INFOTECH journal
ISSN : 2460-1861 (Print), 2615-4250 (Online)
Vol. 9 No. 1, pp. 132-146

PENERAPAN *CLEAN ARCHITECTURE* DALAM MEMBANGUN APLIKASI BERBASIS *MOBILE* DENGAN *FRAMEWORK* GOOGLE FLUTTER

Mohammad Bijantium Sinatria[1], Oman Komarudin, S.Si., M.Kom[2]., Kamal Prihandani, M.Kom[3]
[1,2,3]Informatika, Fakultas Ilmu Komputer, Universitas Singaperbangsa Karawang
Email: bijantium.sinatria@gmail.com

ABSTRACT
Throughout the building of a mobile app using Flutter, it is mostly composed of widgets, which are a collection of lines of code arranged like a tree. Setting the parent of a widget will re-render from the parent to the child of the widget. Building a mobile-based application with Flutter needs to use the right architecture to easily adapt to changes that occur. When making changes, the possibility of unintentional bugs always exists, the effect is to increase the cost of risk. The right architecture will greatly reduce the expenditure on maintenance and greatly reduce the risk of accidental damage. Therefore, it is required that the right architecture will greatly reduce the expenditure on maintenance and greatly reduce the risk of accidental damage. This research aims to apply Clean Architecture and SOLID Principle in building mobile-based applications using Google Flutter framework. The method in this research uses Agile with Scrum and uses the Google Flutter framework. The result will be a framework and implementation of each Clean Architecture layer by applying SOLID Principle.

Keywords: Clean Architecture, SOLID Principle, Flutter, Maintenance.

ABSTRAK
Saat membangun aplikasi berbasis *mobile* menggunakan Flutter, sebagian besar tersusun atas widget, yang merupakan kumpulan baris kode yang tersusun seperti pohon. Mengatur *parent* dari widget akan melakukan proses *render* ulang dari *parent* hingga *child* dari widget. Membangun aplikasi berbasis *mobile* dengan Flutter perlu menggunakan arsitektur yang tepat agar mudah beradaptasi atas perubahan yang terjadi. Saat melakukan perubahan, kemungkinan terjadi *bug* yang tidak disengaja selalu ada, efeknya menambah biaya risiko. Arsitektur yang tepat akan sangat mengurangi pengeluaran untuk melakukan pemeliharaan dan sangat mengurangi risiko kerusakan yang tidak disengaja. Oleh karenanya diperlukan Arsitektur yang tepat akan sangat mengurangi pengeluaran untuk melakukan pemeliharaan dan sangat mengurangi risiko kerusakan yang tidak disengaja. Penelitian ini bertujuan untuk menerapkan *Clean Architecture* dan SOLID *Principle* dalam membangun aplikasi berbasis mobile dengan menggunakan *framework* Google Flutter. Metode dalam penelitian ini menggunakan *Agile with Scrum* dan menggunakan *framework* Google Flutter. Hasilnya akan berupa sebuah kerangka dan penerapan dari setiap layer *Clean Architecture* dengan menerapkan SOLID *Principle*.

Kata Kunci: *Clean Architecture*, SOLID *Principle*, Flutter, *Maintenance*.

Riwayat Artikel :

Tanggal diterima : 10-05-2023

Tanggal revisi : 11-05-2023

Tanggal terbit : 12-05-2023

DOI :

https://doi.org/10.31949/infotech.v9i1.5237

1. PENDAHULUAN

1.1. Latar Belakang

Aplikasi berbasis *mobile* merupakan perangkat lunak yang berjalan pada perangkat keras berupa *smartphone*, tablet ataupun *smartwatch* sekalipun. Aplikasi *mobile* terdiri dari dua suku kata, aplikasi dan *mobile*. Bisa diartikan juga yaitu, aplikasi adalah perangkat lunak yang dirancang untuk memiliki kegunaan tertentu dan *mobile* adalah berpindah dari satu tempat ke tempat yang lain. Banyak sekali manfaat yang ditimbulkan dari pembuatan aplikasi berbasis *mobile*, baik itu dari bidang akademik, politik hingga kesehatan. Aplikasi *mobile* ini mampu mencakup segala kebutuhan yang diperlukan. Dalam proses pembuatan aplikasi *mobile*, banyak teknologi yang bisa digunakan. *Framework* Google Flutter bisa menjadi salah satu *tools* untuk *developer* dalam membangun aplikasi berbasis *mobile*.

Framework Google Flutter adalah *framework* yang dikembangkan oleh tim di Google. Flutter bertujuan untuk melakukan pengembangan perangkat lunak secara *multiplatform* dengan melakukan proses pemrograman hanya dengan satu *codebase* saja (Santoso et al., 2020). Saat membangun aplikasi berbasis *mobile* menggunakan Flutter, sebagian besar tersusun atas widget, yang merupakan kumpulan baris kode yang tersusun seperti pohon. Mengatur *parent* dari widget akan melakukan proses *render* ulang dari *parent* hingga *child* dari widget. Membangun aplikasi berbasis *mobile* dengan Flutter perlu menggunakan arsitektur yang tepat agar mudah beradaptasi atas perubahan yang terjadi (Boukhary & Colmenares, 2019).

Menurut Robert C. Martin (2017) dari semua aspek sistem perangkat lunak, pemeliharaan adalah yang hal paling mahal dan tidak pernah ada fase untuk berakhir dari fitur-fitur baru, mengatasi *bug* dan evaluasi. Hal tersebut menghabiskan banyak sumber daya manusia. Menurutnya juga biaya utama pemeliharaan adalah *spelunking* dan risiko. *Spelunking* adalah biaya mencari tahu lebih dalam lagi melalui perangkat lunak yang ada, untuk mencoba menentukan strategi terbaik untuk menambah fitur baru atau memperbaiki cacat.

Saat melakukan perubahan, kemungkinan terjadi *bug* yang tidak disengaja selalu ada, efeknya menambah biaya risiko. Arsitektur yang tepat akan sangat mengurangi pengeluaran untuk melakukan pemeliharaan dan sangat mengurangi risiko kerusakan yang tidak disengaja (Robert C. Martin, 2017). Selain hal tersebut, kompleksitas perangkat lunak menurunkan kinerja pemeliharaan karena mengganggu pemahaman fungsionalitas perangkat lunak. Kompleksitas perangkat lunak merupakan faktor kunci dari kinerja pemeliharaan karena mempengaruhi aktivitas pemahaman program, meminimalkan kompleksitas dan kontrol kode, kode dapat dengan mudah digunakan kembali dan dipelihara serta mudah diintegrasikan antar lapisan dan komponen dalam aplikasi (Hussam Hourani et al., 2019).

Sistem perangkat lunak yang baik dimulai dengan kode yang bersih. Prinsip SOLID yaitu *Single Responsibility Principle*, *Open Close Principle*, *Liskov Substitution Principle*, *Interface Segregation Principle*, dan *Dependency Inversion Principle* memberitahu kita bagaimana mengatur fungsi dan struktur data menjadi kelas, dan bagaimana kelas-kelas saling terhubung. Tujuan dari prinsip-prinsip ini adalah pembuatan struktur perangkat lunak tingkat menengah yang mentoleransi perubahan, mudah dimengerti, dan merupakan dasar dari komponen yang dapat digunakan di banyak sistem perangkat lunak (Robert C. Martin, 2017).

Clean Architecture menyediakan metode yang digunakan untuk menyusun aplikasi secara arsitektural dan untuk memecahkan masalah manajemen *state*. Tujuan mendasar dari arsitektur adalah pemisahan fungsional dan skalabilitas. Dengan membagi sistem menjadi lapisan-lapisan yang memisahkan logika bisnis dari implementasi yang lebih spesifik (Boukhary & Colmenares, 2019).

Kemudian, arsitektur ini berusaha untuk menjaga agar perangkat lunak tetap fleksibel dan dapat dipelihara (Sanchez et al., 2022). Berdasarkan pemaparan diatas peneliti akan melakukan penerapan *Clean Architecture* dengan bagian dari Prinsip SOLID pada pengembangan aplikasi berbasis *mobile* menggunakan *Framework* Google Flutter yang dapat diikuti oleh *developer* dalam melakukan pengembangan ataupun pemeliharaan perangkat lunak.

1.2. Tinjauan Pustaka

1.2.1. Aplikasi *Mobile*

Perangkat lunak yang berjalan pada *smartphone* baik itu *tablet*, *handphone*, ataupun *smartwatch* dapat disebut juga dengan istilah *Mobile Apps* dan pada dapat berjalan pada sistem operasi yang mendukung perangkat lunak secara *standalone*. *Platform* pengunduhan aplikasi berbasis *mobile* yang tersedia, biasanya dikelola oleh masing masing *operating system*, seperti Apple App, Google Play, Windows Phone dan BlackBerry App (David, 2017).

Aplikasi berbasis *mobile* untuk sekarang paling populer memiliki sistem operasi Android dan iOS. Sistem operasi Android adalah sistem operasi berbasis Linux yang digunakan untuk pengelola sumber daya perangkat keras, baik untuk *smartphone* dan tablet. Secara umum Android adalah platform yang terbuka bagi para *developer* untuk menciptakan aplikasi mereka sendiri untuk

digunakan oleh berbagai perangkat bergerak (Ahmad Josi, 2019).

Sedangkan iOS adalah sistem operasi perangkat bergerak dari Apple Inc. Sistem operasi ini pertama diluncurkan tahun 2007 untuk iPhone dan iPod Touch, dan telah dikembangkan untuk mendukung perangkat Apple lainnya seperti iPad dan Apple TV. Tidak seperti sistem operasi lainnya, Apple tidak melisensikan iOS untuk diinstal di perangkat keras non-Apple (Rheno Widianto, 2017).

1.2.2. *Framework* Google Flutter

Flutter adalah *Software Development Kit* (SDK) dan *framework* untuk tampilan antarmuka pengguna *mobile* yang terkini dan relatif singkat untuk membangun aplikasi *hybrid*. Flutter dirilis pada tahun 2018, dan dikembangkan oleh Google dan merupakan SDK *open-source*, yang berarti komunitas dapat membantu mengembangkan dan meningkatkan kerangka kerja.

Flutter, seperti *framework* pembuatan aplikasi *hybrid* lainnya, menggunakan basis kode tunggal untuk berbagai platform, seperti *web*, *mobile*, dan desktop. *Framework* ini bertujuan untuk memberikan kinerja yang sama seperti aplikasi *mobile* yang dibangun secara *native*. Flutter juga sebagai pesaing langsung React Native, yang merupakan *framework* untuk pengembangan *hybrid*. Alat alternatif tersedia bagi pengembang untuk mempercepat dan mempermudah pengembangan aplikasi (Allain et al., 2020).

1.2.3. *Clean Architecture*

Clean Architecture menyediakan metode yang digunakan untuk menyusun aplikasi secara arsitektural dan untuk memecahkan masalah manajemen *State*. Tujuan mendasar dari arsitektur adalah pemisahan tanggung jawab dan skalabilitas. Dengan membagi sistem menjadi lapisan-lapisan yang memisahkan logika bisnis dari implementasi spesifik *platform*.

Dengan model arsitektur berlapis, aplikasi *mobile* yang dikembangkan bisa menjadi *framework-agnostic*. Logika bisnis aplikasi tidak bergantung *external framework*. Dalam kasus Flutter, lapisan bisnis aplikasi ditulis dengan bahasa pemrograman Dart tanpa mengetahui keberadaan logika tersebut dalam aplikasi Flutter (Boukhary & Colmenares, 2019).

1.2.4. *Object Oriented Programming*

Object Oriented Programming (OOP) adalah pilar untuk membangun perangkat lunak yang terorganisir sehingga perangkat lunak tediri atas kumpulan objek yang berisi *method* dan *property* yang diberlakukan *class*. *Class* adalah kumpulan dari objek yang memiliki *behaviour* yang sama (Kadek Wibowo, 2015). Ada empat pilar OOP yang dapat digunakan.

Inheritance atau warisan adalah mekanisme menurunkan objek baru dari kelas yang sudah ada sebelumnya atau bisa disebut kelas induk. Dalam kelas turunan dapat menambahkan karakteristik khusus (Pandey Mohan, 2015).

Encapsulation dapat disebut sebagai pembungkus bagian-bagian tertentu atau bisa disebut sebagai *property* dan *method* yang dipunyai objek untuk memberikan akses implementasi dan objek sehingga objek lain harus mengikuti encapsulation yang sudah ditentukan pada *property* dan *method* yang digunakan (Rais, 2019).

Polymorpshism Memungkinkan untuk memanipulasi objek subkelas menggunakan referensi superkelas pada banyak tujuan yang berbeda dengan penamaan yang sama namun memiliki respon yang berbeda (Antani & Stefanov, 2017).

Abstraction sebuah cara untuk mempresentasikan dunia objek *class* sesungguhnya yang kompleks menjadi suatu bentuk model yang sederhana dengan menyembunyikan detail yang tidak penting (Pandey Mohan, 2015).

1.2.5. *SOLID Principle*

Prinsip SOLID memberitahu kita bagaimana mengatur fungsi dan struktur data *class* itu harus saling berhubungan. Tujuan dari prinsip-prinsip adalah tolerir atas perubahan, dan mudah dimengerti.

Single Responsibility Principle (SRP) adalah Prinsip terhadap kelas yang seharusnya memiliki satu tanggung jawab saja, yang tidak berarti bahwa suatu kelas hanya dapat melakukan satu hal (Trodin, 2021).

Open/Closed Principle (OCP) menyatakan bahwa kelas harus terbuka untuk ditambahkan namun tertutup untuk modifikasi (Joshi, 2016).

Liskov Substitution Principle (LSP) yaitu metode untuk mewariskan yang mana kelas turunan harus sepenuhnya mendukung substitusi kelas induk. Setiap kelas turunan harus dapat diganti dengan kelas induknya (Turan & TANRIÖVER, 2018).

Interface Segretion Principle (ISP) yaitu ketika fungsionalitas akan digunakan oleh beberapa kelas, interface untuk kelas harus dibuat. Ini berfungsi untuk menghindari kompilasi ulang dan pemindahan antar komponen (Sanchez et al., 2022).

Dependency Inversion Principle (DIP) Kode yang menerapkan kebijakan tingkat tinggi tidak boleh bergantung pada kode yang mengimplementasikan detail tingkat rendah. Sebaliknya, detail harus bergantung pada aturan (Sanchez et al., 2022).

1.2.6. *Application Programming Interface*

Dalam melakukan komunikasi antara aplikasi berbasis mobile dengan data, *developer* aplikasi bisa menggunakan *API*. *API* adalah antarmuka yang digunakan untuk mengakses aplikasi atau layanan dari sebuah program. Tujuan penggunaannya adalah untuk saling berbagi data antar aplikasi atau *platform* yang berbeda (Hasanuddin et al., 2022).

1.2.7. *Design Pattern* MVVM

Model-View-ViewModel adalah *Design Pattern* yang dapat kita temukan dalam membangun aplikasi berbasis *mobile* baik dibangun menggunakan *native* ataupun *framework*. Sebagian besar kode *logic* berada pada *ViewModel*, dan *ViewModel* harus merepresentasikan status dan perilaku dari tampilan atau dapat disebut *User Interface* (UI). Interaksi pada *Model* dan *View* melalui *ViewModel*. Jika digambarkan dapat seperti gambar dibawah ini (Rahman Fajri & Rani, 2022).

1.2.8. BLoC (*Businnes Logic Component)*

Tujuan dalam penggunaan BLoC adalah tujuannya adalah untuk mangalokasikan semua logika bisnis dari komponen UI khusus kerangka kerja menjadi komponen logika bisnis independen (BLoC) kerangka kerja. Setiap perubahan status aplikasi hanya akan diizinkan untuk dilakukan di dalam BLoC. Dengan cara ini, penerapan Flutter dapat menggunakan BLoC yang sama dan redundansi kode dapat diminimalkan (Sebastian Faust, 2020).

1.2.9. Dio *Package*

Dengan package dio dengan mudah untuk menangani berbagai *request* dengan serentak, dengan memberikan berbagai teknik penanganan *error*. Itu juga memungkinkan *programmers* untuk mencegah *boilerplate code* dan memungkinkan programmers untuk memanggil APIs dengan menuliskan beberapa baris saja (Dilkhaz Y. Mohammed & Siddeeq Y. Ameen, 2022).

1.2.10. GetIt *Package*

Biasanya digunakan untuk memfasilitasi dari *database* ke penggunaan BLoC, hal tersebut didaftarkan sebagai sebuah *singleton* dengan menggunakan GetIt *Package*. GetIt adalah sebuah *Service Locator* untuk Flutter. GetIt *instance* selalu tersedia sebagai sebuah *global variable*, memungkinkan untuk mengakses dengan mudah kepada *children* yang telah didaftarkan pada aplikasi (Goncharuk Nikita, 2021).

1.2.11. *Modularization*

Dart memberikan kemampuan untuk mengembangkan aplikasi secara modular, memisahkan setiap fungsionalitas ataupun fitur pada modul-modul. Sehingga setiap modul memiliki tanggung jawabnya masing-masing. Setiap modul dapat mengakses modul lainnya jika dibutuhkan (Sebastian Faust, 2020).

1.2.12. *Singleton Pattern*

Singleton Pattern adalah sebuah *design patterns* dalam pemrograman berorientasi objek yang memungkinkan sebuah kelas tertentu seharusnya hanya memiliki satu *instance*. *Instance* tersebut yang digunakan kapanpun saat dibutuhkan membutuhkan (Sarcar, 2016).

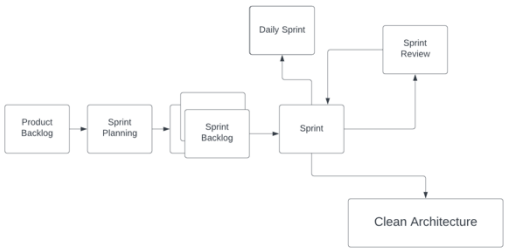
1.2.13. *Maintenance*

Maintenance adalah rangkaian proses dari pemeliharaan perangkat lunak hal tersebut akan dilakukan agar perangkat lunak yang sudah tersedia dapat digunakan secara terus menerus (Christian et al., 2018). 35% dari proses membangun perangkat lunak dihabiskan untuk mencari bagian yang relevan dari kode. Beberapa bagian bahkan dicari berkali kali karena ukuran yang besar dan kompleksitas yang tinggi (Ahrens et al., 2019).

1.2.14. *Framework Scrum*

Scrum adalah kerangka kerja yang memungkinkan pengembangan produk berulang dan inkremental, memungkinkan menyelesaikan sesuatu pada waktu yang tepat, memaksimalkan nilai dari apa yang disampaikan. Tugas dilakukan lebih cepat dan dengan kualitas lebih tinggi oleh tim yang mengatur diri sendiri. Tingkat motivasi diri yang tinggi tercapai dan menjadi alasan mengapa *Scrum* memungkinkan tim mencapai produktivitas yang lebih tinggi dengan lebih cepat (Popli & Chauhan, 2011).

1.3. Metodologi Penelitian

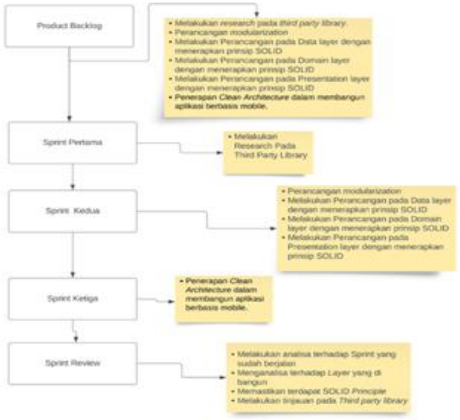


Gambar 1. Tahapan Penelitian

Rancangan penelitian untuk penerapan *Clean Architecture* dalam membangun aplikasi berbasis *mobile* menggunakan *framework* Google Flutter sesuai dengan metodologi penelitian yang sudah ditentukan yaitu menggunakan *Agile with* Scrum tergambar pada gambar 1.

2. PEMBAHASAN

Hasil dari penelitian ini ditunjukan gambar 2 yang berupa kerangka dalam melakukan Penerapan *Clean Architecture* dalam Membangun Aplikasi Berbasis *Mobile* dengan *Framework* Google Flutter.



Gambar 2. Hasil Penelitian

2.1. *Product Backlog*

Setelah melakukan rancangan penelitian, beberapa hal yang dibutuhkan untuk mendukung penelitian ini akan tercantum pada *Product Backlog*. *Product backlog* adalah daftar fitur yang dibutuhkan sebagai bagian dari perencanaan akhir dan merupakan sebuah panduan untuk melakukan perubahan pada penelitian yang akan dilaksanakan. Maka dari itu berikut merupakan *Product Backlog* yang telah ditentukan pada Rancangan Penelitian:

Tabel 1. *Product Backlog*

No	Kebutuhan	Keterangan
1	Melakukan *Research* Pada *Third Party Library.*	*Third Party* Utama yang digunakan yaitu BLoC.
2	Melakukan Perancangan Modularisasi.	Memberikan sebuah gambaran dalam sebuah struktur projek dart.
3.	Melakukan Perancangan pada *Data Layer* dengan menerapkan SOLID *Principle.*	Memberikan gambaran dalam menerapkan SOLID *Principle* pada *Data Layer*.
4.	Melakukan Perancangan pada *Domain Layer* dengan menerapkan SOLID *Principle.*	Memberikan gambaran dalam menerapkan SOLID *Principle* pada *Domain Layer*.
5.	Melakukan Perancangan pada *Presentation Layer* dengan menerapkan SOLID *Principle.*	Memberikan gambaran dalam menerapkan SOLID *Principle* pada *Presentation Layer*.
6.	Melakukan perancangan *Design Pattern* MVVM pada *Data Layer*, *Domain Layer*, dan *Presentation Layer*.	Melakukan pengkodingan dari hasil Perancangan.
7.	Penerapan *Clean Architecture* dalam membangun aplikasi berbasis mobile.	Melakukan tahapan *implementasi hasil dari* langkah perancangan *Data Layer*, *Domain Layer* dan *Presentation*

Agar perencanaan pada tabel 1 dapat direalisasikan sesuai rencana penelitian, penulis membuat aturan yang harus terpenuhi, guna fitur yang dibangun tidak keluar dari *scope* dan tujuan dalam pembuatan ini lebih jelas dan dapat dimengerti oleh pembaca, berikut adalah susunan aturannya.

Tabel 2. Pertanyaan Penelitian

No	Kebutuhan
1	*Third Party* apa yang mendukung *Design Pattern* MVVM dalam membangun aplikasi berbasis *mobile* menggunakan *Framework* Google Flutter ?
2	*Third Party* apa yang dapat mendukung salah satu bagian dari SOLID *Principle* yaitu *Dependency Inversion* dalam menggunakan *Framework* Google Flutter?
3	*Third Party* apa yang dapat mendukung salah satu dari bagian SOLID *Principle* yaitu *Single Responsibility Principle* dalam melakukan proses koneksi HTTP dari *server*?
4	Bagaimana penerapan *Clean Architecture* pada *Design Pattern* MVVM dengan menerapkan bagian SOLID *Principle* dalam *modul authentication* ?

2.2. *Sprint Planning*

Dari rancangan yang telah diusung pada proses *Product Backlog*, peneliti melakukan tahapan untuk membagi beban bekerja agar *Product Backlog* dapat diselesaikan dengan sebaik mungkin. Terdapat tiga *Sprint* pada penelitian ini agar memaksimalkan hasil yang diberikan, pada setiap *Sprint* dilaksanakan selama 14 hari.

Penulis membagi *sprint* menjadi beberapa bagian dilakukan dengan mempertimbangkan beberapa faktor. Pembagian *sprint* memperjelas prioritas pembagian *sprint* mempermudah pemantauan dari *sprint planning* yang telah direncanakan oleh penulis. Dengan membagi *sprint* juga dapat meningkatkan transparansi dalam pengembangan produk penulis secara jelas tahu apa yang harus dilakukan dan bagaimana setiap tugas berkontribusi pada tujuan *sprint* dan proyek secara keseluruhan.

Tabel 3. *Sprint Planning*

No	***Product Backlog***	***Backlog***
1	Melakukan *Research* Pada *Third Party Library.*	Pertama

2	Melakukan perancangan untuk menerapkan modularisasi pada proyek Penerapan *Clean Architecture* Dalam Membangun Aplikasi Berbasis *Mobile* Dengan *Framework* Google Flutter	Kedua
3	Melakukan Perancangan pada *Data Layer* dengan menerapkan bagian dari SOLID *Principle.*	Kedua
4	Melakukan Perancangan pada *Domain Layer* dengan menerapkan SOLID *Principle.*	Kedua
5	Melakukan Perancangan pada *Presentation Layer* dengan menerapkan SOLID *Principle.*	Ketiga
6	Penerapan *Clean Architecture* dalam membangun aplikasi berbasis mobile.	ketiga

2.3. *Sprint* Pertama

Sprint pertama adalah daftar tugas yang harus diselesaikan pada *sprint* pertama dalam penulisan. Salah satu tugas dalam daftar tersebut adalah melakukan analisis terhadap tiga library yang akan digunakan dalam pengembangan aplikasi, yaitu *third party* yang mendukung *design pattern* MVVM, koneksi dengan HTTP, dan *third party* yang mendukung bagian dari SOLID *principle*.

Penulis akan melakukan riset pada *third party* tersebut. Tujuan dari analisis ini adalah untuk mengetahui fitur-fitur apa saja yang disediakan oleh *third party* tersebut, serta kelebihan dan kekurangan dari penggunaannya.

2.3.1. *Third Party Apa Yang Mendukung Design Pattern MVVM Dalam Membangun Aplikasi Berbasis Mobile Menggunakan Framework Google Flutter?*

Pada analisis *third party* menunjukkan bahwa *design pattern* MVVM pada *Framework* Google Flutter didukung dengan berbagai *library state management*, seperti Provider, BLoC, dan GetX, yang masing-masing memiliki kelebihan dan kelemahan, sehingga memungkinkan pengembang untuk memilih *library* yang sesuai dengan kebutuhan proyek.

Tabel 4. Tabel Perbandingan *Third Party* yang mendukung MVVM

No	Fitur	Provider	BLoC	GetX
1	Pendekatan *Statemanagement*	Change Notifier	*Reactive Programming*	*Reactive Programming*
2	Dependency Injection	✓	✗	✓
3	Dokumentasi	Baik	Baik	Baik
4	Popularitas	Populer	Populer	Populer

Dari perbandingan diatas penulis memutuskan menggunakan BLoC sebagai *library state management* yang mewakili *design pattern* MVVM dengan mempertimbangkan dokumentasi, kelengkapan fitur dan dapat mendukung *Dependency Injection* secara terpisah.

```
abstract class CounterEvent {}

class CounterIncrementPressed extends
CounterEvent {}

class CounterDecrementPressed extends
CounterEvent {}

class CounterBloc extends
Bloc<CounterEvent, int> {
  CounterBloc() : super(0) {}
}
```

Gambar 3. Contoh Implementasi BLoC

2.3.2. *Third Party Apa Yang Dapat Mendukung Salah Satu Dari Bagian SOLID Principle Yaitu Single Responsibility Principle Dalam Melakukan Proses Koneksi HTTP Dari Server?*

Salah satu *core* dalam sebuah aplikasi adalah melakukan tahapan integrasi APIs, untuk itu pada *Framework* Google Flutter memerlukan bantuan dari *Third Party*. Dari dokumentasi resmi yang paling popular untuk digunakan ada dua yaitu Dio *package* dan http *Pacakage*. Dua *Third Party* tersebut tentunya memiliki kekurangan dan kelebihan. Namun penulis akan melakukan tahapan perbandingan secara fitur, dan dapat digunakan untuk mewakili salah satu bagian dari SOLID *Principle* yaitu *Single Responsibility Principle*. Hasilnya ditunjukan pada table dibawah ini.

Tabel 5. Tabel Perbandingan *Third Party* yang Mendukung Koneksi HTTP

No	Fitur	Http *package*	Dio *package*
1	*Request Method*	get, post, put, delete, head	get, post, put, delete, patch, head
2	*Error Interceptor*	✗	✓
3	*Base URL*	✗	✓

Pada tabel 5 dapat dilihat, perbandingan antara Dio dan http *package*, keduanya dapat digunakan untuk membantu dalam koneksi HTTP dari server. Namun penulis mendapatkan dua fitur pada Dio *package* yang dapat penulis kembangkan dalam menerapkan bagian dari SOLID *Principle* yaitu *Single Responsibility Principle* pada fitur *Error Interceptor* dan *Base URL*. Dibawah ini adalah potongan kode yang penulis ambil dari dokumentasi cara penggunaan Dio *package*. Pada gambar 4 terdapat contoh penggunaan Dio dari dokumentasi resmi.

```
void main() async {
  var dio = Dio();
  final response = await
dio.get('https://example.com');
  print(response.data);
}
```

Gambar 4. Contoh Implementasi Dio *Package*

2.3.3. *Third Party apa yang dapat mendukung salah satu bagian dari SOLID Principle yaitu Dependency Inversion dalam menggunakan Framework Google Flutter?*

Untuk menerapkan *Dependency Inversion Principle* (DIP) yang merupakan bagian dari SOLID Principle, dapat menggunakan *Third Party*. Sehingga Singleton sudah diatur didalam Third Party tersebut. Ada tiga yang popular pada dokumentasi resmi, berikut spesifikasi dari masing-masing Third Party tersebut.

Tabel 6. Tabel Perbandingan *Third Party* Mendukung *Dependency Inversion*

No	Fitur	GetIt	Kiwi	Injection
1	Terdapat *Register Instance*	✓	✓	✓
2	Terdapat *Lazy Initialization*	✓	✓	✗
3	Konfigurasi Setiap *Platform*	✗	✓	✗
4	Terdapat Dokumentasi yang Mudah Dibaca	✓	✓	✓
5	Tingkat Popularitas	Polular	Sedang	Sangat Popular

Dari spesifikasi diatas penulis hanya membutuhkan *Register Singleton* dan *Lazy Initialization* sehingga GetIt dan Kiwi *package* dapat dipertimbangkan. Karena kedua *package* tersebut penulis memilih GetIt sebagai *third party* yang dapat mendukung *Dependency Inversion*, mempertimbangkan berdasarkan tingkat popularitas dan kebutuhan dari penelitian ini.

Pada dokumentasi resmi GetIt *Package* diciptakan sebagai *Service Locator*, dimana package ini digunakan untuk mendapatkan *instance* dari sebuah *object*. Sebelum mengakses sebuah *object*, daftarkan dalam GetIt pada proses *start-up*. Berikut adalah potongan dari kode pada dokumentasi untuk cara penggunaannya.

```
GetIt getIt = GetIt.instance;

getIt.registerSingleton<AppModel>(AppMod
elImplementation());
getIt.registerLazySingleton<RESTAPI>(()
=>RestAPIImplementation());

// if you want to work just with the
singleton:
GetIt.instance.registerSingleton<AppMode
l>(AppModelImplementation());

GetIt.I.registerLazySingleton<RESTAPI>((
)=>RestAPIImplementation()
);

/// `AppModel` and `RESTAPI` are both
abstract base classes in this example

  print(response.data);
}
```

Gambar 5. Contoh Implementasi GetIt

2.4. *Sprint* Kedua

Setelah melakukan *research* secara keseluruhan pada *third party* yang akan digunakan, penulis akan melanjutkan penelitian untuk membentuk kerangka dari Penerapan *Clean Architecture* Dalam Membangun Aplikasi Berbasis *Mobile* Dengan *Framework* Google Flutter.

2.4.1. Cara Apa Yang Dapat Mendukung Untuk Mempermudah Dalam Membangun Dokumentasi ?

Clean architecture memiliki tiga *layer*, yaitu *domain*, *data* dan *presentation layer*. Modularisasi berfungsi untuk mempermudah dalam melakukan pekerjaan setiap *layer* dan fitur lebih mudah ditemukan, dapat dilakukan proses *development* secara perarel, dengan modularisasi dapat membantu meningkatkan ketahanan aplikasi *mobile* dengan memungkinkan pengembang untuk mengisolasi *bug* atau masalah ke dalam modul tertentu, sehingga tidak menyebar ke seluruh aplikasi selain itu dengan menerapkan modularisasi dapat membantu dalam membangun dokumentasi lebih fokus pada setiap modul tanpa perlu memikirkan modul lain. Pada penulisan ini berfokus pada modul *Authentication*, berikut adalah struktur dari modular pada modul *Authentication*.

Struktur modular dibawah ini terdapat dua *module* yaitu *domain*, dan *data module* besar atau penulis sebut sebagai *parent module*, dan modul yang dibawahinya disebut *child module.* Pada *data module* terdapat *child module datasource*, *mapper*, *model*. Selanjutnya pada *domain module* terdiri dari *entities*, *usecase* dan *repository*. Pada *data* dan *domain module* ini juga mewakili sebagai *data* dan *domain layer* pada *Clean Architecture*.

```
├ domains
  └ authentication_domain
     ├ lib
        ├ data
        │  ├ datasources
        │  ├ mappers
        │  ├ models
        │  │  ├ body
        │  │  └ response
        │  └ repositories
        ├ di
        └ domain
           ├ entities
           │  ├ body
           │  └ response
           ├ repositories
           └ usecases
```

Gambar 6. Struktur Modular *Data* dan *Domain Layer*

Dari gambar 6 *parent module domains* terdapat tiga bagian *child module* yaitu *data module* sebagai *data layer* yang membawahi *datasource* sebagai sumber dari data yang didapat melalui *server*, kemudian ada data *model*, *mapper* dan *repository*. *Child module* berikutnya yaitu *domain* juga sebagai *domain layer* yang membawahi *entities*, *repository* dan *usecases*.

```
├ features
  ├ auth
    ├ lib
      └ presentation
         ├ bloc
         │  ├ sign_in
         └ ui
```

Gambar 7. Struktur Modular *Presentation Layer*

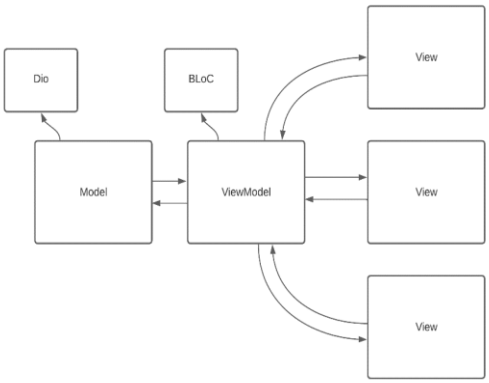
Gambar 7 terdapat parent module yaitu *features* yang didalamnya terdapat presentation *layer* yaitu BLoC yang mewakilkan ViewModel untuk menyimpan seluruh *logic* dari *module authentication* dan UI *module* yang nantinya mewakilkan tampilan untuk pengguna melakukan interaksi pada aplikasi. Selanjutnya gambar 8 ini adalah modul tambahan, pada modul ini penulis fokus untuk menyimpan styling serta utilitas kebutuhan aplikasi, seperti *custom* widget, *extension function* dan *generic class*.

```
├ shared_libraries
  ├ common
  │  ├ lib
  ├ components
  │  ├ lib
  ├ dependencies
  │  ├ lib
  ├ core
     ├ lib
```

Gambar 8. Struktur Modular *Other Layer*

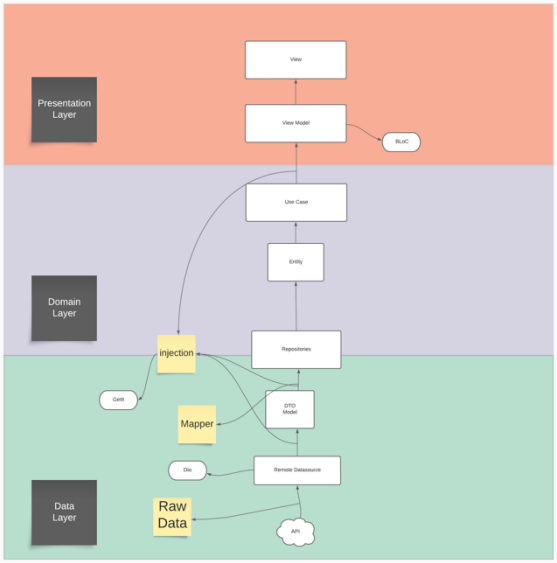
2.4.2. Bagaiamana penerapan *Clean Architecture* pada *Design Pattern* Model-View-ViewModel ?

MVVM dapat diterapkan pada *Clean Architecture*, bagian dari model akan berada pada *data layer*, *view* akan berada pada presentation layer bagian ini akan menampilkan *view* dari aplikasi, selanjutnya ViewModel bagian ini yang akan mencakup semua *logic* pada aplikasi. ViewModel bertugas untuk melakukan manajemen state pada tampilan (*view*) dan memastikan bahwa data yang diperlukan oleh tampilan (*view*) tersedia dan selalu *up-to-date*. ViewModel juga dapat digunakan untuk mengatur interaksi antara tampilan (*view*) dengan model dan menangani operasi yang dilakukan oleh pengguna pada tampilan (view).



Gambar 9. Alur *Design Pattern* MVVM

Gambar 9 adalah bagaimana gambaran dari MVVM, dimana pada Model terdapat juga sebagai *data repository*, terdapat implementasi dari Dio *package* yang membantu koneksi HTTP, lalu terdapat juga *class model* yang bertugas sebagai *business rules* dari aplikasi. Kemudian diteruskan ke ViewModel untuk mengelola *state* ataupun *logic* aplikasi, yang nantinya berguna untuk melakukan interaksi pada View. Berikut diagram alur untuk penerapan *Clean architecture* pada *design pattern* MVVM.



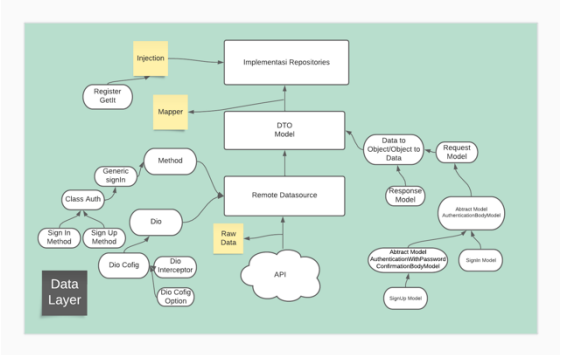
Gambar 10. Gambar Alur Design Pattern MVVM dengan Clean Architecture

Seperti terlihat pada gambar 10, jika menggunakan *clean architecture* memiliki fokusnya masing-masing, sehingga jika ada perubahan pada *data layer*

tidak akan terjadi perubahan secara langsung pada *domain layer* dan *presentation layer*.

2.4.3. Melakukan Perancangan pada Data Layer dengan menerapkan SOLID Principle.

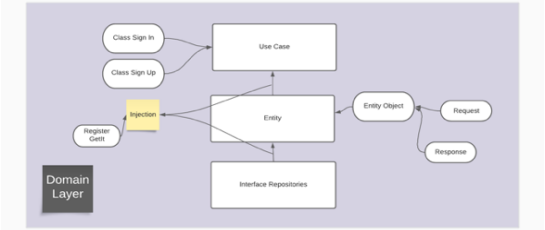
Data layer dapat menggunakan tiga bagian prinsip dari SOLID, yaitu *Single Responsibility Principle* (SRP), *Open Close Principle* (OCP) dan *Liskov Substitution Principle* (LSP). Penulis sudah memisahkan ke dalam sebuah modul pada *directory authentication_domain/lib/data* guna menjaga fokus dari tanggung jawab sebuah *class* (*isolate)* dan memberikan keleluasaan pada saat ingin melakukan tambahan berupa fitur yang sama dengan cara yang berbeda. Berikut gambar 11 menyajikan rancangan *Data Layer*.



Gambar 11. Rancangan *Data Layer*

2.4.4. Melakukan Perancangan pada *Domain Layer* dengan menerapkan SOLID Principle.

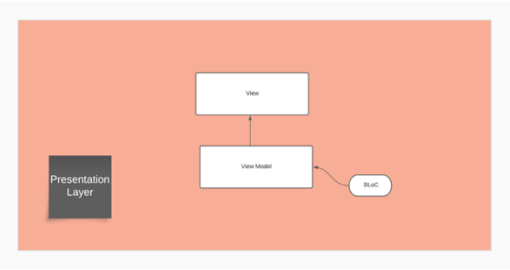
Pada *domain layer* terdiri atas *usecase*, *repository interface*, dan *entity*, berikut adalah rancangan *domain layer*. Ditunjukan pada gambar 12 yang menerapkan *Single Responsibility Principle* (SRP)



Gambar 12. Perancangan *Domain Layer*

2.4.5. Melakukan Perancangan pada *Presentation Layer* dengan menerapkan SOLID Principle.

Pada presentation *layer* bisa menerapkan *Single Responsibility Principle* (SRP) seperti yang ditunjukan pada rancangan gambar 13.



Gambar 13. Perancangan *Presentation Layer*

2.5. *Sprint* Ketiga

Pada penerapan *clean architecture* dalam membangun aplikasi berbasis *mobile* dengan *framework* Google Flutter berfokus pada penerapan *authentication module*. Diawali dengan menuliskan baris kode pada *data layer* selanjutnya *domain layer* dan *presentation layer*. Pada setiap *layer* menerapkan bagian dari SOLID *Principle*. Sebelum menuliskan baris kode pada *datasource*, penulis akan menuliskan baris kode pada *object class model* terlebih dahulu, yang bertugas sebagai menampung data yang didapat maupun dibutuhkan dari *class datasource*. Ditandai *Response* sebagai *object class model* yang akan menampung value *data* kebentuk *object* dan *body* sebagai *object* yang menampung *value* untuk dikirimkan ke *server* melalui APIs. *Class* tersebut dipisahkan menjadi dua bagian *child module* yaitu *response* dan *body module* dengan begitu telah memenuhi *single responsibility principle*.

```
abstract class AuthenticationBodyModel {
  const AuthenticationBodyModel({
    required this.email,
    this.password
  });
    final String email;
    final String? password
}
```

Gambar 14. Penerapan *Authentication Body Model*

Pada gambar 14 menunjukan *class AuthenticationBodyModel* dibuat sebagai *abstract class*, karena *class* ini akan mewarisi kepada *class model* lain sehingga *class* lain hanya perlu menerima warisan tanpa harus membuatnya secara berulang. Karena penulis akan menggunakan *Liskov Substitution Principle* (LSP). Langkah selanjutnya penulis membuat sebuah *model class* yaitu AuthenticationWithPasswordConfirmationBodyMo del yang berisikan turunan dari *model class AuthenticationBodyModel* dapat dilihat pada kode 15 dengan tambahan sebuah *property passwordConfirmation.*

```
abstract class
AuthenticationWithPasswordConfirmationBo
dyModel
extends AuthenticationBodyModel {

AuthenticationWithPasswordConfirmationBo
dyModel({
    required super.email,
    required super.password,
    required this.passwordConfirmation;
});
    final String passwordConfirmation;
}
```

Gambar 15. Penerapan Pada Authentication With Password Confirmation Body Model

Sehingga *model class* yang akan digunakan untuk menampung data dan dikirimkan ke *server* melalui APIs sudah memenuhi bagian dari *Liskov Substitution Principle* (LSP). Berikut kode dari *model class* pada gambar 16 yang menerima warisan dari gambar 17.

```
class SignInWithEmailBodyModel extends
AuthenticationBodyModel {
  SignInWithEmailBodyModel({
  required super.email,
  required super.password
  });
}
```

Gambar 16. Penerapan Pada *Sign In Model*

```
class SignUpBodyModel extends
AuthenticationWithPasswordConfirmationBo
dyModel {
  SignUpBodyModel({
    required super.email,
    required super.password,
    required super.passwordConfirmation,
    required this.name,
  });
  final String name;
}
```

Gambar 17. Penerapan Pada *Sign Up Model*

Selanjutnya penulis akan fokus untuk menuliskan baris kode *remote datasource* yang berfungsi untuk melakukan komunikasi dengan APIs. Penulis menggunakan *method* POST pada fitur *Sign In* dan *Sign Up* sehingga membutuhkan dua *model class* yaitu *Request* dan *Response*. Penulis akan menggunakan *class SignInBodyModel* dan *SignUpBodyModel* untuk mengirimkan data *object* ke APIs dan akan diteruskan ke *server* sehingga mendapatkan *callback* jika sukses maupun gagal melalui *model class SignInResponseModel* dan *SignUpResponseModel* dua *model class* tersebut menangani *data transfer object*.

```
abstract class SignInInterface<T, K> {
  Future<T> signIn({required K
  signInBodyModel});
}
```

Gambar 18. Penerapan *Interface Sign In*

Pada gambar 18 terlihat sebuah *generic class* yang dibuat bertujuan sebagai *common interface* untuk fitur *Sign In*. Nantinya *class* inilah yang akan mewarisi *class* yang memiliki fitur *Sign In* dengan berbagai macam metode. Sehingga dapat memenuhi bagian dari SOLID *Principle* yaitu *Open Close Principle* (OCP).

```
abstract class
AuthenticationRemoteDatasource<T, K>
    implements
SignInInterface<SignInResponseModel,Sign
InWithEmailBodyModel> {

  @override
  Future<SignInResponseModel> signIn({
    required SignInWithEmailBodyModel
signInBodyModel,
  });

  Future<SignUpReponseModel> signUp({
    required SignUpBodyModel
signUpBodyModel,
  });
}
```

Gambar 19. Perancangan Penerapan Abstraksi Authentication Datasource

```
class AuthenticationRemoteDatasourceImpl
    implements
AuthenticationRemoteDatasource {
  const
AuthenticationRemoteDatasourceImpl
({ required this.dio,});

  final Dio dio;

  @override
  Future<SignInResponseModel> signIn(
      {required SignInWithEmailBodyModel
signInBodyModel}) async {}

@override
  Future<SignUpReponseModel>
signUp({ required SignUpBodyModel
signUpBodyModel,}) async {}
}
```

Gambar 20. Kode Penerapan *Authentication Datasource*

Pada gambar 18 dibuatkan *Interface* untuk menerapkan SOLID *Principle* yaitu bagian *Open Close Principle* (OCP), sehingga jika diperlukan untuk membuat fitur *sign in* dengan cara yang berbeda, tidak perlu lagi mengubah kode yang sudah ada pada *class AuthenticationRemoteDatasource* pada gambar 19 dan 20. Namun bisa membuat *class* baru , ini akan mengurangi risiko terhadap kesalahan yang tidak disengaja pada kode yang sudah dibuat sebelumnya. Berikut penulis menyajikan apabila ada fitur *Sign In* dengan metode lain pada kode 21 dan 22.

```
abstract class
AuthenticationOtherRemoteDatasource<T,
K>
    implements
SignInInterface<SignInOtherResponseModel
,SignInOtherBodyModel> {

  @override
  Future<SignInOtherResponseModel>
signIn({
    required SignInOtherBodyModel
signInBodyModel,
  });
}
```

Gambar 21. Penerapan abstraksi *Authentication* pada *Sign In* dengan Metode Lain

```
class SignInOtherRemoteDatasourceImpl
    implements
AuthenticationOtherRemoteDatasource{
const SignInOtherRemoteDatasourceImpl ({
    required this.dio,});

  final Dio dio;

  @override
  Future<SignInOtherResponseModel>
signIn({required SignInOtherBodyModel
sign}) async{}
}
```

Gambar 22. Penerapan *Authentication* pada *Sign In* dengan Metode Lain

Berikutnya, karena kita membutuhkan sebuah *class* yang menjembatani antara *data layer* terhadap *domain layer* yang disebut *repository* dimana pada *data layer* terdapat implementasi dari *repository interface* yang berada pada *domain layer*, maka data *model* tidak dapat digunakan secara langsung pada *class repository*. Karena pada *class repository* akan mengambil *return value* dari *entity* pada *domain layer*. Sehingga butuh suatu *method* yang melakukan tugas untuk *transfer* dari *object model* ke dalam bentuk *object entity*. Gambar 23 adalah baris kode *Mapper*. Baris kode *Mapper* dapat dituliskan pada *response* dan *request model* namun, penulis memisahkan tugas tersebut agar menerapkan bagian dari SOLID *Principle* yaitu *single responsibility principle*.

```
class AuthenticationMapper {
  SignInWithEmailBodyModel

mapSignInWithEmailBodyEntityToSignI
nWithEmailBodyModel(
SignInWithEmailBodyEntity
signInWithEmailBodyEntity) =>
SignInWithEmailBodyModel();
}
```

Gambar 23. *Mapper Authentication*

Setelah *class Mapper* telah dibuat, selanjutnya implementasi dari *repository class* akan lebih mudah untuk diterapkan. Pada *class repository* bertugas untuk menggabungkan berbagai sumber data, bisa dari *remote datasource* ataupun *local datasource* dan bertugas untuk menjembatani antara *data layer* dan *domain layer*. *Interface Repository* sebenarnya berada pada *domain layer* namun implementasinya terjadi pada *data layer* maka dari itu *return value* dari *interface repository* berupa *Entity* pada *domain layer*. Berikut adalah implementasi dari *repository* pada gambar 24.

```
class AuthenticationRepositoryImpl
implements AuthenticationRepository {
  final
AuthenticationRemoteDatasource
authenticationRemoteDatasource;
  final AuthenticationMapper mapper;


  AuthenticationRepositoryImpl({
    required
this.authenticationRemoteDatasource,
    required this.mapper,
  });
}
```

Gambar 24. Implementasi Respository

Setelah *class repository implementation* yang berada pada *data layer* sudah melakukan tugasnya, selanjutnya langkah untuk membuat *Entity* pada *domain layer*. *Entity* benar-benar independen dari *layer* lainnya. Karena *data layer* akan memenuhi kontrak dari *entity* dan *presentation layer* akan bergantung pada kontrak tersebut. Setelah melalui beberapa tahap, ini tahap terakhir sebelum masuk ke *presentation layer*, pada *use case class* ini ditulis harus secara *independent* tidak terpengaruh hal lain dari *layer* lain seperti gambar 25 dan 26.

```
class SignInResponseEntity {
  SignInResponseEntity({
    required this.meta,
    required this.token,
  });

  final MetaResponse meta;
  final String token;
}
```

Gambar 25. Entity Sign In

```
class SignUpBodyEntity {
  SignUpBodyEntity({
    required this.name,
    required this.email,
    required this.password,
    required
this.passwordConfirmation,
  });
  final String name;
  final String email;
  final String password;
  final String passwordConfirmation;
}
```

Gambar 26. *Entity Sign Up*

Pada *usecase* bertugas untuk menerapkan *application business rules* pada aplikasi. Selanjutnya kita akan melakukan *injection* terhadap *instance* dari *class* menggunakan GetIt *package*, dengan begitu kita sudah memenuhi bagian dari SOLID *Principle* yaitu *Dependency Inversion*.

```
class SignInUsecase extends
UseCase<SignInResponseEntity,
SignInBodyEntity> {
  final AuthenticationRepository
authenticationRepository;
  const SignInUsecase({
    required
this.authenticationRepository,});

  @override
  Future<Either<FailureResponse,
SignInResponseEntity>> call(
          SignInBodyEntity params) async
=> authenticationRepository.sigIn(
        signInBodyEntity: params,
      );
}
```

Gambar 27. *Usecase*

```
class AuthenticationDomainInjection {
  AuthenticationDomainInjection() {
    _registerAuthenticationDatasource();
    _registerAuthenticationMapper();
    _registerAuthenticationRepository();
    _registerAuthenticationUsecase();
  }
}
```

Gambar 28. Penerapan *Dependency Inversion*

Setelah *data layer* dan *domain layer* sudah siap, maka gunakan lah BLoC *package* yang bertugas mewakili ViewModel untuk menyimpan *logic* aplikasi. Penggunaan BLoC secara *default* sudah menerapkan *Single Responsibility Principle* (SRP) sehingga penggunaan BLoC sudah mewakili bagian dari SOLID *Principle*. Penerapan baris kode BLoC ditunjukan pada gambar 29. Baris kode tersebut mewakili bagian dari MVVM yaitu ViewModel. Setelah penulisan baris kode BLoC sudah selesai langkah terakhir adalah mengimplementasikan antara BLoC dengan *view* yang sudah dibuat. Pada *class* bagian tampilan (*view)* penulis juga membagi lagi widget yang digunakan berulang kali menjadi bagian tersendiri hal ini dilakukan mengingat untuk menerapkan bagian dari SOLID *Principle*.

```
abstract class SignInState {}

class SignIn extends SignInEvent {
  final SignInWithEmailBodyEntity
signInWithEmailBodyEntity;
  SignIn({
    required
this.signInWithEmailBodyEntity,
  });
}

class EmailnPasswordChecker extends
SignInEvent {}

class SignInBloc extends
Bloc<SignInEvent, SignInState> {
  final SignInWithEmailUsecase usecase;

  SignInBloc({required this.usecase}) :
super(SignInInitial()) {
    on<SignIn>((event, emit) async {}
 }
}
```

Gambar 29. Penerapan *Logic* pada BLoC

Penerapan BLoC untuk menyimpan *state* atau bagian dari ViewModel ini akan berada pada *presentation layer* bagian tampilan (*view*), penulis menggunakan *BlocListener* untuk memperhatikan perubahan *state* yang terjadi. Jika ada perubahan pada *state* maka tampilan (*view*) akan langsung berubah.

```
BlocListener<SignInBloc, SignInState>(
          listener: (context, state) {

if (state is EmailAndPasswordIsEmpty) {
} else if (state is EmailIsEmpty) {

 } else if (state is
EmailAndPasswordCompleted) {
  } else if (state is SignInLoading) {
  } else if (state is SignInSuccess) {
   }
  }
);
```

Gambar 30. Penerapan BLoC pada Bagian Tampilan (*view)*

2.6. *Sprint Review*

Tahap terakhir adalah *sprint review*, setelah melakukan tugas pada *sprint*, hasil dari langkah *sprint* berupa penerapan *clean architecture* dengan SOLID *Principle* yang bertujuan untuk melakukan pengembangan perangkat lunak menggunakan *framework* Google Flutter lebih mudah untuk di-*maintenance*. Berikut adalah ringkasan data berdasarkan *questioner* yang diisi oleh ahli untuk detail dari pertanyaan dapat dilihat pada lampiran

Tabel 7. Hasil Validasi Ahli

Pertanyaan	Ahli		
	1	2	3
RQ1			
Apakah Hasil dari Penelitian Penulis Sudah Menerapkan Clean Architecture?	Ya	Ya	Ya
Apakah Hasil dari Penelitian Penulis Terdapat Data Layer pada Penerapan Clean Architecture?	Ya	Ya	Ya
Apakah Hasil dari Penelitian Penulis Terdapat Domain Layer pada Penerapan Clean Architecture?	Ya	Ya	Ya
Apakah Hasil dari Penelitian Penulis Terdapat Presentation Layer pada Penerapan Clean Architecture?	Ya	Ya	Ya
Apakah Hasil dari Penelitian Penulis Sudah Menerapkan MVVM?	Ya	Ya	Ya
Apakah hasil dari Penelitian Penulis terdapat MVVM dalam Penerapan Clean Architecture?	Ya	Ya	Ya
RQ2			
Apakah Pada Penelitian ini Sudah Terdapat Bagian dari SOLID Principle?	Ya	Ya	Ya
Apakah Penggunaan BLoC Mendukung Single Responsibility Principle ?	Ya	Ya	Ya
Apakah dalam Penelitian ini Sudah Menerapkan BLoC yang Berperan Sebagai ViewModel?	Ya	Ya	Ya
Apakah Penggunaan GetIt Mendukung Dependency Inversion?	Ya	Ya	Ya
Apakah dalam Penelitian ini Sudah Menerapkan GetIt yang Berperan Sebagai Dependency Inversion Pada Prinsip SOLID?	Ya	Ya	Ya
Apakah dalam Penelitian ini Sudah Menerapakan Dio Sebagai Salah Satu *Third Party* HTTP *Connection* yang Mendukung Single Responsibility Principle?	Ya	Ya	Ya
Apakah dalam Penelitian ini Sudah Menerapkan Dio?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Data Layer Terdapat Bagian dari SOLID Principle?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Pada Data Layer Terdapat Open Close Principle?	Ya	Ya	Ya
Dari hasil penelitian ini apakah Data Layer Terdapat Single Responsibility Principle?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Data Layer Terdapat Liskov Substituion Principle?	5/5	5/5	4/5
Seberapa Mudah Data Layer Untuk Menghadapi pengembangan kembali (*Maintenance*) ?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Domain Layer Terdapat Bagian dari SOLID Principle?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Domain Layer Terdapat Single Responsibility Principle?	5/5	5/5	
Seberapa Mudah Domain Layer Untuk Menghadapi Pengembangan Kembali (*Maintenance*) ?	Ya	Ya	Ya
Dari Hasil Penelitian ini Apakah Presentation Layer Terdapat Bagian dari SOLID Principle?	5/5	5/5	4/5
Apakah Mudah Presentation Layer Untuk Menghadapi pengembangan kembali (*Maintenance*) ?	Ya	Ya	Ya
Seberapa Mudah Melakukan Proses Pengembangan Kembali (*Maintenance*) Menggunakan Clean Architecture dan SOLID Principle?	Ya	Ya	Ya
Apakah dalam penerapan Modularisasi Memudahkan Proses Pengembangan Kembali (Maintenance) ?	Ya	Ya	Ya
Apakah dalam penerapan Modularisasi Dapat Mempermudah dalam Membuat Dokumentasi ?	Ya	Ya	Ya
Dari hasil Penelitian ini, Apakah Penulis Konsisten Dalam Menerapkan Clean Architecture dan SOLID Principle?	5/5	5/5	4/5
Menurut Saudara Ahli, Seberapa Konsisten peneitian ini Menerapkan Clean Architecture dan SOLID Principle?	Ya	Ya	Ya

3. KESIMPULAN

Berdasarkan penelitian yang telah dilakukan yaitu Penerapan *Clean Architecture* Dalam Membangun

Aplikasi Berbasis *Mobile* dengan *Framework* Google Flutter, maka dapat disimpulkan bahwa Penelitian ini menghasilkan langkah dalam membangun aplikasi berbasis *mobile* dengan *Framework* Google Flutter. Dengan menerapkan *Clean Architecture* dapat menjadi solusi ketika melakukan pemeliharaan pada baris kode yang telah *existing*.Membangun aplikasi berbasis *mobile* dengan *Clean Architecture* dan menerapkan bagian dari SOLID *Principle* akan memudahkan ketika mengalami perubahan atau penambahan sebuah fitur pada aplikasi.

PUSTAKA

Afif Abdillah, M. (2022). *APLIKASI PENGELOLA KEUANGAN BERBASIS WEBSITE PADA* [Fakultas Teknologi dan Informatika, Universitas Dinamika]. https://repository.dinamika.ac.id/id/eprint/6092/1/18410200043-2022-UniversitasDinamika.pdf

Ahmad Josi. (2019). *Sistem Operasi (Konsep dan Perkembangan Sistem Operasi)*.

Ahrens, M., Schneider, K., & Busch, M. (2019). Attention in software maintenance: An eye tracking study. *Proceedings - 2019 IEEE/ACM 6th International Workshop on Eye Movements in Programming, EMIP 2019*, 2–9. https://doi.org/10.1109/EMIP.2019.00009

Allain, H., Mario Di Francesco, P., & Mario Di Francesco Advisor, P. (2020). *Improving productivity and reducing costs of mobile app development with Flutter and Backend-as-a-Service*. https://aaltodoc.aalto.fi/handle/123456789/97522

Antani, V., & Stefanov, S. (2017). *Object-oriented JavaScript : learn everything you need to know about object-oriented JavaScript with this comprehensive guide*.

Boukhary, S., & Colmenares, E. (2019). A clean approach to flutter development through the flutter clean architecture package. *Proceedings - 6th Annual Conference on Computational Science and Computational Intelligence, CSCI 2019*, 1115–1120. https://doi.org/10.1109/CSCI49370.2019.00211

Christian, A., Hesinto, S., Patra No, J., Sukaraja Kecamatan Prabumulih Selatan, K., & Selatan STMIK Prabumulih, S. (2018). Rancang Bangun Website Sekolah Dengan Menggunakan Framework Bootstrap (Studi Kasus SMP Negeri 6 Prabumulih). *Jurnal SISFOKOM*, *07*. https://doi.org/10.32736/sisfokom.v7i1.278

David. (2017). *PENGEMBANGAN APLIKASI MOBILE OBJEK WISATA SECARA REAL TIME DENGAN AUGMENTED REALITY DI KABUPATEN SUMBA BARAT DAYA*. http://e-journal.uajy.ac.id/11939/

Dilkhaz Y. Mohammed, & Siddeeq Y. Ameen. (2022). Developing Cross-Platform Library Using Flutter. *European Journal of Engineering and Technology Research*. https://doi.org/10.24018/ejeng.2022.7.2.2740

Goncharuk Nikita. (2021). *Development of the mobile application for university enrollees using Flutter* [TOMSK STATE UNIVERSITY (NR TSU)]. https://vital.lib.tsu.ru/vital/access/manager/Repository/vital:14419

Hasanuddin, Asgar, H., & Hartono, B. (2022). RANCANG BANGUN REST API APLIKASI WESHARE SEBAGAI UPAYA MEMPERMUDAH PELAYANANDONASI KEMANUSIAAN. *Jurnal Informatika Teknologi Dan Sains*, *4*. https://doi.org/10.51401

Hussam Hourani, Hiba Wasmi, & Thamer Alrawashdeh. (2019). A Code Complexity Model of Object Oriented Programming (OOP). *Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. https://doi.org/10.1109/JEEIT.2019.8717448

Joshi, B. (2016). Beginning SOLID principles and design patterns for ASP.NET developers. In *Beginning SOLID Principles and Design Patterns for ASP.NET Developers*. Apress Media LLC. https://doi.org/10.1007/978-1-4842-1848-8

Kadek Wibowo. (2015). ANALISA KONSEP OBJECT ORIENTED PROGRAMMING PADA BAHASA PEMROGRAMAN PHP. In *Jurnal Khatulistiwa Informatika* (Vol. 3, Issue DESEMBER). https://doi.org/10.31294/jki.v3i2.1662.g1214

Pandey Mohan, H. (2015). *OBJECT-ORIENTED PROGRAMMING C++ SIMPLIFIED*.

Popli, R., & Chauhan, N. (2011). SCRUM: AN AGILE FRAMEWORK. *International Journal of Information Technology*, *4*(1), 147–149. http://www.csjournals.com/IJITKM/PDF%204-

1/30.Rashmi%20Popli1%20&%20Naresh%20Chauhan2.pdf

Rahman Fajri, A., & Rani, S. (2022). Penerapan Design Pattern MVVM dan Clean Architecture pada Pengembangan Aplikasi Android (Studi Kasus: Aplikasi Agree Partner). *Official Scientific Journals of Universitas Islam Indonesia*.

Rais, M. (2019). Penerapan Konsep Object Oriented Programming Untuk Aplikasi Pembuat Surat. *Jurnal PROtek , 06*(2). https://doi.org/http://dx.doi.org/10.33387/protk.v6i2.1242

Rheno Widianto, S. (2017). Rancang Bangun Aplikasi Telemedika untuk Pasien Diabetes Berbasis Platform iOS. *JURNAL MULTIMEDIA NETWORKING INFORMATICS*, *2*(1), 20. https://doi.org/https://doi.org/10.32722/multinetics.v3i1.1083

Robert C. Martin. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*.

Sanchez, D., Rojas, A. E., & Florez, H. (2022). Towards a Clean Architecture for Android Apps using Model Transformations. *International Journal of Computer Science*, *Vol. 49*(Issue 1). https://www.iaeng.org/IJCS/issues_v49/issue_1/IJCS_49_1_28.pdf

Santoso, S., Surjawan, D. J., & Handoyo, E. D. (2020). Pengembangan Sistem Informasi Tukar Barang Untuk Pemanfaatan Barang Tidak Terpakai dengan Flutter Framework. *Jurnal Teknik Informatika Dan Sistem Informasi*, *6*(3). https://doi.org/10.28932/jutisi.v6i3.3071

Sarcar, V. (2016). *Java design patterns : a tour of 23 gang of four design patterns in Java*.

Sebastian Faust. (2020). *Using Google´s Flutter Framework for the Development of a Large-Scale Reference Application*. https://epb.bibl.th-koeln.de/frontdoor/deliver/index/docId/1498/file/flutter-for-the-dev-of-large-scale-ref-app.pdf

Trodin, M. (2021). *Applying the SOLID principles to JavaScript's React library*. http://uu.diva-portal.org/smash/record.jsf?pid=diva2%3A1598927&dswid=3847

Turan, O., & TANRIÖVER, Ö. Ö. (2018). An Experimental Evaluation of the Effect of SOLID Principles to Microsoft VS Code Metrics. *AJIT-e: Online Academic Journal of Information Technology*, *9*(34), 7–24. https://doi.org/10.5824/1309-1581.2018.4.001.x