



Penggunaan Pola Singleton dalam *Dependency Injection* Dagger Hilt pada Pengembangan Aplikasi Android

Muhammad Sutan Fathir^{1*}, Nabilla Pasha Marcellina²

^{1,2} Informatika, Fakultas Ilmu Komputer, Universitas Singaperbangsa Karawang, Indonesia

Info Artikel

Riwayat Artikel:

Diterima : **16 Mei 2025**

Direvisi : **18 Juni 2025**

Diterbitkan : **25 Juni 2025**

Kata Kunci:

Singleton,
Dependency Injection,
Android,
Clean Architecture

Keywords:

Singleton,
Dependency Injection,
Android,
Clean Architecture

ABSTRAK

Dependency Injection merupakan teknik penting dalam pengembangan aplikasi Android modern untuk meningkatkan modularitas dan kemudahan pengujian, terutama saat menerapkan *Clean Architecture*. Salah satu library yang umum digunakan adalah Dagger Hilt. Namun, penggunaan Dagger Hilt dapat menimbulkan tantangan terkait efisiensi objek yang dikelola dalam siklus hidup aplikasi. Dalam penelitian ini, dilakukan penerapan dan evaluasi pola Singleton pada implementasi Dagger Hilt dalam konteks *Clean Architecture* guna meningkatkan efisiensi pengelolaan objek. Metode yang digunakan adalah pendekatan studi kasus, dengan mengimplementasikan pola Singleton pada modul aplikasi Android yang telah dikembangkan sebelumnya. Aplikasi ini berfungsi sebagai media pengujian untuk menganalisis dampak penerapan terhadap efisiensi memori, kecepatan inisialisasi objek, serta keterbacaan dan pemeliharaan kode. Hasil pengujian menunjukkan bahwa integrasi pola Singleton dalam Dagger Hilt mendukung prinsip *Clean Architecture* dengan mengurangi penggunaan memori dan mempercepat proses inisialisasi tanpa mengorbankan kualitas struktur kode.

ABSTRACT

Dependency Injection is a crucial technique in modern Android application development to improve modularity and testability, especially when applying Clean Architecture. One commonly used library is Dagger Hilt. However, using Dagger Hilt may introduce challenges related to the efficiency of object management within the application's lifecycle. This study applies and evaluates the Singleton pattern within a Dagger Hilt implementation in the context of Clean Architecture to enhance object management efficiency. An case study approach was carried out by integrating the Singleton pattern into modules of a pre-developed Android application. The application serves as a testing medium to analyze the impact of the pattern on memory efficiency, object initialization speed, and code readability and maintainability. The results show that incorporating the Singleton pattern into Dagger Hilt supports Clean Architecture principles by reducing memory usage and accelerating initialization without compromising code quality.

Penulis Korespondensi:

Muhammad Sutan Fathir,
Informatika, Fakultas Ilmu Komputer,
Universitas Singaperbangsa Karawang
Email: muhammadsutanfathir@gmail.com

This is an open access article under the [CC BY-SA](#) license



1. PENDAHULUAN

Perangkat lunak modern pada umumnya terdiri dari berbagai komponen saling terkait yang dirancang agar modular, terstruktur, dan mudah dipelihara. Salah satu pendekatan yang mendukung pencapaian tujuan

tersebut adalah penggunaan *design pattern*, yaitu pola desain berulang yang menawarkan solusi terhadap permasalahan umum dalam rekayasa perangkat lunak [1]. *Singleton pattern* merupakan salah satu pola desain yang populer dan berfungsi untuk memastikan bahwa hanya terdapat satu instansi dari suatu kelas dalam siklus hidup aplikasi [2].

Keterbatasan dalam pembuatan dan pengelolaan instance tunggal seringkali menyebabkan pengembang menghadapi tantangan pada aspek konsistensi data dan efisiensi sumber daya. Oleh karena itu, pola Singleton sering dikombinasikan dengan *Dependency Injection* (DI) untuk memastikan bahwa manajemen dependensi dapat dilakukan secara otomatis, efisien, dan terkontrol [3]. DI sendiri merupakan teknik untuk mengurangi ketergantungan langsung antar komponen dengan memisahkan logika pembuatan objek dari penggunaannya, sehingga meningkatkan fleksibilitas, keterujian (*testability*), dan pemeliharaan kode [4].

Dalam konteks pengembangan aplikasi Android, pendekatan DI mengalami perkembangan pesat dengan hadirnya berbagai *framework*, salah satunya adalah Dagger Hilt, yang dikembangkan oleh Google untuk menyederhanakan integrasi DI dengan arsitektur komponen Android [5]. Dagger Hilt mendukung berbagai skop penyuntikan, termasuk *application scope* yang menggunakan anotasi `@Singleton` untuk menjamin satu instance digunakan secara global. Hal ini penting untuk pengelolaan sumber daya seperti koneksi jaringan (misal: Retrofit), database (Room), atau penyimpanan preferensi [6].

Sejumlah penelitian sebelumnya telah mengkaji manfaat penggunaan *design pattern* dan DI secara terpisah. Telah ditemukan bahwa pola Singleton dapat meningkatkan keterbacaan dan kemudahan pemeliharaan kode dalam pengembangan sistem pembelajaran [7], sementara DI terbukti mampu mempercepat proses pengembangan dan mengurangi kompleksitas arsitektur [8]. Penelitian lain membandingkan berbagai pendekatan DI, termasuk implementasinya dalam Java EE, namun belum mengkaji penggabungannya secara spesifik dengan pola Singleton [9].

Selain itu, beberapa studi menyatakan bahwa penggunaan pola desain berdampak pada kualitas internal kode, seperti keterbacaan dan keterawatan [10], serta keberadaan *code smells* yang mempengaruhi pemeliharaan sistem jangka panjang [11]. Studi lain menekankan pentingnya pemilihan pola desain yang sesuai dalam pengembangan perangkat lunak berbasis modular, termasuk pada arsitektur mikroservis [12].

Penelitian ini hadir untuk mengisi celah tersebut dengan mengkaji secara mendalam penggunaan pola Singleton dalam penerapan DI melalui Dagger Hilt pada pengembangan aplikasi Android. Fokus utamanya adalah bagaimana penggabungan ini berpengaruh terhadap performa aplikasi, efisiensi sumber daya, serta kemudahan pemeliharaan kode. Selain itu, penelitian ini juga mengevaluasi praktik terbaik dalam integrasi keduanya untuk menghasilkan arsitektur aplikasi yang optimal, ringan, dan berkelanjutan.

Dengan pendekatan ini, diharapkan hasil penelitian dapat memberikan kontribusi pada praktik pengembangan aplikasi Android yang lebih efisien dan dapat dijadikan rujukan oleh pengembang maupun akademisi dalam memahami manfaat kolaboratif antara pola Singleton dan DI menggunakan Dagger Hilt.

2. METODE PENELITIAN

Metode penelitian yang digunakan adalah pendekatan studi kasus, yang merupakan salah satu metode kualitatif yang banyak digunakan untuk mengkaji fenomena secara mendalam dalam konteks nyata [13]. Pendekatan ini memungkinkan investigasi intensif terhadap suatu objek atau sistem yang kompleks, sehingga dapat memahami dinamika dan hasil yang terjadi secara detail serta kontekstual. Studi kasus sangat cocok untuk penelitian pengembangan perangkat lunak yang memerlukan evaluasi mendalam terhadap implementasi teknis dan dampaknya pada performa aplikasi [14].

Penelitian ini memfokuskan studi kasus pada penerapan pola Singleton dalam *Dependency Injection* menggunakan Dagger Hilt pada aplikasi Android berbasis *Clean Architecture*. Studi dilakukan dengan mengamati perilaku aplikasi selama siklus hidup tertentu, dengan pencatatan data terkait penggunaan memori, waktu inisialisasi, dan jumlah instance objek. Pendekatan ini bertujuan untuk memperoleh gambaran menyeluruh mengenai pengaruh pola desain terhadap efisiensi pengelolaan dependensi serta performa aplikasi.

2.1. Tahapan Penelitian

Penelitian ini menggunakan pendekatan studi kasus untuk menganalisis penerapan pola Singleton dalam *Dependency Injection* menggunakan Dagger Hilt pada aplikasi Android berbasis *Clean Architecture*. Fokus penelitian terletak pada dua modul utama yang sering digunakan dalam pengembangan aplikasi, yaitu DatabaseModule (Room) dan NetworkModule (Retrofit), yang dikonfigurasi menggunakan anotasi @Singleton dan diinstal pada SingletonComponent [15],[16].

Studi dilakukan dengan menjalankan aplikasi dalam kondisi *fresh install* dan mengamati perilaku runtime aplikasi selama satu menit, berdasarkan skenario aktivitas sebagai berikut:

- Detik 0–30: Aplikasi diluncurkan dan menampilkan HomeFragment dalam MainActivity.
- Detik 30–45: Pengguna melakukan navigasi ke FavoriteActivity.
- Detik 45–60: Pengguna kembali ke HomeFragment dalam MainActivity.

Selama skenario tersebut, dilakukan pencatatan jumlah instance objek yang terbentuk, waktu inisialisasi objek, serta penggunaan memori aplikasi. Pengamatan dilakukan secara berulang untuk memastikan konsistensi data dan mengurangi potensi kesalahan akibat faktor eksternal, seperti latensi perangkat atau gangguan sistem operasi.

2.2. Implementasi Studi Kasus

Aplikasi yang diuji dalam penelitian ini dikembangkan menggunakan pendekatan *Clean Architecture* yang membagi struktur kode ke dalam tiga lapisan utama, yaitu data, domain, dan presentation [17]. Pendekatan ini memungkinkan pemisahan tanggung jawab yang jelas dan memudahkan penerapan pola desain seperti Singleton.

Untuk mengevaluasi pengaruh pola Singleton terhadap efisiensi aplikasi, dibuat implementasi Modul Room dan Retrofit yang diatur menggunakan @InstallIn(SingletonComponent::class) dan anotasi @Singleton. Konfigurasi ini menjamin bahwa objek yang dibutuhkan hanya diinisialisasi satu kali dan digunakan secara global selama siklus hidup aplikasi.

```
@Module
@InstallIn(SingletonComponent::class)
class DatabaseModule {

    @Singleton
    @Provides
    fun provideDatabase(@ApplicationContext context: Context): TeamDatabase {
        val passphrase: ByteArray = SQLiteDatabase.getBytes("dicoding".toCharArray())
        val factory = SupportFactory(passphrase)

        return Room.databaseBuilder(
            context,
            TeamDatabase::class.java, name: "Team.db"
        ).fallbackToDestructiveMigration()
            .openHelperFactory(factory)
            .build()
    }

    @Provides
    fun provideTeamDao(database: TeamDatabase): TeamDao = database.teamDao()
}
```

Gambar 1. Kode DatabaseModule menggunakan SingletonComponent dan @Singleton

```

@Module
@InstallIn(SingletonComponent::class)
class NetworkModule {

    @Provides
    fun provideOkHttpClient(): OkHttpClient {
        val hostname = "thesportsdb.com"
        val certificatePinner = CertificatePinner.Builder()
            .add(hostname, pins: "sha256/07qE5AgzpIvXAx8JRBWLZxWI3vP5VHPoiUhgLH+t/LE=")
            .add(hostname, pins: "sha256/yDu9og255NN56Ef+Bwa9rTrqFQ0EydZ0r1FCh9TdAW4=")
            .add(hostname, pins: "sha256/GQErB2aMPC563kWgOpDVzgz3KJt042PVLu+c5e1/RA=")
            .build()
        return OkHttpClient.Builder()
            .addInterceptor(HttpLoggingInterceptor().setLevel(HttpLoggingInterceptor.Level.BODY))
            .connectTimeout(timeout: 120, TimeUnit.SECONDS)
            .readTimeout(timeout: 120, TimeUnit.SECONDS)
            .certificatePinner(certificatePinner)
            .build()
    }
}

```

Gambar 2. Kode NetworkModule menggunakan SingletonComponent

Potongan kode konfigurasi DatabaseModule dan NetworkModule yang menggunakan anotasi @Singleton dan diinstal pada SingletonComponent. Konfigurasi ini menjamin bahwa instance dari modul database (Room) dan jaringan (Retrofit) hanya dibuat satu kali selama siklus hidup aplikasi dan digunakan secara global. Pendekatan ini mengoptimalkan efisiensi pengelolaan dependensi, menghindari inisialisasi ulang objek yang tidak perlu, serta meningkatkan performa dan stabilitas aplikasi saat navigasi antar-aktivitas berlangsung. *Logging* dilakukan pada setiap aktivitas dan fragment menggunakan Logcat untuk mendeteksi proses inisialisasi objek dan mencatat waktu inisialisasi menggunakan System.nanoTime(). Selain itu, monitoring penggunaan memori dilakukan dengan Android Profiler di Android Studio [18],[19].

2.3. Pengumpulan dan Analisis Data

Data dikumpulkan berlangsung selama 60 detik dengan tiga titik observasi utama: saat aplikasi pertama kali dibuka (detik 0), saat navigasi ke FavoriteActivity (detik 30), dan saat kembali ke HomeFragment (detik 45). Parameter yang dianalisis meliputi:

- Jumlah instance objek Room dan Retrofit, yang berfungsi sebagai indikator utama efektivitas pola Singleton dalam mengelola siklus hidup objek agar tidak terjadi instansiasi berulang yang dapat membebani memori dan performa aplikasi.
- Waktu inisialisasi rata-rata (ms), untuk mengukur seberapa cepat modul-modul penting tersebut dapat siap digunakan saat aplikasi atau komponen terkait diluncurkan, sehingga berdampak pada pengalaman pengguna yang lebih responsif.
- Penggunaan memori (MB), yang menjadi ukuran kritis dalam konteks aplikasi mobile, terutama untuk menghindari konsumsi sumber daya berlebih yang dapat menyebabkan aplikasi menjadi lambat atau rentan terhadap crash akibat kekurangan memori.

Untuk meningkatkan reliabilitas, pengujian dilakukan secara berulang dengan kondisi *fresh install* pada setiap sesi agar kondisi lingkungan pengujian tetap terkontrol dan tidak terpengaruh oleh cache atau data residual aplikasi sebelumnya. Hal ini penting agar hasil pengukuran benar-benar mencerminkan efek dari penerapan pola Singleton dalam konteks *Dependency Injection* menggunakan Dagger Hilt.

Selanjutnya, hasil pengumpulan data ini akan digunakan sebagai dasar untuk mengevaluasi sejauh mana pola Singleton mampu meningkatkan efisiensi pengelolaan dependensi serta dampaknya terhadap performa aplikasi secara keseluruhan, termasuk potensi penghematan penggunaan memori dan percepatan waktu inisialisasi yang diharapkan mampu memberikan pengalaman pengguna yang lebih optimal.

3. HASIL DAN ANALISIS

Penelitian ini mengevaluasi penerapan pola Singleton dalam *Dependency Injection* menggunakan Dagger Hilt pada sebuah aplikasi Android berbasis *Clean Architecture*. Evaluasi dilakukan berdasarkan data yang diperoleh dari proses observasi runtime aplikasi, yang telah dikonfigurasi untuk menggunakan modul Room dan Retrofit dengan anotasi @Singleton serta @InstallIn(SingletonComponent::class). Analisis ini bertujuan

untuk menguji efektivitas pola Singleton dalam pengelolaan objek dan dampaknya terhadap performa serta efisiensi penggunaan sumber daya aplikasi.

3.1. Jumlah Instance Objek

Evaluasi terhadap jumlah instance objek dilakukan melalui pencatatan log aktivitas aplikasi menggunakan Logcat selama siklus hidup aplikasi berlangsung. Fokus observasi berada pada dua komponen utama yang dikelola oleh dependency injection, yaitu Retrofit dan Room Database, yang masing-masing merepresentasikan modul jaringan dan basis data dalam arsitektur aplikasi.

```
2025-05-15 21:06:34.161 9514-9514 DI_DEBUG_Instance com.example.submissionandroidexpert D Retrofit instance created with ID: 137192394
2025-05-15 21:06:34.185 9514-9514 DI_DEBUG_Instance com.example.submissionandroidexpert D Room Database instance created with ID: 1456
```

Gambar 3. Inisialisasi Instance Objek melalui Logcat saat Aplikasi Berjalan

Keberadaan satu instance untuk masing-masing objek menunjukkan bahwa strategi *Dependency Injection* yang digunakan berada dalam lingkup `@SingletonComponent`, yang berarti seluruh objek dikelola sebagai singleton dalam cakupan aplikasi secara keseluruhan. Pendekatan tersebut memastikan bahwa objek hanya diinisialisasi satu kali selama aplikasi berjalan, dan digunakan secara global oleh semua komponen yang membutuhkannya.

Konfigurasi ini mencerminkan prinsip dari pola desain Singleton, yang bertujuan membatasi jumlah instance dari suatu kelas agar hanya satu yang aktif pada suatu waktu. Dalam konteks pengembangan aplikasi Android berbasis *Clean Architecture*, penggunaan pola ini memberikan keuntungan signifikan dalam hal efisiensi memori dan stabilitas pengelolaan dependensi. Objek seperti Retrofit yang berfungsi sebagai client komunikasi jaringan, dan Room Database sebagai pengelola data lokal, merupakan komponen berat yang idealnya tidak dibuat berulang kali. Dengan demikian, pendekatan singleton tidak hanya mengurangi overhead memori, tetapi juga mempercepat waktu akses dan respons aplikasi.

Selain itu, penggunaan pola Singleton juga mengurangi potensi terjadinya inkonsistensi data akibat adanya beberapa instance berbeda yang dapat menghasilkan perilaku tidak terduga. Hal ini sangat penting dalam aplikasi yang mengandalkan data sinkronisasi secara real-time atau pengelolaan state yang konsisten di seluruh modul aplikasi. Keberhasilan pengelolaan instance tunggal ini juga berkontribusi pada pengurangan kompleksitas debugging dan pemeliharaan kode.

3.2. Waktu Inisialisasi Rata-rata

Waktu yang dibutuhkan untuk menginisialisasi objek Room dan Retrofit dicatat menggunakan `System.nanoTime()` selama aplikasi pertama kali dijalankan. Proses inisialisasi hanya terjadi satu kali karena pola Singleton menjamin reuse objek selama siklus hidup aplikasi. Hasil pengukuran menunjukkan bahwa waktu inisialisasi kedua modul tersebut relatif singkat, yang berkontribusi pada kecepatan *startup* aplikasi dan memberikan pengalaman pengguna yang responsif.

Tabel 1. Waktu Inisialisasi

Komponen	Waktu Inisialisasi (ms)
Room (Database Module)	22
Retrofit (Network Module)	17

```
2025-05-15 21:06:34.161 9514-9514 DI_DEBUG_Waktu com.example.submissionandroidexpert D Retrofit initialization time: 17 ms
2025-05-15 21:06:34.185 9514-9514 DI_DEBUG_Waktu com.example.submissionandroidexpert D Room Database initialization time: 22 ms
```

Gambar 4. Cuplikan Logcat Selama Tahap Inisialisasi Sistem

Waktu inisialisasi untuk Room Database sedikit lebih tinggi dibandingkan Retrofit, yang diperkirakan disebabkan oleh proses pembukaan koneksi ke file database lokal dan penguraian struktur tabel awal. Namun, waktu inisialisasi secara keseluruhan tetap berada dalam batas optimal (di bawah 100ms), yang menunjukkan bahwa pemanfaatan Singleton tidak hanya mengurangi frekuensi inisialisasi, tetapi juga menjaga performa pada tahap *startup* aplikasi.

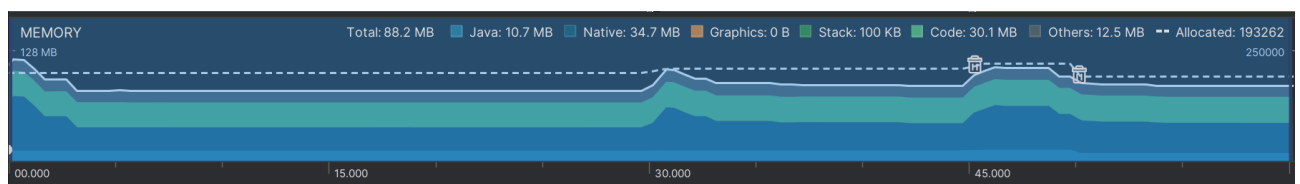
3.3. Penggunaan Memori

Efisiensi memori merupakan salah satu parameter penting dalam pengembangan aplikasi mobile, terutama pada perangkat dengan keterbatasan sumber daya. Pengamatan dilakukan melalui Android Profiler selama durasi 60 detik, mencakup fase inialisasi awal, navigasi antar aktivitas, hingga kembali ke kondisi semula.

Hasil pemantauan menunjukkan bahwa aplikasi mencatat penggunaan memori maksimum sebesar 128 MB pada saat awal eksekusi, kemudian stabil di kisaran 88,2 MB selama proses berlangsung. Profil memori memperlihatkan tidak adanya lonjakan signifikan meskipun terjadi perpindahan antar aktivitas, yang menandakan tidak terjadinya alokasi ulang objek berat seperti Retrofit dan Room Database. Stabilitas ini mendukung asumsi bahwa objek-objek tersebut dikelola sebagai instance tunggal melalui mekanisme Singleton, tanpa pembentukan ulang saat navigasi dilakukan.

Tabel 2. Rincian Penggunaan Memori Aplikasi Berdasarkan Jenis Alokasi Selama Pengujian.

Komponen Memori	Ukuran (MB)
Java Heap	10.7
Native Heap	34.7
Code	30.1
Stack	0.1
Others	12.5
Total	88.2



Gambar 5. Grafik Penggunaan Memori Aplikasi Selama 60 Detik Siklus Hidup Berdasarkan Android Profiler

Grafik memori menunjukkan dua titik alokasi tambahan kecil pada sekitar detik ke-30 dan ke-45, yang berkorelasi dengan interaksi pengguna saat berpindah ke FavoriteActivity dan kembali ke HomeFragment. Namun, fluktuasi ini bersifat minimal dan tidak mengubah tren utama dari penggunaan memori secara keseluruhan.

Dengan memanfaatkan pola Singleton pada level @SingletonComponent, aplikasi mampu menjaga alokasi memori tetap efisien dan konsisten. Efisiensi ini sejalan dengan prinsip *Clean Architecture* yang menekankan pemisahan tanggung jawab serta pengelolaan dependensi secara terkontrol. Hal ini menunjukkan bahwa integrasi pola desain dan framework *Dependency Injection* dapat memberikan dampak langsung terhadap performa aplikasi, terutama dalam hal konsumsi sumber daya.

4. KESIMPULAN

Penelitian ini berhasil mengkaji penerapan pola Singleton dalam *Dependency Injection* menggunakan Dagger Hilt pada pengembangan aplikasi Android berbasis *Clean Architecture*. Hasil penelitian menunjukkan bahwa implementasi pola Singleton secara efektif mengontrol jumlah instance objek penting seperti Room Database dan Retrofit Network Client, sehingga hanya satu instance yang digunakan selama siklus hidup aplikasi. Hal ini memberikan dampak positif berupa pengurangan penggunaan memori dan peningkatan performa aplikasi, khususnya dalam waktu inialisasi objek yang tetap singkat meskipun hanya terjadi sekali pada saat aplikasi diluncurkan.

Analisis runtime memperlihatkan kestabilan penggunaan memori selama aktivitas aplikasi berlangsung, tanpa adanya alokasi ulang objek yang tidak perlu. Ini mendukung prinsip efisiensi sumber daya yang krusial dalam pengembangan aplikasi mobile. Selain itu, penerapan pola Singleton pada modul-modul tersebut juga memberikan kemudahan dalam pemeliharaan kode dengan mengurangi kompleksitas pengelolaan dependensi.

Secara keseluruhan, integrasi pola Singleton dengan Dagger Hilt mendukung prinsip *Clean Architecture* dalam menciptakan aplikasi yang modular, efisien, dan mudah dipelihara. Penelitian ini dapat menjadi referensi praktis bagi pengembang aplikasi Android dalam mengimplementasikan teknik *Dependency Injection* yang efisien dan efektif.

Sebagai prospek pengembangan, penelitian selanjutnya dapat memperluas cakupan studi dengan menguji pola Singleton pada modul lain atau mengkombinasikannya dengan pola desain lain untuk mengoptimalkan performa lebih lanjut. Selain itu, evaluasi pada berbagai tipe perangkat dan kondisi nyata penggunaan juga dapat memperkaya validitas temuan dan meningkatkan aplikasi praktis di dunia pengembangan aplikasi mobile.

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada penyedia Jurnal Sistem Informasi dan Informatika (JUSIFOR) atas kesempatan dan dukungan dalam publikasi hasil penelitian ini. Ucapan terima kasih juga ditujukan kepada para penulis dan peneliti terdahulu yang karyanya dijadikan rujukan dan sumber informasi penting dalam penyusunan penelitian ini. Kontribusi mereka sangat berarti dalam memperkaya kajian serta memperkuat landasan teoritis penelitian.

REFERENSI

- [1] A. R. Fajri, "Penerapan Design Pattern MVVM dan *Clean Architecture* pada Pengembangan Aplikasi Android (Studi Kasus: Aplikasi Agree Partner)," *Off. Sci. Journals Univ. Islam Indones.*, p. 12, 2022.
- [2] K. Aljasser, "Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns," *Comput. Lang. Syst. Struct.*, vol. 45, pp. 1–15, 2016, doi: 10.1016/j.cl.2015.11.002.
- [3] M. D. Ekstrand and M. Ludwig, "*Dependency Injection* with Static Analysis and Context-Aware Policy.," *J. Object Technol.*, vol. 15, no. 1, p. 1:1, 2016, doi: 10.5381/jot.2016.15.1.a1.
- [4] G. Bagus, V. Putraadinatha, D. Dwi, J. Suwawi, and S. Y. Puspitasari, "Pengaruh Design Pattern Terhadap Maintainability Aplikasi Mobile," *E-Proceeding Eng.*, vol. 8, no. 5, p. 10954, 2021.
- [5] Android Developers, "Guide to Dagger Hilt," Google Developers. [Online]. Available: <https://developer.android.com/training/dependency-injection/hilt-android>
- [6] D. Lyon and F. Castellanos, "The parametric singleton design pattern," *J. Object Technol.*, vol. 6, no. 3, pp. 13–23, 2007, doi: 10.5381/jot.2007.6.3.c2.
- [7] A. Kevin and H. C. Kurniawan, "Analisis Pengaruh Design Pattern Terhadap Pemeliharaan Perangkat Lunak Learning Management System," *J. Telemat.*, vol. 18, no. 1, pp. 21–31, 2023, doi: 10.61769/telematika.v18i1.543.
- [8] J. A. N. Hasim, U. Sa'adah, D. I. P. Sari, F. A. Damastuti, and F. N. Koirudin, "Developing Microframework based on Singleton and Abstract Factory Design Pattern," *IES 2022 - 2022 Int. Electron. Symp. Energy Dev. Clim. Chang. Solut. Clean Energy Transition, Proceeding*, no. October, pp. 676–683, 2022, doi: 10.1109/IES55876.2022.9888548.
- [9] F. Younis and O. M. Sallabi, "University of Benghazi Department of Software Engineering *Dependency Injection* (DI) in Java .," no. December, 2018.
- [10] M. Alfadel, K. Aljasser, and M. Alshayeb, *Empirical study of the relationship between design patterns and code smells*, vol. 15, no. 4. 2020. doi: 10.1371/journal.pone.0231731.
- [11] T. Alkhaeir and B. Walter, "The Effect of Code Smells on the Relationship between Design Patterns and Defects," *IEEE Access*, vol. 9, pp. 3360–3373, 2021, doi: 10.1109/ACCESS.2020.3047870.
- [12] A. Qusef, M. Mohammed, and M. O. Elish, "Modularity Analysis of Design Patterns: An Empirical Study," *Int. J. Softw. Eng. Its Appl.*, vol. 11, no. 6, pp. 61–78, 2017, doi: 10.14257/ijseia.2017.11.6.06.
- [13] T. Hollweck, "Robert K. Yin. (2014). Case Study Research Design and Methods (5th ed.).," *Can. J.*

- Progr. Eval.*, vol. 30, no. 1, pp. 108–110, 2015, doi: 10.3138/cjpe.30.1.108.
- [14] D. Assyakurrohim, D. Ikhram, R. A. Sirodj, and M. W. Afgani, "Metode Studi Kasus dalam Penelitian Kualitatif," *J. Pendidik. Sains dan Komput.*, vol. 3, no. 01, pp. 1–9, 2022, doi: 10.47709/jpsk.v3i01.1951.
- [15] K. Kartarina and A. Apriliansyah, "Analisis *Dependency Injection* dan Model-View-Presenter Pada Aplikasi Berbasis Android," *Progresif J. Ilm. Komput.*, vol. 18, no. 1, p. 23, 2022, doi: 10.35889/progresif.v18i1.781.
- [16] Dagger Team, "Migrating to Hilt." [Online]. Available: <https://dagger.dev/hilt/migration-guide.html>
- [17] T. B. Duy, "Reactive Programming and *Clean Architecture* in Android Development," no. April, pp. 1–47, 2017.
- [18] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A Survey of Performance Optimization for Mobile Applications," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 2879–2904, 2022, doi: 10.1109/TSE.2021.3071193.
- [19] F. Stanislav, "Optimize the Performance of Android Mobile Apps Using Android Studio and SDK," vol. 8, no. 3, pp. 11–14, 2024.