Jurnal Pendidikan dan Teknologi Indonesia (JPTI)
Vol. 5, No. 2, Februari 2025, Hal. 277-287
DOI: https://doi.org/10.52436/1.jpti.648
p-ISSN: 2775-4227
e-ISSN: 2775-4219

Pengaruh Test-Driven Development terhadap Metrik Testabilitas Kode dalam Pengembangan Aplikasi Android

Muhammad Rizki Siraj[*1], Novi Setiani[2]

[1,2]Informatika, Universitas Islam Indonesia
Email: [1]21523278@students.uii.ac.id, [2]novi.setiani@uii.ac.id

Abstrak

Test-Driven Development (TDD) adalah metode pengembangan perangkat lunak yang menulis pengujian sebelum implementasi kode untuk meningkatkan testabilitas. Penelitian ini menganalisis dampak TDD terhadap testabilitas kode menggunakan studi kasus aplikasi Android Sedonor. Pengukuran dilakukan sebelum dan sesudah penerapan TDD dengan metrik Response for Class (RFC), Lines of Code (LOC), Lines of Comment (LOCCOM), Weighted Method per Class(WMC), dan Number of Method Calls (NMC).
Hasil menunjukkan bahwa TDD secara signifikan meningkatkan testabilitas. RFC menurun hingga 86,7%, LOC berkurang hingga 91,9%, dan NMC berkurang lebih dari 75%, menunjukkan kode menjadi lebih modular dan efisien. TDD membuat kode lebih mudah diuji, dipelihara, dan siap untuk pengembangan berkelanjutan.

Kata kunci: aplikasi Android, modularitas, pengembangan perangkat lunak, testabilitas kode, Test-Driven Development.

The Impact of Test-Driven Development on Code Testability Metrics in Android Application Development

Abstract

Test-Driven Development (TDD) is a software development method that writes tests before code implementation to improve testability. This study analyzes the impact of TDD on code testability using a case study of the Android application Sedonor. Measurements were conducted before and after TDD implementation using the metrics Response for Class (RFC), Lines of Code (LOC), Lines of Comment (LOCCOM), Weighted Method per Class (WMC), and Number of Method Calls (NMC).
The results show that TDD significantly enhances testability. RFC decreased by up to 86.7%, LOC reduced by up to 91.9%, and NMC dropped by more than 75%, indicating that the code became more modular and efficient. TDD makes the code easier to test, maintain, and prepare for sustainable development.

Keywords: *Android application, code testability, modularity, software development, Test-Driven Development.*

1. PENDAHULUAN

Banyak perangkat lunak yang memiliki tingkat testabilitas rendah, sehingga menyulitkan proses deteksi bug dan pemeliharaan. Testabilitas rendah menyebabkan pengujian menjadi lebih kompleks, memakan waktu lebih lama, dan meningkatkan biaya pengembangan. Kondisi ini menunjukkan pentingnya pengembangan perangkat lunak dengan memperhatikan aspek testabilitas sejak awal. Meskipun Test-Driven Development (TDD) sering diklaim sebagai pendekatan yang efektif untuk meningkatkan testabilitas, studi empiris yang menguji dampaknya secara kuantitatif menggunakan metrik tertentu masih terbatas. Di sisi lain, pendekatan alternatif seperti *testability refactoring* juga menawarkan solusi untuk meningkatkan testabilitas dengan memperbaiki struktur kode tanpa mengubah perilakunya [1].

Pengujian perangkat lunak sendiri adalah elemen krusial dalam pengembangan untuk memastikan kualitas sistem. Proses ini mencakup pemeriksaan berbagai artefak, seperti persyaratan dan modul, guna memastikan perangkat lunak berfungsi dengan baik dan minim kesalahan . Dalam konteks ini, testabilitas memainkan peran penting sebagai ukuran sejauh mana perangkat lunak mendukung proses pengujian. Semakin tinggi testabilitas, semakin mudah mendeteksi kesalahan. Sebaliknya, testabilitas rendah meningkatkan kompleksitas pengujian, membutuhkan lebih banyak waktu, dan menambah beban biaya [2].

Beberapa faktor utama yang memengaruhi testabilitas meliputi kompleksitas kode, modularitas, dan *coupling*. Penelitian Zhou dkk. [3] menunjukkan bahwa kode dengan kompleksitas tinggi memerlukan upaya pengujian yang lebih besar, sehingga meningkatkan biaya pengujian [4] . Kode yang modular diketahui dapat mengurangi beban pengujian hingga 50% [5]. Namun, tingkat *coupling* yang tinggi antar unit kode seringkali

p-ISSN: 2775-4227
e-ISSN: 2775-4219

menjadi hambatan besar dalam pembuatan *test stub*, yang pada akhirnya menurunkan testabilitas [4]. Dalam konteks ini, penelitian Reich & Maalej menunjukkan bahwa *testability refactoring*, seperti *extract method for override* atau *widen access for invocation*, dapat meningkatkan testabilitas kode secara signifikan tanpa menambah kompleksitas desain [1].

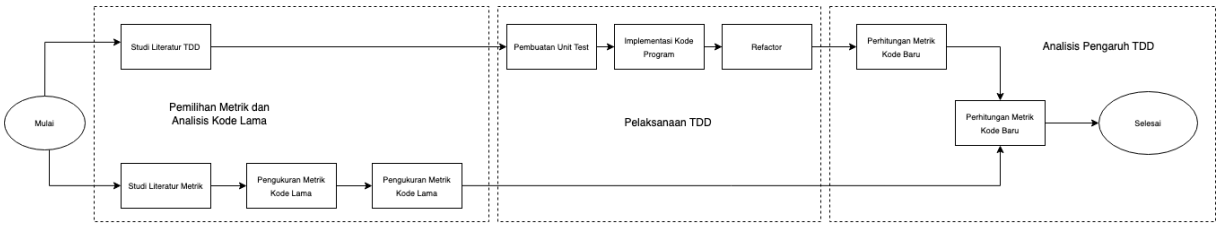
Salah satu pendekatan efektif untuk meningkatkan testabilitas adalah TDD. Dalam TDD, pengembang diwajibkan menuliskan pengujian sebelum mengembangkan kode. Ini memastikan setiap bagian kode memiliki cakupan pengujian yang baik sekaligus mendorong praktik penulisan kode yang lebih berkualitas [6]. Dengan TDD, pengembang dapat mendeteksi kesalahan lebih awal, serta mendorong desain kode yang lebih modular dan terstruktur [7]. Hal ini berdampak positif pada meningkatnya testabilitas karena kode dirancang sejak awal dengan prinsip test-first.

Menurut penelitian yang dilakukan oleh Muhammad Iqbal Naufal Ilmi, Aminudin, dan Zamah Sari, penerapan TDD dapat meningkatkan code coverage lebih dari 100% dan mengurangi cyclomatic complexity sebesar 15,78% [8]. Penelitian ini dilakukan pada platform berbasis web, yang menunjukkan hasil positif terkait kualitas kode. Di sisi lain, dalam penelitian yang dilakukan oleh Ayush Vijaywargi dan Uchinta Kumar Boddapati, yang menggunakan wawancara dan studi literatur, ditemukan bahwa penerapan TDD dalam pengembangan aplikasi Android juga terbukti meningkatkan kualitas kode, terutama dengan deteksi bug yang lebih dini. Berdasarkan hasil dari kedua penelitian ini, terdapat peluang untuk melakukan penelitian lebih lanjut mengenai dampak TDD terhadap testabilitas kode, khususnya dengan mengkaji metode pengembangan ulang atau praktik penerapannya dalam pengembangan perangkat lunak Android [9].

Penelitian ini bertujuan untuk mengukur dampak TDD terhadap testabilitas kode menggunakan metrik seperti Response for Class (RFC), Lines of Code (LOC), Lines of Comment (LOCCOM), Weighted Method per Class (WMC), dan Number of Method Calls (NMC). Studi ini juga membandingkan kualitas kode sebelum dan sesudah penerapan TDD pada aplikasi Android Sedonor, dengan harapan dapat memberikan wawasan yang lebih jelas mengenai efektivitas TDD dalam meningkatkan testabilitas kode [10].

2. METODE PENELITIAN

Metodologi penelitian yang digunakan terdiri dari tiga tahap utama: pemilihan metrik dan analisis kode lama, pengembangan ulang dengan pendekatan Test-Driven Development (TDD), dan analisis perbandingan metrik untuk mengevaluasi pengaruh TDD. Seperti yang dapat dilihat di Gambar 1.



Gambar 1. Diagram Alur Penelitian

2.1. Pemilihan Metrik dan Analisis Kode Lama

2.1.1. Studi Literatur Metrik

Studi literatur digunakan untuk mengidentifikasi metrik yang relevan untuk mengukur testabilitas kode. Berdasarkan penelitian Alenezi sebelumnya, metrik yang dipilih adalah:

1. **Response for Class (RFC):** Mengukur kompleksitas pemanggilan fungsi dalam suatu kelas.
2. **Lines of Code (LOC):** Mengukur ukuran kode secara keseluruhan.
3. **Lines of Comment (LOCCOM):** Mengukur jumlah komentar dalam kode.
4. **Weighted Methods per Class (WMC):** Menilai kompleksitas kelas berdasarkan bobot metode yang ada.
5. **Number of Method Calls (NMC):** Menghitung jumlah pemanggilan metode.

Metrik-metrik ini digunakan karena mampu mencerminkan modularitas, keterbacaan, dan kompleksitas kode, yang merupakan faktor utama dalam testabilitas perangkat lunak [10].

p-ISSN: 2775-4227
e-ISSN: 2775-4219

2.1.2. Pengukuran Metrik Kode Lama

Pengukuran dilakukan pada 7 file kode sumber dari aplikasi Sedonor menggunakan library CK [11]. CK dipilih karena mendukung analisis statis terhadap berbagai metrik kompleksitas kode. Karena CK hanya mendukung bahasa Java, kode aplikasi yang awalnya ditulis dalam Kotlin dikonversi ke Java tanpa mengubah fungsionalitasnya. Pengukuran ini memberikan data awal untuk membandingkan efektivitas metode TDD terhadap testabilitas kode.

Data dikategorikan menjadi dua kelompok:

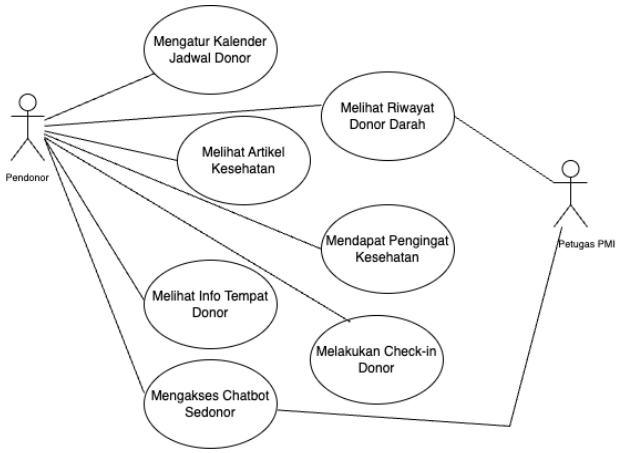
1. Kode lama (sebelum TDD)
2. Kode baru (setelah TDD)

Data dikumpulkan melalui analisis statis menggunakan CK [11], dengan masing-masing metrik dihitung dan disimpan untuk keperluan perbandingan.

2.2. Pelaksanaan TDD

Pelaksanaan TDD diterapkan pada aplikasi Sedonor, aplikasi donor darah berbasis Android yang dikembangkan menggunakan Kotlin. Aplikasi ini memiliki tujuh fitur utama yang melibatkan dua aktor, yaitu pendonor dan petugas PMI, dengan peran berbeda sesuai proses bisnis. Sedonor dipilih sebagai studi kasus karena pengembangannya sebelumnya tidak menggunakan metode terstruktur, sehingga dilakukan pengembangan ulang menggunakan TDD untuk meningkatkan testabilitas kode.

Diagram use case aplikasi Sedonor dapat dilihat pada Gambar 1 berikut:



Gambar 2. Diagram usecase Sedonor

2.2.1. Pembuatan Unit Test

Tahap awal TDD adalah pembuatan unit test menggunakan framework JUnit dan Mockk. Unit test dirancang berdasarkan kebutuhan fitur yang diidentifikasi dari diagram use case aplikasi Sedonor. Untuk setiap fitur utama, minimal 3 unit test dibuat untuk memastikan pengujian mendalam terhadap fungsionalitas spesifik secara independen.

Pendekatan ini memungkinkan pengujian fungsionalitas spesifik secara independen dan memastikan bahwa pengembangan fitur dilakukan secara bertahap dengan pengujian mendalam [9].

Selain itu, pentingnya unit testing dalam pengembangan perangkat lunak sangat ditekankan, karena unit test memungkinkan pengembang untuk mendeteksi kesalahan lebih awal, memastikan bahwa perubahan kode tidak merusak fungsionalitas yang sudah ada, dan meningkatkan kualitas perangkat lunak secara keseluruhan dalam lingkungan pengembangan yang iteratif dan cepat [12].

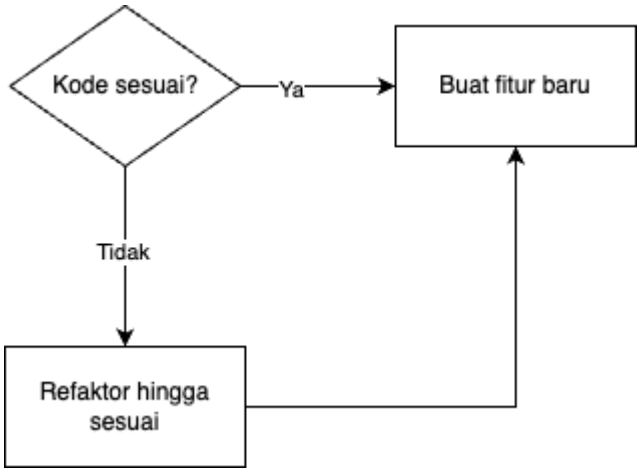
2.2.1. Implementasi Kode Program

Pada tahap ini, kode ditulis untuk memenuhi semua unit test yang telah dirancang sebelumnya. Pendekatan ini berfokus pada implementasi fungsionalitas dasar agar semua tes berhasil lulus, sesuai dengan fase "Green" dalam siklus TDD. Proses ini menghasilkan kode yang berfungsi tetapi belum terstruktur secara optimal [9]. Oleh karena itu, langkah selanjutnya adalah melakukan refactoring untuk meningkatkan kualitas kode.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

2.2.1. Refactoring

Refactoring dilakukan untuk memastikan bahwa kode memenuhi prinsip clean coding, modularitas, dan keterbacaan yang baik. Tahap ini dilakukan secara iteratif, di mana kode yang telah diuji ditinjau dan diperbaiki sebelum melanjutkan ke pengembangan fitur berikutnya. Proses refactoring ini mengacu pada alur yang ditunjukkan dalam Gambar 3.



Gambar 3. Diagram refactoring

Refactoring sangat penting karena dapat meningkatkan kualitas kode tanpa mengubah perilaku eksternal program [13]. Selain itu, refactoring juga membantu menghilangkan *test smells* yang dapat memengaruhi keterbacaan, pemeliharaan, dan efektivitas pengujian [13]. Namun, penelitian menunjukkan bahwa sebagian besar penghapusan test smells bukan hasil dari refactoring yang disengaja, melainkan terjadi sebagai bagian dari pemeliharaan fitur. Oleh karena itu, penerapan refactoring secara sistematis sangat disarankan agar perbaikan kode dilakukan secara optimal dan tidak menyebabkan regresi atau kerentanan baru.

Refactoring juga melibatkan kombinasi analisis manual melalui code review dan penggunaan alat bantu seperti SonarLint, yang memberikan rekomendasi untuk meningkatkan kualitas kode berdasarkan prinsip Clean Architecture [14]. Pendekatan ini tidak hanya memastikan fungsionalitas kode tetap terjaga, tetapi juga meningkatkan efisiensi, keterbacaan, dan testabilitas kode secara keseluruhan.

2.3. Analisis Pengaruh TDD

Analisis dilakukan untuk mengevaluasi pengaruh penerapan metode TDD terhadap kualitas kode aplikasi Sedonor. Proses ini mencakup perhitungan metrik kode baru dan perbandingan dengan kode lama.

2.3.1 Perhitungan Metrik Kode Baru

Penghitungan metrik kode baru dilakukan dengan menggunakan library CK yang sama seperti pada pengukuran kode lama. Penggunaan metrik seperti Response for Class (RFC), Lines of Code (LOC), Lines of Comment (LOCCOM), Weighted Methods per Class (WMC), dan Number of Method Calls (NMC) memastikan konsistensi evaluasi. Hasil pengukuran kode baru digunakan untuk menilai dampak TDD terhadap testabilitas, kompleksitas, dan keterbacaan kode secara objektif.

2.3.2 Perbandingan Hasil Metrik Kode Lama dan Baru

Hasil pengukuran metrik kode lama dan kode baru dibandingkan untuk mengidentifikasi perubahan kualitas kode setelah penerapan TDD. Perbandingan ini mengukur peningkatan pada aspek testabilitas, modularitas, dan keterbacaan kode melalui enam metrik utama (NBI, RFC, LOC, LOCCOM, WMC, dan NMC). Analisis ini memberikan wawasan kuantitatif yang menunjukkan efektivitas refactoring dan pengaruh TDD terhadap pengembangan kode yang lebih terstruktur dan efisien.

3. HASIL DAN PEMBAHASAN

Pembahasan pada bab ini akan dibagi menjadi lima bagian yaitu Hasil Pengukuran Metrik Lama, Hasil TDD, Contoh Pembuatan Unit Test, Implementasi Kode Program, Perhitungan Metrik Kode Baru. Hasil dari

p-ISSN: 2775-4227
e-ISSN: 2775-4219

nilai yang diperoleh akan ditampilkan dalam bentuk tabel untuk nantinya dibandingkan antara TDD dengan Non-TDD.

3.1. Hasil Pengukuran Metrik Kode Lama

Tahapan ini adalah tahap di mana kode lama yang tidak menerapkan metode TDD akan diukur. Tahap pengukuran metrik terhadap kode lama dilakukan untuk mengetahui nilai testabilitas kode yang tidak menggunakan TDD. Metrik yang digunakan untuk pengukuran adalah Response for Class (RFC), Lines of Code (LOC), Lines of Comment (LOCCOM), Weighted Methods per Class (WMC), dan Number of Method Calls (NMC) yang terbukti paling ideal untuk mengukur testabilitas kode [10]. Pengukuran dilakukan menggunakan library CK [11]. Pada Tabel 1 ditunjukkan nilai dari hasil pengukuran metrik testabilitas terhadap kode lama.

Tabel 1. Hasil Pengukuran Metrik Testabilitas pada Kode Lama

Metrik	Hasil Pengukuran Lihat Artikel	Hasil Pengukuran Lihat Lokasi
RFC	31	68
LOC	37	17
LOCCOM	37	19
NMC	194	176
WMC	20	23

Berdasarkan Tabel 1, beberapa metrik testabilitas menunjukkan kondisi kode yang masih cukup sederhana, tetapi tetap memiliki potensi untuk ditingkatkan. Nilai Response for Class (RFC) sebesar 31 (artikel) dan 68 (lokasi) menunjukkan bahwa kelas memiliki interaksi yang cukup kompleks dengan metode lain. Walaupun nilai ini belum terlalu tinggi dan masih di bawah ambang batas kurang dari 100, namun bisa dikurangi lebih lanjut dengan mengurangi dependensi antar metode atau membagi tanggung jawab kelas menjadi lebih modular [15]. Jika kode nantinya berkembang dan semakin banyak metode yang berinteraksi satu sama lain, nilai RFC bisa meningkat drastis, terutama ketika mulai berhubungan dengan kode antarmuka pengguna.

Nilai Lines of Code (LOC) yang berada di angka 37 (artikel) dan 17 (lokasi) menunjukkan bahwa ukuran kelas masih tergolong kecil. Meskipun demikian, jumlah baris kode masih bisa dikurangi lebih lanjut dengan menghindari kode berulang dan menerapkan prinsip refactoring. Jika kode nantinya berkembang dan mulai menangani lebih banyak fitur, termasuk integrasi dengan UI, nilai LOC berpotensi meningkat secara signifikan.

Pada metrik Lines of Comment (LOCCOM), nilai yang diperoleh sebesar 37 (artikel) dan 19 (lokasi) menunjukkan tingkat dokumentasi yang cukup, tetapi masih dapat diperbaiki dengan menambahkan komentar yang lebih informatif pada bagian-bagian yang kompleks. Idealnya, dokumentasi dalam kode mencakup sekitar 20–30% dari total baris kode agar pengembang lain lebih mudah memahami struktur dan alur program [15].

Sementara itu, Number of Method Calls (NMC) memiliki nilai 194 (artikel) dan 176 (lokasi), yang menandakan bahwa kelas memiliki cukup banyak pemanggilan metode. Angka ini bisa meningkat jika kelas semakin besar dan kompleks, terutama jika metode dalam kelas semakin sering berinteraksi dengan metode lain dalam sistem. Walaupun saat ini nilai NMC masih dalam batas wajar, akan lebih baik jika pemanggilan metode dapat diminimalkan untuk menjaga efisiensi dan mengurangi ketergantungan berlebih.

Terakhir, nilai Weighted Methods per Class (WMC) sebesar 20 (artikel) dan 23 (lokasi) menunjukkan bahwa kompleksitas metode dalam kelas masih cukup terkendali [15]. Namun, kompleksitas ini bisa meningkat seiring bertambahnya fitur dan logika dalam kode, terutama jika kode mulai menangani komponen UI yang memiliki banyak interaksi dengan pengguna. Oleh karena itu, mempertahankan struktur kode yang modular sejak awal akan membantu mencegah kenaikan nilai WMC yang tidak terkendali di masa mendatang.

Meskipun nilai-nilai metrik testabilitas saat ini masih tergolong rendah, optimalisasi lebih lanjut tetap diperlukan untuk memastikan kode tetap mudah diuji dan dipelihara. Mengingat ini masih kode dasar yang belum berhubungan dengan komponen UI, kemungkinan besar nilai-nilai ini akan meningkat saat kode berkembang lebih lanjut.

3.2. Pelaksanaan TDD

Tahap ini menjelaskan proses pelaksanaan Test-Driven Development (TDD) yang dilakukan dalam pengembangan perangkat lunak ini. TDD diterapkan melalui siklus iteratif yang terdiri dari tiga langkah utama,

p-ISSN: 2775-4227
e-ISSN: 2775-4219

yaitu menulis unit test, mengimplementasikan kode untuk memenuhi test case tersebut, dan melakukan refactoring pada kode agar lebih efisien tanpa mengubah fungsionalitasnya.

3.2.1. Pembuatan Unit Test

Berdasarkan use case yang telah dibuat, maka dirancang beberapa unit test untuk pengembangan ulang aplikasi Sedonor. Unit test dibuat menggunakan bahasa Kotlin dengan bantuan library JUnit sebagai otomasi tes. JUnit digunakan karena kemampuannya dalam menjalankan pengujian secara otomatis dan memberikan hasil evaluasi yang cepat, sehingga mempermudah proses verifikasi fungsionalitas kode. Adapun contoh *unit test* yang ditulis, ditunjukkan sebagai berikut:

```
@Test
fun `fetch articles successfully from firestore`() = runBlocking {
    val article = Article("Test Title", "Test Content")
    every { mockFirestore.collection("articles") } returns mockCollectionReference
    every { mockCollectionReference.get() } returns mockTask

    mockkStatic("kotlinx.coroutines.tasks.TasksKt")
    coEvery { mockTask.await() } returns mockQuerySnapshot

    every { mockQuerySnapshot.toObjects(Article::class.java) } returns listOf(article)

    val result = repository.fetchArticles()

    assertTrue(result.isSuccess)
    assertEquals(listOf(article), result.getOrNull())
}
```

Fungsi pengujian di atas dirancang untuk menguji apakah proses pengambilan data artikel dari Firestore berhasil dilakukan. Pengujian menggunakan mock dari *library* MockK untuk mensimulasikan perilaku Firestore, sehingga proses pengujian dapat dilakukan tanpa tergantung pada implementasi atau data Firestore yang sebenarnya. Pada tahap ini, pengujian diatur untuk memverifikasi bahwa fungsi *fetchArticles()* dapat mengembalikan daftar artikel dengan benar.

Namun, karena fungsi inti *fetchArticles()* belum diimplementasikan, hasil pengujian ini diharapkan gagal (merah). Hal ini sesuai dengan prinsip TDD, di mana pengujian ditulis terlebih dahulu untuk memastikan bahwa fungsionalitas yang akan dikembangkan telah ditentukan dengan jelas sebelum kode implementasi ditulis. Berikut adalah tangkapan layar hasil pengujian awal yang menunjukkan status gagal (merah), sesuai ekspektasi:

Gambar 4. Hasil Tes Gagal ArticleRepository

Gambar tersebut menunjukkan bahwa pengujian memberikan hasil gagal karena fungsi utama belum diimplementasikan, menandai langkah pertama dalam siklus TDD untuk pengembangan fitur ini.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

3.2.2. Implementasi Kode Program

Setelah unit test ditulis dan menghasilkan hasil gagal (merah), langkah berikutnya adalah mengimplementasikan kode program untuk memenuhi pengujian tersebut. Proses implementasi dilakukan dengan menulis fungsi dasar untuk menyelesaikan satu test case terlebih dahulu. Pada contoh kasus ini, fungsi fetchArticles() diimplementasikan untuk mengambil data artikel dari Firestore.

Berikut adalah implementasi kode awal untuk fungsi *fetchArticles()*:

```
class ArticleRepository(private val firestore: FirebaseFirestore) {
    suspend fun fetchArticles(): Result<List<Article>> {
        return try {
            var collection = firestore.collection("articles")
            var snapshot: QuerySnapshot = collection.get().await()
            Result.success(snapshot.toObjects(Article::class.java))
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Kode di atas memiliki fungsi untuk mengambil data artikel dari Firestore dan mengembalikan objek Result yang berisi daftar artikel jika proses berhasil, atau objek Result yang berisi informasi kesalahan jika terjadi error selama proses pengambilan data. Fungsi ini menggunakan pendekatan berbasis Coroutine dengan *suspend* untuk memastikan operasi berjalan secara asinkron tanpa mengganggu alur utama aplikasi.

Fungsi ini harusnya memenuhi tes yang telah ditulis sebelumnya, yaitu mengembalikan daftar artikel ketika data berhasil diambil dari Firestore. Untuk memverifikasi hal ini, unit test yang telah dibuat dijalankan kembali, dan hasilnya menunjukkan bahwa fungsi ini berhasil memenuhi semua skenario pengujian.

Gambar 5. Hasil Tes Berhasil ArticleRepository

Hasil ini membuktikan bahwa fungsi *fetchArticles()* telah berjalan sesuai dengan spesifikasi yang ditentukan. Oleh karena itu, langkah selanjutnya adalah melakukan *refactoring* untuk menyempurnakan kode sesuai prinsip *clean coding*, sehingga modularitas dan keterbacaan kode dapat lebih ditingkatkan [6].

3.2.3 Refactor

Setelah fungsi *fetchArticles()* berhasil melewati pengujian, langkah selanjutnya adalah melakukan *refactoring* untuk meningkatkan kualitas kode. Proses *refactoring* dilakukan dengan menggunakan alat bantu seperti SonarLint dan *tool* dari IntelliJ IDEA untuk mendeteksi peluang perbaikan pada kode. Alat-alat ini membantu mengidentifikasi potensi masalah, seperti kode yang terlalu kompleks, duplikasi, atau pelanggaran terhadap kaidah *clean coding*.

Sebagai hasil dari refactoring, beberapa perubahan dilakukan untuk meningkatkan modularitas dan keterbacaan kode. Berikut adalah contoh perbaikan yang diterapkan:

a. **Kode sebelum di-*refactor***

```
suspend fun fetchArticles(): Result<List<Article>> {
    return try {
        var collection = firestore.collection("articles")
        var snapshot: QuerySnapshot = collection.get().await()
        Result.success(snapshot.toObjects(Article::class.java))
    } catch (e: Exception) {
        Result.failure(e)
```

p-ISSN: 2775-4227
e-ISSN: 2775-4219

```
    }
}
```

b. **Kode setelah di-*refactor***

```
suspend fun fetchArticles(): Result<List<Article>> {
    return try {
        val snapshot: QuerySnapshot = firestore.collection("articles").get().await()
        Result.success(snapshot.toObjects(Article::class.java))
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

Pada kode hasil *refactor*, variabel sementara *collection* dihapus, dan alur pengambilan data langsung diterapkan dalam satu proses. Selain itu, variabel *snapshot* tetap digunakan namun dideklarasikan dengan kata kunci *val* untuk meningkatkan keamanan dengan memastikan nilainya tidak dapat diubah setelah inisialisasi.

Proses *refactoring* dilakukan secara iteratif dan dicatat melalui beberapa kali *commit* hingga SonarLint dan IntelliJ IDEA tidak lagi mendeteksi potensi perbaikan atau celah dalam kode. Dalam proyek ini, proses *refactoring* tercatat sebanyak 9 kali commit, dengan fokus pada penyederhanaan kode, penghapusan elemen yang tidak perlu, dan penerapan prinsip *clean coding*. Berikut adalah tabel yang merangkum setiap iterasi *refactoring* yang dilakukan:

Tabel 2. Hasil Pengukuran Metrik Testabilitas pada Kode Lama

Fitur	File	Jumlah Testcase	Jumlah Refaktor	Rentang Waktu
Lihat Artikel	ArticleRepository.kt	2	1	Nov 1, 2024 – Nov 10, 2024
	ArticleViewModel.kt	4	3	Nov 1, 2024 – Nov 10, 2024
Lihat Lokasi	LocationRepository.kt	2	1	Nov 6. 2024 – Nov 10, 2024
	LocationViewModel.kt	4	3	Nov 6. 2024 – Nov 10, 2024

3.3. Perhitungan Metrik Kode Baru dan Perbandingan Dengan Kode Lama

Setelah proses implementasi dan refactoring selesai, metrik kode dihitung kembali menggunakan metode yang sama seperti sebelumnya. Hasil pengukuran menunjukkan adanya peningkatan pada aspek testability, dengan pengurangan jumlah baris kode dan perbaikan struktur yang membuat kode lebih mudah diuji dan dipelihara dibandingkan versi sebelumnya. Berikut adalah perbandingan metrik sebelum dan sesudah refactoring:

Tabel 2. Hasil Pengukuran Metrik Testabilitas pada Kode Lama

Metrik	Hasil Pengukuran Lihat Artikel (Sebelum TDD)		Hasil Pengukuran Lihat Lokasi (Sebelum TDD)	
	Sebelum TDD	**Sesudah TDD**	Sebelum TDD	Sesudah TDD
RFC	31	9	68	9
LOC	37	3	17	3
LOCCOM	37	14	19	14
NMC	194	41	176	41
WMC	20	8	23	8

Tabel 2 menunjukkan perbandingan metrik testabilitas kode sebelum dan sesudah penerapan Test-Driven Development (TDD). Terlihat bahwa penerapan TDD berhasil meningkatkan kualitas testabilitas kode secara signifikan dibandingkan dengan kondisi sebelumnya. Sebelum TDD, nilai Response for Class (RFC) untuk fitur Lihat Artikel dan Lihat Lokasimasing-masing adalah 31 dan 68. Nilai ini menunjukkan bahwa kelas memiliki tanggung jawab yang cukup besar dan banyak bergantung pada metode lain, yang dapat meningkatkan kompleksitas kode. Setelah penerapan TDD, nilai RFC turun drastis menjadi 9 untuk kedua fitur, yang berarti terjadi penyederhanaan dependensi antar metode serta peningkatan modularitas. Penurunan ini mencapai 70,9% untuk Lihat Artikel dan 86,7% untuk Lihat Lokasi, menunjukkan bahwa kode menjadi lebih terstruktur dan mudah diuji [15].

Selain itu, jumlah Lines of Code (LOC) juga mengalami perubahan signifikan. Sebelum TDD, LOC untuk Lihat Artikeldan Lihat Lokasi masing-masing adalah 37 dan 17, tetapi setelah TDD, angka ini turun drastis

p-ISSN: 2775-4227
e-ISSN: 2775-4219

menjadi hanya 3 untuk kedua fitur. Penurunan sebesar 91,9% dan 82,3% ini menunjukkan bahwa implementasi TDD mendorong pembuatan kode yang lebih ringkas dan efisien, menghilangkan redundansi serta meningkatkan keterbacaan. Meskipun jumlah baris kode berkurang, Rasio Lines of Comment (LOCCOM) justru meningkat, dari 37 dan 19 sebelum TDD menjadi 14 untuk keduanya setelah TDD. Hal ini mengindikasikan bahwa meskipun kode menjadi lebih singkat, dokumentasi tetap dipertahankan atau bahkan diperbaiki, yang membuatnya lebih mudah dipahami oleh pengembang lain.

Dari sisi jumlah metode per kelas (Number of Method Calls, NMC), terjadi pengurangan drastis dari 194 menjadi 41 pada Lihat Artikel dan dari 176 menjadi 41 pada Lihat Lokasi. Penurunan ini mencapai 78,8% dan 76,7%, menunjukkan bahwa metode dalam kelas menjadi lebih spesifik dan memiliki tanggung jawab yang lebih jelas. Selain itu, nilai Weighted Methods per Class (WMC) juga mengalami penurunan dari 20 dan 23 sebelum TDD menjadi 8 untuk kedua fitur setelah TDD, yang berarti kompleksitas metode telah diminimalkan, sehingga proses pengujian dan pemeliharaan kode menjadi lebih mudah.

Hasil ini sejalan dengan penelitian yang dilakukan oleh Naufal dkk, yang menemukan bahwa penerapan TDD meningkatkan testabilitas kode dengan mengurangi NMC sebesar 50%. Namun, dalam penelitian ini, penurunan NMC mencapai lebih dari 75%, yang menunjukkan bahwa pendekatan TDD yang diterapkan memberikan dampak yang lebih besar terhadap modularitas dan keterpisahan fungsi dalam kode terutama pada aplikasi android. Perbedaan ini kemungkinan disebabkan oleh strategi *refactoring* yang lebih agresif serta penggunaan pengujian berbasis skenario yang lebih ketat.

Hasil pengukuran ini menunjukkan bahwa penerapan TDD secara efektif mengurangi kompleksitas dan meningkatkan kualitas testabilitas kode, menjadikannya lebih sederhana, modular, dan mudah diuji.

Selain meningkatkan kualitas testabilitas melalui metrik, penerapan TDD juga memberikan dampak nyata pada struktur kode. Sebelum TDD, kode untuk mengambil artikel dari Firestore ditulis secara langsung dengan logika yang tercampur dalam satu fungsi besar, sehingga sulit dipahami, diuji, dan dipelihara. Berikut adalah contoh kode sebelum penerapan TDD:

```
db.collection("artikels")
        .get()
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                val document = task.result.documents
                // Konversi data Firestore ke list Artikel
                val artikelList = convertQuerySnapshotToList(task.result)

                // Inisialisasi dan atur adapter
                myAdapter = MyAdapter(this, artikelList as ArrayList<Artikel>)

                // Set listener untuk perpindahan halaman
                myAdapter.setOnItemClickListener(object : MyAdapter.OnItemClickListener {
                    override fun onItemClick(position: Int) {
                        // Dapatkan data artikel pada posisi yang diklik
                        val clickedArtikel = artikelList[position]

                        // Persiapkan intent untuk perpindahan halaman
                        val intent = Intent(this@ArtikelPage, DetailArtikel::class.java)
                        intent.putExtra("judul", clickedArtikel.judul)
                        intent.putExtra("konten", clickedArtikel.konten)
                        intent.putExtra("imageUrl", clickedArtikel.imageUrl)

//                        Log.w("imageUrl", clickedArtikel.imageUrl)
                        // Mulai aktivitas DetailArtikel
                        startActivity(intent)
                    }
                })

                // Set adapter ke RecyclerView
                recyclerView.adapter = myAdapter
            } else {
                // Handle kegagalan
```

p-ISSN: 2775-4227
e-ISSN: 2775-4219

```
                Log.e("Firestore", "Error getting documents.", task.exception)
            }
        }
```

Selain meningkatkan kualitas testabilitas melalui metrik, penerapan TDD juga memberikan dampak nyata pada struktur kode. Sebelum TDD, kode untuk mengambil artikel dari Firestore ditulis secara langsung dengan logika yang tercampur dalam satu fungsi besar, sehingga sulit dipahami, diuji, dan dipelihara. Berikut adalah contoh kode sebelum penerapan TDD:

Pada kode di atas, jika Firestore diganti dengan layanan database lain, pengembang harus memodifikasi setiap fungsi yang menggunakan Firestore secara langsung, yang dapat menyebabkan kesalahan dan biaya pengembangan yang tinggi.

Setelah menerapkan TDD, pendekatan baru menggunakan lapisan abstraksi melalui kelas ArticleRepository, sehingga perubahan di layanan database hanya perlu dilakukan pada kelas ini tanpa memengaruhi bagian kode lainnya. Contoh pendekatan ini adalah sebagai berikut:

```
class ArticleRepository(private val firestore: FirebaseFirestore) {
    suspend fun fetchArticles(): Result<List<Article>> {
        return try {
            val snapshot: QuerySnapshot = firestore.collection("articles").get().await()
            Result.success(snapshot.toObjects(Article::class.java))
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
}
```

Kode hasil TDD lebih sederhana, terstruktur, dan fleksibel untuk menghadapi perubahan di masa mendatang dibandingkan dengan pendekatan awal yang bercampur dan sulit diadaptasi. Hal ini memastikan kode lebih siap untuk pengembangan berkelanjutan dan pemeliharaan jangka panjang.

4. KESIMPULAN

Penerapan Test-Driven Development (TDD) terbukti meningkatkan testabilitas kode secara signifikan. Hasil pengukuran menunjukkan bahwa TDD mengurangi kompleksitas kode, menurunkan jumlah baris kode, serta meningkatkan modularitas dan dokumentasi. Penurunan RFC hingga 86,7%, pengurangan LOC hingga 91,9%, dan penurunan NMC lebih dari 75% menunjukkan bahwa kode menjadi lebih efisien, mudah diuji, dan dipelihara. Dengan demikian, TDD dapat diandalkan sebagai pendekatan yang efektif untuk meningkatkan kualitas kode dalam pengembangan perangkat lunak.

DAFTAR PUSTAKA

[1] P. Reich and W. Maalej, “Testability Refactoring in Pull Requests: Patterns and Trends,” Jan. 2023. doi: 10.48550/arXiv.2303.14253.

[2] J. Mona, “Software Testability (Its Benefits, Limitations, and Facilitation),” *Next Generation of Internet of Things*, Sep. 2022.

[3] Y. Zhou *et al.*, “An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems,” *Science China Information Sciences*, vol. 55, no. 12, pp. 2800–2815, 2012, doi: 10.1007/s114s32-012-4745-x.

[4] V. Garousi, M. Felderer, and F. N. Kılıçaslan, “A survey on software testability,” Apr. 01, 2019, *Elsevier B.V.* doi: 10.1016/j.infsof.2018.12.003.

[5] M. Efatmaneshnik and M. Ryan, “A STUDY OF THE RELATIONSHIP BETWEEN SYSTEM TESTABILITY AND MODULARITY,” *INSIGHT*, vol. 20, pp. 20–24, Mar. 2017, doi: 10.1002/inst.12140.

[6] K. Beck, *Test-driven Development: By Example*. in Addison-Wesley signature series. Addison-Wesley, 2003. [Online]. Available: https://books.google.co.id/books?id=CUlsAQAAQBAJ

[7] M. Rahman, A. Saha, U. Chakraborty, H. Sujana, and S. M. A. Shafi, “Evaluating the impact of Test-Driven Development on Software Quality Enhancement,” *International Journal of Mathematical Sciences and Computing*, vol. 10, pp. 51–76, Sep. 2024, doi: 10.5815/ijmsc.2024.03.05.

p-ISSN: 2775-4227
e-ISSN: 2775-4219

[8] M. I. Naufal, I. #1, A. #2, and Z. Sari, "Dampak Test-Driven Development pada Kualitas Kode," *JEPIN (Jurnal Edukasi dan Penelitian Informatika)*, 2023.

[9] A. Vijaywargi and U. K. Boddapati, "EMBRACING TEST-DRIVEN DEVELOPMENT (TDD) IN MODERN ANDROID APPLICATIONS," *International Research Journal of Modernization in Engineering Technology and Science*, vol. 6, p. 11557, Apr. 2024, doi: 10.56726/IRJMETS53809.

[10] M. Alenezi, *Investigating Software Testability and Test cases Effectiveness*. 2022.

[11] M. Aniche, "CK: Code metrics for Java code by means of static analysis," https://github.com/mauricioaniche/ck.

[12] L. Neves, O. Campos, R. Santos, C. de Magalhaes, I. Santos, and R. de Souza Santos, "Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing," Feb. 2024.

[13] L. Martins, V. Pontillo, H. Costa, F. Ferrucci, F. Palomba, and I. Machado, "Test Code Refactoring Unveiled: Where and How Does It Affect Test Code Quality and Effectiveness?," Feb. 2023. doi: 10.48550/arXiv.2308.09547.

[14] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. in Robert C. Martin series. Prentice Hall, 2009. [Online]. Available: https://books.google.co.id/books?id=hjEFCAAAQBAJ

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994, doi: 10.1109/32.295895.