Optimasi Pengembangan Aplikasi Cross-platform Berbasis Flutter Menggunakan Pendekatan Arsitektur Model MVI (Model-View-Intent)

Muh Alif Al Gibran Arif[1], Dana Sulistyo Kusumo[2], Shinta Yulia Puspitasari[3]

[1,2,3] Universitas Telkom, Bandung
alifalgibran@students.telkomuniversity.ac.id[1], danakusumo@telkomuniversity.ac.id[2], shintayulia@telkomuniversity.ac.id[3]

Abstrak

Permasalahan yang sering muncul dalam membangun sebuah aplikasi berbasis mobile yang cukup besar adalah terlalu banyaknya kelas yang ada yang dapat menyebabkan pengembang kesulitan dalam memodifikasi kelas terkait terutama pada tahap pengembangan selanjutnya. Pada zaman ini, telah banyak *framework* dalam membangun perangkat lunak berbasis *mobile* yang cukup handal. Salah satu yang populer adalah Flutter yang memiliki prinsip *fast building* serta sangat *flexible* memungkinkan aplikasi dapat dengan cepat diselesaikan dengan pendekatan apapun. Selain itu, Flutter juga memungkinkan para pengembang untuk membangun aplikasi *cross-platform* hanya dengan satu *codebase* saja. Namun, menggunakan Flutter saja tidak cukup cepat dalam menyelesaikan sebuah aplikasi, diperlukan pendekatan model arsitektur yang handal agar para pengembang dapat mengefektifkan *state management* serta mereduksi kelas yang ada sehingga pengembangan dikemudian hari dapat dengan mudah dilakukan. Pada penelitian ini akan dibangun sebuah model arsitektur yang dapat digunakan oleh para pengembang aplikasi mobile berbasis Flutter yang menggunakan pendekatan arsitektur model MVI (model-view-intent) dengan prinsip *Single Source of Truth*. Dengan menerapkan konsep MVI pada Flutter diharapkan para pengembang dapat dengan mudah mengembangkan aplikasi serta pada tahap pengembangan selanjutnya pengembang mudah memodifikasi kode programnya. Metodologi penelitian dilakukan dengan membandingkan tingkat *maintainability* model arsitektur yang dibangun dengan model *default* pada Flutter menggunakan *CK Metrics.*

Kata kunci: Flutter, MVI, *Maintanability, single source of truth*

Abstract

The problem that often arises in building a mobile-based application that is quite large is that there are too many classes that can cause developers to find it difficult to modify the related classes, especially at the maintenance stages of development. Nowadays, there are many frameworks for building reliable mobile-based software. One of the popular ones is Flutter which has a fast building principle and is very flexible, allowing applications to be quickly completed with any approach. In addition, Flutter also allows developers to build cross-platform applications with just one codebase. However, using Flutter alone is not fast enough to complete an application, a reliable architectural model approach is needed so that developers can streamline state management and reduce existing classes so that future development can be easily carried out. In this study, an architectural model will be built that can be used by Flutter-based mobile application developers using the MVI (model-view-intent) architectural approach with the Single Source of Truth principle. By applying the MVI concept to Flutter, it is hoped that developers can easily develop applications and at the next development stage developers can easily modify the program code. The research methodology was carried out by comparing the maintainability level of the architectural model built with the default model on Flutter using CK Metrics.

Keywords: : Flutter, MVI, *Maintanability, single source of truth*

5. Pendahuluan

Latar Belakang

Konsep OOP sudah sering digunakan didunia pengembangan perangkat lunak, namun seringkali masalah yang dihadapi oleh aplikasi adalah sulitnya melakukan maintanance. Beberapa aktifitas yang harus dilakukan termasuk menambahkan fitur, menghapus beberapa kode program, mengoreksi error dan lain sebagainya [1]. Oleh karena itu, penting untuk melakukan optimasi yang merupakan suatu aktifitas untuk meningkatkan kualitas perangkat lunak baik dengan meningkatkan desain atau dengan meningkatkan *maintainability* selama proses pengkodean [1]. Dengan hal tersebut kode program akan lebih sederhana dan lebih mudah untuk di-*maintain* seiring perubahan yang dilakukan [1]. Terutama pada aplikasi berbasis mobile, diperlukan manajemen file serta kode program untuk mendorong kualitas dari sebuah aplikasi berbasis mobile [2]. Untuk mengukur Salah satu cara mengukur *maintainability* dari program aplikasi adalah menggunakan CK metrics [3].

Terdapat banyak model arsitektur dalam pengembangan aplikasi berbasis mobile [4]. Salah satunya adalah MVI (model-view-intent) yang diperkenalkan oleh Cycle.js. MVI memungkinkan developer untuk memisahkan bagian model dengan view serta intent yang berfungsi mengatur action pada perubahan model yang mampu menyederhanakan *state management* program sehingga dapat mempercepat proses pengkodean program [5]. Selain itu, kelebihan utama pada MVI adalah prinsip Single Source of Truth (SSOT) yang menempatkan kode inti pada satu file sehingga penulisan program yang lebih sederhana dan mudah dimodifikasi serta menjamin semua data dapat direfleksi yang dapat memudahkan proses *maintenance* dikemudian hari [5].

Dalam pengembangan perangkat lunak berbasis mobile, terdapat banyak SDK ketika kita ingin membangun aplikasi berbasis mobile. SDK yang cukup populer pada saat ini adalah Flutter. Flutter merupakan open-source SDK untuk membuat aplikasi cross platform dengan *high-fidelity* untuk Android, iOS bahkan Web App hanya dengan sekali pengkodean saja [6]. Flutter juga mengedepankan prinsip *fast building* dan bersifat *open-source* dengan basis *state management* [7] sehingga sangat cocok digunakan oleh pengembang yang ingin mengembangkan aplikasi dengan cepat. Karakteristik dari Flutter yang bersifat reaktif dan *flexible (open-source)* memungkinkan para pengembang aplikasi dapat mengadopsi pendekatan model arsitektur seperti MVI yang menunjang aspek *maintainability* suatu aplikasi [6].

Topik dan Batasannya

Berdasarkan latar belakang diatas, salah satu masalah dalam membangun aplikasi adalah sulitnya untuk melakukan *maintenance* [1] program oleh karena itu pada penelitian ini akan berfokus pada optimasi dalam sisi *maintanabilty* aplikasi.

Scope yang pertama adalah bagaimana membangun *Template Code* berdasarkan prinsip MVI pada aplikasi berbasis Flutter. MVI ini memiliki kelebihan tersendiri jika dilihat dari konsep *Separation of Concern* nya, MVI mampu menyederhanakan *state management* pada program aplikasi [5]. Selanjutnya, dari MVI tersebut akan diukur aspek *maintainability* dengan melakukan *benchmark* dengan beberapa *framework* yang umum pada aplikasi berbasis Flutter menggunakan *Assessment Maintainability Metrics*.

Sebagai batasannya, *framework* MVI yang dibangun hanya dirancang untuk aplikasi berbasis Flutter sehingga untuk *tools* dalam membangun aplikasi berbasis mobile lainnya, tidak dapat menggunakan *template code* ini.

Tujuan

MVI memiliki ciri khas dalam *penyederhanaan state* [5], dimana *state management* ini merupakan *concern* dalam pengembangan aplikasi berbasis Flutter. Dalam penelitian ini, akan dikembangkan *template code* dengan menerapkan arsitektur model MVI pada aplikasi berbasis Flutter untuk meningkatkan aspek *maintainability* program. Untuk mengukur aspek *maintainability* dikembangkan aplikasi sederhana yang dibangun berdasarkan *template code* lalu diukur dengan *Assessment Maintainability Metrics*.

Dengan menggunakan *template code* yang dibangun diharapkan para pengembang aplikasi berbasis Flutter dapat dengan mudah melakukan *maintenance* program aplikasi.

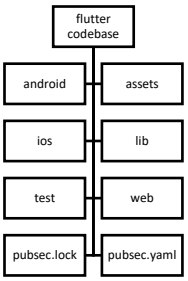
Organisasi Tulisan

Sebagai awalan dalam *paper ini* diberikan fakta serta masalah dalam membangun aplikasi berbasis mobile terutama dalam sisi aspek *maintainability.* Selanjutnya dijelaskan metodologi penelitian dengan mengukur aspek *maintainability* program, baik itu yang *framework* dibangun (MVI) maupun *framework* pembandingnya (no-Pattern). Dan yang terakhir adalah kesimpulan berupa hasil dari penelitian ini.

6. Studi Terkait

6.1 Flutter

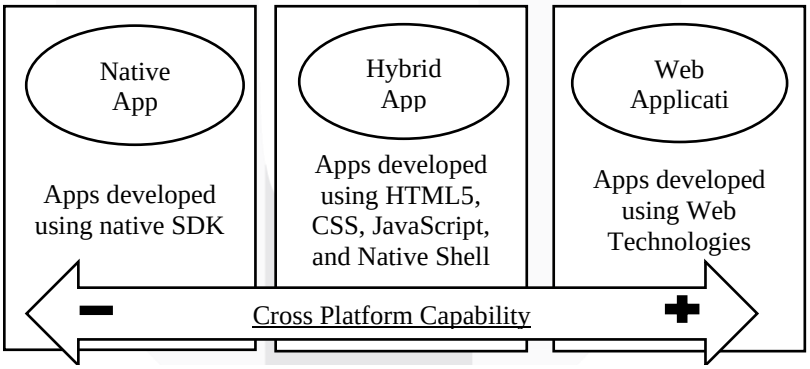
Flutter merupakan UI Google Toolkit untuk membangun aplikasi *cross-platform* yang dikompilasi secara native untuk mobile, web dan desktop dalam satu codebase. Flutter dibangun menggunakan bahasa Dart yang mirip penggabungan bahasa JavaScript dan Java [7]. berikut merupakan kelebihan yang dapat diberikan oleh Flutter [7]. Berikut merupakan struktur dasar dari projek Flutter:



Gambar 1. Struktur Dasar Projek Flutter

- Android: Berisi file dan folder untuk konfigurasi platform bersistem operasi Android, seperti permission dan third-party library.
- Assets: Berisi file gambar dengan bermacam-macam tipe sebagai bahan pada antarmuka.
- Ios: Sama seperti fungsi folder Android, namun konfigurasinya untuk sistem operasi iOS
- Test: Berisi file untuk melakukan unit test pada fungsi program yang telah dibuat.
- Pubseck.lock dan pubsec.yaml: Merupakan file yang berfungsi untuk mengimport third-party library, pada file inilah kita mendefinisikan library apa saja yang ingin kita gunakan pada aplikasi yang dibangun.
- Lib: Pada folder inilah tempat mengimplementasikan kode yang kita tulis, dalam folder ini juga arsitektur yang dibangun berada.

Sedangkan *cross-platform* merupakan istilah dalam dunia rekayasa perangkat lunak untuk aplikasi yang dapat berjalan pada beberapa jenis *device* yang berbeda. untuk mengetahui kapabilitas dari *cross-platform* berikut merupakan gambaran perbandingan strategi pembangungan aplikasi dan menunjukan *multiplatform*-nya, tingkat kesesuaian meningkat dari kiri kekanan [8]:



Gambar 2. Kapabilitas Cross-Platform

Aplikasi *hybrid* mengombinasikan kelebihan dari kedua tipe dari teknik pengembangan aplikasi (native dan web) dan merupakan pilihan terbaik untuk membangun aplikasi *cross-platform.* Oleh karena itu, aplikasi yang dibangun dari *framework* yang berbasis *cross-platform* sering juga disebut aplikasi hybrid [9].

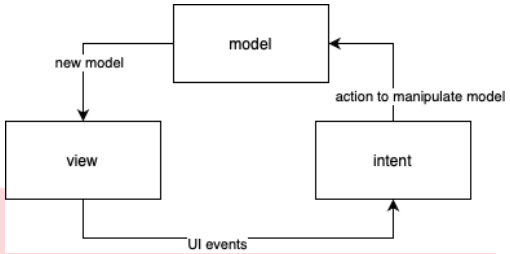
6.2 Model-View-Intent (MVI)

Model-View-Intent dibuat oleh Cycle.Js terinspirasi dari dua JavaScript framework yaitu Redux dan React. Namun, kedua framework tersebut mengadopsi pendekatan MVC yang diperkenalkan oleh Trygve Reenskaug pada 1979 untuk memisahkan view dari model [5].

Pada saat ini, Cycle.Js memperkenalkan *Architect*ure MVI yang memperjelas antar *states* dengan antarmuka. MVI dapat dengan mudah mengatur siklus aliran data yang mana merupakan permasalahan umum dalam proses pengkodean aplikasi. Berikut merupakan penjelasan antara Model, View dan Intent [5]:

- Intent
  Intent pada MVI mengelola masukan dari user (seperti click event) atau bahkan aplikasi itu sendiri dan menerjemahkannya menjadi sesuatu yang dapat menjadi parameter dari Model.

- Model
  Model mengambil output dari Intent sebagai masukan untuk memanupulasi model. Praktik *single source of truth* diterapkan dalam model untuk semua *state* yang ada. Output dari fungsi ini adalah state yang berubah, perubahan state tersebut hanya dilakukan pada satu tempat. Hal Ini berfungsi agar aliran data dapat dengan mudah dideteksi ketika ada state yang berubah sehingga memudahkan pengembang untuk mengatur aliran data pada aplikasi. State yang baru merupakan output dari Model.
- View
  View mengambil hasil output dari Model yaitu model yang baru untuk ditampilkan pada antarmuka.



Gambar 3. Flow MVI

6.3 Singe Source of Truth

Single Source of Truth (SSOT) merupakan praktik dalam mengkonstruksi model informasi dan skema data sedemikian rupa sehingga setiap elemen hanya berada pada satu tempat yang sama atau dengan kata lain pemusatan source-code dalam satu tempat [10]. Jika diimplementasikan pada pengembangan perangkat lunak, SSOT dapat mereduksi panggunaan kelas. Berikut merupakan kelebihan lain dari SSOT [10]:

- Mengeliminasi duplikasi data entries.
- Secara substansial mengurangi waktu yang dihabiskan untuk mengidentifikasi kelas yang diinginkan.
- Entitas atau informasi yang mudah ditemukan.

6.4 Assessment Maintainability Metrics

Terdapat lima metrics yang digunakan untuk mengukur aspek maintainability pada framework dalam membangun aplikasi berbasis Flutter. Berdasarkan Chidamber & Kemerer (CK), kelima metrics ini dapat mengukur aspek nilai maintainability pada suatu system atau program aplikasi berbasis objek [11]. berikut merupakan penjelasan dari setiap *metrics* yang digunakan:

Tabel 1. Mainainability Metrics

Metrics	**Definisi & Tujuan**	**Interpretasi *metrics***	***Assessed Attribute***
WMC	Weight Method per-Class sederhananya adalah jumlah method yang didefinisikan atau diimplementasikan didalam kelas . Tujuannya untuk memprediksi kelas dalam *maintainability* dan *reusability*.	Nilai semakin rendah framework semakin baik	*Class/method*
RFC	Respons sets For Class adalah jumlah method yang berpotensi dieksekusi (langsung atau tidak langsung) dalam response terhadap pesan dari sebuah objek dari sebuah kelas atau dari beberapa method didalam kelas tersebut. Tujuannya untuk mengukur kompleksifitas dalam sebuah kelas dalam hal pemanggilan *method* didalam kelas.	Nilai semakin rendah framework semakin baik	*Class/method*

Metrics	Definisi & Tujuan	Interpretasi *metrics*	*Assessed Attribute*
LCOM	Kohesi mengacu pada seberapa dekat sebuah operasi berjalan dalam suatu kelas terkait secara satu sama lain. Tujuannya untuk mengukur apakah kelas pada sistem memastikan semua *method* yang bekerja bersama untuk mencapai satu tujuan yang terdefinisi dengan baik.	Nilai semakin rendah framework semakin baik	*Class/method/attribute global*
CBO	CBO mengukur interdependence dari dua objek. CBO dari sebuah kelas diukur dengan menghitung jumlah kelas lain yang terkait/coupled dengan kelas tersebut. Tujuannya untuk mengetahui hubungan antar-kelas.	Nilai semakin rendah framework semakin baik	*coupling*
DIT	Depth of Intheritance of Tree in Class dalam hierarki pewarisan didefinisikan sebagai panjang maksimum sifat/method dari parent class ke node/child class paling ujung. Tujuannya untuk mengukur jumlah keturunan langsung dari kelas.	Nilai semakin rendah framework semakin baik	*inheritance*

7. Tahapan Penerapan MVI pada Flutter

7.1 Identifikasi kebutuhan

Dalam membangun sistem, Langkah pertama yang dilakukan adalah mengidentifikasi kebutuhan dalam rangka mengimplementasikan arsitektur model MVI kedalam Flutter. Berdasarkan *flow* dari MVI berikut merupakan kebutuhan untuk setiap entitas pada MVI yang disesuaikan terhadap karakteristik Flutter:

7.1.1 Intent

Fungsi *intent* dalam MVI adalah sebagai *trigger* dalam perubahan *state. Intent* membutuhkan *trigger* baik itu dari *user* atau dari aplikasi itu sendiri.

7.1.2 Model

Implementasi *single source of truth* diterapkan didalam *model* yang bertanggung jawab dalam perubahan *state*. Namun, *model* membutuhkan *action* untuk mengubah *state, action* memiliki tugas untuk menerima *intent* lalu mengubahnya menjadi *state* yang baru. Untuk menunjang perubahan *state* ini, dibutuhkan sebuah *tools/library* untuk kegiatan manajemen *state* tersebut.

7.1.3 View

Dalam rangka mengubah *state, view* akan menampung kondisi tampilan terhadap setiap perubahan *state* yang diberikan. Oleh karena itu, *view* membutuhkan *state* hasil dari *action* pada model.

7.2 Tools/library yang dibutuhkan

Perubahan *state* pada *model* membutuhkan manajemen *state* yang baik. Pada penelitian ini, manajemen state menggunakan library Get[1]. Get mampu menyederhanakan *state management* sehingga dapat menunjang *action* dalam rangka melakukan perubahan *state.*

7.3 Identifikasi penggunaan arsitektur model MVI pada Flutter

Tujuan dari penelitian ini adalah membuat *template code* dengan pendekatan arsitektur model MVI pada Flutter untuk menunjang aspek *maintainability*. Oleh karena itu, perlu adanya cara agar para pengembang dapat menggunakan *template code* yang dibangun. Dari masalah tersebut maka dibangun dua bagian utama, yaitu:

7.3.1 Application

[1] https://pub.dev/packages/get

Para pengembang melakukan pengkodean pada bagian ini, *requirement* yang dibuat oleh pengembang akan diimplementasikan mengikuti *template code* arsitektur model MVI.

7.3.2 Framework

Pada bagian ini bertugas untuk memastikan *template code* yang dibangun sesuai dengan flow pada MVI sehingga pengembang mampu mengimplementasikan *requirement* aplikasi sesuai dengan prinsip MVI yang diimplementasikan pada bagian Application. Untuk memastikan flow tersebut maka dibutuhkan beberapa bagian didalam framework:

7.3.2.1.1 Core

Bagian ini bertugas dalam mastikan *model-view-intent* saling beruhubungan. Sehingga, proses perpindahan antar entitas MVI dapat berjalan sesuai dengan flow

7.3.2.1.2 Model

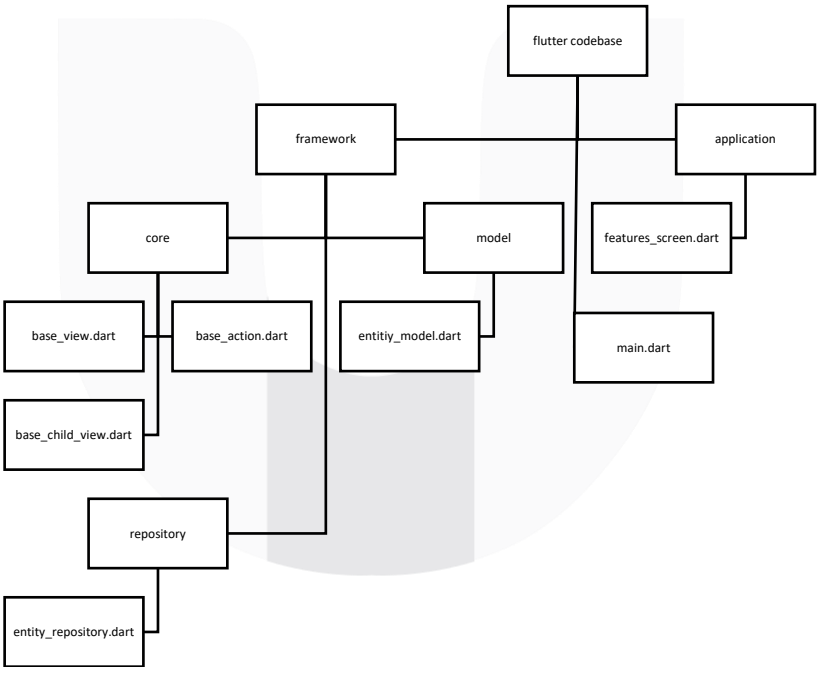
Dalam melakukan pertukaran data dengan API aplikasi biasanya membutuhkan entitas dalam menerjemahkan data dari API lalu di-*decode* kedalam bahasa Dart. Bagian ini bertugas merepresentasikan data agar dapat dijadikan sebagai objek.

7.3.2.1.3 Repository

Seperti aplikasi yang berhubungan pada API pada umumnya, dibutuhkan layer yang dapat melakukan interaksi dengan API agar data dari API dapat diterjemahkan menjadi entitas objek. Bagian ini bertujuan untuk menyelesaikan permasalahan tersebut.

7.4 Rancangan sistem

Dari kebutuhan yang telah didefinisikan di atas maka berikut merupakan konsep dari arsitektur pada aplikasi yang dibangun menggunakan Flutter:



Gambar 4. Struktur besar perancangan sistem

7.4.1 Application

Pada folder Application ini, *file* yang bertugas menampilkan semua *view* yang akan atau kemungkinan diakses oleh user didefinisikan. Sebagai contoh untuk *requirement* login, maka login_screen.dart (penamaan file bebas) berada dalam folder ini. Untuk setiap file yang mendefinisikan *requirement* harus membuat tiga kelas dalam file tersebut. Adapun fungsi dari tiga kelas tersebut adalah:

```dart
class AdjustmentData {
  final int grain;
  final int rack;
  AdjustmentData(this.grain, this.rack);
}

class AdjustmentBehavior
    extends CoreBehavior<AdjusmentView, AdjustmentBehavior, AdjustmentData> {
  @override
  Future<AdjustmentData> initState() async {
    return AdjustmentData(0, 0);
  }
}

class AdjusmentView
    extends CoreView<AdjusmentView, AdjustmentBehavior, AdjustmentData> {
  @override
  AdjustmentBehavior initBehavior() {
    return AdjustmentBehavior();
  }

  @override
  Widget loadScreen(
      BuildContext context, AdjustmentBehavior behavior, AdjustmentData state) {
    return Scaffold();
  }

  @override
  Widget onLoadingView(BuildContext context) {
    return CircularProgressIndicator();
  }
}
```

Gambar 5. Implementasi MVI pada requirement

Gambar di atas merepresentasikan satu *requirement*, artinya dalam satu *requirement* agar mengikuti *flow* MVI maka ketiga kelas di atas harus dideklarasi. Berikut merupakan penjelasan dari tiga kelas tersebut:

7.4.1.1 Data class (model)

Dalam kelas ini bagian model pada MVI didefinisikan. Karena *data class* merepresentasikan *model*, maka harus bersifat *immutable*. Artinya, *state* hanya akan diubah pada satu tempat (*single source of truth)*. Berikut merupakan implementasinya didalam kode program:

```dart
class AdjustmentData {
  final int grain;
  final int rack;
  AdjustmentData(this.grain, this.rack);
}
```

Gambar 6. Representasi Model (MVI)

kelas ini merupakan representasi *state* yang bersifat *immutable*. ini bertujuan agar ketika terdapat perubahan *data*/*state* kita dapat mengetahui perubahan *state* tersebut. tujuannya adalah *flow data* lebih *centralize*, *cleaner* dan *more structured.* untuk perubahan *state* dilakukan didalam kelas *behavior* sebagai representasi dari *action.* Karena merupakan *model* yang merupakan *source*, maka kelas *data* tidak meng-*extends* kelas apapun.

7.4.1.2 Behavior class (action)

Untuk mengelola setiap *action* yang ada dalam rangka men-*trigger* perubahan *state*, kelas ini bertugas dalam mengelola *action* yang diberikan. Berikut merupakan implementasinya kedalam kode program:

```
@override
Future<AdjustmentData> initState() async {
  await prices.getPrices().then((querySnapshot) {
    querySnapshot.docs.forEach((doc) {
      _rack = doc["rack"];
      _grain = doc["grain"];
    });
  });

  return AdjustmentData(
    _grain,
    _rack,
  );
}
```

Gambar 7. Representasi *Action* (MVI)

Setelah mendapatkan data dari API, *state* baru didefinisikan dengan memasukan atribut ke dalam parameter *data class*. ketika *data* pada *model* akan diakses oleh *view* maka *view* dapat memanggil *data class* tersebut.

7.4.1.3 *View class* (*view* dan *intent*)

Di dalam *view* terdapat dua *method* yang berupa *state*, *method* tersebut adalah *onLoadScreen* (ketika *screen* sedang diload) dan *loadScreen* (*screen* telah selesai diload). Tujuan dibuatnya dua *method* ini adalah karena setiap *screen* pasti akan melakukan kedua hal tersebut. Oleh karena itu, kedua hal tersebut dijadikan *state* awal pada view. Selanjutnya dengan menggunakan praktik *single source of truth*, maka *state/data* dapat diakses langsung pada satu tempat yaitu pada kelas *data* (gambar 5). Berikut merupakan contoh pola akses data oleh *view:*

```
@override
Widget loadScreen(
    BuildContext context, AdjustmentBehavior behavior, AdjustmentData state) {

  _controllerGrain.text = state.grain.toString();
  _controllerRack.text = state.rack.toString();
```

Gambar 8. Praktik *Single Source of Truth* (MVI)

Dapat dilihat dari gambar diatas bahwa data class dapat diakses langsung oleh view karena model bersifat *single source of truth*. Setelah mengetahui pola perubahan *state* pada *model* MVI di Flutter. Selanjutnya peran *intent* dalam perubahan *state* adalah dengan memberikan *action*. Berikut merupakan contoh *action* yang mengubah *state:*

```
child: ElevatedButton(
  style: ButtonStyle(
    foregroundColor:
        MaterialStateProperty.all<Color>(Colors.white),
  ), // ButtonStyle
  onPressed: () {
    behavior.changeFocusAndSaveData(
        !behavior.isFocus,
        int.parse(_controllerGrain.text),
        int.parse(_controllerRack.text));
  },
  child: Text('Simpan'),
), // ElevatedButton
```

Gambar 9. Representasi *Intent* dan *View* (MVI)

Pada kelas *view* diatas, *intent* berupa *onPressed* yang akan mentrigger *action.* secara teknis *action* merupakan fungsi yang berpotensi untuk mengubah *state* menjadi *state* yang baru.

```
void changeFocusAndSaveData(bool f, int grain, int rack) async {
  AdjustmentData(grain, rack);
  await prices.setPrices(grain, rack);
  render();
}
```

Gambar **10. Perubahan state oleh action**

Fungsi (*action*) di atas akan mengubah *state*, sehingga terdapat *state* baru. Dari gambar tersebut kelas *data* dipanggil kembali untuk dibuat objeknya sebagai *state* yang baru. fungsi *render* akan me-*rebuild screen* sehingga *state* yang baru dapat ditampilkan pada *view.*

7.4.2 Framework

Folder ini berisi *class* utama dalam pembangunan model arsitektur MVI yang berfungsi untuk memastikan setiap entitas pada MVI saling berhubungan agar pengembang dapat melakukan pengkodean sesuai dengan prinsip MVI pada bagian Application. Berikut merupakan penjelasan bagian framework ini:

7.4.2.1 Core

Agar kelas-kelas (*data, behavior, view*) pada suatu *requirement* didalam bagian Application dapat berhubungan satu sama lain maka bagian core ini berfungsi sebagai jembatan penghubungnya, berikut merupakan *parent classes* penghubung tersebut.

7.4.2.1.1 CoreBehavior.dart

CoreBehavior berfungsi untuk mengatur *action* dari *intent*, setiap perubahan *state* akan diolah pada kelas yang meng-*extends* kelas ini, sehingga kelas ini menghubungkan antara *data/state class* dengan *action*. Dalam kelas inilah dibutuhkan *library* Get untuk menyederhanakan manajemen *state*. Contoh perubahan state adalah ketika user memberikan sebuah perintah yang mana perintah tersebut kita akan sebut sebagai *intent*. *Intent* tersebut menginginkan perubahan data pada *view*, *behavior* akan mengolah dan meneruskan *intent* tersebut ke *model* lalu melakukan perubahan *state* untuk menampilkan perubahan *data* tersebut.

7.4.2.1.2 CoreView.dart

Setiap *view/screen* akan meng-*extend* kelas ini. Kelas ini memiliki hubungan yang kuat dengan kelas *behavior* atau dengan kata lain menghubungkan *view* dengan *action* yang di-*trigger* oleh *intent* dari *user*. Kelas ini bertugas untuk mengatur keadaan *view* pada *screen*. Sebagai contoh, sebelum *view* ditampilkan kepada user, kita menginginkan sebuah *data* dari *model* untuk ditampilkan di-*view* tersebut.

7.4.2.1.3 CoreChildView.dart

Kelas ini hanya dibutuhkan ketika sebuah parent view memiliki child view (biasanya berupa dialog atau bottomsheet) yang ingin meng-extend behavior atau action dari parent view-nya.

7.4.2.2 *Model*:

Folder ini berisi file kelas entitas-entitas data yang diperlukan pada aplikasi. Entitas-entitas tersebut digunakan sebagai bentuk konkrit suatu dari objek. Sebagai contoh objek mobil, mobil memiliki bagian ban, pintu dan lain-lainnya. Bagian-bagian tersebut dijadikan atribut dalam kelas entitas mobil.

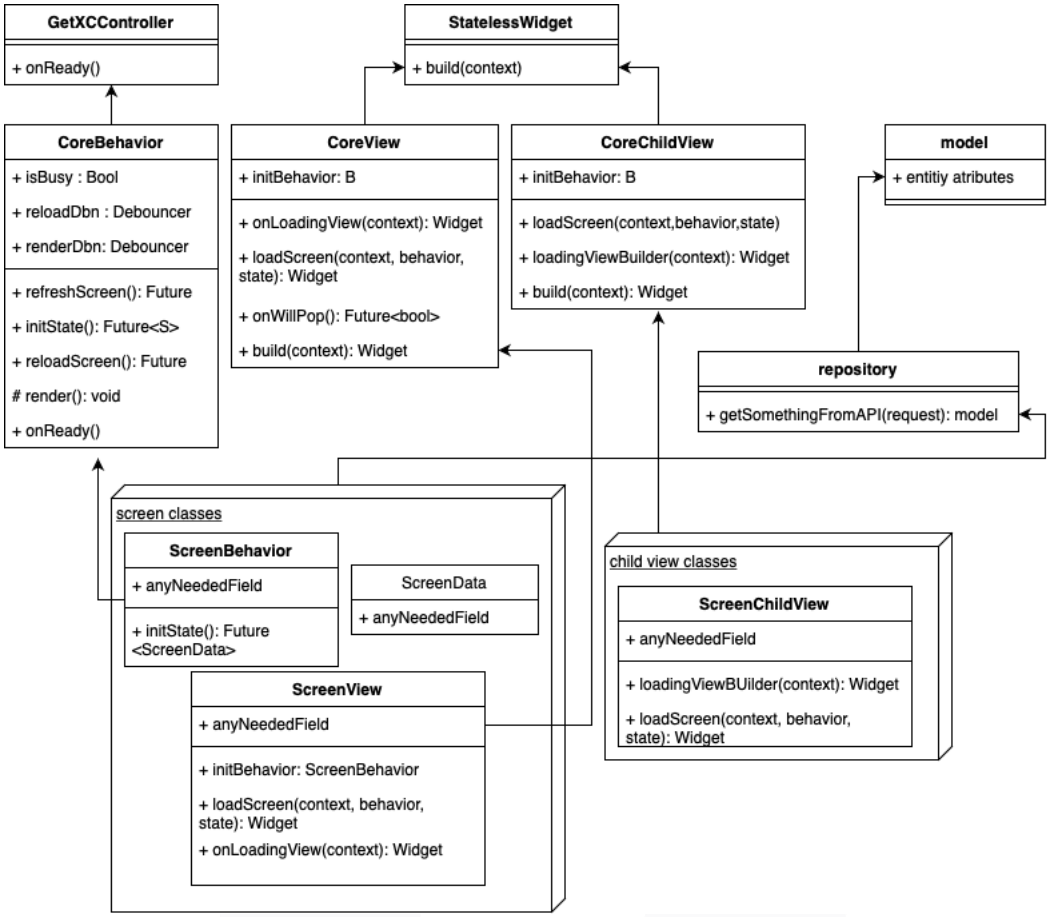
7.4.2.3 *Repository*:

Kelas ini sebagai representasi entitas yang memiliki repository. Tugas dari folder ini menyimpan semua file yang akan berinteraksi dengan API. Sebagai contoh, data mobil pada API akan ditampilkan. Data yang tersebut di-*decode* kedalam bentuk entitas sehingga data tersebut menjadi objek mobil.

7.4.2.4 Main.dart

file ini merupakan file yang pertama kali dijalankan oleh *compiler*, file ini otomatis dibuat ketika projek pertama kali diinisiasi. Fungsi pada kelas Main dimodifikasi untuk menginsiasi kelas-kelas model arsitektur yang dibuat.

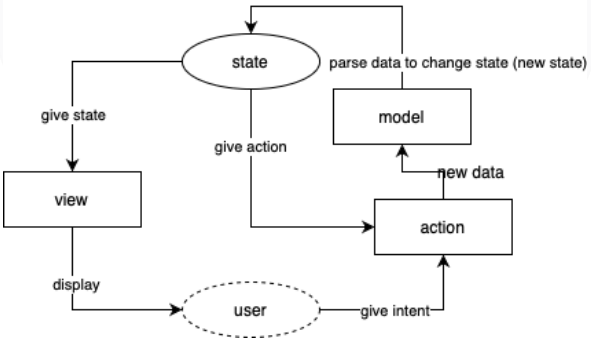
Dari penjelasan rancangan system di atas, berikut merupakan bentuk kelas diagram yang memperlihatkan keseluruhan *template code* yang dibangun:



Gambar 11. Struktur Utama Arsitektur MVI

Sebagai penekanan, bagian Application mencakup bagian *package screen classes* dan *child view classes* sedangkan bagian Framework adalah *model, repository, CoreBehavior*, *CoreView*, dan *CoreChildView*.

Dari implementasi model MVI pada Flutter, berikut merupakan bentuk flow MVI setelah diterapkan pada Flutter:



Gambar 12. Flow MVI pada Flutter

Jika dilihat perubahannya terdapat penekanan entitas yaitu *state* yang diperjelas, hal ini karena flutter sangat *concern* terhadap *state management* sehingga perlu dilakukan manipulasi terhadap struktur flow MVI. *State* dapat berubah jika model baru hasil *output* dari *action* dibutuhkan untuk ditampilkan pada *view.*

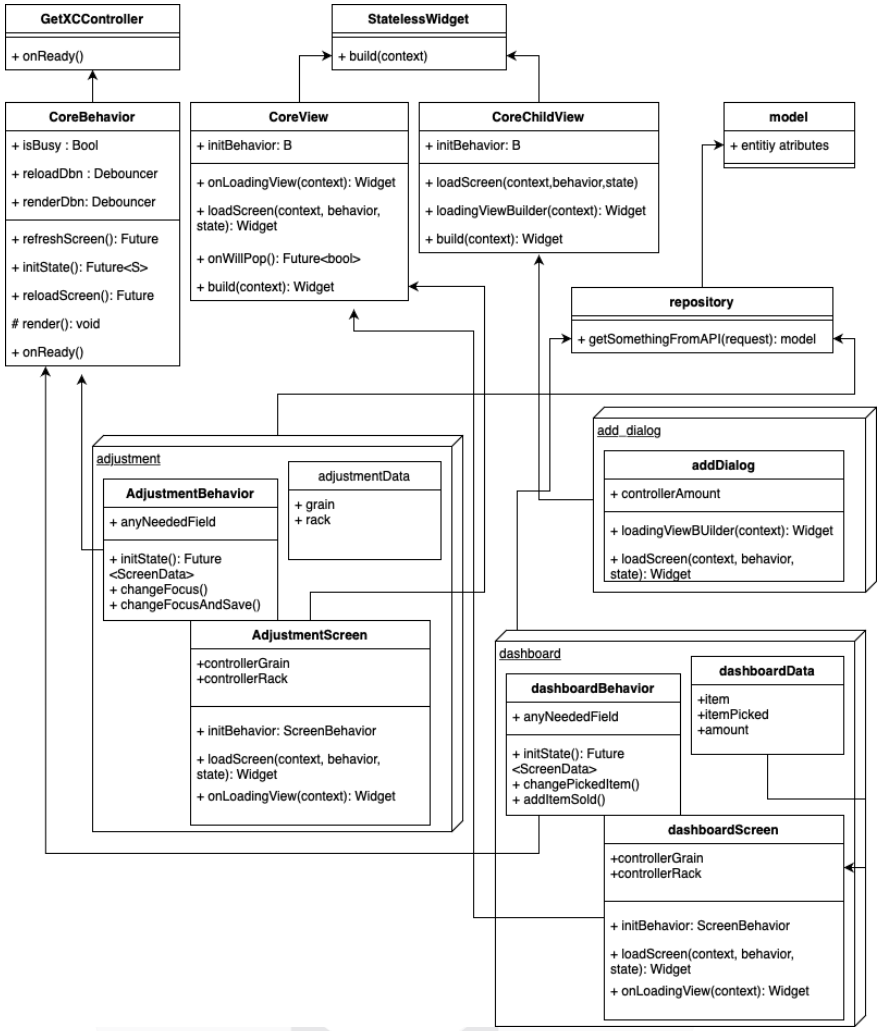
8. Evaluasi

Setelah sistem usulan dibangun dan sistem pembandingnya telah disiapkan maka selanjutnya adalah melakukan pengujian. Pengujian dilakukan dengan mengimplementasikan model arsitektur MVI dan model *default* dari Flutter dengan membuat aplikasi yang sama yaitu aplikasi pencatatan penjualan telur sederhana. Setelah itu diukur tingkat *maintainability* dari kedua model tersebut menggunakan beberapa *maintainability metrics*. Model no-pattern digunakan sebagai pembanding karena ketika aplikasi pertama kali dibuat, Flutter secara default tidak menggunakan *architectural pattern* apapun. No-pattern menjadi *default* dari kode Flutter

agar para *developer* pemula dapat lebih mudah memahami struktur program pada Flutter yang mana masih baru dalam dunia pengembangan aplikasi *mobile*.

8.1 Kelas diagram penjualan telur sederhana pada MVI

Berikut merupakan kelas diagram pada aplikasi penjualan telur sederhana yang mengimplementasikan model arsitektur MVI:

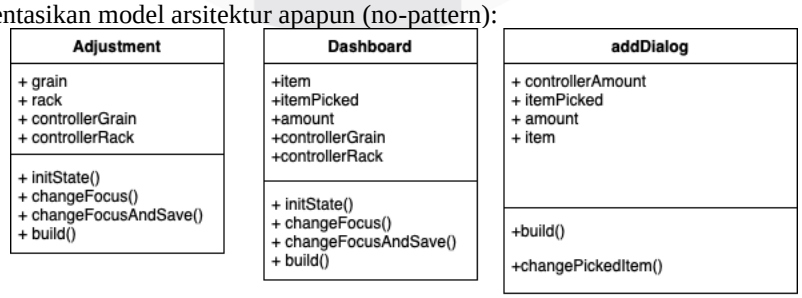


Gambar 13. Penerapan MVI pada aplikasi

Dari kelas diatas dapat dilihat terdapat tiga requirement yaitu menambahkan item (*addDialog*) melakukan penyesuaian harga (*adjustment*) dan menampilkan data penjualan (*dashboard*)

8.2 Kelas diagram penjualan telur sederhana pada no-pattern

Berikut merupakan kelas diagram pada aplikasi penjualan telur sederhana dengan tidak mengimplementasikan model arsitektur apapun (no-pattern):



Gambar 14. Penerapan no-pattern pada aplikasi

Dari kelas diagram diatas dapat dilihat kelas masing-masing kelas *independent* sehingga terlihat adanya *redudansi* atribut serta *method* yang seharusnya dapat digunakan kembali oleh kelas lain. Hal ini dapat menyebabkan nilai WMC meningkat yang mengakibatkan sulitnya menemukan *fault prone.*

8.3 Hasil Pengujian

Berikut merupakan hasil dari pengukuran *assessment maintainability metrics* antara MVI dan no-pattern yang diterapkan pada aplikasi yang sama:

Tabel 2. Hasil Maintanability Metrics

Metrics	**MVI**	**No-pattern**	**Keterangan**
WMC	9	10	pada MVI dihitung semua method, baik itu pada folder *framework* maupun pada folder *features.* Karena saling memiliki hubungan antar kelas, maka pencarian *fault prone* lebih mudah di-*trace.* Sedangkan Pada no-pattern jumlah method akan semakin bertambah seiring jumlah requirement yang menyebabkan sulitnya menemukan *fault prone* pada program.
RFC	6	1	Untuk pemanggilannya fungsi, MVI menggunakan abstract class sebagai jembatan dengan repository. Pada no pattern method langsung diakses didalam view, sehingga tidak perlu ada jembatan diantara fungsi pemanggil dengan fungsi yang dimaksud. Metode pemanggilan ini akan selalu digunakan dalam pemanggilan objek atau data
LCOM	1	2	Dalam kasus ini, karena Flutter mengadopsi fungsional programming dimana fungsional programing tidak terlalu mengandalkan global variable sehingga sulit mengidentifikasi LCOM dari kelas yang ada.
CBO	7	0	Pada MVI, karena terdapat dua screen saja maka jumlah CBO pada view dan behavior berbanding lurus terhadap jumlah screen yang ada. Pada no pattern, setiap kelas independent sehingga tidak ada ketergantungan terhadap kelas lain, walaupun ini terlihat bagus, namun pada proses lebih jauh, kode program akan sangat banyak dalam satu kelas, yang dapat menyebabkan sulitnya *debugging* program.
DIT	2	0	Pada MVI terdapat kelas child view, yang harus melewati view utama sehingga bernilai dua. Untuk no-pattern setiap kelasnya adalah kelas independen maka semua kelas adalah kelas biasa, tidak ada *parent* ataupun *child class.*

Terlihat no-pattern memiliki nilai yang baik, hal ini disebabkan karena aplikasi pembanding yang dibangun hanya memiliki 3 *requirement* yang mana angka tersebut menyebabkan mudahnya untuk me-*maintain* program, sehingga untuk jumlah *requirement* tersebut pada no-pattern aspek *maintainability*-nya akan terlihat bagus. Namun, untuk aplikasi yang memungkinkan akan dilakukan pengembangan lebih lanjut, MVI cukup baik jika dilihat dari nilai RFC diatas yang nilainya akan konstan berapapun jumlah *requirement* yang dibutuhkan. Lalu aspek dengan CBO diatas, MVI memiliki nilai yang berbanding lurus terhadap faktor jumlah *screen*, sehingga berapapun *screen* atau *requirement* yang akan didefinisikan nilainya akan tetap. Hal ini akan berdampak baik dalam aspek *maintainability* sebuah aplikasi. nilai konstan tersebut didapat dari penyederhanaan manajemen *state* yang hanya ada dua pada setiap screennya (onLoadScreen dan loadScreen)

9. Kesimpulan

Berdasarkan hasil evaluasi diatas, *assessment metrics* yang digunakan sudah cukup untuk mengidentifikasi kelebihan dan kekurangan pada MVI dan no-pattern. MVI menawarkan kemudahan dalam membangun aplikasi Flutter dengan mengandalkan pendekatan SoT (*single source of truth*) yang menempatkan *data/state* pada satu *source* serta penyederhanaan *state management*, sehingga memudahkan pengembang dalam melakukan *debuging*, *tracing data*, serta me-*maintainance* program (nilai CBO konstan karena system memiliki dua *state* dasar yang tidak berubah berapapun jumlah *screen*-nya). Untuk pendekatan no-pattern memang terlihat nilai yang baik pada *assessment* di atas. Namun dengan catatan pendekatan ini lebih cocok untuk aplikasi yang terbilang kecil dengan jumlah kurang lebih 3 *requirement* serta tidak ada rencana dalam melakukan pengembangan lebih lanjut. Sehingga, penggunaan no-pattern sangat tidak disarankan untuk digunakan jika aplikasi akan dikembangkan lebih jauh, karena akan berdampak buruk bagi aplikasi yang dibangun sehingga dapat menyebabkan sulitnya untuk me-*maintain* program aplikasi tersebut.

Dari hasil penelitian ini, aspek *maintainability* untuk aplikasi yang akan dikembangkan lebih jauh pada arsitektur model MVI sudah cukup baik. Selanjutnya, MVI menyederhanakan *state management* sehingga pengembang lebih mudah dalam mengatur *state* pada aplikasi. diharapkan Framework MVI yang dibangun dapat menjadi salah satu tools yang dapat berkontribusi dalam dunia pengembang aplikasi berbasis Flutter terutama dalam mendorong aspek maintainability program aplikasi.

REFERENSI

[1] R. Malhotra and A. Chug, "Software Maintanability: Systematic Literature Review and Current Trends," *International Journal of Software Engineering and Knowladge Engineering*, vol. XXVI, no. 8, pp. 1221-1246, 2015.

[2] L. Corral and I. Fronza, "Better Code for Better Apps: A Study on Source Code Quality and Market Success of Android Applications," *ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, no. 2, 2015.

[3] S. K. Dubey and A. Rana, "Assessment of Maintanability Metrics for Object-Oriented Software System," *ACM SIGSOFT Software Engineering*, vol. XXXV, no. 5, pp. 1-4, 2011.

[4] F. E. Shahbudin and F.-F. Chua, "Design Patterns for Developing High Efficiency Mobile Application," *Information Technology & Software Engineering*, vol. III, no. 3, pp. 1-9, 2013.

[5] A. Medeiros, "What If The User Was A Function," in *JSConf*, Budapest, 2015.

[6] M. Madhuram, K. Ashu and M. Andyamanian, "Cross Platform Development using Flutter," *International Journal of Engineering Science and Computing*, vol. IX, no. 4, pp. 21497-21500, 2019.

[7] Google Developers, "Sky: Am Experiment Writing Dart for Mobile," in *Dart Developer Summit*, California, 2015.

[8] G. R., "Smartphone Application Development using CrossPlatform Frameworks," in *Proceedings of the National Conference on Information and Communication Technology*, Mumbai, 2010.

[9] P. R. M. de Andrade and A. B. Albuquerque, "Cross Platform App: A Comparative Study," *Journal of Computer Science and Technology*, pp. 1-4, 2015.

[10] C. Pang and D. Szafron, "Single Source of Truth (SSOT) for Service Oriented Architecture (SOA)," *International Conference on Service-Oriented Computing*, vol. 8831, pp. 575-589, 2014.

[11] S. R. Chidamber, D. P. Darcy and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Transactions on Software Engineering*, vol. VIII, no. 24, pp. 629 - 639, 1998.