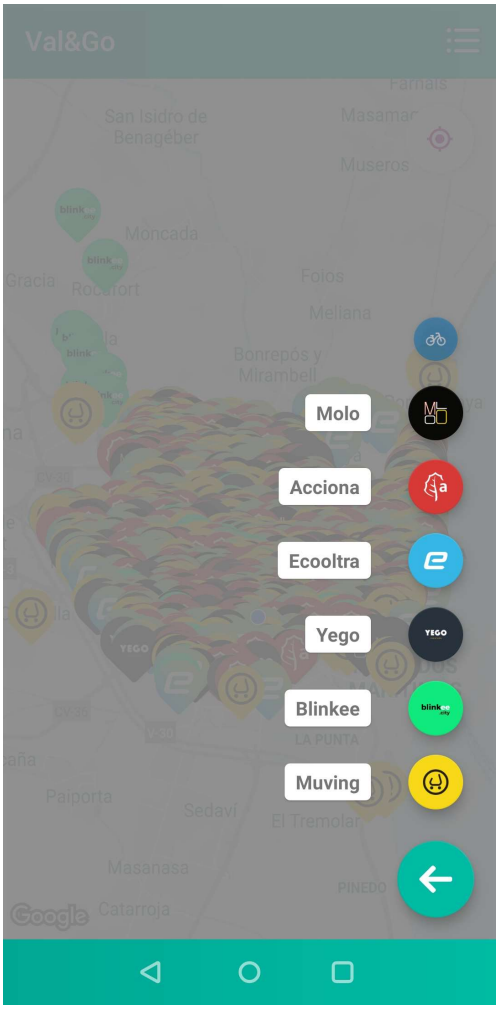


*Ilustración 32: Menú de filtrado desplegado*

Como se aprecia el menú flotante contrasta mucho con el fondo. Esto es importante para mantener la coherencia y no distraer al usuario de los elementos que son más importantes en este momento. Aparecen tantos botones como servicios contiene la aplicación. Una vez seleccionemos uno o varios de estos elementos los marcadores del mapa desaparecerán automáticamente. Para volver atrás solo tenemos que pulsar el mismo botón que hemos utilizado para abrirlo, solo que ahora tiene un icono y color diferentes. A continuación se muestra cómo queda la deselección de alguno de los servicios.

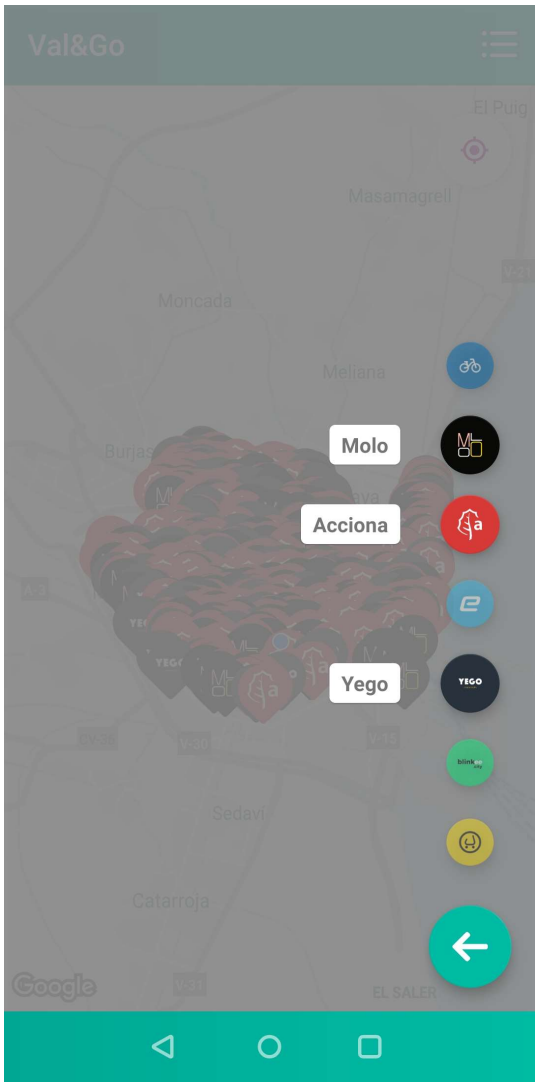


*Ilustración 33: Elementos del menú deseleccionados*

Como se observa los servicios que ya no están activos cambian su tamaño y su color, haciéndose más tenues y dando la sensación de no estar seleccionados. También desaparece la etiqueta con el nombre de la empresa. Por el contrario, los servicios que todavía hay seleccionados quedan visibles y llaman la atención.

Una vez hemos acabado de seleccionar los servicios que nos interesan lo que debemos hacer es presionar el botón con la flecha, que plegará el menú flotante y se transformará en el botón inicial.

El resultado de este proceso es que solo nos aparecerán los vehículos de las plataformas que hemos elegido a nuestro gusto. Esta situación se puede observar en le imagen siguiente.

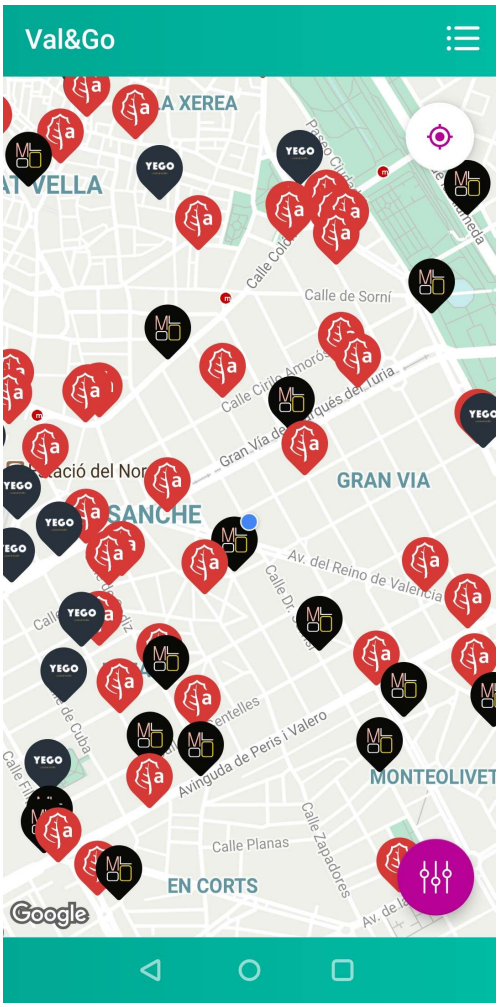


*Ilustración 34: Resultado del filtro aplicado*

El mapa queda mucho más despejado y solo quedan los servicios que nos interesan. Esta es una herramienta muy útil, pues es poco probable que los usuarios utilicen todos los servicios o estén registrados en todos ellos.

Una vez hayamos localizado el vehículo que nos interese solo debemos presionar sobre él. En ese momento nos aparecerá un cuadro de información por la parte inferior con todos los datos del vehículo, su carga, su distancia, su matrícula y también el precio que nos cobra esa compañía por el servicio. Esto último es muy útil ya que permite al usuario comparar precios y encontrar el servicio que más se ajuste a su bolsillo. Además, también se muestra en este cuadro una imagen real del vehículo en cuestión. Con esto se consigue que el usuario identifique mejor el vehículo en la vía urbana. También aparece el logotipo de la empresa.

Aparecen dos botones en la parte inferior de este cuadro. El primero con el nombre *Reservar,* y su acción es abrir la aplicación original del servicio para que el

usuario pueda reservar el vehículo o registrarse si todavía no lo está. El segundo abre la aplicación Google Maps y coloca un marcador en la posición exacta del vehículo. Esto resulta muy útil por si se quiere utilizar las herramientas de la aplicación de Google, como por ejemplo el navegador.

Lo siguiente que pasa al presionar el vehículo es que se muestra la ruta más corta andando. Como ya se ha explicado anteriormente esto se hace mediante una animación fluida. A continuación se muestra el escenario descrito.

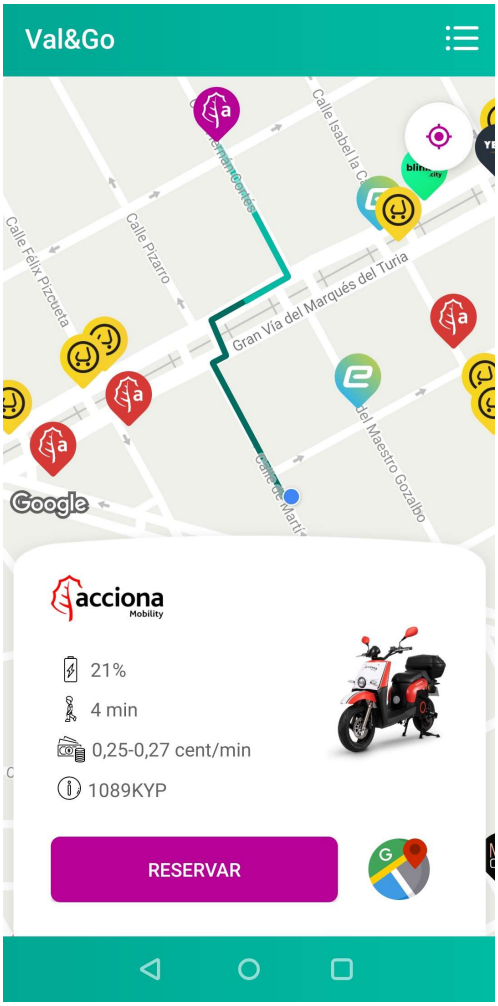


*Ilustración 35: Animación de ruta y cuadro de información*

Se observa que los datos del vehículo van acompañados de iconos que facilitan e identifican inmediatamente de que se trata. Por ejemplo, la carga de la batería va acompañada del icono de una pila, y el tiempo andando hasta el vehículo va acompañado de una persona caminando.

El cuadro de información se puede descartar en cualquier momento desplazándolo hacia abajo o pulsando el botón de atrás. Una vez pasa esto el marcador del mapa queda en estado normal y la animación de ruta desaparece.

Si queremos saber que vehículos tenemos más cerca podemos iniciar la actividad de lista de vehículos pulsado sobre el icono de la barra de herramientas (parte superior de la pantalla). Una vez iniciada la actividad aparecerá una lista reciclable, con diferentes elementos en forma de carta representando los vehículos más cercanos y con algo de información.

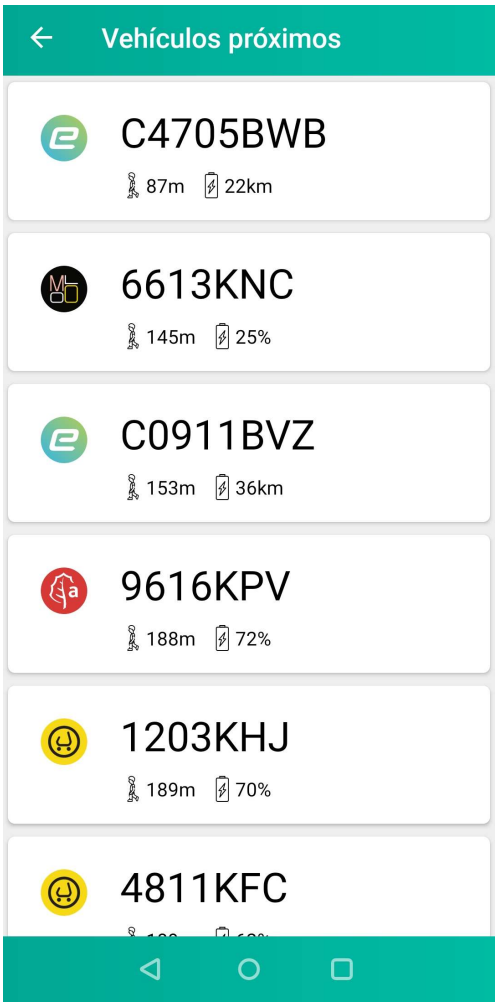


*Ilustración 36: Lista de vehículos cercanos*

En esta lista cada elemento muestra el logotipo de la plataforma con una forma redonda, la matrícula o nombre del vehículo, la distancia, y la carga o autonomía. En la imagen anterior podemos observar como el primer elemento muestra la carga en kilómetros y no en porcentaje. Esto es debido a esa es la información que brinda la API de *Ecooltra,*

Una vez hayamos localizado el vehículo que nos interese podemos pulsar uno de los elementos. Esto nos devolverá a la actividad del mapa y enfocará el marcador correspondiente a dicho vehículo, mostrándonos más información y dándonos la opción de reservar.

# 7. Conclusiones

Para finalizar este proyecto se van a exponer las conclusiones, analizar el trabajo realizado y los resultados obtenidos.

La aplicación se ha desarrollado satisfactoriamente, se han conseguido los objetivos propuestos y se ha obtenido un producto completo y acabado. La estética de la aplicación es la deseada, se ha conseguido plasmar la sencillez propuesta, la coherencia de elementos y colores y la fácil usabilidad.

También se ha conseguido escribir un código ordenado y coherente, objetivo primordial de este proyecto. Se han utilizado patrones de diseño propios de la ingeniería de software y como resultado se ha obtenido un código más ordenado y más legible para el desarrollador y otros programadores que pudieran verlo y modificarlo.

Algo también importante de este proyecto es que se ha desarrollado sin utilizar ningún *framework*. Esta escrito y realizado en Java, es decir, Android nativo. Eso sí, se han utilizado diferentes librerías, ya sean de Android, Java, y otros desarrolladores que también han contribuido a la realización de esta aplicación. El hecho de que no haya intervenido ningún *framework* significa que la aplicación es más ligera y más estable, ya que se reduce el número de dependencias y de llamadas internas.

La labor de escritura de código ha sido una de las partes más importantes y que más tiempo han consumido. Aun así, cabe mencionar que los procesos de ingeniería inversa han sido igual de importantes y al ser una tarea tediosa (pero entretenida) también han consumido un tiempo considerable. El diseño gráfico de elementos, a pesar de ser una faceta que dista de la ingeniería de software o el diseño y la escritura de código, forma una parte indispensable del desarrollo de una aplicación y de la Ingeniería Informática, por lo que no debe dejarse de lado y hay que darle la importancia que tiene.

En resumen, he aprendido múltiples facetas del desarrollo de software, y he perfeccionado conocimientos que ya tenía y que me había proporcionado a la carrera. Por esto el desarrollo de este trabajo ha sido una tarea fructífera y de la que he sacado muchas cosas positivas.

## 7.1 Relación con los estudios cursados

Este proyecto se ha realizado juntando diferentes conocimientos de las distintas asignaturas cursadas en la carrera. Es evidente que las materias básicas del primer ciclo de la carrera han jugado un papel primordial, puesto que sin esos conocimientos básicos no se llega a adquirir conocimientos más avanzados. Cabe destacar asignaturas como **Introducción a la Informática y a la Programación, Programación,** y **Matemática**

**Discreta.** Asignaturas no tan básicas como **Gestión de Proyectos** y **Ingeniería de Software** también han jugado un papel importante en conceptos más avanzados, como la estructuración y tiempos de un proyecto o la realización de un diagrama UML. Asignaturas como **Redes de Computadores** han ayudado a comprender como funciona internet y sus distintos tipos de comunicaciones, como por ejemplo el funcionamiento de un servicio web y las diferentes capas de las que se compone.

Las asignaturas que más repercusión han tenido en este proyecto han sido asignaturas cursadas más recientemente. Estas son asignaturas como **Integración e Interoperabilidad,** que ha jugado un papel importante en los conocimientos de integración de servicios. Este es uno de los principios básicos de la aplicación realizada, puesto que integra múltiples servicios en una misma aplicación. Los conocimientos de interoperabilidad y las herramientas utilizadas en esta asignatura han sido indispensables. Otra asignatura que ha tenido impacto ha sido **Mantenimiento y Evolución de Software**, de la que se han extraído ideas como la ingeniería inversa. La asignatura **Diseño de Software** también ha contribuido al proyecto con temas como los patrones de diseño.

# 8. Trabajos futuros

En este apartado se explican los trabajos que se van a realizar a partir de ahora para mejorar el producto realizado. En primer lugar se añadirán más funcionalidades a la aplicación, que ya habían sido pensadas y diseñadas pero que no se han podido implementar por falta de tiempo. Algunas de ellas son añadir una ventana de ajustes para algunos ajustes del mapa y de las animaciones, o añadir soporte para relojes inteligentes, lo que permitiría localizar un vehículo con uno de estos aparatos.

Se va a intentar añadir más funcionalidades de integración con los servicios, como por ejemplo la posibilidad de realizar la reserva a través de la propia aplicación, aunque esto depende en mayor medida de la disposición de las empresas a colaborar con el proyecto.

Un objetivo primordial es la publicación y distribución de la aplicación en la tienda **Google Play.** Esta es una tarea pendiente que se realizará cuando se añadan más funcionalidades. También es necesario resolver problemas de derechos por los logotipos que aparecen en le aplicación, por ejemplo.

En un futuro también se pretende diseñar el mismo modelo de aplicación para otras ciudades o países. Ya se dispone de servicios que también operan en ciudades como Madrid o Barcelona, por lo que sería todavía más fácil en ciudades como estas.

# 9. Bibliografía


- *Xataka*: https://www.xatakamovil.com/sistemas-operativos/asi-como-android-se-ha-comido-mercado-diez-anos
- *Abalnex:* https://www.abanlex.com/2013/12/la-ingenieria-inversa-es-legal-en-espana/
- *GitHub:* https://ibotpeaches.github.io/Apktool/

  https://github.com/amalChandran/trail-android

  https://github.com/pxb1988/dex2jar
- *BBVAopen4u*: https://bbvaopen4u.com/es/actualidad/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos
- *Wikipedia*: https://es.wikipedia.org/wiki/Extensible_Markup_Language

  https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)

  https://es.wikipedia.org/wiki/Android

  https://es.wikipedia.org/wiki/Especificaci%C3%B3n_de_requisitos_de_software
- *Androidsis:* https://www.androidsis.com/el-ciclo-de-vida-de-una-aplicacion-de-android/
- *BOE:* https://www.boe.es/buscar/act.php?id=BOE-A-1996-8930
- *NinjaMock:* https://ninjamock.com/
- *AndroidAssetStudio:* https://romannurik.github.io/AndroidAssetStudio/
- *Canva:* https://www.canva.com/
- *Android API:* https://developer.android.com/reference