PENERAPAN DESIGN PATTERN MVVM DAN CLEAN ARCHITECTURE PADA PENGEMBANGAN APLIKASI ANDROID (STUDI KASUS: APLIKASI AGREE)

Disusun Oleh:

N a m a	: Arief Rahman Fajri
NIM	: 18523092

PROGRAM STUDI INFORMATIKA – PROGRAM SARJANA
FAKULTAS TEKNOLOGI INDUSTRI
UNIVERSITAS ISLAM INDONESIA

2022

HALAMAN PENGESAHAN DOSEN PEMBIMBING

PENERAPAN DESIGN PATTERN MVVM DAN CLEAN ARHCITECTURE PADA PENGEMBANGAN APLIKASI ANDROID (STUDI KASUS: APLIKASI AGREE)

TUGAS AKHIR JALUR MAGANG

Disusun Oleh:

N a m a : Arief Rahman Fajri

NIM : 18523092

Yogyakarta, 15 Agustus 2022

Pembimbing,

(Septia Rani, S.T., M.Cs.)

HALAMAN PENGESAHAN DOSEN PENGUJI

PENERAPAN DESIGN PATTERN MVVM DAN CLEAN ARHCITECTURE PADA PENGEMBANGAN APLIKASI ANDROID (STUDI KASUS: APLIKASI AGREE)

TUGAS AKHIR JALUR MAGANG

Telah dipertahankan di depan sidang penguji sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer dari Program Studi Informatika – Program Sarjana di Fakultas Teknologi Industri Universitas Islam Indonesia

Yogyakarta, 15 Agustus 2022

Tim Penguji

Septia Rani, S.T., M.Cs. _____

Anggota 1

Dhomas Hatta Fudholi, S.T., M.Eng., Ph.D. _____

Anggota 2

Andhika Giri Persada, S.Kom., M.Eng. _____

Mengetahui,

Ketua Program Studi Informatika – Program Sarjana

Fakultas Teknologi Industri

Universitas Islam Indonesia

(Dr. Raden Teduh Dirgahayu, S.T., M.Sc.)

HALAMAN PERNYATAAN KEASLIAN TUGAS AKHIR

Yang bertanda tangan di bawah ini:

Nama : Arief Rahman Fajri

NIM : 18523092

Tugas akhir dengan judul:

PENERAPAN DESIGN PATTERN MVVM DAN CLEAN ARCHITECTURE PADA PENGEMBANGAN APLIKASI ANDROID (STUDI KASUS: APLIKASI AGREE)

Menyatakan bahwa seluruh komponen dan isi dalam tugas akhir ini adalah hasil karya saya sendiri. Apabila di kemudian hari terbukti ada beberapa bagian dari karya ini adalah bukan hasil karya sendiri, tugas akhir yang diajukan sebagai hasil karya sendiri ini siap ditarik kembali dan siap menanggung risiko dan konsekuensi apapun.

Demikian surat pernyataan ini dibuat, semoga dapat dipergunakan sebagaimana mestinya.

Yogyakarta, 15 Agustus 2022

(Arief Rahman Fajri)

HALAMAN PERSEMBAHAN

Dengan penuh rasa syukur penulisan tugas akhir ini saya persembahkan kepada yang terlibat dalam penulisan ini baik secara langsung maupun tidak langsung.

1. Kepada Allah SWT karena atas rahmat dan karunia-Nya saya dapat menyelesaikan tugas akhir ini.
2. Ayah dan Ibu yang selalu memberi dukungan, doa, nasehat, serta motivasi baik secara moril maupun materil. Terima kasih Ayah dan Ibu atas semua yang telah kalian berikan. Semoga Ayah dan Ibu diberi kesehatan dan umur yang panjang agar kalian bisa melihat aku sukses di kemudian hari.
3. Dosen pembimbing saya Ibu Septia Rani S.T., M.Cs. yang telah membimbing saya dalam menyelesaikan tugas akhir ini.
4. Teman-teman yang selalu mendukung baik dalam keadaan susah maupun senang. Terima kasih sudah menemani saya dalam berproses untuk menjadi orang yang lebih baik. Semoga kita bisa bertemu lagi di kemudian hari dengan jalan sukses kita masing-masing.

HALAMAN MOTO

"Allah tidak membebani seseorang melainkan sesuai kesanggupannya"

(QS. Al-Baqarah ayat 286)

"Sesungguhnya Allah tidak akan mengubah keadaan suatu kaum, kecuali mereka mengubah keadaan mereka sendiri"

(QS Ar-Ra'd ayat 11)

"Jangan terlalu ambil hati dengan ucapan seseorang, kadang manusia punya mulut tapi belum tentu punya pikiran"

(Albert Einstein)

"Sukses adalah guru yang buruk. Sukses menggoda orang yang tekun ke dalam pemikiran bahwa mereka tidak dapat gagal"

(Bill Gates)

"The object of education is to prepare the young to educate themselves throughout their lives"

(Robert Maynard Hutchins)

KATA PENGANTAR

Puji dan syukur saya ucapkan kehadirat Allah SWT atas rahmat dan karunia-Nya saya dapat menyelesaikan tugas akhir saya yang berjudul "**Penerapan *Design Pattern* MVVM Dan *Clean Architecture* Pada Pengembangan Aplikasi Android (Studi Kasus: Aplikasi Agree)**".

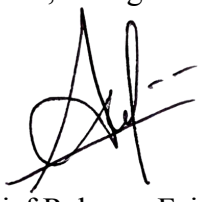
Tujuan pelaksanaan magang adalah agar saya mendapatkan ilmu, pengalaman bekerja, serta keahlian yang dibutuhkan oleh industri-industri teknologi. Penulisan tugas akhir ditujukan untuk memenuhi syarat akademik dalam menyelesaikan Program Pendidikan Strata-1 (S1) Program Studi Informatika di Universitas Islam Indonesia.

Selama proses penulisan tugas akhir ini saya mendapatkan bantuan dari berbagai pihak. Saya ingin menyampaikan rasa terima kasih saya kepada:

1. Allah SWT, yang telah memberikan rahmat dan hidayatnya sehingga laporan tengah dapat diselesaikan.
2. Kedua orang tua yang selalu memberikan dukungan berupa moril dan materil, serta do'a yang selalu diberikan.
3. Bapak Dr. Raden Teduh Dirgahayu, S.T., M.Sc., selaku Ketua Program Studi Informatika Universitas Islam Indonesia.
4. Ibu Septia Rani, S.T., M.Cs., selaku Dosen Pembimbing selama program magang.
5. Bapak Sang Made Naufal, selaku supervisor yang membimbing dan membantu penulis selama magang.
6. Segenap karyawan Agree yang sudah membantu dan membimbing selama melaksanakan magang.
7. Rekan-rekan magang penulis yang saling membantu dan memberikan semangat selama melaksanakan magang.
8. Teman-teman baik di dalam lingkungan Universitas Islam Indonesia ataupun di luar Universitas Islam Indonesia yang sudah membantu penulis menyelesaikan laporan tengah magang.
9. Kontrakan Wisma Adisty yang mendukung dan memberikan semangat kepada penulis untuk menyelesaikan laporan tugas akhir.

Atas bantuan dari berbagai pihak, penulis dapat menyelesaikan tugas akhir ini. Semoga tugas akhir ini dapat memberikan manfaat bagi pembaca. Penulis menyadari bahwa masih ada kekurangan di laporan tengah ini, oleh karena itu penulis terbuka untuk kritik dan saran.

Yogyakarta, 15 Agustus 2022



(Arief Rahman Fajri)

SARI

Indonesia merupakan negara yang memiliki lahan pertanian yang luas. Meskipun demikian, pemanfaatan teknologi pada bidang pertanian masih sedikit, padahal petani membutuhkan pengetahuan dan informasi mengenai perkembangan pasar, perkembangan harga, teknologi untuk produksi, dan juga manajemen penjualan. Untuk membantu mengatasi permasalahan ini, PT Telkom Indonesia berinovasi dengan mengembangkan aplikasi Agree. Agree memiliki tugas untuk menghubungkan semua *stakeholder* yang berperan di sektor pertanian ke dalam suatu ekosistem digital. Agree Partner yang merupakan sub bagian dari aplikasi Agree, memiliki berbagai fitur untuk mendukung produktivitas petani. Aplikasi ini berbasis Android dan membantu petani untuk dapat bermitra dengan perusahaan.

Laporan akhir ini bertujuan untuk membahas implementasi *design pattern* MVVM dan *Clean Architecture* pada pengembangan aplikasi Agree Partner, yang meliputi: struktur *package* yang menggunakan *Clean Architecture*, cara penerapan *design pattern* MVVM dan *Clean Architecture* pada pengembangan aplikasi Android, serta *Dependency Injection* agar *Clean Architecture* dapat berjalan dengan baik. Dengan penerapan *design pattern* MVVM dan *Clean Architecture*, diperoleh hasil yaitu fitur-fitur pada Agree Partner berjalan dengan baik tanpa kendala. Selain itu, dari sisi kualitas proyek, kode menjadi lebih rapi, mudah dibaca, dan mudah dirawat.

Kata kunci: Aplikasi Agree Partner, *Design Pattern* MVVM, *Android Clean Architecture*

GLOSARIUM

Compile	Proses memeriksa kode dan mengubahnya ke bahasa mesin.
Developer	Seseorang yang bertugas mengembangkan suatu sistem.
Retrofit	Library untuk mempermudah pertukaran data melalui REST API.
Room	Library dari Google untuk mempermudah menggunakan SQLite.
Scrum	Kerangka kerja yang digunakan untuk pengembangan suatu produk.
User interface	Tampilan antarmuka pengguna.

DAFTAR ISI

DAFTAR TABEL

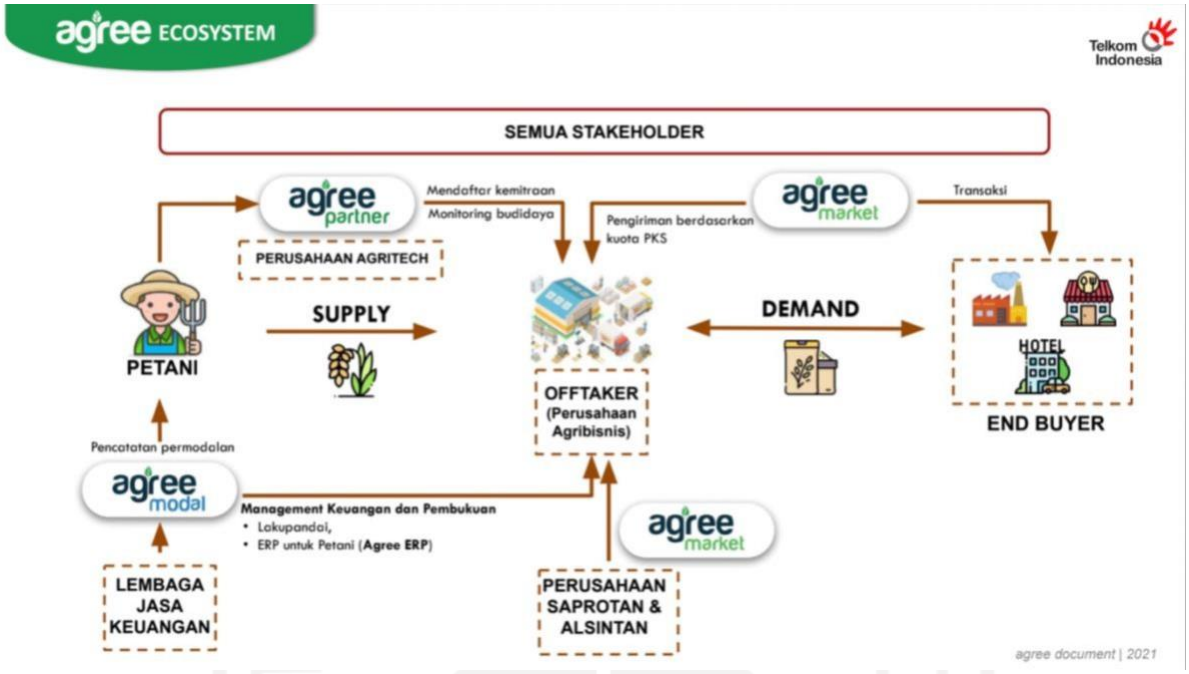
DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1 Latar Belakang

Indonesia merupakan negara yang kaya dengan sumber daya alamnya mulai dari kekayaan hutan hingga lautan. Luas wilayah daratan yang dimiliki Indonesia mencapai 2.01 juta km^2 (*KKP | Kementerian Kelautan Dan Perikanan*, n.d.). Dengan luas wilayah daratan yang luas dan subur, Indonesia juga memiliki lahan pertanian yang luas yang terdiri dari berbagai jenis lahan dengan total luas lahan 36.817.086 ha (Kementerian Pertanian, 2020). Oleh karena itu Indonesia menjadi salah satu negara dengan lahan pertanian yang luas di dunia. Akan tetapi, pemanfaatan teknologi informasi dan komunikasi pada sektor pertanian di Indonesia masih kurang dikarenakan rendahnya tingkat pendidikan dan banyak petani yang buta huruf (Catur Yuantari dkk., 2016). Padahal petani memerlukan pengetahuan dan informasi mengenai berbagai topik, seperti perkembangan pasar, perkembangan harga, teknologi untuk produksi, manajemen penjualan, dan lain-lain.

Berdasarkan masalah tersebut Agree dibangun atas inisiatif PT Telkom Indonesia yang mulai berinovasi untuk mengembangkan ekosistem digital di semua sektor, termasuk sektor pertanian. Agree memiliki tugas utama untuk menghubungkan semua *Stakeholder* yang berperan di sektor pertanian ke dalam suatu ekosistem digital. *Stakeholder* tersebut antara lain petani, agribisnis, *buyer*, pemerintah, lembaga jasa keuangan, pemerintah, dan penyedia jasa saprotan. Ekosistem pertanian digital ini diharapkan mampu membawa manfaat bagi semua *Stakeholder*, baik secara bisnis, operasional, maupun sosial. Ekosistem pertanian digital yang dibangun oleh Agree dapat dilihat pada Gambar 1.1.



Gambar 1.1 Ekosistem Digital Agree

Agree memiliki tiga produk, yaitu Agree Partner, Agree Market, dan Agree Modal. Agree Partner mengusung model kemitraan petani dengan perusahaan dengan fitur *monitoring*, permodalan dan penjualan. Produk-produk tersebut berbasis *website* dan *mobile*. Aplikasi-aplikasi Agree yang berbasis *mobile* (Android) dikembangkan menggunakan Android Studio sebagai *intergated developlment enivornment* (IDE) dan menggunakan bahasa pemrograman Kotlin.

Aplikasi-aplikasi Agree yang berbasis *mobile* (Android) memiliki banyak fitur dan tampilan, sehingga termasuk aplikasi yang kompleks. Agree memiliki banyak *developer* untuk mengembangkan aplikasi Android yang mereka miliki, serta terdapat juga *developer* magang yang berasal dari berbagai universitas di Indonesia. Jika dalam pengembangannya tidak menerapkan *design pattern* dan arsitektur, maka kode akan menjadi tidak konsisten dan sulit untuk dibaca dan dipahami oleh *developer* lain. Pembagian pekerjaan kepada *developer* juga menjadi sulit, sehingga kemungkinan terjadinya konflik saat pengembangan menjadi besar. *Developer* akan kesulitan untuk melakukan pembaruan kode apabila terjadi perubahan pada proses bisnis, serta akan mengalami kesulitan untuk melakukan pengujian dan perawatan pada kode. Dengan demikian, penting untuk menerapkan suatu arsitektur dan *design pattern* pada pengerjaan aplikasi Android.

Untuk menghindari hal tersebut, Agree menerapkan *design pattern Model View ViewModel* (MVVM) dan *Clean Architecture*. Penerapan *design pattern* MVVM memungkinkan mempertahankan data dari perubahan konfigurasi yang terjadi pada smartphone pengguna,

serta mempermudah akses ke suatu data. *Design pattern* MVVM juga membagi kode menjadi beberapa komponen, yaitu *Model*, *View*, dan *ViewModel*. Sementara itu, *Clean Architecture* merupakan sebuah arsitektur yang ditujukan agar kode pada suatu proyek menjadi lebih terstruktur dan rapi, dengan membagi konsentrasi kode menjadi beberapa *layer*.

1.2 Ruang Lingkup

Program magang berlangsung selama enam bulan di Telkom Indonesia, khususnya di Direktorat *Digital Bussiness*. Penulis ditempatkan di *Tribe* yang bernama Agree sebagai seorang Android *developer*. *Tribe* merupakan sekumpulan orang yang mengembangkan suatu produk. Tugas Akhir ini akan berfokus pada proyek Migrasi. Proyek Migrasi adalah pengembangan aplikasi Agree Partner menggunakan SDK Android Studio dan bahasa pemrograman Kotlin untuk menggantikan aplikasi Agree Partner yang sudah ada dan dikembangan menggunakan React Native. Pembahasan akan berfokus kepada pengembangan fitur *Inbox* dengan menerapkan *design pattern* MVVM dan *Clean Architecture* pada proyek Migrasi aplikasi Agree Partner.

1.3 Tujuan

Berdasarkan latar belakang di atas, laporan magang ini ditulis untuk mencapai tujuan sebagai berikut.

a. Menjelaskan cara penerapan *design pattern Model View ViewModel* (MVVM) dalam pengembangan aplikasi Agree Partner.
b. Menjelaskan cara penerapan *Clean Architecture* dalam pengembangan aplikasi Agree Partner.

1.4 Manfaat

Berdasarkan tujuan diatas, laporan akhir ini diharapkan dapat digunakan sebagai:

a. Dokumen cara penerapan *Clean Architecture* pada pengembangan aplikasi *mobile* menggunakan Android Studio dan bahasan pemrograman Kotlin.
b. Dokumen cara penerapan *design pattern* MVVM pada pengembangan aplikasi *mobile* menggunakan Android Studio dan bahasan pemrograman Kotlin.

1.5 Sistematika Penulisan

Adapun tugas akhir ini memiliki sistematika penulisan sebagai berikut.

a. BAB I membahas tentang latar belakang, ruang lingkup magang, tujuan, dan manfaat penulisan dari Penerapan *Design Pattern* MVVM dan *Clean Architecture*.

b. BAB II membahas tentang dasar-dasar teori yang berkaitan dengan *desing pattern* MVVM dan *Clean Architecture*.

c. BAB III menjelaskan tentang pelaksanaan magang, mulai dari *on-boarding* sampai dengan penerapan dari teori-teori yang telah dijelaskan pada bab sebelumnya.

d. BAB IV menjelaskan refleksi pembelajaran magang baik dari sisi teknis dan *non*-teknis.

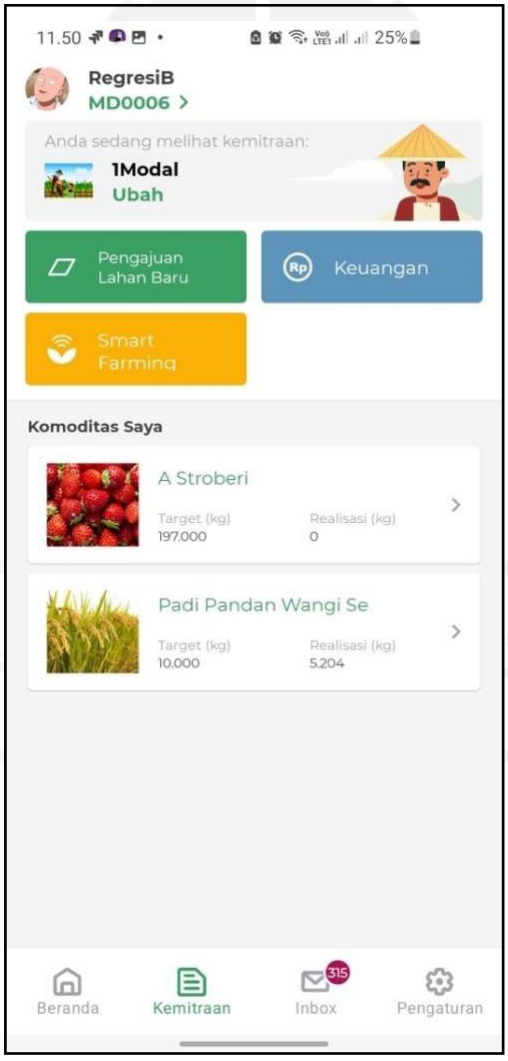
e. BAB V membahas tentang kesimpulan manfaat yang telah diperoleh dari pelaksanaan magang, serta saran dan kritik dari penulis berdasarkan pengalaman yang penulis rasakan selama mengikuti magang.

BAB II
LANDASAN TEORI DAN TINJAUAN PUSTAKA

2.1 Aplikasi Agree Partner

Agree Partner merupakan aplikasi yang membantu petani untuk dapat bermitra dengan perusahaan yang telah terdaftar di Agree. Petani dapat mencari perusahaan agrikultur yang sesuai dengan komoditas yang dimiliki dan mengajukan kemitraan dengan perusahaan tersebut. Petani dapat melaporkan progress aktivitas mereka kepada perusahaan melalui aplikasi. Pada aplikasi Agree Partner terdapat berbagai fitur, seperti Permodalan, Kemitraan, dan lain-lain.

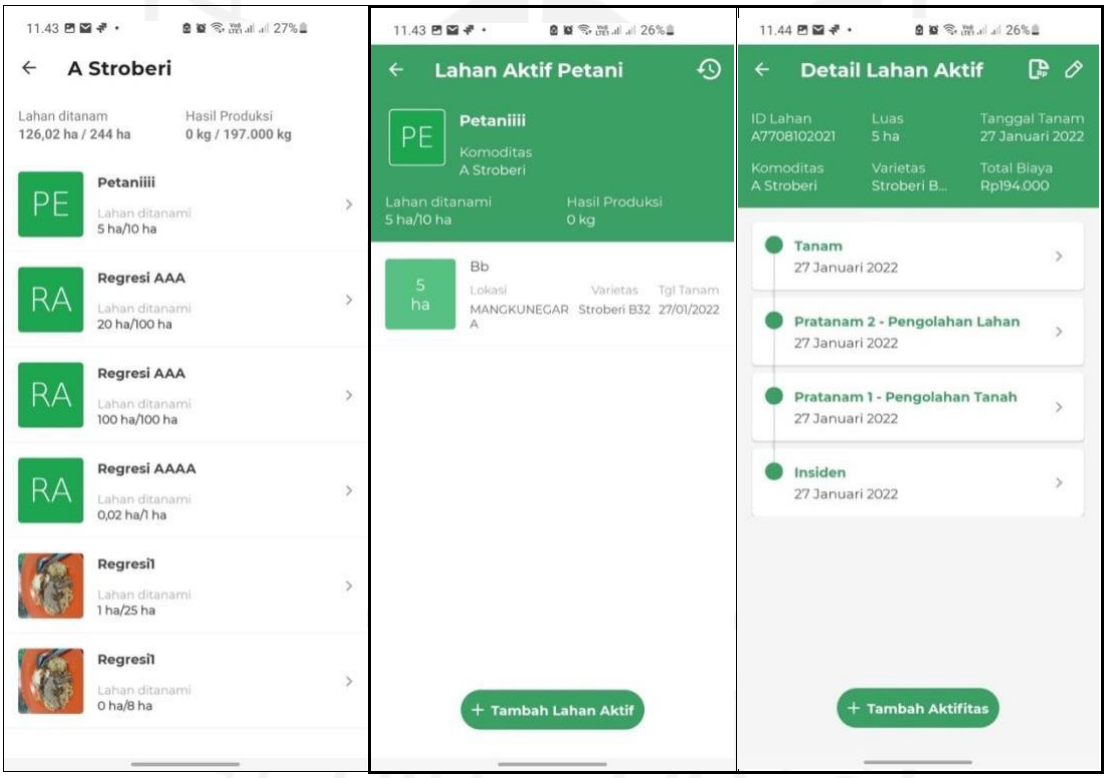
Gambar 2.1 merupakan tampilan fitur Kemitraan di aplikasi Agree Partner. Fitur ini akan muncul apabila petani sudah bermitra dengan salah satu perusahaan Agrikultur.



Gambar 2.1 Fitur Kemitraan

Pada fitur ini mitra dapat mengajukan penanaman komoditas di lahan baru, kemudian dapat juga mengajukan peminjaman permodalan, serta dapat melihat daftar komoditas yang telah disetujui oleh perusahaan Agrikultur.

Ketika petani melakukan klik pada daftar komoditas, maka petani akan dibawa ke halaman fitur Lahan Aktif yang dapat dilihat pada Gambar 2.2. Pada fitur lahan aktif, setiap komoditas dapat dikerjakan lebih dari satu petani, dan setiap petani bisa memiliki lebih dari satu lahan untuk setiap komoditas. Di fitur ini petani dapat melakukan penambahan lahan dan melakukan *update* aktivitas yang dilakukan. Fitur ini bertujuan agar petani dapat melaporkan progres yang sedang dikerjakan terhadap suatu lahan komoditas. Perusahaan dapat memantau aktivitas lahan komoditas tersebut melalui website Agree Dashboard yang dikhususkan untuk perusahaan yang bermitra dengan Agree saja.



Gambar 2.2 Fitur Lahan Aktif

2.2 Android

Dikutip dari (Maya, 2017), Android adalah sistem operasi berbasis Linux yang dirancang dan dibuat untuk perangkat bergerak layar sentuh atau lebih dikenal di masyarakat dengan sebutan *smartphone*. Android pertama kali dikembangkan oleh Android, Inc., dengan Google sebagai pendukung finansial. Pada tahun 2005 Google membeli Android dan merilis sistem

operasi Android secara resmi pada tahun 2007. Ponsel Android pertama kali dijual secara umum pada bulan Oktober 2008.

Dikutip dari (Maya, 2017), Android menerima *input* dari pengguna dengan tindakan sentuhan langsung ke layar perangkat. Sentuhan-sentuhan tersebut dapat berupa ketuk, geser, dan cubit, serta untuk menerima *input* teks, Android akan menampilkan *keyboard* virutal yang dapat digunakan dengan cara diketuk. Selain digunakan pada *smartphone*, Google telah mengembangkan AndroidTV untuk televisi, Android Auto untuk mobil, dan Android Wear untuk perangkat *wearable* seperti jam tangan.

Android adalah sistem operasi dengan sumber terbuka memungkinkan perangkat lunak untuk dimodifikasi secara bebas dan didistribusikan oleh para pembuat perangkat, pengembang aplikasi, nirkabel, dan operator (Maya, 2017). Untuk pengembangan aplikasi berbasis Android digunakan Android Studio sebagai IDE yang direkomendasikan oleh Google dengan bahasa pemrograman Java atau Kotlin. Dalam pengembangan aplikasi Android, Google merekomendasikan *Model View ViewModel* sebagai *design pattern*. Selain *design pattern*, Google juga merekomendasikan agar arsitektur aplikasi dibagi menjadi tiga layer, yaitu UI *Layer*, *Domain Layer*, dan *Data Layer*.

2.2.1 Android Studio

Android Studio adalah IDE resmi untuk pengembangan aplikasi Android yang diperkenalkan ke publik pada tahun 2013 dan dirilis resmi pada tahun 2014. Android Studio diimplementasikan sebagai plugin di Intellij IDEA yang merupakan sebuah Java IDE yang dibuat oleh Jetbrains.

Android Studio merupakan sebuah perubahan penting bagi *developer*. Pertama, karena *developer* dapat migrasi dari Eclipse ke Android Studio yang dirancang khusus untuk Java *developer*. *Developer* juga dapat menggunakan fitur seperti *fast and impressively smart code completition* atau *analysing* dan *refactor tools* yang *powerful*. Kedua, Gradle menjadi *build system* resmi untuk Android, yang berarti banyak kemungkinan yang berakitan dengan pembuatan dan penerapan versi. Dua fungsi paling menarik adalah *build system* dan *flavours*, dimana memungkinkan *developer* untuk membuat versi aplikasi yang tak terbatas (atau bahkan aplikasi yang berbeda) dengan cara yang mudah dengan menggunakan satu basis kode saja.

2.2.2 Kotlin

Kotlin adalah bahasa pemrograman berbasis Java Virtual Machine (JVM) yang dikembangkan oleh JetBrains yanng dikenal juga sebagai sebuah perusahaan yang membuat Intellij IDEA. Intellij IDEA merupakan *Integrated Development Environment* (IDE) untuk Java. Android Studio yang merupakan IDE resmi untuk pengembangan aplikasi Android yang berbasis Intellij.

Kotlin dibuat dengan mempertimbangkan pengembangan Java dan dengan Intellij sebagai IDE utamanya. Berikut dua fitur menarik untuk pengembang aplikasi Android.

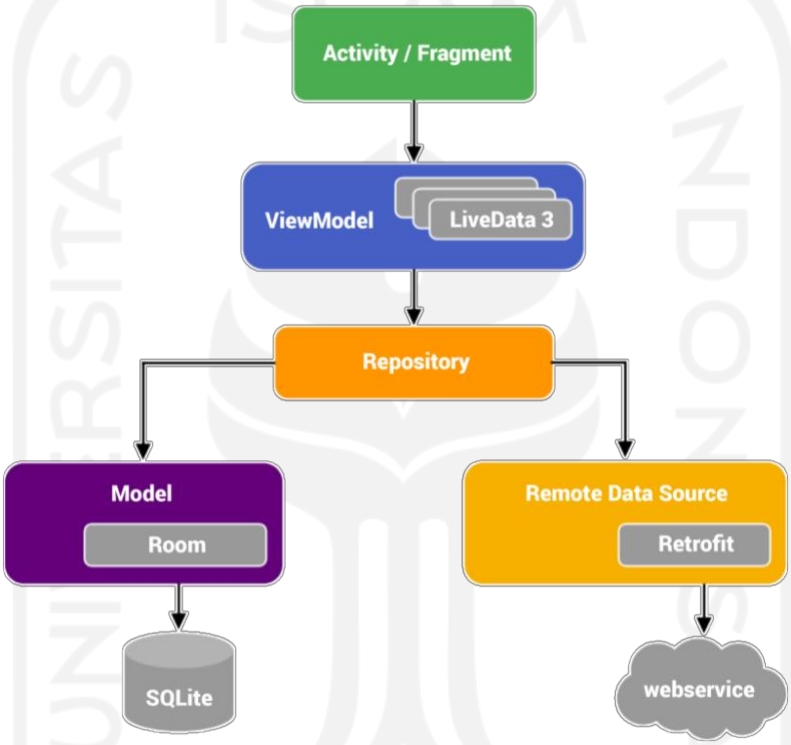
a. Kotlin sangat intuitif dan mudah dipelajari oleh Java *developer* dikarenakan sebagian besar bahasanya mirip dengan Java dan hanya berbeda pada konsep dasarnya.
b. Android Studio dapat mengerti, meng-*compile*, dan menjalankan kode Kotlin. Dukungan untuk bahasa Kotlin di Android Studio berasal dari perusahaan yang mengembangkan IDE.

Kelebihan yang dimiliki Kotlin dibandingkan dengan Java, yaitu:

a. Kotlin lebih ekspresif: *developer* bisa menulis lebih dengan kode yang sedikit.
b. Lebih aman: karena memiliki fitur *null safe*, yang berarti *developer* dapat mengantisipasi kemungkinan terjadinya situasi *null* ketika waktu kompilasi, untuk mencegah *exception* ketika kode dieksekusi. *Developer* harus secara eksplisit menentukan apakah suatu objek dapat bernilai *null* dan kemudian memeriksa *nullity*-nya sebelum menggunakan objek tersebut.
c. *Functional*: Kotlin pada dasarnya merupakan bahasa pemrograman berorientasi objek, bukan bahasa pemrograman fungsional murni. Namun, Kotlin menggunakan banyak konsep dari pemrograman fungsional, seperti *lambda expressions*, untuk menyelesaikan beberapa masalah dengan cara yang lebih mudah.
d. *Extension function*: berarti *developer* dapat meng-*extend* kelas apapun dengan fitur baru meskipun *developer* tidak memiliki akses ke kode sumbernya.
e. *Highly interoperable*: *developer* dapat menggunakan libraries dan kode yang ditulis dengan Java. Karena interoperabilitas antara kedua bahasa sangat baik. Bahkan dimungkinkan untuk membuat proyek campuran dengan bahasa Java dan Kotlin, dengan *file* Kotlin dan Java yang berdampingan.

2.2.3 Android Architecture Components

Android *Architecture Components* merupakan *pattern* yang dikeluarkan oleh Google. Android *Architecture Components* adalah kumpulan *library* yang membantu untuk merancang aplikasi yang kuat, dapat diuji, dan dapat dikelola dengan mudah (*Komponen Arsitektur Android | Developer Android | Android Developers*, n.d.). Berikut Gambar 2.3 merupakan gambaran dari *Android Architecture Components.*



Gambar 2.3 *Android Architecture Components*
Sumber: (*Dicoding Indonesia*, n.d.-a)

Gambar 2.3 merupakan gambaran besar dari *Android Architecture Components*. Berikut penjalasan dari *Android Architecture Components*.

a. *Activity*/*Fragment* berfungsi sebagai *userinterface controller* yang berfungsi untuk menampilkan data dan sebagai *input* aksi dari *user*. *Activity*/*Fragment* berinteraksi dengan *ViewModel* melalui perantara *LiveData* (*Dicoding Indonesia*, n.d.-a).
b. *ViewModel* berguna sebagai pusat komunikasi antara *Repository* dan *Activity*/*Fragment. ViewModel* mengambil data melalui *Repository*. *ViewModel* juga

berfungsi untuk mempertahankan data dari *configuration changes* (*Dicoding Indonesia*, n.d.-a)

c. *LiveData* adalah kelas yang memegang data dan selalu memegang atau menyimpan data versi terbaru. *LiveData* dapat diobservasi dan memberi tahu *observers* jika ada perubahan data (*Dicoding Indonesia*, n.d.-a).
d. *Repository* digunakan untuk mengelola banyak data yang dapat bersumber dari *network* dan *local*. Jika data bersumber dai *local*, maka *Repository* akan menggunakan Room untuk berinteraksi dengan SQLite. Apabila data berasal dari *remote*, maka *Repository* akan berinteraksi dengan *webservice* menggunakan Retrofit (*Dicoding Indonesia*, n.d.-a).
e. Room merupakan *library* yang mempermudah dalam melakukan *query* pada database SQLite di Android (Sutabri, 2016).
f. Retrofit adalah *library* yang mempermudah proses pertukaran data antara aplikasi Android dengan *webservice* melalui REST API (*Dicoding Indonesia*, n.d.-a).

Menurut (*Dicoding Indonesia*, n.d.-a), Gambar 2.3 menunjukkan *flow* bagaimana sebuah *Activity/Fragment* melakukan permintaan data kepada *ViewModel*. Kemudian *ViewModel* meneruskan permintaan tersebut ke *Repository*. Di dalam *Repository*, aplikasi dapat mengetahui apakah data berasal dari *local* atau *network*. Setelah *Repository* mendapatkan data, kemudian data dikirimkan ke *ViewModel*. Ketika terjadi perubahan data, *View* akan otomatis melakukan pembaharuan tampilan berdasarkan data yang diterima.

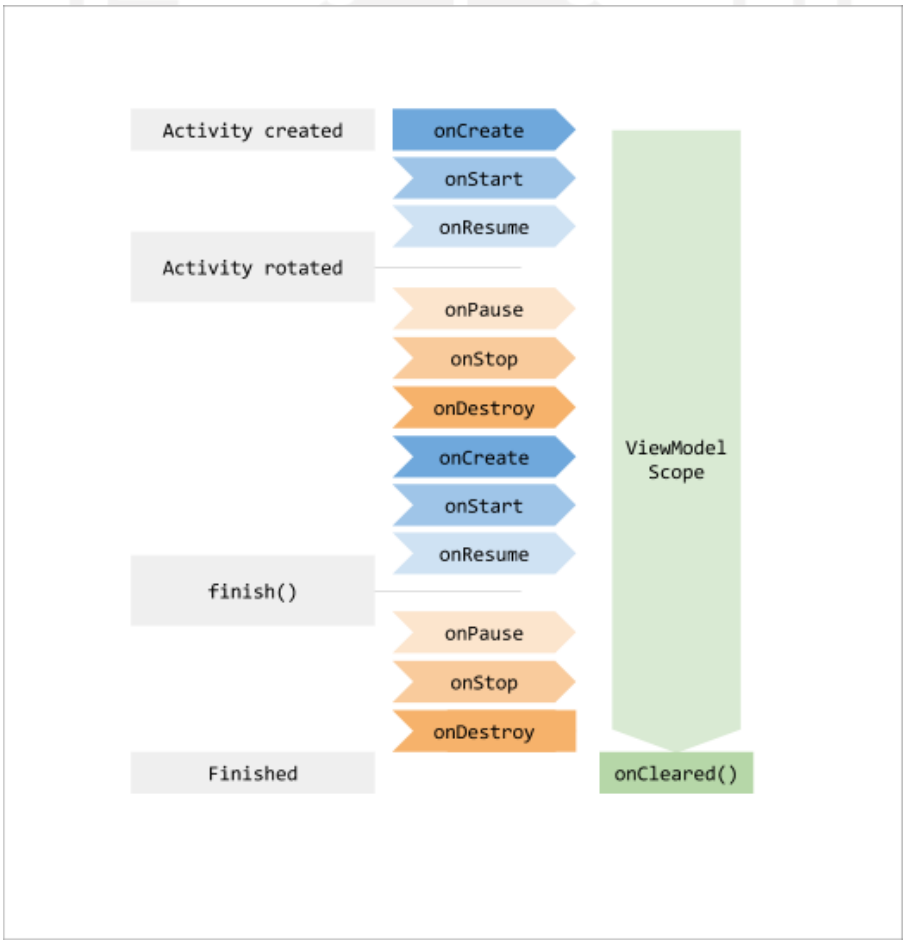
2.2.4 Model View ViewModel

Model View ViewModel pattern diperkenalkan oleh John Gossman (*Software Architect at Microsoft for Windows Presentation Foundation and Silverlight technologies*) pada tahun 2005 di blog pribadinya. MVVM adalah evolusi dari *Presentation Model* (PM) *pattern* yang diperkenalkan Martion Fowler pada tahun 2004.

Salah satu tujuan utama PM *pattern* adalah memisahkan dan mengabstraksi tampilan UI dari *presentation logic* agar UI dapat diuji. Adapun tujuan lain dari PM *pattern* adalah membuat *presentation logic* dapat digunakan oleh UI yang berbeda, serta mengurangi keterikatan antara UI dan kode lainnya.

Model View ViewModel (MVVM) merupakan sebuah *design pattern* yang memiliki tiga komponen, yaitu *Model*, *View*, dan *ViewModel* (Tian Lou, 2016). Berikut adalah penjelasan ketiga komponen tersebut.

a. *Model*, merupakan representasi dari data yang digunakan pada *business logic*. *Model* dapat berupa *Plain Old Java Object* (POJO) atau *Kotlin Data Classes* (Sutabri, 2016).
b. *View* adalah komponen yang terdiri dari *layout resource file* dan *Activity*/*Fragment*. Tampilan pada *layout resource file* akan dikontrol melalui *Activity*/*Fragment* secara dinamis (Tian Lou, 2016).
c. *ViewModel* akan berinteraksi dengan *Model* dan menyiapkan *variable observables* yang akan diobservasi oleh *View*. *ViewModel* bersifat *lifecycle-aware* yang dapat dilihat pada Gambar 2.4 dimana kelas ini akan hidup ketika sebuah kelas *View* telah melalui tahapan *create* dan belum melalui tahapan *destroy*. Kelas *View* hanya akan melakukan satu kali pemanggilan data saja dan data akan dipertahankan di *ViewModel Scope* selama kelas *View* belum melalui tahapan *destroy*. Jika kelas *View* telah melewati tahapan *destroy*, maka data yang ada di *ViewModel* akan dibersihkan (Sutabri, 2016).

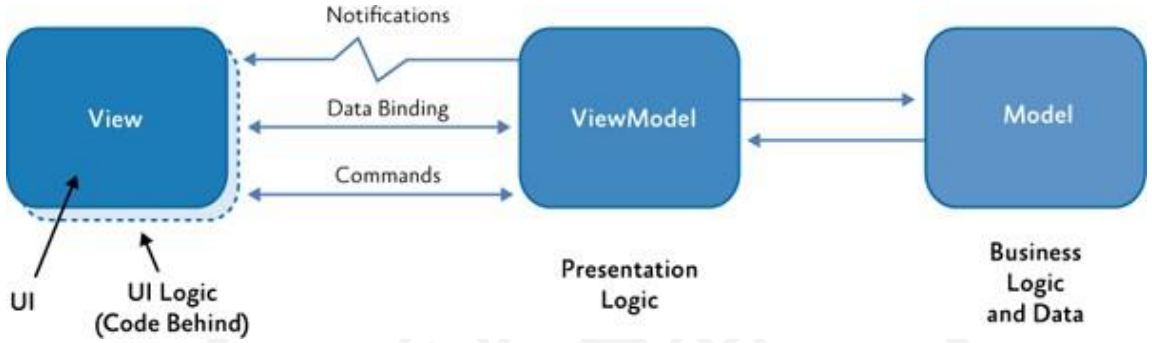


Gambar 2.4 Siklus Hidup *ViewModel*

Sumber: (*Ringkasan ViewModel | Developer Android | Android Developers*, n.d.)

Dikutip dari (*5: Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF | Microsoft Docs*, n.d.), interaksi setiap komponen MVVM dapat dilihat pada Gambar 2.5,

dimana komponen *View* yang terdiri dari UI dan UI *logic*, komponen *ViewModel* yang menangani *presentation logic*, dan komponen *model* yang menangani bisnis *logic* dan data.



Gambar 2.5 Ilustrasi Interkasi MVVM

Sumber: (Sutabri, 2016)

Komponen *View* berinteraksi dengan komponen *ViewModel* melalui *databinding*, *commands*, dan *change notification events* dengan perantara *live data* sebagai *observable data holder* yang dapat diobservasi oleh komponen *View*. Komponen *ViewModel* mengamati dan mengkoordinasikan pembaruan ke *Model*, melakukan konversi, memvalidasi, dan menggabungkan data yang akan ditampilkan di komponen *View*.

2.3 Clean Architecture

(R. Martin, 2017) Selama beberapa dekade telah banyak ide yang membahas tentang arsitektur sistem, atara lain sebagai berikut.

a. *Hexagonal Architecture* atau dikenal juga sebagai *Ports and Adapters*, dikembangkan oleh Alistair Cockburn, dan diadopsi oleh Steve Freeman dan Nat Pryce di buku mereka yang berjudul *Growing Object Oriented Software with Tests*.
b. DCI dari James Coplien dan Trygve Reenskaug.
c. BCE, dikenalkan oleh Ivar Jacobson melalui bukunya yang berjudul *Object Oriented Software Engineering: A Use-Case Driven Approach*.

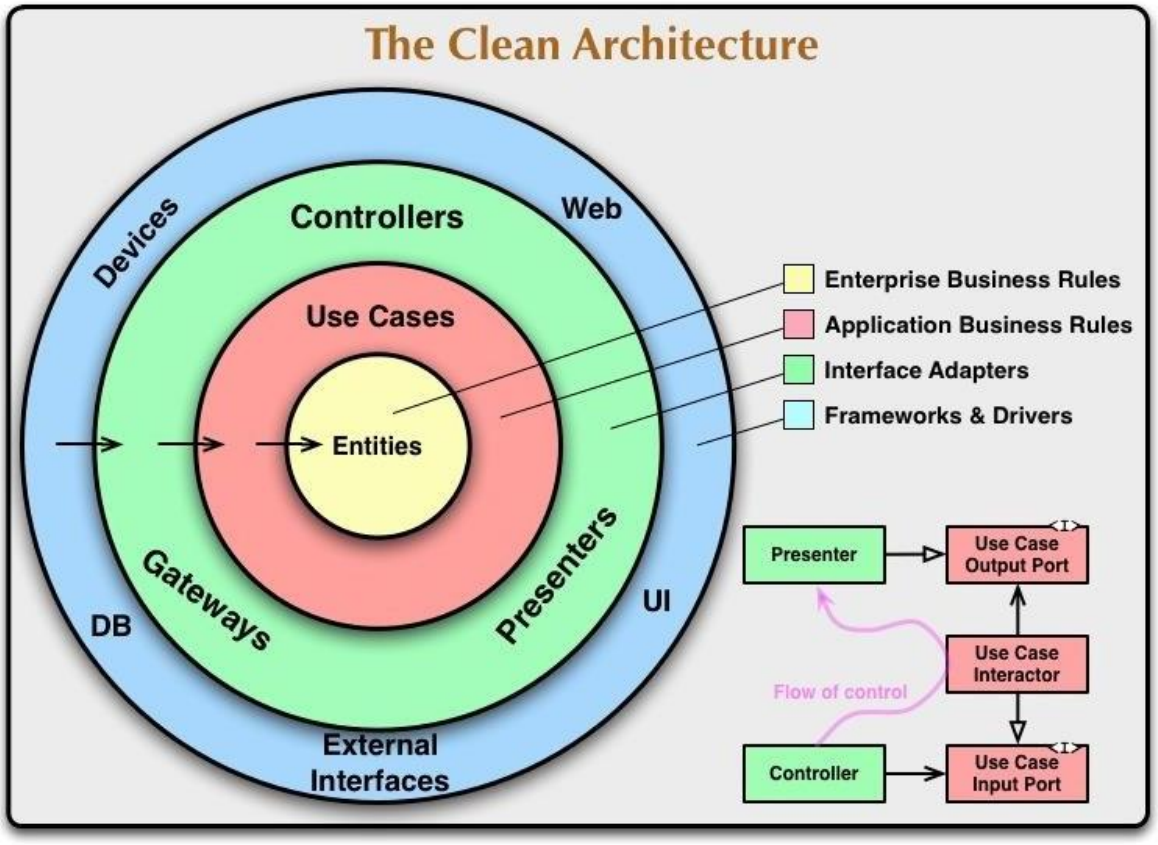
Meskipun setiap arsitektur diatas memiliki detail yang berbeda-beda, tetapi mempunyai tujuan yang sama, yaitu pemisahan kepentingan. Pemisahan kepentingan ini dicapai dengan cara membagi *software* menjadi beberapa *layer*. Setiap arsitektur tersebut setidaknya memiliki satu *layer* untuk *business rules*, dan *layer* lainnya untuk *user* dan *system interfaces*.

(R. Martin, 2017) Masing-masing sistem tersebut menghasilkan sebuah sistem yang memiliki karakteristik sebagai berikut.

a. *Independent of Framework,* tidak tergantung pada implementasi *framework* yang digunakan.

b. *Testable,* kode proses bisnis bisa dites tanpa memerlukan UI, *database*, atau *tools* lainnya.
c. *Independent of Userinterface*, UI dapat diubah dengan mudah tanpa harus mengubah keseluruhan sistem.
d. *Independent of Database,* tidak bergantung dengan *framework database* tertentu dan dapat diganti dengan mudah.
e. *Independent of External,* proses bisnis yang ada tidak perlu mengetahui apa yang ada di *layer* luar.

Gambar 2.6 mengilustrasikan jenis-jenis *layer* dan *data flow* di *Clean Architecture*. *Entities layer* bertanggungjawab terhadap *enterprise business rules*.



Gambar 2.6 *Clean Architecture*

Sumber: (Tung Bui Du, 2017)

(Tung Bui Du, 2017) Agar arsitektur ini bekerja, *developer* harus mematuhi *dependency rule* dan ketergantungan hanya boleh mengarah kedalam. Kode di *layer* dalam tidak boleh mengetahui kode di *layer* luar, serta tidak boleh terpengaruh oleh perubahan yang terjadi di *layer* luar. Developer harus menggunakan *dependency inversion principle* ketika berinteraksi antar *layer*.

Biasanya *entities* berbentuk sebuah *object* atau sekumpulan struktur data dan fungsi. Perubahan pada aplikasi seharusnya tidak mempengaruhi *entity layer*. *Use cases layer* bertanggungjawab terhadap *application business rules*. Perubahan pada layer ini tidak akan mempengaruhi *entities* dan juga layer ini tidak boleh terpengaruh oleh perubahan pada *external layer*. *Interface adapter layer* menyediakan cara mentransformasi format data dari *entities* dan *usecases* ke format yang sesuai untuk *external agency* seperti *database* atau *website*. Kode pada layer ini seharusnya tidak tahu tentang *framework* atau *driver* yang digunakan oleh aplikasi. *Frameworks* dan *drivers layer* terdiri dari *framework* atau *tools* seperti *database*, *web framework*, dan lain-lain.

2.3.1 Entities

(R. Martin, 2017) *Entities* mengenkapsulasi aturan bisnis di sebuah perusahaan. Sebuah *entities* dapat berupa *object* dengan *methods*, atau dapat berupa kumpulan struktur dan fungsi data. Tidak masalah bagaimana bentuk sebuah *entities* selama *entities* tersebut dapat digunakan oleh aplikasi di perusahaan.

Entities memiliki kemungkinan perubahan yang kecil jika terdapat perubahan pada *external layer*. Misalnya, perubahan navigasi atau keamanan aplikasi tidak akan mempengaruhi lapisan *Entities*.

2.3.2 Use Cases

(R. Martin, 2017) *Use cases layer* berisi tentang *appilaction-specific business rules*. *Layer* ini mengenkapsulasi dan mengimplementasikan semua *use cases* pada sistem. *Use cases* ini mengatur aliran data dari dan ke *entities*, dan mengarahkan *entities* untuk menggunakan *Critical Business Rules* untuk mencapai tujuan *use case*.

Perubahan pada *layer* ini tidak mempengaruhi *layer entities*. Layer ini juga tidak terpengaruh oleh perubahan di *external layer*, seperti database, UI, atau *framework*. Tapi perubahan *operations* pada aplikasi akan mempengaruhi *use case*. Jika *use case* berubah, makan kode pada layer ini juga akan terpengaruh.

2.3.3 Interface Adapters

(R. Martin, 2017) *Interface adapters layer* merupakan sekumpulan *adapters* yang mengonversi data dari format yang sesuai dengan *use cases* dan *entities*, ke format yang sesuai dengan *external agency*, seperti *database* atau web. Lapisan ini sepenuhnya berisi arsitektur

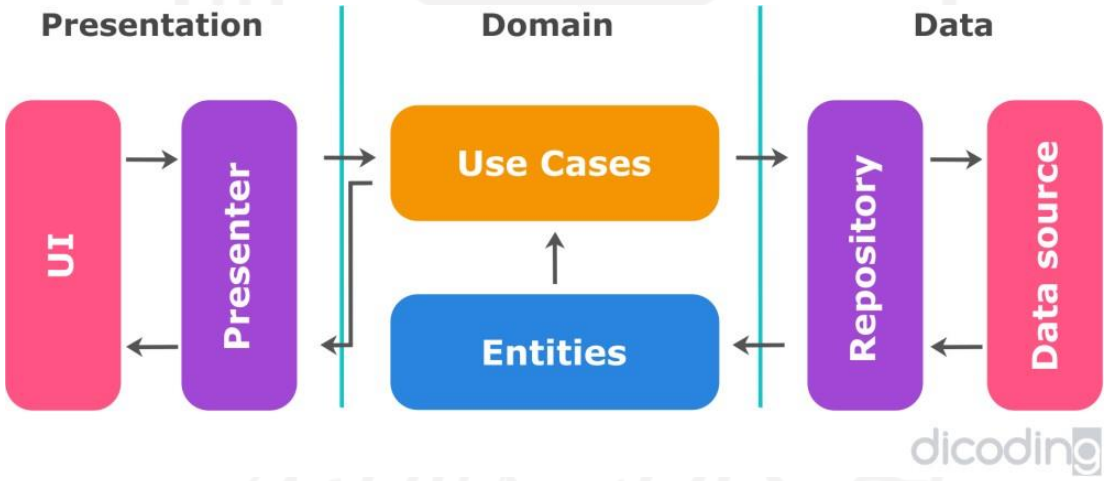
seperti MVC atau lainnya. *Presenters*, *views*, dan *controllers* semuanya termasuk dalam *layer* ini. *Model* kemungkinan hanya struktur data yang dikirim dari *controller* ke *use case*, dan kemudian dari *use case* ke *presenters* dan *views*. Pada layer ini juga dibutuhkan adapter untuk mengonversi data dari *external*, seperti *external service*, ke bentuk *internal* yang digunakan oleh *use cases* dan *entities*.

2.3.4 Framework and Drivers

(R. Martin, 2017) Lapisan terluar dari *clean arhcitecture* pada Gambar 2.6 adalah *framework* dan *drivers*. Lapisan ini pada umumnya terdiri dari *framework* dan *tools* seperti *database* atau *web framework*. Biasanya tidak banyak *code* yang ditulis pada lapisan ini, selain kode yang menghubungkan dengan lapisan yang berada di dalam.

2.3.5 Android Clean Architecture

Penggunaan *Clean Architecture* pada *proyek* Android pada umumnya dibagi menjadi tiga *layer*, yaitu *presentation*, *domain*, dan *data (Dicoding Indonesia, n.d.-b)*. Pembagian *layer* tersebut dapat dilihat pada Gambar 2.7.



Gambar 2.7 *Layer Android Clean Architecture*

Sumber: (*Dicoding Indonesia*, n.d.-b)

Berikut adalah penjelasan mengenai komponen dari setiap *layer* yang terdapat pada Android *Clean Architecture* menurut (*Dicoding Indonesia*, n.d.-b).

a. Pada *Presentation Layer* terdapat UI dan *Presenter*/*ViewModel* yang mengatur tampilan berdasarkan *update* data terbaru. *Presentation Layer* bergantung kepada Use Case di *Domain Layer*.

b. Pada *Domain Layer* terdapat *Entities*, *Use Case*, dan *Repository Interface*. *Layer* ini merupakan *layer* inti yang berkaitan dengan bisnis *model*.
c. *Data Layer* berisi implementasi *Repository* dan *Data Source* yang bisa berasal dari *local data source* (database lokal) atau *remote data source* (*network* atau REST API).

2.3.6 Dependency Inversion Principle dan *Dependency Injection*

Dependency Inversion Principle merupakan sebuah prinsip yang menyatakan bahwa sebuah sistem yang fleksibel merupakan sistem yang kodenya bergantung kepada kelas abstraksi, bukan kelas konkret. Di bahasa yang statis seperti Java, ini berarti penggunaan import harus merujuk kepada sumber modul yang berisi *interface*, *abstarct classes*, atau jenis deklarasi *abstract* lainnya.

Dikutip dari (Dicoding, n.d.) Pada prinsip *Dependency Inversion* ada dua pernyataan yang diutarakan oleh Robert C. Martin. Pernyataan pertama adalah *high-level module* tidak boleh bergantung pada *low-level module*, keduanya harus bergantung pada *abstraction*. Pernyataan kedua adalah *abstraction* tidak boleh bergantung pada *detail*, tetapi *detail* harus bergantung pada *abstraction*. *High-level modules* adalah kelas-kelas yang berurusan dengan kumpulan-kumpulan fungsionalitas. Terdapat kelas-kelas yang mengimplementasikan aturan bisnis sesuai dengan desain yang ditentukan. *Low-level modules* memiliki tanggung jawab pada operasi yang detail. Pada level terendah memungkinkan *modules* ini untuk menulis data ke *database* atau menyampaikan pesan ke sistem informasi.

Dikutip dari (Tung Bui Du, 2017)*, Dependency Injection* adalah sebuah *design pattern* yang menerapkan prinsip *inversion of control*. Pada umumnya *dependency injection* memiliki sebuah *service* yang akan digunakan oleh *client* dan sebuah *interface* yang mendefinisikan bagaimana *client* menggunakan *service* tersebut, serta sebuah *injector* yang menginstansiasi *service* tersebut dan meng-*inject service* ke *client*.

2.4 Scrum

(Schwaber & Sutherland, 2017) *Scrum* bukan merupakan proses, teknik, atau metode, melainkan kerangka kerja di mana dapat mengimplementasikan berbagai proses dan teknik. *Scrum* membuat pekerjaan tekins dan manajemen produk menjadi lebih efektif sehingga dapat terus meningkatkan kualitas lingkungan kerja, tim, dan produk.

Adapun beberapa komponen penyusun Scrum, yaitu *Scrum Team*, acara *Scrum*, artefak, dan lain-lain. Setiap komponen *Scrum* memiliki tujuannya masing-masing. Setiap tujuan tersebut memiliki pengaruh terhadap keberhasilan *Scrum*.

Empirisme merupakan dasar dari *Scrum*, di mana pengetahuan berasal dari pengalaman dan pengambilan keputusan didasarkan apa yang telah diketahui. Pendekatan yang digunakan adalah iteratif dan inkremental. Pendekatan tersebut dapat mengoptimalkan prediktabilitas dan mengendalikan resiko.

Tiga pilar yang menjunjung tinggi implementasi kontrol proses empiris, yaitu transparansi, inspeksi, dan adaptasi. Berikut adalah penjelasan transaparansi, inspeksi, dan adaptasi.

A. Transparansi adalah proses dan progress pekerjaan yang dapat dilihat oleh orang yang terlibat di pekerjaan. Artefak yang memiliki transparansi rendah dapat meningkatkan risiko dan mengurangi nilai. Transparansi memungkinkan untuk dilakukannya inspeksi. Inspeksi tanpa transparansi adalah menyesatkan dan sia-sia.
B. Inspeksi adalah pemeriksaan kemajuan artefak *Scrum* berdasarkan tujuan yang telah disepakati. Artefak harus sering diperiksa untuk mendeteksi varian atau masalah yang terjadi. Inspeksi memungkinkan adanya adaptasi, inspeksi tanpa adaptasi tidak berguna.
C. Adaptasi adalah proses penyesuaian produk jika produk yang dihasilkan tidak dapat diterima. Proses penyesuaian harus dilakukan sesegera mungkin. Suatu *Scrum Team* diharapkan dapat mempelajari hal baru ketika melakukan inspeksi.

2.4.1 Scrum Team

(Schwaber & Sutherland, 2017) *Product Owner*, *Development Team*, dan *Scrum Master* adalah komponen yang membentuk *Scrum Team*. *Scrum Team* adalah tim lintas fungsi yang terorganisir sendiri. Tim yang terorganisir sendiri memilih cara terbaik untuk menyelesaikan pekerjaannya, daripada meminta seseorang di luar tim mengarahkannya. Tim lintas fungsi memiliki semua kemampuan yang diperlukan untuk menyelesaikan pekerjaan, tanpa bergantung pada tim lain yang bukan anggota tim. *Scrum Team* dirancang untuk mengoptimalkan fleksibilitas, kreativitas, dan produktivitas.

2.4.2 Development Team

(Schwaber & Sutherland, 2017) *Development Team* berisikan orang-orang yang profesional di bidangnya. Tim ini memiliki tugas untuk menghasilkan *Icrement* yang telah memenuhi definisi selesai dan akan dirilis di akhir *Sprint*. *Development Team* membuat *Increment* yang dibutuhkan pada saat acara *Sprint Review*. Organisasi membentuk dan memberdayakan *Development Team* untuk mengatur dan memanajemen pekerjaan mereka.

2.4.3 Product Owner

(Schwaber & Sutherland, 2017) *Product Owner* memiliki tugas untuk memaksimalkan kualitas produk yang dihasilkan oleh *Development Team*. *Product Owner* adalah satu-satunya orang yang bertanggung jawab untuk mengelola *Product Backlog*. Manajemen *Product Backlog* meliputi:

A. Menjelaskan *Product Backlog Item* dengan jelas.
B. Mengatur *Product Backlog Item* untuk mencapai tujuan dan misi.
C. Mengoptimalkan nilai pekerjaan yang dilakukan oleh *Development Team*.
D. Memastikan *Development Team* memahami *Product Backlog Item*.
E. Memastikan bahwa *Product Backlog* terlihat, transparan, dan jelas bagi semua orang.

Product Owner adalah seseorang yang mewakili keinginan komite dalam *Product Backlog*. Jika ingin melakukan perubahan terkait prioritas *Product Backlog Item* harus menghubungi *Product Owner*.

2.4.4 Scrum Master

(Schwaber & Sutherland, 2017) *Scrum Master* merupakan orang yang melayani *Scrum Team*. Disaat yang bersamaan, dia juga menjadi pemimpin dari *Scrum Team*. Membantu semua orang untuk memahami teori, praktik, dan nilai *Scrum* merupakan salah satu tugas dari *Scrum Master*. *Scrum Master* melayani *Development Team* dengan cara sebagai berikut.

A. Melatih *Development Team* dalam pengorganisasian diri dan lintas fungsi.
B. Membantu *Development Team* untuk menghasilkan produk berkualitas tinggi.
C. Menghilangkan halangan demi progres *Development Team*.
D. Melatih *Development Team* menerapkan *Scrum* di lingkungan organisasi yang belum menerapkannya.
E. Memfasilitasi acara *Scrum* sesuai yang diminta atau dibutuhkan.

Scrum Master membantu *Product Owner* dengan cara sebagai berikut.

A. Memastikan bahwa tujuan, ruang lingkup, dan domain produk dipahami dengan baik oleh semua orang di *Scrum Team*.
B. Menemukan teknik yang efektif untuk memanajemen *Product Backlog*.
C. Membantu *Scrum Team* memahami apa yang dibutuhkan untuk menghasilkan *Product Backlog Item* yang jelas dan ringkas.
D. Memastikan *Product Owner* mengetahui cara menyusun *Product Backlog* untuk memaksimalkan nilai.
E. Memfasilitasi acara *Scrum* sesuai yang diminta atau dibutuhkan.

Scrum Master melayani organisasi dengan cara sebagai berikut.

A. Memimpin dan melatih organisasi dalam penerapan *Scrum*.
B. Membuat rencana penerapan *Scrum* di dalam organisasi.
C. Membantu karyawan dan *Stakeholder* memahami dan menerapkan *Scrum*, serta pengembangan produk yang empiris.
D. Melakukan suatu perubahan yang bisa meningkatkan produktivitas *Scrum Team*.
E. Berkolaborasi dengan *Scrum Master* lainnya untuk meningkatkan efektivitas penerapan *Scrum* dalam organisasi.

2.4.5 Sprint

(Schwaber & Sutherland, 2017) Inti dari *Scrum* adalah *Sprint*, *Sprint* berdurasi satu bulan atau kurang selama *Increment* produk yang memenuhi definisi selesai dapat digunakan, serta memiliki potensi untuk dirilis. Durasi dari *Sprint* bersifat konsisten selama pengembangan. Setelah *Sprint* berakhir, *Sprint* yang baru dapat dimulai.

Sprint Planning, *Daily Scrums*, *development work*, *Sprint Review*, dan *Sprint Retrospective* merupakan serangkaian acara yang menyusun *Sprint*. Perubahan yang dilakukan selama *Sprint* tidak boleh membahayakan *Sprint Goal*. Perubahan juga tidak boleh menyebabkan menurunnya kualitas dari tujuan.

Setiap *Sprint* dapat dianggap sebagai proyek dengan durasi kurang dari atau maksimal satu bulan. *Sprint* mempunyai tujuan, desain dan rencana yang fleksibel yang akan membantu pembuatan, pekerjaan, dan *product increment* yang dihasilkan. Jika suatu *Sprint* berlangsung terlalu lama, definisi yang sedang dibangun atau dikembangkan bisa berubah, kompleksitas bisa meningkat, dan risiko menjadi lebih tinggi.

Pembatalan *Sprint* bisa dilakukan sebelum *Sprint* berakhir. Pembatalan *Sprint* dilakukan oleh *Product Owner*, walaupun ia melakukannya di bawah pengaruh *Stakeholders*,

Development Team, atau *Scrum Master*. Pembatalan dilakukan jika *Sprint Goal* menjadi usang. Perubahan tujuan perusahaan, teknologi, dan pasar dapat menjadi penyebab usangnya suatu *Sprint Goal*. Tapi, pembatalan jarang dilakukan dikarenakan durasi *Sprint* yang cenderung singkat.

2.4.6 Sprint Planning

(Schwaber & Sutherland, 2017) *Sprint Planning* merupakan acara perencanaan terkait pekerjaan atau tugas yang akan dilakukan ketika *Sprint*. Kolaborasi dari seluruh anggota tim diperlukan untuk membuat rencana ini.

Development Team melakukan perkiraan terkait fungsionalitas yang akan dikembangkan ketika *Sprint*. *Sprint Goal* dan *Product Backlog Item* yang harus diselesaikan untuk mencapai *Sprint Goal* akan dijelaskan oleh *Product Owner*. *Development Team* memiliki wewenang untuk memilih *Product Backlog Item* apa saja yang akan dimasukkan ke *Sprint*.

Cara pengembangan fungsionalitas agar menjadi *Increment* yang memenuhi *Definition of Done* ditentukan oleh *Development Team*. *Product Backlog Item* yang dipilih serta rencana yang diperlukan untuk menyelesaikannya disebut *Sprint Backlog*. Jika *Product Backlog Item* yang dipilih terlalu banyak atau terlalu sedikit, *Development Team* bisa melakukan negosiasi ulang *Product Backlog Item* yang dipilih dengan *Product Owner*.

2.4.7 Sprint Goal

Setiap *Sprint* memiliki tujuan. Tujuan dari *Sprint* disebut dengan *Sprint Goal*. *Sprint Goal* bisa dicapai melalui implementasi *Product Backlog*. Penetapan *Sprint Goal* dilakukan ketika acara *Sprint Planning*. *Development Team* menjadikan *Sprint Goal* sebagai panduan mereka dalam membangun *Increment*.Saat *Development Team* bekerja, mereka harus mengingat *Sprint Goal. Sprint Goal* dapat dipenuhi dengan mengimplementasikan fungsionalitas dan teknologi.

2.4.8 Sprint Review

(Schwaber & Sutherland, 2017) Di akhir *Sprint*, *Increment* yang telah selesai dikerjakan akan dicek. Jika diperlukan, *Product Backlog* juga akan disesuaikan. Proses pengecekan dan penyesuaian tersebut disebut dengan *Sprint Review*. *Sprint Review* menghasilkan revisi *Product Backlog* untuk *Sprint* selanjutnya. Penyesuaian *Product Backlog* secara menyeluruh dilakukan jika bisa memenuhi peluang yang baru.

2.4.9 Sprint Retrospective

(Schwaber & Sutherland, 2017) Setelah *Sprint* berakhir, *Scrum Team* melakukan evaluasi terkait jalannya *Sprint*. Hasil dari evaluasi tersebut dapat dijadikan pembelajaran untuk meningkatkan kualitas *Sprint* selanjutnya. *Sprint Retrospective* dilakukan setelah *Sprint Review* dan sebelum *Sprint Planning* selanjutnya. Tujuan dari *Sprint Retrospective*, yaitu:

a. Memeriksa bagaimana jalannya Sprint terakhir.
b. Mengidentifikasi dan mengurutkan *item* yang bisa berjalan dengan baik dan memiliki potensi untuk meningkatkan kualitas produk.
c. Membuat rencana untuk mengimplementasikan perbaikan pada cara *Scrum Team* bekerja.

Scrum Team didorong untuk melakukan peningkatan proses dan praktik pengembangan agar menjadi lebih efektif untuk *Sprint* selanjutnya. *Scrum Team* membuat rencana untuk meningkatkan kualitas produk dengan meningkatkan proses kerja atau mengadaptasi *Definition of Done* yang sesuai dan tidak bertentangan dengan standar produk atau organisasi.

Pada saat *Sprint Retrospective* berakhir, *Scrum Team* telah mengidentifikasi perbaikan yang akan diimplementasikan pada *Sprint* selanjutnya. Implementasi perbaikan ini pada *Sprint* selanjutnya adalah adaptasi dari inspeksi *Scrum Team* itu sendiri.

2.4.10 Product Backlog

(Schwaber & Sutherland, 2017) *Product Backlog* adalah daftar semua yang diketahui dan dibutuhkan suatu produk. Orang yang bertanggung jawab terkait *Product Backlog* adalah *Product Owner*. *Product Backlog* tidak pernah lengkap. Perkembangan di awal menjabarkan persyaratan yang awalnya diketahui dan paling dipahami. *Product Backlog* memiliki sifat yang dinamis, ia akan terus berkembang sesuai dengan lingkungannya. Perkembangan *Product Backlog* ini bertujuan untuk melakukan identifikasi terkait kebutuhan suatu produk agar sesuai, kompetitif, dan berguna.

Saat suatu produk digunakan dan memperoleh nilai, dan pasar memberikan umpan balik, *Product Backlog* menjadi daftar yang lebih besar dan lebih lengkap. Persyaratan senantiasa berubah, jadi *Product Backlog* adalah artefak yang memiliki sifat dinamis. Perubahan pada *Product Backlog* dapat disebabkan oleh perubahan bisnis, kondisi pasar, atau teknologi.

2.4.11 Sprint Backlog

(Schwaber & Sutherland, 2017) Setiap *Sprint Backlog* memiliki *Product Backlog Item* dan rencana untuk menyelesaikannya, serta mewujudkan *Sprint Goal*. Kumpulan *Product Backlog Item* dan rencananya disebut dengan *Sprint Backlog*. *Sprint Backlog* memperlihatkan semua pekerjaan yang diidentifikasi oleh *Development Team* sebagai hal yang diperlukan untuk memenuhi *Sprint Goal*.

Development Team memodifikasi *Sprint Backlog* selama *Sprint*. Perubahan yang dilakukan pada *Sprint Backlog* akan dijelaskan pada saat *Daily Scrum*. Saat *Sprint* berjalan, ada kemungkinan munculnya *Sprint Backlog* baru. Hal ini terjadi karena *Development Team* mengetahui lebih banyak pekerjaan yang diperlukan untuk mencapai *Sprint Goal*.

Pekerjaan baru yang muncul akan ditambahakan ke *Sprint Backlog*. Saat pekerjaan dilakukan atau diselesaikan, perkiraan waktu pekerjaan yang tersisa diperbarui. *Sprint Backlog* hanya bisa diubah oleh *Development Team* ketika *Sprint* sedang berlangsung.

2.4.12 Increment

(Schwaber & Sutherland, 2017) Jumlah *Product Backlog Item* yang telah diselesaikan dalam satu *Sprint* disebut dengan *Increment*. Ketika *Sprint* berakhir, *Increment* harus selesai dan dapat digunakan, serta memenuhi *Definition of Done* dari *Scrum Team*. *Increment* adalah kumpulan pekerjaan yang dapat diperiksa dan diselesaikan yang mendukung empirisme. *Increment* harus dalam kondisi yang bisa digunakan, terlepas *Product Owner* memutuskan untuk merilisnya atau tidak.

2.5 Tinjauan Pustaka

Terdapat beberapa penelitian yang telah dilakukan terkait pengembangan aplikasi Android menggunakan Android Studio. Seperti penelitian yang dilakukan oleh (Makarenko dkk., 2017) yang berjudul "*An Architectural Approach for Quality Improving of Android Applications Development which Implemented to Communication Application for Mechatronics Robot Laboratory Onaft*". Penelitian tersebut menerapkan *Clean Architecture* dan *Model View Controller design pattern*. Pada penelitian tersebut masih menerapkan MVC *design pattern* yang mana tidak sesuai dengan rekomendasi dari Google saat ini, yaitu MVVM *design pattern*.

Penelitian yang dilakukan oleh (Bui & Technology, 2020) yang berjudul "*Android Applications for Student's Personal Finance*" menjelaskan tentang penerapan MVVM *design*

pattern pada pengembangan aplikasi Android menggunakan Android Studio dengan bahasa Kotlin. Tetapi pada penelitian ini tidak dijelaskan mengenai *Clean Architecture* dan cara penerapannya pada pengembangan aplikasi Android.

Penelitian yang berjudul "Pengembangan Aplikasi Berbasis Android Untuk Meningkatkan Kepatuhan Pasien Gagal Jantung Dalam Merawat Diri Di Rumah" oleh (Yoga dkk., 2021) ini membahas tentang pengembangan aplikasi berbasis Android, mulai dari metode pengembangan, *database*, *design pattern*, dan arsitektur aplikasi. Pada penelitian ini juga menjelaskan cara implementasi MVVM pada pengembangan aplikasi, tetapi untuk *Clean Architecture* sendiri tidak ada penjelasan cara penerapannya pada saat pengembangan.

Penelitian lainya yang dilakukan oleh (Sharma dkk., 2015) yang berjudul "*E-Agro Android Application (Integrated Farming Management Systems for sustainable development of farmers)"* merupakan penelitian tentang aplikasi android yang berfungsi untuk membantu petani menjadwalkan pemupukan. Pada penelitian ini tidak menjelaskan mengenai pengembangan secara teknis, terutama tentang arsitektur dan *design pattern* yang digunakan pada aplikasi.

Penelitian oleh (Aldiansyah dkk., 2021) yang berjudul "Pengembangan Aplikasi Ngobat: Aplikasi Ketaatan Regimen Pengobatan menggunakan Gamification pada Platform Android" adalah penelitian tentang pengembangan aplikasi yang bertujuan untuk menjadi pengingat guna membantu pengguna dalam regimen pengobatan. Pada penelitian ini dibahas mengenai design pattern MVVM yang diterapkan dalam pengmebangan aplikasi, akan tetapi pengembangan tersebut tidak menerapkan *Clean Architecture* dalam pengembangannya.

Berdasarkan penelitian di atas, dapat dilihat pada Tabel 2.1 apakah pengembangan aplikasi Android pada penelitian-penelitian tersebut telah menerapkan *Clean Architecture* dan *design pattern* MVVM.

Tabel 2.1 Arsitektur dan *Design Pattern* Aplikasi

No	Judul Penelitian	Arsitektur Aplikasi	*Design Pattern*
1	*An Architectural Approach for Quality Improving of Android Applications Development which Implemented to Communication Application for Mechatronics Robot Laboratory Onaft*	*Clean Architecture*	MVC

2	*Android Applications for Student's Personal Finance*	*Clean Architecture*	MVVM
3	Pengembangan Aplikasi Berbasis Android Untuk Meningkatkan Kepatuhan Pasien Gagal Jantung Dalam Merawat Diri Di Rumah	Tidak Diketahui	MVVM
4	Pengembangan Aplikasi Ngobat: Aplikasi Ketaatan Regimen Pengobatan menggunakan Gamification pada Platform Android	Tidak Diketahui	MVVM
5	Pengembangan Aplikasi Ngobat: Aplikasi Ketaatan Regimen Pengobatan menggunakan Gamification pada Platform Android	Tidak Diketahui	MVVM

Berdasarkan tabel Tabel 2.1 diketahui bahwa dari lima pengembangan aplikasi Android, dua diantaranya telah menerapkan arsitektur dan *design pattern*, sementara yang lainnya hanya menerapkan *design pattern*. Hanya satu dari lima penelitian tersebut yang menerapkan *Clean Arhitecture* dan *design pattern* MVVM.

BAB III
PELAKSANAAN MAGANG

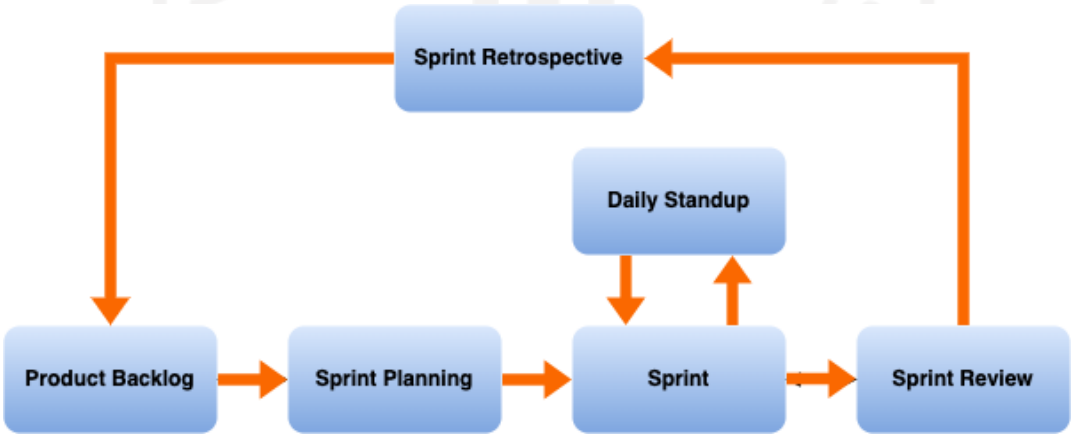
3.1 Android Developer

Program magang di Telkom Direktorat Digital Business membuka beberapa posisi, yaitu *developer*, *data scientist*, *designer*, *researcher*, *general*, *engineer*, dan *marketing*. Penulis berposisi sebagai Android *developer* yang ditempatkan di Agree dan bertugas untuk mengembangkan aplikasi *mobile* yang berbasis Android.

Di Agree, penulis mengerjakan proyek Migrasi. Proyek migrasi adalah proyek pembuatan aplikasi Agree Partner menggunakan Android *Native* untuk menggantikan Aplikasi Agree Partner yang sebelumnya sudah dikembangkan menggunakan ReactNative. Pembahasan pelaksanaan magang ini akan berfokus pada fitur *Inbox* yang penulis kerjakan pada proyek Migrasi dengan menerapkan *Clean Architecrure* dan *design pattern* MVVM.

3.2 Manajemen Proyek

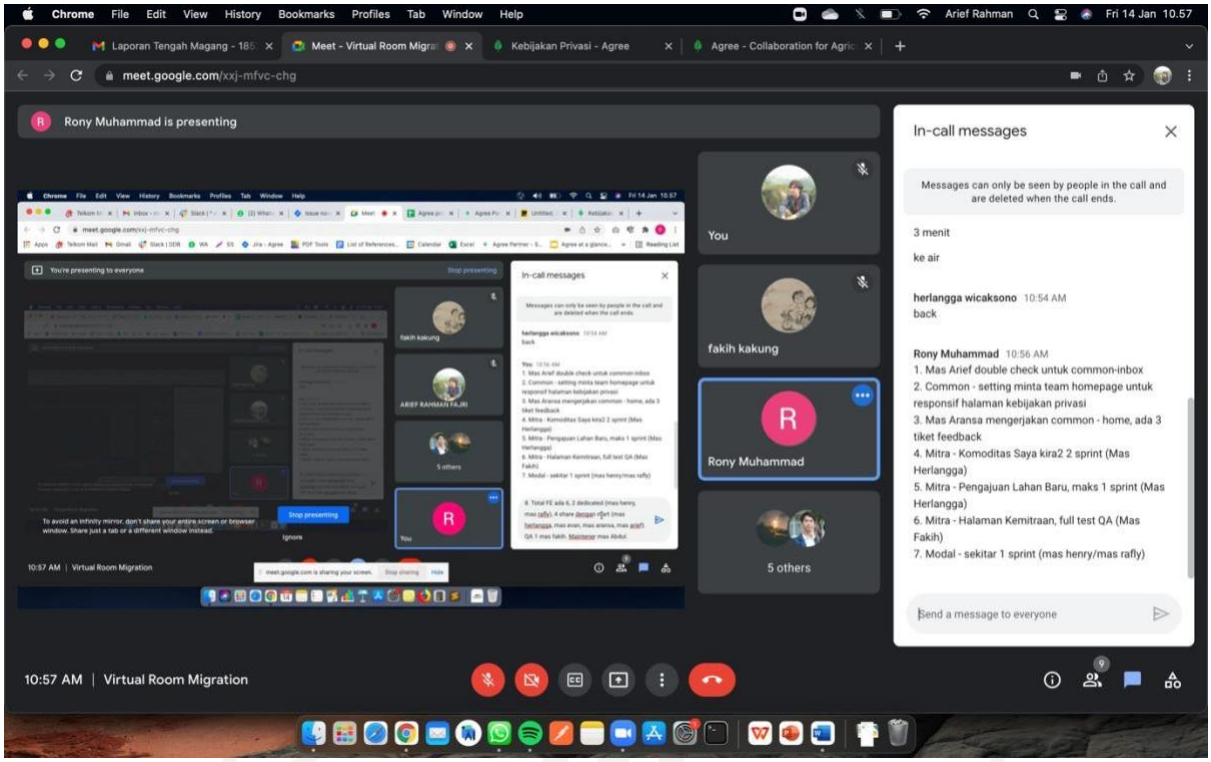
Metodologi yang digunakan dalam pengembangan aplikasi Agree Partner adalah *Scrum*. Gambar 3.1 mengilustrasikan proses pengembangan aplikasi Agree Partner.



Gambar 3.1 Tahap Pengembangan Aplikasi Agree Partner

Product Backlog merupakan daftar pekerjaan yang bertujuan untuk meningkatkan kualitas produk. *Product Owner* akan melakukan pembahasan terkait *Product Backlog* yang akan memecah *Product Backlog* menjadi *items* yang lebih kecil dan detil. Jika *Product Backlog* dianggap sudah siap, maka akan dilanjutkan ke tahap *Sprint Planning*.

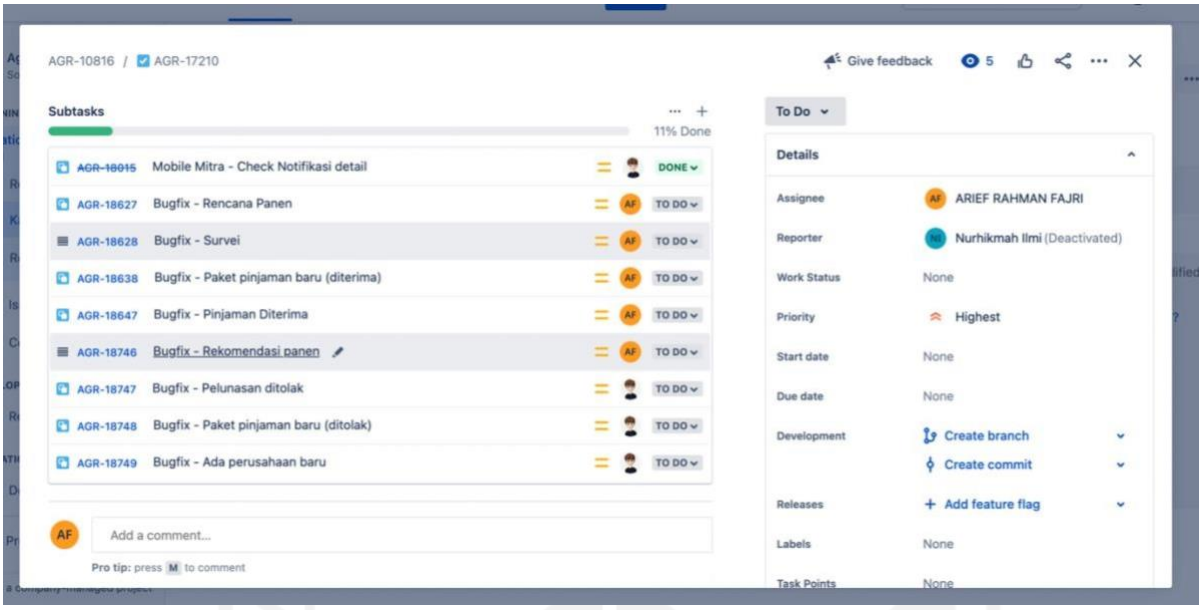
Pada tahap *Sprint Planning Product Owner* akan membahas mengapa *Product Backlog Item* yang dibawa ke *Sprint Planning* penting untuk meningkatkan kualitas produk. Gambar 3.2 adalah acara *Sprint Planning* pada proyek Migrasi yang dilakukan bersama dengan *Product Owner* dan *developer*.



Gambar 3.2 *Sprint Planning* proyek *Migration*

Product Owner bersama *developer* akan berdiskusi untuk memilih *Product Backlog Item* yang akan dimasukkan ke dalam *Sprint*. Ketika *Product Backlog Item* sudah terpilih, maka akan dilanjutkan ke acara *Sprint*.

Sprint adalah proses pengerjaan *Backlog Item* yang sudah terpilih dengan durasi dua minggu (10 hari kerja). Lama waktu *Sprint* dapat disesuaikan dengan kebutuhan perusahaan. Gambar 3.3 merupakan beberpa *Backlog Item* yang penulis kerjakan. Setiap *developer* memiliki *Backlog Item* masing-masing.



Gambar 3.3 *Backlog Item Sprint*

Dengan penerapan *Clean Architecture* memudahkan pembagian *Backlog Item*, serta memperjelas tanggung jawab dari setiap *developer* terhadap *Backlog Item* yang dikerjakan. Pemisahan konsentrasi yang didapatkan dari penerapan *Clean Architecture* juga memperkecil terjadinya konflik ketika *developer* melakukan pengembangan, karena *developer* tidak akan melakukan perubahan pada kode yang tidak berkaitan dengan *Backlog Item*-nya.

Pada saat *Sprint* berjalan, akan dilakukan *daily standup* yang dilakukan setiap hari dengan waktu dan tempat yang telah disepakati secara bersama. Gambar 3.4 merupakan acara *Daily standup.*

Gambar 3.4 *Daily Standup*

Acara ini bertujuan untuk memeriksa kemajuan dan perkembangan *Sprint Backlog*, serta kendala yang dialami selama proses pengerjaan. Kendala-kendala yang dialami oleh *developer* dapat didiskusikan pada *daily standup*, kemudian bersama-sama mencari solusi untuk menyelesaikannya.

Setelah *Sprint* telah selesai dilakukan, maka akan dilanjutkan ke tahap *Sprint Review*. *Developer* akan melakukan demo *Sprint Backlog* yang telah selesai dikerjakan kepada *Product Owner*. Kemudian akan dilakukan inspeksi terkait *Sprint Backlog* yang belum dapat diselesaikan dalam satu *Sprint* dan menentukan potensi *backlog* untuk *Sprint* selanjutnya.

Sprint Retrospective adalah proses perencanaan untuk meningkatkan kualitas dan efektivitas *Sprint*. *Scrum Team* akan melakukan pengkajian jalannya *Sprint* dan melakukan identifikasi untuk mengetahui apa saja hal yang dapat meningkatkan efektivitas *Sprint*, serta masalah-masalah yang dapat mengurangi efektivitas *Sprint*. Pengkajian tersebut berguna untuk meningkatkan kualitas dan efektivitas *Sprint* selanjutnya.

3.3 Pengembangan Fitur Inbox pada Proyek Migrasi

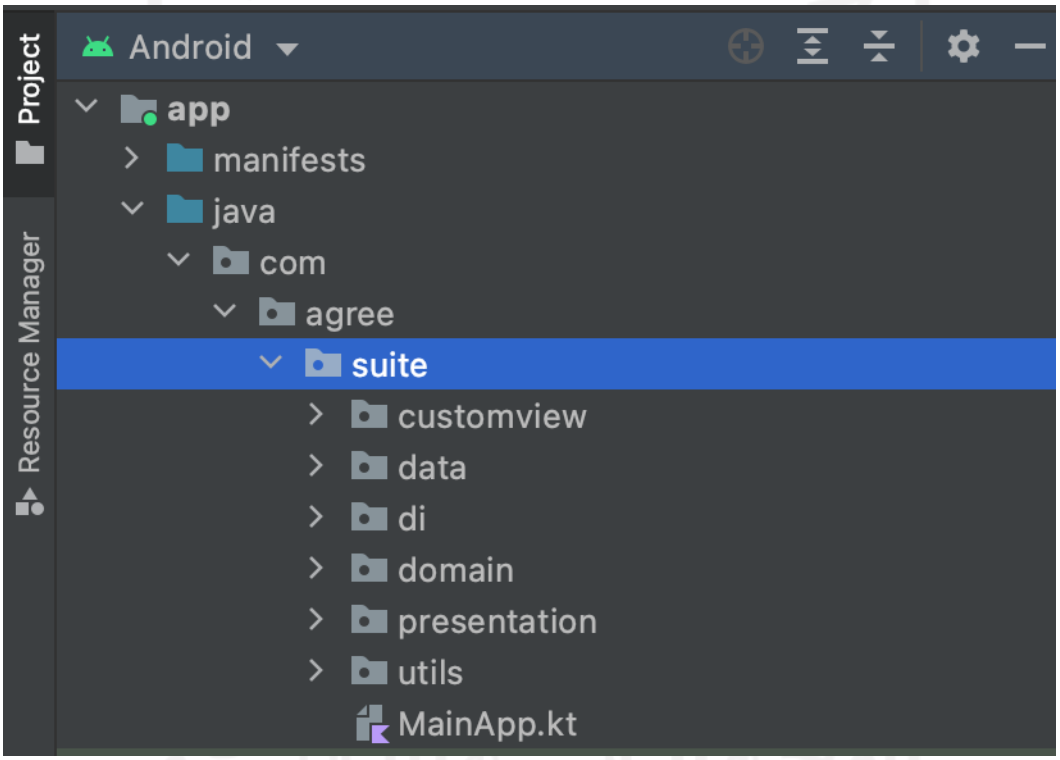
Proyek migrasi adalah proyek pembuatan aplikasi Agree Partner menggunakan Android *Native* untuk menggantikan Aplikasi Agree Partner yang sebelumnya sudah dikembangkan menggunakan ReactNative. Alasan pemindahan pengembangan ini dikarenakan isu peforma aplikasi yang dikembangan menggunakan Android *Native* lebih baik ketimbang menggunakan ReactNative.

Fitur *Inbox* merupakan salah satu fitur yang terdapat pada aplikasi Agree Partner. Fitur ini bertujuan untuk menerima pesan terkait berita dan informasi terbaru. Pesan-pesan tersebut terdiri dari beberapa jenis, seperti informasi panen, informasi pendaftaran, informasi pinjaman, informasi umum, pengumuman, dan lainnnya. Fitur ini dikembangan dengan menerapkan *Clean Architecture* dan *design pattern* MVVM.

3.3.1 Penerapan *Clean Architecture* dan *design pattern* MVVM pada Pengembangan Fitur *Inbox*

Pada pengembangan aplikasi Agree Partner, penerapan *Clean Arhcitecture* dilakukan dengan membagi proyek menjadi tiga lapisan yang direpresentasikan dengan *package.* Lapisan-lapisan tersebut yaitu *data package* (*entities layer*), *domain package* (*use case layer*), dan *presentation* (*interface adapter layer*) yang dapat dilihat pada Gambar 3.5. Penerapan

design pattern MVVM dilakukan dengan membagi kode menjadi tiga komponen, yaitu *Model*, *View*, dan *ViewModel*. Komponen *Model* yang berkaitan dengan data dan logika bisnis akan ditempatkan pada *data package*. *Data package* merupakan representasi dari *entities layer* pada *Clean Architecture*. *Data package* menangani proses bisnis perusahaan serta pertukaran data melalui *web service*. *Domain package* merupakan representasi dari *use case layer* yang mengenkapsulasi proses bisnis yang terjadi pada *data package*. *Domain pakcage* meneruskan *request* data dari *presentation package* menuju *data package*. *Presentation layer* merupakan lapisan yang bertanggung jawab menampilkan data yang telah diterima dari *data package* melalui *domain package*. *Presentation package* merupakan representasi dari *interface adapter layer* yang mana pada lapisan ini terdapat komponen *View* dan *ViewModel*.

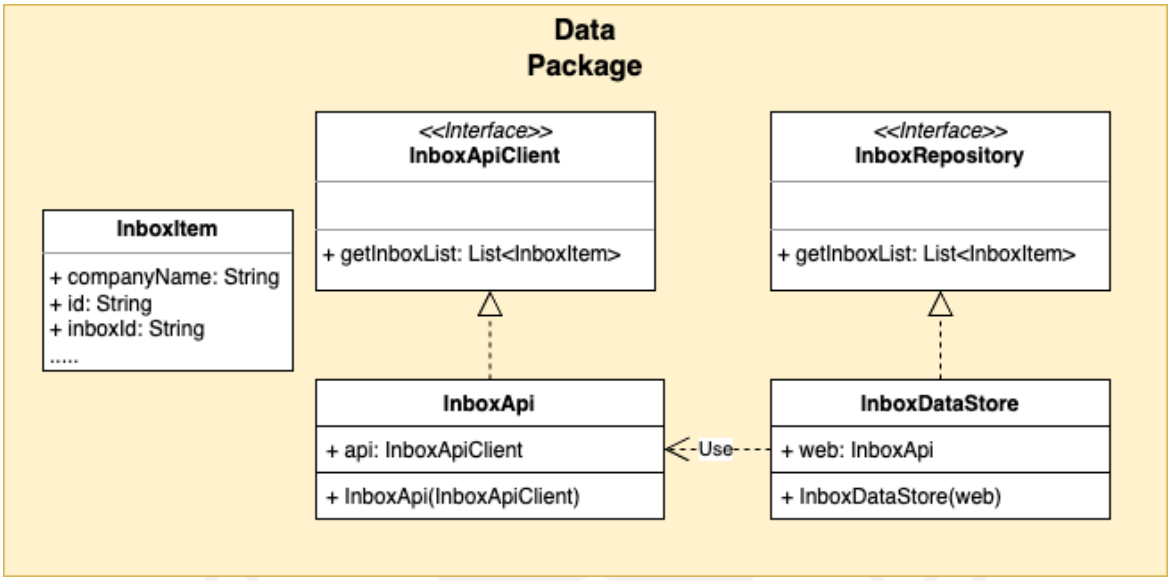


Gambar 3.5 Pembagian *Package Data, Domain, dan Presentation*

A. Data Package

Data package merupakan *layer* yang mengangani proses pengambilan data baik dari *local* maupun dari *network*. Lapisan ini berisikan kelas dan fungsi-fungsi yang merupakan proses bisnis suatu perusahaan. Gambar 3.6 merupakan kelas diagram pada *data package*. Pada lapisan ini terdapat *interface InboxApiClient* dan kelas *InboxApi* yang mengimplementasikan *interface* tersebut. Serta juga terdapat *interface InboxRepository* dan kelas *InboxDataStore*

yang mengimplementasikan *interface InboxRepository*. Kelas *InboxDataStore* memiliki *dependency* kepada kelas *InboxApi*.



Gambar 3.6 Diagram kelas *data package*

Gambar 3.7 merupakan sebuah *interface InboxApiClient* yang digunakan sebagai salah satu parameter untuk melakukan instansiasi Retrofit sebagai *service* yang terhubung ke server. Setiap pembuatan, pengambilan, pembaruan dan penghapusan data yang berkaitan dengan fitur *Inbox* dilakukan melalui *service* ini untuk terhubung ke server. *Interface InboxApiClient* juga digunakan sebagai parameter pada kelas *InboxApi*. Fungsi-fungsi yang terdapat pada Gambar 3.7 merupakan suatu proses bisnis dari fitur *Inbox*, seperti fungsi *getInboxList()* yang merupakan fungsi untuk mengambil daftar pesan dan fungsi *getInboxDetail*() untuk mengambil detil pesan.

```
interface InboxApiClient {

    @GET("")
    fun getInboxList(
        @QueryMap map: Map<String, String>
    ): Single<PagingApiResponse<List<InboxItem>>>

    @GET("")
    fun getInboxDetail(@Path("inboxId") inboxId: String):
Single<DevApiResponse<InboxItem>>

    @DELETE("")
    fun  deleteOneInbox(
        @Path("inboxId") inboxId: String
    ): Single<DevApiResponse<JsonElement>>
}
```

Gambar 3.7 Kode *Interface InboxApiClient*

Kelas *InboxItem* pada Gambar 3.8 digunakan untuk menyimpan data sementara hasil *response* dari REST API. Kelas *InboxItem* mencerminkan aturan bisnis dari perusahaan. Kelas

InboxItem digunakan sebagai nilai kembali pada fungsi-fungsi di *interface InboxApiClient*, kelas *InboxApi*, *interface InboxRepository*, dan kelas *InboxDataStore*. Kelas *InboxItem* juga memiliki fungsi *toInbox()* yang digunakan untuk melakukan perubahan tipe kelas ke kelas *Inbox* yang ada pada *package domain*. Perubahan tipe kelas tersebut dilakukan agar sesuai dengan format yang dibutuhkan oleh *presentation layer*.

```
data class InboxItem(
    @SerializedName("companyName")
    val companyName: String?,
    @SerializedName("_id")
    val id: String?,
    @SerializedName("inboxId")
    val inboxId: String?,
    . . .
    . . .
    @SerializedName("loanPackageTenor")
    val loanPackageTenor: String?,
    @SerializedName("farmerName")
    val farmerName: String?
) {
    fun toInbox() = Inbox(
        companyName = companyName.orEmpty(),
        id = id.orEmpty(),
        inboxId = inboxId.orEmpty(),
        . . .
        . . .
        loanPackageTenor = loanPackageTenor.orEmpty(),
        farmerName = farmerName.orEmpty()
    )
}
```

Gambar 3.8 Kelas *InboxItem*

Kelas *InboxApi* pada Gambar 3.9 memiliki parameter sebuah *interface InboxApiClient* yang digunakan untuk melakukan pemanggilan fungsi untuk melakukan pengambilan, pembuatan, pembaruan atau penghapusan data (proses bisnis perusahaan).

```
class InboxAPi(private val api: InboxApiClient) : InboxApiClient, WebService {
      override fun getInboxList(map: Map<String, String>):
      Single<PagingApiResponse<List<InboxItem>>> {
          return api.getInboxList(map)
      }

      override fun getInboxDetail(inboxId: String):
      Single<DevApiResponse<InboxItem>> {
          return api.getInboxDetail(inboxId)
      }

      override fun deleteOneInbox(inboxId: String):
      Single<DevApiResponse<JsonElement>> {
          return api.deleteOneInbox(inboxId)
      }

}
```

Gambar 3.9 Cuplikan Kelas *InboxApi*

Selain *interface InboxApiClient* dan kelas implementasinya, pada *data package* juga terdapat *interface InboxRepository* dan juga kelas *InboxDataStore* yang mengimplementasikan *interface InboxRepository*.

```
interface InboxRepository : DevRepository {

    fun getInboxList(
        map: Map<String, String>
    ): Single<Pair<List<InboxItem>, MetaPagingResponse>>

    fun getInboxDetail(inboxId: String): Single<InboxItem>

    fun deleteOneInbox(inboxId: String): Single<JsonElement>
}
```

Gambar 3.10 Cuplikan *Interface Repository*

Interface InboxRepository pada Gambar 3.10 memiliki fungsi-fungsi yang berkaitan dengan proses bisnis yang akan diimplementasikan oleh kelas *InboxDataStore* dan juga digunakan sebagai parameter *constructor* kelas *InboxInteractor*. Pada kelas *InboxDataStore* terdapat dua parameter di *constructor*-nya, yaitu *InboxAPI* dan *DbService*. Parameter objek *InboxAPI* digunakan untuk mengambil data dari *network*, sedangkan *DbService* digunakan untuk mengambil data dari penyimpanan lokal.

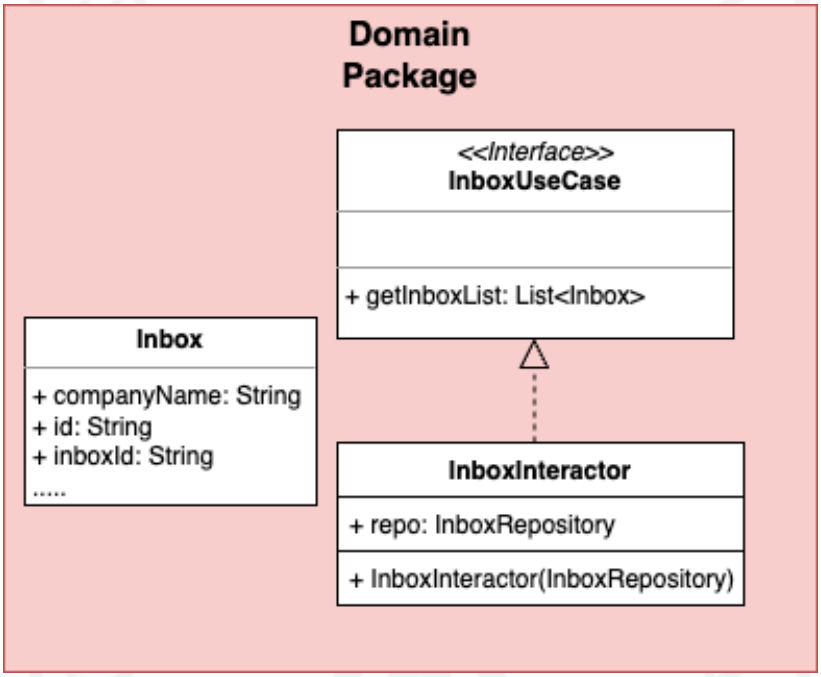
```
class InboxDataStore(web: InboxAPi, db: DbService?) : InboxRepository {

      override val dbService = db
      override val webService = web

      override fun getInboxList(map: Map<String, String>):
      Single<Pair<List<InboxItem>, MetaPagingResponse>> {
          return webService.getInboxList(map).map {
              val data = it.data ?: emptyList()
              val meta = it.meta ?: MetaPagingResponse()
              Pair(data, meta)
          }
      }

      override fun getInboxDetail(inboxId: String): Single<InboxItem> {
          return webService.getInboxDetail(inboxId).map { it.data }
      }

      override fun deleteOneInbox(inboxId: String): Single<JsonElement> {
          return webService.deleteOneInbox(inboxId).map { it.data }
      }
 }
```

Gambar 3.11 Cuplikan Kelas *InboxDataStore*

Kelas *InboxDataStore* pada Gambar 3.11 berfungsi sebagai *mediator* antara *domain package* dengan *data package*. Kelas *InboxDataStore* akan melakukan proses bisnis perusahaan yang berkaitan dengan pengambilan, penghapusan, pembaruan, atau pembuatan data. *Interface InboxRepository* akan digunakan oleh kelas *InboxInteractor* untuk mengambil, membuat, menghapus, atau memperbarui data dengan memanggil fungsi-fungsi yang ada pada *interface InboxRepository*.

B. Domain Package

Domain package merupakan *layer* yang mengenkapsulasi dan mengimplementasikan *use case* pada sistem. Layer ini merupakan representasi dari *use case layer* pada *Clean Architecture*. *Layer* ini bertugas sebagai mediator atau penghubung antara *presentation package* dan *data package*. Gambar 3.12 merupakan diagram kelas di *domain package*. Pada *package* ini didefenisikan kelas *Inbox*, *interface InboxUseCase* yang mengenkapsulasi *use case* pada fitur *Inbox* seperti mengambil daftar pesan. Kelas *InboxInteractor* merupakan kelas yang mengimplementasikan *InboxUseCase*.



Gambar 3.12 Diagram kelas *domain package*

Tujuan pembuatan kelas *Inbox* pada Gambar 3.13 adalah agar data yang diterima oleh kelas *view* tidak memiliki *property* yang bernilai *null*. Data yang dierima dari *data package* akan di *mapping* untuk menghindari *model* yang memiliki *property* bernilai *null*. Proses *mapping* ini dilakukan pada kelas *InboxInteractor*.

```
data class Inbox(
    val companyName: String,
    val id: String,
    val inboxId: String,
    . . .
    . . .
    . . .
    val loanPackageTenor: String,
    val farmerName: String
)
```

Gambar 3.13 Cuplikan Kelas Model

Interface InboxUseCase pada Gambar 3.14 memiliki fungsi-fungsi yang berkaitan dengan proses bisnis yang dimiliki oleh perusahaan. *Interface InboxUseCase* akan diimplementasikan pada kelas *InboxInteractor* pada Gambar 3.15.

```
interface InboxUseCase {
    fun getInboxList(
        map: Map<String, String>
    ): Single<Pair<List<Inbox>, MetaInbox>>

    fun getInboxDetail(inboxId: String): Single<Inbox>

    fun deleteOneInbox(inboxId: String): Single<JsonElement>
}
```

Gambar 3.14 Cuplikan *Interface InboxUseCase*

Kelas *InboxInteractor* memiliki parameter *interface InboxRepository* yang ada pada *data package*, dimana *interface InboxRepository* tersebut digunakan oleh kelas *InboxInteractor* untuk melakukan pengambilan atau pengiriman data sesuai dengan proses bisnis perusahaan.

Pada Gambar 3.15, kelas *InboxInteractor* memiliki suatu fungsi seperti *getInboxDetail()*, dimana pada fungsi tersebut menggunakan fungsi yang ada pada *interface InboxRepository* untuk melakukan pengambilan data yang bersumber dari *network*.

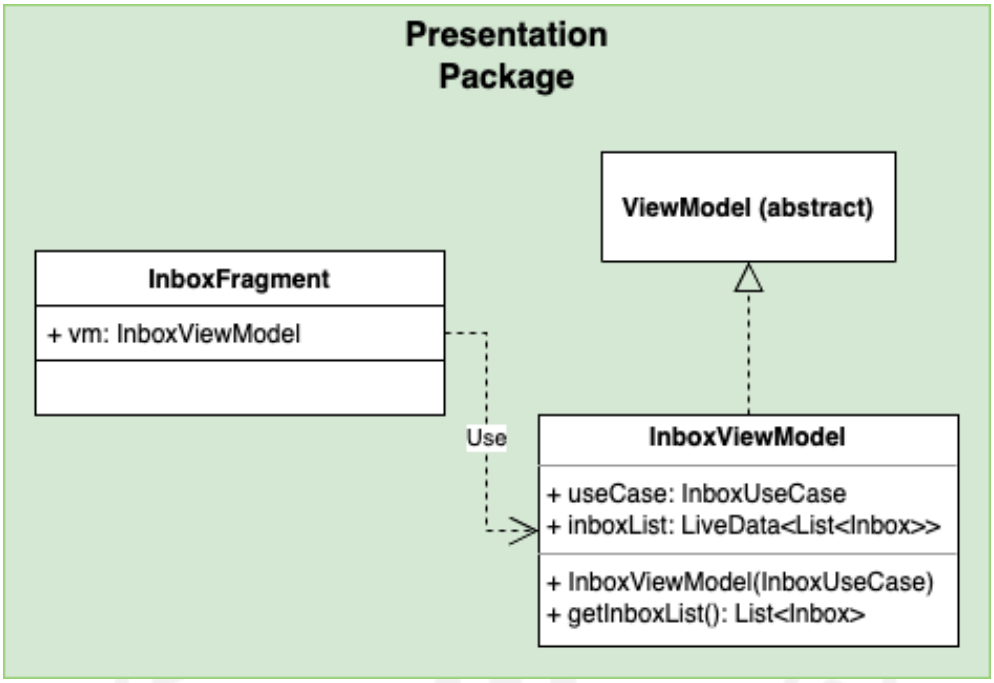
```
class InboxInteractor(private val repo: InboxRepository) : InboxUseCase {
    override fun getInboxList(map: Map<String, String>): Single<Pair<List<Inbox>,
MetaInbox>> {
        return repo.getInboxList(map).map {
            val (inboxList, metaItem) = it
            val data = ArrayList<Inbox>().apply {
                inboxList.forEach { inboxItem ->
                    this.add(inboxItem.toInbox())
                }
            }
            val meta = MetaInbox(metaItem.maxPage ?: 1, metaItem.maxData ?: 0,
metaItem.unreadInbox ?: 0)
            return@map Pair(data, meta)
        }
    }

    override fun getInboxDetail(inboxId: String): Single<Inbox> {
        return repo.getInboxDetail(inboxId).map { it.toInbox() }
    }

    override fun deleteOneInbox(inboxId: String): Single<JsonElement> {
        return repo.deleteOneInbox(inboxId).map { it }
    }
}
```

Gambar 3.15 Cuplikan Kelas *InboxInteractor*

Fungsi *getInboxDetail()* di kelas *InboxInteractor* pada Gambar 3.15 melakukan *mapping* ke Kelas *Inbox* agar *model* yang diterima oleh *viewmodel* tidak memiliki *property* yang bernilai *null*. *Mapping* tersebut dilakukan menggunakan fungsi *toInbox()* yang ada pada kelas *InboxItem* pada *data package*.

C. Presentation Package

Presentation package merupakan representasi dari *interface adapter layer* pada *Clean Architecture*. *Layer* ini bertugas menampilkan data terbaru yang diterima. Gambar 3.16 merupakan diagram kelas pada *presentation package*. Dimana pada lapisan ini terdapat komponen *View*, yaitu kelas *InboxFragment*, serta komponen *ViewModel*, yaitu kelas *InboxViewModel*.



Gambar 3.16 Diagram kelas *presentation package*

Pada *layer* ini diterapkannya *design pattern Model View ViewModel*. Hal tersebut dapat dilihat pada Gambar 3.16 dengan digunakannya kelas *InboxViewModel* yang mengimplementasikan kelas abstrak *ViewModel*. Kelas *InboxViewModel* digunakan sebagai kelas yang memegang data hasil *request* dari *data package*. Data tersebut disimpan menggunakan *LiveData* di kelas *InboxViewModel* yang nantinya akan diobservasi oleh kelas *InboxFragment*.

Pada Gambar 3.17 kelas *InboxViewModel* memiliki parameter *interface InboxUseCase* yang akan digunakan untuk melakukan proses bisnis, yaitu mengirim data atau mengambil data. Fungsi *getInboxList()* pada Gambar 3.17 merupakan fungsi yang digunakan untuk mengambil data pesan dari server dengan memanggil fungsi *getInboxList()* melalui *interface InboxUseCase.* Pada Gambar 3.17 terdapat dua *variable*, yaitu *_inboxList* dan *inboxList*. *Variable _inboxList* merupakan *MutableLiveData*, yaitu *LiveData* yang *value*-nya dapat diubah, sedangkan *variable inboxList* merupakan *LiveData* yang bersifat *immutable*, yaitu

value-nya hanya dapat dibaca dan tidak dapat diubah. *MutableLiveData* digunakan di *ViewModel* untuk menampung data terbaru yang kemudian digunakan untuk melakukan instansiasi *LiveData*. *Fragment* akan melakukan observasi pada *variable LiveData* untuk mengambil data terbaru dan kemudian melakukan *update* tampilan berdasarkan data yang diterima.

```
class InboxViewModel(
    private val useCase: InboxUseCase,
    private val compositeDisposable: CompositeDisposable
): DevViewModel(compositeDisposable) {

   private var _inboxList =    MutableLiveData<VmData<List<Inbox>>>().apply { value
   = VmData.Default() }
   val inboxList: LiveData<VmData<List<Inbox>>>
       get() = _inboxList

   fun getInboxList() {
       _inboxList.value = VmData.loading()
       resetPage()
       useCase.getInboxList(query)
           .compose(singleScheduler())
           .subscribe({
               val (inboxList, meta) = it
               _maxPage.value = meta.maxPage
               _totalData.value = meta.maxData
               _unreadInbox.value = meta.unreadInbox
               when {
                   inboxList.isEmpty() -> _inboxList.value = VmData.empty()
                   else -> {
                       _inboxList.value = VmData.success(inboxList)
                       if (meta.maxPage > page) {
                           page++
                           query["page"] = page.toString()
                       }
                   }
               }
           }, {
               it.printStackTrace()
               if (it is HttpException) {
                   val errorBody = it.response()?.errorBody()?.string()
                   if (errorBody != null) {
                       val errorMessage = JSONObject(errorBody).get("message") as
   String
                       _inboxList.value = VmData.Failure(null, errorMessage)
                       _totalData.value = 0
                   }
               } else {
                   val errorMessage = it.localizedMessage
                   _inboxList.value = VmData.Failure(null, errorMessage)
                   _totalData.value = 0
               }
           }).let(compositeDisposable::add)
   }

}
```

Gambar 3.17 Cuplikan Kelas *ViewModel*

Kelas *InboxFragment* adalah kelas yang menagani aksi yang dilakukan oleh pengguna. Aksi pengguna seperti mengetuk fitur Inbox, maka kelas *InboxFragment* akan mengeksekusi fungsi *onCreate()* pada Gambar 3.18. Dimana pada fungsi *onCreate()* tersebut terdapat pemanggilan fungsi *getInboxList()* pada kelas *InboxViewModel*.

```
class InboxFragment : DevFragment(), OnLoadMoreListener, InboxListCallback {


   private val menuNavController: NavController? by lazy {
   activity?.findNavController(R.id.nav_host_fragment_menu) }
   private val vm by sharedViewModel<InboxViewModel>()

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       vm.getInboxList()
       vm.listMultipleSelect.clear()
   }


   vm.inboxList.observe(viewLifecycleOwner) {
       when (it) {
           is VmData.Success -> {
               // Menampilkan daftar pesan
           }
           is VmData.Empty -> {
               // Menampilkan peringatan pesan tidak ditemukan
           }
           is VmData.Loading -> {
               // Menampilkan loading ketika memuat pesan
           }
           is VmData.Failure -> {
               // Menampilkan alasan kegagalan dalam memuat pesan
           }
       }
   }

}
```

Gambar 3.18 Cuplikan Kelas *Fragment*

Kemudian fungsi di kelas *InboxViewModel* akan memanggil fungsi dari kelas *interface InboxUseCase*. Kemudian kelas *InboxInteractor* akan memanggil fungsi dari *interface InboxRepository*. Lalu kelas *InboxDataStore* akan melakukan pemanggilan fungsi melalui kelas *InboxApi*. Kelas *InboxApi* akan memanggil fungsi dari *interface InboxApiClient* untuk melakukan pengambilan data dari server dengan bantuan *library* Retrofit. Data yang diterima kemudian akan disimpan pada *variable inboxList* yang merupakan sebuah *LiveData* di kelas *InboxViewModel*. *LiveData* adalah sebuah kelas penyimpanan data yang dapat diobservasi dan bersifat *lifecycle-aware*. *Variable inboxList* akan diobservasi untuk mengambil data terbaru dan kemudian kelas *InboxFragment* akan melakukan *update* tampilan menggunakan data tersebut.

Kode *vm.inboxList.observe(viewLifecycleOwner)* pada Gambar 3.18 merupakan kode yang digunakan untuk melakukan observasi *variable inboxList* di kelas *InboxViewModel*. Kelas *InboxFragment* akan menangani tampilan sesuai dengan data yang diterima. Jika data yang diterima sukses dan tidak kosong, maka akan menampilkan daftar pesan. Jika data kosong, maka akan menampilkan peringatan pesan tidak ditemukan. Jika data *loading*, maka akan menampilkan tampilan *loading*. Jika data gagal, maka akan menampilkan alasan kegagalan ketika memuat pesan.

3.3.2 Penerapan Aturan Ketergantungan *Clean Architecture*

Setiap *layer* pada *Clean Architecture* memiliki *dependency* antar kelasnya. Untuk memenuhi *dependecy* yang dibutuhkan dari setiap *layer*, maka dibutuhkan sebuah *service* yang menyediakan kebutuhan setiap *layer*. *Dependency injection package* merupakan *package* yang menyediakan dan menginjeksi setiap kebutuhan dari *package data*, *domain*, dan *presentation*. Untuk melakukan injeksi dependensi digunakan Koin yang merupakan sebuah *dependency injection framework*.

Pada Gambar 3.19, *Service, interface,* atau *class* yang dibutuhkan akan di instansiasi di dalam fungsi *module* yang merupakan tempat untuk mendefinisikan fungsi-fungsi Koin. Ketika suatu kelas membutuhkan suatu objek kelas atau *interface*, maka kelas atau *interface* yang sudah diinstansiasi tersebut akan diinjeksikan ke kelas yang membutuhkannya.

```
val reqresModule = module {
    single {
        createReactiveService(
            InboxApiClient::class.java,
            get(),
            get(named(BASE_ACTIVITY_URL))
        )
    }
    single { InboxApi(get()) }

    single<InboxRepository> { InboxDataStore(get(), null) }

    single<InboxUseCase> { InboxInteractor(get()) }

    viewModel { InboxViewModel(get(), get()) }
}
```

Gambar 3.19 Kode instansiasi *service*, *interface*, *class*, dan *viewmodel* menggunakan Koin

Untuk mendefiniskan *service, interface* atau kelas digunakan fungsi *single* atau *factory*. Fungsi *single* akan melakukan instansiasi *service, interface* atau *class* hanya satu kali, sedangkan fungsi *factory* akan melakukan instansiasi baru setiap pemanggilan *service, interface* atau kelas. Untuk *ViewModel* diinstansiasi menggunakan fungsi *viewModel*.

Agar *variable* Koin *module* yang sudah dibuat dapat digunakan, maka variable tersebut harus dideklarasikan di dalam fungsi *startKoin* di *Application class*. Dikarenakan pengembangan aplikasi Agree Partner menggunakan *codebase* dari Telkom, maka terdapat perbedaan cara mendeklarasikan *variable* Koin *module*.

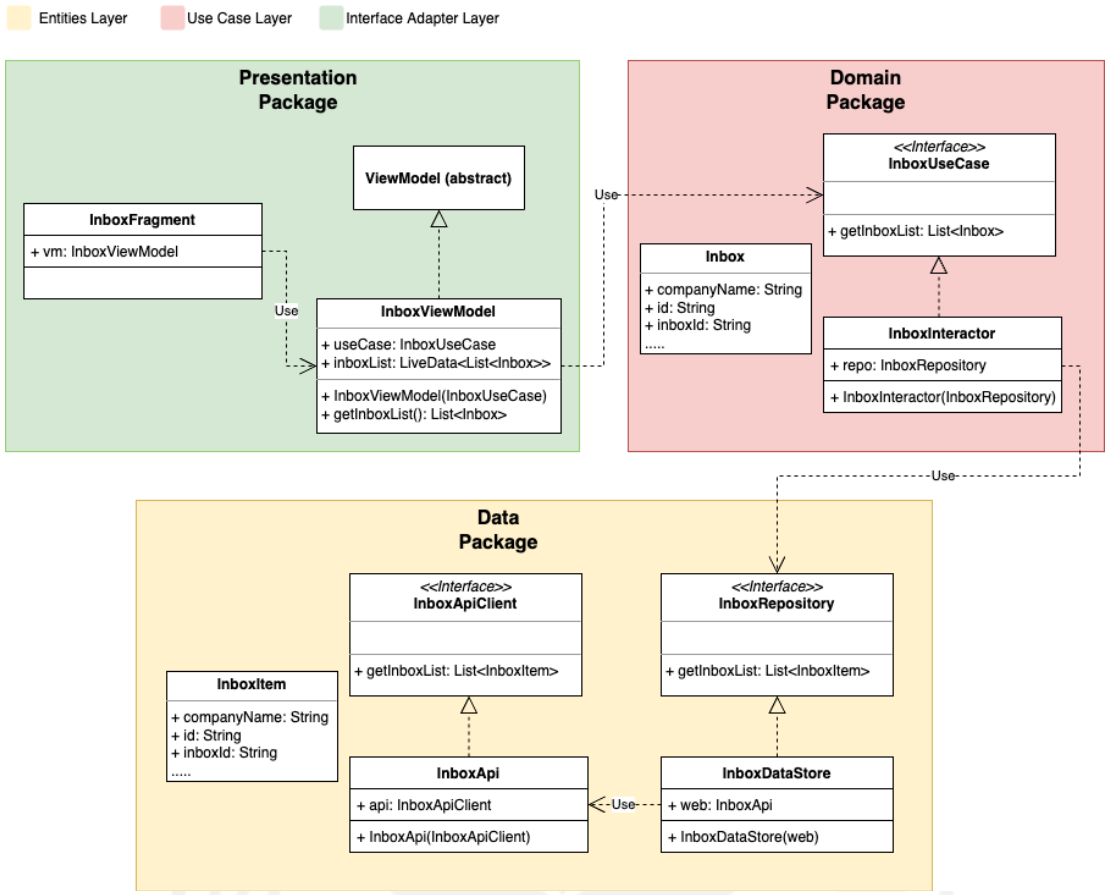
```
class MainApp : DevApplication() {
    override fun defineKoinModules() = arrayListOf(
        libModule,
        apiModule,
        reqresModule
    )
}
```

Gambar 3.20 *Application class*

Pada Gambar 3.20 *variable* Koin *module* yang telah dibuat dideklarasikan di dalam fungsi *defineKoinModules()* yang di-*override* dari DevApplication() yang merupakan sebuah *abstract class*.

3.4 Hasil Pembuatan Fitur *Inbox* dengan Menerapkan *Design Pattern* MVVM dan *Clean Architecture*

Pembuatan fitur *Inbox* dengan menerapkan *design pattern* MVVM dan *Clean Architecture* menghasilkan *layer* seperti Gambar 3.21.



Gambar 3.21 Stuktur *package* dengan menerapkan *design pattern* MVVM dan *Clean Architecture*

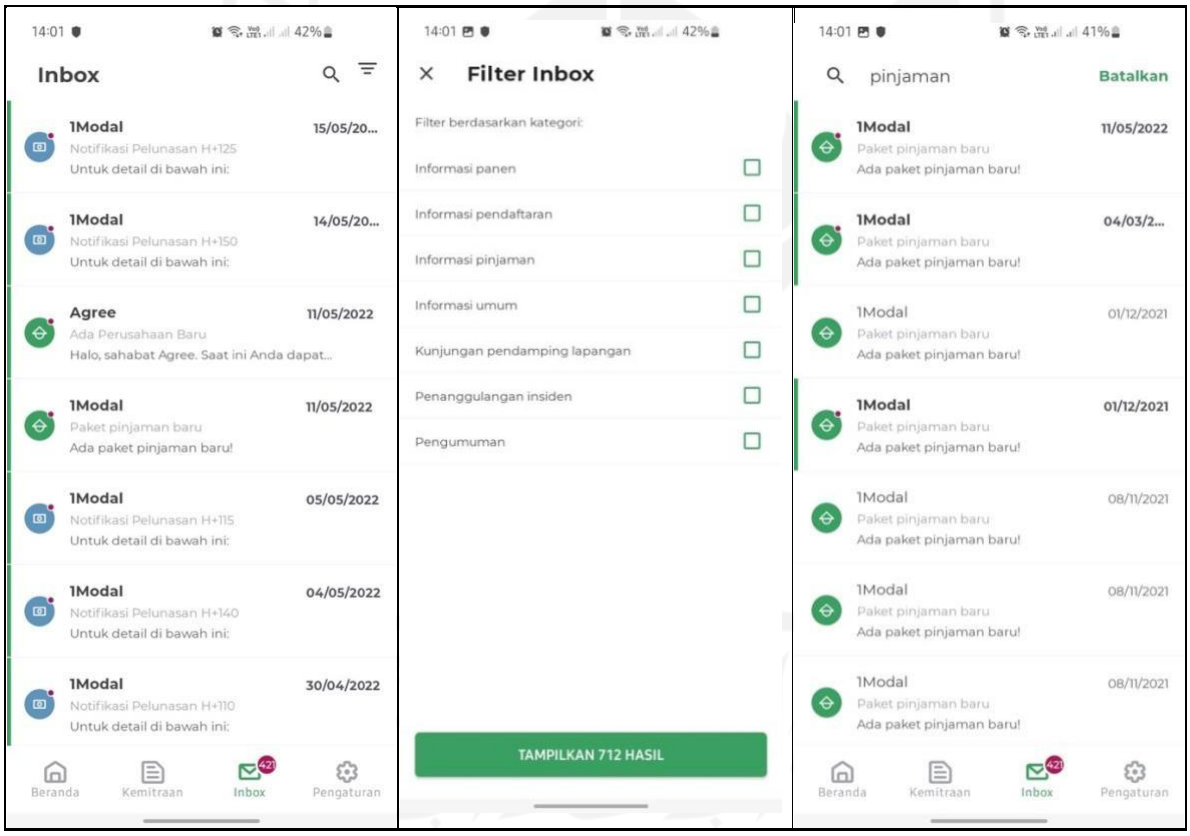
Presentation package yang berada pada lapisan *interface adapter layer* bertanggung jawab untuk melakukan pembaharuan tampilan berdasarkan data yang telah diperoleh dari *viewmodel*. Pada *presentation package* terdapat komponen tampilan, yaitu kelas *InboxFragment* yang mengatur tampilan pada layar *smartphone.* Pada lapisan ini juga diterapkannya *design pattern* MVVM, dimana kelas *InboxViewModel* yang mengimplementasikan kelas abstrak *ViewModel* bertugas untuk menghubungkan komponen *view* (*presentation package*) dengan komponen model (*data package*) melalui bantuan *use case layer* (*domain package*).

Domain package bertindak sebagai mediator atau penghubung antara *presentation package* dan *data package*. Kelas *InboxViewModel* pada *presentation package* akan melakukan *request* melalui *domain package* menggunakan *interface InboxUseCase* dan kemudian *domain package* meneruskan *request* tersebut ke *data package* menggunakan *interface InboxRepository.*

Data package akan melakukan *request* data ke server melalui REST API dengan menggunakan kelas *InboxApi*, kemudian mengembalikan data ke *domain package* dan diteruskan ke *presentation package*. Data tersebut akan disimpan oleh *variable InboxList* yang merupakan sebuah *LiveData*.

InboxFragment di *presentation package* mendapatkan data melalui *InboxViewModel* dengan menggunakan *interface InboxUseCase* yang ada pada *domain package*. Sementara itu *domain package* mendapatkan data dari *data package* dengan menggunakan *interface InboxRepository*. Data yang disimpan oleh *variable inboxList* akan diobservasi oleh *InboxFragment* untuk melakukan pembaharuan tampilan berdasarkan data yang disimpan.

Gambar 3.22 merupakan tampilan dari fitur *Inbox*, dimana pada fitur *Inbox* sendiri terdapat fitur *search* dan *filter*.



Gambar 3.22 Cuplikan Fitur *Inbox*

Fitur *search* berguna untuk melakukan pencarian pesan berdasarkan nama pesan, sedangkan fitur *filter* dapat digunakan untuk melakukan pencarian pesan berdasarkan kategori pesan. Fitur-fitur tersebut dikembangan dengan memanfaatkan *design pattern* MVVM dan *Clean Architecture*. Jika terdapat penambahan fitur di fitur Inbox, programmer hanya perlu menambahkan kode pada *package data*, *domain*, dan *presentation* yang berkaitan dengan fitur *Inbox*, sehingga tidak mengganggu kode fitur yang lainnya.

3.5 Manfaat Penerapan *Design Pattern* MVVM dan *Clean Architecture*

Manfaat yang didapatkan dari penerapan *Clean Architecture* dan *design pattern* MVVM adalah sebagai berikut.

1. *Design pattern* MVVM memisahkan antara logika tampilan dan logika bisnis. Sehingga memudahkan pengembangan secara paralel. Pengembangan kelas tampilan antarmuka dapat dilakukan bersamaan dengan pengembangan kelas data oleh *developer* yang berbeda.
2. *Developer* dapat dengan mudah untuk menambahkan fitur baru.
3. Kelas *InboxViewModel* dapat digunakan pada *Activity* atau *Fragment* lain, sehingga mengurangi kode *boiler plate*. Hal tersebut dapat dilakukan dengan membuat *object* dari kelas *InboxViewModel* pada kelas *Activity* atau *Fragment* lain. Pembuatan *object InboxViewModel* tersebut dapat dilakukan dengan menuliskan kode "*private val vm by sharedViewModel<InboxViewModel>()"* seperti pada Gambar 3.18.
4. Dengan *Clean Architecture*, kosentrasi kode dapat dipisahkan. Pada Gambar 3.21 dimana kode yang berkaitan dengan logika bisnis akan berada pada *data package* (*entities layer*) dan kode yang berkaitan dengan tampilan akan berada pada *presentation package* (*interface adapter layer*).
5. Penerapan *design pattern* MVVM dan *Clean Architecture* membuat kode menjadi lebih mudah dibaca dan dipahami oleh *developer lain*. Alur dari kode dapat dengan mudah dibaca. Pada Gambar 3.21 dimana ketika terjadi input aksi dari pengguna, maka *ViewModel* yang ada pada *presentation package* (*interface adapter layer*) akan meneruskan *request* ke *domain package* (*use case layer*). Kemudian *request* tersebut diteruskan ke *data package* (*entities layer*). Lalu data akan dikembalikan dari *data package* ke *ViewModel* melalui *domain package*. Data yang sudah tersimpan di *ViewModel* dapat di observasi oleh kelas *Activity* atau *Fragment* untuk melakukan pembaharuan tampilan.
6. Penerapan *design pattern* MVVM memudahkan kelas tampilan untuk mengakses data. Seperti pada Gambar 3.18, dimana kelas *InboxFragment* hanya perlu membuat *object vm* yang merupakan instansiasi dari kelas *InboxViewModel* dan kemudian melakukan pemanggilan fungsi yang terdapat pada kelas *InboxViewModel* untuk mendapatkan data.

7. Memudahkan pembagian *Backlog Item* kepada *developer* pada saat *sprint*, seperti pada Gambar 3.3 dan memperjelas tanggung jawab setiap *developer* terhadap *Backlog Item* yang dikerjakan.
8. Pemisahan kosentrasi pada kode membuat kemungkinan konflik pada saat pengembangan menjadi kecil. Hal tersebut dikarenakan *developer* tidak akan menggangu kode yang tidak berkaitan dengan *Backlog Item*-nya. Jika *developer* melakukan perubahan desain antarmuka, maka *developer* hanya perlu mengubah kelas *Activity* atau *Fragment* yang ada pada *presentation package*.

BAB IV
REFLEKSI PELAKSANAAN MAGANG

4.1 Relevansi Akademik

Aplikasi Agree Partner yang berbasis Android dalam pengembangannya menerapkan *Clean Architecture* dan *design pattern* MVVM. Pada pelaksanaannya tidak terdapat perbedaan antara teori *Android Clean Architecture* dan penerapannya di pengembangan aplikasi Agree Partner. Teori-teori yang dijelaskan diterapkan dengan baik pada pengembangan aplikasi Agree Partner. Aplikasi Agree Partner dibagi menjadi tiga *layer*, yaitu *data package* (*entities layer*), *domain package* (*use case layer*), dan *presentation package* (*interface adapter layer*) yang merepresentasikan *layer* pada *Clean Architecture*.

Penerapan MVVM pada pengembangan aplikasi Agree Partner juga tidak terdapat perbedaan dengan teori yang dijelaskan. Pada kelas *ViewModel* menerapkan *live data* sebagai *observable data holder* yang dapat diobservasi oleh komponen *View*. Kelas *ViewModel* juga mengamati dan mengkoordinasikan pembaruan dari data package, kemudian diteruskan ke domain package dan menyimpan hasil pembaruan *Model* ke dalam *variable LiveData*. Kemudian *variable LiveData* tersebut akan digunakan oleh komponen *View* untuk memperbaharui tampilan antarmuka aplikasi.

Pengembangan aplikasi di Agree menerapkan kerangka kerja *SCRUM*. *Scrum Team* di proyek Agree Partner terdiri dari *SCRUM master*, *developer*, dan *Product Owner*. Pengembangan dimulai dengan *Sprint Planning*, dimana *Product Owner* akan memberikan *Product Backlog Item* dan kemudian akan dibahas bersama *developer* untuk menentukan *backlog item* apa saja yang akan dibawa ke tahap *Sprint*. Kemudian *Sprint* akan dilaksanakan selama dua minggu. Setelah *Sprint* selesai dilakukan, akan dilakukan *Sprint Review* untuk melihat hasil pengerjaan *backlog item*. Acara terakhir ang dilakukan adalah *Sprint* Retrospective yang bertujuan untuk mengevaluasi kekurangan dari Sprint yang telah selesai agar Sprint selanjutnya menjadi lebih baik. Proses-proses *SCRUM* yang dilakukan di Agree juga telah sesuai dengan teori yang dijelaskan pada Bab II.

4.2 Pembelajaran Magang

Setelah menjalani magang selama 6 bulan, penulis mendapatkan banyak pengalaman pada saat bekerja. Pengalaman-pengalaman tersebut belum pernah penulis dapatkan sebelumnya. Pada saat magang, penulis dihadapkan dengan masalah yang sesungguhnya dan harus dapat menyelesaikan masalah tersebut dengan mengaplikasikan ilmu yang sudah dipelajari selama kuliah. Pembelajaran yang penulis dapatkan ketika menjalani magang dapat dikategorikan menjadi pembelajaran teknis dan non-teknis.

A. Teknis

Sebagai seorang Android *programmer,* penulis ditugaskan di dua proyek dan mengerjakan berbagai macam fitur. Banyak ilmu yang didapatkan ketika mengerjakan proyek tersebut. Rekan-rekan programmer lainnya juga berbagi ilmu tentang *best practice* pada saat melakukan *coding.* Ilmu-ilmu yang diberikan menjadi bekal untuk penulis agar bisa meng-*upgrade skill* menjadi lebih baik. Berikut adalah pembelajaran teknis yang penulis dapatkan ketika melaksanakan magang.

a. *Model View ViewModel*

   Model View ViewModel merupakan salah satu *design pattern* yang digunakan pada saat mengembangkan aplikasi Android. *Design pattern* ini menjadi penghubung antara UI *layer* dan *Data layer*. Selain itu, MVVM juga memisahkan UI *logic* dan *business logic* sehingga UI dapat dengan mudah diubah.

   Selama magang penulis menggunakan *design pattern* MVVM untuk mengembangan fitur *Inbox*. *Design pattern* ini sangat membantu dalam membuat kode menjadi lebih bersih karena memisahkan UI *logic* dan *business logic*. Sehingga sangat memudahkan penulis untuk melakukan pembaharuan UI atau penambahan fitur seperti fitur *Search* dan fitur *Filter*.

b. *Android Clean Architecture*

   Clean Architecture bertujuan untuk memisahkan kepentingan dengan membagi aplikasi menjadi beberapa *layer*. Dalam proyek magang yang penulis kerjakan, aplikasi dibagi menjadi tiga *layer*, yaitu *data package* (*entities layer*), *domain package* (*use case layer*), dan *presentation package* (*interface adapter layer*). *Data package* berfokus kepada proses pengambilan dan pengiriman data, *presentation package* berfokus kepada tampilan *User Interface*, dan *domain package* berfungsi sebagai pusat komunikasi antara *presentation package* dan *data package*.

Dalam pengembangan aplikasi AgreePartner diterapkan *Clean Archicture* agar kualitas kode lebih baik dan mudah dipahami oleh programmer yang lain. Dikarenakan banyak programmer yang mengerjakan proyek tersebut. Bagi penulis *Clean Architecture* sangat membantu penulis untuk memahami *flow* dari kode-kode yang ada. Karena *Clean Architecture* membuat kode-kode menjadi lebih terstruktur dan mudah dipahami. *Clean Architecture* juga mempermudah pengembangan karena *programmer* tidak akan mengubah kode yang tidak termasuk dalam fitur tersebut, sehingga memperkecil kemungkinan terjadinya konflik pada saat pengembangan.

c. Git

Git merupakan *software version control system* yang berfungsi untuk mencatat semua perubahan dalam suatu proyek. Aplikasi Agree Partner memiliki banyak *progammer* dan fitur sehingga diperlukan sebuah *version control system* untuk mengatur *flow* pengerjaan sebuah fitur agar tidak terjadi konflik. Penulis mempelajari bagaimana menerapkan sebuah Git *flow* yang baik untuk menghindari konflik antar programmer. Selain itu penulis juga belajar cara meng-*cloning project* dari *repostiory*, serta belajar melakukan push, pull, dan *merging* antar *branch*.

B. Non-Teknis

Pembelajaran non-teknis yang penulis rasakan secara pribadi selama melaksanakan magang di Agree adalah sebagai berikut.

a. Komunikasi

Selama magang penulis dituntut bekerja di bawah tekanan dan harus bisa bekerja sama dengan baik dalam sebuah tim. Komunikasi menjadi hal yang penting pada saat bekerja sama dalam sebuah tim. Jika komunikasi berjalan dengan lancar, maka tugas-tugas yang ada dapat diselesaikan dengan baik pula dan memperkecil kesalahan karena miskomunikasi. Komunikasi di dalam tim biasa dilakukan melalui Whatsapp dan Google Meet.

b. Manajemen Waktu

Setiap fitur yang dikembangkan dilakukan dalam satu *Sprint* yang berdurasi 10 hari kerja, sehingga penulis harus bisa memanajemen waktu dengan baik agar fitur dapat diselesaikan dengan tepat waktu. Karena hal tersebut, *skill* manajemen waktu menjadi salah satu *skill* yang penting. Manajemen waktu dilakukan sesuai dengan prioritas pekerjaan dan *hard skill* yang di miliki.

c. Memperluas Relasi

Selama magang penulis juga mendapat banyak relasi baru dengan rekan-rekan *programmer* lain yang lebih senior. Tidak hanya rekan *programmer*, penulis juga mendapat relasi dengan *quality assurance*, *SCRUM master*, *back-end engineer,* dan *project manager*. Relas-relasi tersebut bermanfaat untuk perkembangan karir penulis kedepannya.

d. Mengetahui Industri Teknologi

Sebelumnya penluis tidak mengetahui bagaimana sebuah industri teknologi bekerja dan seperti apa budaya kerja di dalam industri teknologi. Setelah melaksanakan magang barulah penulis mengetahui bagaimana industri teknologi bekerja. Industri teknologi bekerja dengan banyak orang dan tim di dalamnya. Setiap tim berkolaborasi dengan tim lainnya dalam bekerja untuk mencapai *Product Goal* yang telah ditetapkan.

4.3 Hambatan dan Tantangan Magang

Hambatan yang penulis rasakan ketika melakukan magang secara daring adalah komunikasi. Magang secara daring membuat komunikasi hanya terbatas melalui Whatsapp dan Google Meet. Kedua *software* tersebut memerlukan internet, sehingga apabila terjadi mati listrik atau gangguan internet membuat komunikasi menjadi terkendala dan dapat berakibat menghambat pekerjaan.

Tantangan yang harus penulis hadapi adalah mengatur atau memanajemen waktu. Hal tersebut dikarenakan penulis memiliki kelas kuliah serta tugas magang. Penulis melakukan manajemen waktu berdasarkan prioritas. Jika suatu tugas memiliki prioritas yang tinggi, maka penulis akan mendahulukan tugas tersebut, kemudian mengerjakan tugas yang prioritasnya lebih rendah.

BAB V
PENUTUP

5.1 Kesimpulan

Dari penerapan design pattern MVVM dan *Clean Architecture*, terdapat beberapa hal yang dapat disimpulkan, yaitu sebagai berikut.

a. Penerapan *Clean Architecture* pada pengembangan aplikasi Android menggunakan Android Studio diimplementasikan dengan membagi proyek menjadi tiga *layer*, yaitu *entities layer* (*data package*), *use case layer* (*domain package*), dan *interface adapter layer* (*presentation package*).
b. *Design pattern Model View ViewModel* dapat diterapkan bersamaan dengan *Clean Architecture*. Penerapan *design pattern* MVVM ini dilakukan pada *interface adapter layer* (*presentation package*) dengan membuat kelas yang mengimplementasikan kelas abstrak *ViewModel*.
c. Agar *Clean Architecture* berjalan dengan baik, *developer* harus mematuhi prinsip *Dependency Inversion Principle*.
d. Memudahkan pembagian *Backlog Item* kepada *developer* dan memperjelas tanggung jawab *developer* terkait *Backlog Item* yang dikerjakan.
e. Membuat kode menjadi lebih mudah dibaca dan dipahami oleh *developer* lain. Hal ini dikarenakan alur kode yang jelas, mulai dari proses pengambilan data sampai dengan menampilkan data.

5.2 Saran

Clean Architecture memiliki tujuan untuk membuat sistem yang *Independent of Framework*, *Testable*, *Independent of Userinterface*, *Independent of Database*, dan *Independent of External*. Hanya saja pada penerapannya di proyek *AgreePartner* tidak selalu menerapkan *Unit Test*, sehingga belum memenuhi satu tujuan dari *Clean Architecture*, yaitu *Testable*. Tujuan tersebut dapat dipenuhi dengan menerapkan *Unit Test* dan *Instrument Test* pada proyek aplikasi Agree Partner.

DAFTAR PUSTAKA

5: Implementing the MVVM Pattern Using the Prism Library 5.0 for WPF | Microsoft Docs. (n.d.). Retrieved March 15, 2022, from https://docs.microsoft.com/en-us/previous-versions/msp-n-p/gg405484(v=pandp.40)?redirectedfrom=MSDN

Aldiansyah, M. W., Kharisma, A. P., & Arwani, I. (2021). *Pengembangan Aplikasi Ngobat : Aplikasi Ketaatan Regimen Pengobatan menggunakan Gamification pada Platform Android*. *5*(8), 3600–3608.

Bui, H., & Technology, I. (2020). *Android Application for Students ' Personal Finances*.

Catur Yuantari, M., Kurniadi, A., & Kesehatan, F. (2016). Pemanfaatan Teknologi Informasi Untuk Meningkatkan Pemasaran Hasil Pertanian Di Desa Curut Kecamatan Penawangan Kabupaten Grobogan Jawa Tengah. *Techno.COM*, *15*(1), 43–47.

Dicoding Indonesia. (n.d.-a). Retrieved March 15, 2022, from https://www.dicoding.com/academies/129/tutorials/4437

Dicoding Indonesia. (n.d.-b). Retrieved March 15, 2022, from https://www.dicoding.com/academies/165/tutorials/10289?from=10284

Kementerian Pertanian. (2020). *Pusat Data dan Sistem Informasi Pertanian Sekretariat Jenderal-Kementerian Pertanian Center for Agriculture Data and Information System Secretariat General-Ministry of Agriculture 2020*. 1–201.

KKP | Kementerian Kelautan dan Perikanan. (n.d.). Retrieved May 15, 2022, from https://kkp.go.id/djprl/artikel/21045-konservasi-perairan-sebagai-upaya-menjaga-potensi-kelautan-dan-perikanan-indonesia

Komponen Arsitektur Android | Developer Android | Android Developers. (n.d.). Retrieved March 10, 2022, from https://developer.android.com/topic/libraries/architecture

Makarenko, V., Olshevska, O., & Kornienko, Y. (2017). an Architectural Approach for Quality Improving of Android Applications Development Which Implemented To Communication Application for Mechatronics Robot Laboratory Onaft. *Automation of Technological and Business Processes*, *9*(3), 8–13. https://doi.org/10.15673/atbp.v9i3.714

Maya, U. G. (2017). *Bab 2 Sejarah Android*. 5–14. http://repository.untag-sby.ac.id/514/3/BAB 2.pdf

Ringkasan ViewModel | Developer Android | Android Developers. (n.d.). Retrieved May 15,

2022, from https://developer.android.com/topic/libraries/architecture/viewmodel?hl=id

Schwaber, K., & Sutherland, J. (2017). The Scrum Guide: The Definitive The Rules of the Game. *Scrum.Org and ScrumInc*, *November*, 19. http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf

Sharma, S., Patodkar, V., Simant, S., & Shah, C. (2015). E-Agro Android Application (Integrated Farming Management Systems for sustainable development of farmers). *International Journal of Engineering Research and General Science*, *3*(1), 4–8.

Sutabri. (2016). Bab Ii Tinjauan Pustaka Dan Dasar Teori. *Journal of Chemical Information and Modeling*, *53*(9), 1689–1699.

Tian Lou. (2016). *A comparison of Android Native App Architecture Master ' s Programme in ICT Innovation A Comparison of Android Native App Architecture – MVC , MVP and MVVM*. 57.

Tung Bui Du. (2017). *Reactive Programming and Clean Architecture in Android Development*. *April*, 47.

Yoga, I. K., Putra, D., Kharisma, A. P., & Arwani, I. (2021). Pengembangan Aplikasi Berbasis Android Untuk Meningkatkan Kepatuhan Pasien Gagal Jantung Dalam Merawat Diri Di Rumah. *Jurnal Pengembangan Teknologi Informasi Dan Ilmu Komputer*, *5*(7), 2976–2985.