

# UNIVERSIDAD NACIONAL DE CÓRDOBA



---

PROYECTO INTEGRADOR DE INGENIERÍA EN COMPUTACIÓN

---

## Diseño e Implementación de una solución IoT para el control y monitoreo remoto de cerramientos eléctricos

---

*Autor:*

Esteban Andrés MORALES

*Matrícula:*35.104.714

Facultad de Ciencias Exáctas, Físicas y Naturales

Laboratorio de Arquitectura de Redes y Computadoras.

*Director Docente:*

Mg.Ing. Miguel SOLINAS

5 de noviembre de 2020

# Índice general

<b>Contenido</b>	<b>I</b>
<b>Lista de Figuras</b>	<b>III</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Dominio del problema . . . . .	1
1.3. Objetivos . . . . .	1
1.3.1. General . . . . .	1
1.3.2. Objetivos Particulares . . . . .	1
1.3.3. Objetivos Específicos . . . . .	1
1.4. Metodología de Trabajo . . . . .	2
<b>2. Marco Teórico</b>	<b>3</b>
2.1. Cerramientos Eléctricos . . . . .	3
2.1.1. Principio de Funcionamiento . . . . .	3
2.1.1.1. Solenoide y Perno . . . . .	4
2.1.1.2. Solenoide y Clavija . . . . .	4
2.1.1.3. Electroimán . . . . .	5
2.1.1.4. Motor Eléctrico . . . . .	5
2.1.2. Controladores . . . . .	6
2.1.2.1. Sensores . . . . .	7
2.1.2.1.1. Sensores de Fin de Carrera . . . . .	7
2.1.2.1.2. Encoder Rotativo . . . . .	8
2.1.2.2. Monitoreo y Alerta . . . . .	8
2.1.2.2.1. Monitoreo Remoto . . . . .	9
2.1.2.3. Llaves Electrónicas y Control Remoto . . . . .	9
2.1.2.3.1. Llaves RFID . . . . .	9
2.1.2.3.2. Control Remoto RF . . . . .	10
2.1.2.3.3. Un problema de seguridad . . . . .	11
2.1.3. Cerramientos Electrónicos IoT . . . . .	11
2.1.3.1. Smart Alarms . . . . .	11
2.1.3.2. Smart Door Locks . . . . .	12
2.1.3.3. Smart Garage Door . . . . .	12
2.1.4. Conclusión . . . . .	14
<b>3. Clean Architecture</b>	<b>15</b>
3.1. Aplicación Cliente . . . . .	15

3.2.	Uncle Bob Clean Architecture . . . . .	15
3.2.1.	Dominio Transparente . . . . .	16
3.2.2.	Regla de Dependencias . . . . .	16
3.2.3.	Principio de Abstracción . . . . .	17
3.2.4.	Comunicación entre Capas . . . . .	19
3.2.5.	Diseño de las Capas . . . . .	20
3.2.6.	Presentation Layer: MVP . . . . .	21
3.2.7.	Domain Layer: Commander Pattern . . . . .	22
3.2.8.	Data Layer: Repository Pattern . . . . .	25
3.3.	Programación Reactiva . . . . .	27
3.3.1.	Patrón Observer . . . . .	28
3.3.2.	Patrón Iterator . . . . .	28
3.3.3.	Programación Reactiva y Clean Architecture . . . . .	28
<b>4.</b>	<b>Diseño</b>	<b>30</b>
4.1.	Dominio del Problema . . . . .	31
4.1.1.	Definición de Casos de Usos y Escenarios . . . . .	31
4.1.2.	Definición de Requerimientos Funcionales y de Sistema . . . . .	31
4.2.	Arquitectura de la Aplicación . . . . .	31
4.2.1.	Módulos y Paquetes . . . . .	31
4.2.2.	Estructura de Capas . . . . .	31
4.2.3.	Inyección de Dependencias . . . . .	31
4.2.4.	Objetos Android y Ciclo de Vida . . . . .	31
4.3.	Mitigación de Errores . . . . .	31
4.3.1.	Debugging . . . . .	31
4.3.2.	Logging Remoto . . . . .	31
4.4.	Entorno de Trabajo . . . . .	31
4.4.1.	Objetos Falsos . . . . .	31
4.4.2.	Entorno de Pruebas . . . . .	31
4.4.3.	Broker MQTT local . . . . .	31
<b>5.</b>	<b>Conclusiones</b>	<b>32</b>
<b>Bibliografía</b>		<b>34</b>

# Índice de figuras

2.1. Cerramientos de perno . . . . .	4
2.2. Cerramientos de perno . . . . .	5
2.3. Cerradura Electroimán . . . . .	5
2.4. Motor Portones Automatizados . . . . .	6
2.5. Controladores de Acceso . . . . .	7
2.6. Sensores de fin de carrera . . . . .	8
2.7. Encoder Rotativo . . . . .	8
2.8. Llaves RFID . . . . .	10
2.9. Controles RF . . . . .	10
2.10. Smart Alarms . . . . .	12
2.11. Smart Locks . . . . .	13
2.12. Smart Garage Gates . . . . .	13
3.1. Principio de Dependencias . . . . .	17
3.2. Abstraction Principle . . . . .	18
3.3. Abstraction Principle . . . . .	18
3.4. Layer Communication . . . . .	19
3.5. Dependencia de Módulos . . . . .	20
3.6. MVP Components . . . . .	21
3.7. MVP Sequence . . . . .	22
3.8. Commander Classes . . . . .	23
3.9. MVP Components . . . . .	23
3.10. Commander Review . . . . .	24
3.11. Repository Pattern Class Diagram . . . . .	25
3.12. Repository Pattern Detailed Class Diagram . . . . .	26
3.13. Modified Repository Pattern Class Diagram . . . . .	27
3.14. Observer Class Diagram . . . . .	28
3.15. Iterator Class Diagram . . . . .	29
3.16. Modified Repository Pattern Class Diagram . . . . .	29

# **Capítulo 1**

## **Introducción**

### **1.1. Motivación**

sadasasd

### **1.2. Dominio del problema**

asdasd

### **1.3. Objetivos**

asdasd

#### **1.3.1. General**

asdasd

#### **1.3.2. Objetivos Particulares**

asdasd

#### **1.3.3. Objetivos Específicos**

asdasd

## 1.4. Metodología de Trabajo

asdasd

## **Capítulo 2**

# **Marco Teórico**

### **2.1. Cerramientos Eléctricos**

Cuando hablamos de cerramientos eléctricos se hace referencia a todos los dispositivos de funcionamiento electromecánico destinados a ofrecer la función de cerradura para aperturas de instalaciones domiciliarias, comerciales e industriales. Se listan a continuación ejemplos de estos dispositivos.

- Cerraduras por electro-imán.
- Cerraduras de perno.
- Portones automatizados.
- Barreras automáticas.
- Pasadores eléctricos.
- Traba-pestillo eléctrico.
- Pestillo eléctrico para cajones o guarda equipajes.
- Puertas de Ascensor con control de acceso.

#### **2.1.1. Principio de Funcionamiento**

Aunque todos los cerramientos eléctricos comparten la misma funcionalidad existe una variedad acotada de principios de funcionamiento, diseño industrial o factor de forma entre los cuales podemos mencionar:

- Solenoide y Perno

- Solenoide y Clavija
- Electroimán
- Motor Eléctrico

### 2.1.1.1. Solenoide y Perno

Un solenoide es enrollado alrededor de un eje cilíndrico dieléctrico hueco que a su vez envuelve un perno también cilíndrico pero metálico. Este solenoide está conectado en un circuito de corriente continua. Cuando se alimenta dicho circuito el campo magnético inducido en el núcleo del solenoide genera corrientes de Foucault sobre el perno que es desplazado por la fuerza del campo magnético, generando así la acción de traba o bloqueo sobre el herraje de la abertura. Este tipo de principio de funcionamiento es el que emplean dispositivos tales como los pestillos eléctricos y las cerraduras de perno que se muestran en la figura 2.1.



FIGURA 2.1: Se muestran ejemplos de cerramientos eléctricos que funcionan con solenoide y perno.

### 2.1.1.2. Solenoide y Clavija

De una forma similar a la descripta para el caso del perno, un solenoide se utiliza para formar un pequeño electroimán en el interior del dispositivo. La fuerza inducida deforma de manera elástica una clavija o chapa metálica delgada en forma de ele. Esta deformación libera el mecanismo que bloquea el movimiento del pestillo. Este es el caso del traba-pestillo eléctrico o del pasador eléctrico cuyos ejemplos se muestran en la figura 2.2.



---

FIGURA 2.2: Se muestran ejemplos de cerramientos eléctricos que funcionan con solenoide y clavija.

#### 2.1.1.3. Electroimán

En esta forma de funcionamiento se construye un electroimán de dimensiones considerables capaz de ejercer una fuerza de atracción magnética cercana a los 300 kgF sobre una chapa metálica que también se incluye como parte del cerramiento. Por lo general incluyen en su circuito una etapa de compensación de factor de potencia para remanencia cero. Este es el caso de la cerradura por electroimán que se muestra en la figura 2.3.



---

FIGURA 2.3: Se muestran un ejemplo de cerradura eléctrica que funcionan por electroimán.

#### 2.1.1.4. Motor Eléctrico

Para el caso de grandes portones de garage o acceso vehicular se utilizan motores de corriente alterna o brushless de corriente continua con su respectivos controladores. Estos motores interactúan mecánicamente con otras interfaces mecánicas instaladas en

las aperturas para posibilitar la tarea de apertura o cierre. Adicionalmente es necesario la incorporación de sensores que indiquen el estado del sistema y aseguran su correcto funcionamiento. Todos los tipos de automatización de portones así como los sistemas de apertura con cortinas metálicas comparten el mismo principio de funcionamiento y utilizan motores similares al que se muestra en la figura 2.4.



---

FIGURA 2.4: Motor empleado en sistemas de automatización de portones de garaje o accesos vehiculares.

## 2.2. Controladores

Todos estos cerramientos eléctricos no funcionan por si mismos sino que requieren de un dispositivo controlador. Algunos ejemplos se muestran en la figura 2.5.

Estos controladores son circuitos electrónicos que se encargan de generar las señales de activación y/o control de los cerramientos eléctricos. El usuario podrá actuar sobre el cerramiento eléctrico a través de diversos modos de accionamiento se mencionan los más habituales:

- portero telefónico
- Contraseña por teclado
- Lector biométrico
- Llave electrónica
- Botón externo

Para dar soporte a una o múltiples formas de accionamiento el controlador ofrece una interfaz de configuración y los componentes necesarios para su operación. En el caso de los cerramientos a motor es necesario incluir sensores de apertura y funcionamiento.

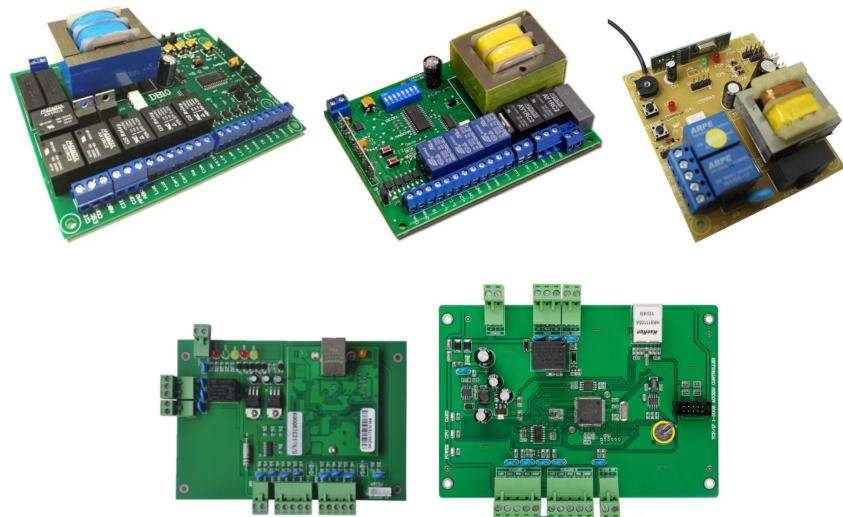


FIGURA 2.5: Algunos ejemplos de controladores de cerramientos eléctricos.

### 2.2.1. Sensores

Por lo general solo son empleados por los controladores para cerramientos a motor. Existen varios tipos de sensores para estos controladores se mencionan los más utilizados:

- Sensores de Fin de Carrera
- Encoder Digital

#### 2.2.1.1. Sensores de Fin de Carrera

Se utiliza el plural para este caso de sensado porque se utilizan de a pares. Se trata de un par de sensores binarios que indican apertura o cierre total del cerramiento. Suelen venir en dos presentaciones con modo de funcionamiento distinto.

**Por proximidad magnética:** En este caso cada uno de los sensores está compuesto por un imán permanente y un switch encapsulado como se muestra en la figura 2.6(a). Este encapsulado contiene en su interior un filamento de material ferromagnético flexible que al aproximarse lo suficiente al imán se desplaza y cierra el circuito.

**Por choque mecánico:** Estos sensores son switches mecánicos muy sensibles que se instalan en los límites del marco de la abertura y al mínimo contacto con la parte móvil cierran sus circuitos. En la figura 2.6(b) se muestran ejemplos de estos dispositivos.

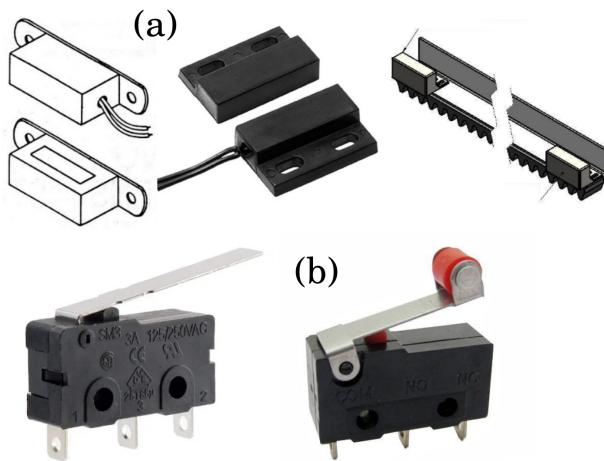


FIGURA 2.6: Ejemplos de sensores de fin de carrera.(a) Sensores por proximidad magnética, (b) Sensores por choque mecánico.

### 2.2.1.2. Encoder Rotativo

Un encoder es un sensor de movimiento mecánico que genera señales digitales en respuesta al movimiento y puede proveer información sobre la posición, la velocidad y la dirección del movimiento. En particular el encoder rotativo responde al movimiento de rotación de un eje. Por lo general los controladores de cerramientos eléctricos a motor utilizan encoders rotativos incrementales que generan un tren de pulsos que se puede utilizar para determinar la posición y la velocidad del eje del motor. En la figura 2.7 se observa la forma del tren de pulsos que genera el encoder.

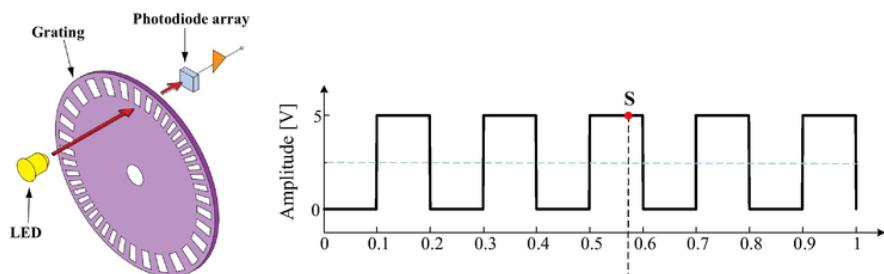


FIGURA 2.7: Ilustración simplificada de un encoder rotativo con salida de onda cuadrada TTL.

### 2.2.2. Monitoreo y Alerta

La mayoría de los controladores poseen interfaces que permiten la conexión local de luminarias testigo, así como bocinas de alerta. Algunos modelos incluyen un puerto serie

(RJ45, RJ14) capaz de comunicarse mediante comandos estándar con los tradicionales sistemas de alarma.

### 2.2.2.1. Monitoreo Remoto

Al momento de iniciar el presente proyecto integrador no existían controladoras de acceso con la funcionalidad de monitoreo remoto incorporada de fábrica. La alternativa para tener información del cerramiento eléctrico y el estado de la abertura que opera consiste en instalar un sensor de apertura y conectarlo a una central de alarma. Estos sensores son baratos y su principio de funcionamiento coincide con el descripto para los sensores de fin de carrera y que se puede consultar en la sección [2.2.1.1](#). Las centrales de alarma suelen estar conectadas por cableado telefónico a un servicio de monitoreo que lanza las notificaciones pertinentes según se hayan configurado. Una basta mayoría de proveedoras de servicio de monitoreo de alarmas ahora ofrecen plataformas web o aplicaciones móviles que permiten el monitoreo de las centrales en tiempo real.

### 2.2.3. Llaves Electrónicas y Control Remoto

Actualmente la mayoría de los controladores de acceso soportan algún tipo de llave electrónica. Una llave electrónica es un dispositivo electrónico que le permite al usuario operar el cerramiento eléctrico. Existen diversos tipos de llaves electrónicas pero sólo se mencionan las más utilizadas.

- Llaves RFID
- Control Remoto RF

#### 2.2.3.1. Llaves RFID

RFID (Radio-frequency identification) es un sistema de almacenamiento y recuperación de datos a distancias cortas por emisión de radiofrecuencia de baja potencia. Una etiqueta RFID consiste en un pequeño circuito receptor y transmisor de radio. Cuando se activa por un pulso de interrogación electromagnético de un dispositivo lector de RFID cercano, la etiqueta transmite datos digitales, generalmente un número identificatorio de vuelta al lector. Estas etiquetas o circuitos RFID son tan diminutos que pueden incluirse en objetos de tamaño reducido como llaveros y tarjetas, algunos ejemplos se muestran en la figura [2.8](#). El usuario del sistema de acceso aproximará esta llave al lector RFID incluido en el controlador y se producirá el intercambio del número identificatorio. En caso de que ese dato esté registrado en el controlador el usuario podrá controlar el cerramiento eléctrico.



---

FIGURA 2.8: Llaveros y tarjetas con etiquetas RFID.

### 2.2.3.2. Control Remoto RF

RF es el acrónimo de Radio-Frequency y hace referencia a las tecnologías que emplean sistemas de comunicación por emisión de ondas electromagnéticas. Por lo general se utilizan anchos de bandas libres sin restricciones gubernamentales con potencias de transmisión baja. En el caso del control remoto por RF un código numérico identificatorio se transmite al controlador que incluye un circuito receptor comúnmente en las frecuencias que van de los 200 MHz a los 500 Mhz. El rango de acción de estos dispositivos es de aproximadamente 100 mts en línea de visión. El controlador de acceso permite la actualización del código identificatorio en caso de que ya esté empleado al momento de la configuración. En algunas implementaciones este código es rotatorio y cambia con el uso. En la figura 2.9 se muestran algunos ejemplos.



---

FIGURA 2.9: Controles remotos RF en diferentes presentaciones.

### 2.2.3.3. Un problema de seguridad

Tanto el empleo de llaves RFID como controles RF presentan un problema de seguridad en estos sistemas de acceso electrónico. Conceden acceso a quien porta el token de identificación sin importar si se trata del dueño real. En ningún caso se utilizan protocolos para la identificación del solicitante. Así mismo tampoco se emplean algoritmos de encripción de datos por lo que el código de identificación se transmite en texto plano, por lo menos este es el caso para la mayoría de las implementaciones.

Por esta razón existen métodos de hacking ya documentados que involucran dos técnicas principales.

**Clonado por lectura RFID:** En este caso simplemente basta con tener un lector RF y acceso a la etiqueta RFID empleada por la llave a ser clonada. Una vez leído el código se crea una etiqueta con el mismo código y se obtiene acceso.

**Clonado por intercepción RF:** Esta técnica implica un ataque por sniffing del código de identificación. Teniendo en cuenta que la frecuencia y el tipo de modulación empleado por los sistemas actuales son conocidos, resulta en una tarea sencilla empleando un transeptor RF y un decodificador por software.

**Inhibición de Señal:** Esta técnica implica la utilización de un transmisor que contamina con ruido el ancho de banda de operación del control remoto impidiendo la correcta comunicación con el controlador del cerramiento.

## 2.3. Cerramientos Electrónicos IoT

IoT (Internet of Things) describe la red de objetos físicos que incorporan circuitos con microcontroladores, software embebido, sensores y componentes de comunicación con el propósito de intercambiar datos con otros objetos conectados y servicios en línea a través de internet. Teniendo en cuenta que las aberturas de un edificio son objetos físicos y que los cerramientos eléctricos ofrecen un modo de accionamiento electrónico para estos objetos, es posible concebir la adaptación de los mismos para convertirlos en implementaciones IoT. Con el ánimo de ilustrar el panorama de productos IoT orientados a la seguridad y al control de acceso se mencionan las categorías más relevantes.

- Smart Alarms
- Smart Door Locks
- Smart Garage Doors

### 2.3.1. Smart Alarms

Estos productos proponen la instalación y configuración de una red de sensores (comúnmente inalámbricos) en una topología tipo estrella. En el centro de la red se encuentra un

dispositivo electrónico denominado *gateway* que posee al menos dos interfaces de comunicación principales, una para conectarse a la red de sensores y otra para acceder a internet. Esta "central" o gateway recibe las actualizaciones de estado de los sensores y las reenvía por internet al servicio de monitoreo. El usuario puede acceder a los datos almacenados por el servicio a través de un cliente web o aplicación móvil. Adicionalmente estos clientes ofrecen una GUI para la configuración tanto de la central como para los sensores. En la figura 2.10 se muestran ejemplos de los productos ofrecidos por los fabricantes más populares.



FIGURA 2.10: Familia de productos de los fabricantes mas conocidos.

### 2.3.2. Smart Door Locks

Esta categoría de productos comprenden dispositivos electromecánicos que reemplazan o se adaptan a las tradicionales cerraduras de tambor radial comúnmente utilizadas en puertas residenciales estadounidenses. Estos dispositivos incluyen el sistema embebido capaz de conectarse a internet a través de WiFi o Bluetooth (mediante el uso de un gateway). En todos los casos se ofrece una aplicación móvil que hace de llave virtual y permite el acceso a estos cerramientos inteligentes. Adicionalmente el usuario administrador podrá agregar y remover usuarios autorizados a operar el dispositivo. En la figura 2.11 se pueden observar ejemplos reales de productos disponibles en el mercado.

### 2.3.3. Smart Garage Door

Para el caso de los portones de garaje existen diversas implementaciones que agregan estos objetos a la familia de productos IoT.

Debido a que la mayoría de los controladores ofrecen al menos una interfaz para conexión de botón externo para apertura y cierre. Diversas startups desarrollaron implementaciones de dispositivos a modo de accesorios compatibles con una familia de controladores. Este es el caso de ISmartGate, Nexx NXG-100 y Garadget.

Luego los fabricantes más importantes de controladores y motores para portones de



FIGURA 2.11: Modelos más comercializados de cerraduras inteligentes.

garaje de estados unidos realizaron sus propias implementaciones y las incorporaron a sus controladores como una funcionalidad built-in. Este es el caso de la empresa Genie con su linea de productos Aladdin. Por otro lado la empresa Chamberlane optó por el enfoque de agregar la misma funcionalidad mediante el uso accesorios que ofrecen con su línea myQ.

Mas recientemente, en el transcurso del año 2019 PPA, la empresa más grande de Sudamérica especializada en automatización de aberturas residenciales e industriales. Presentó un gateway IoT (SPIRIT) compatible con una amplia familia de sensores y actuadores que también fabrican. Entre ellos un módulo que permite el accionamiento y monitoreo remoto de portones automatizados. Todas las implementaciones ofrecen una aplicación móvil que se usará para el monitoreo, control y configuración del producto. En la figura 2.12 se muestran ejemplos de estos productos.



FIGURA 2.12: Modelos más comercializados de automatización inteligente para portones de garaje.

## 2.4. Conclusión

Los cerramientos eléctricos presentan una interesante gama de dispositivos para los que pueden plantearse la extensión de funcionalidad con el objetivo de convertirlos en productos IoT. Algunas de las funcionalidades que se entienden deseables para este tipo de dispositivos son el monitoreo remoto, el control a distancia y la configuración remota. Adicionalmente aparece una oportunidad de mejora al plantear los problemas de seguridad que tienen las implementaciones actuales y que podrían ser mitigados al utilizar protocolos de comunicación modernos.

## Capítulo 3

# Clean Architecture

En el presente capítulo se describe el proceso de desarrollo del sistema que se compone en 3 partes, las cuales se detallan en cada caso el procedimiento de diseño e implementación utilizando como referencia el Documento de Especificación de Requerimientos de Sistema de *Software* propuesto por la *IEEE*.

### 3.1. Aplicación Cliente

Para realizar la primera implementación de la aplicación cliente se eligió la plataforma de desarrollo para dispositivos android, más precisamente teléfonos inteligentes y tabletas.

El objetivo principal de emplear una estructura fija para la implementación del proyecto es utilizar un único "lenguaje arquitectónico" que resulte familiar a los integrantes de un posible equipo de desarrollo así como transversal tanto para la implementación android, iOS o cualquier otra plataforma que pueda aparecer durante la vida útil del producto. De esta manera no es necesario pagar un costo demasiado alto al incluir una implementación del mismo sistema para una plataforma distinta. Los equipos de cada una de estas implementaciones podrán discutir aspectos de diseño, validar reglas de negocio y evacuar dudas sin tener en cuenta los detalles de las plataformas, así mismo será más fácil conservar coherencia y mostrar armonía entre las implementaciones nativas para dichas plataformas.

### 3.2. Uncle Bob Clean Architecture

También conocida como arquitectura de capas (onion architecture). El punto principal de este enfoque es que la lógica de negocio, también conocido como dominio, está en el centro del universo (Al medio entre las entradas del sistema y las salidas)[\[1\]](#).

### 3.2.1. Dominio Transparente

Cuando se listan los directorios de un proyecto que cumple con los lineamientos de esta arquitectura, con tan solo leer el nombre de las carpetas debería ser posible casi de inmediato tener una idea de qué se trata esta aplicación, independientemente de la tecnología. Todo lo demás es un *detalle de implementación*[2].

Por ejemplo, la persistencia es un detalle. Definir una interfaz con el objetivo de establecer la responsabilidad con un contrato (Contrato de Persistencia), de esta forma uno podría implementar de una manera rápida e insuficiente una estrategia de persistencia en memoria RAM y no pensar en ello demasiado sino hasta que la lógica de negocio esté completamente definida. Una vez definidos los requerimientos de persistencia y verificados con el cliente, se puede proceder a tomar una decisión definitiva de **cómo** deberán persistirse los datos.

Almacenamiento en una base de datos local, comunicación remota con un servicio REST, almacenamiento sobre el sistema de archivos, ante este panorama incluso sería razonable pensar el planteo de la creación de un esquema de cache, y sin embargo no es poco frecuente que luego de lalicitación de requerimientos resulte que el sistema no tiene que persistir ningún resultado en absoluto.

En una frase: *las capas internas contienen lógica de negocios, las capas externas contienen detalles de implementación*. Adicionalmente esta arquitectura debe cumplir con un conjunto de características:

- Regla de dependencia
- Abstracción
- Comunicación entre capas

### 3.2.2. Regla de Dependencias

La regla de dependencia se ilustra en la figura 3.1:

Las capas externas deben depender de las capas internas. Permaneciendo en el centro las entidades del dominio inmediatamente seguidas por los objetos que encapsulan la lógica de negocio y que tienen acceso a tal dominio. Se muestran tres flechas en un recuadro rojo que representan el sentido de las dependencias. En lugar de "depende", tal vez sea mejor usar términos como "ve", "conoce" o "está consciente de...". En estos términos, las capas externas ven, conocen y son conscientes de las capas internas, pero las capas internas no ven ni conocen, ni son conscientes de, las capas externas. Como dijimos anteriormente, las capas internas contienen lógica de negocios y las capas externas contienen detalles de implementación. Combinado con la regla de dependencia, se deduce que la lógica de negocio no ve, ni conoce, detalles de implementación.

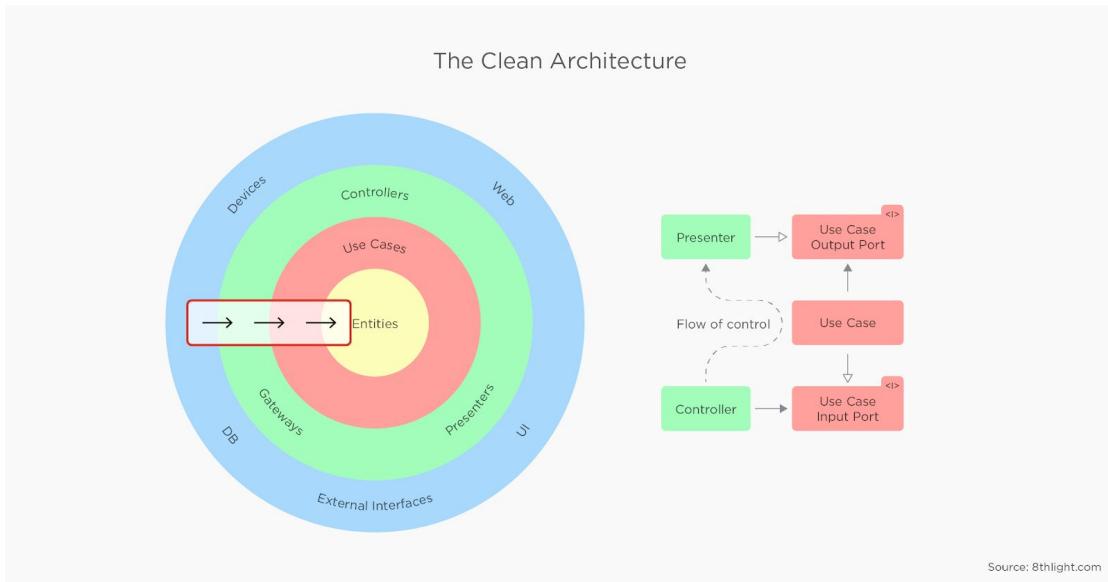


FIGURA 3.1: Esquema de dependencias para una arquitectura en capas.

No existe una única forma de implementar esta regla, dependerá del encargado del proyecto. Una estrategia consiste en colocar las clases de cada capa en paquetes diferentes, poniendo especial cuidado en no importar paquetes "externos" en paquetes "internos". Sin embargo, si algún programador del equipo no es consciente del principio de dependencias, nada les impediría romperlo. Un mejor enfoque sería separar las capas en diferentes módulos de construcción independiente, y ajustar las dependencias en el archivo de construcción para que la capa interna simplemente no pueda utilizar la capa externa, sin embargo este enfoque implica un exhaustivo conocimiento de la herramienta de construcción de la plataforma para la que se está desarrollando.

### 3.2.3. Principio de Abstracción

El principio de la abstracción ya se ha insinuado antes. Postula que a medida que se están moviendo hacia el centro del diagrama, las implementaciones se vuelven más abstractas, agnósticas de plataformas y frameworks. Eso tiene sentido, como lo repetimos anteriormente el círculo interno contiene lógica de negocios y el círculo exterior contiene detalles de implementación.

Incluso puede plantearse el mismo componente lógico dividido entre varias capas, como se muestra en el diagrama 3.2. La parte más abstracta se puede definir en la capa interna, y la parte más concreta en la capa externa.

De esta manera, la lógica de negocios podría producir como un efecto secundario que se muestren notificaciones del sistema por ejemplo, pero no sabe nada acerca de los detalles de la implementación (cómo se implementan las notificaciones para una plataforma dada), por lo tanto la regla de las dependencias se conserva.

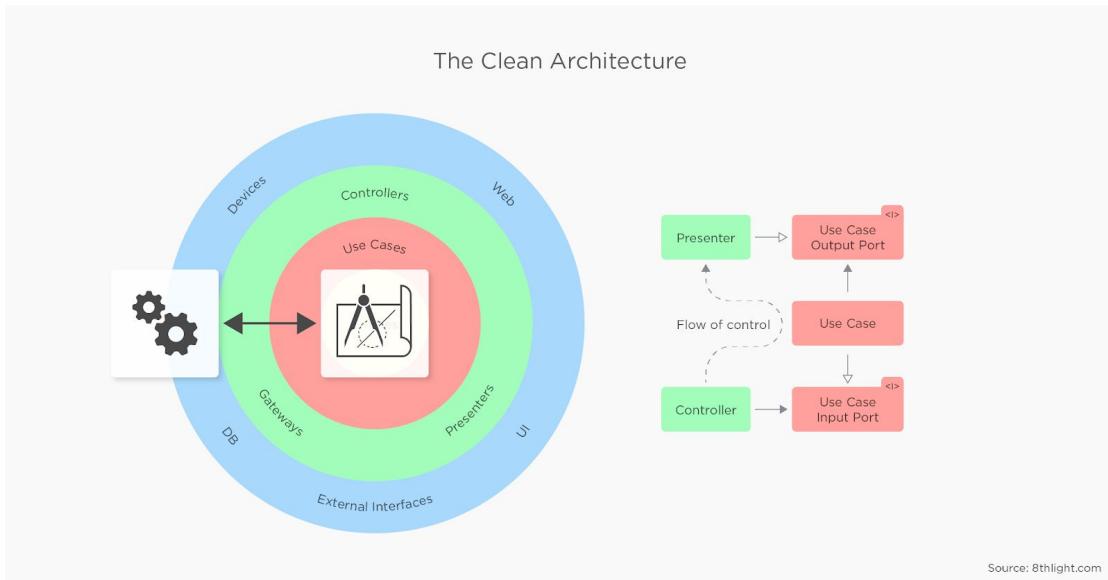


FIGURA 3.2: Principio de Abstracción en una arquitectura por capas.

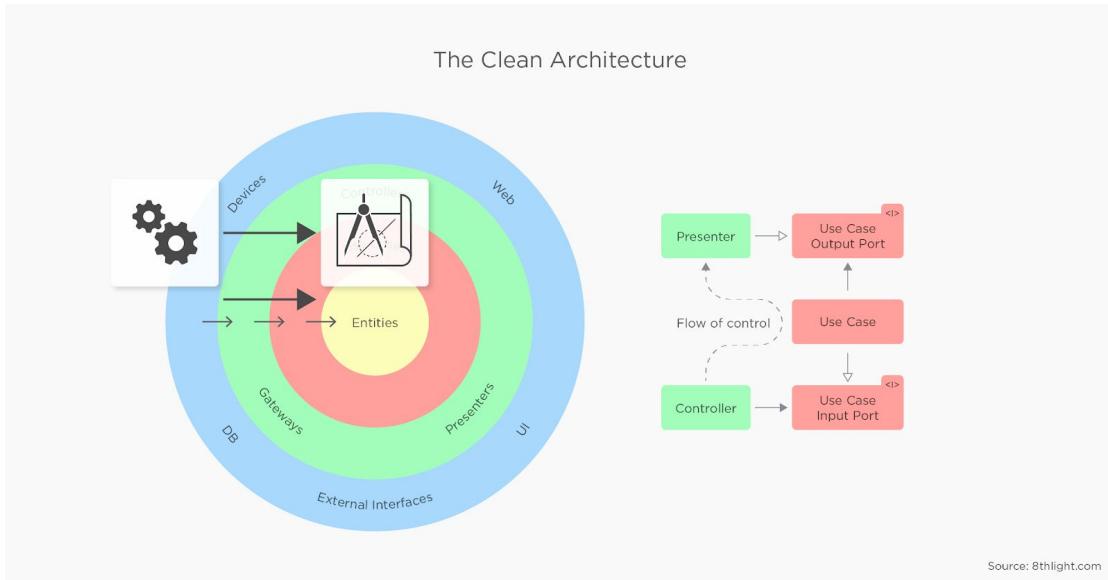


FIGURA 3.3: Principio de Abstracción en una arquitectura por capas.

### 3.2.4. Comunicación entre Capas

La lógica del negocio está en el centro del diagrama y debe mediar entre los sistemas externos de salida y los sistemas externos de entrada como la interfaz de usuario, pero ni siquiera sabe que esos dos tipos existen. Esto es un desafío de comunicación y flujo de datos. Necesitamos que los datos sean capaces de fluir de las capas externas a las internas y viceversa, pero la regla de dependencia no lo permite.

Solo tenemos dos capas, la verde y la roja. La capa verde es exterior y sabe sobre la capa roja, la roja es interior y solo se conoce a sí mismo. Necesitamos que los datos fluyan desde el verde al rojo y viceversa. La solución propuesta se muestra en el diagrama 3.4:

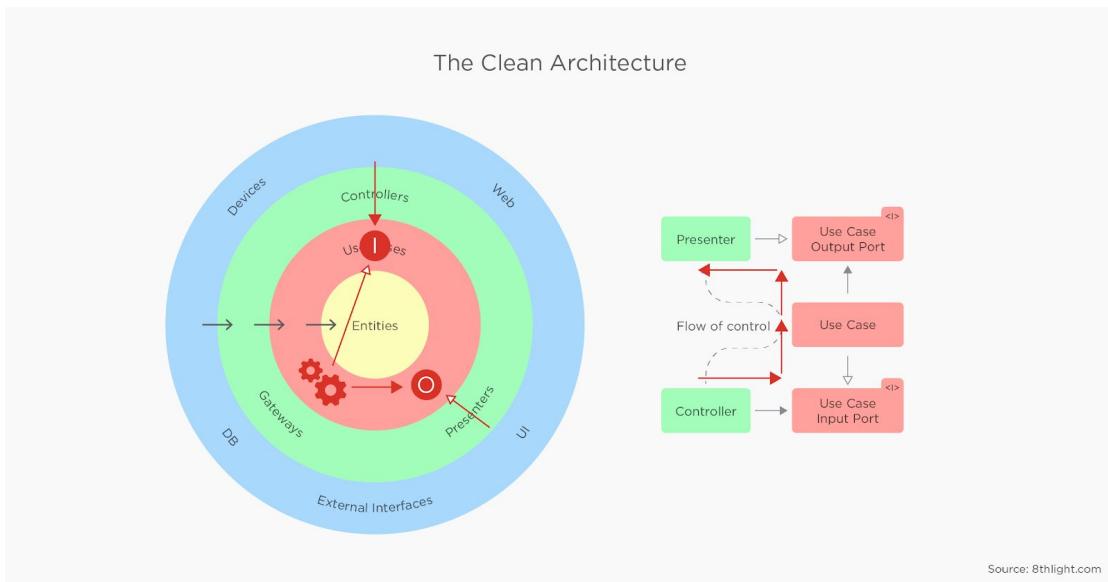


FIGURA 3.4: Comunicación entre capas.

En la parte inferior derecha del diagrama 3.4 muestra el flujo de datos. Los datos van desde el controlador, a través del puerto de entrada del caso de uso (o reemplazar el caso de uso con el componente de su elección), luego a través del propio caso de uso y después a través del puerto de salida del caso de uso al presentador.

El controlador tiene un puerto de entrada, literalmente tiene una referencia a él. Llama a un método en él, de modo que los datos van del controlador al puerto de entrada. Pero el puerto de entrada es una interfaz, y la implementación real es el caso de uso: por lo que ha llamado un método en un caso de uso y los flujos de datos al caso de uso. El caso de uso hace algo y quiere enviar los datos de vuelta. Tiene una referencia al puerto de salida, ya que el puerto de salida está definido en la misma capa, por lo que puede llamar al método en él. Por lo tanto, los datos van al puerto de salida. Y finalmente, el presentador es, o implementa, el puerto de salida.

### 3.2.5. Diseño de las Capas

Los mejores intentos de implementación para proyectos android de esta arquitectura vienen de la mano de un desarrollador argentino Fernando Cejas [3] (SoundCloud) y los ejemplos de los Blueprints de arquitectura de Google[4].

Ambas propuestas dividen la implementación en tres capas principales, una capa de presentación, una capa de dominio y la capa de datos. Cada una de estas capas tiene una responsabilidad bien definida y se comunica con una única capa respetando la regla de dependencia establecida anteriormente cuando se definían los lineamientos generales de la arquitectura. La cadena de dependencias puede apreciarse en la figura 3.5 la capa de presentación le comunica las demandas del usuario a partir de las interacciones recibidas a la capa de dominio que a su vez realiza las solicitudes de datos a la capa de datos que una vez resueltos le comunica los resultados a la capa de dominio que ejecutará la lógica de negocios pertinente generando una respuesta que producirá los efectos deseados en la capa de presentación.

A continuación se describe brevemente las responsabilidades de cada capa.

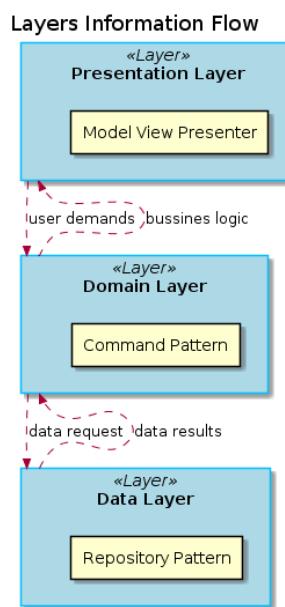


FIGURA 3.5: Esquema de dependencias para una arquitectura en capas.

- **Presentation Layer:** Esta capa se encarga de presentar la interfaz de usuario, esto es, mostrar por pantalla los objetos visuales correspondientes y recibir los eventos de interacción que realiza el usuario. Para la implementación se recomienda el empleo del patrón de diseño conocido como **MVP (Model View Presenter)**.
- **Domain Layer:** Esta capa contiene toda la lógica de negocio. La capa de dominio contiene las clases denominadas casos de uso o interactores según la literatura. Estos objetos encapsulan los escenarios contemplados por la lógica de negocio y

son ejecutados por la capa de presentación. Estos casos de uso representan todas las acciones posibles admitidas por el sistema y que pueden ser compuestas en la implementación por los desarrolladores siempre desde la capa de presentación. Para la implementación de estos casos de uso se sugiere la utilización del patrón de diseño conocido como **Command Pattern**.

- Data Layer: Esta capa administra la adquisición de datos y es capaz de utilizar diferentes orígenes de datos, así como la lógica de cache o persistencia temporal. Esta capa se suele implementar utilizando el patrón de diseño conocido como **Repository Pattern**.

### 3.2.6. Presentation Layer: MVP

El patrón de arquitectura que se utiliza en la capa de presentación de ambas implementaciones se conoce como Modelo-Vista-Presentador. La idea detrás del patrón es concentrar la lógica de la interacción con el usuario en una entidad conocida como presentador, las operaciones directamente relacionadas con la manipulación de objetos gráficos y la captura de acciones de usuario están delegadas a la entidad Vista, finalmente la adquisición de datos y la ejecución de los algoritmos que encapsulan la lógica de negocio forman parte de las entidades modelo en el patrón [5]. El diagrama de componentes 3.6 describe la relación entre los objetos principales propuestos por el patrón.

**Model View Presenter Overview**

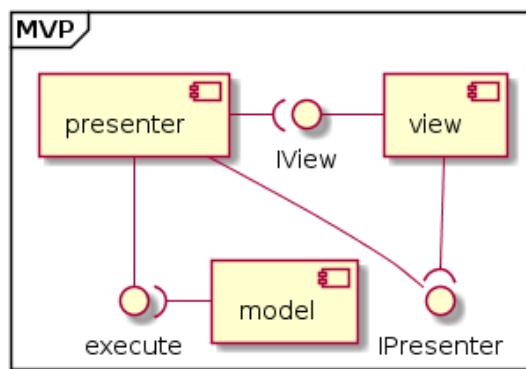


FIGURA 3.6: Diagrama de componentes del patrón.

Es posible deducir el esquema de comunicación entre los componentes a partir del diagrama. La vista se comunica de manera bidireccional con el presentador y cuando es necesario el presentador se comunica de manera unidireccional con el modelo.

Una convención para la implementación del patrón es tratar de generar vistas completamente ajenas de cualquier lógica operativa y agnósticas del estado de la aplicación. Esto las convierte en un mero instrumento de interfaz entre lo que percibe el usuario y sus reacciones.

Otra de las convenciones sugiere utilizar objetos modelo-vista en la comunicación entre el presentador y la vista para estandarizar el tipo de mensaje y el proceso de actualización de la vista.

En el caso de las implementaciones antes mencionadas la interfaz con el modelo es satisfecha mediante el uso de objetos casos de uso ó interactoros, ambos términos suelen utilizarse de manera intercambiable.

La secuencia de mensajes que son intercambiados entre los objetos del patrón se ilustran en el diagrama de secuencias 3.7

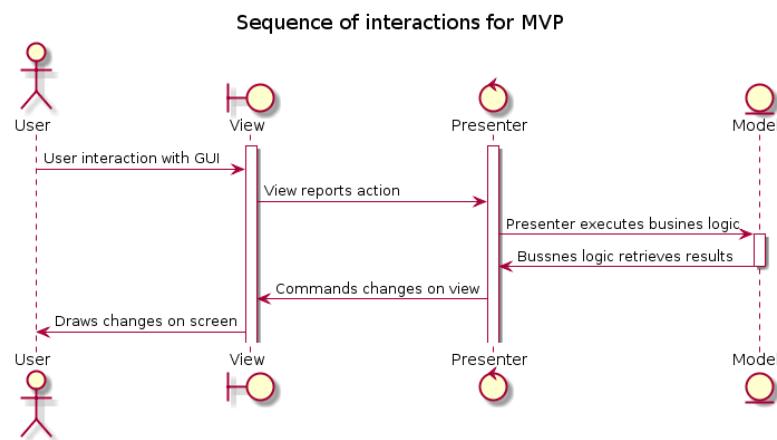


FIGURA 3.7: Diagrama de secuencia para una interacción con el usuario utilizando MVP.

### 3.2.7. Domain Layer: Commander Pattern

El patrón de diseño conocido como Commander se utiliza para abstraer la ejecución de procedimientos mediante la implementación de entidades comando [6]. Estos objetos ejecutan un único algoritmo y encapsulan la lógica de negocio de la aplicación o sistema. Originalmente el diseño contempla 4 entidades principales que se pueden apreciar en el diagrama de clases de la figura 3.8:

1. Cliente: Este componente se encarga de crear las instancias de cada comando y distribuirlas entre los correspondientes invocadores.
2. Receptor: Es la entidad que se ve afectada por la ejecución de un comando. Puede ser compartida por varios comandos o bien un único comando puede interactuar con varios receptores en su ejecución.
3. Comando: Esta entidad contiene la implementación del algoritmo o lógica de ejecución.
4. Invocador: Se encarga de ejecutar instancias de comandos.

### Command Pattern

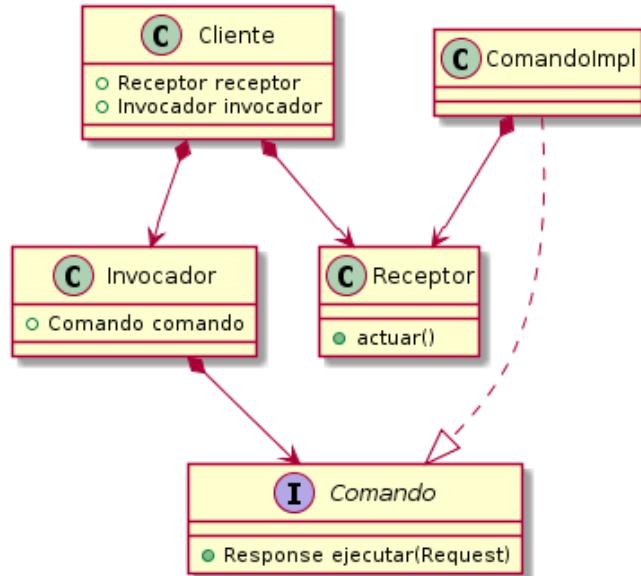


FIGURA 3.8: Diagrama de clases para el planteo inicial del patrón Commander.

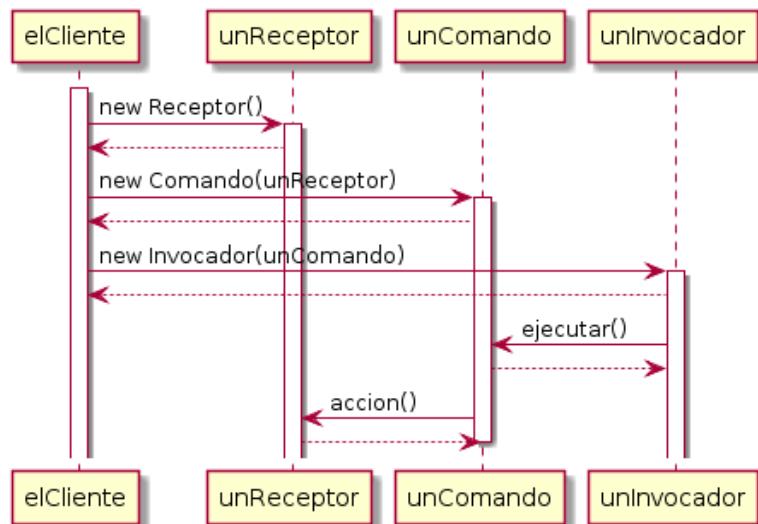


FIGURA 3.9: Diagrama de secuencia para el patrón Commander.

El enfoque inicial sugiere la implementación de un comando por cada una de las operaciones soportadas por el sistema o aplicación. Sin embargo en sistemas suficientemente grandes la diversidad de funcionalidades soportadas es tan grande que el diseño propuesto se vuelve impráctico. Para mitigar este problema se suele implementar de manera adicional una modificación que permite la ejecución paramétrica de los comandos para reducir al máximo la cantidad de comandos implementados. Esta modificación permite diversas alternativas de implementación pero la más utilizada es incorporar conceptos del patrón Request-Response dónde el caso de uso se trata como una entidad de caja negra que admite Solicitudes y emite Respuestas estandarizadas para cada caso.

- Solicitud (Request): Un objeto que contiene el conjunto de parámetros de entrada que deben ser satisfechos para poder realizar la ejecución de la rutina del comando.
- Respuesta (Response): Un objeto que contiene los valores que se obtuvieron de la ejecución del algoritmo del comando.

Por lo tanto puede inferirse el flujo de operación y ejecución de los comandos.

1. El cliente crea instancias de comandos y sus correspondientes invocadores.
2. El invocador crea e inicializa los objetos solicitud necesarios para ejecutar cada comando.
3. El invocado ejecuta los comandos llamando al método "ejecutar" implementado por cada comando pasando como parámetro la solicitud previamente creada.
4. El invocador observa los resultados en espera activa implementando el patrón Observer o mediante algún esquema de callback.

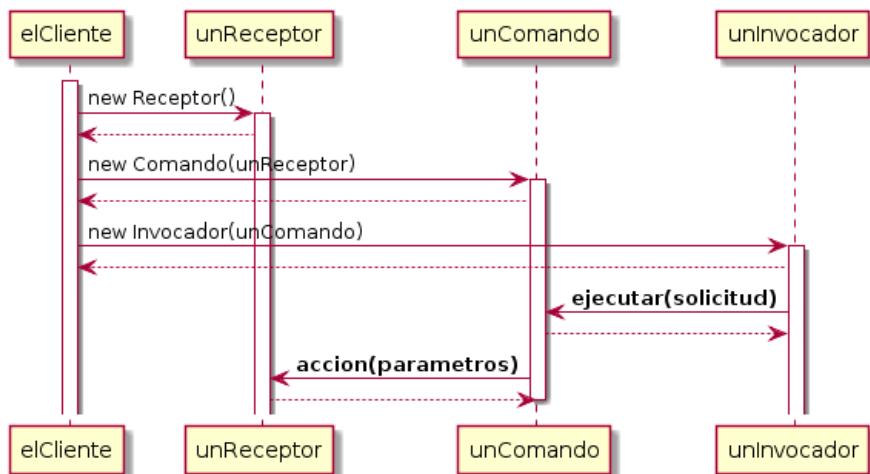


FIGURA 3.10: Diagrama de secuencia para el diseño revisado.

Siguiendo los lineamientos de la arquitectura propuesta los autores denominan a los comandos: Casos de Uso, ó Interactores.

Como una nota relevante de implementación se recomienda ejecutar las rutinas de los comandos en un hilo/proceso separado para evitar bloquear el proceso principal de la aplicación.

### 3.2.8. Data Layer: Repository Pattern

En la capa de datos se propone la implementación de un patrón de diseño conocido como Repository(Repositorio). Originalmente se concibe a este diseño como una forma de estandarizar la implementación y el uso de los objetos DAO (Data Access Object) comúnmente utilizados para mapear objetos entidad con las persistencias en la base de datos [7]. Adicionalmente este patrón encapsula en la clase repositorio todos los métodos particulares de filtrado, procesamiento calculado y ordenamiento de entidades. Sin embargo se define una interfaz genérica que deberá ser respetada por todas las implementaciones de repositorios para todo el sistema independientemente de la entidad que atienda. En el diagrama de clases de la figura 3.11 se puede apreciar el diseño original del patrón.

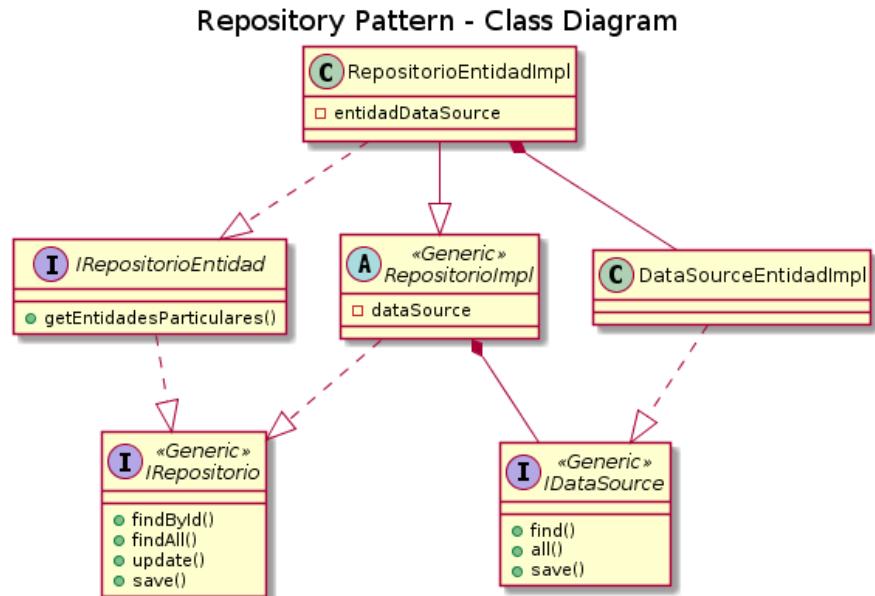


FIGURA 3.11: Diagrama de clases del patrón Repository.

Como puede apreciarse en el diagrama se definen:

- IRepository: es una interfaz genérica que establece el contrato básico que deben respetar todas las implementaciones de repositorios.

- **RepositoryImpl:** es una clase genérica que establece la interacción con una fuente de datos genérica.
- **IRepositoryEntidad:** es la interfaz que *Especifica* la interfaz genérica de repositorio y establece el contrato o métodos particulares que deberá cumplir la implementación concreta de repositorio para esta Entidad en particular.
- **RepositoryEntidadImpl:** es la clase que *Especifica* la implementación genérica de repositorio e implementa los métodos particulares para esta Entidad en particular.

En una repaso más detallado del diagrama puede observarse que existen dos interfaces genéricas para acceso de datos IRepository y IDatasource, esto podría generar confusión y generar duplicación de código. Adicionalmente en cualquier implementación moderna de sistemas con persistencia es prácticamente mandatorio el empleo de frameworks que soportan ORM (Object Relational Mapping) out-of-the-box. En la figura 3.12 se puede observar la modificación sobre la propuesta original del patrón de diseño.

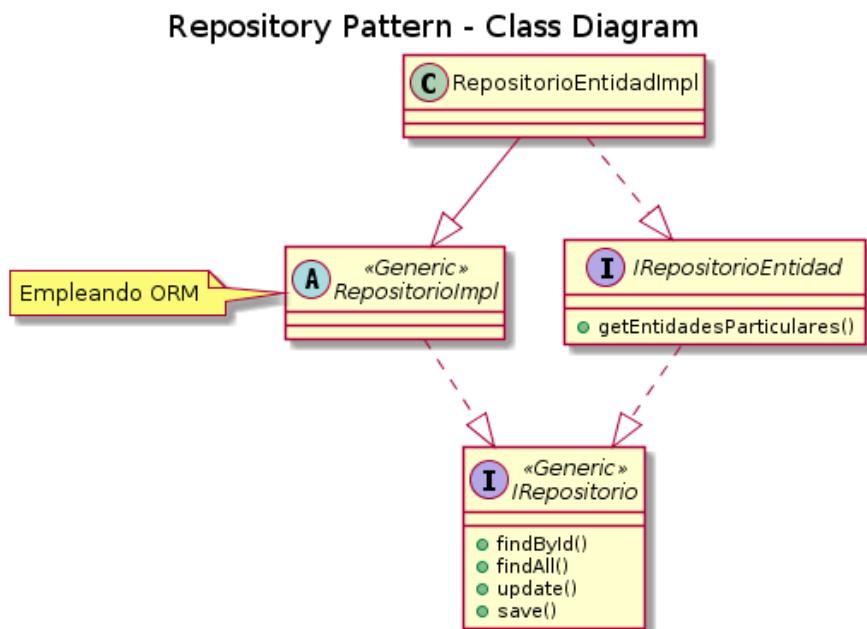


FIGURA 3.12: Diagrama de clases detallado del patrón Repository.

Por lo antes expuesto parece razonable plantear una fusión entre el concepto de repositorio y fuente de datos. Coloquialmente es fácil de entender ya que un repositorio definitivamente es una fuente de datos. Si además se quita la estandarización por genéricos se consigue un diseño más sencillo y que genera menos código estructural o scaffold manteniendo un único contrato o interfaz de acceso a los datos. En la figura 3.13 se puede observar la modificación mencionada y el diseño final propuesto para la implementación del patrón.

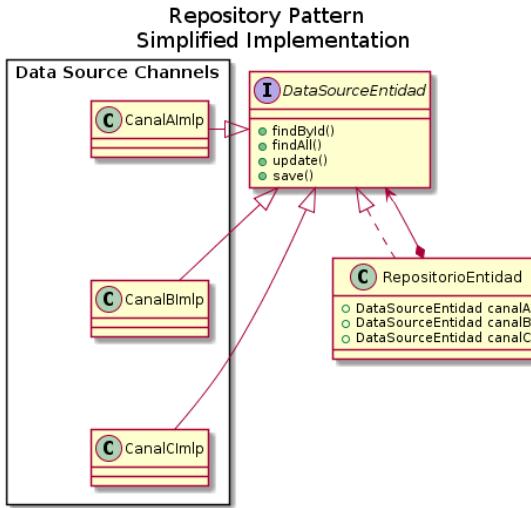


FIGURA 3.13: Diagrama de clases del patrón Repository modificado.

Principalmente orientado a encapsular la manipulación, selección, priorización y mantenimiento de diversas fuentes u orígenes de datos, este esquema de repositorios modificado permite que el peticonario se comunique con una única interfaz para solicitar operaciones sobre datos permaneciendo agnóstico del origen de datos sobre el cual tendrán impacto. Implementar una política de caching local se convierte en una tarea sencilla de implementar y mantener. Esta es la implementación del patrón que se observó en los códigos estudiados.

### 3.3. Programación Reactiva

La programación reactiva es un paradigma de programación basado en la gestión de flujos de datos asíncronos (streams) y en la propagación del cambio. Este paradigma está enfocado en el trabajo con flujos de datos finitos o infinitos de manera asíncrona, permitiendo que estos datos se propaguen generando cambios en la aplicación, es decir, "reaccionan" a los datos ejecutando una serie de eventos. Su principal característica es el uso de llamadas asíncronas no bloqueantes siempre que sea posible. Esto incluye no sólo las habituales llamadas a recursos muy lentos a través de la red, sino a todo aquello que sea posible, como las llamadas a base de datos, la gestión de peticiones y en general todo el flujo de llamadas. La mayoría de los lenguajes de programación más populares soportan este paradigma de programación a través de la inclusión de librerías estándar que implementan internamente el patrón de diseño Observer para la definición de los objetos emisores de eventos (streams) y los consumidores o observadores (subscribers) que son notificados cada vez que se produce un evento. Adicionalmente suelen implementar el patrón Iterator para convertir colecciones de datos en flujos asíncronos o streams.

### 3.3.1. Patrón Observer

Según este patrón, hay un sujeto que es el productor de la información (stream) y por otro lado hay uno o varios consumidores de esta información. En java, por ejemplo, el sujeto sería el objeto Observable y el consumidor el objeto Suscriber. Los observables son los encargados de propagar la información y notificar sus cambios, para ello proporciona métodos a partir de los cuales los consumidores pueden suscribirse o cancelar la suscripción de sus flujos de datos. Los consumidores, por su parte, deciden cuándo quiere suscribirse o cancelar la suscripción a un sujeto, además, ellos mismos son los encargados de actualizar su propio estado cuando el sujeto les notifica de un cambio en el stream de datos. En la figura 3.14 se muestra el diagrama de clases del patrón observer.

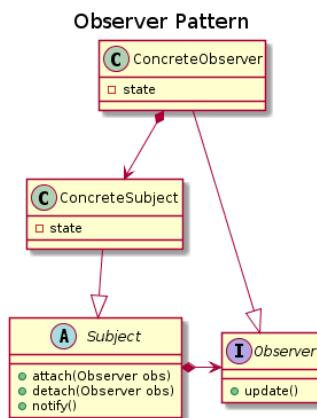


FIGURA 3.14: Diagrama de clases del patrón Observer.

### 3.3.2. Patrón Iterator

El patrón iterador nos permite recorrer contenedores de información, por ejemplo un arreglo o una lista de objetos sin necesidad de conocer el tipo de contenido o el tamaño de la colección de objetos. En la figura 3.15 se muestra el diagrama de clases tradicional de su implementación.

### 3.3.3. Programación Reactiva y Clean Architecture

Aplicando programación reactiva es posible observar un sentido deliberado del flujo de datos al considerar la dependencia entre las capas definidas por la arquitectura y sus respectivos componentes. En la figura 3.16 se puede apreciar este flujo de dato en las flechas azules que indican el intercambio de datos. Las flechas rojas indican las invocaciones.

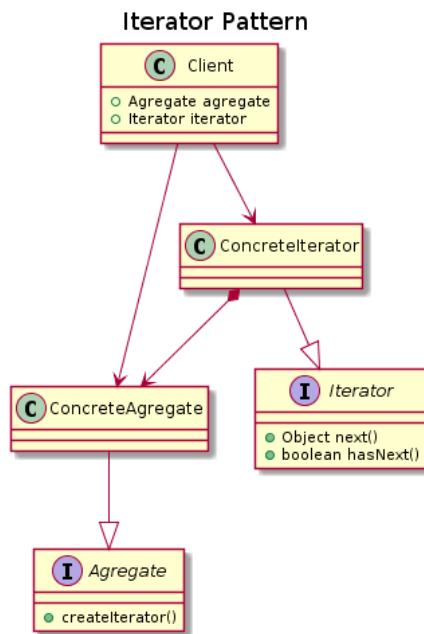


FIGURA 3.15: Diagrama de clases del patrón Iterador.

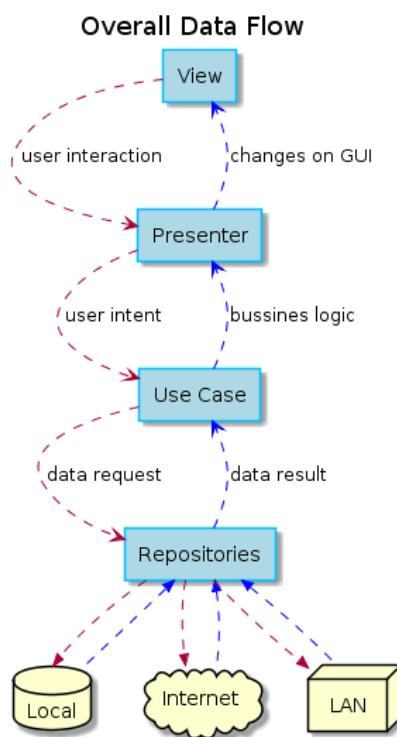


FIGURA 3.16: Diagrama de clases del patrón Repository modificado.



# Capítulo 4

## Diseño

### 4.1. Dominio del Problema

#### 4.1.1. Definición de Casos de Usos y Escenarios

#### 4.1.2. Definición de Requerimientos Funcionales y de Sistema

### 4.2. Arquitectura de la Aplicación

#### 4.2.1. Módulos y Paquetes

#### 4.2.2. Estructura de Capas

#### 4.2.3. Inyección de Dependencias

#### 4.2.4. Objetos Android y Ciclo de Vida

### 4.3. Mitigación de Errores

#### 4.3.1. Debugging

#### 4.3.2. Logging Remoto

### 4.4. Entorno de Trabajo

#### 4.4.1. Objetos Falsos

#### 4.4.2. Entorno de Pruebas

#### 4.4.3. Broker MQTT local

## Capítulo 5

# Conclusiones

Durante el desarrollo de la presente práctica se entendieron las ventajas de implementar un sistema de software utilizando un patrón de arquitectura. La inspección de código escrito por profesionales del área introdujo conceptos de programación Java avanzados tales como el uso de clases anónimas [8] y la implementación de clases y métodos genéricos[9]. Se pudieron observar las técnicas empleadas para realizar las pruebas sobre el código implementado y fue necesario invertir tiempo en la investigación de las librerías y framework para pruebas (UnitTest, Integration Tests) tales como Mockito [10] (creación de objetos mock(maquetas), stubs(comportamiento forzado) y spies (espías))y Espresso [11] (Simulación e interacción con objetos del framework android).

Luego de estudiar las implementaciones se fueron evidenciando los beneficios de aplicar una arquitectura de estas características. La modularización en componentes con responsabilidades reducidas y bien definidas permite seguir el flujo de ejecución del código con facilidad y como consecuencia directa el rastreo de bugs reduce el radio de ubicación del código involucrado a unas pocas líneas en muy poco tiempo. Dado que la lógica de negocios está encapsulada en la capa de dominio, su ejecución es completamente independiente de los componentes del framework ofreciendo la posibilidad de exportar/traducir la lógica a otros lenguajes, frameworks y sistemas operativos. Tanto los presentadores como las vistas tienen contratos que deberían respetarse en cualquier plataforma por lo que el planteo inicial de la capa de presentación permite el desarrollo en paralelo de implementaciones nativas. El uso de una abstracción de repositorios en la capa de datos permite la inclusión de diversos orígenes de datos o canales que ofrecen mayor flexibilidad al momento de establecer los niveles de redundancia soportados y los esquemas de actualización disponibles. En una buena implementación, la organización del código en directorios y paquetes debería facilitar la identificación de los componentes de arquitectura y la discriminación de funcionalidades. De manera indirecta se observó que la implementación introduce un procedimiento de trabajo repetitivo tanto para la adición de nuevas funcionalidades como para la remoción de errores y la inspección del código en general.

Como una desventaja notoria se menciona la empinada curva de aprendizaje para la inclusión de nuevos miembros en un hipotético equipo de desarrollo. Así mismo se hizo evidente que todos los conceptos de abstracción que fueron introducidos se traducen en un aumento notable en la cantidad de líneas de código meramente dedicadas a mantener la estructura del diseño pero que no proveen una funcionalidad concreta al sistema. Finalmente, se observaron inconsistencias entre el planteo teórico de la arquitectura y la implementación real del software mayormente por la dificultad técnica y concesiones que se tuvieron en cuenta para disminuir la verbosidad de algunos componentes o interacciones (e.g. la violación de la regla de dependencias en los casos de uso).

# Bibliografía

- [1] Robert C. Martin (Uncle Bob). The clean architecture, 2012. URL <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.
- [2] Tomislav Homan (FIVE). Android Architecture whole series, 2016. URL <https://five.agency/android-architecture-part-1-every-new-beginning-is-hard/>.
- [3] Fernando Cejas. Architecting android...the clean way?, 2014. URL <https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/>.
- [4] Android Team. Android architecture blueprints [beta] - mvp + clean architecture, 2015. URL <https://github.com/android/architecture-samples>.
- [5] Antonio Leiva. Mvp for android: how to organize the presentation layer, 2018. URL <https://antonioleiva.com/mvp-android/>.
- [6] James Sugrue. Java anonymous class, 2010. URL <https://dzone.com/articles/design-patterns-command>.
- [7] Wolfgang Ofner. Repository and unit of work pattern, 2018. URL <https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/>.
- [8] Pankaj. Java anonymous class, 2018. URL <https://www.journaldev.com/12534/java-anonymous-class>.
- [9] Cecilio Álvarez Caules. Uso de java generics, 2014. URL <https://www.arquitecturajava.com/uso-de-java-generics/>.
- [10] Fabian Pfaff (Vogella) Lars Vogel. Unit tests with mockito, 2018. URL <http://www.vogella.com/tutorials/Mockito/article.html>.
- [11] Maksim Akifev. Android automation testing: Getting started with espresso, 2018. URL <https://medium.com/@akifev/android-automation-testing-getting-started-with-espresso-a6f8cb50746a>.