RIC

Creación de una guía para la utilización de la arquitectura *Clean Architecture* en aplicaciones Android con lenguaje de programación Java, utilizando como modelo la implementación de un prototipo de monitoreo deportivo para un equipo de *ultimate*

Clean Architecture and how this architecture is implemented in the of the developing an application on Android using the Java programming language

José Cruz Torres[1] & Euclides Samaniego González[2*]

[1]Licenciatura en Ingeniería de Sistemas y Computación – Facultad de Ingeniería de Sistemas Computacionales – Universidad Tecnológica de Panamá, [2]Grupo de Investigación en Inteligencia Computacional – GIICOM-Facultad de Ingeniería de sistemas Computacionales – Universidad Tecnológica de Panamá

Resumen En este documento se examinan los fallos que presentan las prácticas de desarrollo actual a la hora de escribir aplicaciones que sean sostenibles en un mundo de constante cambio, así como la importancia de esto. Se analizan los distintos patrones de diseño utilizados para la implementación de la arquitectura, además de los principios fundamentales necesarios generar código sostenible. Se definen los principios SOLID, así como la explicación, justificación y uso de cada una de sus partes. Se presenta cómo estos principios son utilizados para promover la mantenibilidad y la adaptabilidad del código a lo largo del tiempo. Se estudian los antecedentes del desarrollo móvil y su progresión con el paso del tiempo, así como las plataformas más comerciales y las tendencias que existen para estas plataformas. Se describen los antecedentes de la arquitectura *CLEAN Architecture* y cómo se implementa esta arquitectura en el proceso de desarrollo de una aplicación en Android utilizando el lenguaje de programación Java. Se realiza un análisis de los componentes del *framework* de Android, el uso de cada uno de estos componentes, como son integrados dentro de la aplicación y cuáles son las mejores prácticas de desarrollo en cuanto al uso de estos componentes. El resultado de esta implementación es un prototipo funcional que puede ser utilizado como base para la confección de otras aplicaciones de cualquier índole.

Palabras claves Java, *Clean Architecture, Android,* diseño de *software.*

Abstract This document aims to examine the errors that the current development practice have when is time to write applications that are compatible in a world of constant change, and the importance of this. We analyze the different design patterns used for the implementation of the architecture, in addition to the fundamental principles necessary to generate maintainable code. We define the SOLID principles, their explanation, justification and each one of its parts. It shows how these principles are used to promote the maintainability and adaptability of the code over time. We study the backgrounds of the current mobile development and its progression over time, the most commercials platforms and the tendencies in each platform. The document describe the background of the CLEAN Architecture and how this architecture is implemented in the of the developing an application on Android using the Java programming language. We analyze the components of the Android framework, the use of each of these components, how they are integrated inside the application and which are the best development practices for the usage of these components. The result of this implementation is a functional prototype that can be used for the creation of other applications of any kind.

Keywords Java, Clean Architecture, Android, software design.

*** Corresponding author:** euclides.samanifego@utp.ac.pa

1. Introducción

A la hora de desarrollar aplicaciones en Android existe una documentación exhaustiva de cómo funciona el *framework* de Android, sin embargo no existe una guía que especifique cómo se debe estructurar el proyecto, no solo para completarlo a corto plazo, sino que también permite darle mantenimiento a largo plazo.

La arquitectura que se busca implementar se llama *"Clean Architecture"* [1] y fue propuesta por el ingeniero y desarrollador Robert C. Martin.

A través de esta arquitectura se busca crear sistemas que sean independientes de un *framework,* que pueden ser probadas, (*Testing*), independiente de la implementación, abstraer la interfaz gráfica y que sea independiente de la base de datos e independiente de cualquier agente externo.

Este trabajo busca crear una guía que permita a los desarrolladores poder generar aplicaciones que sean escalables y fáciles de mantener.

El libro base de este documento es *"Clean code: a handbook of agile software craftsmanship* (2009)" de Robert Martin [2].

En el desarrollo de este proyecto se explican los antecedentes de la tecnología móvil, las ofertas que existen en el mercado así como las ventajas que ofrece este ecosistema, además de la descripción de los grandes bloques que componen a una aplicación en Android.

De igual forma, se mencionan los conceptos principales, las reglas y la teoría que justifican el funcionamiento de la arquitectura *Clean Architecture* y cómo esta arquitectura se puede ajustar en un entorno de Android, se definen cada uno de sus componentes y la razón de su existencia. Además se discutirán los patrones usados en cada una de las capas de la arquitectura, cuáles son sus beneficios y su implementación.

2. Android

2.1 Antecedentes

La historia de la programación tiene sus orígenes en la Inglaterra del siglo XIX, con Charles Babbage, que siendo un matemático brillante y una de las mentes más originales de su época, creó la primera máquina computarizada llamada "Máquina Analítica", que sirvió como precedente para las computadoras que conocemos hoy en día [3].

La tecnología siguió mejorando y surgió lo que se conoce como el Microprocesador, que permitió que las computadoras evolucionaran a un tamaño más reducido, lo que permite a los fabricantes desarrollar computadoras de manera más fácil y económica. Las computadoras personales se popularizaron y fueron abarcando más mercado, solucionaban de manera rápida y eficiente las necesidades de las empresas, organizaban los datos y agilizaban sus operaciones, pero estos dispositivos tenían una debilidad: no eran portables.

Los teléfonos celulares eran máquinas portables que servían para realizar y recibir llamadas, eran el medio de comunicación principal de las empresas, pero tenían muy poco poder computacional, aprovechando el tamaño que tenían estos dispositivos se visionó lo que serían los *smartphones* de hoy en día.

2.2 Historia

Android es una plataforma *Open Source* [4] diseñada para dispositivos móviles. Se encuentra en celulares, tabletas, televisores, carros, relojes y consolas.

Se encuentra respaldada por Google y pertenece a la *Open Handset Alliance* [5]. El objetivo de esta alianza es acelerar la innovación en dispositivos móviles y ofrecer a los consumidores una rica, menos costosa y mejor experiencia móvil.

2.3 Bloques principales

A la hora de desarrollar aplicaciones se deben dividir la aplicación en bloques, como si se tratara de piezas de un rompecabezas que al juntarlas generan una figura.

Al empezar es una buena práctica segmentar la aplicación de arriba hacia abajo, pensando primero en términos de pantallas,

características e interacciones entre ellas, luego en especificaciones.

Para mejorar la interpretación de los bloques es bueno imaginarlos como líneas y círculos, esto permite visualizar con facilidad cuál es el flujo de la aplicación, te permite ver la aplicación como un todo, como los componentes se van uniendo unos con otros y todo comienza a tener sentido.

2.3.1 *Activity*

Un *Activity* representa una pantalla que ve el usuario, una sola a la vez. Una aplicación contiene más de un *Activity* y puede moverse adelante y atrás entre ellos. Esta representa la parte más visual de la aplicación, junto con los *Fragments*, *BroadcastReceivers y ContentProviders* viven dentro de un hilo principal llamado *UI thread* o hilo de la interfaz gráfica.

2.3.2 *Intents*

Los *Intents* son mensajes asíncronos que permiten a los componentes de la aplicación pedir una funcionalidad de otro componente en Android.

Permiten interactuar con componentes de la misma aplicación así como componentes de otras aplicaciones. Por ejemplo, un *Activity* puede iniciar un proceso externo como tomar una foto.

2.3.3 *Services*

Los *Services* son procesos que corren en el fondo de la aplicación Background, y no tienen ningún tipo de componente para la interacción con el usuario. Pueden realizar las mismas acciones que los *Activity,* pero sin ningún tipo de interfaz gráfica.

2.3.4 *Content Providers*

Los *Content Providers* son interfaces para compartir información entre aplicaciones. Por defecto, cada aplicación corre en su propio entorno, *Sandbox,* así que toda la información que pertenece a la aplicación se encuentra aislada de otras aplicaciones del sistema.

2.3.5 *Broadcast Receivers*

Los *Broadcast Receivers* son implementaciones de Android de un sistema global del mecanismo *publish/subscribe* [6], o de manera más precisa el patrón Observer [7]. El receptor es sencillamente un código dormido que se activa una vez que ocurre el evento al que está subscrito.

2.3.6 *Fragments*

Los *Fragments* son objetos controladores que los *Activity* pueden usar para ejecutar una tarea. Esta tarea usualmente es manejar la interfaz del usuario. La interfaz puede ser una pantalla completa o parte de esta.

2.3.7 *Application Context*

Todos los componentes previos, *Activity, Fragment, Services, Intents, Content Providers* y *Broadcast Receivers*, cuando los combinas sirven para generar una aplicación.

Pero es más preciso decir que viven bajo el mismo *Application Context.*

Application Context se refiere al entorno de la aplicación y a los procesos dentro de los cuales los componentes están corriendo. Esto permite a las aplicaciones compartir información y recursos entre los bloques que la conforman.

Un *Application Context* es creado cuando el primer componente de la aplicación es instanciado, independientemente si el componente es un *Activity,* un *Service* o cualquier otro componente. El *Application Context* vive mientras la aplicación viva.

Por esta razón es independiente del ciclo de vida de los *Activity* o de los *Fragment.* Cabe resaltar que todas las clases que provienen de los *Activity* o *Fragment* son subclases de la clase *Context* as que poseen métodos para extraer el *Context* de la aplicación [8].

2.4 Recursos estáticos

2.4.1 *Drawable*

Un *Drawable* es un recurso general para representar gráficos que pueden ser representados.

Cada *Drawable* es guardado dentro de la carpeta res/*drawable*. Usualmente se guardan *bitmaps* para diferentes resoluciones [9].

2.4.2 *Layout*

El elemento básico para generar una vista es un objeto *View* que es creado por la clase *View* y ocupa un área rectangular en la pantalla responsable de dibujar y manejar las interacciones con el usuario y los eventos que esto genera.

2.4.3 *Menu*

Los *Menu* son usados como componentes de interfaces en muchos tipos de aplicaciones.

Para proveer una experiencia de usuario familiar y consistente, se debe usar el API de Menú que ofrece Android, que presenta las acciones y otras opciones en los *Activity*.

2.4.4 *MipMap*

El recurso *MipMap* es diferente a los otros tipos de recursos. Este recurso solamente se usa para guardar los íconos del *Launcher* de la aplicación, es relativamente nuevo, surgió en la versión 1.1 de Android Studio en febrero del 2015 [10].

2.4.5 *Values*

Los recursos *Values* son valores simples que contienen valores, estos valores son accedidos de manera frecuente y permiten ajustar la aplicación con mayor facilidad sin necesidad de modificar el código. Estos *Values* contienen, *strings, integers* y *colors*.

2.5 *App Manifest*

Toda aplicación en Android necesita un archivo AndroidManifest.xml, específicamente tiene que ser ese nombre, en la raíz del proyecto. Este archivo provee información esencial a Android sobre cómo está armada la aplicación, información que el sistema necesita antes de correr la aplicación. [11]

El AndroidManifest.xml cumple lo siguiente:

- Le da nombre al paquete de Java de la aplicación. El nombre del paquete sirve como identificador único de la aplicación.
- Describe los componentes de la aplicación, que incluyen a los *Activity, Services, Broadcast, Receivers y Content Providers* que componen la aplicación. También nombra las clases que implementan cada uno de los componentes y publica sus capacidades. Estas declaraciones informan a Android de los componentes y las condiciones en las que pueden ser lanzados.
- Determina el proceso que hospeda a los componentes de la aplicación.
- Declara los permisos que la aplicación tiene que tener acceso, para proteger las partes de la API de Android así como las interacciones con otras aplicaciones.
- También la instrumentación de las clases para poder perfilar la aplicación mientras se está ejecutando. Estas declaraciones solo están presentes cuando la aplicación está siendo desarrollada y eliminadas cuando la aplicación está publicada.
- Declara la versión mínima de Android que la aplicación requiere.
- Lista las librerías a las que la aplicación debe estar ligada.

3. *CLEAN Architecture*

3.1 Antecedentes

Cuando se están realizando sistemas, al pasar del tiempo el sistema va incrementando de tamaño y de complejidad, los problemas de diseño van más allá que los algoritmos y la estructura de datos para la computación.

Los diseños y sus especificaciones en general, generan otro tipo de problema: protocolos de comunicación, sincronización, acceso de datos, asignación de funcionalidades, distribución física, escalabilidad y rendimiento, etc.

Esto ocurre al nivel de diseño de la arquitectura de *software*. Es importante reconocer los paradigmas comunes que ocurren en las relaciones de alto nivel entre los sistemas, para que puedan ser entendidos y así puedan surgir nuevos sistemas basados en los viejos.

Obtener la arquitectura correcta es crucial para el diseño del sistema, la arquitectura equivocada puede ser desastrosa. Es necesario un entendimiento de los principios de la arquitectura de *software*.

Una representación de la arquitectura del sistema es esencial para analizar y describir las propiedades a alto nivel de sistemas complejos [12].

3.2 Arquitectura

El objetivo principal de una arquitectura es decir el "qué" es una cosa, no el "cómo" está hecha. Este concepto tiene la misma funcionalidad que la arquitectura que se usa para la creación de edificios.

En la arquitectura la base se encuentra en los planos y estos nos dicen cómo son los edificios, nos dicen si tienen puerta, si tienen salones, si tienen baños, etc. A simple vista de ver los planos puedes ver si se trata de una iglesia, una biblioteca o una casa, no te dice de qué materiales está hecho el edificio pero te dicen qué tipo de edificación es.

Se podrá decir que el edificio no está ligado a los materiales que los componen, la construcción podría hacerse con vidrio, con cemento, incluso con barro y aun así los planos no tendrían motivo para cambiar.

Este mismo concepto es el que utilizó el señor Robert C. Martin para definir cómo funciona la arquitectura de *software*, la arquitectura es sobre intención. Robert Martin busca que a la hora de desarrollar aplicaciones, nos enfoquemos en las reglas de negocio, no con qué *frameworks* o librerías la vamos a crear.

La propuesta de Robert Martin es una arquitectura agnóstica, que no importa que ocurran alrededor de las reglas de negocio, estas no tendrán que cambiar. Podemos cambiar la base de datos, podemos cambiar la interfaz gráfica, podemos quitar y poner librerías y con todo esto las reglas de negocio no deben de cambiar.

Martin define a las base de datos, a las vistas, a las librerías, a los *frameworks*, como "detalles", son "*plugins*" de nuestra aplicación, la aplicación puede perfectamente sobrevivir sin ellas.

Las reglas de negocio solo reciben y producen objetos, y sus propiedades. Los "detalles" no saben cuáles son las reglas de negocio, solo saben qué producen, es un mecanismo de caja negra.

Este tipo de arquitectura fomenta la independencia de agentes externos, y esto facilita el testing, permite la reusabilidad de componentes. Se enfoca en las reglas del negocio, esas reglas que son la base de la aplicación no conocen absolutamente nada del mundo exterior.

Esta arquitectura que él propone no es nueva, Robert C. Martin la creó en base a las ideas de caso de uso propuestas por Ivar Jacobson en la época de 1990 [13]. En la figura 1 se muestra cómo está compuesta la arquitectura "Clean Architecture" del Dr. Robert Martin [1].

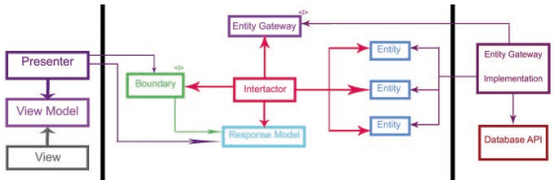

Figura 1. Diagrama de la arquitectura.

3.2.1 *Entities*

Los *Entities* son objetos de negocio, funciones o estructuras de datos, que contiene las reglas del negocio que son independientes de la aplicación.

Esto significa que múltiples aplicaciones pueden compartir las mismas *Entities*, porque los *Entities* no tienen que cambiar entre aplicaciones para ser usables por todas estas. Estos objetos encapsulan las reglas más generales y de alto nivel. Son aquellas reglas que es muy poco probable que cambien si algún agente externo cambia.

Por ejemplo, uno no pensaría en cambiar estos objetos si se cambia la navegación de la página o la seguridad. Ningún cambio operacional de alguna aplicación en particular debe afectar a los *Entities*.

3.2.2 *Interactors*

Los *Use Case* que utiliza esta arquitectura están basados en los propuestos por Ivar Jacobson, en su libro *Object-oriented software engineering: a use case driven approach.*

Jacobson trata a los *Use Case* como un objeto que tiene un conjunto de *Entities* y un conjunto de procesos que se van a ejecutar, solo declaran "que" hacen sin especificar el "cómo" lo hacen.

Jacobson nombró a estos objetos que contienen *Use Case* como *Controllers*, sin embargo Robert C. Martin, los renombró Interactor para que no existiera confusión con el controlador del patrón MVC (*Model-View-Controllers*).

La declaración de un Interactor utiliza el patrón *Command-pattern* para su ejecución, donde existe una función *execute*() que ejecuta todas las funciones necesarias para que funcione el *Use Case.*

El corazón de toda la arquitectura se encuentra en esta capa, esta capa representa todas las reglas del negocio específicas de la aplicación. Todo el flujo de la aplicación ocurre en esta capa, usando *Entities*, pero nunca cambiándolas.

Robert C. Martin establece que esta capa es el centro de la aplicación y todo lo que existe afuera de esta capa son *plugins*. De esta forma los *plugins* pueden cambiar, sin tener que realizar cambios a la capa de aplicación.

3.2.3 *Boundaries*

Los *Boundaries* son una interface que traduce la información externa en un formato que sea más conveniente para los agentes externos, así como también traduce la información que sale [14].

Ningún código que se encuentra en la capa interna de los adaptadores debe conocer nada de las base de datos. Si la base de datos es SQL o NoSQL, eso no debe importar a la capa de aplicación.

3.2.4 *Frameworks & Drivers*

Esta representa la capa más externa, compuesta de *Frameworks* y herramientas como base de datos.

Generalmente en esta capa no se escribe mucho código, además del código que pega la comunicación hacia las capas internas. Esta es la capa en donde todos los detalles viven, la web, la base de datos, las librerías, los *drivers*, todos estos son detalles.

3.2.5 *The Dependency Rule*

Saber cómo es el flujo de cada una de las partes es vital para el funcionamiento de esta arquitectura.

Una dependencia no es más que una pieza de código que necesita otro para poder funcionar. En el caso de esta arquitectura, los módulos de las capas superiores dependen de los módulos interiores. La regla clave e indiscutible del *Clean Architecture* es la siguiente:

El flujo de las dependencias del código solo ocurre desde afuera hacia adentro. Ninguna capa interna debe conocer nada sobre lo que ocurra en las capas externas.

4. Implementación de *"CLEAN Architecture"*

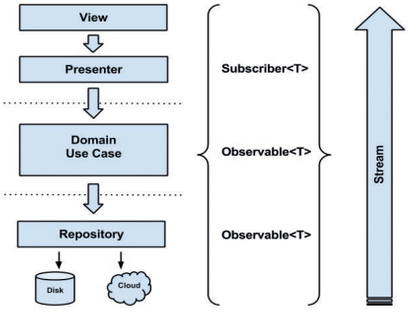

Figura 2. Aproximación de la arquitectura.

4.1 Presentación

Es en esta capa donde existe lo que ve el usuario, botones, entrada de texto, vista, diálogos, gestos, animaciones. En esta capa se puede usar cualquier patrón que se utilice en la creación de interfaces, puede ser MVC, MVVM o MVP.

En la figura 2 se muestra cúal es la aproximación que se usará para la implementación de esta arquitectura.

4.1.1 *Exception*

En esta sección se generan mensajes de acuerdo a las excepciones que ocurran la capa de presentación o en las capas interiores.

4.1.2 *Internal/DI*

Aquí se manejan las dependencias del proyecto, haciendo uso del principio *Inversion of Control Principle,* permitiendo que las dependencias sean satisfechas en tiempo de ejecución sin saber con detalles quién la satisface.

4.1.3 *Mapper*

En esta área se colocan objetos cuyo trabajo es transformar modelos de la capa de dominio a modelos de la capa de presentación. Dentro de la capa de presentación solo se usan modelos de esta capa, se realiza de esta manera para promover la independencia entre capas.

4.1.4 *Model*

Aquí es donde se encontrarán todos los modelos de los que depende la capa de la presentación. En ella solo se encuentran objetos POJO (*Plain Old Java Object*).

4.1.5 *Navigation*

En esta área existe un archivo llamado Navigator.java cuya función es contener funciones que permiten la navegación entre *Activity.* En ella se envía el *Context* y la información que sea requerida por otros *Activity.*

4.1.6 *Presenter*

Aquí es donde habita toda la lógica de la capa de presentación, en ella se declara una interfaz llamada *Presenter* donde, todos los *Presenter* que se encuentren en esta carpeta se tienen que implementar.

La interfaz tiene 3 métodos que implementa, que son cuando se resume, pausa y se destruye la vista. La función principal de los *Presenter* es hacer que las vistas reaccionen cuando ocurra un evento, y también de acuerdo con el modelo, decirle a la vista que y como tiene que renderizar.

4.1.7 *View*

En esta sección van todos los archivos relacionados con la vista, aquí se encuentran los *Activity,* los *Fragment* y los *Adapter.*

4.1.8 *App*

Toda aplicación de Android tiene que tener una clase que represente el ámbito de la aplicación, esta clase representa el ciclo de vida de la aplicación entera, y se encuentra ajena y aislada al ciclo de vida de los *Activity y Fragment.*

Esta clase es el punto de inicio de la aplicación, aquí es el lugar donde se debe cargar todos aquellos componentes de los que requiere la aplicación para funcionar, en esta clase es donde inicializamos el *Dependency Injection* y empezamos a cargar aquellas dependencias que son Singleton, ya que estas dependencias vivirán mientras la aplicación viva.

4.1.9 *UIThread*

En Android existe un hilo principal para la ejecución de la aplicación, este hilo se llama *UIThread,* todos los componentes principales como *Activity*, *Service*, *ContentProviders y BroadcastRecever* ocurren este hilo.

4.2 *Domain*

De todas las capas esta es la más importante. Es en esta capa donde vive el corazón de la aplicación, lo más preciado que tiene la aplicación, las reglas de negocio. Todo ajeno a esta capa *son "detalles",* son *plugins* que se adhieren a nuestra aplicación. Nuestra aplicación puede vivir sin ellas, ya que no dependemos de ellas.

4.2.1 *Exception*

Aquí se encuentra el manejador de errores de la capa de dominio, en ella se encuentran 2 archivos, una interfaz llamada *ErrorBundle* que representa una envoltura o *wrapper* para manejar los errores, y una clase llamada *DefaultErrorBundle* que implementa la interfaz *ErrorBundle* y es una clase que es usada por los

presenter para mostrar los mensajes de errores en la pantalla.

4.2.2 *Executor*

En este paquete se trabaja con interfaces que manejan la concurrencia y trabajan con los hilos de la aplicación. Se usa esta capa para poder trabajar las operaciones de manera asíncrona y de esta forma mejorar la experiencia de usuario.

4.2.3 *Interactor*

Aquí es donde viven las reglas de negocio. En este paquete se encuentran todos los casos de uso que existen en la aplicación, todas las otras capas de la aplicación dependen de ellos, pero ellos no dependen de nadie. Esta es la zona de seguridad donde si llega a ocurrir algún cambio en la aplicación, esta área no se ve afectada.

4.2.4 *Repository*

Este es el *Boundary* entre la capa de dominio y la capa de dato, en esta capa solo existen interfaces cuya función es describir cómo deben ser declaradas las funciones que serán utilizadas por los *UseCase.*

4.3 Data

Esta es la capa donde persisten los datos de la aplicación, cualquier información que tenga que ser guardada o que requiera ser obtenida, vive dentro de esta capa.

4.3.1 *Cache*

Esto es un espacio temporal para guardar información que el programa quiera que persista, esta información es de acceso rápido y es usada de manera frecuente por la aplicación. Puede ser un archivo local en el celular o una base datos locales como SQLite.

4.3.2 *Entity*

Los *Entity* son objetos POJO que representan un objeto en la capa de Datos, al igual que los Modelos en la capa de presentación.

4.3.3 *Exception*

Aquí se manejan todas las excepciones relacionadas con la capa de datos, es en esta área donde se debe pensar cuales son los tipos de errores que deseo que la aplicación notifique al usuario o aquellos eventos que deseo que mi aplicación maneje sin que el usuario intervenga.

4.3.4 *Executor*

En esta área ocurre la implementación de los *Executor* de la capa de dominio, si alguna funcionalidad de mi aplicación puede ser realizada sin necesidad del *UIThread,* es recomendable que lo realice en otro hilo, para mejorar la experiencia de usuario.

4.3.5 *Repository*

Esta puede considerarse la parte vital y más compleja de la capa de datos, toda la capa de datos está pensada y ajustada para un patrón de desarrollo llamado Repository Pattern, donde se maneja la lógica de los datos, si se guardan de manera local o en un servidor remoto, cuál es el proceso para guardar la data. Esta sección es la abstracción de persistencia que posee la aplicación.

5. Conclusiones

El objetivo de este proyecto es documentar y analizar la arquitectura *CLEAN Architecture* propuesta por Robert C. Martin, demostrando cuál es su utilidad para el desarrollo de aplicaciones, que no son solamente aplicaciones móviles, como el prototipo de este proyecto, sino que dicha arquitectura es un modelo de abstracción que puede ser usado en cualquier tipo de programa, ya sean servidores, aplicaciones de escritorio, aplicaciones web y otros por el estilo.

Se realizó un análisis del *framework* Android para el desarrollo de aplicaciones móviles y su importancia en el mundo actual. Igualmente se describió como Google género y continúa innovando el entorno de desarrollo para los programadores, a través de herramientas como IDEs, tutoriales, códigos y documentación.

Referencias

[1] *Uncle Bob. The Clean Architecture.* URL: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html, Agosto 2012. En linea; Consulta 30 de Julio 2016.

[2] Robert Cecil Martin. *Clean code: a handbook of agile software craftsmanship.* NJ: Prentice Hall International, Upper Saddle River, 2009.

[3] Fernando Cejas. *Architecting Android. . .* The clean way? URL: http:// fernandocejas.com/2014/09/03/architecting-android-the-clean-way, Septiembre 2014. En línea; Consulta 29 de Septiembre 2016.

[4] OpenSource. *What is open source software?* URL: https://opensource.com/resources/what-open-source, 2016. En línea; Consulta 2 de Octubre 2016.

[5] Margaret Rouse. *Open Handset Alliance* (OHA). URL: http://searchmobilecomputing.techtarget.com/definition/Open-Handset-Alliance, Marzo 2008. En línea; Consulta 2 de Octubre 2016.

[6] Oracle. *Using the Publish-Subscribe Model for Applications.* URL: https://docs.oracle.com/cd/B10501_01/appdev.920/a96590/adg15pub.htm, 2002. En línea; Consulta 3 de Octubre 2016.

[7] tutorialspoint. *Design Patterns - Observer Pattern.* URL: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm, 2016. En línea; Consulta 3 de Octubre 2016.

[8] GSMARENA. *Form factor.* URL: http://www.gsmarena.com/glossary.php3?term=form-factor, 2016. En línea; Consulta 3 de Octubre 2016.

[9] Lars Vogel. *Android Drawables - Tutorial.* URL: http://www.vogella.com/tutorials/AndroidDrawables/article.html, 2016. En línea; Consulta 4 de Octubre 2016.

[10] Android. *Android Studio 1.1.* URL: http://tools.android.com/download/studio/canary/1-1-0, 2015. En línea; Consulta 5 de Octubre 2016.

[11] Android. *App Manifest.* URL: https://developer.android.com/guide/topics/manifest/manifest-intro.html, 2016. En línea; Consulta 5 de Octubre 2016.

[12] Mary Shaw David Garlan. *An Introduction to Software Architecture.* World Scientic Publishing Company, 1994.

[13] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach.* ACM Press; Addison-Wesley Pub, revised edition, 1992.

[14] Luis Zamith. *Clean Architecture.* URL: https://subvisual.co/blog/posts/20-clean-architecture, 2013. En línea; Consulta 8 de Octubre 2016.