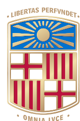


Trabajo final de grado

GRADO EN INGENIERÍA INFORMÁTICA

Facultad de Matemáticas
Universidad de Barcelona

CLEAN ARCHITECTURE Y RXJAVA EN ANDROID

Autor: Marc González Díez

Director: Àlex Pardo Fernández
Realitzat a: Departament de Matemàtica Aplicada i Anàlisi. (UB)

Barcelona, 30 de junio de 2016

Abstract

Mobile development is relatively new. Often companies have to pay for mistakes made in the past resulting in unstable applications, which are difficult to maintain and with a huge investment of time for bug fixes and refactoring. This document will review and apply the best technologies available in the Android Development as well as the architecture proposed by Robert C Martin, called Clean Architecture, and the best Software Design patterns, leading to an application that meets the SOLID principles, with scalable architecture, maintainable and efficient for the developer and the company, trying with this to help a potential reader and contribute with my study in application development paradigm.

Resum

El desenvolupament Mobile és una cosa relativament nova. Sovint les empreses han de pagar errors comesos en el passat donant lloc a aplicacions inestables, difícils de mantenir i amb una gran inversió de temps en corregir els errors i refactoring. En aquest treball s'estudiaràn i aplicaràn les millors tecnologíes disponibles al Desenvolupament Android així com l'Arquitectura proposta per Robert C Martin anomenada Clean Architecture i els millors patrons de Disseny de Software, donant lloc a una aplicació que compleixi els principis SOLID, sigui escalable, mantenible i eficient de cara al desenvolupador i per l'empresa, intentant així, ajudar a un posible lector i aportar el meu estudi al paradigma del desenvolupament d'aplicacions mòbils.

Resumen

El desarrollo Mobile es algo relativamente nuevo. A menudo las empresas tienen que pagar errores cometidos en el pasado dando lugar a aplicaciones inestables, difíciles de mantener y con una gran inversión de tiempo en el arreglo de errores y refactoring. En este trabajo se estudiarán y aplicarán las mejores tecnologías disponibles en el Desarrollo Android así como la Arquitectura propuesta por Robert C Martin llamada Clean Architecture y los mejores patrones de Diseño de Software, dando lugar a una aplicación que cumpla los principios SOLID, sea escalable, mantenible y eficiente de cara al desarrollador y a la empresa, intentando así ayudar a un posible lector, y aportar mi estudio al paradigma del desarollo de aplicaciones.

Agradecimientos

A mis padres, por sacrificar el presente que consiguieron, para darme un futuro que no tuvieron.
A Àlex, por estar siempre dispuesto a ayudarme, por las buenas ideas aportadas a este trabajo y por su apoyo como tutor.
A Carles, por enseñarme todo lo que sé sobre UI en Android.
A Sergi, por transmitirme la pasión por el Clean Code.
A Cristina, por su apoyo incondicional.

"Live as if you were to die tomorrow. Learn as if you were to live forever."

— Mahatma Gandhi

Índice

1. Introducción **1**

1.1. Contexto 1

1.2. Motivación 2

1.3. Objetivos 3

1.4. State of the Art 3

1.5. TweetStatt 4

1.5.1. Contexto y motivación para la aplicación 4

1.6. Tecnologías utilizadas 5

2. Conceptos básicos **6**

2.1. Activities 6

2.2. Fragments 7

2.3. Principios SOLID 9

3. Planificación y costes **9**

3.1. Planificación inicial 9

3.2. Planificación real 10

3.3. Costes de TwettStat 11

4. Clean Architecture **12**

4.1. Definición 12

4.1.1. Ventajas 15

4.1.2. Desventajas 15

4.2. Aplicación en Android 16

4.2.1. Módulo presentation 16

4.2.2. Módulo domain 16

4.2.3. Módulo data 16

5. Desarrollo **17**

5.1. Módulo presentation 17

5.1.1. Patrón Model View Presenter 17

5.1.2. Model View ViewModel 19

5.1.3. ButterKnife 20

5.1.4. Picasso 22
5.1.5. Dagger2 25
5.1.6. Aplicación del módulo presentation en TweetStatt 31
5.1.7. Estructura del Módulo Presentation 37
5.2. Módulo dominio 39
5.2.1. RxJava 39
5.2.2. Use Cases 46
5.2.3. Patrón Repository 47
5.2.4. Objetos Business Object 48
5.2.5. Aplicación del módulo dominio en TweetStatt 48
5.2.6. Estructura del Módulo Domain 52
5.3. Módulo data 53
5.3.1. Objetos Data Transfer Object 53
5.3.2. Objetos Value Object 54
5.3.3. Patrón Event Bus 54
5.3.4. Patrón DataStore 56
5.3.5. Retrofit 57
5.3.6. Realm.io 61
5.3.7. Fabric SDK 66
5.3.8. Aplicación del módulo data en TweetStatt 67
5.3.9. Estructura del Módulo Data 76

6. Conclusiones y trabajo futuro **77**
6.1. Conclusiones 77
6.2. Ampliaciones 78

7. Anexo **78**
7.1. Instalación y ejecución de la entrega 78
7.2. Términos y conceptos 79

1. Introducción

1.1. Contexto

El Sistema Operativo Android [1], nace de la mano del Ingeniero Informático Andy Rubin en 2003, bajo la empresa Android Inc. Éste ex-empleado de Apple vende Android Inc. a Google en el año 2005, aunque éste Sistema Operativo no vió la luz en el mercado hasta el 22 de Octubre del 2008, con el lanzamiento del primer dispositivo que incorporaba Android como Sistema Operativo, por parte de la marca HTC.
En la actualidad, Android cuenta con la mayor parte del mercado en España, como se muestra en la figura 1.

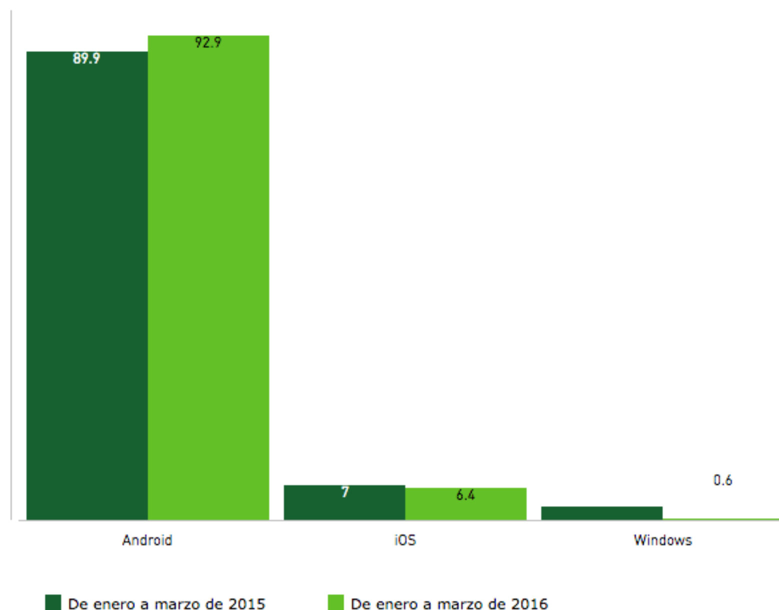


Figura 1: Cuota de mercado de dispositivos móviles en España
Fuente: elandroidlibre.com

Desde su salida al mercado hasta la actualidad, han pasado ya varios años. Empezó compitiendo con el Sistema Operativo Symbian, y no fue hasta 2010-2012 que superó a éste en cuota de mercado, tal y como se muestra en la figura 2. En ese momento las empresas empiezan a ver negocio en esta plataforma y, gracias a ello, Android empieza a crecer a un ritmo muy alto superando a Symbian, iOS y cualquier otra plataforma orientada a dispositivos móviles.

Todos los crecimientos rápidos tienen sus ventajas y desventajas; cuando una plataforma de Software crece rápido, y las empresas quieren negocio, el tiempo juega en contra. Es por este motivo, cuando en el año 2010-2012 muchos Ingenieros Informáticos y Desarrolladores que hasta la fecha estaban desarrollando para otras plataformas como Servidor, Aplicaciones Web y un largo etcétera vieron en la plataforma Android un cambio con prospección de futuro en su carrera profesional.

Había, y actualmente hay, una gran demanda de desarrolladores Android, pero poca oferta y al ser una plataforma nueva, todos los desarrolladores, independientemente del tiempo que llevaran, no tenían experiencia en el concepto Mobile. Ésto sumado a que por aquel entonces, la comunidad era pequeña y el soporte que daba la plataforma a los desarrolladores era muy limitado comparado con el actual, hizo que apenas se tuvieran en cuenta y no se invirtiera tiempo en Arquitecturas, Patrones de Diseño y mejores prácticas obteniendo, años después, productos (aplicaciones) que habían escalado y para nada eran mantenibles dando lugar, al mismo tiempo, a aplicaciones lentas, inestables y poco fiables.

Actualmente, los dispositivos móviles son cada vez más potentes, permitiendo a los desarrolladores crear y distribuir aplicaciones con más carga de trabajo, con más funcionalidades y, por lo tanto, hace falta invertir tiempo en buscar y aplicar la mejor arquitectura posible en Android.

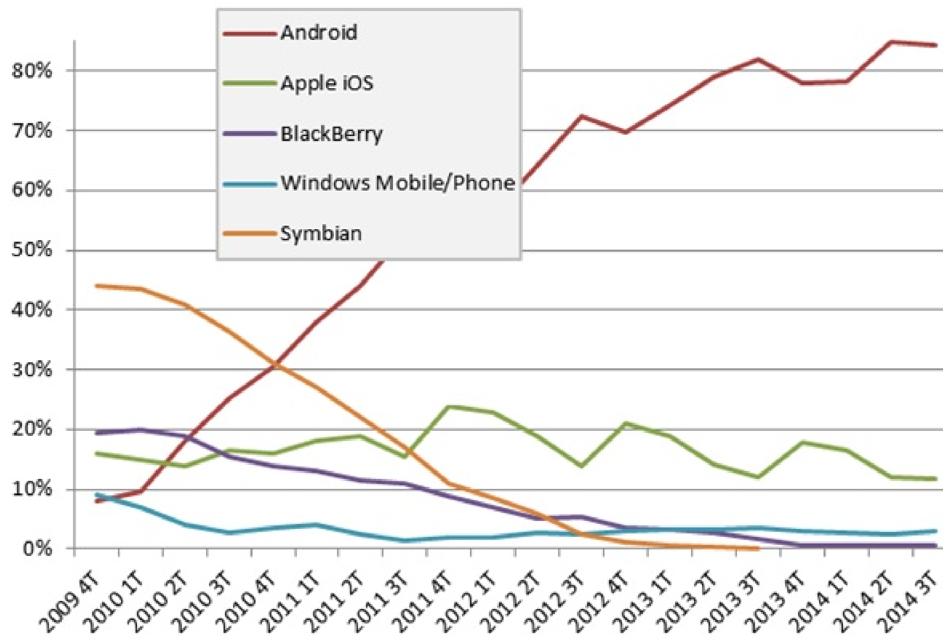


Figura 2: Histórico de la cuota de mercado de las plataformas móviles
Fuente: Pontificia Universidad Católica del Perú (PUCP)

1.2. Motivación

Dado el contexto anterior, la motivación de este trabajo es clara; se necesita encontrar arquitecturas que permitan obtener proyectos escalables, de fácil mantenimiento, con una tasa ínfima de errores, junto a una buena estabilidad y rendimiento.

Personalmente, desde mi incorporación al mercado laboral como Desarrollador Android, he podido comprobar en primera persona los problemas mencionados anteriormente y cómo eso afecta actualmente a empresas y trabajadores, llevando a éstas a invertir tiempo en formación para que un nuevo desarrollador se pueda unir

a un proyecto, o gastando grandes cantidades de tiempo refactorizando código y consiguiendo que sea muy tedioso para el desarrollador trabajar en ello.

1.3. Objetivos

Este trabajo tiene como objetivo investigar, entender y aplicar la mejor arquitectura posible para una aplicación Android, así como los mejores frameworks disponibles y los patrones de diseño más adecuados para las necesidades de un dispositivo móvil. Teniendo siempre en mente los principios SOLID, se trabajará para conseguir una aplicación que cumpla con la motivación del proyecto.

Para ello, a lo largo de la realización del trabajo y de la memoria, hay dos partes diferenciadas, la parte de investigación y la parte del desarrollo de la aplicación siendo, la última, un ejemplo-guía para los futuros lectores y un método de apoyo para reforzar lo aprendido.

1.4. State of the Art

En los proyectos de aplicaciones móviles, en el mejor de los casos, se realizaba una separación a nivel de código en dos capas, de manera que se diferenciaba la capa propia de nuestra aplicación que contendría nuestro modelo de datos, nuestra funcionalidad y capa de negocio y, por otro lado, la capa que se encargaba de la comunicación con el servidor, representado en la figura 3. Para llevar a la realidad esta arquitecutra, se separaban las capas de datos que nuestra base de datos y aplicación iba a tratar, con la capa de datos que nuestro servidor iba a recibir y enviar, de manera que la única diferencia entre éstas era el tipo de dato que trataban: *Data Transfer Object* para los objetos que iban a viajar al Servidor y *Value Object*, para los Objetos que nuestra aplicación iba a guardar.

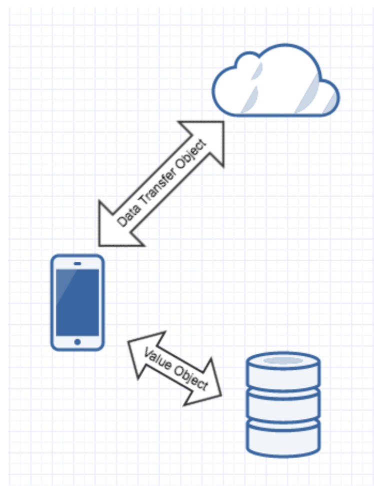


Figura 3: Diagrama a alto nivel con utilización de capa de datos propia y externa

En el peor de los casos, esta separación no existía y el modelo de datos que recibíamos del servidor se aplicaban a lo largo de nuestra aplicación, obteniendo un

gran acoplamiento a éste, y dependiente completamente a él y a sus cambios, como podemos observar en la figura 4.

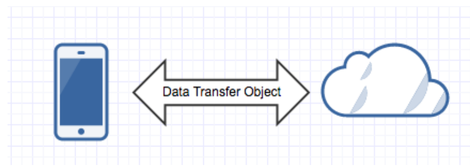


Figura 4: Diagrama a alto nivel con utilización de una sóla capa de datos

Ambas ideas y aplicaciones son malas a simple vista y cuanto más las estudias peor te parecen. Pongamos de ejemplo el caso de separación en dos capas. Tenemos dos capas de datos, por lo que desacoplamos en cierta parte nuestro modelo de datos del de Servidor. Pero tenemos varios problemas añadidos, las aplicaciones suelen hacer operaciones con los datos, se utilizan para mostrarse al usuario, se replican en base de datos, etc. Con una sola separación de ésta, los datos que viajaran a lo largo de nuestra aplicación serán los mismos. Es decir, podemos tener un objeto que represente a un usuario con 20 campos, con información irrelevante a la hora de mostrar al usuario su perfil. Por lo tanto, en tiempo de ejecución tendremos objetos mucho más pesados de lo necesario.

1.5. TweetStatt

Para la aplicación de los conceptos investigados y aprendidos, se realizará el desarrollo de una aplicación bautizada como TweetStatt. Esta aplicación seguirá punto por punto lo explicado en esta memoria, de manera que podría servir como guía a un futuro lector, y me resultará de gran ayuda para asimilar todo lo aprendido.

1.5.1. Contexto y motivación para la aplicación

TweetStatt es una aplicación que trabajará con la API de Twitter, recibiendo datos sobre tus propios tweets, el de tu *timeline*, se podrán realizar búsquedas a nivel global y tendrá un listado de *hashtags* a tiempo real sobre el que interactuar.

Recibe este nombre ya que su funcionalidad será la de realizar pequeños gráficos sobre los datos obtenidos de Twitter, en base a Ciudad, Localidad o Fecha de creación del *Tweet*.

Todos estos datos serán replicados en una base de datos local, para que la aplicación siga completamente funcional aún estando sin conexión, resultando unas estadísticas mucho mejores conforme más se utiliza la aplicación ya que el número de muestras disponibles aumentará.

Las estadísticas generadas se guardarán también en la base de datos, pudiendo acceder a ellas en cualquier momento y visualizarlas otra vez.

La funcionalidad de esta aplicación viene dada por varios motivos: en primer lugar, para aplicar todo lo aprendido necesitaba desarrollar una aplicación que tuviera conexión con un servidor REST, que pudiera replicar los datos en local, para

así tener una Base de Datos, que el usuario pudiera interactuar con ella de varias maneras consiguiendo así un buen número de casos de uso y que tuviera que realizar forzosamente una gran cantidad de trabajo en paralelo ya que el volumen de datos a tratar fuera relativamente grande (en una aplicación móvil). En segundo lugar, no quería desarrollar una copia de otra aplicación; cuando realicé el estudio de mercado de ésta, ví que apenas habían aplicaciones con esa funcionalidad, más allá de algunas que te contabilizaban el número de seguidores que tenías y sólo una que realizaba lo que he mencionado anteriormente.

1.6. Tecnologías utilizadas

Para la realización de esta aplicación he utilizado las siguientes tecnologías y frameworks, que se explicarán a lo largo de la memoria.

- Java 8
- Android SDK
- ButterKnife - http://jakewharton.github.io/butterknife/
- Retrofit - http://square.github.io/retrofit/
- Dagger2 - http://google.github.io/dagger/
- Realm.io - https://realm.io/
- Picasso - http://square.github.io/picasso/
- ReactiveX Java - http://reactivex.io/
- Fabric SDK - https://get.fabric.io/
- Gson - https://github.com/google/gson

2. Conceptos básicos

2.1. Activities

Una *Activity* [2] es un componente de aplicación que proporciona una pantalla que permite al usuario interactuar con el fin de hacer algo, por lo tanto, son la base de toda aplicación.
Una aplicación, por lo general, se compone de varias *Activities* que están ligadas entre ellas, para poder establecer una navegación. Por lo general, una *Activity* en una aplicación se especifica como la *Activity "Launcher"*, que se presenta al usuario al iniciar la aplicación por primera vez. Cada vez que se inicia una nueva *Activity*, se detiene la anterior, pero el Sistema Operativo conserva la *Activity* en una pila, y se añade ésta nueva en el top. El *Stack* de Android funciona como un LIFO "Último en entrar, primero en salir", por lo que, por ejemplo, cuando el usuario realiza la acción *'Back'*, la *Activity* en la que estaba se extrae de la pila, se destruye y se reanuda la anterior que será la primera en el *Stack* en ese momento.

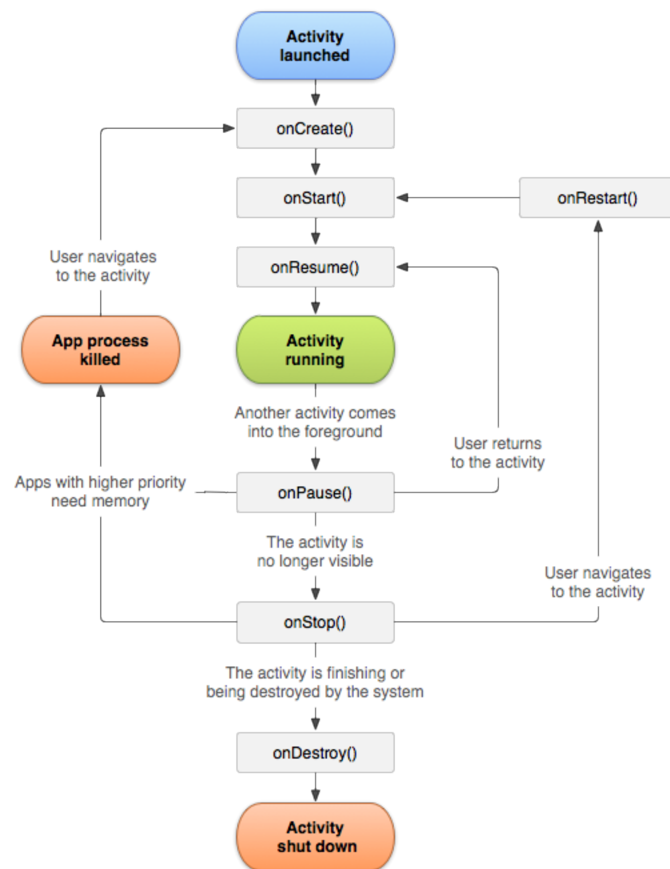


Figura 5: Ciclo de vida de una Activity
Fuente: developer.android.com

Cuando una *Activity* se detiene, se comunica el cambio en el estado del ciclo de vida a través de los *Callback* de la Activity. Hay varios *Callback* que podría

recibir una *Activity*, representados en la figura 5, debido a cambios de estado en ella realizados por el sistema; por ejemplo, la creación, la detención, la reanudación, la destrucción, en cada estado el Sistema realizará sus acciones y si nosotros aprovechamos ese *Callback*, cuando acabe podemos ejecutar las acciones que nosotros queramos. El ejemplo más básico para estos casos, es que cuando una *Activity* se detiene, ya sea porque el usuario bloquea el móvil o cambia de Aplicación, los datos que ésta tuviera se liberan, produciendo una pérdida de éstos y el desarrollador deberá recuperarlos en el estado de *onResume*, es decir, cuando la *Activity* vuelve a la vida. Para este caso, contamos en el *onCreate* con un objeto tipo *Bundle* para poder añadir lo que necesitemos y tenerlo disponible en el *onResume*.

2.2. Fragments

Un *Fragment* [2] representa un comportamiento o una porción de interfaz de usuario en una actividad, como se observa en la figura 6. Se pueden combinar múltiples fragments en una sola *Activity* para formar una interfaz completa de usuario multi-panel y la reutilización de un *Fragment* en múltiples *Activities*. Un *Fragment* puede ser una parte de la interfaz, o ésta completamente, y aunque un *Fragment* tiene su propio ciclo de vida y recibe sus propios eventos de entrada, se puede modificar, añadir y ocultar mientras la *Activity* a la que esté/estén esté en marcha.

Son utilizados popularmente para delegar trabajo de la *Activity* en porciones de pantalla, por su capacidad de reutilización (e.g. pongamos una pantalla principal que contiene una lista de items, y a su vez, en otra pantalla de la aplicación podemos ver los items marcados como favoritos; lo mejor en ese caso sería tener un *Fragment* que se encargara de mostrar esos items para poder reutilizarlos en ambas pantallas.) Por ello, la mejor práctica en desarrollo Android, es tener el mayor número de *Fragments* posibles y que las *Activities* deleguen la mayor carga a éstos.

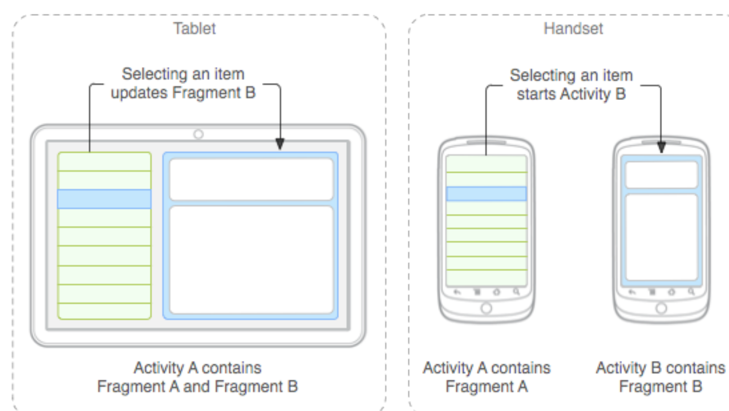


Figura 6: Ejemplo de utilización de los fragment
Fuente: developer.android.com

Un *Fragment* siempre está asociado a una *Activity*, cuando se detiene la *Activity*, se detiene el *Fragment* y, cuando se destruye la *Activity*, el *Fragment* también lo

hace. Los *Fragment*, también tienen su propio *Stack* asociado a la *Activity*, por lo tanto, si tuviéramos una *Activity* con dos *Fragments* superpuestos, si el usuario hiciera *'Back'*, quedaría activo el primer *Fragment*. Estos *Fragment* también tienen sus propios *Callbacks* de eventos, representados en la figura 7.

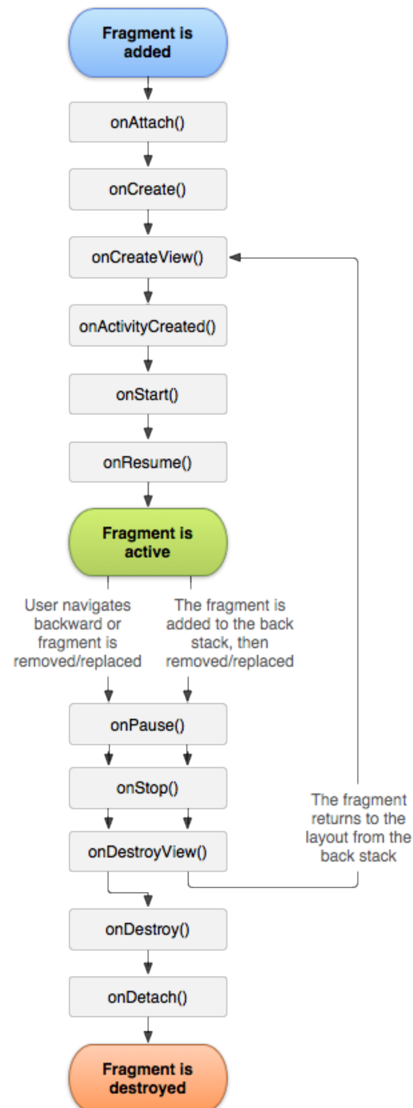


Figura 7: Ciclo de vida de un Fragment
Fuente: developer.android.com

Después de esta breve explicación, podemos entender lo importante que resulta para un desarrollador Android el control absoluto y conocimiento sobre estos estados.

2.3. Principios SOLID

En Ingeniería del Software se conoce como principios S.O.L.I.D [3], el acrónimo introducido por Robert C. Martin sobre el año 2000, representa cinco principios para la programación orientada a objetos y el diseño de ésta. La aplicación de estos principios nos ayudan a crear un software de calidad y son introducidos como conceptos básicos en esta memoria los cuales se referenciarán constantemente a lo largo de ésta, ya que durante toda la realización he tenido muy presente estos principios así como los patrones GRASP que consideraba necesarios para este proyecto.

- Principio de responsabilidad única (*Single responsibility principle*): Un objeto sólo debería tener una responsabilidad, teniendo así una estructura mucho más clara y evitando posibles problemas de mantenimiento en un futuro.

- Principio de abierto/cerrado (*Open/closed principle*): "Las entidades en Sofware, deben estar abiertas a extensión y cerradas a modificación", principio relacionado directamente con la escalabilidad del proyecto.

- Principio de sustitución de Liskov (*Liskov substitution principle*): Los objetos de un Software, deberán poder ser substituidos por objetos que extiendan de éste, sin alterar el funcionamiento del programa.

- Principio de segregación de la interfaz (*Interface segregation principle*): Muchas interficies de propósito específico son mejores que una de propósito general.

- Principio de inversión de la dependencia (*Dependency inversion principle*): Tu Software deberá depender de abstracciones, no de las implementaciones.

3. Planificación y costes

3.1. Planificación inicial

En la planificación inicial se destinaron cuatro semanas para definición del proyecto, de la aplicación, decisión de arquitectura y *frameworks* a utilizar y planificación del trabajo. Aproximadamente unas ocho semanas para el desarrollo e implementación de la aplicación a medida que se investigaba y aplicaban los frameworks y arquitectura escogidas y, finalmente, el último mes para trabajar en la memoria. Se elaboraron una serie de *user stories* siguiendo la metodología *SCRUM* definiendo *Sprints* de dos semanas en los que se planificaba implementación de las *user stories* (*as a user I want...*) escogidas y las *user stories* de investigación (*as a developer I want...*) tal y como se puede ver en la figura 8. Como detalle, la *user story* llamada *"As a developer I want to implement Clean Architecture in all application"* no tiene tiempo establecido ya que iba a ser una constante en todo el trabajo y se predecía que iban a haber refactors y cambios constantes durante la vida de éste, a medida que el estudio sobre ésta iba avanzando.

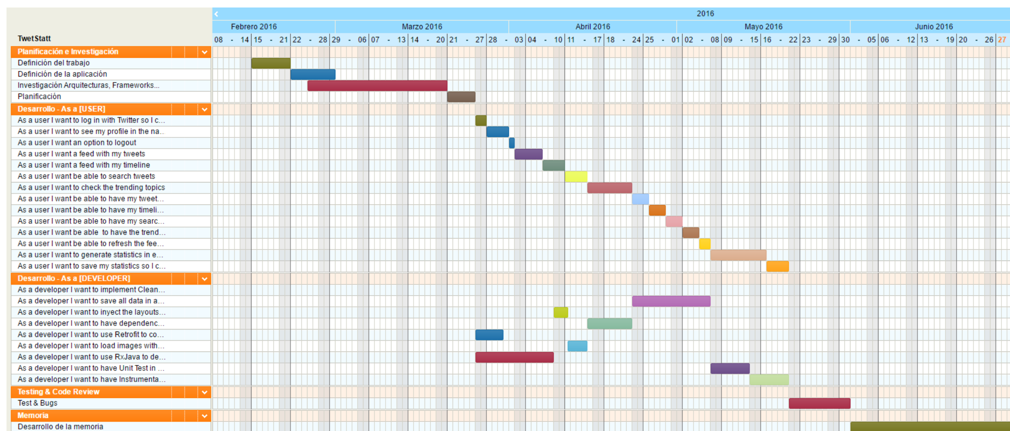


Figura 8: Gráfico sobre el reparto de tiempo en el desarrollo de TwettStat

3.2. Planificación real

Se recortó el tiempo en investigación ya que pude definir de manera más rápida a la prevista la arquitectura y *frameworks* que iba a utilizar, de manera que el desarrollo empezó algo antes de lo esperado. Cuando empezó el desarrollo, los problemas de tiempo y previsión se vieron en el primer *Sprint* al empezar la comunicación con el Servidor (problemas que serán comentados durante la memoria) cosa que implicó un mayor tiempo en investigación y aplicación de *RxJava*. Posteriormente, otro de los grandes desajustes de tiempo vino provocado por la aplicación de *Dagger2*, *framework* con una gran curva de aprendizaje que en ese momento se descartó tras emplear tiempo y no obtener resutlados, para evitar dejar el trabajo bloqueado y perder demasiado tiempo. Más adelante, tras asistir al *Workshop* que ofrecía el *Google Developers Group* en Barcelona y tiempo de investigación extra sobre esta herramienta se pudo planificar de nuevo y aplicarlo, aunque más tarde como se puede ver en la figura y sacrificando la parte de testing Unitario y de Instrumentación previstos para el final del desarrollo, aunque también se añadió la implementación de un Bus de Eventos *Reactive*. Finalmente, la planificación real quedó como se puede observar en la figura 9. Los *refactors* necesarios y normales que suceden al desarrollar con metodlogía *SCRUM* están incluidos en los tiempos que se visualizan para las User Stories.

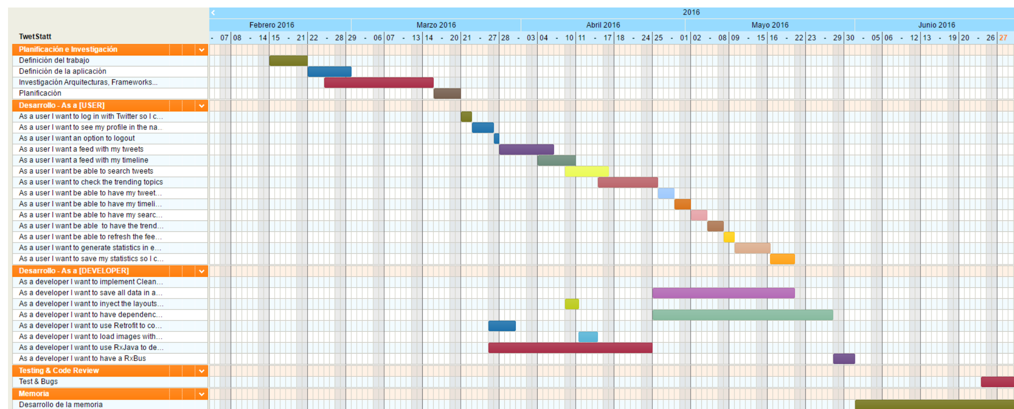


Figura 9: Gráfico sobre el reparto de tiempo en el desarrollo de TwettStat

3.3. Costes de TwettStat

Para el cálculo del coste que supondría llevar a la realidad en un ámbito de desarrollo profesional la aplicación mostrada en este trabajo, se ha hecho una abstracción del tiempo utilizado y posteriormente un reparto de éste de alto nivel. Se ha tenido en cuenta sólo el tiempo empleado en desarrollo. Se estima una dedicación total de 224 horas para el desarrollo empleadas como se puede observar en la figura 10 en la que se realiza una separación en cuanto a tareas a alto nivel de:

- Guardado en Base de Datos.
- Generación de Estadísticas.
- Crear el servicio para la API de manera *ReactiveX*.
- Testing de la aplicación.
- Aplicación de *Clean Architecture* y todos los *Frameworks* utilizados.
- Diseño e implementación de la Interfaz de Usuario.

Una vez calculado el total de horas invertidas y el reparto de ellas, suponiendo un desarrollador el cual cobre la hora de trabajo a 15 euros la hora supondría el coste total representado en la figura 11.

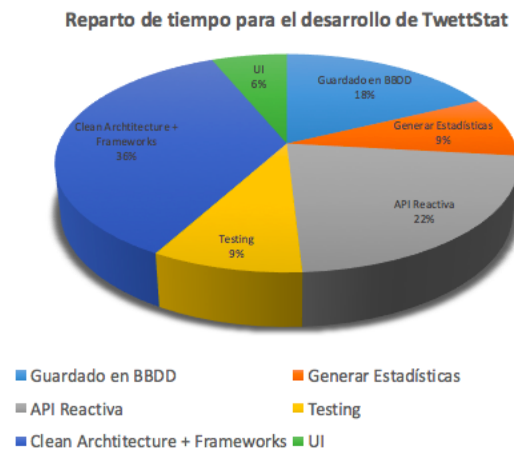


Figura 10: Gráfico sobre el reparto de tiempo en el desarrollo de TwettStat

TwettStat

Total

3.360,00 €

Elemento	Coste
Guardado en BBDD (40h)	600,00 €
Generación de Estadísticas (20h)	300,00 €
Comunicación con API Reactive (50h)	750,00 €
Testing Manual (20h)	300,00 €
Clean Architecture + Frameworks (80h)	1.200,00 €
Diseño de Interfaz (14h)	210,00 €

Figura 11: Costes finales en el desarrollo de TwettStat

4. Clean Architecture

4.1. Definición

La idea de *Clean Architecture* [6] nace en la mente de Uncle Bob, pseudónimo para Robert C. Martin, ya mencionado en esta memoria. En sus años de experiencia como Ingeniero de Software y Consultor, observó que todos los proyectos en los que trabajaba cometían errores de arquitectura, y todos repetían unos patrones problemáticos, de manera que con el paso del tiempo se volvía tedioso lidiar con éstos.

Clean Architecture nace como una vuelta de tuerca a la Arquitectura Hexagonal, Arquitectura *"Cebolla"* entre otras, con unos objetivos muy claros y aquí listados:

- Independiente de los *Frameworks*: La arquitectura no debe depender de ningún

Framework. Si se consigue esto, podemos ser nosotros quienes utilicemos al *Framework*, y no viceversa. Nuestra arquitectura debe ser nuestra, no de terceros.

- Testeable: Las capas de negocio deben poder ser testeadas sin tener que lidiar con la *UI*, la base de datos, el Servidor o cualquier otra parte presente en nuestro Software.
- Independiente de la *UI*: La interfaz de usuario puede cambiar constantemente, sin afectar al resto de Software.
- Independiente de la Base de Datos: Se debe poder cambiar, por ejemplo, entre *Oracle SQL Server*, a *Mongo* sin que nuestra capa de negocio se vea afectada.
- Independiente de cualquier agente externo: Tus reglas de negocio no deben saber nada del "mundo exterior"que las rodea.

Bajo estas premisas, Uncle Bob propone el diagrama que podemos observar en la figura 12.

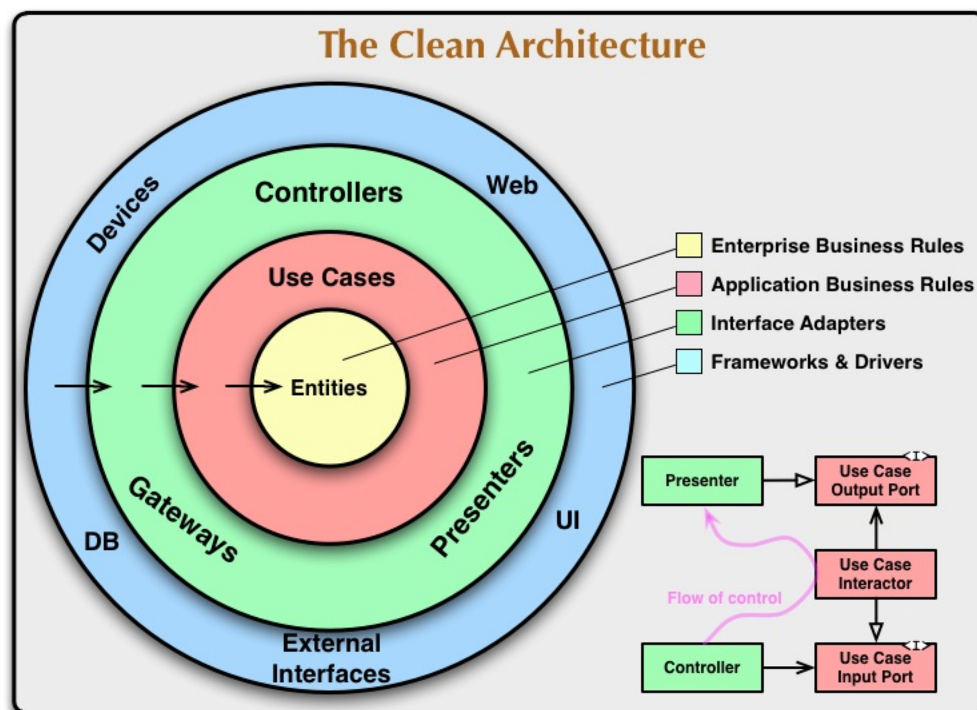


Figura 12: Abstracción de Clean Architecture
Fuente: cleancoder.com

Regla de Dependencias

En el diagrama presentado, aparecen unas flechas que marcan la regla de dependencias de nuestra arquitectura. Estas flechas unidas a las diferentes capas que

podemos observar marcadas por cada color, definen una capa más en nuestra Arquitectura. Cuanto mas externa sea la capa, más de alto nivel será nuestro Software, es decir, las capas externas son mecanismos y las internas son políticas.

Esta regla de dependencias nos marca que el Software, nuestro código fuente, sólo puede acceder a su siguiente capa interior, conociendo sólo a ésta y a ninguna otra. Las entidades no conocerán a los Casos de Uso (también llamados *Interactors* en *Clean Architecture*) y éstos, a su vez, no conocerán a la capa que haga uso de ellos.

La capa más externa, como se puede observar, deberá ser la que esté mas expuesta a cambios, ya sea la Interfaz de Usuario, nuestra Base de Datos... de manera que creamos un Software robusto y cerrado a cambios externos.

Entidades

Las entidades presentes en el diagrama, son las encargadas de ser las representantes de nuestras reglas de negocio. Nuestras entidades pueden ser objetos con sus respectivos métodos o una serie de funciones o estructuras de datos: serán nuestros Objetos de Negocio, aquellos que encapsulan las capas de más alto nivel y deben ser los menos expuestos a cambios externos. Que nuestra *UI* cambie no debería afectar a nuestras Entidades, por ejemplo.

Casos de Uso

El Software en esta capa contiene reglas de negocio específicas. Encapsulará e implementará todos los casos de uso del sistema. Estos Casos de Uso son los que orquestran el flujo de datos de nuestra aplicación, accedendiendo a las Entidades y proporcionando datos a capas superiores.

Esta capa puede estar expuesta a cambios, ya que la funcionalidad de nuestra aplicación puede cambiar, ya sea añadiendo, modificando o eliminando algún caso de ésta, pero nunca deberá afectar a nuestra capa de Entidades.

Esta capa de Casos de Uso, no deberá ser afectada por cambios externos tampoco, ya sean cambios en la *UI*, en *Frameworks* que utilicemos, o por nuestra Base de Datos.

Adaptadores de Interfície

En esta capa, tendremos un conjunto de adaptadores que convertirán los datos recibidos gracia a los Casos de Uso, al formato de datos más correcto o necesario para ser llevado a la capa superficial ya sea a nuestra base de datos, a la Interfaz, a algún *Framework* utilizado... En esta capa se presenta el objeto de tipo Modelo y Uncle Bob propone una arquitectura MVC para esta capa.

Propone MVC ya que, recibirá peticiones de agentes externos, nuestra Vista en el famoso MVC y nuestro Controlador pedirá estos datos a los Casos de Uso y,

finalmente, retornará objetos de tipo Modelo.

Por lo tanto, la misión de esta capa es desacoplar nuestro modelo de datos interno del externo.

Frameworks y Interficies Externas

La capa más externa, como ya he mencionado, debe estar compuesta por los *Frameworks* utilizados, por ejemplo, nuestra BBDD, la *UI*... en esta capa deberá haber muy poco código, pues será una capa completamente naive, sólo recibirá datos y los mostrará si es *UI*, los guardará si es BBDD, o los enviará si es Servidor.

¿Se puede modificar el Diagrama de Uncle Bob?

Si y no. Uncle Bob propone éstos cuatro círculos a modo de abstracción para poder entender su Arquitectura e idea. Nuestra aplicación podrá tener más si así lo consideramos, siempre y cuando respetemos las reglas de dependencia anteriormente mencionadas. Pero nunca tendrá menos, ya que perderíamos el desacoplamiento que pretende esta Arquitectura.

4.1.1. Ventajas

Aplicando *Clean Architecture* y respetando las normas establecidas independientemente de si posteriormente se implementa mejor o peor, como mínimo obtendremos un Software robusto y cumpliendo los objetivos de ésta. Nos proporcionará un Software fácil de mantener, puesto que el desacoplamiento que propone entre nuestras capas, proporcionará a nuestro programa del principio *SOLID Open/Closed*, ya que estará abierto a extensión pero cerrado a modificación; conseguiremos que nuestro Software sea nuestro y no dependa de cambios externos.

4.1.2. Desventajas

La única desventaja que puede tener esta Arquitectura es si la aplicamos a proyectos pequeños y que sepamos que no van a escalar. ¿Porqué? Puede ser muy tedioso si nuestro proyecto es pequeño descomponer éste en tantas capas, tantos tipos de Objetos, definición de los Casos de Uso pueden contener código duplicado y un gasto extra de tiempo que no merecería la pena si el proyecto es pequeño.

Por supuesto, siempre que sepamos que nuestro Software no va a escalar. Pero, ¿Cuántas veces se comete ese error? Se cae en el error de pensar, este programa no escalará o este programa no se tendrá que mantener... Y no se invierte tiempo de Arquitectura. Por lo tanto, esta desventaja es algo ambigua, y siempre como desarrollador recomendaré invertir tiempo en la Arquitectura, ya que ésta nos hará la vida más fácil.

4.2. Aplicación en Android

Para la aplicación en Android de *Clean Architecture*, primero necesitamos saber que necesidades o problemas tiene una aplicación en Android, es decir, pensar en sus componentes externos y desacoplarlos de nuestras reglas de negocio.

En una aplicación Android, tendremos varios elementos susceptibles a cambios, nuestra base de datos puede cambiar, nuestra Interfaz puede cambiar y nuestro Servidor que nutre de datos a nuestra aplicación también puede cambiar.

¿Qué no debe cambiar?

Nuestro núcleo como aplicación, nuestras reglas de negocio, nuestro modelo de datos y nuestra funcionalidad, cerrada a cambios y abierta a extensión. Para conseguir esto, en Android, se propone, primero, una separación a nivel de Código Fuente en tres módulos, que nos ayudarán como programadores a seguir la idea de *Clean Architecture*, estableciendo dependencias entre estos, y definiendo qué y cómo actuará cada módulo de nuestra aplicación.

A continuación, una breve explicación sobre qué contendrá cada módulo, ya que serán explicados mas adelante en extensión.

4.2.1. Módulo presentation

Nuestro módulo *presentation* en Android, contendrá las capas de Adaptadores de Interfície y *UI*, es decir, recibirá datos de capas inferiores y se encargará de nutrir con estos a nuestra Interfaz de Usuario.

4.2.2. Módulo domain

Este módulo contendrá los Casos de Uso presentes en el diagrama de *Clean Architecture* y será ésta quién dirija el flujo de datos conectando entidades con capas superiores. También tendrá nuestros objetos de negocio, propios de la aplicación, nuestro modelo de datos.

4.2.3. Módulo data

El módulo data será el encargado de la conexión con los agentes externos Servidor y Base de Datos, proporcionando a los casos de uso los datos necesarios y desacoplando el modelo de datos propio del servidor de el nuestro. Hará lo mismo con nuestra base de datos.

5. Desarrollo

5.1. Módulo presentation

Nuestro módulo presentation contendrá todas las vistas de nuestra Aplicación, *Activies* y *Fragments* que recibirán y permitirán al usuario interactuar con nuestra aplicación. A su vez, este módulo será el encargado de escuchar los eventos producidos tanto por el Sistema Operativo, como por el usuario, ya sea pidiendo datos hacia capas más internas, restaurando éstos o dando la orden de persistir datos ante una acción del usuario.

Para la implementación de este módulo se proponen dos patrones de diseño; tendremos el patrón *Model-View-Presenter* y el patrón *Model-View-ViewModel*, así como librerías y *Frameworks* que nos ayudarán a conseguir una gran arquitectura para este módulo y seguir los puntos SOLID.

5.1.1. Patrón Model View Presenter

El patrón *Model View Presenter*, en adelante MVP, nace del famoso patrón de diseño Modelo Vista Controlador, en adelante MVC. El patrón MVP propone la separación en Objetos Modelo, Objetos de Vista y Objetos Presenter, como podemos observar en la figura 13.

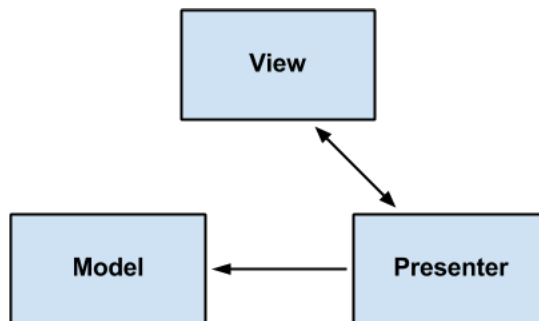


Figura 13: Model View Presenter

Para seguir el patrón MVP tendremos en cuenta los siguientes puntos [7]:

- Nuestro módulo de *presentation*, sólo trabajará con objetos *Model*; estos objetos contendrán únicamente la información necesaria para mostrarse en la Interfaz obteniendo, así, una capa de datos mucho más liviana y con los datos en memoria estrictamente necesarios.
- Los objetos Vista, en Android, *Activities* y *Fragments*, recibirán datos y los mostrarán y avisarán al *Presenter* de los eventos que vayan sucediendo a lo largo de su vida. En ningún caso contendrán lógica. A su vez, estas Vistas se comunicarán con el Presenter a través de Interfícies desconociendo, así, su implementación, consiguiendo desacoplar la Vista del *Presenter*, y permitiéndonos que el conjunto sea mucho más testeable.

- Los objetos *Presenter* serán los encargados de estar a la escucha de los eventos ocurridos en los objetos Vista y realizar las las acciones correspondientes para nutrir de datos a ésta. A su vez, éste formateará los datos que reciba para transformarlos a objetos de tipo *Model* y así conseguir el desacoplamiento entre la Interfaz y nuestra capa de datos propia. También será el encargado de notificar eventos a la Vista, e.g. cuando el Usuario utilice el botón Guardar, la Vista utilizará el método de la Interfície save, el *Presenter* delegará en el Caso de Uso correspondiente y, al acabar, será el *Presenter* quien avise a la Vista de que debe dar *feedback* al Usuario.

Puede parecer muy parecido a MVC. ¿Son lo mismo? No, no lo son. Entre estos dos patrones hay una gran diferencia que marca los pros y contras de cada uno de estos patrones y nos permite saber cuándo utilizarlos.

En el patrón MVC, es el Controlador quien decide qué Vista mostrar, qué Vista crear y, por lo tanto, varias vistas dependenderán del mismo Controlador, como se muestra en la figura 14. Sin embargo, en el patrón MVP, es la Vista quien crea al *Presenter*, y cada vista tendrá su presenter con el que se comunicará a través de una Interfície, tal y como se muestra en la figura 15.

¿Cuándo utilizar MVP y cuándo MVC? La respuesta es clara: según la dimensión de tu proyecto. En un proyecto pequeño, nos puede resultar útil MVC, tener un sólo controlador que interactúe con unas pocas vistas y las nutra de datos, centralizando en un sólo objeto la lógica de esta capa. En un proyecto grande o con previsión de gran escalabilidad, se utilizará MVP, ya que tener un Objeto Presenter para cada Objeto Vista nos permitirá tener asociado a cada pantalla el objeto que la opera resultando, así, más fácil de mantener, de entender y de extender/modificar. Si tenemos N vistas, y utilizamos MVC, esas N vistas estarán orquestradas por el mismo Controlador, resultando tedioso lidiar con él. En Android, la respuesta es clara, MVP. Android está ligado fuertemente a las vistas, tendremos *Activities* y *Fragments*, cada pantalla que creemos, deberá tener como mínimo una *Activity* y un *Fragment*, y centralizar todo el un *Controller* dará muchos problemas.

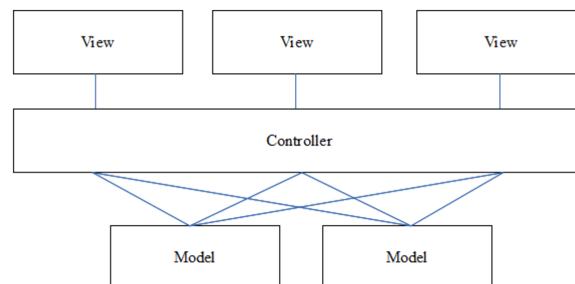


Figura 14: Model View Controller

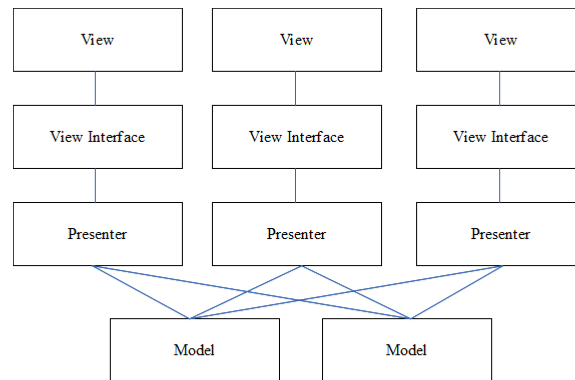


Figura 15: Model View Presenter

5.1.2. Model View ViewModel

Otra opción para la arquitectura de nuestro módulo de presentación es el Patrón *Model-View-ViewModel*, en adelante MVVM. Este patrón propone una relación directa entre los Objetos Vista y los Objetos *View-Model*, representada en la figura 16. Los Objetos Modelo y Vista serán lo mismo y funcionarán igual que en los patrones vistos MVP y MVC. La diferencia está en el objeto que los comunica. En el patrón MVVM, el Objeto que los comunica es el *Model-View*, éste recibe los eventos de la Vista y los delega en el Modelo, y cuando obtiene estos datos, los formatea como hacía el *Presenter* en el patrón MVP pero, en este caso, cuando el Objeto *View-Model* recibe los datos que necesita, se muestran directamente. ¿Cómo? Haciendo uso del *Data-Binding*, presente en muchos lenguajes y plataformas y muy utilizado.

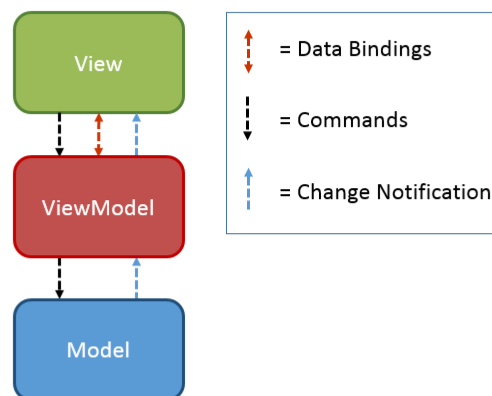


Figura 16: Model View View-Model
Fuente: Racher Software Ltd.

5.1.2.1 Data Binding en Android

¿Qué es el *Data Binding*? El *Data Binding* en Android [8]fue anunciado junto a la llegada de Android Marshmallow y supuso un gran cambio en éste, ya que permitía al desarrollador olvidarse de actualizar los datos manualmente si éstos cambiaban y, también, olvidar la inyección vistas (explicada más adelante).

Esto es posible gracias a que, los archivos en extensión .xml utilizados en Android para el diseño y creación de la Interfaz conocen exactamente qué parámetro de qué Objeto se muestra en ese campo. Por lo tanto, cada vez que cambia el contenido de ese objeto, la Interfaz es automáticamente cambiada. Ese objeto es el *ViewModel*, el cual se encargará de pedir los datos, recibirlos, y actualizar lo necesario para que la vista se vea automáticamente actualizada.

Por lo tanto, nos ahorra la inyección de vistas, la actualización manual de los datos, la restauración de éstos... ¿Por qué descartarlo?

Actualmente los *Data Bindings de Android* aún están en fase de desarrollo, son muy susceptibles a cambios y no son completamente estables, lo que puede llevarnos a errores en tiempo de ejecución. Aun así, estoy seguro de que cuando el *Data Binding* en Android sea estable, será el más utilizado y dejará de lado a MVP ó MVC.

5.1.3. ButterKnife

5.1.3.1 Introducción a la inyección de vistas

Cuando se desarrolla para la plataforma Android, una parte muy importante es el diseño de la Interfaz de Usuario. Android utiliza *Layouts*, archivos en extensión XML en los que definimos el aspecto de nuestras vistas. Android nos proporciona objetos nativos como *Buttons*, *TextViews* y un largo etcétera que son completamente customizables y, además, cada objeto tiene sus propios eventos.

```
 <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
             android:layout_width="match_parent"
             android:layout_height="match_parent"
             android:orientation="vertical" >
    <TextView android:id="@+id/text"
             android:layout_width="wrap_content"
             android:layout_height="wrap_content"
             android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="Hello, I am a Button" />
</LinearLayout>
```

En este ejemplo de layout, tenemos declarados un objeto de tipo *TextView*, utilizados para mostrar texto y un objeto *Button* para que el usuario pueda hacer uso de él. Éstos layouts se asocian a una *Activity* o *Fragment* en el momento de su creación, de manera que ésta puede acceder a los eventos que proporcionan los objetos en el *Layout*.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

¿Cómo se accede a estos botones desde la *Activity* o *Fragment*? Una vez hemos asociado ese *Layout* a la *Activity* o *Fragment* deseada si, por ejemplo, queremos cambiar el texto visible en nuestro objeto TextView, deberemos hacer:

```
TextView textView = (TextView) findViewById(R.id.text);
textView.setText("Text Changed!");
```

¿Y si queremos saber cuándo el Usuario hace click en el botón?

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      textView.setText("User clicked button");
   }
});
```

Como podemos ver es algo simple pero, en una aplicación real, los *Layouts* no tendrán solo un botón y un campo dónde mostrar texto, pudiendo llegar a un gran número de líneas de código, sólo para poder acceder a las vistas desde nuestra *Activity* o *Fragment*.

Es aquí donde entra en juego *ButterKnife*. Nos permite, mediante anotaciones tan simples como @Bind o @OnClick, realizar las acciones anteriores, de Tal manera que todo lo anterior se resumiría a:

```
@Bind(R.id.text)
TextView textView;
textView.setText("Text Changed!");

@OnClick(R.id.button)
public void changeText(){
   textView.setText("User clicked button");
}
```

Es de gran utilidad la utilización de *ButterKnife* en nuestro módulo de Presentación para lidiar con el gran número de vistas que en una aplicación se utilizan, dejando el código mucho más limpio y con una inyección de vistas mucho más rápida y sencilla. Estas asociaciones se producen en el momento en que nosotros asociamos una *Activity* o *Fragment* con un *Layout* y hacemos la llamada *ButterKnife.bind(this)* en el estado *onCreate* de ésta.

5.1.3.2 Ventajas y desventajas

Las ventajas utilizando ButterKnife se pueden ver en el pequeño ejemplo mostrado; permiten al desarrollador trabajar con las vistas de una manera mucho más cómoda, enlazando estas desde la *Activity* o *Fragment* de una manera muy rápida, y clara, dejando el código mucho más limpio que si no la utilizáramos. La principal desventaja es que estamos añadiendo una librería por parte de terceros a nuestro proyecto, pero que podemos despreciar sabiendo que la empresa desarrolladora y que mantiene esta librería es Square y, que el proyecto, es *Open Source*.

5.1.4. Picasso

5.1.4.1 Introducción a inyección de Imágenes

Las aplicaciones Android cada vez tienen más dependencia a las imágenes, desde que se introdujeron en nuestra vida las redes sociales como Facebook, Twitter, Instagram y un largo etcétera, hecho que ha dotado a toda aplicación de un componente social en mayor o menor medida. Pero esta dependencia hacia la carga de imágenes no se debe sólo a un factor social; si, por ejemplo, estamos desarrollando una aplicación para un supermercado, tendremos un gran número de imágenes de productos y, en el caso de las redes sociales, las imágenes de perfil, por ejemplo, se repiten muchas veces.

Las imágenes tienen un tamaño considerable y, normalmente, no las tendremos en nuestro dispositivo cuando el usuario instale nuestra aplicación, de manera que tendremos que hacer un uso de Internet.

Dado este contexto, se encuentra la necesidad de tener en caché, ya sea en memoria o en disco, las imágenes que nuestra aplicación descarga para hacer un uso eficiente de éstas, ahorrando tarifa de datos y mejorando los tiempos de carga de nuestra aplicación.

Picasso nos ofrece una manera de cachear estas imágenes, pudiendo acceder a ellas una vez descargadas desde disco o desde memoria, según queramos configurarlo.

¿Cómo lo hace?

Picasso utiliza un cliente Http, desarrollado por ellos mismos, llamado OkHttp muy utilizado en el desarrollo Android a nivel global, para realizar las conexiones necesarias y descargar la imagen deseada. Para el cacheo de imágenes, tenemos dos opciones: cacheo en memoria y cacheo en disco, o utilizar ambas a la vez [10].

El cacheo en memoria crea una estructura HashMap en tiempo de ejecución, en el que se inserta cada imagen descargada, con la URL como *Key* y la imagen como *Value* para que, en caso de que intentemos descargar otra vez la misma imagen, no haya necesidad de acudir a Internet y tenerla disponible al momento. Es en tiempo de ejecución, por lo tanto, cuando nuestra aplicación sea cerrada, esta HashMap y las imágenes serán eliminadas. Al ser en memoria, el acceso a éstas será muy rápido.

El cacheo en disco, funciona exactamente igual que el cacheo en memoria, pero este guardará las imágenes descargadas en la memoria interna del dispositivo, de manera que sólo se borrarán si la aplicación es desinstalada del dispositivo. Al ser en disco, cada acceso a una Imagen será más lento que el cacheo en memoria.

¿Cuál utilizaremos?

Lo mas eficiente es utilizar ambos, ya que obtendremos el mayor beneficio. Cuando el usuario instale nuestra aplicación, las imágenes se irán guardando en disco y mientras la aplicación siga en ejecución se hará uso del cacheo en memoria, y cuando la aplicación sea cerrada y posteriormente abierta, se accederá a disco para crear una caché en memoria, de manera que no tendremos que hacer uso de Internet. Estas cachés tendrán el tiempo de caducidad que nosotros configuremos y podrán ser reseteadas cuando nosotros queramos, a veces una imagen puede ser diferente y tener la misma URL.

5.1.4.2 Otras opciones

Para la carga de Imágenes tenemos otras opciones, tales como *Glide*, *Universal Image Loader*, *Fresco*, *Volley...* todas ofrecen, en un principio, la misma funcionalidad y resulta algo complicado decidirse por una de ellas. Nos ayudaremos de la comparativa mostrada en la figura 17.

	Picasso	UImageLoader	Glide	Fresco
Creator	Square Inc	Sergey Tarasevich	Bumptech	Facebook
Maturity(Oldest)	★★★★½	★★★★★	★★★★☆	★★★½☆
Volley and OKHTTP	✓	✓	✓	✓
Used Personally	✓	✓	✓	✗
Customizability	★★★★☆	★★★★½	★★★★½	★★★★☆
Network Image Use	★★★★★	★★★★☆	★★★★½	★★★★☆
GIF Support	✗	✗	✓	⚠
Image cropping	★★★★☆	★★★★☆	★★★★☆	★★★★★
Ease of use	★★★★★	★★★★½	★★★★★	★★★★☆
Speed	★★★★½	★★★★☆	★★★★½	★★★★½
Documentation	★★★★½	★★★★½	★★★★½	★★★★★
Disk + Mem Cache	✓	✓	✓	✓
Personal Rating	★★★★★	★★★★☆	★★★★½	★★★½☆
USP	Fast,most popular	Most mature	= Picasso	Ashmem, By Facebook

Figura 17: Comparativa de diferentes Librerías para carga de Imágenes
Fuente: stackoverflow.com

Utilizaremos una u otra según nuestras necesidades y conocimiento. Si es nuestra primera vez utilizando una librería de este tipo y no necesitamos la carga de GIF's, utilizaremos *Picasso*, gracias a su facilidad de uso y gran comunidad detrás de ella.

Si necesitamos la carga de GIF's, utilizaremos *Glide*, y si necesitamos costumizar la librería necesariamente, podremos utilizar *Universal Image Loader* o *Fresco*.

5.1.4.3 Ventajas y desventajas

Las ventajas si utilizamos Picasso serán: una mayor rapidez de nuestra aplicación y un menor consumo de datos para nuestro futuro usuario. Además, nos podremos olvidar de ser nosotros quien realice todo el sistema de cacheo de imágenes, a parte de otras opciones que *Picasso* nos permite, como por ejemplo, transformaciones de estas imágenes.

La desventaja vuelve a ser la misma que con ButterKnife: añadimos una librería de terceros, pero esta también esta desarrollada por Square Inc. y es Open Source.

Ejemplo de uso

```
Picasso.with(context) //Necesita el Context
   .load(model.getThumbnailUrl()) //URL de la imagen
   .error(R.drawable.logo) //Imagen que mostraremos en caso de error
   .placeholder(R.drawable.logo) //Imagen que mostraremos mientras la
      real se descarga
   .transform(new CircleTransformation()) //Si queremos aplicar alguna
      transformación a esta
   .centerCrop() //Como recortaremos la imagen para ajustarla al tamaño
      deseado
   .fit() //Rellenaremos todo el espacio disponible
   .into(holder.thumbnail); //ImageView de destino
```

5.1.5. Dagger2

5.1.5.1 Introducción a IoC

En Ingeniería del Software [11], IoC es el término para hacer referencia a *Inversion Of Control*, un principio de diseño que afecta directamente a como debemos desarrollar nuestro Software.

> ”Don’t call us, we’ll call you”
> Hollywood Principle

Esta cita, fuertemente relacionada con el concepto de IoC, explica qué es y porqué sucede. Anteriormente, cuando nosotros escribíamos código, éste era lineal y procedural. Sabíamos en que momento íbamos a darle el control a la plataforma para que ella hiciera su trabajo, es decir, el flujo de trabajo era lineal. La necesidad de *Inversion Of Control* se produce cuando, es la plataforma o el framework quién realiza sus acciones y posteriormente nos da el control a nosotros, pudiendo ejecutar las acciones deseadas y posteriormente devolviéndole el control al mismo. Con todo lo explicado en esta memoria hasta este momento, podemos ver como, si desarrollamos para Android nuestro código está fuertemente unido a cuando el Sistema nos da el control, por lo tanto si queremos crear un buen Software, necesitaremos aplicar los principios de la *Inversion Of Control*, definidos una vez más por Robert C Martin y Martin Fowler, éste último introdujo el mencionado patrón MVVM. Para poder aplicar el concepto de IoC, debemos saber que su principal objetivo es desacoplar la ejecución de una tarea de su implementación.

5.1.5.2 Inyección de dependencias

La inyección de dependencias nace como consecuencia de la *Inversion Of Control*, y está directamente relacionada con SOLID. La D significa *Dependency Inversion Principle*. La inyección de dependencias tiene como objetivo desacoplar, en Programación Orientada a Objetos, un Objeto de la Implementación de Otro, para ser más claro, se propone un ejemplo:

Imaginamos una clase Vehículo, todo Vehículo tiene una clase Motor, por lo tanto Vehiculo depende fuertemente de Motor. ¿Cómo podemos desacoplar Vehículo de Motor? Inyectando esta dependencia.

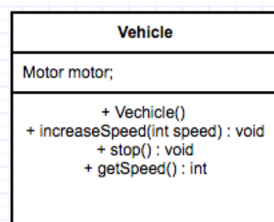


Figura 18: Vehículo crea Motor en su constructor

La figura 18 representa nuestro caso base, en el que la clase Vehículo crea directamente en su constructor una instancia de Motor, fuertemente acoplada. Veamos cómo podemos desacoplarlo.

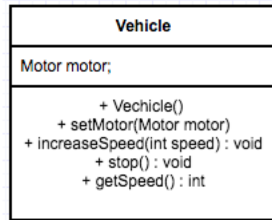


Figura 19: Inyección por Setter

No es mala idea, si Vehículo recibe el Motor por un Setter, la creación de éste ya no depende de la clase Vehículo y estamos desacoplando Vehículo de Motor ya que éste no debe crearlo, tal y como se muestra en la figura 19 pero, aún así, ¿Y si olvidamos setear ese Motor?

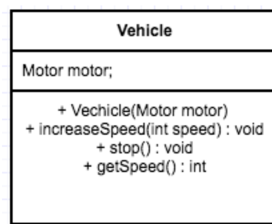


Figura 20: Inyección por Constructor

Vamos mejorando, ahora cuando el desarrollador quiera crear un Vehículo, se está obligando a que éste envíe la instancia de Motor, como se observa en la figura 20. ¿Se puede mejorar? Seamos un poco más SOLID.

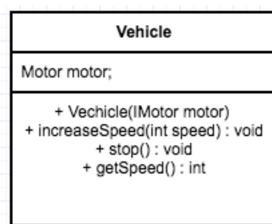


Figura 21: Inyección por Constructor e Interfície

Si nosotros proporcionamos la instancia que consideremos oportuna de Motor y Vehículo sólo conoce la Interficie de éste, tenemos completamente desacoplado Vehículo de Motor, como se observa en la figura 21. Buen trabajo.

5.1.5.3 Patrón Factory

¿Podríamos mejorarlo?

Partiendo del último caso, en el que se consigue desacoplar la clase Vehículo de Motor, delegamos la responsabilidad de nutrir a Vehículo con la implementación deseada de la Interfície Motor a la clase u objeto que haga uso de Vehículo. Si seguimos este ejemplo y tenemos la Interfície Motor, a la que el desarrollador deberá pasarle la implementación correcta, hay un patrón que resulta óptimo para esta necesidad. Ese patrón es el *Factory*, haciéndo uso de este patrón, conseguiríamos delegar la creación de la implementación de la Interfície Motor a nuestro MotorFactory, de manera que desacoplaríamos la clase que haga uso de Vehículo con las implementaciones de la Interfície Motor.

Patrón Factory

Esta es la esencia de *Dagger2*, un creador de Factories que tratará a nuestros objetos como Interfícies a las que dar una implementación. Estas Factories son creadas en tiempo de compilación, de manera que no sufriremos una penalización de rendimiento en tiempo de ejecución. Para que Dagger2 sepa como crear estas Factories necesitaremos hacer uso de los Actors, que serán explicados en el siguiente punto.

Como se puede ver en la figura 22, el patrón *Factory* consiste en utilizar una clase constructora llamada *Factory* que será la encargada de retornar la implementación de una Interfície para ser consumida por la clase que utilice este *Factory*. Aplicado al caso visto, deberíamos tener el siguiente escenario:

- Motor.java Clase simple para representar el Motor de un Vehículo.
- Vehiculo.java Nuestra clase que depende de Motor.java, se le proporciona la instancia de Motor en el constrcutor de Vehiculo.
- IFaceMotor.java Interfície que implementará Motor.java, contendrá la abstracción de los métodos que todas las intancias Motor.java deberán tener.
- MotorFactory.java Nuestra *Factory*, tendrá un método que retorne una implementación de IFaceMotor, normalmente este método recibe como parámetro algún dato para hacer de descriminador en el momento de creación de la implementación demandada.

Cada vez que nosotros inyectemos una dependencia en *Dagger2* se realizará este proceso, ahorrándonos este trabajo y añadiendo funcionalidades extras.

5.1.5.4 Actors

A continuación se enumeran los *Actors* disponibles en *Dagger2* [12].

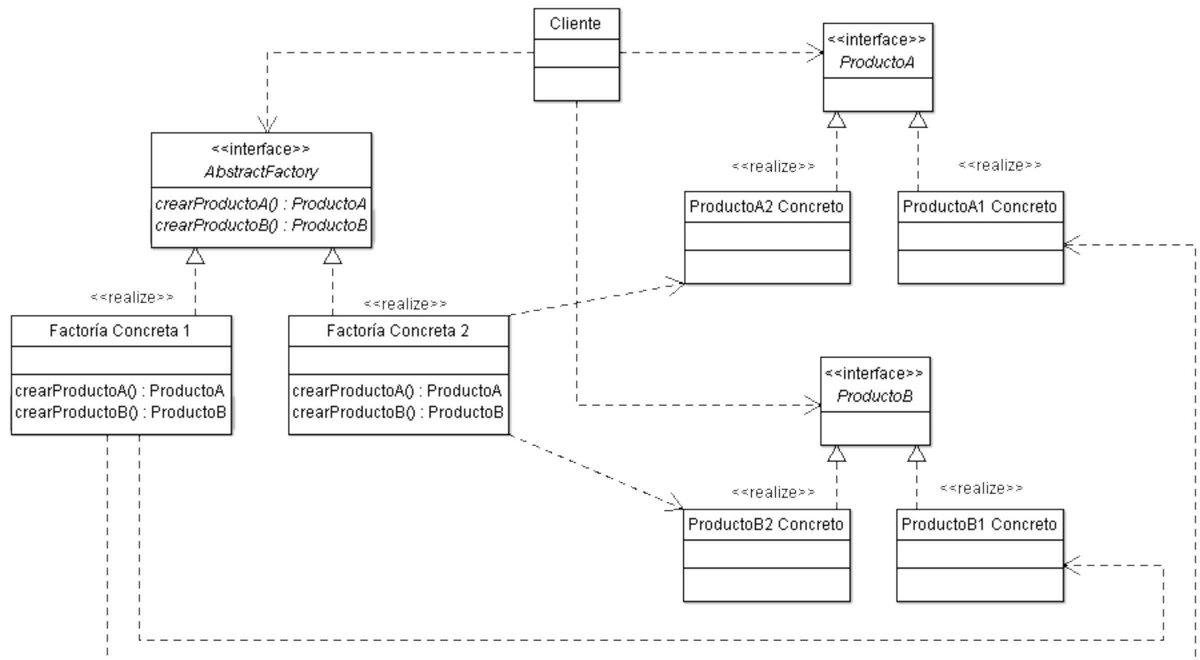


Figura 22: Patrón Factory
Fuente: wikipedia.com

- @Inject: Con esta anotación, se pide la inyección de la dependencia. Es decir, pedimos a *Dagger2* que inyecte la clase, parámetro o atributo que se necesita. *Dagger2* creará la instancia de la clase que haya sido anotada y, a su vez, si ésta en su interior tiene más dependencias, las resolverá también. Es nuestra manera de iniciar el proceso de inyección.

- @Module: Las clases que contengan la anotación *Module*, serán aquellas que proveen las dependencias.

- @Provide: Esta anotación se utiliza en las clases *Module*, y nos sirve para decir a *Dagger* cómo queremos que construya la dependencia mencionada.

- @Component: Los Component son los Inyectores, hace de puente entre la etiqueta @Inject, que demanda la inyección y la etiqueta @Module, que proporciona el cómo inyectar. La responsabilidad del *Component* es unirlos.

- @Scope: Los *Scopes* son muy útiles en *Dagger 2*, nos sirve para indicar a *Dagger* cómo tiene que manejar las instancias que posee y cuando debe liberarlas. Por ejemplo, la etiqueta en un *Scope* de @PerActivity, indica que las instancias de los objetos que posea en ese *Scope*, vivirán mientras la *Activity* asignada viva, lo mismo para @PerFragment, para un *Fragment*. Si no existe, se mantendrá durante toda la vida de la Aplicación.

- @Qualifier: Nos permite delimitar el uso de nuestros *Component*, es decir, cuándo podremos utilizar éstos para inyectar dependencias. Por ejemplo, disponemos de las etiquetas @ForActivity, si sólo se podrá utilizar en *Scopes* de *Activities*, @ForApplication, si sólo podremos utilizarla en *Scopes* de *Application*. A su vez, nos servirá para indicar a *Dagger* qué tipo de Contexto debe utilizar, ya que no es lo mismo *Context*, que *ApplicationContext*.

- @Singleton: Si utilizamos esta anotación, *Dagger2* creará un patrón *Singleton* para el objeto que nosotros anotemos, de manera que sólo existirá una instancia de ese Objeto. El funcionamiento del patrón *Singleton* se explicará a continuación.

Los *Actors* que marcan el funcionamiento de *Dagger2* son *Inject*, *Component* y *Module* como podemos observar en la figura 23.

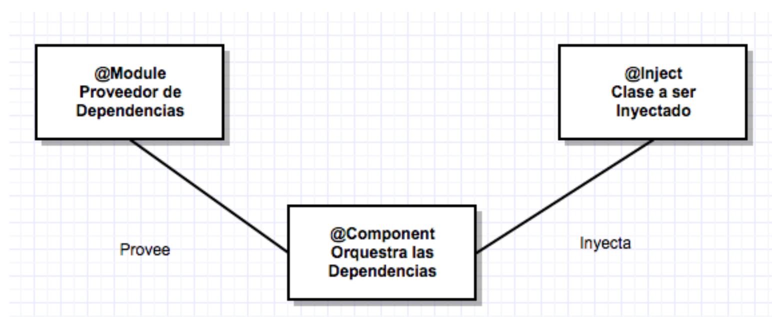


Figura 23: Abstracción de los Actores Module, Inject y Component

Patrón Singleton

El patrón *Singleton* es un patrón de diseño que tiene como objetivo restringir la creación de objetos de una misma clase. De manera que nos proporciona la seguridad de tener sólo una instancia del mismo objeto en tiempo de ejecución y, además, proporcionar un punto de acceso a este objeto, tal y como se puede apreciar en la figura 24.

Para implementar este patrón, crearemos un método estático, en el caso de Java, que retornará la instancia del Objeto y, si aún no existe, se creará. Para asegurar que la clase no puede ser instanciada sin utilizar el patrón se modifica la visibilidad y el acceso del constructor por defecto, de manera que habrá que hacer uso del método, normalmente llamado *getInstance();* para poder acceder al objeto.

El patrón singleton provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase estático y público.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

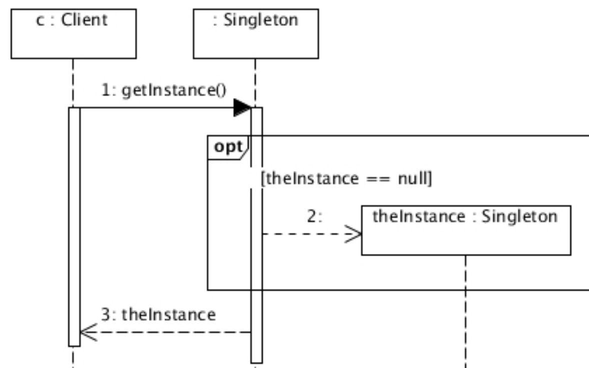


Figura 24: Diagrama de Secuencia en el patrón Singleton
Fuente: vainolo.com

5.1.6. Aplicación del módulo presentation en TweetStatt

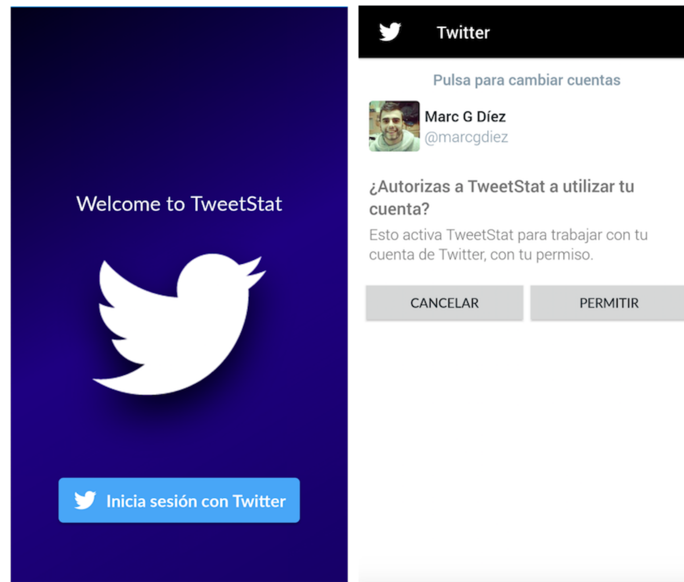
5.1.6.1 Diseño de la Interfície Gráfica

Este trabajo se centra en la arquitectura de la aplicación, pero también pretende poder ser utilizado a modo de guía para un desarrollador, por lo tanto se mostrarán las diversas pantallas y se explicará los componentes gráficos utilizados. Todos los componentes que se mencionarán a continuación pueden encontrarse en la documentación de Android [13].

Pantalla LoginActivity

Para esta pantalla, mostrada en la figura 25 se ha utilizado un *RelativeLayout*, que contiene el background que podemos ver, un *ImageView* con el logo de Twitter, y finalmente, un *Button* para que el usuario pueda logearse con su cuenta de Twitter y poder utilizar la aplicación. Se encuentra en el proyecto con el nombre de activity_login_screen.xml

Cuando el usuario haga click en el botón del Login se lanzará la pantalla siguiente, ofrecida por el SDK de Fabric, el cual se explicará más adelante.



(a) LoginActivity (b) Login By Twitter

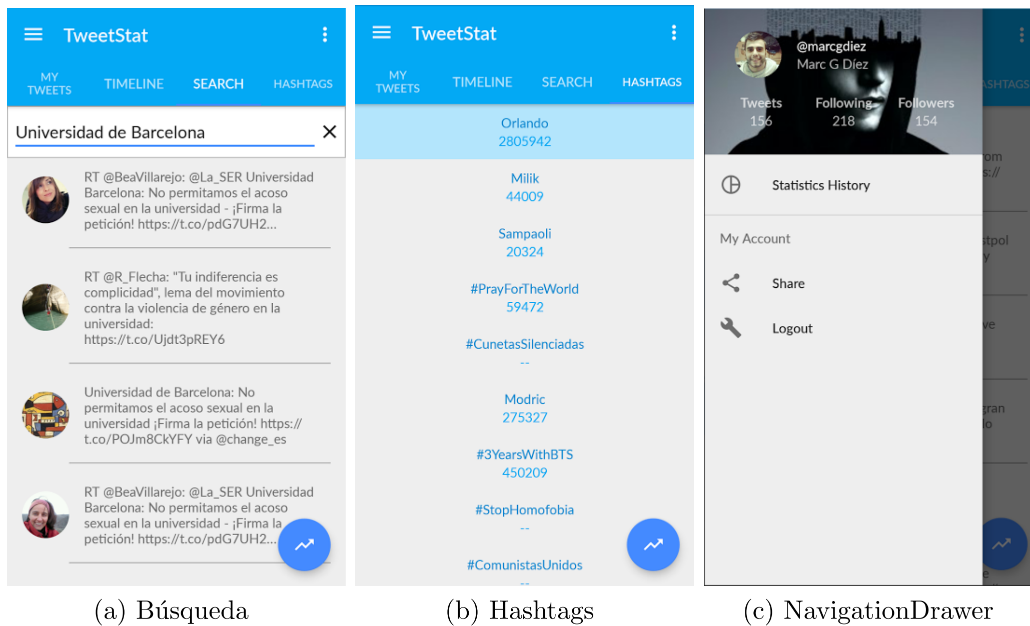
Figura 25: Pantallas de Login

Cuando el usuario presione el botón de Permitir, se lanzará la siguiente Activity.

Pantalla Principal

Figura 26: Pantalla Principal

La pantalla mostrada en la figura 26 será nuestra MainActivity. Para esta *Activity* se han utilizado los componentes *Toolbar*, para la barra superior horizontal, presente ya en todas las pantallas, los componentes *ViewPagers* y *TabLayout* para poder crear esa barra horizontal que contiene las pestañas que se pueden ver en la imagen. Todo esto, estará dentro de un *DrawerLayout* y un *NavigationView* para poder dibujar la pantalla de Navegación que se abrirá si presionamos el icono cercano



(a) Búsqueda (b) Hashtags (c) NavigationDrawer

Figura 27: Varios Fragments en la MainActivity

a TweetStatt o si hacemos un *swipe* de izquierda a derecha. También contiene el *Floating Action Button*, presente en la parte inferior de la pantalla. Podemos encontrar este *Layout* bajo el nombre de activity_main.xml.

El resto de contenido de la pantalla, donde podemos ver, en este caso, mis tweets, pertenecerá a cada *Fragment* respectivamente. Para las listas de tweets se ha utilizado el componente *RecyclerView*, *RecyclerAdapter* y *ViewHolder* necesarios para el funcionamiento de éste. En la figura 27 se muestran las pantallas mencionadas con anterioridad.

Pantalla de Estadísticas

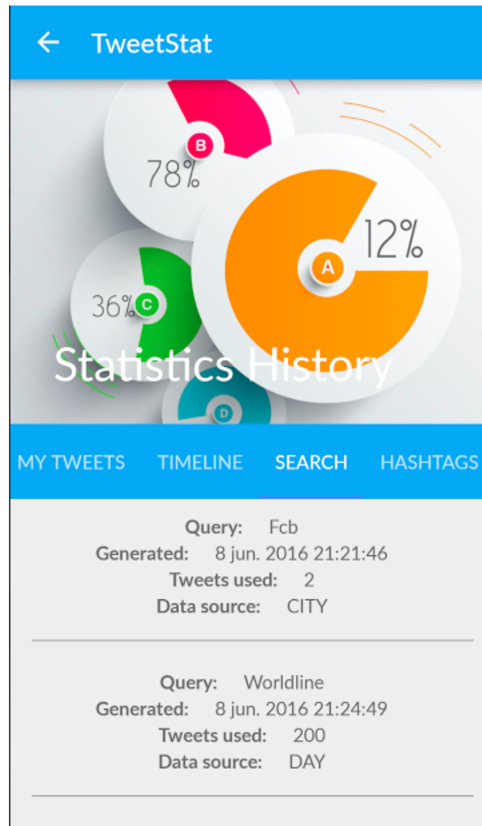
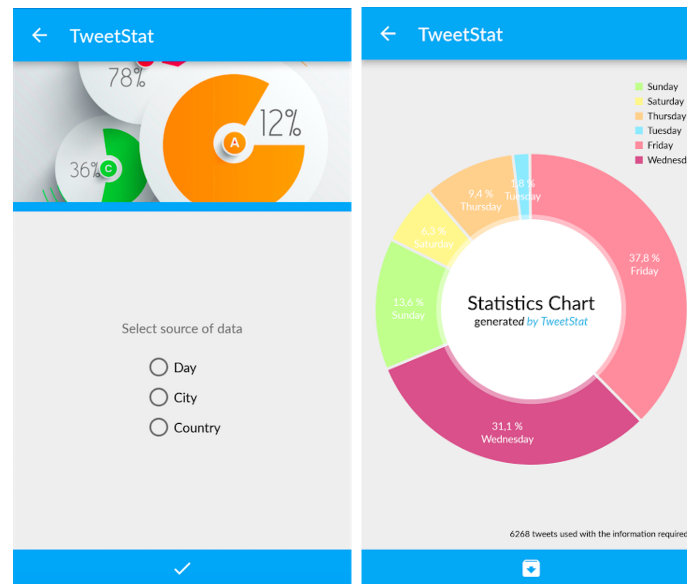


Figura 28: Pantalla Historial de Estadísticas

Para esta pantalla, mostrada en la figura 28, se utiliza el *Layout* presente en el proyecto bajo el nombre history_statistics_activity.xml. En éste, se encuentran los componentes *CoordinatorLayout*, componente nativo de Android que permite, añadiendo en su interior el componente *CollapsingToolbarLayout*, realizar el efecto “contracción”de la imagen al deslizar la pantalla. A su vez, contiene un *ViewPager* y un *TabLayout*, como la *MainActivity* para poder utilizar la barra de pestañas y que cada *Fragment* pueda trabajar.

Pantalla FullGraphActivity

Estas dos pantallas, mostradas en la figura 29, conviven en la misma *Activity*, y está hecha para ver como podemos jugar con la visibilidad de los *Layouts* ante acciones del usuario. Están definidos dos *Layouts* en esta *Activity*: el primero permite al usuario seleccionar entre DÍa, Ciudad y Localidad. El segundo, la *bottom bar*, para realizar la estadísitica. Cuando la estadística está lista, este *Layout* se oculta y se muestra el gráfico con la *bottom bar* para poder realizar la acción de guardado.



(a) Selección de la fuente de datos

(b) Presentación del Gráfico generado

Figura 29: Pantallas para generar y presentar las estadísticas

5.1.6.2 Implementación de Activities y Fragments

Como se ha dicho anteriormente, es una buena práctica en Android utilizar el mayor número de *Fragments* gracias a la gran capacidad de reutilización, por lo tanto para el diseño de vistas, casi todas las *Activities* tienen uno o más *Fragments* asociados, las únicas *Activities* que no contienen *Fragment* son aquellas que son muy simples en implementación.

- LoginActivity: Esta *Activity* no contiene *Fragments*, ya que su única función es delegar en el SDK de Fabric para la realización del Login del Usuario.

- MainActivity: Es la *Activity* principal, contiene SearchTweetsFragment, HashtagsTweetsFragment, HomeTimelineTweetsFragment y UserTimelineTweetsFragment, todos estos *Fragments* extienden de BaseFragment que contiene métodos que se repiten en todos los *Fragment*, así como instancias de Objetos útiles, como el RxBus, que se explicará más adelante. A su vez, para la implementación del patrón MVP, estos *Fragments* contenidos en la MainActivity se comunican con la interfície TweetsPresenter, que recibirá diferentes implementaciones.

- HistoryStatisticsActivity: Esta *Activity* será la encargada de presentar al Usuario todas las estadísticas que haya guardado en la aplicación, en las cuatro categorías, según la fuente de datos; por ello, contendrá los *Fragment* HashtagsStatisticFragment, SearchStatisticFragment, HomeTimelineStatisticsFragment y UserTimelineStatisticsFragment: todos estos Fragments también heredan de BaseFragment y se comunican con la interfície StatisticsPresenter, para implementar el patrón MVP, también recibirá diferentes implementaciones.

- FullGraphActivity: Esta *Activity* no contiene *Fragments*, su función es la de presentar al usuario el gráfico realizado y se comunica con StatisticsPresenter.
- TwettStattBaseActivity: Es la *Activity* de la cual heredan todas las anteriores, como BaseFragment para los Fragments. Esta contiene métodos útiles para todas las *Activities*, así como instancias útiles para todas las *Activities*, e.g. la Toolbar.

5.1.6.3 Presenters

Como dicta el patrón MVP, las Vistas, explicadas en el anterior apartado, se comunican con interfícies de tipo *Presenter* y, a su vez, éstas reciben una implementación pero sin conocerla. Al mismo tiempo, cada Vista debe tener su implementación de *Presenter*, por lo tanto, tendremos tantos *presenter* como vistas hayan en nuestra aplicación. Todos los *Presenter* implementados, realizan la función de estar a la escucha de la *Activity* o *Fragment*, pedir los datos o generarlos y devolverlos a la vista.

- MainUserPresenter: Este es el *Presenter* al cual accede MainActivity para la carga de datos del Usuario que se pintan en el *NavigationDrawer*. Implementa la interfície UserPresenter y contendrá el Caso de Uso necesario, una referencia a la Vista asociada y el objeto de datos *UserModel*. Este será llamado y creado por la Vista al iniciarse y recibirá la petición por parte de la vista de obtener los datos del User. Cuando esto ocurra, MainUserPresenter ejecutará el *Interactor* correcto y retornará los datos a la vista para que los pinte.
- SearchTweetsPresenter, HashtagsTweetsPresenter, HomeTimelineTweetsPresenter y UserTimelinePresenter: Todos estos *presenter* implementan la interfície TweetsPresenter y serán utilizados por todos los TweetsFragment mencionados en el punto anterior. La única diferencia entre ellos será el caso de Uso que ejecuten y, en el caso de HashtagsTweetsPresenter el modelo de datos que tratarán. Todos tratarán con *TweetsModel*, el objeto Modelo para presentar un *Tweet*, y HashtagTweetsPresenter tratará con HashtagsModel, el objeto Modelo necesario para presentar un *Hashtag*.
- SearchStatisticsPresenter, HashtagStatisticsPresenter, HomeTimelineStatisticsPresenter y UserStatisticsPresenter: Estos *Presenter* implementan la interfície StatisticsPresenter y serán utilizados por los Fragment StatisticFragment. Como el resto de *Presenter*, ejecutarán cada uno el caso de uso correspondiente y trataran todos objetos modelo de tipo StatisticsModel, para poder presentar al usuario las diferentes listas de estadísticas.
- FullGraphPresenter: Es el presenter que utiliza FullGraphActivity para presentar los gráficos generados. Este recibirá un objeto de negocio por parte de la base de datos y lo transformará a la HashMap necesaria para la presentación del gráfico.

5.1.6.4 Model

Ahora es el turno de los objetos de Modelo, estos objetos contendrán la información mínima requerida para mostrarse al Usuario. Pongamos el ejemplo de la lista de *Tweets*, para mostrar un *tweet* tal y como se ha diseñado la aplicación sólo necesitamos la *URL* del Avatar del Usuario y el contenido del *tweet*, por lo tanto hemos de despreciar en este tipo de objeto el resto de datos. Contamos con varios objetos *Model*, presentados a continuación.

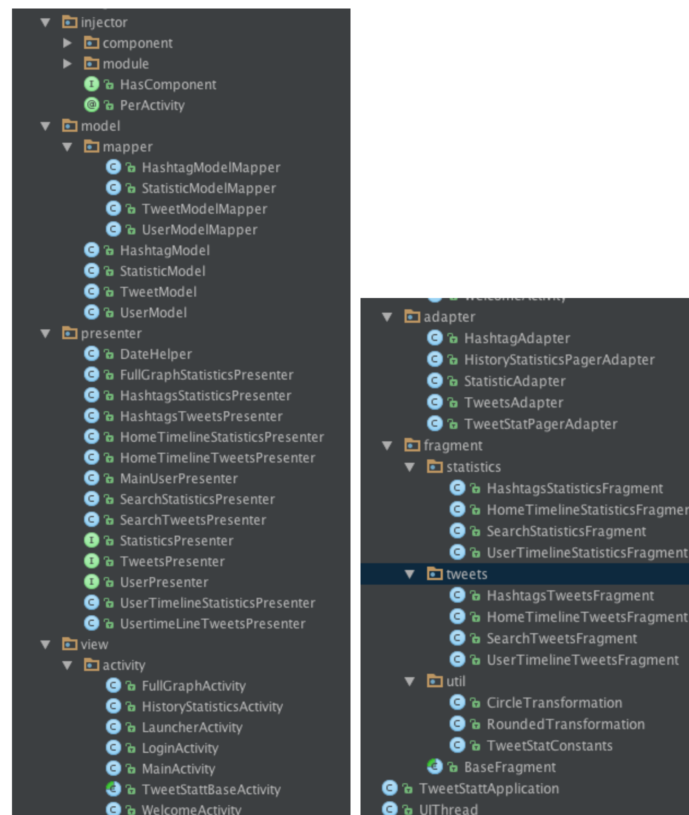
- UserModel: Cuando recibimos los datos de Usuario logeado, este contiene muchos datos que para la Vista son innecesarios. Por lo tanto, este objeto *UserModel*, sólo contendrá el nombre, su *nickname* en Twitter, la *URL* del avatar, la *URL* de su *header*, el número de *tweets*, el número de seguidores y el número de personas a las que sigue, es decir, los datos estrictamente necesarios que se pintan en la interfaz de Usuario.
- TweetModel: Contendrá únicamente la *URL* del Avatar de la persona creadora del *Tweet* y el texto del mismo. Evitando, así, que la vista reciba una gran cantidad de datos innecesarios para ella.
- HashtagModel: Este es el objeto Modelo utilizado para presentar los diferentes *Hashtags* que sean *Trending Topic* en el momento que el usuario actualice. Para presentar estos sólo necesitaremos el número de *Tweets* relacionados con ese *Hashtag*, el nombre que se pintará y la *query* que debemos hacer para obtener los *tweets* de este *hashtag*.
- StatisticModel: Objeto Modelo utilizado para presentar al Usuario su historial de estadísitcas. Contiene una ID, para poder recuperar los datos de la BBDD, la fuente de datos, para poder discriminar a la hora de mostrarlos, la fecha en la que fue generada esa estadística y, en caso de ser búsqueda o *hashtag*, la *query* realizada.

Model Mappers

Para poder realizar la conversión de Objeto de Negocio a Objeto de tipo Modelo que será el utilizado por nuestras vistas, se implementan varios *Mappers* que tendrán como función recibir un objeto de Negocio y retornar un objeto Modelo. Cada objeto de tipo Modelo tiene su Mapper asociado. Para esta aplicación se han desarrollado UserModelMapper, TweetModelMapper, HashtagModelMapper y StatisticsModelMapper, de manera que todos los *Presenter* puedan realizar la transformación de Objeto de Negocio a Objeto de Modelo necesitado por las vistas, discriminando los datos innecesarios.

5.1.7. Estructura del Módulo Presentation

Finalmente, nuestro módulo *presentation* queda en la estructura del proyecto en Android Studio como se puede observar en la figura 30.



(a) Packages injector, model, presenter y Activities

(b) Packages Adapter y Fragments

Figura 30: Estructura a nivel Código Fuente del módulo Presentation

5.2. Módulo dominio

Este módulo contendrá los Casos de Uso, también llamados *Interactors*, presentes en el diagrama de *Clean Architecture* y será esta unidad la encargada de dirigir el flujo de datos conectando entidades con capas superiores. También tendrá nuestros objetos de negocio, propios de la aplicación, nuestro modelo de datos y será la encargada de realizar toda la funcionalidad de la aplicación. Para el desarrollo de este módulo y su explicación se procede a explicar las tecnologías investigadas y su posterior aplicación.

5.2.1. RxJava

RxJava es un *Framework* que trata de facilitar la programación asíncrona y basada en eventos mediante el uso de *Observables*. Se basa en el patrón *Observer* al que le añade operadores que nos permiten componer distintos *Observables* además de ayudarnos a abstraernos de problemas cómo *threading* y sincronización, ya que incorpora una gran ayuda para tratar estos últimos.

RxJava es ideal para usar en Android [14], ya que para conseguir una buena experiencia de usuario se deben hacer todas las operaciones en un hilo en *background*, lo que obliga a todos los desarrolladores de Android a lidiar con problemas de concurrencia y asincronía. Por ejemplo, se recomienda que toda operación que supere los 250ms de duración, no se haga en el hilo principal para que el usuario no note el famoso *"lag"* y tener una mejor experiencia de usuario, e.g. en Windows 8, la pantalla conocida como *"Metro"*, el tiempo máximo establecido para los desarrolladores fue de 50 ms y, de esa manera, cualquier operación que superara este tiempo debía ir un *thread*.

Por ello RxJava se presenta como la alternativa perfecta a otras abstracciones comúnmente usadas en Android como *AsyncTask*, la cual presenta varios problemas tales como la falta de *composability*(capacidad de componer distintas unidades), *memory leaks* y diferencias de funcionamiento en las distintas versiones de Android. Además RxJava se integra con otros *frameworks* ampliamente usados en Android como *Retrofit*, que será explicado en esta memoria más adelante.

5.2.1.1 Introducción a la programación reactiva

5.2.1.2 Patrón Observer

RxJava nace del patrón *Observer*. Esta adaptación, realizada por la empresa Netflix y liberada como *framework* de *Open Source* - bajo NetflixOSS, la plataforma Open Source de la empresa - es uno de los secretos de su Software para la escalabilidad y rendimiento que obtiene en su producto. El patrón *Observer* necesita de los objetos presentados en la figura 31.

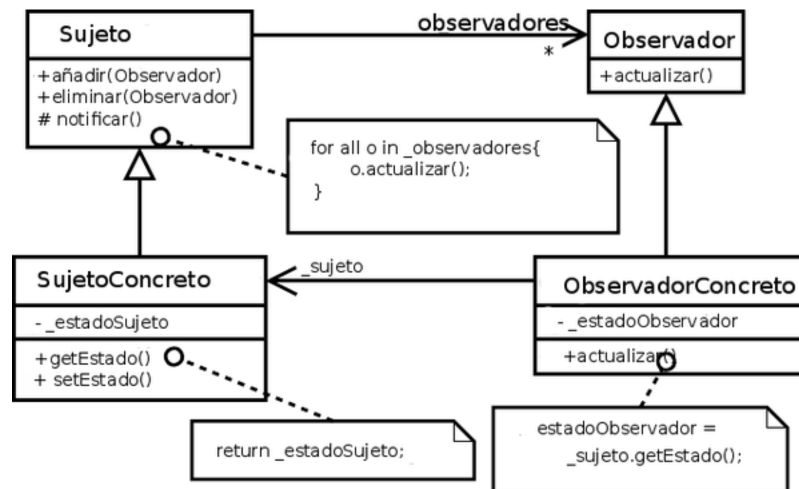


Figura 31: Patrón Observer
Fuente: wikipedia.com

- Sujeto (*Subject*): El sujeto proporciona una interfaz para agregar (*attach*) y eliminar (*detach)*observadores. El Sujeto conoce a todos sus observadores.
- Observador (Observer): Define el método que usa el sujeto para notificar cambios en su estado (*update/notify*).
- Sujeto Concreto (*ConcreteSubject*): Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- Observador Concreto (*ConcreteObserver*): Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

RxJava nos proporciona las implementaciones de todos estos objetos de manera que nos permite utilizar el patrón *Observer* sin tener que implementarlo desde cero. Nos proporciona las clases *Observable* y *Observer*, añadiendo a los primeros una serie de *Operators* con los que realizar acciones sobre ellos, y como ayuda al *threading* y asíncronía una serie de *Schedulers*, que se comunicarán siguiendo este Patrón, como se muestra en la figura 32. Partiendo de la base de la definición del Patrón *Observer* y los objetos necesarios, veremos como se ha implementado por parte de Netflix - aunque ahora se conoce como *ReactiveX* - y sus características.

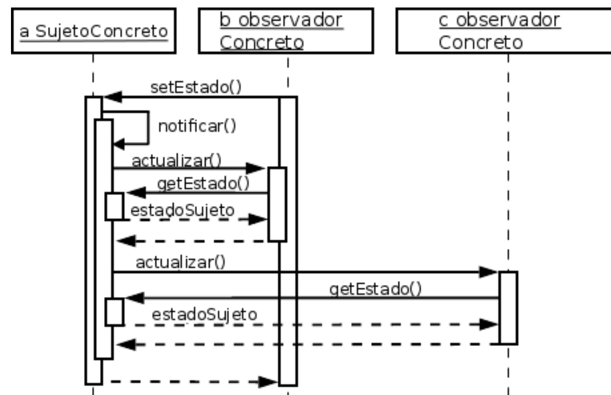


Figura 32: Secuencia Notificación y actualización en el patrón Observer
Fuente: wikipedia.com

5.2.1.3 Observables

En la programación *ReactiveX* [15], un *Observer*, en adelante *Subscriber* para facilitar la compresión, se subscribe a un *Observable*. El *Subscriber* reaciona a los objetos u objeto que el *Observable* emita. Este patrón facilita las operaciones concurrentes ya que no necesitas bloquear el *thread* principal mientras este *Observable* emite objetos.

Notificaciones por parte de los Observables

¿Cómo sabemos cuándo un *Observable* emite datos? Como sabemos, un *Subscriber*, se subscribe a un *Observable*, de manera que este tendrá conciencia de que el *Observable* emite datos. Para ello, el *Subscriber* implementa tres métodos que serán llamados por el *Observable* ante esos eventos.

- onNext(T Object): Este método será llamado por el *Observable* cada vez que emita un Objeto y el *Subscriber* lo recibirá como parámetro.
- onCompleted(): Este método será llamado cuando el *Observable* haya acabado de emitir objetos.
- onError(): Este método será llamado cuando al intentar crear el *Observable*, o durante la vida de el mismo ocurra algún error, e.g. un *SocketTimeOut* en una llamada al Servidor.

5.2.1.4 Operators

Los operators son las funciones que nos permiten realizar operaciones sobre los *Observables*. La mayoría de *Operators* reciben como parámetro un *Observable* y retornan un *Observable* una vez realizada su operación, por lo tanto y para sacar su máximo potencial, lo ideal es concatenar estas operaciones sabiendo que se ejecutarán todas ellas cada vez que el *Observable* emita un objeto. También existen

Operators para crear *Observables*. A continuación se muestra una breve lista sobre los *Operators* más utilizados, ya que existen una gran cantidad. Todas las imágenes que se muestran a continuación son extraídas de la documentación oficial de *ReactiveX* (http://reactivex.io)

Map Operator

Este operador permite aplicar la función deseada a cada item que emita el *Observable* y retorna otro *Observable* con los resultados de ésta. Podemos ver como ejemplo la figura 33.

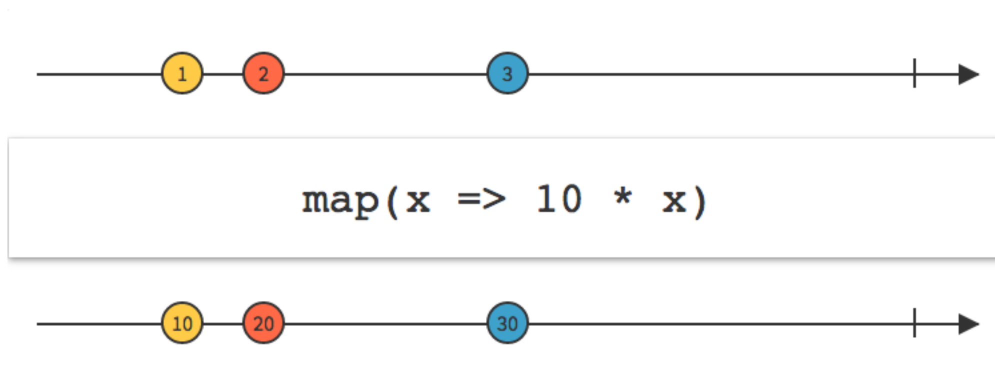


Figura 33: Map Operator

Concat Operator

Este operador concatena el output de múltiples *Observables* de manera que los convierte en uno sólo, con todos los ítems emitidos por el primer Observable y, después, los del segundo y así sucesivamente. Para ello, *Concat* se suscribe a cada *Observable* que haya recibido como parámetro y espera al *OnComplete* de cada uno para proceder con el siguiente. Una extensión muy útil es aplicar la función *.first()* justo después del *Concat*, de manera que obtendremos los resultados del primer *Observable* que emita ítems, esto resulta de gran utilidad. Podemos ver como ejemplo la figura 34.

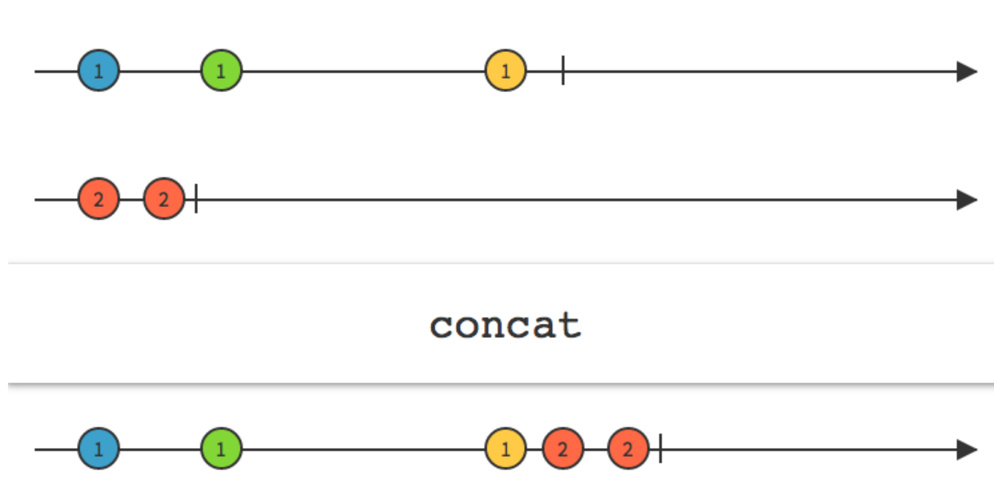


Figura 34: Concat Operator

DoOnNext Operator

Este operador recibe como parámetro cada ítem emitido por el *Observable*, permite aplicarle la función deseada y retornara el *Oservable* que ha recibido, sin modificarlo. Podemos ver como ejemplo la figura 35.

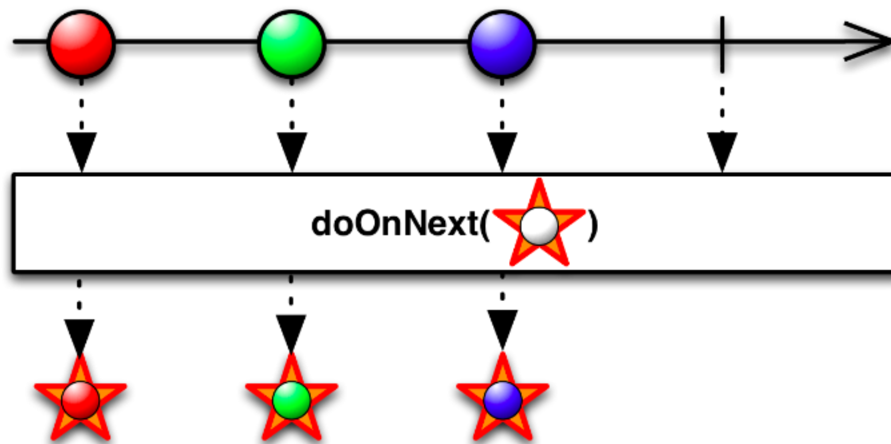


Figura 35: DoOnNext Operator

DoOnCompleted Operator

Este *Operator* estará a la escucha del evento *OnCompleted* por parte del *Observable* y nos permitirá aplicar una función justo en ese momento. Podemos ver como ejemplo la figura 36.

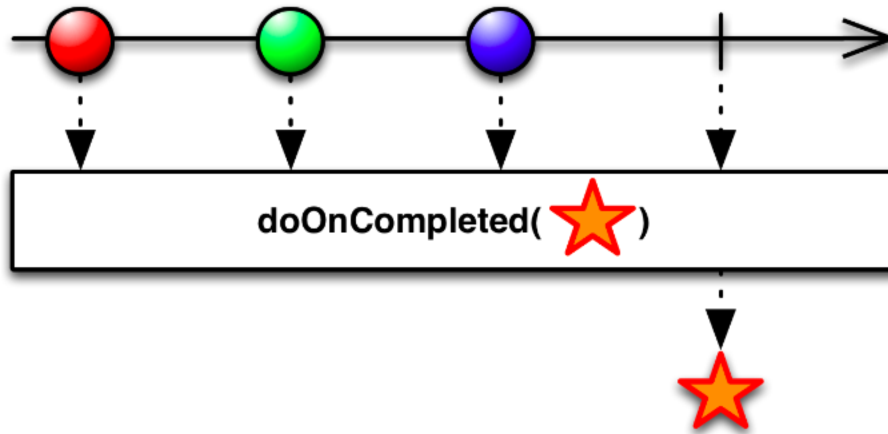


Figura 36: DoOnCompleted Operator

FlatMap Operator

Este *Operator* nos permite recibir como parámetro un *Observable*, aplicarle una función y retornar los *Observables* resultantes. FlatMap hace uso de *Merge* para mezclar estos *Observables* resultantes en un sólo *Stream*. Este *Operator* es muy útil cuando, por ejemplo, recibimos *Observables* que como atributos tienen algún *Observable* y queremos recuperar y aplicar funciones sobre éstos y, que además, el *Observable*resultante sea la mezcla de todos éstos. Podemos ver como ejemplo la figura 37.

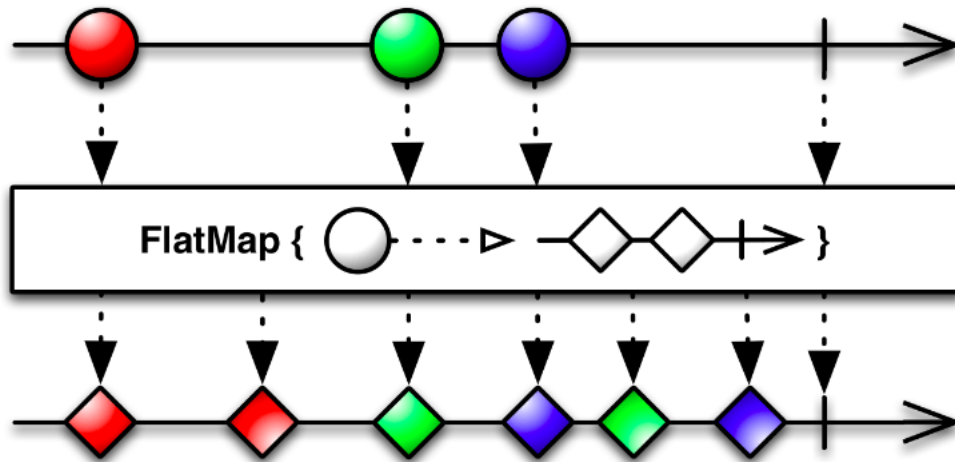


Figura 37: FlatMap Operator

Merge Operator

Este operador combina los items emitidos por múltiples *Observables* de manera que el resultante es sólo uno, el cual contiene todos los items emitidos por el resto. La diferencia con el operador *Concat* es que éste sí intercala los items obtenidos en orden de llegada. Podemos ver como ejemplo la figura 38.

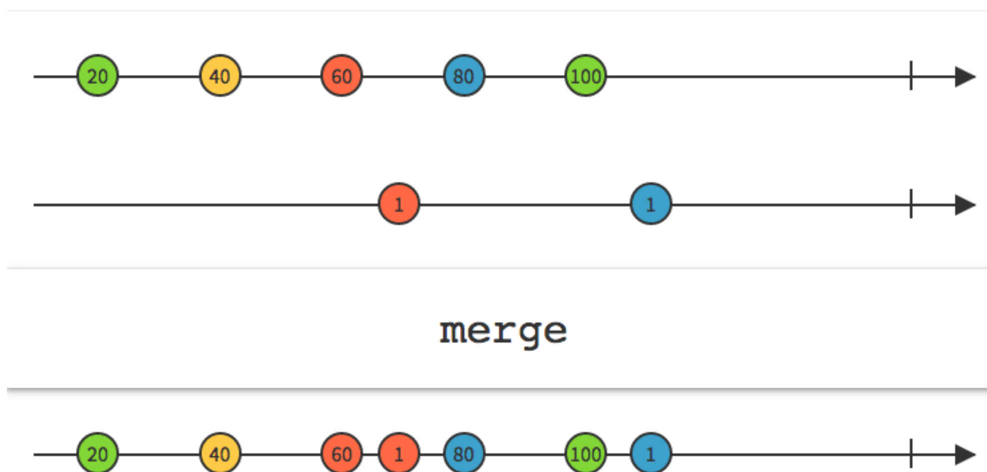


Figura 38: Merge Operator

5.2.1.5 Schedulers

Por defecto, estos *Observables* si no indicamos lo contrario, se ejecutarán en el *thread* principal, para evitar esto, disponemos de los *Schedulers* [16]. Gracias a los *Schedulers* podemos introducir *multithreading* en nuestra cascada de *Operators* y podemos elegir en qué *Scheduler* harán las operaciones; incluso algunos *Operators*

pueden recibir el *Scheduler* deseado como parámetro, de manera que este *Operator* realizará la acción deseada en ese *Scheduler*.

Por defecto, la cadena de *Operators* sobre un *Observable* se realizará el mismo *thread* en el que el *Subscriber* se haya suscrito. Para cambiar este comportamiento disponemos del *Operador* SubscribeOn, especificando un *Scheduler* diferente donde el *Observable* debe trabajar. A su vez, disponemos del operador ObserveOn, que especifica el *Scheduler* en el cual el *Observable* deberá emitir las notificaciones a sus *Subscribers*.

RxJava dispone de varios *Schedulers* predefinidos.

- *Schedulers.computation()*: *Scheduler* adecuado para trabajo computacional, el número de threads bajo este *Scheduler* será el número de procesadores que disponga el dispositivo sobre el cual trabajamos.
- *Schedulers.from(executor)*: Nos permite especificar nuestro propio *Executor* para que sea utilizado como *Scheduler*.
- *Schedulers.immediate()*: Trabajara directamente en el *thread* desde el que se llame a los Operators.
- *Schedulers.io()*: El *Scheduler* para realizar operaciones *Input Output* como llamadas al Servidor. Este *Scheduler* posee una *thread-pool* que irá creciendo tanto como se necesite. Por ello no se recomienda para operaciones computacionales, para ello se utilizará el *Scheduler.computation()* donde está limitado el número de *threads*.
- *Schedulers.newThread()*: Para cada *Operator* creará un nuevo *Thread*.
- *Schedulers.trampoline()*: Los operadores se irán concatenando en una *Queue* y se irán ejecutando por el orden de *Queue*.

5.2.2. Use Cases

5.2.2.1 Definición

Para la aplicación en Android de el concepto *Use Case*, tendremos en cuenta que aunque reciban el mismo nombre, los Casos de Uso normalmente definidos en el diseño de Software y los Casos de Uso definidos en *Clean Architecture* no son lo mismo, aunque en muchos casos coincidan. E.g. “El usuario podrá compartir la aplicación mediante un botón Share”, este requisito, representaría un Caso de Uso en Diseño de Software tradicional, pero no un Caso de Uso en *Clean Architecture*.

Un *Use Case* para el desarrollo en Android vendrá definido por toda aquella acción que realice el Usuario o Aplicación que implique un acceso a datos, ya sea a BBDD, Servidor, o a otra fuente.

Estos Casos de Uso son los encargados de unir el módulo *presentation* con el módulo dominio, y de realizar las acciones necesarias para delegar y retornar los

datos demandados. Representarán las funcionalidades de nuestra aplicación y se podrá aplicar lógica de negocio en éstos.

La función de éstos es clave para el buen desarrollo de *Clean Architecture*, ya que son los encargados de manejar el flujo de datos de la aplicación y los que hacen de soporte a los *Presenters* para llevar a cabo las funcionalidades de la aplicación.

5.2.2.2 Ventajas y Desventajas

Si realizamos un buen análisis de estos Casos de Uso permiten, en primer lugar, una abstracción hacia la capa superior (módulo *presentation*) de manera que asignando nombres descriptivos a estos Casos de Uso podemos saber qué acción realizará sin saber su implementación. En segundo lugar, restringen el acceso a datos haciéndolo sólo posible si utilizamos éstos Casos de Uso, evitando código duplicado y accesos a datos desde varios lugares. Y, por último, nos permiten extender las funcionalidades de manera sencilla, añadiendo casos de uso e implementándolos, dando lugar a un código entendible y extensible.

5.2.3. Patrón Repository

5.2.3.1 Definición

El patrón *Repository* trabaja conjuntamente con el Patrón *DataStore*, éste último se explicará más adelante. El patrón *Repository* [18], mostrado en la figura 39, está relacionado con el acceso a datos y nos permite tener una abstracción de la implementación de acceso a datos en nuestras aplicaciones, de modo que nuestra lógica de negocio no conozca ni esté acoplada a la fuente de datos. En pocas palabras, esto quiere decir que el repositorio actúa como un intermediario entre nuestra lógica de negocio y nuestra lógica de acceso a datos para que se centralice en un sólo punto y, de esta forma, se logre evitar redundancia de código. Como se ha mencionado antes, al ser una abstracción del acceso a datos nos permite desacoplar y testear de una forma más sencilla nuestro código, ya que al estar desacoplado podemos generar pruebas unitarias con mayor facilidad. Adicionalmente, al estar centralizado en un solo punto de acceso, podemos reutilizar nuestra lógica de acceso a datos desde cualquier punto de la aplicación.

Uno de los escenarios donde más se suele usar el patrón repositorio, es cuando tenemos múltiples fuentes de datos y, normalmente, ese es nuestro escenario. Como aplicación *Mobile* tendremos como fuente de datos el Servidor, y a su vez nuestra propia base de datos. En otros casos, si necesitamos obtener información de un *wearable*, de un dispositivo introducido en vehículo, etcétera, tendremos aún más fuentes de datos.

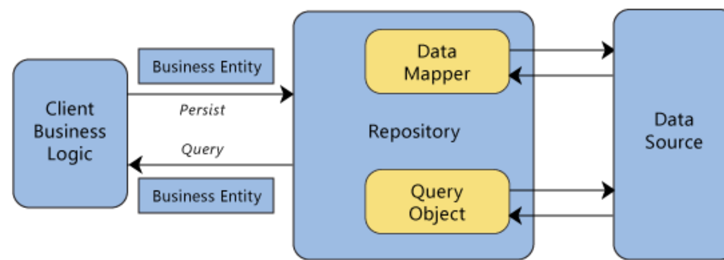


Figura 39: Abstracción del Patrón Repository
Fuente: msdn.microsoft.com

5.2.3.2 Ventajas

Utilizando este patrón evitaremos:

- Código duplicado.
- Muchos errores de programación.
- Acceso "*libre*" a nuestros datos.
- Centralizaremos el acceso a éstos.
- Nos permitirá crear tests Unitarios de manera mucho más fácil gracias al poco acoplamiento.

5.2.4. Objetos Business Object

Los objetos de Negocio, en inglés *Business Object* y, en adelante BO, permiten a nuestro Software desacoplar el modelo de datos utilizado por el Servidor, y el modelo de Datos utilizado por la BBDD, de manera que hacemos que sea independiente de ambos, teniendo un modelo de datos propio para nuestra aplicación que no se verá afectado por el Servidor, por la BBDD o por la *UI* (recordamos que en la UI se utilizan objetos Model). Estos objetos BO contendrán todos los datos necesario para poder realizar las funcionalidades presentes en nuestra aplicación.

5.2.5. Aplicación del módulo dominio en TweetStatt

Casos de Uso

Todos los Casos de Uso aquí listados extienden de la clase creada para ahorrarnos líneas de código que se repiten cuando utilizamos RxJava. Como se ha dicho cuando trabajamos con ReactiveX lo importante es paralelizar el trabajo que realizamos siempre en hilos, para ello teníamos como herramientas el *SubscribeOn* y el *ObserveOn*.

```
public void execute(Subscriber UseCaseSubscriber) {
   this.subscription = this.buildUseCaseObservable()
      .subscribeOn(Schedulers.from(threadExecutor))
      .observeOn(postExecutionThread.getScheduler())
      .subscribe(UseCaseSubscriber);
}
```

Casos de Uso detectados y desarrollados en TweettStatt:

- GetHashtagsStatisticsUseCase: Este Caso de Uso trabajará con la interfície Repository StatisticsRepository, desconociendo su implementación pero pudiendo acceder al método *getStatisticsHashtags();* La implementación de este se la proporcionará el *Presenter* que haga uso de este *UseCase*.

- GetHomeTimelineStatisticsUseCase: De la misma manera que el anterior, trabaja con StatisticsRepository para poder acceder al método *getStatisticsHomeTimeline();* Recuperará la lista de estadísticas generadas en base al HomeTimeline en forma de Objetos BO StatisticBo.

- GetSearchStatisticsUseCase: Trabajará con StatisticsRepository para poder acceder al método *getSearchStatistics();* Recuperará la lista de estadísticas generadas en base a búsquedas en forma de Objetos BO StatisticBo.

- GetTimelineStatisticsUseCase: Trabajará con StatisticsRepository para poder acceder al método *getTimelineStatistics();* Recuperará la lista de estadísticas generadas en base al Timeline en forma de Objetos BO StatisticBo.

- GetStatisticByIdUseCase: Trabajará con StatisticsRepository para poder acceder al método *getStatisticById(int id)*; Recuperará la estadística generada en base a la ID que se le envíe en forma de Objetos BO StatisticBo.

- PersistStatisticUseCase: Trabajará con StatisticsRepository, pudiendo acceder al método *persistStatistic(StatisticBo bo);* enviándole un StatisticBo para ser persistido en BBDD.

- GetHashtagsUseCase: Trabajará con TweetsRepository, pudiendo acceder al método *getHashtags(boolean refresh);* obteniendo una lista de HashtagsBos, que representará la lista de *Hashtags TrendingTopic* en ese momento.

- GetHomeTimelineUseCase: Trabajará con TweetsRepository pudiendo acceder al método *getTweetsHometimeline(String userName, boolean refresh);* obteniendo una lista de TweetsBo, que será la lista de Tweets de ese Usuario.

- GetSearchUseCase: Trabajará con TweetsRepository para poder acceder al método *getTweetsBySearch(String query, boolean refresh);* obteniendo una lista de TweetsBo que será la lista obtenida al realizar esa búsqueda.

- GetTimelineUseCase: Trabajará con TweetsRepository para poder acceder al método *getTweetsTimeline(String userName, boolean refresh);* obteniendo una lista de TweetsBo que será la lista de *Tweets* de las personas a las que sigue ese Usuario.

- GetTweetsByHashtagUseCase: Trabajará con TweetsRepository para poder acceder al método *getTweetsByHashtag(String hashtag, boolean refresh);* obteniendo una lista de TweetBo que será la lista de *Tweets* relacionados con el *hashtag* demandado.

- GetUserDataUseCase: Trabajará con UserRepository para poder acceder al método *getUserData(boolean hasConnection);* que retornará un objeto UserBo con los datos del Usuario logeado.

Objetos de Negocio

Para este módulo dominio y en concreto la aplicación TweetStat se han utilizado los objetos de Negocio siguientes:

- UserBo
- TweetBo
- HashtagBo
- StatisticBo

Estos contendrán todos los datos necesarios para realizar toda la funcionalidad de nuestra aplicación, descartando los datos innecesarios que provengan de Servidor, o los datos que necesitemos exclusivamente para su guardado en BBDD. Para que quede más clara la diferencia entre *Model* y *Business Object*, se propone el ejemplo de los Objetos *TweetBo* (negocio) y *TweetModel(UI).*

```
TweetBo.java
/**
 * Business Object that represents a Tweet in the domain layer.
 */
@Data //Anotación para el plugin de Android Studio ``Lombok" que genera
    los Setters y Getters sin ensuciar el código.
public class TweetBo {
    private long id;
    private String title;
    private String description;
    private String thumbnailUrl;
    private double latitude;
    private double longitude;
    private String country;
    private String city;
    private String createdAt;
    private String location;
}
```

TweetModel.java

```
/**
 * Model Object that represents a Tweet in the presentation layer.
 */
@Data //Anotación para el plugin de Android Studio ``Lombok" que genera
    los Setters y Getters sin ensuciar el código.
public class TweetModel {
    private long id;
    private String tweet;
    private String thumbnailUrl;
}
```

Patrón Repository

Ya que TweetStat tendrá tres tipos de datos que obtener, Estadísticas, *Tweets* o Usuario, se ha realizado una separación en tres interfícies, una para cada tipo de datos, que serán implementadas en el módulo data. Estas Interfícies permitirán a los Casos de Uso el acceso a datos y continuar con el flujo de éstos.

Interfície TweetsRepository:

```
Observable<List<TweetBo>> getTweetsBySearch(String query, boolean
    refresh);
Observable<List<TweetBo>> getTweetsByHashtag(String hashtag, boolean
    refresh);
Observable<List<TweetBo>> getTweetsHometimeline(String userName,
    boolean refresh);
Observable<List<TweetBo>> getTweetsTimeline(String userName, boolean
    refresh);
Observable<List<HashtagBo>> getHashtags(boolean refresh);
```

Interfície UserRepository:

```
Observable<UserBo> getUserData(boolean hasConnection);
```

Interfície StatisticsRepository:

```
Observable<List<StatisticBo>> getStatisticsTimeline();
Observable<List<StatisticBo>> getStatisticsHomeTimeline();
Observable<List<StatisticBo>> getStatisticsSearch();
Observable<List<StatisticBo>> getStatisticsHashtags();
Observable<StatisticBo> getStatisticById(long id);
Observable<Void> persistStatistic(StatisticBo bo);
```

5.2.6. Estructura del Módulo Domain

Finalmente, nuestro módulo *domain* queda en la estructura del proyecto en Android Studio como se puede observar en la figura 40.

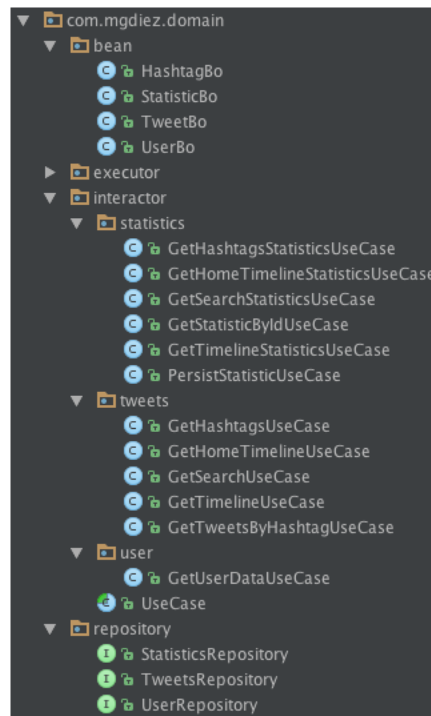


Figura 40: Estructura a nivel Código Fuente del módulo Dominio

5.3. Módulo data

El módulo *data* será el encargado de la conexión con los agentes externos Servidor, y Base de Datos, proporcionando a los casos de uso los datos necesarios, y desacoplando el modelo de datos propio del servidor de el nuestro. Hará lo mismo con nuestra base de datos.

5.3.1. Objetos Data Transfer Object

Los Objetos de Transferencia de Datos, en inglés *Data Transfer Objet* y, en adelante DTO por sus siglas en inglés, es un objeto que transporta datos entre procesos. La razón por la cual existen estos tipos de Objeto viene dada por la necesidad de comunicación entre cliente-servidor, en peticiones *GET* o *POST*, por ejemplo, se recibirán datos en el cliente o se introducirán datos en el Servidor, de manera que estos objetos DTO nos permiten mediante estructuras *JSON* enviar los datos o recibirlos de una manera mucho más eficiente que si tuviéramos que actualizar o recibir cada campo por separado. Por lo tanto, estos DTOs serán los encargados de viajar entre cliente-servidor o viceversa. Esta será su única función, objetos contenedores de datos, sin lógica de negocio (para ello están los BO) y no requieran de testeo.

Nos permite la separación entre el modelo de datos del servidor, y el modelo de datos del cliente.

5.3.2. Objetos Value Object

Los Objetos de Valores, en inglés *Value Object* y, en adelante VO, en *Clean Architecture* es el tipo de Objeto que será utilizado para la comunicación con la Base de Datos, siendo así del tipo que el *Framework* o tipo de BBDD necesite. También serán como los DTO, objetos sin lógica de negocio, serán simples contenedores que serán guardados y recuperados. La utilización de los VO nos permite, una vez más, desacoplar el modelo de datos del *Framework* o BBDD que utilicemos de nuestro modelo de datos haciendo, así, que nuestra aplicación sea independiente de éstos.

Como curiosidad, debemos saber que la primera edición de *"Core J2EE Design Patterns"* tuvo tanto éxito que el término *"Value Object"* se convirtió en sinónimo de DTO aún siendo esto era erróneo y, en cierta forma, se sigue manteniendo hasta la fecha, pero debemos saber que no es así, aunque en algunos libros o documentaciones se pueda encontrar mezclado. Tras estas definiciones podemos observar como DTO y VO no son lo mismo.

5.3.3. Patrón Event Bus

5.3.3.1 Introducción

Como se ha ido viendo a lo largo de este trabajo, el desarrollo de Software para Android, es una de las máximas expresiones en lo que a Programación Orientada a Eventos se refiere, ya que tenemos los eventos que genera el usuario, los eventos de entrada y salida de datos y los propios eventos de la plataforma sobre la cual trabajamos. Para solucionar los problemas que surgían de cara a tratar estos eventos, se utilizaba el concepto *Listener*; un *Listener*, en esencia, es una interfície, que nosotros como desarrolladores definíamos en la clase que iba a producir el evento, e implementábamos en la clase que debía recibirlo. El problema de estos *Listeners*, es que el número de eventos a implementar crecía exponencialmente con cada funcionalidad añadida a nuestra aplicación, de manera que resultaba tedioso lidiar con todas estas intefícies. Como hemos visto anteriormente, el uso de RxJava nos ayuda en gran medida a substituir estos eventos, pero, seguimos teniendo eventos que no podemos controlar con RxJava y es aquí donde entra en juego el concepto de Bus de Eventos.

5.3.3.2 Definición

Un Bus de Eventos [17], por definición, nos resultará muy familiar al concepto de programación reactiva, ya que a nivel de implementación se basa en el mismo Patrón *Observer*. En un Bus tendremos un objeto *"Bus"* al cual se podrán suscribir nuestro objetos *Subscriber* de manera que todo evento que el Bus emita será recibido por los objetos que se hayan suscrito. Recibe el nombre de Bus ya que el concepto es el mismo que la topología en Red de tipo Bus, mostrada en la figura 41.

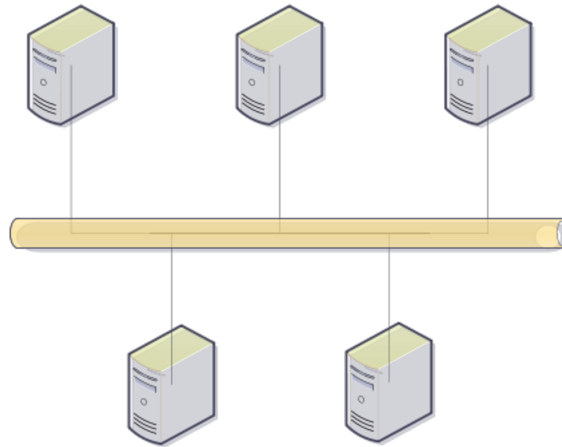


Figura 41: Topología Bus utilizada en Redes
Fuente: wikipedia.com

5.3.3.3 RxBus

Reactive posee un tipo de Objeto llamado *Subject*, un *Subject* es una especie de puente que está disponible en algunas implementaciones de *ReactiveX*, en el caso de RxJava existe y éste actúa a la vez como *Observer* y como un *Observable*. Gracias a este comportamiento híbrido, como *Observer* nos permite suscribirnos a eventos y como *Observable* nos permite emitir eventos, como se muestra en la figura 42. Veremos en la parte de aplicación como podemos aprovechar este objeto híbrido llamado Subject para poder implementar nuestro propio bus de eventos haciendo uso de *ReactiveX*.

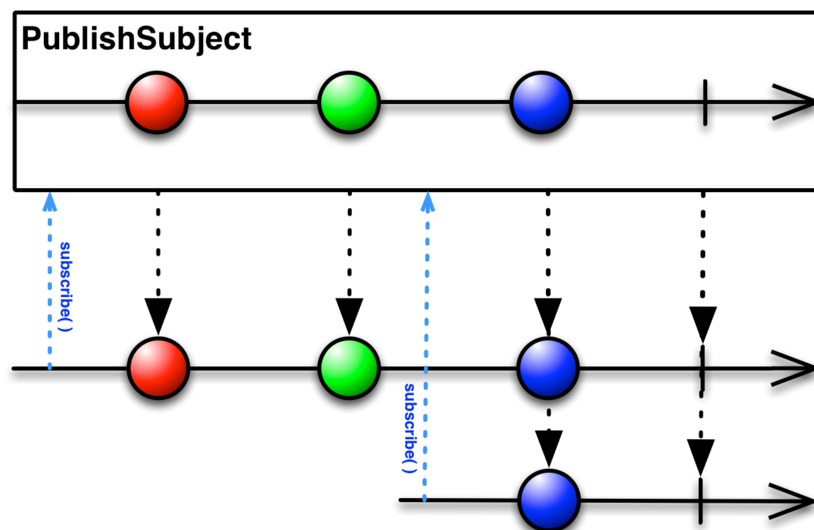


Figura 42: Funcionamiento de un Objeto tipo PublishSubject
Fuente: reactivex.io

5.3.3.4 Ventajas

La utilización de un Bus de Eventos nos permite un mejor uso y trato de los eventos, mejora el proyecto en escalabilidad, ya que en caso de que un nuevo objeto necesite estar a la escucha de determinado evento, sólo tendremos que suscribirlo al Bus y no implementar interfícies nuevas y, a su vez, mejora el rendimiento de la aplicación y la experiencia de usuario. Por ejemplo, reaccionando al momento cuando el usuario marca un icono que debe cambiar de color o realiza acciones que supondrían una recarga de datos, con la utilización de un Bus de Eventos eliminamos la necesidad de la recarga, ya que en todo momento sabremos que está haciendo el usuario.

5.3.4. Patrón DataStore

5.3.4.1 Definición

Como vimos con el Patrón *Repository*, éste trabajaba conjuntamente con los *DataStore*, las diversas implementaciones que nuestra Interfície *Repository* puede recibir, trabajarán con con los diferentes *DataStore* que posea nuestra aplicación. Estos *DataStore* implementarán una Interficie de manera que todos los *DataStore* que trabajen con un tipo de datos reciban los mismos métodos.

Nuestra interfície *DataStore*, normalmente recibe dos implementaciones, la que recibe los datos de Servidor y la que recibe los datos de BBDD, siempre ampliable a cada tipo de fuente de datos que tenga nuestra aplicación. Deberá haber, a su vez, poniendo el ejemplo de la aplicación que se desarrolla en este trabajo, cada tipo de datos que recibirá un *DataStore*, obteniendo en este caso TweetsDataStore, para todas las funciones necesarias ya sean para persistir o recuperar *Tweets*, StatisticsRepository para las funcionalidades relacionadas con las estadísitcas.

5.3.4.2 Patrón Factory DataStore

Si queremos mejorar el acoplamiento que habrá entre implementaciones de *Repository* y *Datastore* (ya que *Repository* se comunicará con *Datastore*), podemos aprovechar que hemos definido que cada *Datastore* será una interfície implementada por los diferentes tipos de datos e incorporar el Patrón *Factory* ya explicado en esta memoria, a los objetos de tipo *DataStore*. Este *DataStoreFactory* recibiría, por ejemplo, si el dispositivo dispone de conexión o no y retornará la implementación del *DataStore* que trabajará con el Servidor en caso de disponer, o con local en caso de estar offline. Según las necesidades de nuestro proyecto, nuestros requisitos para este *Factory* pueden aumentar y no sólo depender de si tiene el dispositivo conectividad o no, dando lugar a *Factories* demasiado grandes por lo que que puede resultar contraproducente utilizar este patrón con nuestros *DataStore*.

5.3.4.3 Ventajas

Utilizando el patrón *DataStore*, conseguiremos aislar el acceso a datos, sea cual sea la fuente a sólo las clases *DataStore*. De manera que tendremos centralizado qué clase está actuando sobre estas fuentes de datos y, a su vez, tendremos unas Interfícies para saber que métodos podemos utilizar para recibir datos, teniendo un software más mantenible y extendible.

5.3.5. Retrofit

5.3.5.1 Definición

En la actualidad la gran mayoría de aplicaciones necesitan trabajar con una API *Rest*. Por este motivo existen gran variedad de *Frameworks* en el mercado que nos ayudan con esta tarea. Retrofit [19] es un *Framework* desarollado por Square Inc. que podemos utilizar como cliente REST.

	One Discussion	Dashboard (7 requests)	25 Discussions
AsyncTask	941 ms	4,539 ms	13,957 ms
Volley	560 ms	2,202 ms	4,275 ms
Retrofit	312 ms	889 ms	1,059 ms

Figura 43: Comparativa Framework REST
Fuente: tempos21.com

En la comparativa mostrada en la figura 43 podemos observar como el claro ganador es *Retrofit*, frente al uso del cliente nativo *Android Http* y el uso de *AsyncTask* y frente al cliente desarrollado por Google, *Volley*. Por este motivo y por la gran cantidad de desarrolladores usando *Retrofit* se ha decidido utilizar ésta sobre las otras opciones.

Al tratarse de un cliente REST permite realizar operaciones HTTP:

- GET: Petición al Servidor en la cual esperaremos una respuesta con datos.
- POST: Petición al Servidor en la que esperaremos un OK/NOK.
- PUT: Funcionamiento igual que POST, pero deberemos tener en cuenta que PUT es idempotente, mientras que POST no lo es.
- DELETE: Petición de borrado, normalmente esperaremos OK/NOK.
- HEAD: Funcionamiento igual que GET pero esta respuesta no contrendrá body.

A su vez, nos permitirá un amplía configuración, desde escojer cómo parsear los objetos de llegada en caso de ser una petición GET (donde utilizaremos nuestros objetos DTO), hasta tener definidos como Headers los parámetros que necesitáramos.

Parsers

Retrofit trabaja con los siguientes parsers:

- Gson: com.squareup.retrofit2:converter-gson
- Jackson: com.squareup.retrofit2:converter-jackson
- Moshi: com.squareup.retrofit2:converter-moshi
- Protobuf: com.squareup.retrofit2:converter-protobuf
- Wire: com.squareup.retrofit2:converter-wire
- Simple XML: com.squareup.retrofit2:converter-simplexml

De los parsers listados, *Gson* y *Jackson* son los más utilizados. Para este proyecto se utilizará uno de ellos, veamos a continuación cómo rinden.

Como se puede ver en la tabla comparativa entre *Gson* y *Jackson*, en la figura 44, para objetos de poco tamaño *Gson* rinde mejor y con un amplio margen. Sin embargo, si el tamaño de datos aumenta, *Jackson* funciona mejor, así que la decisión vendrá según el tamaño de datos a parsear.

Para nuestra aplicación, TweetStatt, el objeto que más recibiremos serán los *Tweets*, objetos pequeños y por lo tanto se utilizará *Gson*.

	SIMPLE		COMPLEX	
	10 KB - 10 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	0.0127	0.0047	0.0098	0.0045
JSON-TO-MAP	0.0041	0.0019	0.0036	0.0022
JSON-TO-JAVA OBJECT	0.0025	0.0022	0.0031	0.0014
	10 KB - 50 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	0.0016	0.0012	0.0017	0.0009
JSON-TO-MAP	0.0009	0.0005	0.0009	0.0004
JSON-TO-JAVA OBJECT	0.0010	0.0006	0.0008	0.0005
	100 MB - 10 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	0.7931	1.3238	0.6420	1.1242
JSON-TO-MAP	2.0673	1.7297	2.2599	8.9618
JSON-TO-JAVA OBJECT	0.8787	1.1406	0.7519	1.3390
	100 MB - 50 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	0.5227	1.2624	0.4945	0.9668
JSON-TO-MAP	1.5713	1.5554	2.2547	8.8385
JSON-TO-JAVA OBJECT	0.8525	1.2020	0.7388	1.3133
	250 MB - 10 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	1.1921	3.3292	1.1989	2.5987
JSON-TO-MAP	2.8610	15.3583	2.1853	6.5023
JSON-TO-JAVA OBJECT	2.0399	3.6117	1.8959	3.0178
	250 MB - 50 rounds (seconds)			
Functionality	JACKSON	GSON	JACKSON	GSON
JAVA OBJECT-TO-JSON	1.1454	3.4003	1.0921	2.5494
JSON-TO-MAP	2.2306	14.3814	2.1587	6.7834
JSON-TO-JAVA OBJECT	1.8467	3.1354	1.9394	3.4411

Figura 44: Comparativa Gson y Jackson
Fuente: baeldung.com

Anotaciones Retrofit

Una vez definidos los métodos *HTTP* que *Retrofit* nos permite utilizar, veamos como indicar que tipo de petición estamos configurando.

- @GET("url")
- @POST("url")
- @PUT("url")
- @DELETE("url")
- @HEAD("url")

Cómo utilizarlo

Como primer paso para nuestro uso de *Retrofit*, crearemos nuestro objeto *Retrofit*, a continuación se muestra la manera más básica de crearlo, pudiendo añadir en este paso configuraciones que nosotros deseemos.

```
Retrofit retrofit = Retrofit.Builder()
   .baseUrl(ApiConstants.ENDPOINT)
   .client(httpClient)
   .addConverterFactory(GsonConverterFactory.create())
   .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
   .build();
```

Posteriormente necesitaremos declarar una interfície en la cual configuraremos los Servicios *REST* que vayamos a utilizar.

```
interface HashtagsApiService {
   @GET("/1.1/trends/place.json")
   void getHashtags(@Query("id") String id, Callback<List<HashtagDto>>
       cb);
}
```

Donde HashtagDto es el DTO al cual será parseada la respuesta del Servidor, por lo tanto deberá tener la estructura que necesita *Gson* (o el parseador que escojamos). Una vez tengamos estos dos pasos, asignaremos a nuestro objeto *Retrofit* la Interfície que debe utilizar.

```
retrofit.create(HashtagsApiService.class);
```

Una vez hecho esto, podremos realizar la llamada al Servidor realizando:

```
retrofit.getHashtags(id, new Callback<List<HashtagDto>>(){...});
```

¿Async o Sync?

Retrofit puede realizar las llamadas de manera síncrona o asíncrona, la manera que tiene *Retrofit* para discrepar entre una u otra es la siguiente:

- Asíncrona: Si nosotros como Objeto de retorno en la interfície utilizada (en el ejemplo HashtagsApiService) utilizamos *Void* y le damos como parámetro un *Callback*, *Retrofit* ejecutará la petición de manera asíncrona. También debemos saber, como es nuestro caso, que si trabajamos con RxJava, el objeto de retorno será un *Observable* de manera que aún no siendo *Void*, se ejecutará de manera asíncrona.
- Síncrona: *Retrofit* ejecutará la petición de manera síncrona cuando como Objeto de retorno se le defina cualquier tipo de Objeto que no sea *Void* o *Obser-*

vable.

¿Cuál deberemos utilizar? Nunca queremos bloquear el *thread* principal, por ello utilizaremos siempre la forma Asíncrona de *Retrofit*. De hecho, si utilizamos la forma Síncrona, en versiones Android 4.0 en adelante, esta incluida, obtendremos un error del tipo *NetworkOnMainThreadException*.

5.3.5.2 Ventajas

Utilizando *Retrofit* obtendremos una mejora de rendimiento frente a la manera nativa, como hemos podido ver en la tabla comparativa y, a su vez, podremos separar las peticiones de los diferentes servicios, podríamos tener TweetsService, con todas las peticiones relacionadas con los *Tweets*, etcétera. También, si no utilizamos RxJava, mediante un *Callback*, *Retrofit* ejecutará en un *thread* y en paralelo todas las llamadas sin tener que implementarlo nosotros.

5.3.6. Realm.io

5.3.6.1 Definición

Realm es un *Framework* para plataformas *Mobile* que tiene como objetivo substituir a las antiguas BBDD de Android nativas en *SQLite*, y realizar operaciones de *ORM (Object Relational Mapping)*.

Realm actualmente es utilizada por grandes empresas como Amazon, Google, Cisco, IBM, Intel y un largo etcétera. veamos qué tiene de especial.

Otras opciones

Empezaremos evaluando *Realm* frente a otras opciones disponibles en el mercado, para ello hacemos uso de la comparativa extraída de *realm.io* mostrada en las figuras 45, 46 y 47:

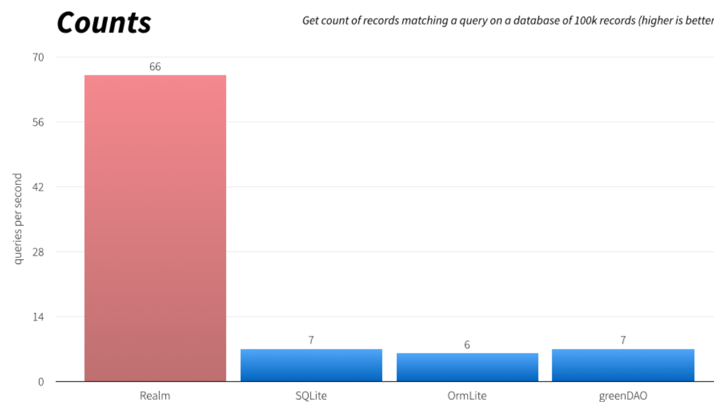


Figura 45: Comparativa Counts

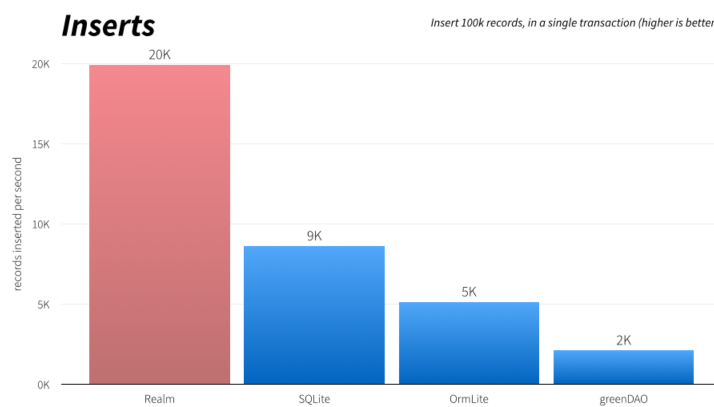


Figura 46: Comparativa Inserts

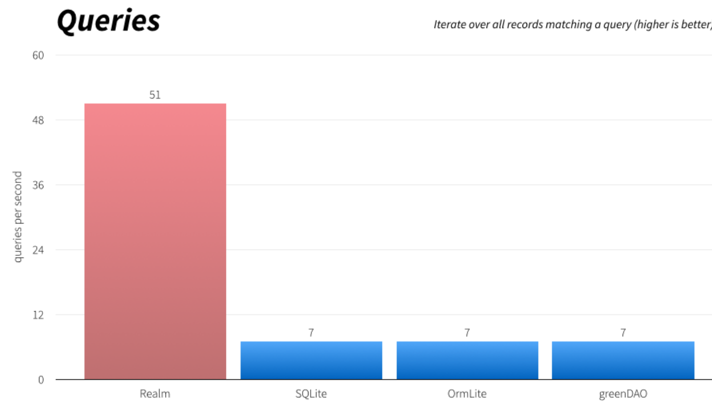


Figura 47: Comparativa Queries

Podemos ver como Realm resulta claro ganador, por lo que en este trabajo se utilizará Realm como Base de Datos.

¿Cómo lo hace?

Realm [20], a diferencia de sus competidores, no utiliza como base *SQLite*, ya que tiene su propio "motor", el cual no es *Open Source*. Además, de este motor se conoce que está escrito en C++, por lo que no tiene que ser interpretado por la *JVM* (*Java Virtual Machine*, en el caso de Android, *Dalvik* anteriormente y *ART* en la actualidad). Estos dos factores hacen que *Realm* obtenga tanta ventaja respecto a sus competidores.

5.3.6.2 Utilización de Realm

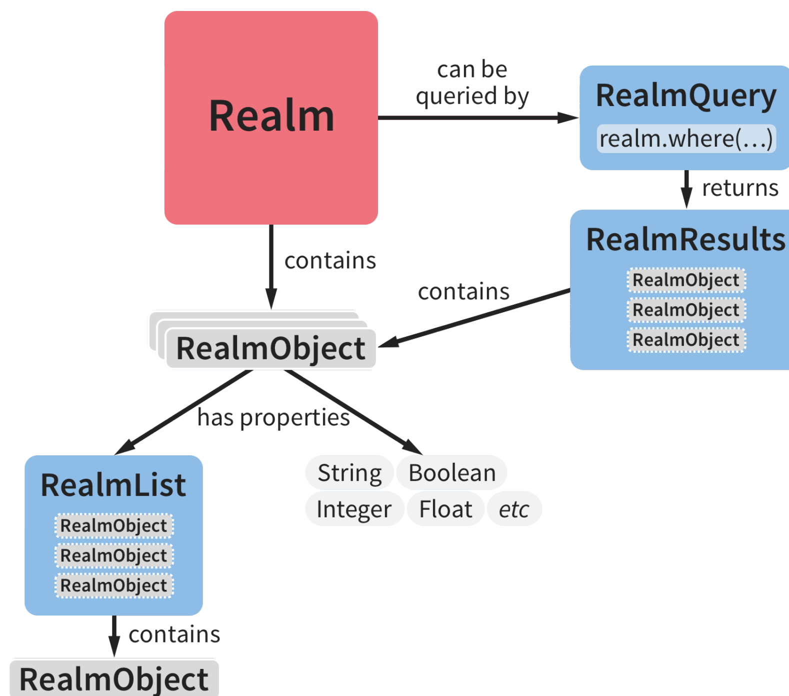


Figura 48: Overview de Realm
Fuente: realm.io

Objetos de Realm

Como vemos en el diagrama mostrado en la figura 48, aparecen varios tipos de Objetos que necesitamos conocer para hacer uso de *Realm*.

- RealmObject: Todo objeto que nosotros queramos persistir en la base de datos, deberá extender de *RealmObject*. ¿Porqué? *Realm* necesita añadir funcionalidades extra a nuestros objetos para poder realizar su trabajo. Los objetos que hagamos extender de *RealmObject*, serán nuestros VO explicados anteriormente. Estos VO deberán llevar la anotación de *Realm @RealmClass* y la anotación *@PrimaryKey* sobre el atributo que sea la *Primary Key*.
- RealmList: Este tipo de Objeto es el List de Java para *Realm*. Nuestras listas de Objetos deberán ser $RealmList< ObjetoVo >$.
- RealmQuery: Las *Querys* a BBDD en *Realm* son llamadas *RealmQuery*, y podremos realizarlas con la función *.where(ObjetoVo.class)* y añadiendo parámetros como *.findAll()* o *.equalTo()* haciendo las *querys* de un uso mucho más simple que el *SQL* base.
- RealmResults: Cuando hagamos una *RealmQuery* obtendremos *RealmResults*, que contendrán los ObjetosVo resultantes de dicha *query*.

Creación de Realm

Para crear nuestra base de datos *Realm*, deberemos:

Crear una configuración de *Realm*, en este caso la base.

```
RealmConfiguration realmConfiguration = new
   RealmConfiguration.Builder(this).build();
      Realm.setDefaultConfiguration(realmConfiguration);
```

Podremos acceder a ella realizando:

```
realm = Realm.getDefaultInstance();
```

Ya que *Realm* guardará la instancia haciendo uso del Patrón *Singleton* por lo que podremos acceder a ella desde cualquier punto de la aplicación.

Ejemplo extraído de la aplicación desarrollada para persistir datos:

```
Realm realm = RealmHelper.getInstance(context);
List<TweetVo> tweetVos = TweetVoMapper.toVo(tweets);

realm.beginTransaction();
realm.copyToRealmOrUpdate(tweetVos);
realm.commitTransaction();

realm.close();
```

Ejemplo extraído de la aplicación desarrollada para recuperar datos:

```
RealmResults<TweetVo> tweetVos =
   realm.where(TweetVo.class).equalTo(TweetVo.QUERY, search).findAll();
List<TweetBo> tweetBos = TweetVoMapper.toBo(tweetVos);

if (tweetBos != null && !tweetBos.isEmpty()) {
  return Observable.just(tweetBos);
}
return Observable.empty();
```

5.3.6.3 Best Practices

Async Queries

Los creadores de *Realm* aseguran que es suficientemente rápido como para ejecutar *querys* en el *thread* principal. Aún así, debemos saber cómo realizar consultas a BBDD de manera asíncrona. Tenemos dos opciones: utilizando RxJava y ejecutando esa transacción dentro del propio hilo que crea *ReactiveX*, o haciendo uso de la función *executeTransactionAsync()* que recibirá como parámetro un objeto

Transaction y dos *Callback*, uno en caso de *Succes* y otro en caso de *Error*, los últimos dos opcionales.

Android Lifecycle y Realm

Como se ha visto en el ejemplo de uso, nuestra instance de *Realm* se puede abrir y cerrar. Para un uso eficiente de *Realm* deberá iniciar y cerrar en los siguientes puntos:

- En la clase que extienda de *Application*, crearemos la instancia de *Realm* y la configuraremos de la forma que necesitemos.
- Recuperaremos la instancia en el *onCreate()* y en el *onResume()* en la *Activity* y *Fragment* que utilicemos como base y de las cuales extenderán el resto.
- Cerraremos la instancia en el *onStop()* y *onDestroy()* en la *Activity* y *Fragment* que utilicemos como base y de las cuales extenderán el resto.

Si utilizamos el patrón DataStore, la implementación que acceda la base de datos Realm deberá encargarse de abrir y cerrar la instancia en cada transacción.

Listas de Strings, Integers...

Como hemos visto, si queremos guardar en nuestra base de datos *Realm* una Lista de Objetos, esta lista será del tipo *RealmList< ObjetoRealm >*. Para guardar listas en *Realm* de datos primitivos como *Strings* o *Integers*, deberemos crear un objeto StringVo que en su interior contenga el objeto primitivo *String*, en este caso; obtendremos un *RealmList< StringVo >* que será apto para el uso por parte de *Realm*. Es un pequeño *"Workaround"* que se tiene que realizar hasta que *Realm* de soporte completo a este tipo de listas.

5.3.6.4 Ventajas

Realm es fácil de utilizar, como hemos visto es rápido y estable, los archivos .realm que crea se pueden traspasar a otras plataformas, por ejemplo, podríamos traspasar la BBDD creada en un dispositivo Android a un dispositivo iOS. Además, tiene funcionalidades añadidas como BBDD volátiles (en tiempo de ejecución, como una caché), proporciona encriptación de la BBDD, entre otras.

5.3.7. Fabric SDK

Fabric [25] es la plataforma de Desarrollo de Software creada por Twitter Inc. la cual ofrece varios servicios:

- Twitter SDK : Cliente para la comunicación conTwitter.

- Crashlytics : Crashes a tiempo real y estadísticas sobre estos.
- Digits: Nos ayuda a realizar un login con número de teléfono.
- MoPub: Alternativa a Google para monetizar nuestras aplicaciones.
- answers: Analíticas sobre nuestra aplicación.
- Cognito: Servicio Cloud para el guardado de datos.
- Stripe: Plataforma para el pago desde plataformas Mobile.
- AppSee: Estadísticas sobre cómo utilizan los usuarios tu aplicación.
- Mapbox: Nos permite customizar los mapas de Google para tener un estilo propio.
- Nuance: Librería para añadir Speech Recognition a nuestra aplicación.

Y desde hace muy poco, cuenta con un *plug-in* para los principales IDEs de desarrollo mobile, IntelliJ, Android Studio y XCode, lo que hace aún más fácil su instalación y la actualización de estos SDK's.

5.3.7.1 Funcionalidad

Con todas las funcionalidades que *Fabric* nos ofrece, resulta de gran utilidad conocer esta plataforma y su uso. Para ello, en este proyecto se utiliza el SDK de Twitter que *Fabric* nos ofrece. El cliente de Twitter de *Fabric* nos ofrece:

- Login con Twitter
- Un pequeño cliente de la *API Rest* de Twitter
- *Layouts* originales de la aplicación Twitter para visualizar *Tweets*.

5.3.8. Aplicación del módulo data en TweetStatt

Implementación de los Repositories

En este punto, implementaremos las interfícies definidas anteriormente y que recordamos, serán las utilizadas por nuestros casos de uso. Estas implementaciones deberán trabajar con los *DataStore* para continuar el flujo de datos y, además, nos permitirán realizar decisiones sobre que *DataStore* utilizar, algo realmente útil, como vamos a ver.

- StatisticsRepositoryImpl: La implementación encargada de pedir al *DataStore* correcto todos los datos relacionados con estadísitcas, ya sean las estadísticas generadas y guardadas o una estadística en concreto para volver a visualizarla.

- UserRepositoryImpl: La implementación encargada de retornar los datos del usuario que utiliza nuestra aplicación. En este caso, acudirá a Servidor si el dispositivo tiene conexión y en caso contrario a la base de datos.
- TweetsRepositoryImpl: La implementación que se encarga de pedir al *DataStore* correcto todos los datos relacionados con los *Tweets* y *Trending Topics*.

Para ver los beneficios que aporta el patrón *Repository* junto a *ReactiveX* se muestra a continuación el ejemplo de TweetsRepositoryImpl, en la función que recoje los *tweets* del *timeline* del usuario.

```
@Override
public Observable<List<TweetBo>> getTweetsHometimeline(String userName,
    boolean refresh) {
   CloudTweetDatastore cloudTweetDatastore = new
       CloudTweetDatastore(context);
   LocalTweetDatastore localTweetDatastore = new
       LocalTweetDatastore(context);

   if (NetworkUtil.isNetworkAvailable(context) && refresh) {
      return
          Observable.concat(localTweetDatastore.getTweetsHometimeline(userName),
         cloudTweetDatastore.getTweetsHometimeline(userName));
        } else {
      return
          Observable.concat(localTweetDatastore.getTweetsHometimeline(userName),
         cloudTweetDatastore.getTweetsHometimeline(userName))
         .first();
        }
}
```

Podemos observar como haciendo uso de los *Observables* podemos realizar varias acciones en una sola línea de código. Primero, se crean los objetos *DataStore* que implementan las interfícies de TweetDatastore, una que acude a la BBBDD y otra a Servidor. Posteriormente, se mira si el usuario ha forzado el *refresh* y si el dispositivo tiene conexión, si se cumple, se hace uso de la función *Observable.concat()* explicada en el apartado de RxJava y que nos permite retornar los *Tweets* que se reciben de Servidor con los que ya teníamos guardados en la base de datos y retornarlos en un sólo *Observable*. Si el dispositivo no tiene conexión o no se ha forzado el *refresh*, se utiliza la misma función pero añadiendo *.first()*, de esta manera conseguimos que, si el usuario tiene *Tweets* en base de datos, sólo se retornaran estos sin acceder a Servidor, y si no tiene ningún *tweet* en base de datos irá a servidor. Como se puede ver, gracias a *ReactiveX* podemos realizar esta lógica de negocio en apenas 4 líneas de código.

En el caso de la implementación de StatisticsRepositoryImpl, como siempre accederá a nuestra base de datos, simplemente hace de puente entre el caso de uso y nuestro *DataStore*, como se muestra a continuación poniendo el ejemplo de la recogida de estadísticas generadas desde una búsqueda.

```
@Override
public Observable<List<StatisticBo>> getStatisticsSearch() {
   return new LocalStatisticDatastore(context).getStatisticsSearch();
}
```

DataStores

Cada *Repository* trabajará con N *Datastore* según N fuentes de datos tengamos en nuestra aplicación. Estos *DataStore* implementarán una intefície que dependerá del tipo de dato con el que trabajen, por ello en nuestro proyecto tenemos las interfícies StatitisticsDatastore, TweetDatastore y UserDatastore; implementadas por CloudTweetDatastore, LocalTweetDatastore para la interfície TweetDatastore, CloudUserDatastore y LocalUserDatastore para la interfaz UserDatastore y por último LocalStatisticDatastore para StatisticDatastore.

Se explicará a continuación una función de CloudTweetDatastore y LocalTweetDatastore para el caso de recogida de los *tweets* propios de el usuario, el resto de *Datastore* se encuentran en el proyecto con los nombres anteriormente mencionados.

CloudTweetDatastore, función para recojer los *tweets* de un usuario:

```
@Override
public Observable<List<TweetBo>> getTweetsUsertimeline(String username){
   return twitterApiService.getTweetsUsertimeline(username,
       getLastModificationTweet(USER_TIMELINE))
      .doOnNext(new TweetsActionPersist(USER_TIMELINE))
      .map(TweetsDtoMapper::toBo);
}
```

En este caso, recibimos como parámetro el nombre del usuario que esta utilizando nuestra aplicación, y como CloudDatastore debemos realizar las acciones necesarias para pedir a Servidor los datos demandados y retornar objetos de negocio para su uso en capas superiores. Como podemos ver, se utilizan los operators *.doOnNext()* y *.map()*, explicados en el apartado de RxJava, para hacer un uso máximo de *ReactiveX*. En esta línea de código estamos realizando la llamada a la clase TwitterApiService (que se explicará a continuación) y le proporcionamos el nombre del usuario y la ID del último *tweet* que tenemos en base de datos para no recibir *Tweets* que ya tenemos; posteriormente con el operator *.doOnNext()* y nuestro objeto de tipo *Action1* llamado TweetsActionPersist, estamos consiguiendo que cada *Observable* que llegue sea guardado en base de datos y finalmente con nuestro operador *.map()* conseguimos que el *Observable* recibido se mapee a Objeto BO haciendo uso una vez más de las *Lambdas Expressions* de Java 8 con la función estática toBo de la clase TweetsDtoMapper. En resumen, realizamos la llamada a Servidor, guardamos en Base de Datos y mapeamos a Objeto de Negocio en una sola línea de código, todo gracias a *ReactiveX* y a concatenar operators.

Se han mencionado los objetos de tipo *ActionN*, donde N es el número de paráme-

tros que reciben. En nuestro caso se utiliza *Action1* ya que reciben la lista de *Tweets* obtenida del servidor. Se muestra a continuación a modo de ejemplo el guardado en base de datos de una lista de tweets.

```
private class TweetsActionPersist implements Action1<List<Tweet>> {
   private String type;
   public TweetsActionPersist(String type) {
           this.type = type;
       }

       @Override
       public void call(List<Tweet> tweets) {
           if (tweets != null && !tweets.isEmpty()) {
               saveLastModificationTweet(type, tweets.get(0).getId());
               Realm realm = RealmHelper.getInstance(context);
               List<TweetVo> tweetVos = TweetVoMapper.toVo(tweets);

               realm.beginTransaction();
               realm.copyToRealmOrUpdate(tweetVos);
               realm.commitTransaction();

               realm.close();
           }
       }
}
```

Como podemos ver, primero guardamos en *SharedPreferences* la ID del último *tweet*, posteriormente se recoje la instancia de *Realm*, se mapean estos *Tweets* a objetos de Tipo *Value Object* y, por último, abrimos una transacción, guardamos en BBDD y hacemos *commit* para finalmente cerrar la instancia de *Realm*.

Twitter Api Service

Para la comunicación con la API de Twitter se ha utilizado y extendido la clase TwitterApiClient que proporciona el SDK de Twitter creado por *Fabric*. Para la utilización de esta clase, deberemos crear una clase propia que extienda de la mencionada y al constructor pasarle la sesión activa del usuario *Twitter.getSessionManager().getActiveSession()*.

Como requiere nuestra aplicación, deberemos implementar las llamadas de:

- Los *tweets* del usuario
- El *timeline* del usuario
- Servicio de búsqueda
- *Hashtags* que son *trending topic*

- Datos del usuario

Al trabajar con el cliente de la API que proporciona *Fabric* y RxJava nos encontramos con varios problemas. En primer lugar, el cliente de *Fabric* trabaja con *Retrofit*, pero en su versión 1.0 y la versión de *Retrofit* que es compatible con *ReactiveX* es la 2.0. En consecuencia a esto, este cliente trabaja en sus llamadas con *Callbacks*, incompatibles con nuestra metodología *ReactiveX*, ya que si hacemos uso de *Retrofit* 2.0 es éste quien, al recibir como parámetro de retorno un *Observable*, realiza la llamada de forma asíncrona y se encarga de realizar los eventos *onNext(T Object)*, *onCompleted* y *onError*. En segundo lugar, veremos que el servicio de *Trending Topics* no está implementado en el SDK de *Fabric*, de manera que deberemos extender y añadir este servicio a su cliente y hacer uso a la vez de los objetos que ellos implementaron.

Ya que tenemos un *Callback* en la llamada, deberemos realizar nosotros manualmente la gestión de eventos *ReactiveX* haciendo uso de los eventos que proporciona el *Callback*. En primer lugar se implementaron estos servicios de manera que recibían el objeto *Subscriber* que creaba el *Presenter* desde la vista, y desde el *Callback* hacer referencia a ese *Subscriber* y nutrirlo de los eventos *onNext(T Object)* con los datos y *onCompleted()*. Es una solución que funciona, pero como hemos visto en la Implementación de los *Repositories*, la manera de sacar más provecho a *ReactiveX* es trabajando con *Observables* y ejecutando *Operators* sobre éstos, algo incompatible con la primera solución implementada ya que al recibir un *Subscriber* como parámetro y darle manualmente los eventos, no estaba retornando los datos en forma de *Observable* para poder trabajar con ellos.

Como segunda solución, se intentó crear el flujo de datos creando un *Observable* en el momento de la recepción de datos para aprovechar *ReactiveX*; al tener que retornar un *Observable* desde nuestra clase TwitterApiService y el *Callback* ser asíncrono se producía que, el *Observable* que retornábamos estaba vacío y este *Observable* vacío llegaba hasta el *Subscriber* de manera que los datos que posteriormente llegaban en el *subscriber* se perdían.

Finalmente, se descubrió la función *Observable.create()*, que recibe como parámetro un *Subscriber* (que será el *Subscriber* que nosotros asociemos a ese *Observable* en la ejecución del Caso de Uso) de manera que podemos avisar directamente a nuestro *Subscriber*, y a su vez, retornar como *Observable* esos datos para poder realizar las acciones vistas anteriormente.

E.g. Función que retorna un *Observable* de la lista de *Tweets* del *Timeline* de el usuario.

```
public Observable<List<Tweet>> getTweetsHometimeline(Long lastId) {
   return Observable.create(subscriber ->
       twitterApiClient.getStatusesService().homeTimeline(200,
      lastId, null, null, null, null, null, new Callback<List<Tweet>>() {
         @Override
                     public void success(Result<List<Tweet>> result) {
                              subscriber.onNext(result.data);
                              subscriber.onCompleted();
                     }

                     @Override
                     public void failure(TwitterException e) {
                              subscriber.onError(e);
                     }
                }
        ));
}
```

Como se puede ver, *Observable.create()* nos proporciona el *Subscriber* (en la función mostrada se utilizan las *Lambda Expressions* de Java 8) de manera que podemos retornar ese *Observable* y gestionar los eventos *onNext(T Object)*, *onCompleted()* y *onError()* desde el *Callback* de *Fabric*. En este caso se realiza el *onNext* y posteriormente el *onCompleted*, ya que Twitter retorna toda la lista de *Tweets* de esta manera y, una vez recibamos esa lista, nuestro stream habrá finalizado. Al empezar este trabajo la API retornaba los Tweets de uno en uno, por lo que la función era diferente, pero posteriormente fue modificada.

Las llamadas a estos Servicios de *Fabric* permiten los parámetros que Twitter permite y que podemos encontrar en su documentación. En el ejemplo mostrado nos encontramos con los siguientes parámetros, todos opcionales, por orden:

- count: Número de *Tweets* que recibimos, máximo de 200 y 20 por defecto. (*Integer*)
- since_id: Sólo retornará los *Tweets* con una ID superior a la específicada, si no existe, retornará los últimos. (*Long*)
- max_id: Sólo retornará los *Tweets* con una ID inferior a la especificada, si no existe, retornará los úlitmos. (*Long*)
- trim_user: Si es *true*, retornará con cada *Tweet* un Objeto *User* con la información del autor. (*Boolean*)
- exclude_replies: Si es *true*, no recibiremos los *Tweets* que sean una respuesta a otro. (*Boolean*)

- contributor_details: Si es *true*, recibiremos el nombre del autor del *Tweet*. Por defecto solo recibimos la ID de éste. (*Boolean*)
- include_entities: Si es *true*, no recibiremos información de localización del *Tweet*. Por defecto es *false*. (*Boolean*)

Sólo haremos uso del *count*, para recibir el máximo permitido y del *since_id*, para recibir sólo aquellos *Tweets* que el usuario no haya descargado ya.

Para la implementación del servicio de *Trending Topics* se hizo uso de *Retrofit 1.0*, obligado por el cliente de *Fabric*. Se crea la interifície que contendrá las llamadas (como se ha explicado en el apartado de *Retrofit*)

```
interface HashtagsApiService {
   @GET("/1.1/trends/place.json")
   void getHashtags(@Query("id") String id, Callback<List<HashtagDto>>
       cb);
}
```

El servicio de *Trending Topics* está alojado en */1.1/trends/place.json* y recibe como parámetro la ID del país sobre el cual se quiere recibir la información y el *Callback* necesario para que esta acción se realice de manera asícrona y el cliente de *Fabric* la acepte como extensión. Este *Callback* recibe una lista de HashtagsDtos, un objeto propio, con los campos que necesitamos y que se obtienen de la documentación de la API de Twitter y se parsean con *Gson*. HashtagsDtos como objeto propio y no del SDK de *Fabric*, deberá implementar la interfície Identifiable para que ésta lo acepte.

HashtagsDto contendrá una Collection de TrendsList, un objeto propio.

Contenido de HashtagsDto para la trendsList.

```
@SerializedName("trends")
public final Collection<TrendsList> trendsList;
```

Contenido de TrendsList. Podemos observar que la información que requiere *Gson* para poder parsear se la debemos anotar con @SerializedName(”Nombre de el campo en el JSON que envía la API”):

```
@SerializedName("tweet_volume")
public final Integer nTweets;
@SerializedName("events")
public final String events;
@SerializedName("name")
public final String name;
@SerializedName("promoted_content")
public final String promotedContent;
@SerializedName("query")
public final String queryString;
@SerializedName("url")
```

```java
public final String url;
```

Una vez creada la interfície que contiene nuestra llamada, y los objetos necesarios para parsear y recibir los datos de la API, deberemos nutrir a *Retrofit* de esta Interfície para poder realizar la llamada.

```java
public HashtagsApiService getHashtagsApiService() {
   return getService(HashtagsApiService.class);
}
```

getService es un método presente en la clase TwitterApiClient de Fabric que, a su vez, hace uso de *Retrofit*. Una vez hecho esto ya podremos realizar la llamada de la siguiente forma:

```java
getHashtagsApiService().getHashtags("23424950", new
    Callback<List<HashtagDto>>() {...}
```

Donde la ID es la *WOEID (Where On Earth ID)* de Yahoo. Este sistema permite asociar una ID a diferentes Países, como por ejemplo 23424950 para España, 12578034 para Catalunya, 22454274 para UK, etcétera.

El resto de llamadas se pueden encontrar en TwitterApiService.java.

DTOs y VOs

Como se ha explicado previamente, los Objetos DTO contendrán la información tal cual llega de Servidor, y los Objetos VO contendrán los datos que se guardarán en Base de Datos formateados como nuestro *Framework* necesite. En nuestro caso, al trabajar con el cliente de *Fabric*, recibiremos los datos en los Objetos de *Fabric*, de manera que podemos aprovechar estos objetos a modo de DTO, lo mismo para los objetos de tipo Usuario.

Objetos VO utilizados:

- TweetVo
- StatisticVo
- UserVo
- HashtagVo
- IntegerVo
- StringVo

Objetos DTO propios:

- HashtagDto

- TrendsListDto

Todos los VO y DTO tendrán asociados sus respectivos *Mappers* para realizar las transformaciones necesarias y delegar, si fuera necesario, lógica en ellos en el caso de los VO para su posterior guardado. Se pueden encontrar en el proyecto bajo los nombres HashtagDtoMapper, TweetsDtoMapper, UserDtoMapper en el caso de los objetos que serán mapeados a Bo y HashtagVoMapper, StatisticVoMapper, TweetVoMapper, UserVoMapper que realizarán el mapeo DTO a VO y VO a BO.

RxBus

Como se ha explicado, RxJava posee un objeto llamado *Subject* con la capacidad de ser *Observable* y Observer al mismo tiempo. Para ello, deberemos crear un objeto propio, accesible desde todos los puntos en los que se puedan enviar eventos y necesitemos recibirlos, y que pueda enviarlos.

Como necesitamos que sea accesible y de única instancia utilizaremos el patrón *Singleton*.

```
private static RxBus RXBUS_INSTANCE;
public static RxBus getInstance() {
   if (RXBUS_INSTANCE == null) {
      RXBUS_INSTANCE = new RxBus();
   }
        return RXBUS_INSTANCE;
}
```

Esto lo podríamos substituir haciendo uso de *Dagger2* en el *ApplicationModule* quedando así

```
@Provides
@Singleton
RxBus provideRxBus(){
   return new RxBus();
}
```

Y pudiendo realizar un @Inject donde lo necesitemos.

Una vez tenemos el patrón, deberemos crear el objeto tipo *Subject* como atributo de nuestra clase RxBus.

```
private final Subject<Object, Object> bus = new
    SerializedSubject<>(PublishSubject.create());
```

Dotamos a nuestro RxBus del método necesario para enviar un evento.

```
public void send(Object o) {
   bus.onNext(o);
}
```

Y el método necesario para poder suscribirse al RxBus.

```
public Observable<Object> toObservable() {
   return bus;
}
```

Y ya está implementado nuestro RxBus, sólo quedaría suscribirse a él para recibir eventos, y utilizarlo para enviarlos. Para enviar un evento solo tendremos que crear un Objeto significativo de ese evento, en el cual podríamos añadir atributos si así lo necesitáramos y enviarlo de la siguiente manera.

```
rxBus.send(new EventoDeterminado());
```

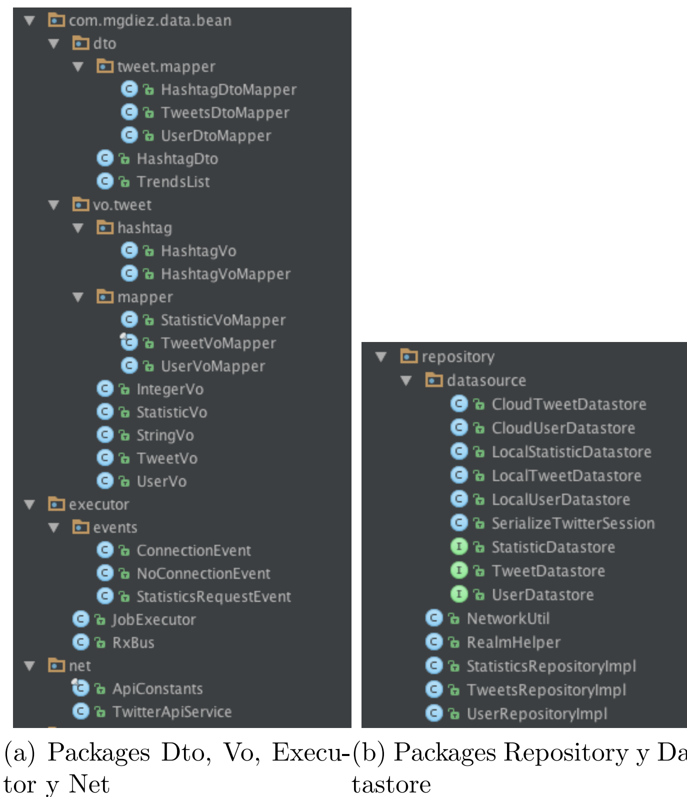
Y para recibir eventos lo haremos de la siguiente manera.

```
subscriptions.add(rxBus.toObservable().subscribe(new Action1<Object>() {
   @Override
   public void call(Object o) {
      if (o instanceof EventoDeterminado) {
         //Evento determinado ha ocurrido.
               }
               if (o instanceof EventoDeterminadoDos) {
         //Evento Determinado Dos ha ocurrido.
      }
   }
}));
```

En el proyecto se ha hecho uso del RxBus para notificar los diversos *Fragments* y *Activities* de los eventos producidos por la conexión del dispositivo móvil para mostrar un *Snackbar* si no tiene conexión o retirarlo cuando cuando retorne, información obtenida gracias a un *Receiver* de Android. Haciendo uso del objeto *Subject* podemos conseguir un Bus de Eventos de manera sencilla.

5.3.9. Estructura del Módulo Data

Finalmente, nuestro módulo *data* queda en la estructura del proyecto en Android Studio como se puede observar en la figura 49.



(a) Packages Dto, Vo, Executor y Net

(b) Packages Repository y Datastore

Figura 49: Estructura a nivel Código Fuente del módulo Data

6. Conclusiones y trabajo futuro

6.1. Conclusiones

El proyecto tenía unos objetivos muy ambiciosos, crear y aplicar la mejor arquitectura posible con los mejores frameworks disponibles para Android. Tras el planteamiento inicial de las tecnologías a aplicar, se lograron todas tal y como se ha explicado en esta memoria a falta de testing unitario y de instrumentación. Este último no dio tiempo a implementarlo dado el elevado gasto de tiempo en investigación de el resto de tecnologías, las curvas de aprendizaje de algunas de ellas eran bastante altas (e.g. *Dagger2* y *RxJava*) y los problemas surgidos por la limitación de la API y el *Framework* de *Fabric*. Aun así, estoy más que satisfecho del trabajo realizado, de lo aprendido y de lo aplicado a lo largo de este tiempo; se ha construido una arquitectura que cumple los patrones SOLID, con un elevado grado de escalabilidad y muy mantenible, siendo así, una de las mejores opciones cuando nos enfrentemos a la creación desde cero de una aplicación tanto para un futuro lector como para enterno profesional.

En cuanto a la funcionalidad de la aplicación, tiene la necesaria para poder cumplir los objetivos marcados a nivel de arquitectura y tecnologías, aún encontrándonos con limitaciones relacionadas con los datos sobre los *tweets* que Twitter ofrece, de manera que los objetivos iniciales de cara a generar un mayor número de estadísticas

se vieron mermados, pero sin afectar a los objetivos principales de este trabajo. Es una buena aplicación sobre la que se puede continuar trabajando una vez finalizado este trabajo, con mucha perspectiva de mejora de funcionalidades y ampliaciones, con un público bastante extenso (e.g. *Community Managers* que realizan búsquedas sobre su empresa o simples usuarios de Twitter).

6.2. Ampliaciones

Como ampliaciones por la parte de Arquitectura y/o implementación, se podrá extender este proyecto aplicando toda la parte de *Testing* que le falta, haciendo uso de herramientas como *Roboelectrics*, *Espresso* y *Mockito* para aprovechar otro beneficio que *Clean Architecture* nos ofrece, que es el poder Testear de manera Unitaria de forma mucho más sencilla gracias al poco acoplamiento entre clases y módulos, y realizando Test de Instrumentación Android, gracias a que la vista está completamente desacoplada de lógica. Además, en un futuro, se podrá substituir el patrón MVP por el patrón MVVM en el módulo app.

Por la parte de funcionalidades de la aplicación, tenemos un gran margen de ampliación y trabajo. La API de Twitter está bastante limitada en cuanto a lo que tasas de llamadas y cantidades de *Tweets* que podemos obtener, de manera que resultaría muy bueno para la aplicación la implementación de un *Service* que trabajara cuando el usuario estuviera conectado a WiFi y que fuera pidiendo tweets a la API de manera automática, consiguiendo así una gran base de datos sin la necesidad de que el usuario utilizara la aplicación.

También se podría ampliar la funcionalidad de la aplicación añadiendo un seguimiento a esos *Tweets*, teniendo N estadísticas generadas sobre un mismo tema, una búsqueda por ejemplo, realizar un estudio de cómo avanza esa temática, si hay más gente hablando de ello o menos, como cambian los lugares en los que se *Twittea* sobre ello, etcétera. También podríamos añadir un componente social permitiendo al usuario compartir esas estadísticas generadas.

7. Anexo

7.1. Instalación y ejecución de la entrega

Para la ejecución del Código Fuente entregado junto a esta memoria se necesitará como mínimo Android Studio 1.5, recomendado 2.0 en adelante, y los entornos Java 1.7 y Java 1.8 (para hacer uso de las Lambdas Expressions) y el plugin de Android Studio llamado *Lombok*.

- Android Studio - https://developer.android.com/studio/index.html
- Lombok - https://projectlombok.org/setup/android.html

- Java 8 - http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- Java 7 - http://www.oracle.com/technetwork/es/java/javase/downloads/jdk7-downloads-1880260.html

Y por último un dispositivo con Android API Level igual o superior a 16 (Android 4.1 en adelante). Cuando dispongamos de estos elementos, podremos importar de manera directa en Android Studio la carpeta que contiene el código fuente. Una vez hayamos importado el código fuente a Android Studio, deberemos hacer un *Build* para que se generen los ficheros necesarios (e.g. Como se ha explicado, *Dagger2* crea los archivos en tiempo de compilación) Además, para facilitar la instalación de la aplicación, se proporciona el archivo instalable en sistema operativo Android con extensión *.apk* en la carpeta *src*, junto a la carpeta llamada *tweetStatt* que contiene todo el proyecto. Para la instalación del archivo, deberemos traspasarlo al dispositivo Android y ejecutarlo. Para la ejecución de TweetStatt se necesita la aplicación de Twitter instalada en el dispositivo, ya que hace uso de esta para el *Login*, sin ella no funcionará.

7.2. Términos y conceptos

Symbian: Sistema Operativo Mobile de Nokia nacido en el 1997 que fue utilizado por empresas como Sony, Samsung, LG, Motorola y que se dejó de dar soporte en el 2013 ante el poco uso de este.

SCRUM: Método de desarrollo Agile, que establece una serie de ceremonias y buenas prácticas de cara a un desarrollo de Software eficiente.

Sprint: Unidad de tiempo utilizada en la metodología SCRUM para definir etapas en las que se planifican tareas a realizar por el equipo.

Roboelectrics: Framework utilizado para los test de Instrumentación, similar a Espresso, pero de terceros.

Espresso: Framework que nos permite realizar pruebas de Instrumentación Android y así detectar problemas en el funcionamiento de la UI de nuestra aplicación. Propiedad de Google.

Mockito: Framework que nos permite falsear objetos Android como pueden ser el Context para poder realizar Test Unitarios en partes del código que dependan de este tipo de objetos.

Android Service: Componentes de una aplicación que no necesitan de Interfície, y trabajan en segundo plano.

AsyncTask: Objeto proporcionado por Android para realizar acciones en background de manera que el hilo principal no quede bloqueado.

Memory Leaks: Error en programación que ocurre cuando un bloque de memoria no es liberado pero tampoco utilizado.

Stack: Estructura de Datos en la que se accede a los datos guardados de manera LIFO; último en entrar, primero en salir.

Queue: Estructura de Datos en la que se accede a los datos guardados de manera FIFO; primero en entrar, primero en salir.

Wearable: Aquellos dispositivos que por definición, se llevan encima, en la ropa, debajo, muñecas, etcétera.

JSON: Objeto de texto ligero para el intercambio de datos. Acrónimo de JavaScript Object Notation.

REST: Protocolo Cliente-Servidor sin Estado en el que cada petición lleva la información necesaria para ser procesada y tiene unas operaciones definidas como GET, PUT y DELETE.

SQLite: Sistema de Gestión de Bases de Datos Relacionales, ofrecida por Android como opción Nativa para el guardado de datos.

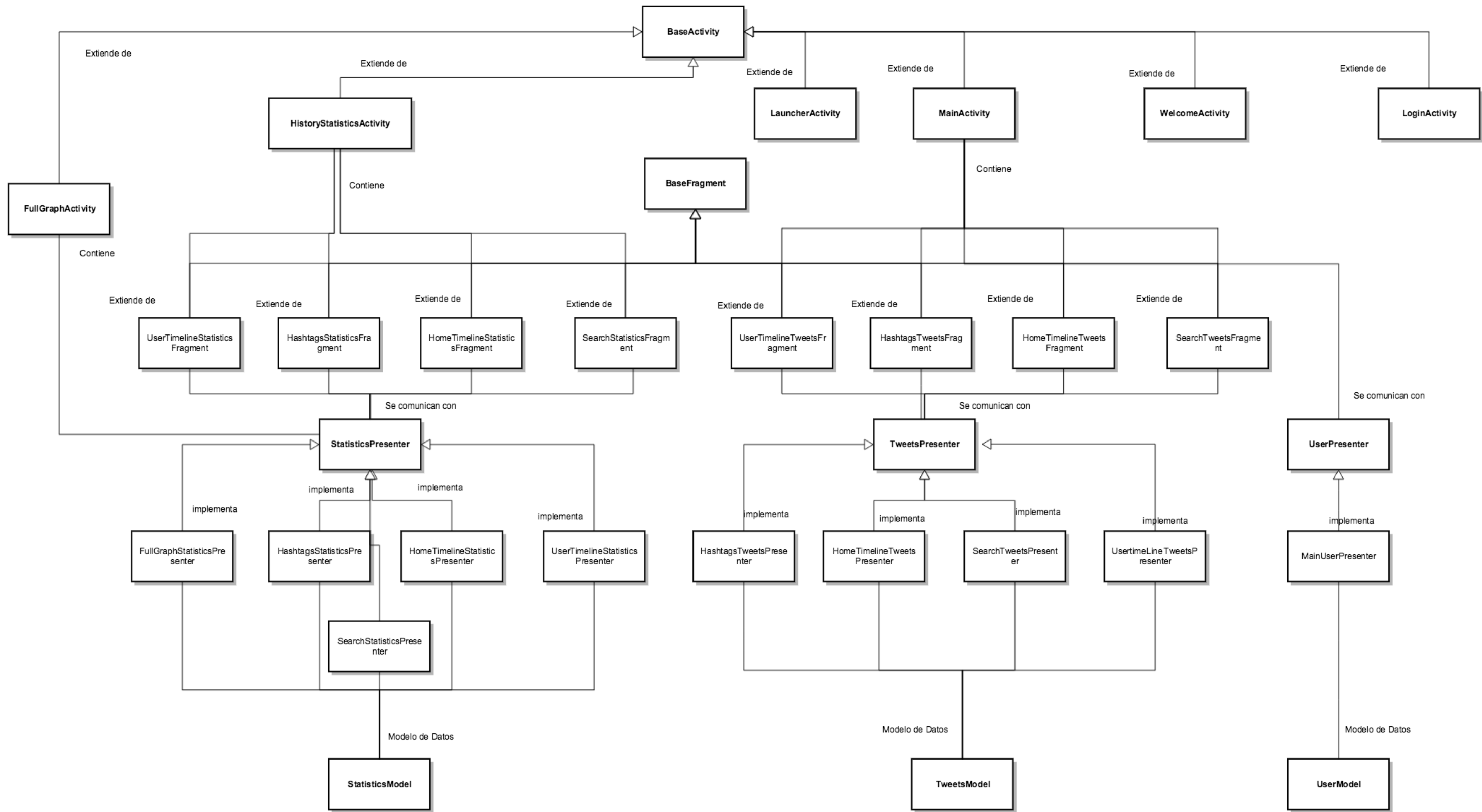
ORM: Acrónimo de *Object Relational Mapping*, es una técnica de programación utilizado para convertir los objetos utilizados en Programación Orientada a Objetos a una base de Datos Relacional.

Shared Preferences: Espacio proporcionado por Android para guardar datos de tipo primitivo útiles para la aplicación que permanecen guardados hasta que la aplicación es desinstalada.

Lambdas Expressions: Expresiones que permiten simplificar la escritura de código y simplificar su lectura, añadidas a Java en su versión 8, tales como :: o -¿.

Receiver: Es un componente ofrecido por Android de manera nativa cuya función es estar a la escucha y avisarnos de eventos que sucedan en el dispotivio móvil, por ejemplo, nivel de batería, cambio en la conectividad del dispositivo, etcétera.

7.2.0.1 Diagrama de clases módulo Presentation



BaseActivity
Extiende de
Extiende de
HistoryStatisticsActivity
Extiende de
LauncherActivity
Extiende de
MainActivity
Extiende de
WelcomeActivity
Extiende de
LoginActivity
FullGraphActivity
Contiene
BaseFragment
Contiene
Contiene
Extiende de
UserTimelineStatistics Fragment
Extiende de
HashtagsStatisticsFragment
Extiende de
HomeTimelineStatisticsFragment
Extiende de
SearchStatisticsFragment
Extiende de
UserTimelineTweetsFragment
Extiende de
HashtagsTweetsFragment
Extiende de
HomeTimelineTweets Fragment
Extiende de
SearchTweetsFragment
Se comunican con
StatisticsPresenter
Se comunican con
TweetsPresenter
Se comunican con
UserPresenter
implementa
implementa
implementa
implementa
FullGraphStatisticsPresenter
HashtagsStatisticsPresenter
HomeTimelineStatisticsPresenter
UserTimelineStatistics Presenter
SearchStatisticsPresenter
implementa
implementa
implementa
implementa
HashtagsTweetsPresenter
HomeTimelineTweets Presenter
SearchTweetsPresenter
UsertimeLineTweetsPresenter
implementa
MainUserPresenter
Modelo de Datos
StatisticsModel
Modelo de Datos
TweetsModel
Modelo de Datos
UserModel

Referencias

[1] Historia de Android
https://es.wikipedia.org/wiki/Android

[2] Android Developer Documentation. (Inglés)
https://developer.android.com

[3] Robert C. Martin
Agile Software Development, Principles, Patterns, and Practices (Alan Apt Series). (Inglés) 20 de Noviembre del 2002.

[4] Robert C. Martin *Clean Coder, The A Code of Conduct for Professional Programmers (Robert C. Martin Series)*. (Inglés) 26 de Mayo del 2011.

[5] Robert C. Martin *Clean code: A Handbook of Agile Software Craftsmanship.* (Inglés) 1 de Junio de 2008

[6] Clean Coders. (Inglés)
https://cleancoders.com

[7] Model View Presenter en Android
https://erikcaffrey.github.io/2015/11/03/mvp/

[8] Data Binding en Android. (Inglés)
https://developer.android.com/topic/libraries/data-binding/index.html

[9] ButterKnife Documentation. (Inglés)
http://jakewharton.github.io/butterknife/

[10] Picasso Documentation. (Inglés)
http://square.github.io/picasso/

[11] Inversion of Control. (Inglés)
https://en.wikipedia.org/wiki/Inversion_of_control

[12] Sergi Castillo, Google Developers Group *Dagger2 Workshop*. (Inglés) 7 de Mayo del 2016.

[13] Android Documentation Layouts. (Inglés)
https://developer.android.com/guide/topics/ui/declaring-layout.html

[14] Introduction to RxJava for Android. (Inglés)
http://www.philosophicalhacker.com/2015/06/16/introduction-to-rxjava-for-androi

[15] Iván Morguillo *RxJava Essentials*. (Inglés) 27 de Mayo del 2015.

[16] ReactiveX. (Inglés)
http://reactivex.io

[17] Event Bus Explained. (Inglés)
`https://github.com/google/guava/wiki/EventBusExplained`

[18] Repository Pattern. (Inglés)
`https://msdn.microsoft.com/en-us/library/ff649690.aspx`

[19] Retrofit Documentation. (Inglés)
`http://square.github.io/retrofit/`

[20] Realm.io. (Inglés)
`http://realm.io`

[21] Stack Overflow. (Inglés)
`http://stackoverflow.com`

[22] Retrofit 2.0 Introduction. (Inglés)
`https://inthecheesefactory.com/blog/retrofit-2.0/en`

[23] Lee Campbell *Introduction to Rx*. (Inglés) 2012

[24] RxJava as event bus, the right way (Inglés)
`https://lorentzos.com/rxjava-as-event-bus-the-right-way`

[25] Fabric Documentation (Inglés)
`https://docs.fabric.io/android/fabric/overview.html`