Information System Journal (INFOS)
Vol. 7, No. 1, Mei 2024, pp. 20-33
E-ISSN : 2655-142X , P-ISSN : 2655-190X, DOI:https://doi.org/10.24076/infosjournal.2024v7i01.1579

REFACTORING APLIKASI XYZ MENGGUNAKAN PENDEKATAN *CLEAN ARCHITECTURE*

Bowo Nugroho [1), Nur Fajri Azhar [2), Putra Cendikia Subekti [3)
[1), 2),3)] *Informatika, Jurusan Matematika dan Teknologi Informasi, Institut Teknologi Kalimantan*
email : *bowo.nugroho@lecturer.itk.ac.id*[1)], *fajri@lecturer.itk.ac.id*[2)], 11201074@student.itk.ac.id[3)]

INFO ARTIKEL

Riwayat Artikel:

Diterima Maret, 2024
Revisi Mei, 2024
Terbit Mei, 2024

Penulis Korespondensi:

Bowo Nugroho
Informatika, Jurusan Matematika dan Teknologi Informasi, Institut Teknologi Kalimantan

Email:
bowo.nugroho@lecturer.itk.ac.id

ABSTRAK

Implementasi *Clean Architecture* dalam pengembangan aplikasi *Android* menggunakan bahasa pemrograman *Kotlin*, dengan fokus pada penerapan prinsip desain SOLID. Evaluasi dilakukan pada dampak *Clean Architecture* terhadap kompleksitas kode yang menunjukkan penurunan rata-rata nilai kompleksitas dari 1,5 menjadi 1,4. Penelitian juga menganalisis perubahan kinerja aplikasi sebelum dan setelah menerapkan *Clean Architecture,* dengan penekanan pada penggunaan CPU, memori dan waktu eksekusi, Metodologi penelitian mengikuti pendekatan *waterfall* untuk *refactoring* aplikasi XYZ. Hasil penelitian memberikan pemahaman mendalam tentang perubahan kode dalam implementasi *Clean Architecture* dalam pengembangan aplikasi XYZ. Kesimpulannya, penerapan *Clean Architecture* dapat mengurangi kompleksitas kode dan berpotensi meningkatkan kinerja aplikasi, memberikan landasan yang kuat untuk pengembangan perangkat lunak yang lebih efisien dan dapat dipelajari oleh peneliti di bidang ini.

Kata Kunci :

Clean Architecture; *Kotlin*; *Android*

ABSTRACT

Clean Architecture in Android application development using the Kotlin programming language, with a focus on applying the principles of SOLID design. Evaluation was conducted on the impact of Clean Architecture on code complexity revealing an average complexity score decrease from 1.5 to 1.4. The research methodology follows a waterfall approach for refactoring the XYZ application. The findings provide a deep understanding of code changes in the implementation of Clean Architecture in the development of the XYZ application. In conclusion, the adoption of Clean Architecture can reduce code complexity and potentially improve application performance, providing a strong foundation for more efficient software development that can be studied by researchers in this field.

***Keywords* :**
Clean Architecture; Kotlin; Android

1. PENDAHULUAN

Topik mengenai penghematan energi telah mendapat perhatian dalam beberapa tahun terakhir. Hal ini terjadi karena kekhawatiran global terkait dengan kehabisan sumber daya energi yang semakin cepat, masalah lingkungan, kenaikan harga energi, yang diiringi dengan peningkatan konsumsi energi. Oleh karena itu, penghematan energi merupakan langkah kritis menuju keberlanjutan energi di kota-kota. Pengembangan kota melalui pendekatan pintar untuk mencapai kota lebih efisien secara energi memainkan peran penting dalam pembangunan berkelanjutan. Oleh karena itu, penghematan energi dan pengurangan karbon merupakan faktor krusial untuk mencapai kota dan masyarakat yang berkelanjutan [1].

Aplikasi XYZ bertujuan untuk memberikan edukasi mengenai seberapa besar dampak yang dapat ditimbulkan oleh penggunaan perkakas elektronik di dalam rumah tangga, manfaaat dari aplikasi ini adalah meningkatkan penghematan pemakaian energi listrik di sektor rumah tangga. Aplikasi XYZ dalam tahap pengembangannya tidak menerapkan pola arsitektur tertentu, maka kode program dapat menjadi tidak konsisten dan sulit dipahami oleh pengembang lain. Pemisahan tugas kepada para pengembang juga akan menjadi tugas yang rumit, meningkatkan risiko konflik saat proyek tumbuh lebih besar. Pengembang juga akan menghadapi kesulitan saat harus memperbarui kode jika ada perubahan dalam proses bisnis, dan mereka juga akan mengalami hambatan saat melakukan pengujian dan pemeliharaan kode. Oleh karena itu, penting untuk menerapkan suatu kerangka kerja arsitektur dan pola desain dalam pengembangan aplikasi *Android*.

Clean Architecture mendorong *programer* untuk memisahkan *business rules* yang lebih stabil (*higher-level abstractions*) dengan detail-detail teknis yang mudah berubah (*lower-level details*). *Clean Architecture* dilakukan dengan menetapkan batasan yang jelas antara kedua unsur tersebut. Ketergantungan kode harus mengarah ke dalam, ke arah kebijakan tingkat yang lebih tinggi [2]. Implementasi *Clean Architecture* membuat suatu perangkat lunak menjadi *modular*, mudah untuk dikembangkan dan mudah untuk diuji [3], [4]. Keuntungan yang didapat dari penggunaan arsitektur ini terlihat ketika diimplementasikan dalam proyek skala besar [5].

Penelitian terdahulu yang menerapkan *Clean Architecture* berjudul "Pengembangan Aplikasi Perangkat Bergerak Untuk Pencarian Partner Lomba Berbasis *Android* Dengan Penerapan *Clean Architecture"* menggunakan 3-layer yaitu *domain layer, data layer, dan presentation layer* yang diimplementasikan pada 19 halaman antarmuka. Penelitian ini menghasilkan nilai efektivitas sebesar 90,76% [6].

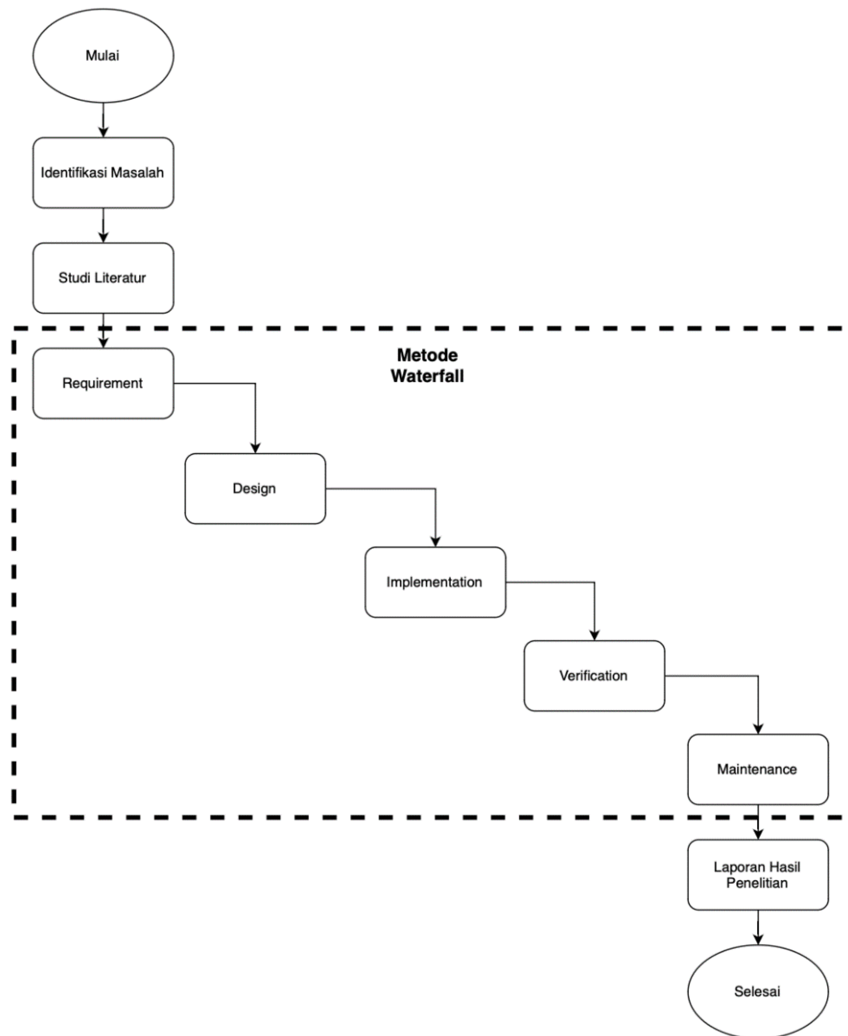
Penelitian berjudul "Aplikasi perangkat bergerak system informasi *event* berbasis *Android*" mengimplementasikan *Clean Architecture* pada 29 kebutuhan fungsional dan 2 kebutuhan non fungsional memperoleh hasil pengujian *black box* sebesar 100% [7]. Penelitian lain mengenai *clean architecture* memiliki kesimpulan kode yang dibangun dengan menggunakan arsitektur ini memiliki tingkat *maintainability* yang baik [8], [9].

Refactoring adalah proses mengubah sistem perangkat lunak sedemikian rupa sehingga tidak mengubah perilaku eksternal kode, namun memperbaiki struktur internalnya [10], [11]. *Refactoring* adalah cara yang disiplin untuk membersihkan kode yang meminimalkan kemungkinan munculnya *bug*. *Refactoring* dapat mengambil desain yang buruk, bahkan kacau, dan mengerjakannya kembali menjadi kode yang dirancang dengan baik. Efek kumulatif dari perubahan kecil ini dapat meningkatkan desain secara signifikan [12]. *Refactoring* perlu dilakukan karena kebutuhan perangkat lunak yang terus berkembang yang berarti perangkat lunak akan ditingkatkan, dimodifikasi, dan disesuaikan dengan kebutuhan baru sehingga kode menjadi lebih kompleks dan berubah dari desain aslinya [13], [14], [15].

Penelitian ini bertujuan untuk melakukan *Refactoring* terhadap aplikasi XYZ. Hasil dari proses *Refactoring* kemudian dianalisis. Analisis dilakukan untuk mengetahui bagaimana dampak *Clean Architecture* terhadap *Cyclomatic Complexcity,* ukuran aplikasi, penggunaan ram, dan penggunaan SoC dari gawai.

2. METODOLOGI PENELITIAN

Tahapan pertama yang dilakukan adalah identifikasi masalah pada aplikasi XYZ yaitu terkait struktur atau arsitektur kode yang digunakan pada aplikasi XYZ. Tahap kedua yaitu studi literatur yaitu pengkajian referensi dari jurnal, *e-book,* atau bahan lainnya yang berkaitan dengan *state of the art* mengenai *clean architecture* dan *refactoring*. Tahap ketiga yaitu analisis *requirement*, dimana pada tahap ini dilakukan pengumpulan dan analisis data dari semua kebutuhan aplikasi XYZ. Tahap keempat adalah *design* untuk mengembangkan sistem berdasarkan analisis data kebutuhan yang sudah dilakukan. Tahap ini melibatkan pembangunan desain antarmuka pengguna (UI) dan desain arsitektur aplikasi XYZ. Tahap kelima *implementation* untuk mengimplementasi desain yang telah dibuat pada tahapa *design*. Tahap *verification* atau *testing* adalah yang berisi *system testing*. Tahap keenam adalah *maintenance* yang berisi perbaikan dari aplikasi yang sudah dibangun. Tahap terakhir pada penilitian ini yaitu penulisan laporan hasil penelitian yang mendokumentasikan seluruh tahapan dan hasil penilitian yang telah dilakukan. Gambar 1., merupakan alir penelitian dari penelitian ini.



Gambar 1. Diagram Alir Penelitian.

3. HASIL DAN PEMBAHASAN

Proses *Refactoring* dilakukan dengan 2 tahapan yaitu perubahan struktur proyek dan implementasi kode.

3.1 Struktur Proyek

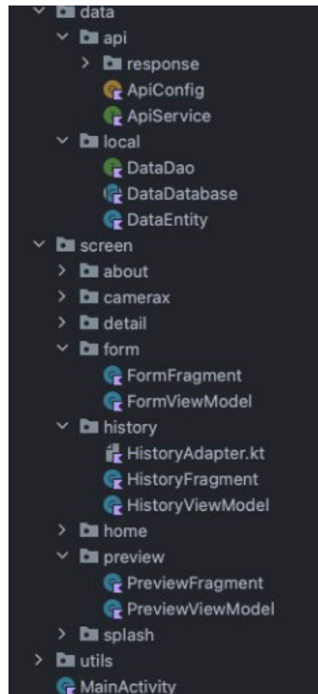
Perubahan struktur proyek merupakan proses perubahan penyesuaian folder-folder di dalam proyek sesuai dengan prinsip *clean architecture.* Gambar 2., merupakan struktur proyek sebelum penerapan *clean architecture*. Proyek terbagi menjadi 3 komponen folder yaitu:

1. *Folder* data yang bertanggung jawab atas komponen dan *service* untuk mengambil data dari sumber data.
2. *Folder screen* berisi komponen yang mewakili tiap *screen* berisi logika dari masing-masing *user interface*.
3. *Folder utils* yang berisi kelas yang digunakan sebagai alat.

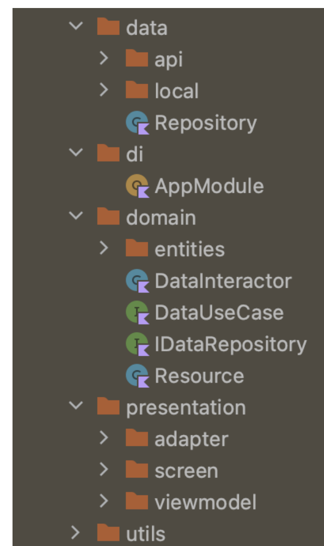
Gambar 3., merupakan struktur proyek XYZ setelah *refactoring* dengan menerapkan *Clean Architecture*. Struktur proyek menjadi:

1. *Folder* data berisi komponen *Data Layer* yaitu yang bertanggung jawab untuk mengolah data yang berasal dari lokal maupun jaringan.
2. *Folder* domain berisi *Domain Layer* yaitu komponen yang akan digunakan oleh *layer* atau lapisan lain dan *layer* lain akan bergantung kuat pada layer yang ada di *folder* domain.

3. *Folder* presentasi terbagi menjadi beberapa folder, yaitu:
   a) *Folder screen* yang memiliki peranan sebagai komponen utama dalam tampilan pengguna dan bertanggung jawab atas keseluruhan logika yang berhubungan dengan antarmuka pengguna.
   b) *Folder viewmodel* adalah komponen yang berfungsi sebagai penghubung antara komponen screen dan komponen *usecase*. Hal ini memungkinkan pengelolaan data dan logika bisnis dalam tampilan pengguna.
   c) *Folder* adapter berperan sebagai pengatur cara data ditampilkan dalam bentuk yang sesuai. Contoh manfaat adapter adalah dalam penyajian tampilan yang lebih kompleks.



Gambar 2. struktur proyek sebelum *clean architecture.*



Gambar 3. struktur proyek setelah *clean architecture*

3.2 Implementasi kode

Implementasi kode merupakan proses mengimplementasikan kode kedalam proyek mengikuti struktur kode yang sudah di buat sebelumnya.

3.2.1 Screen

Screen merupakan bagian kode yang menghandel antarmuka. Dalam bahasa pemrograman kotlin antarmuka dalam suatu *activity* direpresentasikan oleh *Fragment*.

```
package com.example.xyzmvvm.screen.history

...

class HistoryFragment : Fragment() {
    private var _binding: FragmentHistoryBinding? = null
    private val binding get() = _binding!!
    private val historyAdapter: HistoryAdapter by lazy {
        HistoryAdapter { item ->
            val action = HistoryFragmentDirections.actionHistoryFragmentToDetailFragment(item)
            findNavController().navigate(action)
        }
    }
    private lateinit var historyViewModel: HistoryViewModel
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentHistoryBinding.inflate(inflater, container, false)
        return binding.root
    }
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        val factory = ViewModelFactory(requireActivity().applicationContext)
        historyViewModel = ViewModelProvider(requireActivity(),
factory)[HistoryViewModel::class.java]
        setHistory()
    }
    private fun setHistory() {
        with(binding.rvHistory) {
            layoutManager = LinearLayoutManager(requireContext())
            adapter = historyAdapter
        }
 ...
}
```

Gambar 4. *Fragment* Sebelum *Refactor.*

```
package com.zenith.xyz.presentation.screen

...

@AndroidEntryPoint
class HistoryFragment : Fragment() {
    private var _binding: FragmentHistoryBinding? = null
    private val binding get() = _binding!!
    private val historyAdapter: HistoryAdapter by lazy {
        HistoryAdapter { item ->
            val action = HistoryFragmentDirections.actionHistoryFragmentToDetailFragment(item)
            findNavController().navigate(action)
        }
    }
    private val historyViewModel: HistoryViewModel by viewModels()
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentHistoryBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        setHistory()
    }
 ...
}
```

Gambar 5. *Fragment* setelah *Refactor.*

Gambar 4., merupakan contoh kode *Fragment* sebelum dan Gambar 5., merupakan contoh kode setelah *refactoring*. Perubahan yang sama pada logika tiap *User Interface*. Perubahan seperti anotasi *@AndroidEntryPoint* yang digunakan untuk menandai bahwa *fragment* ini akan diinjeksi oleh *Hilt*, yang merupakan kerangka kerja *dependency injection* di Android. *Inisiasi HistoryViewModel* tidak lagi memerlukan *ViewModelFactory* melainkan menggunakan *viewModels()* dengan bantuan kerangka kerja *Hilt* sehingga *Fragment* bisa memanfaatkan *service* dari *HistoryViewModel*.

3.2.2 ViewModel

Viewmodel adalah komponen yang berfungsi sebagai perantara antara *screen* dan *usecase*, memungkinkan pengelolaan data dan logika bisnis dalam antarmuka.

```
package com.example.xyzmvvm.screen.preview

 ...

class PreviewViewModel(private val apiService: ApiService, private val dataDao: DataDao): ViewModel()
{
    private val executorService: ExecutorService = Executors.newSingleThreadExecutor()
    fun uploadPhoto(file: File): LiveData<Result<DataResponse>> {
        val itemResponse = MutableLiveData<Result<DataResponse>>()
        itemResponse.value = Result.Loading
        val newFile = reduceFileImage(file)
        val fileReady = newFile.asRequestBody("image/jpg".toMediaType())
        val multipart: MultipartBody.Part = MultipartBody.Part.createFormData(
            "file",
            newFile.name,
            fileReady
        )
        apiService.uploadPhoto(multipart).enqueue(object : Callback<DataResponse> {
            override fun onResponse(call: Call<DataResponse>, response: Response<DataResponse>) {
                if (response.isSuccessful) {
                    if (response.body()?.itemInfo != null) {
                        val item = response.body()?.itemInfo
                        item?.let {
                            val dataReada = DataEntity(
                                0,
                                item.lokasi,
                                item.averageEnergy.toString(),
                                ...
                            )
                            executorService.execute { dataDao.insertData(dataReady) }
                        }
                        itemResponse.value = Result.Success(response.body()!!)
                    } else {
                        itemResponse.value = Result.Error("Can't detect photo")
                    }
                }
            }
            override fun onFailure(call: Call<DataResponse>, t: Throwable) {
                itemResponse.value = Result.Error(t.message ?: "Unknown Error")
            }
        })
        return itemResponse
    }
}
```

Gambar 6. *ViewModel* Sebelum *Clean Architecture*.

```
package com.zenith.xyz.presentation.viewmodel

...

@HiltViewModel
class PreviewViewModel @Inject constructor(private val dataUseCase: DataUseCase) : ViewModel() {
    fun uploadPhoto(file: File) : LiveData<Resource<ItemData>> {
        return dataUseCase.uploadPhoto(file)
    }
}
```

Gambar 7. *ViewModel* Sebelum *Clean Architecture*.

Gambar 6., menunjukan bahwa *ViewModel* berhubungan langsung dengan *ApiService* sebagai sumber data. *DataDao* sebagai *constructor*, masih menggunakan *Java IO* yaitu *executor service* dengan *Single Thread Executor*. Logika bisnis yang berperan mengambil data masih berada pada *ViewModel*, dimana bagian *presentation* tidak boleh bergantung langsung kepada bagian data.

Gambar 7., menunjukan perubahan pada baris kode dengan menggunakan anotasi *@HiltViewModel*. Hal ini merupakan implementasi *ViewModel* dengan menggunakan *Hilt*. Anotasi *@Inject* berfungsi untuk melakukan injeksi dengan *DataUseCase*. *DataUseCase* memiliki abstraksi sehingga tidak bergantung langsung lagi pada sumber data.

3.2.3 *Domain Layer*

Layer ini adalah *layer* yang akan digunakan oleh *layer* lain atau dapat dikatakan bahwa *layer* lain akan bergantung langsung pada *layer* ini.

```
package com.zenith.xyz.domain

...

interface DataUseCase {
    fun uploadPhoto(file: File) : LiveData<Resource<ItemData>>
    fun getAllHistory() : LiveData<List<ItemData>>
    fun getDevices(location: String): LiveData<Resource<List<Device>>>
    fun calculateEnergy(location: String, device: String, deviceType: String)
            : LiveData<Resource<Energy>>
}
```

Gambar 8. *DataUseCase.*

Gambar 8., merupakan contoh *Interface DataUseCase* berperan sebagai kontrak yang mendefinisikan metode-metode yang bertugas mengelola beragam operasi terkait data. Metode yang ada pada *DataUseCase* sama dengan metode yang ada pada *Interface Repository* atau dalam proyek aplikasi ini bernama *IDataRepository*.

```
package com.zenith.xyz.domain

...

class DataInteractor @Inject constructor(private val repository: IDataRepository) : DataUseCase {
    override fun uploadPhoto(file: File): LiveData<Resource<ItemData>> {
        return repository.uploadPhoto(file)
    }
    override fun getAllHistory(): LiveData<List<ItemData>> {
        return repository.getAllHistory()
    }
    override fun getDevices(location: String): LiveData<Resource<List<Device>>> {
        return repository.getDevicesData(location)
    }
    override fun calculateEnergy(location: String, device: String, deviceType: String)
            : LiveData<Resource<Energy>> {
        return repository.calculateEnergy(location, device, deviceType)
    }
}
```

Gambar 9. *DataInteractor.*

Gambar 9., merupakan implementasi dari kelas *DataInteractor*, yang diinjeksi dengan sebuah *instance* dari *IDataRepository*. *Interface DataUseCase* diimplementasikan pada kelas *DataInteractor*. Kelas ini memiliki sejumlah metode yang bertugas sebagai pengelola operasi data dalam aplikasi. Seluruh metode dalam *DataInteractor* mengambil argumen yang sesuai dan mengembalikan *LiveData* yang merupakan *response* dari *repository* yang telah diinjeksi. *DataInteractor* berperan sebagai perantara antara lapisan pengguna dan lapisan data, penghubungkan masukan pengguna dengan operasi yang dieksekusi pada repository data, dan memberikan hasil berupa *LiveData*.

```
package com.zenith.xyz.domain

...

interface IDataRepository {
    fun uploadPhoto(file: File) : LiveData<Resource<ItemData>>
    fun getAllHistory() : LiveData<List<ItemData>>
    fun getDevicesData(location: String) : LiveData<Resource<List<Device>>>
    fun calculateEnergy(location: String, device: String, deviceType: String)
            : LiveData<Resource<Energy>>
}
```

Gambar 10. *IDataRepository.*

Gambar 10., merupakan *Interface IDataRepository* yang mendefinisikan metode-metode yang dapat digunakan oleh kelas yang bertugas untuk mengelola dan menyediakan data. Kontrak-kontrak ini digunakan sebagai pengatur interaksi dengan data dan *response* pada aplikasi dengan dukungan *LiveData*, yang memudahkan pemantauan dan pembaruan data.

```
package com.zenith.xyz.domain.entities

...

@Parcelize
data class ItemData(
    val lokasi: String? = null,
    val averageEnergy: Double? = null,

    ...

) : Parcelable
```

Gambar 11. *ItemData*.

Gambar 11., merupakan contoh kode yang mendefinisikan sebuah data *class* dalam bahasa *Kotlin* yang diberi anotasi *@Parcelize*. Data *class* merupakan representasi entitas data dengan properti-properti yang sesuai. *ItemData* merupakan data *class* yang memiliki properti yang menjadi perwakilan atribut dari entitas data, seperti lokasi, sumber, dan sebagainya. Anotasi *@Parcelize* merupakan bagian dari *Android's Parcelize framework* mengimplementasikan *Parcelable* secara otomatis pada kelas *ItemData*. Hal ini memudahkan interaksi objek *ItemData* antara komponen *Android*, seperti antara *activity* atau *fragment*. Data *class* dalam *Kotlin* secara memiliki berbagai fungsi seperti *equals*, *toString*, dan lainnya yang terkait dengan data.

3.2.4 Data Layer

Setelah *Refactoring* terdapat beberapa kelas tambahan *repository*, *NetworkDataSource*, *LocalDataSource* pada sumber data.

```
package com.zenith.xyz.data

...

class Repository @Inject constructor(
    private val localDataSource: LocalDataSource,
    private val networkDataSource: NetworkDataSource
) :
    IDataRepository {
    private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO
    override fun uploadPhoto(file: File): LiveData<Resource<ItemData>> = liveData {
        emit(Resource.Loading())
        when (val response =
            networkDataSource.uploadPhoto(file)
        ) {
            is ApiResponse.Success -> {
                val dataEntity = DataMapper.mapItemResponseToEntities(response.data)
                val dataReady = DataMapper.mapItemResponseToDomain(response.data)
                withContext(ioDispatcher) {
                    localDataSource.insertData(dataEntity)
                }
                emit(Resource.Success(dataReady))
            }
            is ApiResponse.Error -> {
                emit(Resource.Error(response.errorMessage))
            }
        }
    }
```

Gambar 12. *Repository* Bagian 1.

Pada Gambar 12., merupakan implementasi kelas *Repository* yang berperan mengolah data dengan menggunakan konsep *Dependency Injection* dengan anotasi *@Inject*. Kelas *Repository* mengimplementasikan kelas *IDataRepository*. Konstruktor kelas *Repository* membutuhkan parameter *LocalDataSource* dan *NetworkDataSource*. *IoDispatcher* yang sebagai penjadwal tugas *input* dan *output*. Metode *uploadPhoto* diimplementasikan dari *interface* dan memberikan nilai kembalian objek *LiveData* yang dibungkus dengan kelas *Resource*, selama proses pengunggahan foto ke server melalui *NetworkDataSource*. Ketika pengunggahan sukses data di-*maps* ke *format* yang sesuai untuk disimpan secara lokal.

Information System Journal (INFOS)
Vol. 7, No. 1, Mei 2024, pp. 20-33
E-ISSN : 2655-142X , P-ISSN : 2655-190X, DOI:https://doi.org/10.24076/infosjournal.2024v7i01.1579

```
override fun getAllHistory(): LiveData<List<ItemData>> = liveData {
        withContext(ioDispatcher) {
            val history = localDataSource.getAllHistory()
            val historyDomain = DataMapper.mapItemEntityToDomain(history)
            emit(historyDomain)
        }
    }
    override fun getDevicesData(location: String): LiveData<Resource<List<Device>>> = liveData {
        when (val response =
            networkDataSource.getDevices(location)
        ) {
            is ApiResponse.Success -> {
                val dataReady = DataMapper.mapListDeviceResponseToDomain(
                    response.data.devices?.filterNotNull()
                        ?: emptyList()
                )
                emit(Resource.Success(dataReady))
            }
            is ApiResponse.Error -> {
                emit(Resource.Error(response.errorMessage))
            }
        }
    }
   ...
}
```

Gambar 13. *Repository* Bagian 2.

Gambar 13., fungsi *getAllHistory* pada kelas *Repository* berfungsi mengembalikan objek *LiveData* yang memancarkan daftar riwayat *ItemData* yang dilakukan secara *asynchronous* menggunakan *Coroutine* dengan penjadwal tugas *input dan output.* Data riwayat diambil dari sumber data lokal menggunakan *localDataSource*, kemudian di-*maps* ke *domain model* menggunakan *DataMapper*. Hasil pemetaan tersebut kemudian dikirim sebagai *LiveData*.

```
package com.zenith.xyz.data.api

...

class NetworkDataSource @Inject constructor(private val apiService: ApiService) {
    suspend fun uploadPhoto(file: File): ApiResponse<ItemResponse> {
        val newFile = reduceFileImage(file)
        val fileReady = newFile.asRequestBody("image/jpg".toMediaType())
        val multipart: MultipartBody.Part = MultipartBody.Part.createFormData(
            "file",
            newFile.name,
            fileReady
        )
        return try {
            val response = apiService.uploadPhoto(multipart)
            ApiResponse.Success(response)
        } catch (e: Exception) {
            ApiResponse.Error(e.message ?: "")
        }
    }
    suspend fun getDevices(location: String): ApiResponse<DeviceResponse> {
        return try {
            val response = apiService.getDevices(location)
            ApiResponse.Success(response)
        } catch (e: Exception) {
            ApiResponse.Error(e.message ?: "")
        }
    }

   ...

}
```

Gambar 14. *NetworkDataSource.*

Gambar 14., merupakan kelas *NetworkDataSource* yang bertugas dalam komunikasi dengan API melalui objek *ApiService*.

```
package com.zenith.xyz.data.local

...

class LocalDataSource @Inject constructor(private val dataDao: DataDao) {
    fun getAllHistory() : List<DataEntity> = dataDao.getAllHistory()
    suspend fun insertData(data: DataEntity) = dataDao.insertData(data)
}
```

Gambar 15. *LocalDataSource.*

Gambar 15., merupakan contoh implementasi dari kelas *Kotlin* yang disebut *LocalDataSource*, yang bertanggung jawab atas akses dan manipulasi data lokal dalam aplikasi dengan menggunakan DAO (*Data Access Object*).

3.2.5 Pengujian

Hasil Pengujian Fungsional pada aplikasi tertera pada Tabel 1.

Tabel 1. Hasil Pengujian.

Id	*Condition*	*Steps*	*Input*	*Expected Result*	*Status*
1	Mengambil data dari *file* yang dipindai	Masukkan file dengan tipe JPG, PNG atau JPEG	*file*:Air Conditioner.jpg	Data Pendingin Ruangan sesuai *Entities*	Berhasil
2	Mendapatkan data *history* ketika terdapat data *history*	Memanggil fungsi untuk mengambil data *history*	-	Mendapatkan Data *History* sesuai *Entities*	Berhasil
3	Mendapatkan *data history* ketika *data history* kosong atau belum pernah memindai	Memanggil fungsi untuk mengambil data history	-	Data Kosong	Berhasil
4	Mendapatkan kumpulan *data device* dan mendapatkan *data*	1. Memasukkan lokasi 2. Memanggil fungsi untuk mengambil *data*	*location*: Dapur	Mendapatkan kumpulan *data device* yang berada di dapur	Berhasil
5	Mendapatkan error ketika *input* lokasi yang tidak tersedia di *database*	1. Memasukkan lokasi 2. Memanggil fungsi untuk mengambil *data*	*location*: Jalan	"*No Data*"	Berhasil
6	Melakukan perhitangan penggunaan energi	1. Memasukkan lokasi 2. Memasukkan *device* 3. Memasukkan *device type*	*location*: Dapur *device*: Oven *devicetype*: *Plastic-Based Oven*	Mendapatkan data *energy_usage* sesuai *entitiy*	Berhasil

3.2.6 Perbandingan *Cyclomatic Complexity*

Cyclomatic Complexity adalah sebuah metrik yang digunakan untuk mengukur kerumitan alur kontrol dari suatu modul dalam perangkat lunak. Apabila nilai *Cyclomatic Complexity* semakin tinggi, maka modul tersebut akan menjadi semakin sulit untuk diuji dan dikelola.

Gambar 16., dan Gambar 17., merupakan hasil pengujian *Cyclomatic Complexity* sebelum dan sesudah *refactoring*. Pengujian dilakukan dengan menggunakan *plugin* CodeMR. Berdasarkan hasil pengujian terdapat penurunan rata-rata *Cyclomatic Complexity* dari 1.5 menjadi 1.4.

Information System Journal (INFOS)
Vol. 7, No. 1, Mei 2024, pp. 20-33
E-ISSN : 2655-142X , P-ISSN : 2655-190X, DOI:https://doi.org/10.24076/infosjournal.2024v7i01.1579

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	FormFragment					109	low-medium	low-medium	medium-high	low-medium
2	CameraFragment					88	low-medium	low-medium	low-medium	low-medium
3	HomeFragment					73	low-medium	low-medium	medium-high	low-medium
4	PreviewFragment					60	low-medium	low-medium	low-medium	low-medium
5	HistoryFragment					29	low-medium	low-medium	low	low
6	SplashFragment					24	low-medium	low-medium	low	low
7	MainActivity					38	medium-high	low	low	low
8	PreviewViewModel					43	low-medium	low	low	low
9	FormViewModel					38	low-medium	low	low	low
10	HistoryAdapter					38	low-medium	low	low-medium	low
11	ItemInfo					31	low-medium	low	medium-high	medium-high
12	DataEntity					19	low-medium	low	medium-high	medium-high
13	DetailFragment					17	low-medium	low	low	low
14	DataDatabase					14	low-medium	low	low	low
15	HistoryCallback					12	low-medium	low	low	low
16	ViewModelFactory					12	low-medium	low	low	low
17	AboutFragment					6	low-medium	low	low	low
18	HistoryViewModel					2	low-medium	low	low	low
19	UtilsKt					75	low	low	medium-high	low-medium

Gambar 16 *Cyclomatic Complexity* sebelum *Refactoring*

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	CameraFragment					94	low-medium	medium-high	low-medium	low-medium
2	PreviewFragment					58	low-medium	medium-high	low-medium	low-medium
3	SplashFragment					25	low-medium	medium-high	low	low
4	AppModule					53	low	medium-high	high	low-medium
5	FormFragment					108	low-medium	low-medium	medium-high	low-medium
6	HomeFragment					76	low-medium	low-medium	medium-high	low-medium
7	DetailFragment					33	low-medium	low-medium	low-medium	low
8	HistoryFragment					28	low-medium	low-medium	low-medium	low
9	Repository					51	low	low-medium	low	low-medium
10	MainActivity					39	medium-high	low	low	low
11	EcoScan					2	medium-high	low	low	low
12	HistoryAdapter					38	low-medium	low	low-medium	low
13	ItemInfo					30	low-medium	low	medium-high	medium-high
14	DataEntity					19	low-medium	low	medium-high	medium-high
15	ItemData					17	low-medium	low	medium-high	medium-high
16	HistoryCallback					10	low-medium	low	low	low
17	FormViewModel					7	low-medium	low	low	low
18	AboutFragment					6	low-medium	low	low	low
19	HistoryViewModel					4	low-medium	low	low	low
20	PreviewViewModel					4	low-medium	low	low	low
21	DataDatabase					3	low-medium	low	low	low
22	UtilsKt					75	low	low	medium-high	low-medium
23	DataMapper					71	low	low	low	low-medium

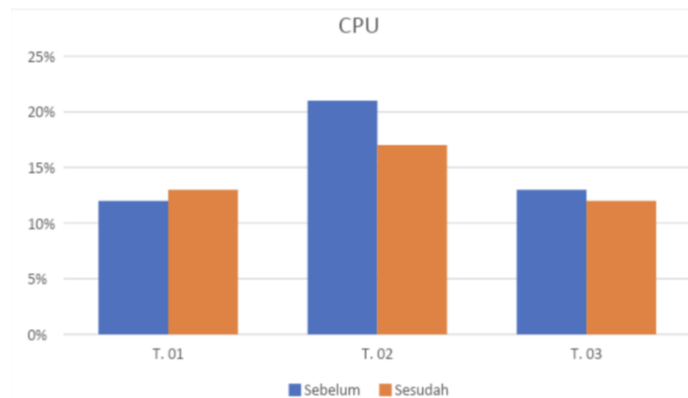
Gambar 17 *Cyclomatic Complexity* setelah *Refactoring*

3.2.7 Performa Aplikasi

Pengujian ini dilakukan untuk memastikan performa aplikasi XYZ sebelum dan sesudah implementasi *Clean Architecture* terkait CPU, Memori, dan waktu Eksekusi menggunakan.

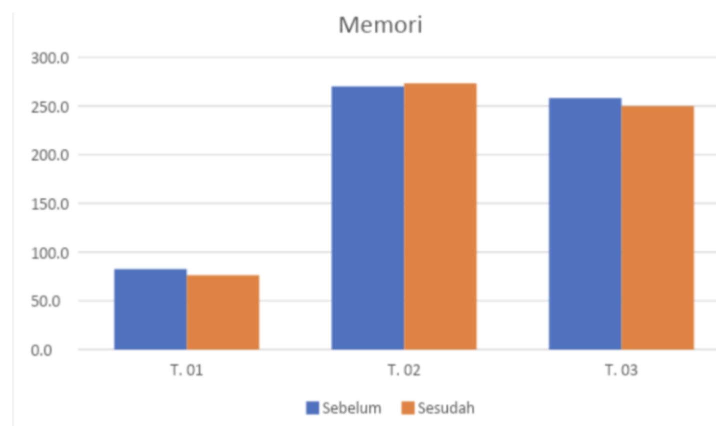
Tabel 2. Kebutuhan Uji Performa.

Test.ID	Deskripsi
T.01	*Splash Screen*
T.02	Pindai Gambar (*Preview Screen* ke *Detail Screen* dengan data)
T.03	Melihat *detail history* yang dipilih



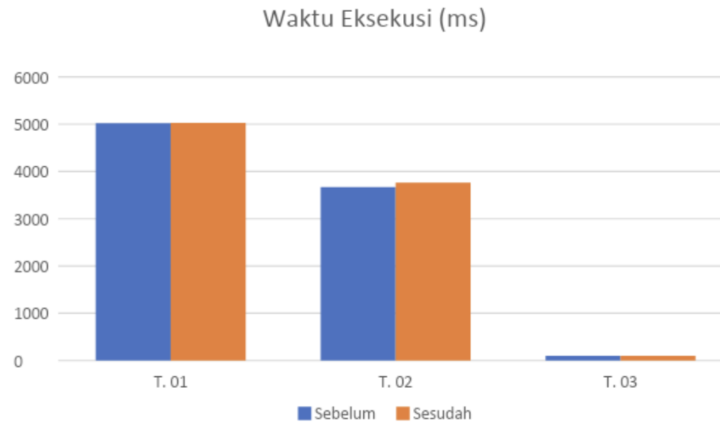
Gambar 18. Diagram Penggunaan CPU.

Pengujian dilakukan di perangkat dengan fitur *Profiler*. *Profiler* akan menunjukan penggunaan CPU dan aktivitas di aplikasi, Pengujian dilakukan dengan menjalankan pengujian yang ada pada Tabel 2 dan dapat dilihat hasilnya pada Gambar 18., dari hasil yang didapat hanya pada pengujian T. 01 sesudah implementasi *Clean Architecture* memiliki penggunaan CPU yang lebih tinggi.



Gambar 19. Diagram Penggunaan Memori.

Pengujian dilakukan di perangkat dengan fitur *Profiler*. *Profiler* akan menunjukan penggunaan Memori dan aktivitas di aplikasi, Pengujian dilakukan dengan menjalankan pengujian yang ada pada Tabel 2 dan dapat dilihat hasilnya pada Gambar 19., dari hasil yang didapat hanya pada pengujian T. 02 sesudah implementasi *Clean Architecture* memiliki penggunaan Memori yang lebih tinggi.



Gambar 20. Diagram Waktu Eksekusi.

Pengujian dilakukan di kode dengan menambahkan kode untuk menghitung waktu eksekusi. Kode tersebut akan menunjukan waktu eksekusi dan aktivitas di aplikasi XYZ, Pengujian dilakukan dengan menjalankan pengujian yang ada pada Tabel 2 dan dapat dilihat hasilnya pada Gambar 20., dari hasil yang didapat hanya pada pengujian T. 03 waktu eksekusi yang dihasilkan bernilai sama setelah dan sebelum implementasi *Clean Architecture*. *Cyclomatic Complexity* mengalami penurunan dari 1.5 menjadi 1.4.

4. KESIMPULAN

Refactoring Clean Architecture pada aplikasi XYZ berhasil dilakukan. *Refactoring* kode dilakukan untuk membagi kode menjadi 3 layer, yaitu data *layer*, *domain layer*, dan *presentation layer*. *Functional testing* yang dilakukan menunjukan 100% keberhasilan. penggunaan CPU setelah implementasi, terjadi perubahan yang cukup signifikan. *Test case* Pindai Gambar dan Melihat *Detail History* menunjukkan penggunaan CPU yang lebih efisien, sementara pada *Splash Screen*, penggunaan CPU lebih tinggi. Adapun pada penggunaan memori, kedua versi aplikasi menunjukkan penggunaan memori yang relatif serupa, tanpa perbedaan yang signifikan. Terakhir, dalam hal *execution time*, terlihat sedikit perbedaan pada *test case* Pindai Gambar setelah implementasi *Clean Architecture* memiliki *execution time* menjadi sedikit lebih lama dibandingkan sebelumnya.

DAFTAR PUSTAKA

[1] O. Sadeghian *et al.*, "A comprehensive review on energy saving options and saving potential in low voltage electricity distribution networks: Building and public lighting," *Sustain Cities Soc*, vol. 72, p. 103064, Sep. 2021, doi: 10.1016/j.scs.2021.103064.
[2] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. USA: Prentice Hall Press, 2017.
[3] D. Bui, "Reactive programming and clean architecture in Android development," 2017.
[4] M. Knill, "Refactoring to clean architecture," *Obtenido de*, 2019.
[5] J. C. Martínez Z, C. Henao, F. Henao, and E. Zapata, "Utilización de Arquitecturas Limpias para Trabajo con Buenas Prácticas en la Construcción de Aplicaciones Java.," *Revista Innovación y Desarrollo Sostenible*, vol. 1, no. 2, pp. 133–140, Feb. 2021, doi: 10.47185/27113760.v1n2.37.
[6] T. Saifulloh, A. P. Kharisma, and D. W. Brata, "Pengembangan Aplikasi Perangkat Bergerak Pencarian Partner Lomba berbasis Android menggunakan Clean Architecture," *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, vol. 7, no. 4, pp. 1549–1559, Mar. 2023.
[7] I. D. Muchlison, A. P. Kharisma, and I. Arwani, "Pengembangan Aplikasi Perangkat Bergerak Sistem Informasi Event di bidang Teknologi Informasi berbasis Android," *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, vol. 6, no. 1, pp. 282–291, Jan. 2022.
[8] Aflah Taqiu Sondha, Umi Sa'adah, Fadilah Fahrul Hardiansyah, and Maulidan Bagus Afridian Rasyid, "Framework dan Code Generator Pengembangan Aplikasi Android dengan Menerapkan Prinsip Clean Architecture," *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, vol. 9, no. 4, pp. 327–335, Dec. 2020, doi: 10.22146/jnteti.v9i4.572.
[9] Moh. H. Badrudduja and R. E. Putra, "Penerapan Clean Architecture pada Aplikasi Pemesanan Makanan menggunakan Metode Slope One Algorithm," *Journal of Informatics and Computer Science (JINACS)*, vol. 3, no. 04, pp. 506–514, Jun. 2022, doi: 10.26740/jinacs.v3n04.p506-514.
[10] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[11] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *Journal of Systems and Software*, vol. 167, p. 110610, Sep. 2020, doi: 10.1016/j.jss.2020.110610.

[12] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, “A large-scale empirical exploration on refactoring activities in open source software projects,” *Sci Comput Program*, vol. 180, pp. 1–15, Jul. 2019, doi: 10.1016/j.scico.2019.05.002.

[13] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[14] D. Fraivert and D. H. Lorenz, “Language Support for Refactorability Decay Prevention,” in *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, New York, NY, USA: ACM, Nov. 2022, pp. 122–134. doi: 10.1145/3564719.3568688.

[15] T. Haendler and G. Neumann, “A Framework for the Assessment and Training of Software Refactoring Competences,” in *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, SCITEPRESS - Science and Technology Publications, 2019, pp. 307–316. doi: 10.5220/0008350803070316.

.