Penerapan Design Pattern MVVM dan Clean Architecture pada Pengembangan Aplikasi Android (Studi Kasus: Aplikasi Agree Partner)

Arief Rahman Fajri
Program Studi Informatika – Program Sarjana
Universitas Islam Indonesia
Yogyakarta, Indonesia
18523092@students.uii.ac.id

Septia Rani
Jurusan Informatika
Universitas Islam Indonesia
Yogyakarta, Indonesia
septia.rani@uii.ac.id

***Abstract*—Agree Partner adalah aplikasi yang menghubungkan petani dan perusahaan mitra. Agree Partner merupakan produk dari unit kerja Agree yang berada dibawah binaan PT Telkom Indonesia. Aplikasi Agree Partner memiliki banyak fitur dan dikembangkan oleh banyak *programmer*. Dikarenakan fitur aplikasi yang terus bertambah dan seringnya terjadi pergantian *programmer*, maka diperlukan penerapan suatu arsitektur dan *design pattern* agar memudahkan proses pengembangan dan perawatan kode. Makalah 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.**

Keywords—Aplikasi Agree Partner, Android, Clean Architecture, design pattern MVVM

I. PENDAHULUAN

Indonesia merupakan negara yang kaya dengan sumber daya alam, mulai dari kekayaan hutan hingga lautan. Luas wilayah daratan yang dimiliki Indonesia mencapai 2,01 juta km^2 [1]. 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 [2]. Meskipun Indonesia menjadi salah satu negara dengan lahan pertanian yang luas di dunia, pemanfaatan teknologi informasi dan komunikasi pada sektor pertanian di Indonesia masih kurang dikarenakan rendahnya tingkat pendidikan dan banyak petani yang buta huruf [3]. Padahal petani memerlukan pengetahuan dan informasi mengenai berbagai topik, seperti perkembangan pasar, perkembangan harga, teknologi untuk produksi, manajemen penjualan, dan lain-lain.

Berdasarkan permasalahan tersebut, unit kerja Agree dibentuk 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, dan penyedia jasa saprotan. Ekosistem pertanian digital ini diharapkan mampu membawa manfaat bagi semua *stakeholder*, baik secara bisnis, operasional, maupun sosial.

Agree memiliki tiga produk, yaitu Agree Partner, Agree Market, dan Agree Modal. Produk-produk tersebut berbasis *website* dan *mobile*. Makalah ini akan berfokus kepada aplikasi Agree Partner. Aplikasi Agree Partner 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. Selain itu, petani dapat melaporkan progress aktivitas mereka kepada perusahaan melalui aplikasi. Aplikasi Agree Partner dikembangkan menggunakan Android Studio sebagai *integrated development environment* (IDE) dan menggunakan bahasa pemrograman Kotlin.

Salah satu kesalahan yang sering dilakukan oleh *developer* pada saat proses pengembangan aplikasi di antaranya yaitu penulisan kode logika proses bisnis dan kode logika tampilan yang ditulis dalam satu *class* yang sama. Hal ini dapat menyebabkan kode menjadi susah untuk dibaca, diperbaharui, dites, dan dirawat. Kurangnya penerapan *design pattern* dan arsitektur pada pengembangan sebuah aplikasi juga menyebabkan kesulitan bagi *developer* untuk memahami *flow* dari kode proyek tersebut. Apalagi jika proyek tersebut sering mengalami pergantian *developer*, maka *developer* baru akan mengalami kesulitan untuk membaca dan memahami kode pada proyek tersebut.

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. 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.

Selanjutnya, pembahasan pada makalah ini berfokus pada proses 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.

Penerapan *Clean Architecture* dan *design pattern* MVVM ini bertujuan agar kode pada aplikasi Agree Partner menjadi terorganisir, mudah dirawat, dan mudah dipahami.

Diharapkan makalah ini dapat menjadi panduan penerapan *Clean Architecture* dan *design pattern* MVVM pada pengembagan aplikasi Android.

II. LANDASAN TEORI

A. *Android*

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 [4].

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* virtual 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 [4].

Android merupakan sistem operasi *open source* yang memungkinkan perangkat lunak untuk dimodifikasi secara bebas dan didistribusikan oleh para pembuat perangkat, pengembang aplikasi, nirkabel, dan operator [4]. Android Studio dapat digunakan untuk pengembangan aplikasi Android dengan menggunakan bahasa pemrograman Java atau Kotlin. *Model View ViewModel* menjadi *design pattern* yang direkomendasikan untuk pengembangan aplikasi Android. Selain *design pattern*, Google juga merekomendasikan agar arsitektur aplikasi dibagi menjadi tiga *layer*, yaitu UI *Layer*, *Domain Layer*, dan *Data Layer*.

B. *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 [5]. Gambar 1 menunjukkan gambaran dari *Android Architecture Components*. Berikut ini penjelasan dari *Android Architecture Components*.

1. *Activity/Fragment* berfungsi sebagai *user interface controller* yang berfungsi untuk menampilkan data dan sebagai masukan aksi dari *user*. *Activity/Fragment* berinteraksi dengan *ViewModel* melalui perantara *LiveData* [6].

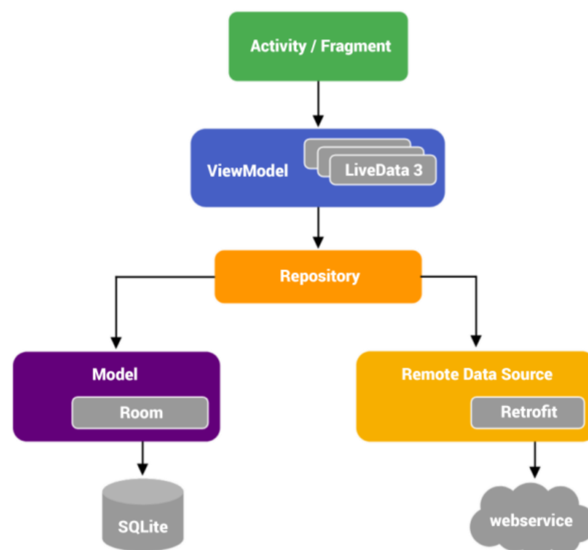
2. *ViewModel* berguna sebagai pusat komunikasi antara Repository dan *Activity/Fragment*. *ViewModel* mengambil data melalui *Repository*. *ViewModel* juga berfungsi untuk mempertahankan data dari perubahan konfigurasi [6].

3. *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 [6].

4. *Repository* digunakan untuk mengelola banyak data yang dapat bersumber dari *network* dan *local*. Jika data bersumber dari *local*, maka *Repository* akan menggunakan Room untuk berinteraksi dengan SQLite. Apabila data berasal dari *remote/network*, maka Repository akan berinteraksi dengan *web service* menggunakan Retrofit [6].

5. Room merupakan *library* yang mempermudah dalam melakukan *query* pada basisdata SQLite di Android [7].

6. Retrofit adalah *library* yang mempermudah proses pertukaran data antara aplikasi Android dengan *web service* melalui REST API [6].



Gambar 1. Android Architecture Components (Sumber: [6])

Gambar 1 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 melakukan pembaharuan tampilan berdasarkan data yang diterima [6].

C. *Model View ViewModel*

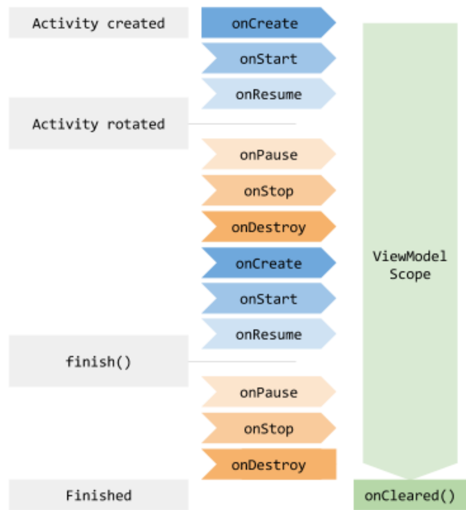
Model View ViewModel (MVVM) merupakan sebuah *design pattern* yang memiliki tiga komponen, yaitu *Model*, *View*, dan *ViewModel* [8]. Berikut adalah penjelasan ketiga komponen tersebut.

1. *Model* merupakan representasi dari data yang digunakan pada *business logic*. Model dapat berupa *Plain Old Java Object* (POJO) atau Kotlin *Data Classes* [7].

2. *View* adalah komponen yang terdiri dari *layout resource file* dan *Activity/Fragment*. Tampilan pada *layout resource file* akan dikontrol melalui *Activity/Fragment* secara dinamis [8].

3. *ViewModel* akan berinteraksi dengan *Model* dan menyiapkan *observables variable* yang akan diobservasi oleh *View*. *ViewModel* bersifat *lifecycle-aware* yang dapat dilihat pada Gambar 2 yang mana kelas ini akan hidup ketika sebuah kelas *View* telah melalui tahapan *create* dan belum melalui tahapan *destroy*. Pada kelas *View*, pemanggilan data hanya dilakukan satu kali dan data akan dipertahankan di *ViewModel Scope* selama kelas *View*

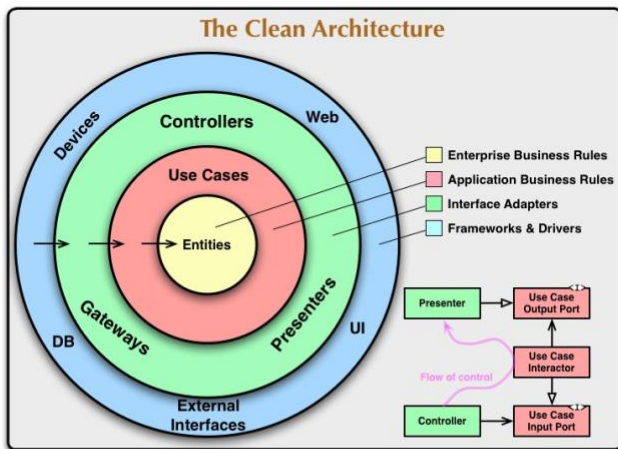
belum melalui tahapan *destroy*. Jika kelas *View* telah melewati tahapan *destroy*, maka data yang ada di *ViewModel* akan dibersihkan [7].



Gambar 2. Siklus hidup *ViewModel* (Sumber: [5])

D. *Clean Architecture*

Gambar 3 mengilustrasikan jenis-jenis *layer* dan *data flow* di *Clean Architecture*. *Entities layer* bertanggungjawab terhadap *enterprise business rules* [9].

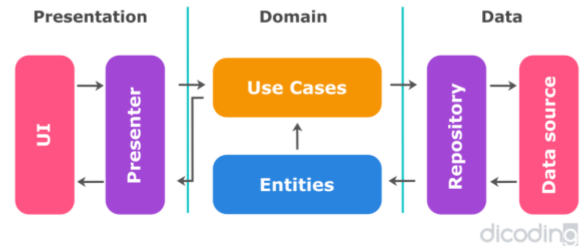


Gambar 3. *Clean Architecture* (Sumber: [10])

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 mengetahui tentang *framework* atau *driver* yang digunakan oleh aplikasi. *Frameworks* dan *drivers layer* terdiri dari *framework* atau *tools* seperti *database*, *web framework*, dan lain-lain.

Agar *Clean Architecture* dapat bekerja dengan baik, *developer* harus mematuhi *dependency rule* dan ketergantungan hanya boleh mengarah ke dalam [9]. 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*.

Penggunaan *Clean Architecture* pada proyek Android pada umumnya dibagi menjadi tiga *layer*, yaitu *presentation*, *domain*, dan *data* [11]. Pembagian *layer* tersebut dapat dilihat pada Gambar 4.



Gambar 4. *Layer Android Clean Architecture* (Sumber: [11])

Berikut adalah penjelasan mengenai komponen dari setiap *layer* yang terdapat pada Android *Clean Architecture* menurut [11].

1. *Presentation Layer* berisi UI dan *Presenter/ViewModel* yang akan mengatur tampilan berdasarkan *update* data terbaru. UI akan bergantung kepada *Use Case*.
2. *Domain Layer* berisi *Entities*, *Use Case*, dan *Repository Interface*. *Layer* ini merupakan *layer* inti yang berkaitan dengan bisnis model.
3. *Data Layer* berisi implementasi *Repository* dan *Data Source* yang bisa berasal dari *local data source* (basisdata lokal) atau *remote data source* (*network* atau REST API).

E. *Dependency Inversion Principle dan Dependency Injection*

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* [12]. *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 basisdata atau menyampaikan pesan ke sistem informasi.

Dependency Injection (DI) adalah sebuah *design pattern* yang menerapkan prinsip *inversion of control* [9]. Pada umumnya DI 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*.

F. Kajian Pustaka

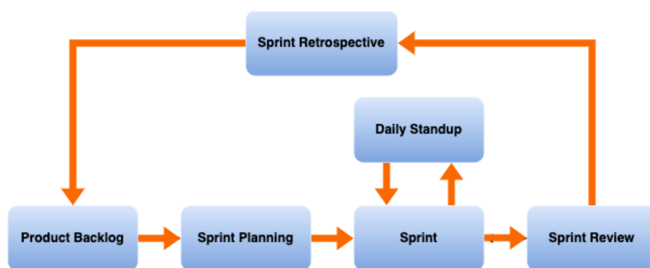
Terdapat beberapa penelitian yang telah dilakukan terkait pengembangan aplikasi Android menggunakan Android Studio. Seperti penelitian yang berjudul *An Architectural Approach for Quality Improving of Android Applications Development which Implemented to Communication Application for Mechatronics Robot Laboratory Onaft* [13]. 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 lain yang berjudul *Android Applications for Student's Personal Finance* [14]. Penelitian ini menerapkan *design pattern* MVVM dan *Clean Architecture*. Penelitian tersebut menjelaskan tentang penerapan *design pattern* MVVM pada pengembangan aplikasi Android menggunakan Android Studio dengan bahasa Kotlin. Tetapi tidak ada penjelasan mengenai penerapan *Clean Architecture* pada pengembangan aplikasi.

Melihat penjelasan yang kurang mengenai penerapan *Clean Architecture* dan *design pattern* MVVM pada suatu proyek pengembangan aplikasi Android. Diharapkan penelitian ini dapat menjadi panduan bagi pembaca untuk menerapkan *Clean Architecture* dan *design pattern* MVVM dalam pengembangan aplikasi Android.

III. METODOLOGI

Metodologi yang digunakan dalam pengembangan aplikasi Agree Partner adalah *scrum*. Gambar 5 mengilustrasikan proses pengembangan aplikasi Agree Partner.



Gambar 5. 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 nilai produk. Kemudian *product owner* bersama *developer* akan berdiskusi untuk memilih *product backlog item* yang akan dimasukkan ke dalam *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. Pada saat *sprint* berjalan, akan dilakukan *daily standup* yang dilakukan setiap hari dengan waktu dan tempat yang telah disepakati secara bersama. *Daily standup* bertujuan untuk memeriksa kemajuan dan perkembangan *sprint backlog*, serta kendala yang dialami selama proses pengerjaan.

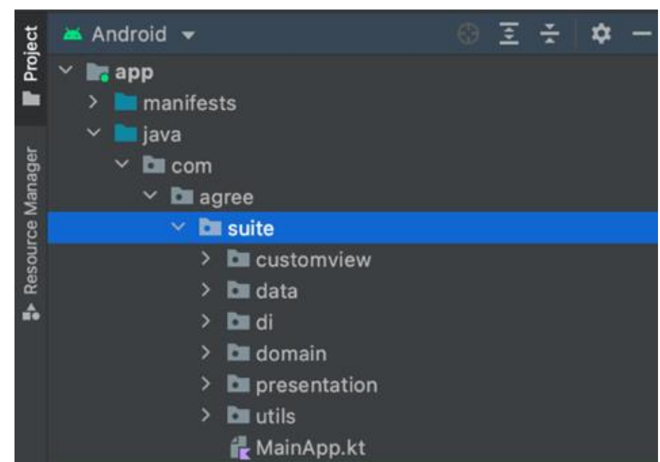
Setelah *sprint* telah selesai dilakukan, maka akan dilanjutkan ke tahap *sprint review*. *Developer* akan melakukan demo *sprint backlog* yang telah selesai dikerjakan kepada *product owner*. Kemudian akan dilakukan inspeksi terkait *sprint backlog* yang belum dapat diselesaikan dalam satu *sprint* 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.

IV. PEMBAHASAN DAN HASIL

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.

Pada pengembangan aplikasi Agree Partner dibuat tiga *package* utama, yaitu *data*, *domain*, dan *presentation* (ditunjukkan pada Gambar 6). *Data package* berisi implementasi *interface repository* dan *data source* yang dapat bersumber dari *local* atau *network*. Lalu pada *domain package* berisi *interface usecase*, serta kelas *interactor* yang merupakan implementasi dari *interface usecase*. Kemudian pada *presentation package* terdiri dari kelas *fragment* dan kelas *viewmodel*.



Gambar 6. Pembagian Package *Data,* Domain, dan *Presentation*

Pembagian ketiga *package* seperti pada Gambar 6 tersebut merupakan salah satu penerapan *Clean Architecture* pada pengembangan aplikasi Android, yaitu membagi kode menjadi tiga *layer*, yaitu *data*, *domain*, dan *presentation*. Setiap *package* berisi kode yang sesuai dengan konsentrasinya masing-masing.

A. Data Package

Data package merupakan *layer* yang mengangani proses pengambilan data baik dari *local* maupun dari *network*. Pada *data package* terdapat banyak *interface* yang berisi fungsi-fungsi pemanggilan API dengan menggunakan *library* Retrofit untuk melakukannya dan terdapat juga kelas-kelas yang mengimplementasikan *interface-interface* tersebut. *Interface* API dan kelas implementasinya dapat dilihat pada Gambar 7 dan Gambar 8.

```
interface InboxApiClient {
    @GET()
    fun getInboxList(
        @QueryMap map: Map<String, String>
    ): Single<PagingApiResponse<List<InboxItem>>>

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

Gambar 7. *Interface InboxApiClient*

Interface pada Gambar 7 merupakan *service* yang digunakan untuk melakukan proses pengambilan dan pengiriman data. *Interface* tersebut digunakan sebagai parameter untuk melakukan instansiasi Retrofit.

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

Gambar 8. Kelas *InboxApi*

Kelas *InboxApi* pada Gambar 8 merupakan kelas yang mengimplementasikan *interface* pada Gambar 7. Kelas ini berisikan detil dari setiap fungsi yang di-*override* dari *interface* pada Gambar 7, serta mengembalikan nilai hasil dari pengambilan dan pengiriman data.

Selain *interface InboxApiClient* dan kelas *InboxApi*, pada *data package* juga terdapat *interface InboxRepository* dan juga kelas *InboxDataStore*. *InboxDataStore* merupakan kelas yang mengimplementasikan *interface InboxRepository*. Pada kelas *InboxDataStore* terdapat dua parameter di konstruktornya, yaitu *InboxAPI* dan *DbService*. Parameter *InboxAPI* digunakan untuk mengambil data dari *network*, sedangkan *DbService* digunakan untuk mengambil data dari penyimpanan lokal. Implementasi dari *interface repository* dapat dilihat pada Gambar 9 dan Gambar 10.

```
interface InboxRepository : DevRepository {

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

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

Gambar 9. *Interface Repository*

Interface InboxRepository pada Gambar 9 memiliki fungsi-fungsi yang berkaitan dengan proses bisnis yang akan diimplementasikan oleh kelas *InboxDataStore* dan juga digunakan sebagai parameter *constructor* kelas *InboxInteractor*.

```
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)
        }
    }
}
```

Gambar 10. Implementasi *Interface Repository*

Kelas *InboxDataStore* berfungsi sebagai *mediator* antara *domain package* dengan *data package*. Kelas *InboxDataStore* akan melakukan pengambilan, penghapusan, pembaruan, atau pembuatan data dan meneruskan data ke *interface InboxRepository*. *Interface InboxRepository* akan digunakan oleh kelas *InboxInteractor* untuk mengambil, membuat, menghapus, atau memperbarui data.

B. Domain Package

Domain package merupakan *layer* yang mengatur komunikasi antara *presentation package* dan *data package*. Pada *package* ini didefinisikan kelas *model*, kelas *interface usecase*, dan kelas *interactor* yang merupakan kelas yang mengimplementasikan *usecase*. *Interface usecase* akan digunakan oleh *viewmodel* untuk melakukan *bisnis logic* seperti mengambil atau mengirimkan data. Sementara itu, kelas *interactor* akan menggunakan *interface repository* untuk melakukan *request* pengambilan atau pengiriman data.

Interface InboxUsecase memiliki fungsi yang berkaitan dengan proses bisnis yang dimiliki oleh perusahaan. *Interface* pada Gambar 11 akan diimplementasikan pada kelas *interactor* pada Gambar 12.

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


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

Gambar 11. *Interface InboxUseCase*

Pada Gambar 12, kelas *InboxInteractor* memiliki beberapa fungsi yang di-*override* dari *interface InboxUsecase*. Kelas *InboxInteractor* memiliki parameter *interface InboxRepository* yang ada pada *data package*, yang mana *interface InboxRepository* tersebut digunakan oleh kelas *InboxInteractor* untuk melakukan pengambilan atau pengiriman data sesuai dengan proses bisnis perusahaan.

```
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)
        }
    }
}
```

Gambar 12. Kelas *Interactor*

Data yang telah didapatkan kemudian dilakukan *mapping* untuk menghindari *property* yang bernilai *null*.

C. *Presentation Package*

Presentation package merupakan representasi dari *presentation layer* yang mana *layer* ini menampilkan data terbaru yang diterima. *Presentation package* terdiri dari kelas *fragment* dan kelas *viewmodel*. Kelas *fragment* akan melakukan *request* melalui fungsi yang ada di kelas *viewmodel*, kemudian fungsi di kelas *viewmodel* akan memanggil fungsi dari kelas *interface usecase*. Data yang dikembalikan oleh kelas *viewmodel* berupa *LiveData*. *LiveData* adalah sebuah kelas penyimpanan data yang dapat diobservasi dan bersifat *lifecycle-aware*. *LiveData* tersebut akan diobservasi oleh *fragment* untuk mendapatkan data terbaru dan kemudian *fragment* akan melakukan *update* tampilan berdasarkan data terbaru 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({
                //melakukan update variable
    _inboxList
            }).let(compositeDisposable::add)
    }

}
```

Gambar 13. Kelas *ViewModel*

Pada Gambar 13, kelas *InboxViewModel* memiliki parameter *interface InboxUseCase* yang akan digunakan untuk melakukan proses bisnis, yaitu mengirim data atau mengambil data. Selain itu, terdapat dua *variable*, yaitu *_inboxList* dan *inboxList*. *Variable _inboxList* merupakan *MutableLiveData*, yaitu *LiveData* yang nilainya dapat diubah. Sementara itu, *variable inboxList* merupakan *LiveData* yang bersifat *immutable*, yaitu nilainya hanya dapat dibaca dan tidak dapat diubah. *MutableLiveData* digunakan di *ViewModel* untuk menampung data terbaru yang kemudian digunakan untuk melakukan instansiasi *LiveData*. Kelas *InboxFragment* akan melakukan observasi pada *variable LiveData* untuk mengambil data terbaru dan kemudian melakukan *update* tampilan berdasarkan data yang diterima.

Kode *vm.getInboxList()* pada Gambar 14 merupakan pengambilan data melalui *variable viewmodel*. *Viewmodel* akan melakukan pengambilan data dan mengirimkan data ke *variable inboxList* yang mana merupakan sebuah *LiveData*.

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


    private val vm by
    sharedViewModel<InboxViewModel>()

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

    vm.inboxList.observe(viewLifecycleOwner) {
        when (it) {
            is VmData.Success -> {
                // Update Tampilan
            }
            is VmData.Empty -> {
                // Update Tampilan
            }
            is VmData.Loading -> {
                // Update Tampilan
            }
            is VmData.Failure -> {
                // Update Tampilan
            }
        }
    }

}
```

Gambar 14. Kelas *Fragment*

Variable inboxList akan diobservasi untuk mengambil data terbaru dan kemudian *fragment* akan melakukan *update* tampilan menggunakan data tersebut. Kode *vm.inboxList.observe(viewLifecycleOwner)* pada Gambar 14 merupakan kode yang digunakan untuk melakukan observasi *variable inboxList* di kelas *InboxViewModel*. Data yang didapatkan digunakan untuk melakukan pembaharuan tampilan.

D. *Dependecy Injection Package*

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 15, *service, interface,* atau *class* yang dibutuhkan akan diinstansiasi dan diinjeksi di dalam fungsi *module* yang merupakan tempat untuk mendefinisikan fungsi-fungsi Koin.

```
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 15. Kode instansiasi *service, interface, class,* dan *viewmodel* menggunakan Koin

Untuk mendefinisikan *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 PT Telkom, maka terdapat perbedaan cara mendeklarasikan *variable* Koin *module*.

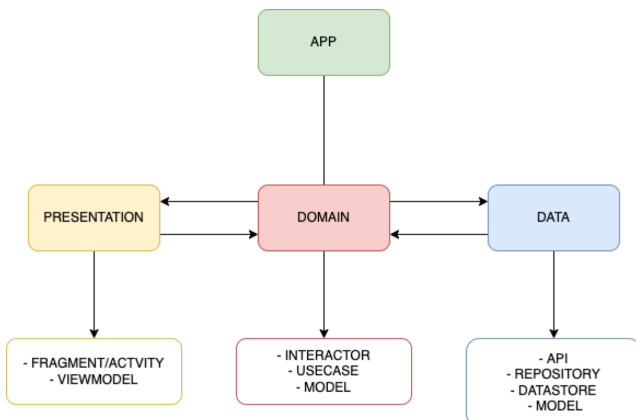
```
class MainApp : DevApplication() {

    override fun defineKoinModules() = arrayListOf(
             libModule,
             apiModule,
             reqresModule
        )
}
```

Gambar 16. Kode deklarasi *module koin*

Pada Gambar 16, *variable* Koin *module* yang telah dibuat dideklarasikan di dalam fungsi *defineKoinModules()* yang di-*override* dari DevApplication() yang merupakan sebuah *abstract class*.

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

Pembuatan fitur *Inbox* dengan menerapkan *design pattern* MVVM dan *Clean Architecture* menghasilkan struktur *package* seperti Gambar 17.



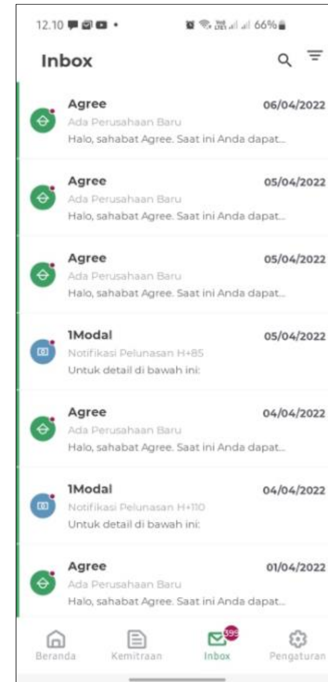
Gambar 17. Struktur *package* dengan menerapkan *design pattern* MVVM dan *Clean Architecture*

Presentation package bertanggung jawab untuk melakukan pembaharuan tampilan berdasarkan data yang telah diperoleh dari *viewmodel*. Di *presentation package* hanya terdapat UI *logic* yang mengatur tampilan pada layar *smartphone*. Sementara itu, *data package* bertanggung jawab untuk proses pengambilan dan pengiriman data ke server.

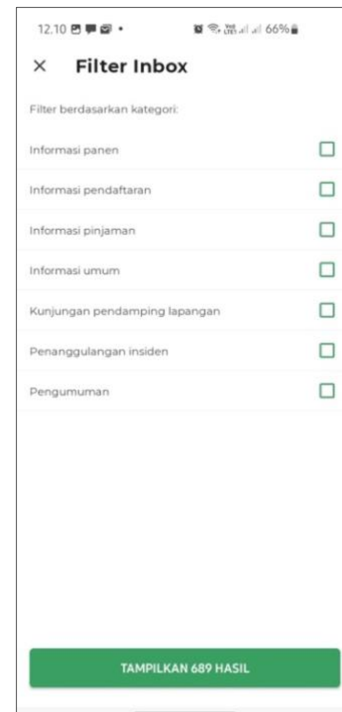
Domain package bertindak sebagai pusat komunikasi antara *presentation package* dan *data package*. *Presentation package* akan melakukan *request* melalui *domain package* dan kemudian *domain package* meneruskan *request* tersebut ke *data package*. Lalu *data package* akan mengembalikan *data* ke *domain package* dan kemudian diteruskan *presentation package*.

Fragment atau *activity* di *presentation package* mendapatkan data melalui *viewmodel* dengan menggunakan *interface usecase* yang ada pada *domain package*. Sementara itu *domain package* mendapatkan data dari *data package* dengan menggunakan *interface repository*. Menggunakan *interface* untuk berkomunikasi antar *package* merupakan salah satu prinsip ketergantungan dari *dependency inversion principle* yang digunakan di *Clean Architecture*.

Fitur *Inbox* adalah fitur yang menampilkan pesan pemberitahuan kepada pengguna. Pesan pemberitahuan tersebut dapat berupa informasi panen, pendaftaran, pinjaman, umum, dan lain-lain. Pada Gambar 18 dapat dilihat contoh tampilan fitur *Inbox* yang menampilkan daftar pesan. Pada fitur *Inbox* terdapat salah satu fitur tambahan yaitu fitur *Filter*. Fitur *Filter* berguna untuk menampilkan daftar pesan berdasarkan kategori yang dipilih (lihat Gambar 19).

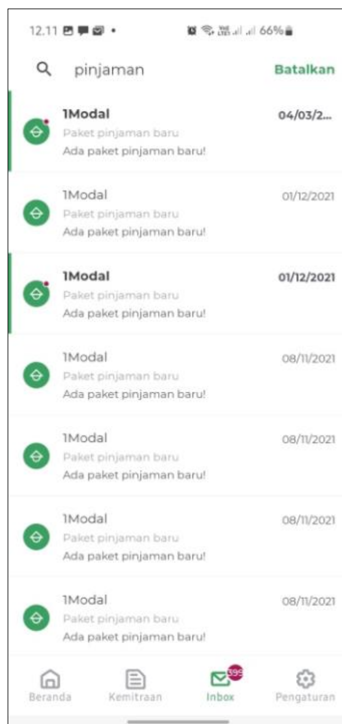


Gambar 18. Tampilan UI fitur *Inbox*



Gambar 19. Tampilan UI fitur *Filter Inbox*

Pada fitur *Inbox* juga terdapat fitur *Search*. Fitur ini berfungsi untuk melakukan pencarian pesan pemberitahuan berdasarkan nama pesan. Gambar 20 merupakan contoh tampilan dari hasil pencarian di fitur *inbox*.



Gambar 20. Tampilan UI fitur *Search Inbox*

V. KESIMPULAN

A. Kesimpulan

Berdasarkan pengembangan aplikasi *mobile* Agree Partner dengan menerapkan *design pattern MVVM* dan *Clean Acrhitecture*, dapat disimpulkan bahwa penerapan *design pattern* MVVM dan *Clean Architecture* membuat proses pengembangan aplikasi Agree Partner menjadi lebih terorganisir dan meningkatkan kualitas proyek. Selain itu, proses perawatan kode atau aplikasi menjadi lebih mudah. Kode juga menjadi lebih mudah untuk dibaca dan dipahami oleh *programmer* lain. Penerapan *Clean Architecture* mengurangi keterikatan *antar class* karena tidak langsung bergantung kepada *class*, melainkan kepada *interface* atau *abstract class*. Adapun dengan diterapkannya *MVVM,* kode antara proses bisnis dan UI dapat dipisahkan, sehingga ketika melakukan *redesign* UI, hanya kode yang berkaitan dengan UI saja yang harus diubah.

B. 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 pengembangan aplikasi.

REFERENSI

[1] "KKP | Kementerian Kelautan dan Perikanan." https://kkp.go.id/djprl/artikel/21045-konservasi-perairan-sebagai-upaya-menjaga-potensi-kelautan-dan-perikanan-indonesia (accessed Mar. 23, 2022).

[2] Kementerian Pertanian, "Pusat Data dan Sistem Informasi Pertanian Sekretariat Jenderal-Kementerian Pertanian Center for Agriculture Data and Information System Secretariat General-Ministry of Agriculture 2020," pp. 1–201, 2020.

[3] M. Catur Yuantari, A. Kurniadi, and F. Kesehatan, "Pemanfaatan Teknologi Informasi Untuk Meningkatkan Pemasaran Hasil Pertanian Di Desa Curut Kecamatan Penawangan Kabupaten Grobogan Jawa Tengah," *Techno.COM*, vol. 15, no. 1, pp. 43–47, 2016.

[4] U. G. Maya, "Bab 2 Sejarah Android," pp. 5–14, 2017, [Online]. Available: http://repository.untag-sby.ac.id/514/3/BAB 2.pdf

[5] "Komponen Arsitektur Android | Developer Android | Android Developers." https://developer.android.com/topic/libraries/architecture (accessed Mar. 10, 2022).

[6] "Dicoding Indonesia." https://www.dicoding.com/academies/129/tutorials/4437 (accessed Mar. 15, 2022).

[7] Sutabri, "Bab Ii Tinjauan Pustaka Dan Dasar Teori," *J. Chem. Inf. Model.*, vol. 53, no. 9, pp. 1689–1699, 2016.

[8] Tian Lou, "A comparison of Android Native App Architecture Master ' s Programme in ICT Innovation A Comparison of Android Native App Architecture – MVC , MVP and MVVM," p. 57, 2016.

[9] Tung Bui Du, "Reactive Programming and Clean Architecture in Android Development," no. April, p. 47, 2017.

[10] "Clean Coder Blog." https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (accessed Mar. 15, 2022).

[11] "Dicoding Indonesia." https://www.dicoding.com/academies/165/tutorials/10289?from=10284 (accessed Mar. 15, 2022).

[12] "Dicoding Indonesia." https://www.dicoding.com/academies/169/tutorials/7830 (accessed Mar. 10, 2022).

[13] V. Makarenko, O. Olshevska, and Y. Kornienko, "an Architectural Approach for Quality Improving of Android Applications Development Which Implemented To Communication Application for Mechatronics Robot Laboratory Onaft," *Autom. Technol. Bus. Process.*, vol. 9, no. 3, pp. 8–13, 2017, doi: 10.15673/atbp.v9i3.714.

[14] H. Bui and I. Technology, "Android Application for Students ' Personal Finances," 2020.