JOISIE (Journal Of Information Systems And Informatics Engineering)
Vol. 8, No.2, Desember 2024, Hlm 369-380
Received: 26 November 2024, Revised: 23 Desember 2024, Accepted: 31 Desember2024
p- ISSN: 2503-5304
e- ISSN: 2527-3116

IMPLEMENTASI *CLEAN ARCHITECTURE* PADA APLIKASI MOBILE AL-QURAN BERBASIS FLUTTER

M. Zihad Azziqra[1)], Ilyas Nuryasin[2)]

[1,2] Prodi Informatika, Universitas Muhammadiyah Malang, Jl. Raya Tlogomas No.246, Babatan, Tegalgondo, Kota Malang.

email: [1]azziqrazihad@webmail.umm.ac.id, [2]ilyas@umm.ac.id

Abstract

Flutter-based mobile application development is gaining popularity due to its cross-platform development efficiency. However, maintainability is a major challenge to overcome in application development. Software maintenance costs are estimated to account for up to 79%-90% of the total application development costs, so maintainability becomes an important factor in ensuring software quality. Based on these issues, this research focuses on the implementation of Clean Architecture in Flutter-based mobile applications, using the Al-Quran application as a case study, and evaluates its maintainability level using McCall's Software Quality Model. Clean Architecture was chosen for its approach of separating business logic, data access, and user interface into structured layers. The implementation results show a maintainability level of 79%, which is rated as "good". Clean Architecture successfully supports application modularity and flexibility through the application of SOLID principles, specifically the Single Responsibility Principle and the Dependency Inversion Principle. This research provides insight into the effectiveness of implementing Clean Architecture in the development of Flutter-based mobile applications.

Keywords: *Android, Clean Architecture, Flutter, Maintainability, McCall Software Quality Model.*

Abstrak

Pengembangan aplikasi mobile berbasis Flutter semakin populer karena efisiensi pengembangan *multi-platform* yang ditawarkannya. Namun, *maintainability* menjadi tantangan utama yang harus diatasi dalam pengembangan aplikasi. Biaya pemeliharaan perangkat lunak diperkirakan menyumbang hingga 79%-90% dari total biaya pengembangan aplikasi, sehingga *maintainability* menjadi faktor penting dalam memastikan kualitas perangkat lunak. Berdasarkan permasalahan tersebut, penelitian ini berfokus pada implementasi *Clean Architecture* pada aplikasi mobile berbasis Flutter, dengan aplikasi Al-Quran sebagai studi kasus, serta mengevaluasi tingkat *maintainability*-nya menggunakan *McCall's Software Quality Model*. *Clean Architecture* dipilih karena pendekatannya yang memisahkan logika bisnis, akses data, dan antarmuka pengguna ke dalam lapisan-lapisan yang terstruktur. Hasil implementasi menunjukkan tingkat *maintainability* sebesar 79%, yang dikategorikan "baik". *Clean Architecture* berhasil mendukung modularitas dan fleksibilitas aplikasi melalui penerapan prinsip SOLID, khususnya *Single Responsibility Principle* dan *Dependency Inversion Principle*. Penelitian ini memberikan wawasan tentang efektivitas penerapan *Clean Architecture* dalam pengembangan aplikasi mobile berbasis Flutter.

Kata kunci*:* *Android, Clean Architecture, Flutter, Maintainability, McCall Software Quality Model.*

1. PENDAHULUAN

Perkembangan teknologi yang pesat telah membawa perubahan signifikan dalam pengembangan perangkat lunak, termasuk aplikasi mobile yang kini menjadi salah satu alat utama dalam berbagai sektor kehidupan. Aplikasi mobile, yang dirancang untuk perangkat seperti smartphone dan tablet, kini menjadi solusi untuk meningkatkan efisiensi dan produktivitas pengguna (Ikhwan dkk., 2023). Di Indonesia, penggunaan aplikasi berbasis Android mencapai 92,03% pada Februari 2021, dengan lebih dari 2,9 juta aplikasi terdaftar di Google Play Store (Badrudduja & Putra, 2022), yang

https://doi.org/10.35145/joisie.v8i2.4763

menunjukkan adanya persaingan sengit antar pengembang untuk menciptakan aplikasi yang lebih baik dan lebih efisien.

Salah satu platform pengembangan aplikasi yang sedang populer adalah Flutter, yang memungkinkan pengembang untuk membuat aplikasi *multi-platform* (Android, iOS, dan Web) dengan satu kode sumber (Bhagat, 2022; Muslim dkk., 2022). Flutter menawarkan kemudahan dalam pengembangan aplikasi, mengurangi kebutuhan pengkodean ganda untuk berbagai platform, dan memungkinkan proses pengembangan yang lebih cepat dan efisien (Widiarta dkk., 2021).

Namun, meskipun pengembangan aplikasi semakin cepat, tantangan utama yang dihadapi adalah *maintainability*. Biaya pemeliharaan perangkat lunak diperkirakan menyumbang hingga 79%-90% dari total biaya pengembangan (Sondha dkk., 2020), yang menunjukkan pentingnya merancang aplikasi dengan arsitektur yang memungkinkan pemeliharaan yang mudah. Semakin tinggi *maintainability* dari suatu aplikasi, maka semakin mudah pula proses pemeliharaannya sehingga biaya, usaha dan waktu yang dibutuhkan akan semakin berkurang (Deta Aditya dkk., 2022; Laksono dkk., 2024). *Clean Architecture* adalah salah satu pendekatan yang dapat meningkatkan *maintainability*, karena menyediakan struktur yang jelas untuk aplikasi, memungkinkan perubahan dan pembaruan yang lebih mudah tanpa memengaruhi bagian lain dari sistem (Subagio & Muttaqin, 2022).

Dalam konteks ini, penerapan *Clean Architecture* pada aplikasi berbasis Flutter dapat memberikan solusi yang efektif untuk masalah pemeliharaan aplikasi. *Clean Architecture* pertama kali diperkenalkan oleh Robert C. Martin (Anhar dkk., 2024). Berfokus pada pemisahan kode aplikasi menjadi beberapa lapisan, yang masing-masing memiliki tanggung jawab yang jelas (Wijayanto dkk., 2023). Pendekatan ini bertujuan untuk membuat aplikasi lebih mudah dikelola, diuji, dan diperbarui tanpa mempengaruhi bagian lain dari aplikasi (Yadati, 2023). Struktur ini memisahkan logika bisnis, antarmuka pengguna, dan lapisan data, sehingga mempermudah pengelolaan kode dan meminimalkan ketergantungan antar bagian aplikasi.

Selain itu, untuk mencapai tingkat *maintainability* yang tinggi, *Clean Architecture* mengadopsi prinsip SOLID, yaitu lima prinsip desain perangkat lunak yang bertujuan untuk menciptakan sistem yang mudah dipahami, adaptif terhadap perubahan, dan dapat digunakan kembali (Cabral dkk., 2024). Kelima prinsip dalam SOLID adalah *Single Responsibility Principle* (SRP), *Open/Closed Principle* (OCP), *Liskov Substitution Principle* (LSP), *Interface Segregation Principle* (ISP), dan *Dependency Inversion Principle* (DIP) (Sanchez dkk., 2022). Penerapan prinsip SOLID dapat meningkatkan modularitas, fleksibilitas, dan keterbacaan kode, yang semuanya berkontribusi pada kemudahan pemeliharaan aplikasi (Yanakiev dkk., 2024).

Penelitian yang dilakukan oleh Sinatria dkk. (2023), dengan tujuan untuk memberikan gambaran dan panduan kepada pengembang tentang bagaimana mengorganisir aplikasi berbasis Flutter dengan pendekatan *Clean Architecture*. Fokus utama dari penelitian ini adalah pada pembuatan modul-modul modular dalam arsitektur aplikasi yang dapat digunakan kembali, serta implementasi prinsip SOLID untuk meningkatkan fleksibilitas dan kemudahan pengembangan aplikasi.

Penelitian yang dilakukan oleh Abdillah dkk. (2024), mengimplementasikan *Clean Architecture* pada aplikasi smart parking dengan fokus pada meningkatkan skalabilitas dan kemudahan pemeliharaan. Penelitian ini menekankan pada struktur arsitektur yang modular dan terorganisir untuk menghadapi perubahan kebutuhan sistem.

Berdasarkan beberapa penelitian terdahulu yang telah dilakukan, belum ada penelitian yang secara spesifik mengevaluasi tingkat *maintainability* dari aplikasi berbasis Flutter yang menerapkan *Clean Architecture* menggunakan metrik kualitas perangkat lunak yang terukur. Oleh karena itu, penelitian ini bertujuan untuk mengimplementasikan *Clean Architecture* pada aplikasi mobile berbasis Flutter sekaligus mengevaluasi tingkat *maintainability*-nya menggunakan *McCall's Software Quality Model*. Penelitian ini akan menggunakan aplikasi Al-Quran sebagai objek implementasi. Aplikasi ini memiliki fitur-fitur utama seperti pembacaan ayat-ayat Al-Quran, terjemahan, tafsir, dan bookmark untuk menandai ayat-ayat yang telah dibaca. Implementasi *Clean Architecture* pada aplikasi ini

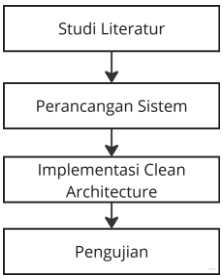
https://doi.org/10.35145/joisie.v8i2.4763

diharapkan dapat memberikan wawasan yang lebih terukur mengenai penerapan dan efektivitas pendekatan ini dalam konteks aplikasi mobile berbasis Flutter.

2. METODE PENELITIAN

2.1 KERANGKA PENELITIAN

Terdapat beberapa tahapan yang telah diurutkan secara sistematis agar sesuai dengan pokok pembahasan untuk mendapatkan hasil yang optimal dalam penelitian. Tahapan penelitian dapat dilihat pada gambar 1.

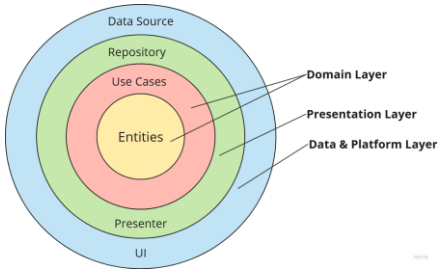


Gambar 1. Kerangka Penelitian

1. Studi Literatur
   Studi literatur dilakukan untuk mengumpulkan dan menganalisis literatur yang relevan terkait *Clean Architecture*, prinsip SOLID, dan praktik terbaik dalam pengembangan aplikasi mobile. Tahap ini dapat membantu penulis memperoleh pemahaman teoritis mengenai penelitian serta membantu memahami konsep-konsep yang akan diterapkan (Rusydi & Nuryasin, 2024).
2. Perancangan Sistem
   Pada tahap ini, dilakukan perancangan sistem dengan pendekatan *Clean Architecture*. Proses ini melibatkan pembuatan struktur arsitektur berdasarkan prinsip SOLID. Perancangan ini bertujuan untuk memberikan landasan yang jelas sebelum implementasi dilakukan.
3. Implementasi *Clean Architecture*
   Tahap ini merupakan proses inti dari penelitian, di mana *Clean Architecture* diterapkan pada pengembangan aplikasi mobile berbasis Flutter.
4. Pengujian
   Pada tahap ini dilakukan pengujian untuk mengukur seberapa tingkat *maintainability* dari aplikasi. Pengujian *maintainability* dilakukan dengan menggunakan metode *McCall's Software Quality Model*.

2.2 CLEAN ARCHITECTURE

Tujuan dari pola desain *Clean Architecture* adalah menyediakan standar yang memungkinkan pemisahan logika bisnis dari lapisan antarmuka pengguna. Pemisahan ini bertujuan untuk memastikan bahwa perubahan pada satu lapisan tidak berdampak langsung pada lapisan lain, sehingga meningkatkan fleksibilitas dan efisiensi pemeliharaan (Yadati, 2023). Gambaran dari *Clean Architecture* dapat dilihat pada gambar 2.



Gambar 2. Diagram *Clean Architecture*

https://doi.org/10.35145/joisie.v8i2.4763

Clean Architecture, yang terdiri dari domain *layer, data layer,* dan *presentation layer*. Model arsitektur berlapis seperti ini memungkinkan aplikasi mobile yang dikembangkan bersifat *framework-agnostic*, yaitu logika bisnisnya tidak terikat pada *framework* eksternal tertentu, sehingga mempermudah proses pemeliharaan dan pengembangan di masa depan (Wijayanto dkk., 2023).

Penerapan *Clean Architecture* juga mengikuti prinsip-prinsip SOLID. SOLID meliputi lima prinsip utama, yaitu:

a. *Single Responsibility Principle* (SRP) memastikan setiap kelas atau modul memiliki satu tanggung jawab utama sehingga setiap komponen akan memiliki peran yang jelas dan spesifik (Sinatria dkk., 2023).
b. *Open Closed Principle* (OCP) yaitu kelas atau modul sebaiknya terbuka untuk ekstensi namun tertutup untuk modifikasi, sehingga kelas bisa diperluas tanpa harus memodifikasi kode yang ada (Cabral dkk., 2024).
c. *Liskov Substitution Principle* (LSP) menyatakan bahwa objek dari kelas turunan harus dapat menggantikan objek dari kelas induknya tanpa menyebabkan ketidaksesuaian atau perubahan pada keabsahan program (Ramachandrappa, 2024).
d. *Interface Segregation Principle* (ISP) menekankan bahwa jika sebuah fungsionalitas akan dimanfaatkan oleh berbagai kelas, maka perlu dibuat interface yang spesifik untuk setiap kelas. Tujuan dari prinsip ini adalah mencegah terjadinya kompilasi ulang yang tidak diperlukan dan mengurangi ketergantungan antar komponen (Sanchez dkk., 2022).
e. *Dependency Inversion Principle* (DIP) mengurangi ketergantungan pada kelas konkret dengan berfokus pada abstraksi (Sinatria dkk., 2023).

Penerapkan prinsip-prinsip tersebut dapat meningkatkan keterbacaan, mengurangi kompleksitas kode, mudah untuk di uji, kode dapat digunakan kembali, dan mengurangi ketergantungan yang tinggi antar kode serta memungkinkan pengembangan dan pemeliharaan aplikasi yang lebih efektif dan efisien (Oktafiani & Saputra, 2022).

2.3 MCCALL'S SOFTWARE QUALITY MODEL

McCall's Software Quality Model menyediakan pendekatan yang terstruktur untuk mengevaluasi dan meningkatkan kualitas perangkat lunak. Model ini memiliki tiga aspek utama: *Product Operation*, yang mencakup sifat-sifat operasional perangkat lunak; *Product Revision*, yang menyoroti kemampuan perangkat lunak untuk menghadapi perubahan; dan *Product Transition*, yang menggambarkan kemampuan perangkat lunak untuk beradaptasi dengan lingkungan baru (Salamudin dkk., 2024).

Aspek *maintainability* sendiri termasuk kedalam *Product Revision* (Mulatsari dkk., 2023). *Quality metric* pada *McCall's Software Quality Model* dapat dihitung menggunakan rumus (1).

$$F = \frac{C_1.M_1 \ + \ C_2.M_2 \ + \cdots + \ C_n.M_n}{C_1 \ + \ C_2 \ + \cdots + \ C_n \ = \ 1} \qquad (1)$$

$$0 \ \leq M \leq 1 \mid 0 \leq C \leq 1$$

Quality Metric (F) dihitung dengan menjumlahkan hasil perkalian setiap *quality factor* (M) dengan bobotnya (C). Total bobot pada *quality metric* berkisar antara 0 hingga 1, dengan jumlah keseluruhan bobot adalah 1. Setiap *quality metric* memiliki *quality factor* tertentu yang memengaruhi nilainya. Nilai masing-masing *quality factor* diperoleh dengan membandingkan jumlah *positive case* terhadap *test case* yang relevan. Dalam *McCall's Software Quality Model*, kualitas *maintainability* ditentukan oleh beberapa faktor, yaitu *simplicity, conciseness, consistency, instrumentation, modularity,* dan *self-documentation*. (Muthmainnah & Putro, 2023).

Sondha dkk. (2020) menjelaskan bahwa *conciseness* mengukur rasio antara jumlah total kelas dan *Logical Lines of Code* (LLOC) dalam sebuah aplikasi. *Modularity* adalah *quality factor* yang menilai tingkat ketergantungan antar komponen untuk menentukan apakah suatu kelas memiliki tingkat *coupling* yang tinggi atau rendah. *Self-documentation* mengacu pada sejauh mana kode sumber menyediakan dokumentasi yang jelas, sedangkan *simplicity* merujuk pada kemudahan dalam memahami program. Kedua faktor ini, *self-documentation* dan *simplicity*, diukur berdasarkan nilai

https://doi.org/10.35145/joisie.v8i2.4763

clean code, yang mencakup aspek seperti konvensi penamaan, aturan kapitalisasi, dan komentar. Dalam penelitian ini, setiap *quality factor* untuk menghitung *maintainability* diberi bobot yang sama, yaitu 0,167.

Total *quality metric* digunakan untuk mengevaluasi kelayakan aspek yang dianalisis, dengan kategori yang dibagi menjadi lima berdasarkan rentang persentase. Pembagian kategori kelayakan sesuai rentang persentase dapat dilihat pada tabel 1.

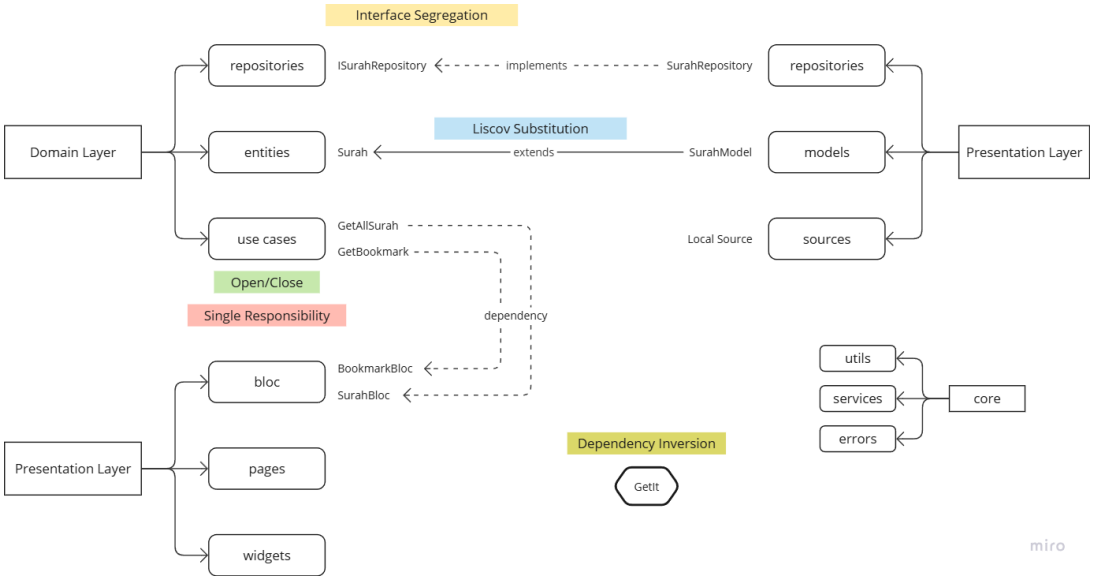
Tabel 1. Kategori kelayakan (Hanes dkk., 2020)

No.	Kategori	Presentase
1	Sangat Baik	81% - 100%
2	Baik	61% - 80%
3	Cukup Baik	41% - 80%
4	Tidak Baik	21% - 40%
5	Sangat Tidak Baik	< 21%

3. HASIL DAN PEMBAHASAN

3.1 PERANCANGAN SISTEM

Tahapan perancangan sistem ini bertujuan untuk memisahkan logika bisnis dari tampilan aplikasi dengan menerapkan *Clean Architecture*. Desain sistem mencakup tiga layer utama yaitu *domain layer, data layer*, dan *presentation layer*. Struktur perancangan divisualisasikan pada gambar 3.



Gambar 3. Diagram rancangan arsitektur sistem berdasarkan *Clean Architecture* dan prinsip SOLID

Diagram rancangan sistem pada gambar 3 menunjukkan penerapan *Clean Architecture*. Tujuan utama dari desain tersebut adalah untuk mencapai pemisahan tanggung jawab yang jelas (Wijayanto dkk., 2023).

Domain layer mencakup logika inti aplikasi yang meliputi *entities*, *use cases*, dan *interface repository*. *Entities* merepresentasikan objek utama dalam *domain*, sedangkan *use cases* bertanggung jawab untuk mengatur alur bisnis aplikasi. Pada lapisan ini, prinsip *Single Responsibility Principle* (SRP) diterapkan untuk memastikan bahwa setiap kelas hanya memiliki satu tanggung jawab utama. Penerapan *Open/Close Principle* (OCP) memungkinkan penambahan fungsionalitas baru tanpa harus mengubah kode yang telah ada. Sementara itu, penerapan *Dependency Inversion Principle* (DIP) memastikan bahwa lapisan ini bergantung pada abstraksi, bukan implementasi. GetIt digunakan

https://doi.org/10.35145/joisie.v8i2.4763

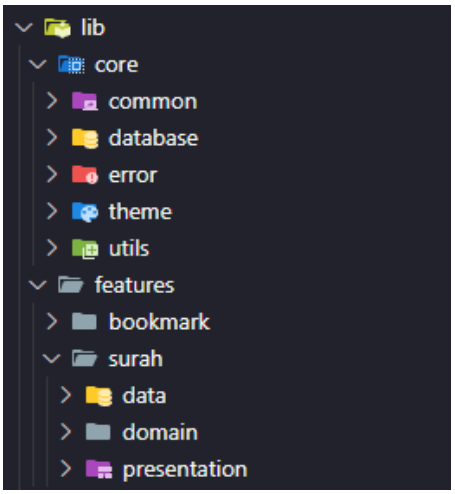
sebagai *dependency injection* untuk menyuntikkan dependensi antar lapisan, memungkinkan objek seperti *use cases* atau *repository* diakses tanpa perlu mengetahui detail implementasinya.

Data layer berfungsi untuk mengelola akses data dari sumber lokal maupun eksternal. Lapisan ini terdiri dari *repositories*, *models*, dan *sources*. *Repositories* berperan sebagai penghubung antara *domain layer* dan *data source*, sementara *models* bertugas memetakan data ke dalam format yang sesuai dengan kebutuhan *domain*. Penerapan *Interface Segregation Principle* (ISP) memastikan bahwa interface dipecah menjadi bagian yang lebih spesifik untuk menghindari ketergantungan yang tidak perlu. Selain itu, *Liskov Substitution Principle* (LSP) diterapkan untuk memastikan bahwa setiap *subclass* dapat menggantikan *superclass* tanpa memengaruhi fungsi aplikasi.

Presentation layer menangani penyajian data kepada pengguna dengan menggunakan bloc sebagai *state management* utama. Bloc berfungsi sebagai mediator antara logika bisnis dan antarmuka pengguna, memastikan bahwa perubahan data ditampilkan secara reaktif. *Core layer* menyediakan fungsi pendukung seperti *utils*, *services*, dan *error handling* yang digunakan bersama oleh berbagai lapisan.

3.2 IMPLEMENTASI

Pada tahap implementasi, penelitian ini menerapkan *Clean Architecture* dalam pengembangan aplikasi Al-Quran berbasis Flutter. Struktur *Clean Architecture* yang diterapkan terdiri dari dua komponen, yaitu *core* dan *features*, seperti yang ditampilkan pada gambar 4.



Gambar 4. Struktur folder *Clean Architecture*

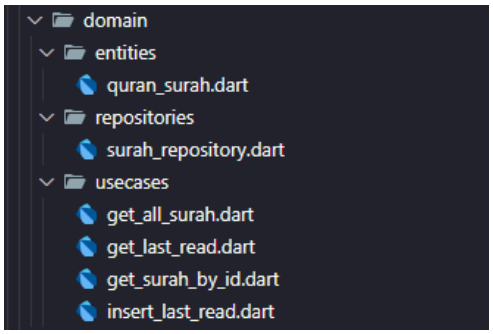
Lapisan *core* bertindak sebagai penghubung antara inti aplikasi, yang berisi logika bisnis dan entitas, dengan lingkungan eksternal. Tujuan utama dari lapisan *core* adalah untuk memastikan modularitas dan fleksibilitas sistem, sehingga pembaruan atau penggantian dependensi eksternal dapat dilakukan tanpa memengaruhi inti aplikasi.

Sementara itu, lapisan *features* bertanggung jawab mengelola dan memisahkan fitur-fitur aplikasi menjadi beberapa bagian yang lebih kecil sesuai fungsi masing-masing. Pemisahan ini bertujuan untuk membagi basis kode menjadi komponen-komponen yang terorganisir, modular, dan sesuai dengan kebutuhan setiap fitur aplikasi. Setiap fitur dalam lapisan ini terdiri dari tiga lapisan utama, yaitu *domain layer*, *data layer*, dan *presentation layer*, yang dirancang berdasarkan prinsip *Clean Architecture*.

3.2.1 Domain Layer

Untuk memastikan aplikasi yang dapat dipelihara dan dikembangkan secara skalabel, keberadaan *domain layer* menjadi sangat penting. Dengan memusatkan logika bisnis utama pada lapisan ini, pengembang memiliki fleksibilitas untuk menyesuaikan dan meningkatkan fungsi aplikasi tanpa mempengaruhi komponen sistem lainnya.

https://doi.org/10.35145/joisie.v8i2.4763



Gambar 5. Struktur *domain layer*

Berdasarkan gambar 5, *domain layer* terdiri dari *usecases*, *repositories* dan *entities*. *Entities* merepresentasikan objek bisnis inti seperti "*quran_surah.dart"*, yang bertugas memodelkan dan mengelola data utama yang terkait dengan aplikasi. Sementara itu, *repositories* seperti "*surah_repository.dart"* bertindak sebagai penghubung antara *domain layer* dan *data layer*, menyediakan metode-metode untuk mengakses dan memanipulasi data yang diperlukan. Komponen *usecases* merepresentasikan operasi atau tindakan bisnis spesifik, seperti "*get_all_surah.dart"*, *"get_last_read.dart"*,*"get_surah_by_id.dart"*dan "*insert_last_read"*, yang dirancang untuk menjalankan alur kerja bisnis tertentu secara efisien.

Pada layer ini dapat diterapkan dua prinsip SOLID yaitu *single responsibility principle* (SRP) dan *dependency inversion principle* (DIP). Contoh implementasi dari prinsip-prinsip tersebut dapat dilihat pada gambar 6.

```
abstract class SurahRepository {
  Future<Either<Failure, List<QuranSurah>>> getAllSurah();
  Future<Either<Failure, List<QuranSurah>>> getSurahById(int id);
  Future<Either<Failure, void>> removeBookmark(int id);
  Future<Either<Failure, List<Bookmark>>> getBookmarks();
  Future<Either<Failure, void>> insertLastRead(LastRead lastRead);
  Future<Either<Failure, LastRead>> getLastRead();
  Future<Either<Failure, void>> addBookmark(
    Ayah ayat,
    int nomorSurah,
    int numberInJuz,
    String namaSurah,
    String via,
  );
}
```

Gambar 6. *Class SurahRepository*

Pada gambar 6, terdapat kelas *"SurahRepository"* yang merupakan sebuah kelas abstrak yang mendefinisikan berbagai metode untuk interaksi dengan data. Kelas ini berfungsi sebagai antarmuka yang mendefinisikan kontrak umum yang terkait pada fitur Surah tanpa memberikan detail implementasinya. Kemudian, kelas tersebut akan diimplementasikan oleh kelas yang ada pada *usecase*.

```
class GetAllSurah implements Usecase<List<QuranSurah>, NoParams> {
  final SurahRepository surahRepository;

  GetAllSurah(this.surahRepository);

  @override
  Future<Either<Failure, List<QuranSurah>>> call(NoParams params) async {
    return await surahRepository.getAllSurah();
  }
}
```

Gambar 7. *Class GetAllSurah*

Salah satunya adalah kelas "*GetAllSurah"* yang bergantung pada abstraksi "*SurahRepository"* untuk menampilkan data surah, bukan pada implementasi konkret dari *"SurahRepository"*. Ini merupakan contoh penerapan *Dependency Inversion Principle* (DIP). Untuk mendukung penerapannya digunakan library GetIt untuk *dependency injection*. Dalam hal ini, *"SurahRepositoryImpl"* didaftarkan pada GetIt sebagai implementasi dari *"SurahRepository"*, sehingga *"GetAllSurah"* dapat memperoleh abstraksinya secara otomatis. Contoh kode pendaftaran dapat dilihat pada gambar 8.

https://doi.org/10.35145/joisie.v8i2.4763

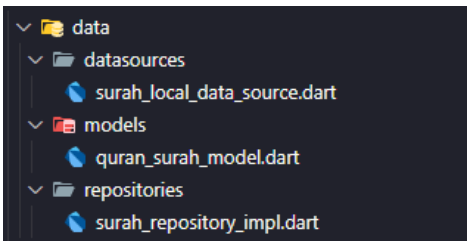
```
void _initSurah() {
  // DataSource
  serviceLocator
    ..registerFactory<SurahLocalDataSource>(
      () => SurahLocalDataSourceImpl(),
    )
    // Repository
    ..registerFactory<SurahRepository>(
      () => SurahRepositoryImpl(
        serviceLocator(),
        serviceLocator(),
      ),
    )
    // kode lainnya
}
```

Gambar 8. Pendaftaran *dependency* menggunakan GetIt

Selain itu, terdapat implementasi dari *single responsibility principle* (SRP) yang terdapat pada *usecases*, dapat dilihat pada pada gambar 5. Masing-masing *usecase* memiliki tanggung jawab tunggal untuk menyelesaikan satu alur kerja bisnis yang spesifik.

3.2.2 Data Layer

Data layer memiliki peran utama dalam pengelolaan sumber data, *repository*, dan model dalam arsitektur aplikasi. Lapisan ini berperan sebagai penghubung antara domain aplikasi dengan berbagai sumber data eksternal, serta memfasilitasi akses dan manipulasi data.



Gambar 9. Struktur *data layer*

Berdasarkan Gambar 9, struktur data layer terdiri atas tiga bagian utama: *datasources*, *models*, dan *repositories*. *Datasources*, seperti file "*surah_local_data_source.dart"*, bertugas mengambil dan menyimpan data dari sumber lokal dalam format aslinya. Models, seperti file "*quran_surah_model.dart"*, merepresentasikan data dari datasource dan mengonversinya ke format yang siap digunakan di *domain layer*. *Repositories*, seperti file "*surah_repository_impl.dart*", menghubungkan *data layer* dan *domain layer* dengan mengambil data dari *datasource*, mengubahnya ke model yang sesuai, dan menyediakannya untuk *usecase*.

Pada data layer, tiga prinsip SOLID dapat diterapkan, yaitu *Single Responsibility Principle* (SRP), *Open/Closed Principle* (OCP), dan *Liskov Substitution Principle* (LSP). Contoh penerapannya dalam sebuah aplikasi dapat dilihat pada gambar 10, 11, dan 12.

```
abstract class SurahLocalDataSource {
  Future<List<QuranSurahModel>> getAllSurah();
  Future<List<QuranSurahModel>> getSurahById(int id);
}
```

Gambar 10. Implementasi OCP pada *data layer*

Pada gambar 10, terdapat kelas "*SurahLocalDataSource"* yang merupakan sebuah kelas abstrak yang mendefinisikan metode untuk mendapatkan data Surah dari sumber lokal. Kelas ini mengikuti *open/closed principle* (OCP). Dengan menggunakan kelas abstrak ini, fungsionalitas kelas dapat diperluas dengan membuat kelas turunan yang mengimplementasikan metode yang didefinisikan tanpa harus mengubah kelas "*SurahLocalDataSource"* itu sendiri.

https://doi.org/10.35145/joisie.v8i2.4763

```dart
class SurahLocalDataSourceImpl extends SurahLocalDataSource {
  @override
  Future<List<QuranSurahModel>> getAllSurah() async {
    try {
      String jsonString = await rootBundle.loadString('surah.json');
      List<dynamic> jsonList = json.decode(jsonString);
      List<QuranSurahModel> surahList = QuranSurahModel.fromJsonList(jsonList);
      return surahList;
    } catch (e) {
      throw LocalException(e.toString());
    }
  }
  // kode lainnya
}
```

Gambar 11. Implementasi SRP pada *data layer*

Pada gambar 11, "*SurahLocalDataSourceImpl"* mengimplementasi "*SurahLocalDataSource"*. Kelas ini bertanggung jawab untuk membaca data Surah dari sumber lokal. Kelas ini menerapkan *Single Responsibility Principle* (SRP) karena hanya memiliki satu tanggung jawab utama yaitu pengelolaan data Surah dari sumber lokal. Dengan memisahkan tanggung jawab ini ke dalam kelas tersendiri, kelas tersebut fokus pada satu tugas spesifik. Pemisahan tanggung jawab ini meminimalkan kompleksitas dan membuat kode lebih terstruktur.

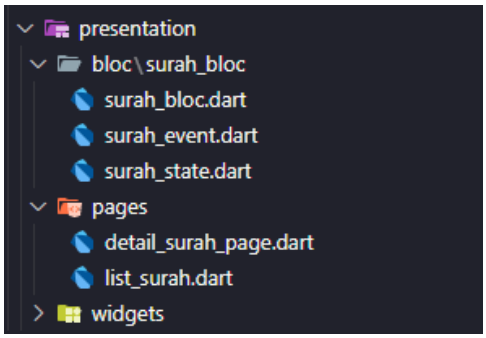
```dart
class SurahModel extends Surah {
  const SurahModel({
    required super.nomor,
    required super.nama,
    required super.namaLatin,
    required super.jumlahAyat,
  });

  factory SurahModel.fromJson(Map<String, dynamic> json) {
    return SurahModel(
      nomor: json["nomor"],
      nama: json["nama"],
      namaLatin: json["namaLatin"],
      jumlahAyat: json["jumlahAyat"],
    );
  }

  @override
  Map<String, dynamic> toJson() => {
        "nomor": nomor,
        "nama": nama,
        "namaLatin": namaLatin,
        "jumlahAyat": jumlahAyat,
        };
}
```

Gambar 12. Implementasi LSP pada *data layer*

Pada gambar 12, terdapat kelas *"SurahModel"* yang merupakan turunan dari kelas Surah. Kelas ini menerapkan *Liskov Substitution Principle* (LSP). Kelas SurahModel memperluas kelas Surah dengan menambahkan metode tambahan seperti *"fromJson"* dan *"toJson"* untuk konversi data. Dengan demikian, objek *"SurahModel"* dapat digunakan di mana saja objek Surah digunakan, tanpa memerlukan perubahan pada kode yang menggunakan objek Surah.

3.2.3 Presentation Layer

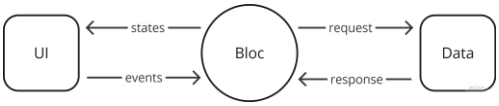
Presentation layer adalah lapisan dalam arsitektur aplikasi yang berfungsi sebagai antarmuka utama antara pengguna dan sistem menggunakan *Bloc pattern*.



Gambar 13. Struktur *presentation layer*

Berdasarkan gambar 13, *presentation layer* terdiri dari bloc, *pages*, dan *widgets*. Bloc bertanggung jawab atas pengelolaan *state* aplikasi dan logika bisnis yang terkait dengan tampilan (Munawar dkk., 2021). Arsitektur dari *Bloc pattern* dapat dilihat pada gambar 14.

https://doi.org/10.35145/joisie.v8i2.4763



Gambar 14. Bloc *pattern* (Zulistiyan dkk., 2024)

Pada bagian bloc terdiri dari tiga komponen utama, yaitu *"surah_bloc.dart"* yang menjadi inti implementasi dari Bloc untuk fitur surah, *"surah_event.dart"* yang mendefinisikan berbagai event yang dapat memicu perubahan state, dan *"surah_state.dart"* yang berisi berbagai state yang dapat terjadi selama pengelolaan fitur surah.

Pages merepresentasikan halaman-halaman dalam aplikasi. Sedangkan *widgets* adalah elemen-elemen UI dasar yang digunakan untuk membangun tampilan. Setiap *widget* memiliki tanggung jawab tunggal untuk menampilkan satu elemen visual atau interaksi tertentu.

3.3 PENGUJIAN MAINTAINABILITY

Pengujian dilakukan dengan menghitung nilai dari *quality factor*, yaitu *simplicity, conciseness, consistency, instrumentation, modularity* dan *self-documentation*. Hasil dari pengujian ini memberikan gambaran tentang tingkat *maintainability* pada aplikasi. Hasil pengujian dapat dilihat pada tabel 2.

Tabel 2. Hasil Pengujian Tingkat *Maintainability*

Quality Factor	***Positive Case* dan *Test Case***	**Nilai**	**M**	**C**	**F**
Conciseness	Banyaknya *Class*	129	0,11	0,167	0,02
	LLOC	1185			
Consistency	Fitur sesuai desain	4	1	0,167	0,17
	Fitur yang di desain	4			
Instrumentation	Fitur punya instrumentasi	4	1	0,167	0,17
	Fitur yang seharusnya punya instrumentasi	4			
Modularity	*Class* yang *loose coupling*	118	0.91	0,167	0,15
	Banyaknya *Class*	129			
Self-Documentation	*Class* yang *clean code*	111	0.86	0,167	0,14
	Banyaknya *Class*	129			
Simplicity	Fungsi yang *clean code*	150	0.82	0,167	0,14
	Banyaknya fungsi	182			
Total					**0,79**

Hasil pengujian pada tabel 2 menunjukkan bahwa tingkat *maintainability* aplikasi mencapai 0,79 atau 79%, yang berada dalam kategori "baik" menurut skala penilaian pada tabel 1. Faktor *modularity* memperoleh nilai tertinggi sebesar 0,15 dari total 0,167, menunjukkan bahwa arsitektur aplikasi berhasil meminimalkan ketergantungan antar komponen. Modularitas yang baik mencerminkan keberhasilan penerapan prinsip desain seperti *Single Responsibility Principle* (SRP) dan *Dependency Inversion Principle* (DIP), yang memisahkan tanggung jawab setiap komponen dengan jelas.

Sebaliknya, quality factor *conciseness* memiliki nilai terendah, yaitu 0,02, akibat tingginya jumlah *Logical Lines of Code* (LLOC) pada beberapa kelas, terutama di *presentation layer*. Hal ini disebabkan oleh sifat *presentation layer* yang cenderung menampung banyak elemen visual dan logika interaksi pengguna. Kondisi ini dapat diatasi dengan melakukan refaktorisasi kode melaui teknik seperti *extract widget* atau *extract method*, sehingga elemen-elemen UI yang berulang atau kompleks dipisahkan menjadi widget atau metode tersendiri.

4. SIMPULAN

Berdasarkan hasil pengujian *maintainability*, implementasi *Clean Architecture* pada aplikasi Al-Quran berbasis Flutter dapat dinyatakan berhasil dengan tingkat *maintainability* mencapai 79%, yang berada dalam kategori "baik". Aplikasi yang dikembangkan berjalan dengan baik, memenuhi semua fungsionalitas utama, seperti pembacaan ayat-ayat Al-Quran, terjemahan, tafsir, dan bookmark untuk menandai ayat yang telah dibaca. Faktor *modularity* memberikan kontribusi tertinggi terhadap

https://doi.org/10.35145/joisie.v8i2.4763

maintainability, menegaskan bahwa prinsip *Single Responsibility* dan *Dependency Inversion* telah diterapkan dengan baik untuk memisahkan tanggung jawab setiap komponen.

Namun, faktor *conciseness* menunjukkan nilai yang lebih rendah akibat tingginya jumlah *Logical Lines of Code* (LLOC) pada beberapa komponen, terutama di *presentation layer*. Hal ini menunjukkan bahwa meskipun aplikasi berjalan dengan baik, diperlukan upaya refaktorisasi kode untuk mengurangi kompleksitas, meningkatkan keterbacaan, dan menjaga efisiensi pengembangan di masa mendatang.

Dengan hasil yang diperoleh, implementasi ini memberikan dasar yang kokoh untuk pengembangan aplikasi yang lebih baik di masa depan, meskipun perbaikan lanjutan pada aspek pengelolaan kode dan pengujian tambahan tetap diperlukan untuk menjaga kualitas perangkat lunak.

5. DAFTAR PUSTAKA

Abdillah, F., Sirojudin, A., Amin, M. Y., & Atmadji, E. S. J. (2024). Analisis Skalabilitas Aplikasi Smart Parking Melalui Penerapan Clean Architecture. *METHOMIKA Jurnal Manajemen Informatika dan Komputerisasi Akuntansi*, *8*(1), 96–104. https://doi.org/10.46880/jmika.Vol8No1.pp96-104

Anhar, F. F., Swari, M. H. P., & Aditiawan, F. P. (2024). Analisis Perbandingan Implementasi Clean Architecture Menggunakan MVP, MVI, Dan MVVM Pada Pengembangan Aplikasi Android Native. *Jupiter: Publikasi Ilmu Keteknikan Industri, Teknik Elektro dan Informatika*, *2*(2), 181–191. https://doi.org/10.61132/jupiter.v2i2.155

Badrudduja, Moh. H., & Putra, R. E. (2022). Penerapan Clean Architecture pada Aplikasi Pemesanan Makanan menggunakan Metode Slope One Algorithm. *Journal of Informatics and Computer Science (JINACS)*, *3*(04), 506–514. https://doi.org/10.26740/jinacs.v3n04.p506-514

Bhagat, S. A. (2022). Review on Mobile Application Development Based on Flutter Platform. *International Journal for Research in Applied Science and Engineering Technology*, *10*(1), 803–809. https://doi.org/10.22214/ijraset.2022.39920

Cabral, R., Kalinowski, M., Baldassarre, M. T., Villamizar, H., Escovedo, T., & Lopes, H. (2024). Investigating the Impact of SOLID Design Principles on Machine Learning Code Understanding. *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, 7–17. https://doi.org/10.1145/3644815.3644957

Deta Aditya, M., Susanty, M., & Artikel, I. (2022). Studi Komparasi Maintainability Antara Aplikasi yang Dikembangkan dengan Framework Flutter dan React Native. *JURNAL INFORMATIKA*, *9*(2). http://ejournal.bsi.ac.id/ejurnal/index.php/ji

Hanes, H., Angela, A., & Br Sembiring, S. (2020). Pengukuran Kualitas Website Penjualan Tiket Dengan Menggunakan Metode Mccall. *JTIK (Jurnal Teknik Informatika Kaputama)*, *4*(2), 81–88. https://doi.org/10.59697/jtik.v4i2.595

Ikhwan, A., Khalis Nugraha, R., Syahnur, E. A., & Ridho, R. (2023). Perancangan Aplikasi Penilaian Kinerja Driver Menggunakan Kodular di Pt Perkebunan Nusantara III Berbasis Mobile. *JOISIE Journal Of Information System And Informatics Engineering*, *7*(2), 364–374. https://doi.org/https://doi.org/10.35145/joisie.v7i2.4001

Laksono, W. P., Satria, B., Wicaksana, T., Wijasena, A. Y., & Sahria, Y. (2024). Implementasi Clean Architecture Dalam Membangun Aplikasi Mobile Menggunakan Flutter. *Nusantara Journal of Multidisciplinary Science*, *2*(1), 173–180. https://jurnal.intekom.id/index.php/njms

Mulatsari, W. E., Candrasari, D. M., & Suyudi, S. (2023). Sistem Informasi Pelayanan Administrasi Kependudukan Kelurahan Kenteng Berbasis Website dengan Uji Kualitas Sistem Menggunakan Metode Mccall Software Quality. *Joined Journal (Journal of Informatics Education)*, *6*(1), 22. https://doi.org/10.31331/joined.v6i1.2597

Munawar, G., Prayoga, R. R., Jumiyani, R., & Syalsabila, A. (2021). Performance Analysis of BLoC and Provider State Management Library on Flutter. *Jurnal Mantik*, *5*(3), 1591–1597.

Muslim, Sari, R. P., & Rahmayuda, S. (2022). Implementasi Framework Flutter Pada Sistem Informasi Perpustakaan Masjid (Studi Kasus: Masjid Di Kota Pontianak). *Coding Jurnal Komputer dan Aplikasi*, *10*(1), 46–59. https://doi.org/https://dx.doi.org/10.26418/coding.v10i01.52178

Muthmainnah, S. F., & Putro, H. P. (2023). Pengujian Nonfungsional dengan Pendekatan McCall's Factor pada Perspektif Product Revision. *AUTOMATA*, *4*(2).

Oktafiani, I., & Saputra, M. F. A. (2022). Pengembangan Aplikasi SOLID-Calculator untuk Pengukuran Kualitas Desain Diagram Kelas. *Jurnal Sains dan Informatika*, *8*(1). https://doi.org/10.22216/jsi.v8i1.959

Ramachandrappa, N. C. (2024). SOLID Design Principles in Software Engineering. *International Journal of Computer Trends and Technology*, *72*(9), 18–23. https://doi.org/10.14445/22312803/IJCTT-V72I9P104

Rusydi, M. H., & Nuryasin, I. (2024). Perancangan UI/UX Aplikasi Hidup Sehat Berbasis Mobile Menggunakan Metode Design Thinking. *JOISIE Journal Of Information Systems And Informatics Engineering*, *8*(1), 54–64. https://doi.org/10.35145/joisie.v8i1.4168

Salamudin, S., Hartati, S., Saputro, H., & Meilantika, D. (2024). Evaluasi Kualitas Portal E-lerning Universitas Mahakarya Asia Menggunakan Metode MCCALL. *Jurnal Nasional Ilmu Komputer*, *5*(1), 16–24. https://doi.org/10.47747/jurnalnik.v5i1.1606

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

Sinatria, M. B., Oman Komarudin, & Kamal Prihamdani. (2023). Penerapan Clean Architecture Dalam Membangun Aplikasi Berbasis Mobile Dengan Framework Google Flutter. *INFOTECH journal*, *9*(1), 132–146. https://doi.org/10.31949/infotech.v9i1.5237

Sondha, A. T., Sa'adah, U., Hardiansyah, F. F., & Rasyid, M. B. A. (2020). Framework dan Code Generator Pengembangan Aplikasi Android dengan Menerapkan Prinsip Clean Architecture. *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, *9*(4), 327–335. https://doi.org/10.22146/jnteti.v9i4.572

Subagio, A. W., & Muttaqin, F. (2022). Penerapan Clean Architecture pada Pengembangan Sistem Payment Point Online Bank. *Jurusan Teknik Elektro*, *32*(2), 324–333. https://doi.org/http://dx.doi.org/10.17977/um034v32i2p324-333

Widiarta, I. M., Julkarnain, M., & Imanulloh, J. (2021). Rancang Bangun Aplikasi Uts In Me Berbasis Android Menggunakan Flutter Dengan Metode Rapid Application Development. *Jurnal Informatika Teknologi dan Sains*, *3*(4), 447–452. https://doi.org/10.51401/jinteks.v3i4.1323

Wijayanto, R. A., Hajar, R. R., & Sejati, P. (2023). Implementing Flutter Clean Architecture for Mobile Tourism Application Development. *International Journal of Computer Applications*, *185*(39), 23–30.

Yadati, N. S. P. K. (2023). Architecture Design (MVVM + Clean Architecture). *Journal of Artificial Intelligence, Machine Learning and Data Science*, *1*(3), 703–706. https://doi.org/10.51219/JAIMLD/naga-satya-praveen-kumar-yadati/177

Yanakiev, I., Lazar, B.-M., & Capiluppi, A. (2024). Applying SOLID principles for the refactoring of legacy code: An experience report. *Journal of Systems and Software*, *220*, 112254. https://doi.org/10.1016/j.jss.2024.112254

Zulistiyan, M., Adrian, M., & Wibowo, Y. F. A. (2024). Performance Analysis of BLoC and GetX State Management Library on Flutter. *Journal of Information System Research (JOSH)*, *5*(2), 583–591. https://doi.org/10.47065/josh.v5i2.4698

https://doi.org/10.35145/joisie.v8i2.4763