# ANALISIS DAMPAK *TEST-DRIVEN DEVELOPMENT* TERHADAP *TESTABILITY* PADA KODE PROGRAM STUDI KASUS: APLIKASI SEDONOR




Disusun Oleh:

N a m a : Muhammad Rizki Siraj

NIM : 21523278

**PROGRAM STUDI INFORMATIKA – PROGRAM SARJANA**

**FAKULTAS TEKNOLOGI INDUSTRI**

**UNIVERSITAS ISLAM INDONESIA**

**2025**

**HALAMAN PENGESAHAN DOSEN PEMBIMBING**

# ANALISIS DAMPAK *TEST-DRIVEN DEVELOPMENT* TERHADAP *TESTABILITY* PADA KODE PROGRAM STUDI KASUS: APLIKASI SEDONOR

## TUGAS AKHIR

Disusun Oleh:


N a m a : Muhammad Rizki Siraj

NIM : 21523278


Yogyakarta, 5 Mei 2025

Pembimbing,



( Dr. Novi Setiani, S.T, M.T )

**HALAMAN PENGESAHAN DOSEN PENGUJI**

# ANALISIS DAMPAK *TEST-DRIVEN DEVELOPMENT* TERHADAP *TESTABILITY* PADA KODE PROGRAM STUDI KASUS: APLIKASI SEDONOR

## TUGAS AKHIR

Telah dipertahankan di depan sidang penguji sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer dari Program Studi Informatika – Program Sarjana di Fakultas Teknologi Industri Universitas Islam Indonesia

Yogyakarta, 26 Mei 2025

Tim Penguji

Dr. Novi Setiani, S.T, M.T. _____

**Anggota 1**

Chanifah Indah Ratnasari, S.Kom, M.Kom _____

**Anggota 2**

Nur Wijayaning Rahayu, S.Kom., M.Cs. _____

Mengetahui,
Ketua Program Studi Informatika – Program Sarjana
Fakultas Teknologi Industri
Universitas Islam Indonesia

(Dhomas Hatta Fudholi, S.T., M.Eng., Ph.D.)

## HALAMAN PERNYATAAN KEASLIAN TUGAS AKHIR

Yang bertanda tangan di bawah ini:

Nama : Muhammad Rizki Siraj

NIM : 21523278

Tugas akhir dengan judul:

# ANALISIS DAMPAK *TEST-DRIVEN DEVELOPMENT* TERHADAP *TESTABILITY* PADA KODE PROGRAM STUDI KASUS: APLIKASI SEDONOR

Menyatakan bahwa seluruh komponen dan isi dalam tugas akhir ini adalah hasil karya saya sendiri. Apabila di kemudian hari terbukti ada beberapa bagian dari karya ini adalah bukan hasil karya sendiri, tugas akhir yang diajukan sebagai hasil karya sendiri ini siap ditarik kembali dan siap menanggung risiko dan konsekuensi apapun.

Demikian surat pernyataan ini dibuat, semoga dapat dipergunakan sebagaimana mestinya.

Yogyakarta, 6 Mei 2025




( Muhammad Rizki Siraj )

# HALAMAN PERSEMBAHAN

Segala puji bagi Allah Subhanahu wa Ta'ala. Dengan penuh rasa syukur, karya tugas akhir ini saya persembahkan kepada kedua orang tua tercinta sebagai bentuk penghargaan dan terima kasih atas segala doa, dukungan, nasihat, serta kasih sayang yang tiada henti. Semoga langkah ini menjadi awal dari upaya membahagiakan Ayah dan Ibu.

# HALAMAN MOTO

*"Wahai anak-anakku, pergi dan carilah berita tentang Yusuf beserta saudaranya. Janganlah kamu berputus asa dari rahmat Allah. Sesungguhnya tidak ada yang berputus asa dari rahmat Allah, kecuali kaum yang kafir."*

(QS. Yusuf:87)

# KATA PENGANTAR

*Assalamu'alakum Warrahmatullahi Wabarakatuh*

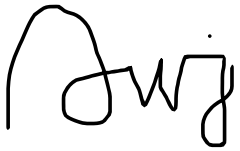
Segala puji dan syukur penulis panjatkan ke hadirat Allah Swt. atas limpahan rahmat dan hidayah-Nya, sehingga penulis dapat menyelesaikan penyusunan tugas akhir ini. Penulisan laporan ini turut menjadi wadah untuk mencurahkan berbagai perasaan, mulai dari suka cita hingga tantangan yang dialami sepanjang prosesnya. Dalam proses penyelesaian tugas akhir ini, penulis memperoleh banyak dukungan dan bantuan dari berbagai pihak. Oleh karena itu, dengan penuh rasa hormat dan terima kasih, penulis ingin menyampaikan apresiasi kepada:

1. Allah Swt. yang senantiasa memberikan rahmat dan karunia-Nya sehingga penulis mampu menyelesaikan tugas akhir ini.
2. Kedua orang tua serta abang dan kakak yang senantiasa berdoa dan memberikan dukungan dalam bentuk apa pun kepada penulis.
3. Bapak Prof. Fathul Wahid, S.T., M.Sc., Ph.D. selaku Rektor Universitas Islam Indonesia.
4. Bapak Prof., Dr., Ir., Hari Purnomo, M.T., IPU, ASEAN.Eng. selaku Pimpinan Fakultas Teknologi Industri Universitas Islam Indonesia.
5. Bapak Dr. Raden Teduh Dirghayu, S.T., M.Sc. selaku Ketua Jurusan Informatika Universitas Islam Indonesia.
6. Bapak Dhomas Hatta Fudholi, S.T., M.Eng., Ph.D. selaku Ketua Program Studi Informatika Universitas Islam Indonesia.
7. Ibu Dr. Novi Setiani, S.T., M.T. selaku dosen pembimbing tugas akhir yang senantiasa memberikan ilmu, arahan serta waktu untuk membimbing penulis mengerjakan tugas akhir ini.
8. Ibu Sri Mulyati, S.Kom., M.Kom. selaku dosen pembimbing akademik penulis.
9. Seluruh dosen serta staf pengajar Program Studi Informatika Universitas Islam Indonesia.
10. Rekan-rekan perkuliahan saya yang senantiasa memberikan dukungan serta menemani saya dalam pengerjaan tugas akhir ini.
11. Sahabat dekat penulis, Zahwa Almira Kayla, yang senantiasa menemani dan memberikan dukungan kepada penulis selama pengerjaan tugas akhir ini.
12. Sahabat Student Staff Informatika UII dan GDSC UII yang telah memberikan kenangan serta dukungan yang baik kepada penulis selama pengerjaan tugas akhir.

13. Sahabat-sahabat saya, khususnya Naela, Juan, Satriya, Anra, Bintang, Ihwan, Nabil, Nabila, Asista, Kharisma, Yasmine, Nadia, Jihan, Fikri, dan yang tidak dapat saya sebutkan satu per satu, yang selalu peduli atas segala permasalahan dan memberikan dukungan.

*Wassalamu'alaikum warrahmatullahi wabarakatuh*

Yogyakarta, 9 Mei 2024



(Muhammad Rizki Siraj)

# SARI

Dalam pengembangan perangkat lunak, aspek *testability* memegang peran penting dalam menjamin kualitas sistem. Penelitian ini bertujuan untuk menganalisis dampak penerapan metode *Test-Driven Development* (TDD) terhadap *testability* kode, dengan studi kasus pada aplikasi Android bernama Sedonor. Aplikasi ini dipilih karena memiliki kompleksitas fitur yang cukup tinggi, sehingga menjadi objek yang relevan untuk mengkaji efektivitas TDD dalam meningkatkan keterujian kode. Penelitian dilakukan dengan pendekatan eksperimen, di mana aplikasi Sedonor dikembangkan ulang menggunakan metode TDD dan hasilnya dibandingkan dengan versi asli yang dikembangkan tanpa TDD. Pengukuran *testability* dilakukan menggunakan metrik *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), dan *Number of Method Calls* (NMC). Hasil penelitian menunjukkan bahwa penerapan TDD memberikan pengaruh positif terhadap peningkatan *testability* kode, ditandai dengan penurunan kompleksitas logika, peningkatan dokumentasi, serta modularitas kode yang lebih baik. Selain pengukuran kuantitatif, evaluasi juga dilakukan melalui *expert review* untuk menilai aspek *clean architecture* dan *testability* kode. Temuan dari penelitian ini memberikan rekomendasi bahwa TDD dapat diadopsi sebagai praktik standar dalam pengembangan aplikasi Android yang membutuhkan keterujian dan kualitas kode tinggi.

Kata kunci: *Test-Driven Development*, *testability* kode, aplikasi Android, metrik perangkat lunak, TDD.

# GLOSARIUM

| | |
|---|---|
| *Test-Driven Development* (TDD) | Metode pengembangan perangkat lunak di mana pengujian ditulis terlebih dahulu sebelum implementasi kode, mengikuti siklus *Red-Green-Refactor*. |
| *Testability* | Tingkat kemudahan suatu perangkat lunak untuk diuji dalam konteks pengujian tertentu. |
| *Unit testing* | Pengujian perangkat lunak pada bagian terkecil dari sistem, seperti fungsi atau metode, untuk memastikan fungsionalitasnya secara independen. |
| JUnit | *Framework* pengujian unit berbasis Java yang digunakan untuk menulis dan menjalankan tes secara otomatis. |
| *Mocking* | Teknik dalam pengujian perangkat lunak untuk mensimulasikan objek atau dependensi eksternal agar pengujian dapat dilakukan secara terisolasi. |

# DAFTAR ISI

HALAMAN JUDUL ..... i
HALAMAN PENGESAHAN DOSEN PEMBIMBING ..... ii
HALAMAN PENGESAHAN DOSEN PENGUJI ..... iii
HALAMAN PERNYATAAN KEASLIAN TUGAS AKHIR ..... iv
HALAMAN PERSEMBAHAN ..... v
HALAMAN MOTO ..... vi
KATA PENGANTAR ..... vii
SARI ..... ix
GLOSARIUM ..... x
DAFTAR ISI ..... xi
DAFTAR TABEL ..... xii
DAFTAR GAMBAR ..... xiii
DAFTAR PERSAMAAN ..... xiv
BAB I PENDAHULUAN ..... 1
1.1 Latar Belakang ..... 1
1.2 Rumusan Masalah ..... 3
1.3 Batasan Masalah ..... 3
1.4 Tujuan Penelitian ..... 3
1.5 Manfaat Penelitian ..... 3
1.6 Metode Penelitian ..... 4
1.7 Sistematika Penelitian ..... 4
BAB II LANDASAN TEORI DAN KAJIAN PUSTAKA ..... 6
2.1 *Test-Driven Development* (TDD) ..... 6
2.2 *Unit testing* ..... 6
2.3 JUnit ..... 7
2.4 *Testability* Perangkat Lunak ..... 8
2.5 Metrik Pengukuran *Testability* ..... 9
2.6 Kajian Pustaka ..... 11
BAB III METODE PENELITIAN ..... 15
3.1 Alur Proses ..... 15
3.2 Pemilihan Metrik dan Analisis Kode Lama ..... 16
3.3 Pelaksanaan TDD ..... 27
3.4 Analisis Pengaruh TDD ..... 30
BAB IV HASIL DAN PEMBAHASAN ..... 33
4.1 Perhitungan Metrik dan Analisis Kode Lama ..... 33
4.2 Pelaksanaan TDD ..... 39
4.3 Pengukuran Metrik Kode Baru ..... 56
4.4 *Expert review* Terhadap Kode Lama dan Kode Baru ..... 61
4.5 Kesimpulan Hasil Penelitian ..... 65
BAB V KESIMPULAN ..... 66
5.1 Kesimpulan ..... 66
5.2 Saran ..... 67
DAFTAR PUSTAKA ..... 69
LAMPIRAN ..... 71

# DAFTAR TABEL

Tabel 2.1 Kajian Pustaka Penelitian Terdahulu ..... 11
Tabel 3.1 Cakupan Metrik CK dan Metrik Alenezi ..... 18
Tabel 3.2 Hasil Perhitungan Metrik Untuk Kode Simulasi ..... 20
Tabel 3.3 Karakteristik Teknik ThinkMate dan Sedonor ..... 23
Tabel 3.4 Ambang Batas Metrik ..... 27
Tabel 3.5 Pemetaan *Usecase* ..... 28
Tabel 4.1 Hasil Perhitungan Metrik Kode Lama ..... 37
Tabel 4.2 Pemetaan Kelas yang Berkaitan Terhadap *use case* ..... 40
Tabel 4.3 Desain *Test Case* Melihat Artikel Berhasil ..... 41
Tabel 4.4 Desain *Test Case* Melihat Artikel Menghasilkan Eror ..... 41
Tabel 4.5 *Test Case* Memasukkan Data ke Riwayat Berhasil ..... 46
Tabel 4.6 *Test Case* Memasukkan Data ke Riwayat Gagal ..... 46
Tabel 4.7 Desain *Test Case* Mengakses *Chatbot* Berhasil ..... 51
Tabel 4.8 *Test Case* Mengakses *Chatbot* Gagal ..... 51
Tabel 4.9 Hasil Pengukuran Metrik Kode Baru ..... 57
Tabel 4.10 Perbandingan Rata-rata Metrik Kode Lama dan Baru ..... 59
Tabel 4.11 Tabel Identitas Responden ..... 61
Tabel 4.12 Hasil Wawancara Responden ..... 62

# DAFTAR GAMBAR

Gambar 2.1 Potongan Kode JUnit ..... 8
Gambar 3.1 Diagram Alur ..... 16
Gambar 3.2 Potongan Kode Simulasi Perhitungan Metrik ..... 20
Gambar 3.3 Diagram *Usecase* ..... 28
Gambar 3.4 Diagram *Refactor* ..... 30
Gambar 4.1 Melakukan *Decompiling* ..... 35
Gambar 4.2 Menjalankan *Library* Lewat *Terminal* ..... 37
Gambar 4.3 Kode *Test Refactoring* ..... 42
Gambar 4.4 Kode *Test Refactoring* ..... 42
Gambar 4.5 Hasil *Test ArticleRepository* Gagal ..... 43
Gambar 4.6 Kode Implementasi *Refactoring* ..... 43
Gambar 4.7 Hasil *Test ArticleRepository* Berhasil ..... 44
Gambar 4.8 Kode *ArticleRepository* Sebelum Di*refactor* ..... 45
Gambar 4.9 Kode *Refactoring* Setelah Di*refactor* ..... 45
Gambar 4.10 Kode *Test DonorHistoryRepository* ..... 47
Gambar 4.11 Kode *Test DonorHistoryRepository* ..... 48
Gambar 4.12 Hasil *Test DonorHistoryRepository* Gagal ..... 48
Gambar 4.13 Kode Implementasi *DonorHistoryRepository* ..... 49
Gambar 4.14 Hasil *Test DonorHistoryRepository* Berhasil ..... 49
Gambar 4.15 Kode *DonorHistoryRepository* Setelah Di*refactor* ..... 50
Gambar 4.16 Kode *ChatbotRepository* ..... 52
Gambar 4.17 Kode *Test ChatbotRepository* ..... 53
Gambar 4.18 Hasil *Test Chatbot* Repostiory ..... 53
Gambar 4.19 Kode Implementasi *ChatbotRepository* ..... 54
Gambar 4.20 Hasil *Test ChatbotRepository* Berhasil ..... 54
Gambar 4.21 Kode *ChatbotRepository* Setelah Di*refactor* ..... 55

# DAFTAR PERSAMAAN

Persamaan *Response for class* (2.1) ..... 9
Persamaan *Lines of Code*s (2.2) ..... 10
Persamaan *Lines of Comment*s (2.3) ..... 10
Persamaan *Weight method per class* (2.4) ..... 10
Persamaan *Cyclomatic complexity* (2.5) ..... 10
Persamaan *Number of Method Calls* (2.6) ..... 10

# BAB I
# PENDAHULUAN

## 1.1 Latar Belakang

Pada pengembangan perangkat lunak, pengujian merupakan aspek fundamental untuk menjamin kualitas sebuah sistem. Pengujian perangkat lunak adalah kemampuan untuk memeriksa berbagai artefak perangkat lunak, seperti persyaratan, modul, dan dokumen (Mona, 2022). Pengujian perangkat lunak dilakukan untuk memastikan perangkat lunak yang berjalan dengan baik dan minim eror (Mona, 2022).

Namun, tidak semua perangkat lunak memiliki tingkat kemudahan yang sama dalam pengujian; beberapa lebih mudah diuji dibandingkan dengan yang lain. Hal ini sangat bergantung pada tingkat *testability* dari perangkat lunak tersebut. *Testability* perangkat lunak adalah sejauh mana perangkat lunak dapat mendukung proses pengujian dalam konteks tertentu. Semakin tinggi *testability*-nya, semakin mudah untuk menemukan kesalahan dalam sistem selama pengujian. Sebaliknya, *testability* yang rendah meningkatkan kompleksitas dan durasi pengujian, yang pada akhirnya mengurangi peluang untuk menemukan kesalahan dalam sistem (Gəruslu et al., 2018).

Salah satu faktor yang mempengaruhi *testability* perangkat lunak adalah kualitas kode di dalamnya. Beberapa aspek seperti kompleksitas kode, modularitas, dan *coupling* memiliki pengaruh besar terhadap kualitas kode yang dapat diuji. Penelitian oleh Zhou et al. (2012) menunjukkan bahwa *class* dengan kompleksitas tinggi membutuhkan usaha yang lebih besar untuk pengujian unit dibandingkan dengan *class* yang lebih sederhana. Menguji kode yang kompleks membutuhkan biaya pengujian yang lebih besar (Garousi et al., 2019). Sebaliknya, kode dengan kompleksitas yang rendah dan modularitas yang tinggi cenderung lebih mudah dan murah untuk diuji. Hal ini didukung oleh penelitian Efatmaneshnik & Ryan (2017) yang menyatakan bahwa sistem yang dibangun dengan arsitektur modular dua lapis menurunkan biaya pengujian hampir 50% dibandingkan dengan sistem yang tidak memiliki lapisan modular sama sekali. *Coupling* juga mempengaruhi *testability* secara signifikan. Garousi et al. (2019) menyebutkan bahwa pembuatan *test stub* diperlukan untuk pengujian unit dan integrasi ketika unit yang diuji bergantung pada unit lain yang belum diuji atau bahkan belum dikodekan. *Coupling* antar unit ini akan meningkatkan upaya pembuatan *stub* karena semakin tinggi

tingkat *coupling*, semakin banyak fitur yang diperlukan dalam *stub*, yang pada akhirnya menurunkan *testability* (Garousi et al., 2019).

Salah satu pendekatan yang terbukti efektif dalam meningkatkan *testability* kode adalah *Test-Driven Development* (TDD). TDD adalah metode pengembangan perangkat lunak yang memiliki pendekatan unik. TDD mengharuskan pengembang untuk menuliskan pengujian terlebih dahulu sebelum mengembangkan kode, yang tidak hanya memastikan bahwa setiap bagian kode memiliki cakupan pengujian yang baik tetapi mendorong praktik penulisan kode yang baik juga (Beck, 2003).

Penerapan TDD dapat memberikan banyak manfaat untuk meningkatkan *testability* kode. Menurut penelitian yang dilakukan Naufal et al. (2023) penerapan TDD dapat menurunkan *cyclomatic complexity* hingga 15,87%. Dengan TDD pengembang dapat mendeteksi kesalahan lebih dini dan mendorong perancangan kode yang lebih baik (Rahman et al., 2024). Hal ini tentu saja berdampak ke naiknya *testability* kode karena pengembangan yang menganut metode *test first*.

Penelitian ini akan mengeksplorasi dampak TDD terhadap *testability* kode dengan studi kasus aplikasi Android Sedonor. Sedonor adalah aplikasi Android dengan tema kesehatan yang memiliki beberapa fitur dengan kompleksitas yang tinggi. Beberapa fitur Sedonor bergantung pada layanan pihak ketiga yang bersifat dinamis dan berpotensi berubah sewaktu-waktu, seperti API lokasi atau notifikasi. Kompleksitas ini membuka peluang pengembangan lebih lanjut sekaligus menuntut struktur kode yang memiliki *testability* yang baik agar mudah dirawat dan disesuaikan. Oleh karena itu, Sedonor menjadi objek yang relevan untuk penerapan *Test-Driven Development* (TDD) dalam pengembangan perangkat lunaknya. Hal ini juga didukung oleh penelitian Vijaywargi & Boddapati (2024) yang menyatakan bahwa TDD merupakan prinsip vital dalam pengembangan aplikasi Android modern.

Dalam penelitian ini, aplikasi Sedonor akan dibuat ulang menggunakan prinsip TDD, di mana pengujian dituliskan terlebih dahulu sebelum penullisan kode. Kode hasil penulisan dengan TDD akan dibandingkan dengan kode versi asli yang dikembangkan tanpa TDD untuk melihat perbedaan dalam hal *testability*. Kode lama dan baru akan diukur secara kuantiatif melalui metrik dan secara kualitatif melalui wawancara ke ahli. Metrik-metrik yang akan diukur meliputi *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), *Number of Method Calls* (NMC) karena metrik-metrik tersebut adalah pengukur paling ideal untuk mengukur *testability* kode (Alenezi, 2022).

## 1.2 Rumusan Masalah

Perumusan masalah pada penelitian ini adalah sebagai berikut:

a. Bagaimana penerapan *Test-Driven Development* (TDD) mempengaruhi *testability* kode pada aplikasi Android yang memiliki fitur kompleks, seperti aplikasi Sedonor?
b. Apakah kode yang dikembangkan menggunakan prinsip TDD memiliki *testability* yang lebih baik dibandingkan dengan kode yang dikembangkan tanpa TDD, dilihat dari metrik-metrik seperti *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), *Number of Method Calls* (NMC)?

## 1.3 Batasan Masalah

Batasan masalah pada penelitian ini adalah sebagai berikut:

a. Penelitian hanya mencakup pengembangan ulang aplikasi ini dengan menggunakan pendekatan TDD dan perbandingan dengan versi aplikasi yang dikembangkan tanpa TDD.
b. Penelitian akan fokus pada pengukuran *testability* kode menggunakan metrik tertentu, seperti *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), *Number of Method Calls* (NMC).

## 1.4 Tujuan Penelitian

Tujuan penelitian ini adalah sebagai berikut:

a. Menganalisis dampak penerapan *Test-Driven Development* (TDD) terhadap *testability* kode pada aplikasi Android Sedonor yang memiliki fitur dengan kompleksitas tinggi.
b. Mengukur dan membandingkan metrik *testability* kode antara pengembangan dengan TDD dan non-TDD, termasuk *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), *Number of Method Calls* (NMC).

## 1.5 Manfaat Penelitian

Manfaat dari penelitian ini didefinisikan sebagai berikut:

a. Penelitian ini memberikan hasil mengenai manfaat TDD dalam meningkatkan *testability* kode, sehingga dapat membantu pengembang dalam membuat keputusan untuk mengadopsi TDD dalam proyek mereka.

b. Penelitian ini dapat menjadi acuan untuk meningkatkan pengembangan perangkat lunak yang lebih efisien dan berkualitas.
c. Bagi pihak pengembang aplikasi Sedonor, penelitian ini memberikan *insight* langsung mengenai kekuatan dan kelemahan struktur kode yang ada, serta rekomendasi pengembangan ulang berbasis TDD untuk meningkatkan *testability* kode ke depannya.

## 1.6 Metode Penelitian

Dalam penelitian, metode yang akan digunakan adalah sebagai berikut:

1. Pemilihan Metrik dan Analisis Kode Lama: Sebelum menerapkan TDD, penulis terlebih dahulu melakukan studi literatur untuk menentukan metrik yang tepat, kemudian mengukur dan menganalisis kode lama berdasarkan metrik tersebut.
2. Pelaksanaan TDD: Kode kemudian dikembangkan ulang menggunakan metode TDD yang mencakup pembuatan *unit test*, implementasi kode program, dan *refactor*.
3. Analisis Pengaruh TDD: Setelah proses pengembangan ulang selesai, kode diuji kembali menggunakan metrik yang sama. Selanjutnya, dilakukan *expert review* untuk mengevaluasi kualitas kode dari aspek arsitektur, modularitas, dan *testability*. Hasil dari tahap ini dianalisis dan dibandingkan dengan hasil sebelumnya.

## 1.7 Sistematika Penelitian

Struktur penulisan laporan dari penelitian dijelaskan pada pembahasan dibawah ini:

a. BAB I PENDAHULUAN

   Bab ini berisi tentang gambaran umum tentang topik yang dibahas. Mencakup tentang pembahasan latar belakang, rumusan masalah, batasan masalah, tujuan penelitian, manfaat penelitian, metode penelitian, dan struktur laporan.

b. BAB II LANDASAN TEORI DAN KAJIAN PUSTAKA

   Bab ini memuat teori dan definisi yang terkait dengan penelitian baik dari buku, jurnal, maupun website. Selain itu bab ini juga berisi referensi artikel penulisan yang mengimplementasikan setiap subbab.

c. BAB III METODOLOGI PENELITIAN

   Bab ini berisi tentang penjelasan tentang metode penelitian yang digunakan serta langkah-langkah yang digunakan dalam penelitian ini. Bab ini yang menjadi alat pengujian untuk Bab IV.

d. BAB IV HASIL DAN PEMBAHASAN

Bab ini berisi tentang hasil pengembangan ulang dengan prinsip TDD yang telah dilakukan dan menguraikan analisis dari pembahasan setiap metrik-metrik yang telah ditentukan di awal untuk mengukur *testability* kode. Hasil dari bab ini akan menjadi acuan bab selanjutnya.

e. BAB V KESIMPULAN DAN SARAN

Bab ini berisi rangkuman dan kesimpulan dari hasil penelitian, serta saran-saran yang dapat membantu pengembangan analisis di masa depan. Saran-saran ini diharapkan dapat memberikan arahan untuk penelitian selanjutnya dan meningkatkan kualitas penelitian di bidang terkait.

# BAB II
# LANDASAN TEORI DAN KAJIAN PUSTAKA

## 2.1 *Test-Driven Development* (TDD)

*Test-Driven Development* (TDD) merupakan pendekatan dalam pengembangan perangkat lunak yang menekankan pentingnya pengujian sebagai bagian utama dari proses pengembangan. Metode ini sering dipakai pada pengembangan *agile* seperti *Exteme Programming* (XP) (Rahman et al., 2024). Dalam pelaksanaannya, pengembang umumnya memulai dengan membuat tes yang awalnya tidak berhasil, kemudian menulis kode secukupnya agar tes tersebut dapat lolos, dan akhirnya melakukan *refactoring* untuk memperbaiki desain serta meningkatkan keterbacaan kode. (Vijaywargi & Boddapati, 2024).

Seiring dengan perkembangan metodologi *agile*, tren TDD juga mengalami peningkatan. Menurut penelitian yang dilakukan oleh Vijaywargi & Bodappati (2024), penggunaan TDD secara global di bidang pengembangan perangkat lunak mengalami peningkatan sebesar 22% dalam 3 tahun terakhir. Penelitian yang sama juga mencatat 92% pengembang yang menerapkan prinsip TDD mengintegrasikan *framework-framework* pengujian ke dalam proyek mereka, membuktikan bahwa praktik ini semakin menjadi standar dalam siklus pengembangan perangkat lunak. Peningkatan ini tentunya didasari oleh praktik TDD yang memberikan banyak keuntungan.

Menurut (Rahman et al., 2024), pengembangan perangkat lunak menggunakan TDD dapat memberikan banyak manfaat. Praktik TDD, yang mengharuskan penulisan tes sebelum menulis kode, mampu mendeteksi kesalahan sejak dini dan mengurangi isu-isu kritis dalam pengembangan. Dalam lingkungan seperti ini, *test suite* TDD berfungsi sebagai *safety net* yang melindungi proses pengembangan dari potensi masalah yang tidak diinginkan. Praktik TDD, yang sangat terfokus dan erat kaitannya dengan *unit testing*, memungkinkan pendeteksian *bug* sejak dini, sehingga mencegah perangkat lunak dari menghadapi masalah besar selama proses pengembangan.

## 2.2 *Unit testing*

*Unit testing* adalah salah satu pengujian dalam perangkat lunak yang menguji komponen terkecil dan memastikan bahwa komponen-komponen kecil tersebut dapat berjalan dengan

lancar. Dalam konteks ini, *unit testing* berfokus untuk memeriksa unit-unit terkecil dalam perangkat lunak seperti fungsi, metode, kelas untuk memastikan keakuratan dan fungsionalitasnya secara terisolir (Abran et al., 2001). Pengujian ini dilakukan dengan membuat serangkaian tes otomatis yang mengisolasi setiap unit dari bagian lain dari aplikasi, sehingga memungkinkan pengembang untuk menemukan dan memperbaiki kesalahan secara lebih cepat dan spesifik.

Keuntungan utama dari *unit testing* adalah menghasilkan kode yang lebih rapi dan minim *bug*. Menurut penelitian yang dilakukan oleh Grano et al. (2020), *unit testing* terbukti dapat meningkatkan kualitas kode melalui deteksi *bug* lebih awal, peningkatan kemampuan pemeliharaan (*maintainability*), serta memastikan bahwa kode dapat diuji dengan baik (*testability*). *Unit testing* berperan penting dalam menjaga stabilitas kode, memungkinkan *refactoring* yang aman, dan membantu dalam mengurangi risiko kesalahan di tahap pengembangan yang lebih lanjut.

Menurut Vijaywargi & Boddapati (2024), *unit testing* juga memiliki peran penting dalam praktik TDD. Dengan menerapkan *unit testing* dalam TDD, pengembang dapat memastikan bahwa setiap perubahan kode tetap memenuhi spesifikasi fungsionalitas yang diharapkan dan tidak menyebabkan gangguan pada bagian lain dari sistem. Hal ini membantu menjaga stabilitas perangkat lunak secara keseluruhan selama proses pengembangan. *Framework* seperti JUnit sering digunakan oleh pengembang sebagai alat bantu untuk menjalankan pengujian ini, memastikan bahwa setiap bagian kode dapat diujicobakan secara terpisah sebelum diintegrasikan dengan komponen lainnya .

## 2.3 JUnit

JUnit adalah salah satu *framework* pengujian unit yang sangat populer untuk aplikasi Java, dan dirancang khusus untuk membantu pengembang melakukan pengujian pada bagian terkecil dari kode mereka, yaitu unit. Fokus utama JUnit adalah memungkinkan pengembang untuk menulis dan menjalankan tes secara otomatis serta memastikan bahwa kode berfungsi seperti yang diharapkan. JUnit mendorong "pengujian terlebih dahulu kemudian pengkodean" yang menekankan pada pembuatan rincian pengujian terlebih dahulu, kemudian bagian aplikasi dari kode harus dievaluasi. Ini meningkatkan efisiensi *programmer* dan keandalan kode sistem, yang pada gilirannya mengurangi frustrasi *programmer* dan waktu *debug* (Venkatesan et al., 2020). JUnit dirancang untuk bekerja dengan aplikasi berbasis Java, dan mendukung berbagai jenis pengujian seperti pengujian unit, pengujian integrasi, dan pengujian fungsional.

Dalam pengembangan perangkat lunak menggunakan JUnit, pengujian kode dilakukan dengan cara yang lebih terstruktur dibandingkan pengembangan tanpa *framework* pengujian seperti JUnit. Sebagai contoh, pada Gambar 2.1 terdapat penerapan kode sederhana menggunakan JUnit untuk memeriksa apakah dua nilai sama.

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class CalculatorTest {
    @Test
    public void testAddition() {
        Calculator calc = new Calculator();
        int result = calc.add(2, 3);
        assertEquals(5, result);
    }
}
```

Gambar 2.1 Potongan Kode JUnit

Pada kode di atas, metode *assertEquals(5, result)* digunakan untuk memverifikasi apakah hasil operasi dari metode *add()* sesuai dengan yang diharapkan, yaitu lima. Penggunaan fungsi *assertsions* seperti *assertEquals* sangat membantu dalam memverifikasi hasil pengujian secara otomatis. Hal ini tentu jauh lebih efisien dibandingkan cara primitif yang mengharuskan pengembang menulis kode terpisah untuk menampilkan hasil secara visual, yang tidak hanya memakan waktu, tetapi juga rentan terhadap kesalahan, terutama dalam proyek berskala besar.

Dengan kata lain, penggunaan JUnit dalam pengembangan perangkat lunak memastikan bahwa pengujian dilakukan secara konsisten dan memberikan *feedback* yang lebih cepat serta lebih akurat dibandingkan pengembangan tanpa alat pengujian formal. Penggunaan JUnit juga mempermudah penerapan TDD dengan meningkatkan *testability* perangkat lunak.

## 2.4 *Testability* Perangkat Lunak

*Testability* perangkat lunak mengacu pada kemudahan dalam menguji suatu bagian perangkat lunak, seperti sistem, modul, atau dokumen persyaratan dan desain, dalam konteks pengujian tertentu. Tingkat *testability* ini dipengaruhi oleh beberapa faktor, termasuk desain perangkat lunak, struktur kode, dan metode pengujian yang memadai. Semakin tinggi *testability* perangkat lunak, semakin mudah proses pengujiannya. Dengan meningkatkan *testability*, seorang pengembang dapat menciptakan perangkat lunak berkualitas tinggi yang lebih mudah diidentifikasi kesalahannya dan diperbaiki (Garousi et al., 2019).

Peningkatan *testability* perangkat lunak dapat dilakukan melalui beberapa pendekatan. Menurut penelitian Garousi et al. (2019), *testability* perangkat lunak dapat ditingkatkan dengan meningkatkan keterlihatan, menambahkan *assertion*, dan meningkatkan *controllability*.

Peningkatan keterlihatan memungkinkan pengembang lebih mudah memantau keluaran serta perilaku sistem selama pengujian, sehingga mempermudah proses identifikasi masalah. Selain itu, peningkatan *controllability* penting agar setiap komponen perangkat lunak dapat diatur dan diuji secara terpisah, tanpa saling bergantung satu sama lain.

Di sisi lain, penerapan metode *Test-Driven Development* (TDD) membantu memastikan perangkat lunak dirancang dengan prinsip-prinsip terbaik, yang berfokus pada pengujian di tahap paling awal. Salah satu prinsip utama TDD adalah memulai dengan membuat pengujian terkecil atau *unit test* terlebih dahulu, sebelum menulis kode yang sebenarnya. Prinsip ini memungkinkan pengembang untuk mengidentifikasi kesalahan pada level paling mendasar, sehingga mengurangi risiko *bug* atau masalah besar muncul di kemudian hari. Hal ini yang membuat *testability* perangkat lunak menjadi optimal dengan penggunaan metode TDD.

## 2.5 Metrik Pengukuran *Testability*

Pengukuran *testability* perangkat lunak bertujuan untuk mengevaluasi sejauh mana perangkat lunak dapat diuji secara efektif dan efisien. Metrik-metrik ini membantu pengembang dalam memahami kompleksitas sistem, memprediksi kesulitan dalam pengujian, serta mengidentifikasi area yang memerlukan perbaikan. Dalam penelitian yang dilakukan oleh Alenezi (2022), beberapa metrik pengukuran *testability* diidentifikasi dan diberi peringkat berdasarkan dampaknya terhadap *testability* perangkat lunak. Penelitian ini menggunakan model *data mining* dan sejumlah metrik *data mining*, seperti *Gain Ratio, Information Gain, Symmetrical Uncertainty, dan OneR*, untuk menentukan metrik yang memiliki pengaruh signifikan terhadap *testability* perangkat lunak. Berdasarkan hasil penelitian tersebut, lima metrik teratas yang paling berpengaruh terhadap *testability* perangkat lunak adalah sebagai berikut:

1. *Response for Class* (RFC)

   RFC menghitung jumlah total metode unik yang dapat dipanggil sebagai respon atas pesan yang diterima oleh sebuah kelas. Ini mencakup metode di dalam kelas tersebut dan metode yang dipanggil melalui objek lain. Persamaan untuk menghitung RFC ditulis sebagai:

   $$RFC = |Unique\ (\sum_{i=1}^{n} Metode\ yang\ dapat\ dipanggil\ dalam\ kelas)| \qquad (2.1)$$

   di mana *n* adalah jumlah metode yang dipanggil di dalam kelas dan *"unique"* berarti pemanggilan metode yang sama.
2. *Lines of Code* (LOC)

LOC menghitung total baris kode dalam sebuah kelas atau metode, tidak termasuk baris kosong dan komentar. LOC sering digunakan sebagai indikator awal dari kompleksitas kode; semakin banyak baris, semakin besar kemungkinan adanya kesalahan atau *bug* yang sulit ditemukan. Persamaan untuk menghitung LOC ditulis sebagai:

$$LOC = Total\ baris\ kode - Baris\ kosong - Komentar \quad (2.2)$$

3. *Lines of Comment* (LOCCOM)

   LOCCOM mengukur jumlah baris yang digunakan untuk komentar dalam kode sumber. Metrik ini mencerminkan seberapa baik dokumentasi dan keterbacaan kode. Kode yang didokumentasikan dengan baik lebih mudah dipahami dan diuji. Persamaan untuk menghitung LOCCOM ditulis sebagai:

   $$LOCCOM = \sum_{l=1}^{N} Baris\ komentar\ (i) \quad (2.3)$$

4. *Weight Method per Class* (WMC)

   WMC menghitung total kompleksitas dari semua metode dalam sebuah kelas. Kompleksitas metode biasanya diukur menggunakan kompleksitas McCabe atau jumlah instruksi logika dalam metode. Semakin tinggi nilai WMC, semakin sulit untuk menguji kelas tersebut karena peningkatan kompleksitas logika. Persamaan untuk menghitung WMC ditulis sebagai:

   $$WMC = \sum_{i=1}^{n} Kompleksitas\ Metode\ (i) \quad (2.4)$$

   di mana *n* adalah jumlah metode dalam kelas, dan Kompleksitas Metode dapat dihitung dengan:

   $$Cyclomatic\ Complexity = E - N + 2P \quad (2.5)$$

   *E* adalah jumlah cabang, *N* adalah jumlah simpul, dan *P* adalah jumlah komponen terhubung.

5. *Number of Method Calls* (NMC)

   NMC menghitung jumlah metode yang dipanggil oleh suatu kelas. Metrik ini memberikan gambaran tentang seberapa banyak suatu kelas atau metode berinteraksi dengan bagian lain dari sistem. Semakin banyak panggilan metode, semakin tinggi kebutuhan pengujian yang menyeluruh. Persamaan untuk menghitung NMC ditulis sebagai:

   $$NMC = \sum_{i=1}^{n} Panggilan\ metode\ (i) \quad (2.6)$$

   di mana *n* adalah jumlah metode yang dipanggil di dalam kelas tersebut.

## 2.6 Kajian Pustaka

Penelitian ini menggunakan kajian pustaka. Analisis dilakukan terhadap objek yang digunakan dalam penelitian berdasarkan kajian tersebut.

### 2.6.1 Penelitian terdahulu

Bagian ini akan membahas penelitian sebelumnya yang relevan dengan topik penelitian yang sedang dibahas. Melalui berbagai referensi, penulis berusaha mengidentifikasi perbedaan pada setiap topik yang telah dipilih. Tinjauan ini memberikan gambaran umum tentang berbagai penelitian terkait, dengan tujuan untuk memperjelas posisi penelitian yang dilakukan oleh penulis. Tabel yang merangkum penelitian-penelitian tersebut dapat dilihat pada Tabel 2.1.

Tabel 2.1 Kajian Pustaka Penelitian Terdahulu

| No | Judul | Peneliti dan Tahun | Batasan | Metode | Hasil |
|---|---|---|---|---|---|
| 1 | Dampak *Test-Driven Development* pada Kualitas Kode | Muhammad Iqbal Naufal Ilmi, Aminudin, Zamah Sari (2023) | Penelitian ini terbatas pada satu repositori yang diambil dari GitHub dan berbasis web. | Analisis langsung dengan prinsip TDD dan non-TDD. | Hasilnya menunjukkan bahwa penggunaan TDD meningkatkan *co-de coverage* lebih dari 100%, mengurangi *cyclo matic complexity* sebe-sar 15,78%, meningkatkan *hal -stead volume* sebesar 35%, dan *maintainabili-ty index* sebesar 11%. |

| | | | | | |
|---|---|---|---|---|---|
| 2 | *Test-Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review* | Dua Agha, Rashida Sohail, Ramsha Qaboolio, Sania Bhatti (2023) | Penelitian ini hanya melakukan *Systematic Literature Review* (SLR) tanpa menguji perangkat lunak spesifik secara langsung. | *Systematic Literature Review* (SLR) terhadap beberapa literatur. | TDD secara signifikan meningkatkan *co-de coverage*, mengurangi cacat kode, meningkatkan kualitas internal dan eksternal perangkat lunak, serta meningkatkan kepuasan pengembang. |
| 3 | *Embracing Test-Driven Development (TDD) in Modern Android Applications* | Ayush Vijaywargi dan Uchinta Kumar Boddapati (2024) | Penelitian ini hanya melakukan pendekatan studi kasus dan survei terhadap beberapa proyek Android yang menggunakan TDD dalam pengemba-ngannya. | Penelitian ini menggunakan pendekatan studi kasus dan survei terhadap berbagai proyek Android untuk mengevaluasi penerapan *Test-Driven Development* (TDD). Menggunakan alat-alat seperti JUnit, Espresso, dan Mockito, penelitian menyoroti dampak TDD pada | TDD meningkatkan kualitas kode, menurunkan densitas *bug* hingga 40%, dan memberikan peningkatan produktivitas pengembang sebesar 35%. |

| | | | | | |
|---|---|---|---|---|---|
| | | | | pengurangan cacat kode dan peningkatan produktivitas. | |
| 4 | *Evaluating the impact of Test-Driven Development on Software Quality Enhancement* | Md. Sydur Rahman, Aditya Kumar Saha, Uma Chakraborty, Humaira Tabassum Sujana, dan S. M. Abdullah Shafi (2024) | Penelitian ini hanya menggunakan survei dan wawancara untuk memperoleh hasil tanpa analisis perangkat lunak secara langsung. | Penelitian ini menggunakan survei dan wawancara kualitatif terhadap pengembang perangkat lunak untuk memahami dampak TDD pada berbagai aspek kualitas perangkat lunak. Pendekatan ini membantu menilai efektivitas TDD melalui kombinasi metode kuantitatif dan kualitatif. | TDD terbukti mempercepat deteksi cacat di awal pengembangan, meningkatkan desain kode modular, dan mengurangi biaya perbaikan kesalahan. |
| 5 | Implementasi *Test-Driven Development* Pada Rancang Bangun Aplikasi E-Voting PEMIRA Universitas Dinamika | Sebastianus Sembara (2022) | Fokus pada pengujian awal, tantangan implementasi pada aplikasi *e-voting*. Penelitian ini juga menggunakan *platform* | Penelitian ini menerapkan metode *Test-Driven Development* (TDD) dengan *framework* PHPUnit untuk pengembangan aplikasi *e-voting*. | TDD terbukti meningkatkan pengujian aplikasi hingga 100% sukses, meningkatkan tingkat penerimaan aplikasi hingga 89,82%. |

| | | | | | |
|---|---|---|---|---|---|
| | | | berbasis web sebagai objek penelitian. | | |

### 2.6.2 Telaah Kajian Pustaka

Subbab ini merangkum berbagai penelitian terkait implementasi *Test-Driven Development* (TDD) dalam pengembangan perangkat lunak, dengan variasi yang signifikan pada pendekatan dan objek yang digunakan. Beberapa penelitian sebelumnya, seperti yang dilakukan oleh Naufal et al. (2023), berfokus pada pengujian aplikasi berbasis web. Mereka menemukan bahwa penerapan TDD meningkatkan *code coverage* lebih dari 100% dan mengurangi *cyclomatic complexity* sebesar 15,78%. Di sisi lain, penelitian oleh Agha et al. (2023) dalam karya ilmiah berjudul "*Test-Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review*" menggunakan pendekatan *Systematic Literature Review* (SLR) tanpa pengujian langsung pada perangkat lunak. Meskipun hanya berbasis literatur, mereka menyimpulkan bahwa TDD dapat secara signifikan mengurangi cacat kode dan meningkatkan kepuasan pengembang. Selain itu, penelitian oleh Vijaywargi & Boddapati (2024) lebih menekankan pada aplikasi Android, namun penelitian ini berbasis studi kasus dan survei, bukan pada analisis perangkat lunak secara langsung. Hal ini menunjukkan adanya potensi penerapan TDD pada berbagai basis teknologi yang berbeda.

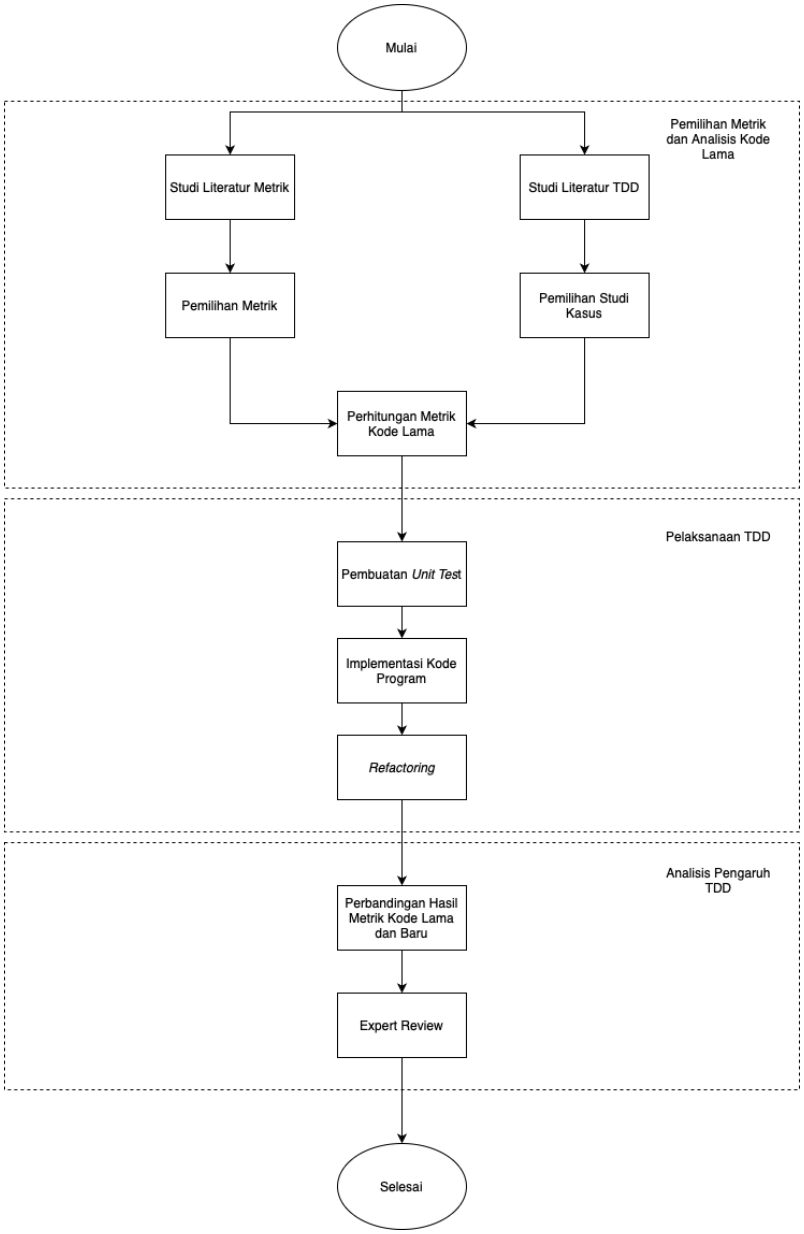
Penelitian ini mencoba untuk melengkapi kesenjangan yang ada dalam penelitian-penelitian sebelumnya dengan melakukan pengujian langsung pada perangkat lunak berbasis Android. Dalam penelitian ini, fokus utama ada pada proses *refactoring* dan analisis menyeluruh terhadap kualitas perangkat lunak, khususnya pada dampak TDD terhadap *testability* perangkat lunak. Pengujian ini dilakukan dalam siklus penuh TDD pada lingkungan aplikasi Android, yang memungkinkan untuk membandingkan dampak TDD terhadap performa perangkat lunak secara langsung dan memberikan hasil yang lebih terukur dibandingkan pendekatan berbasis survei atau literatur sebelumnya.

# BAB III
# METODE PENELITIAN

Pada Bab 3 ini, akan dijelaskan metodologi penelitian yang digunakan dalam penelitian ini, termasuk langkah-langkah yang diambil untuk menganalisis dampak penerapan *Test-Driven Development* (TDD) terhadap *testability* kode pada aplikasi Android Sedonor. Penelitian ini mengadopsi pendekatan eksperimen, yang mencakup pengukuran metrik kode sebelum dan setelah penerapan TDD, serta analisis kualitatif melalui wawancara dengan ahli untuk menilai perbaikan yang terjadi.

## 3.1 Alur Proses

Penelitian ini mengikuti alur proses yang digambarkan dalam diagram alur pada Gambar 3.1, yang terdiri dari tiga tahapan utama. Tahap pertama adalah pemilihan metrik dan analisis kode lama. Studi literatur dilakukan untuk mengidentifikasi metrik yang relevan dan metode pengukuran yang tepat. Setelah itu akan dilakukan penghitungan metrik-metrik terhadap kode lama. Tahap kedua adalah pengembangan ulang aplikasi dengan pendekatan TDD, di mana proses pengujian dilakukan secara berulang melalui siklus TDD: menulis tes, menulis kode untuk lulus tes, dan melakukan *refactoring*. Tahap ketiga adalah analisis pengaruh TDD dengan menghitung metrik terhadap kode baru dan menganalisis perbandingannya terhadap kode lama. Setiap tahap dalam proses ini akan dibahas secara mendetail dalam bab ini, namun fokus akan diberikan pada analisis perbandingan metrik *testability* antara pendekatan non-TDD dan TDD




Gambar 3.1 Diagram Alur

## 3.2 Pemilihan Metrik dan Analisis Kode Lama

Subbab ini menjelaskan langkah awal dalam proses analisis terhadap kualitas kode sebelum penerapan *Test-Driven Development* (TDD). Tahapan yang dibahas meliputi pemilihan metrik yang relevan untuk mengukur aspek *testability* dan kompleksitas kode, pemilihan studi kasus berupa fitur-fitur tertentu dalam aplikasi yang akan dianalisis, serta proses perhitungan metrik terhadap kode lama yang dikembangkan tanpa pendekatan TDD.

### 3.2.1 Studi Literatur Metrik

Studi literatur merupakan metode penting dalam tahap awal penelitian ini, yang digunakan untuk mengidentifikasi dan memilih metrik yang paling relevan dalam mengukur *testability* kode pada perangkat lunak. Melalui eksplorasi terhadap jurnal ilmiah, artikel akademik, dan publikasi terpercaya, peneliti bertujuan untuk menemukan metrik yang tidak hanya populer digunakan dalam penelitian sebelumnya, tetapi juga memiliki justifikasi empiris yang kuat sebagai indikator *testability*.

Salah satu referensi utama yang dijadikan dasar pemilihan metrik adalah penelitian oleh Alenezi (2022). Dalam studinya, Alenezi melakukan pendekatan berbasis *data mining* untuk mengevaluasi puluhan metrik perangkat lunak, dan menghasilkan peringkat 10 metrik terbaik yang paling berpengaruh terhadap *testability*. Proses evaluasi dilakukan menggunakan berbagai metode seleksi atribut, seperti *Gain Ratio, Information Gain, Symmetrical Uncertainty,* dan *OneR*, sehingga daftar yang dihasilkan bersifat objektif dan berbasis statistik.

Dari hasil perangkingan diperoleh sepuluh metrik teratas, yaitu:

1. NBI (*Number of Base Classes Inherited*)
2. RFC (*Response for Class*)
3. LOC (*Lines of Code*)
4. LOCCOM (*Lines of Comment*)
5. WMC (*Weighted Methods per Class*)
6. NMC (*Number of Method Calls*)
7. NMCE (*Number of Method Calls to External Classes*)
8. CAM (*Cohesion Among Methods of Class*)
9. T-NMC (*Total Number of Method Calls*)
10. NPM (*Number of Public Methods*)

Kesepuluh metrik ini direkomendasikan oleh Alenezi sebagai indikator utama dalam mengukur sejauh mana kode program dapat diuji secara efektif. Meskipun demikian, dalam konteks penelitian ini, penulis hanya menggunakan enam dari sepuluh metrik tersebut. Kelima metrik yang dipilih adalah:

1. RFC (*Response for Class*),
2. LOC (*Lines of Code*),
3. LOCCOM (*Lines of Comment*),
4. WMC (*Weighted Methods per Class*), dan
5. NMC (*Number of Method Calls*).

Pemilihan lima metrik ini dilakukan dengan mempertimbangkan ketersediaan alat bantu pengukuran, yaitu *CK Library*, sebuah *library* analisis statis yang mendukung perhitungan otomatis untuk metrik-metrik tertentu. *CK Library* terbukti mampu menghitung RFC, LOC, LOCCOM, WMC, dan NMC secara langsung dari *codebase*. Sementara itu, metrik lain seperti NBI, CAM, NMCE, T-NMC, dan NPM tidak didukung secara langsung oleh *CK Library* (Aniche, 2019), sehingga dikeluarkan dari ruang lingkup pengukuran untuk menjaga konsistensi metodologis dan validitas hasil.

Untuk memperjelas cakupan metrik yang didukung oleh CK *Library* dan yang digunakan dalam penelitian ini, berikut disajikan Tabel 3.1 sebagai referensi. Tabel ini menggambarkan metrik yang dipilih untuk mengukur *testability* kode pada aplikasi Sedonor.

Tabel 3.1 Cakupan Metrik CK dan Metrik Alenezi

| **Metrik Yang Disarankan Penelitian Alenezi** | **Didukung CK *Library*** |
|---|---|
| NBI (*Number of Base Classes Inherited*) | Tidak |
| RFC (*Response for a Class*) | Ya |
| LOC (*Lines of Code*) | Ya |
| LOCCOM (*Lines of Comment*) | Ya |
| WMC (*Weighted Methods per Class*) | Ya |
| NMC (*Number of Method Calls*) | Ya |
| NMCE (*Number of Method Calls to External Classes*) | Tidak |
| CAM (*Cohesion Among Methods of Class*) | Tidak |
| T-NMC (*Total Number of Method Calls*) | Tidak |
| NPM (*Number of Public Methods*) | Tidak |

Setelah ditentukan lima metrik yang akan digunakan dalam penelitian ini, langkah selanjutnya adalah memahami secara konseptual fungsi dan karakteristik masing-masing metrik tersebut. Penjelasan tiap metrik bertujuan untuk memberikan gambaran mengenai peran metrik dalam mengukur aspek-aspek penting dari *testability*, seperti kompleksitas, keterbacaan, dan keterhubungan antar komponen dalam kode program. Berikut adalah uraian lengkap mengenai kelima metrik yang digunakan dalam penelitian ini:

1. *Response for Class* (RFC)

RFC menghitung jumlah total metode unik yang dapat dipanggil sebagai respon dari pesan yang diterima oleh suatu kelas. Ini termasuk metode di dalam kelas itu sendiri dan metode yang dipanggil melalui objek lain.

2. *Lines of Code* (LOC)

   LOC menghitung jumlah total baris kode dalam sebuah kelas atau metode, tidak termasuk baris kosong dan komentar. LOC sering digunakan sebagai indikator awal dari kompleksitas kode; semakin banyak baris kode, semakin tinggi kemungkinan adanya kesalahan atau *bug* yang sulit ditemukan.

3. *Lines of Comment* (LOCCOM)

   LOCCOM mengukur jumlah baris yang digunakan untuk komentar dalam kode sumber. Metrik ini memberikan gambaran tentang dokumentasi dan keterbacaan kode. Kode yang terdokumentasi dengan baik lebih mudah dipahami dan diuji.

4. *Weight Method per Class* (WMC)

   WMC menghitung jumlah total kompleksitas dari semua metode dalam sebuah kelas. Kompleksitas metode biasanya diukur menggunakan kompleksitas McCabe atau jumlah instruksi logika dalam metode tersebut. Semakin tinggi WMC, semakin sulit untuk menguji kelas tersebut karena kompleksitas logika yang meningkat.

5. *Number of Method Calls* (NMC)

   NMC menghitung jumlah panggilan metode yang dilakukan oleh suatu kelas. Metrik ini memberikan gambaran tentang seberapa sering kelas atau metode tertentu berinteraksi dengan bagian lain dari sistem. Semakin banyak panggilan metode, semakin tinggi kebutuhan untuk pengujian yang komprehensif.

Simulasi pengukuran dari metrik-metrik tersebut dapat dilihat dari studi kasus pada potongan kode di Gambar 3.2. Gambar ini memberikan ilustrasi mengenai penerapan metrik dalam analisis kode pada aplikasi Sedonor.



```java
class Person {
        protected String nama;
        protected String alamat;

        public Person(String nama, String alamat) {
            this.nama = nama;
            this.alamat = alamat;
        }

        public void hello() {
            System.out.println("Hello namaku " + nama);
        }
    }

    class Mahasiswa extends Person {
        private String nim;

        Mahasiswa(String nama, String alamat) {
            super(nama, alamat);
        }

        public void setNim(String nim) {
            this.nim = nim;
        }

        public void dataDiri() {
            System.out.println(String.format(format:"Nama : %s\nNIM : %s",nama,nim));
        }

    }



public class ProjectOOP1 {


    Run | Debug
    public static void main(String[] args) {
        // TODO code application logic here
        Person orang = new Person(nama:"Jonas",alamat:"Batam");
        orang.hello();

        Person orang2 = new Person(nama:"Dorothy",alamat:"Tulungagung");
        orang2.hello();

        Mahasiswa mhs1 = new Mahasiswa(nama:"Jahames",alamat:"Denpasar");
        mhs1.setNim(nim:"215232776");
        mhs1.dataDiri();
        mhs1.hello();
    }

}
```

Gambar 3.2 Potongan Kode Simulasi Perhitungan Metrik

Dari kode ini, hasil pengukuran metrik dapat dilihat pada Tabel 3.2. Tabel ini menunjukkan nilai metrik yang diperoleh dari analisis kode dalam aplikasi Sedonor.

Tabel 3.2 Hasil Perhitungan Metrik Untuk Kode Simulasi

| Metrik | Hasil Metrik | Penjelasan |
|---|---|---|
| *Response for Class* (RFC) | 6 | Hasil ini didapat dari *method* unik yang dipanggil dari masing-masing kelas. Kelas *Person* memiliki 2, kelas |

| | | |
|---|---|---|
| | | *Mahasiswa* memiliki 1, dan kelas *ProjectOOP1* memiliki 1. |
| *Lines of Code*s (LOC) | 35 | LOC mengacu pada keseluruhan baris kode yang bukan merupakan baris kosong dan komen. |
| *Lines of Comment* (LOCCOM) | 1 | Hasil ini mengacu pada jumlah total baris komentar yang ditulis dalam kode program. |
| *Weight Method per Class* (WMC) | 6 | Hasil ini didapat dari jumlah instruksi cabang (*branch*) dalam metode-metode di setiap kelas. Jika tidak ada cabang, kompleksitas minimal adalah 1 untuk setiap metode. Terlihat dari kelas *Person* memiliki 2, kelas *Mahasiswa* memiliki 3 dan kelas *ProjectOOPJava1* memiliki 1. |
| *Number of Method Calls* (NMC) | 5 | NMC dihitung dari seberapa banyak method dipanggil dalam suatu kelas. Terdapat 5 panggilan metode di kelas *ProjectOOP1*. |

### 3.2.2 Studi Literatur TDD

Pada subbab ini, studi literatur dilakukan untuk mengulas konsep dan implementasi *Test-Driven Development* (TDD) berdasarkan literatur yang relevan. *Test-Driven Development* (TDD) adalah metode pengembangan perangkat lunak yang menekankan pada penulisan tes terlebih dahulu sebelum menulis kode yang sebenarnya (Beck, 2003). Prinsip dasar dari *Test-*

*Driven Development* adalah siklus "*Red-Green-Refactor*". *Red* mengacu pada kondisi di mana kode tes yang awalnya dibuat oleh pengembang menghasilkan eror di konsol. *Green* menggambarkan kondisi ketika kode yang ditulis berhasil melewati seluruh rangkaian tes tanpa menghasilkan eror. *Refactor* adalah langkah untuk mengoptimalkan kode yang telah dibuat agar lebih rapi dan mudah dibaca. Dalam praktiknya, pengembang biasanya memulai dengan membuat tes yang gagal, kemudian melanjutkan dengan menulis kode seminimal mungkin agar tes tersebut berhasil, dan terakhir melakukan *refactoring* kode untuk meningkatkan desain dan keterbacaan kode (Vijaywargi & Boddapati, 2024).

### 3.2.3 Pemilihan Studi Kasus

Pada subbab ini, penulis menjelaskan proses pemilihan aplikasi yang digunakan sebagai studi kasus dalam penelitian. Pemilihan dilakukan melalui dua tahap utama, yaitu perbandingan antar kandidat aplikasi dan penetapan objek utama studi kasus. Setelah aplikasi yang paling sesuai ditentukan, penulis juga membatasi ruang lingkup analisis dengan memilih sejumlah fitur dari aplikasi tersebut yang akan digunakan dalam pengukuran metrik *testability* dan proses *refactoring*. Pembahasan dalam subbab ini dibagi ke dalam dua bagian: perbandingan antara kandidat aplikasi, serta penentuan enam fitur dari Sedonor yang akan dianalisis lebih lanjut dalam penelitian ini.

#### A. Perbandingan Studi Kasus

Setelah melakukan pemilihan metrik, penulis terlebih dahulu melakukan identifikasi terhadap beberapa aplikasi yang memungkinkan untuk diteliti. Pemilihan ini didasarkan pada pertimbangan praktis, yaitu kemudahan akses terhadap kode sumber, tingkat kompleksitas aplikasi, serta kesesuaian dengan topik penelitian yang mengharuskan aplikasi berbasis *mobile* dan dapat dijalankan pada perangkat Android. Dari proses penjajakan awal ini, terdapat dua aplikasi yang layak untuk dipertimbangkan, yaitu ThinkMate dan Sedonor.

ThinkMate adalah aplikasi berbasis *multi-platform* yang dikembangkan menggunakan bahasa pemrograman Go. Aplikasi ini digunakan sebagai sarana bantu edukasi daring, dengan sejumlah fitur yang tergolong kompleks seperti autentikasi JWT (*JSON Web Token*), pengelolaan data pengguna, serta koneksi ke *remote database*. Di sisi arsitektur, ThinkMate telah menerapkan *layered architecture* yang memisahkan tanggung jawab logika aplikasi dalam lapisan *controller*, *service*, dan *data access*. Kode sumbernya dapat diakses secara

internal sehingga memungkinkan dilakukan pengukuran metrik dan eksplorasi kode secara menyeluruh.

Sedonor, di sisi lain, merupakan aplikasi Android *native* yang dikembangkan dengan bahasa Kotlin. Aplikasi ini berfokus pada layanan donor darah, dan memiliki fitur-fitur utama seperti informasi lokasi donor, riwayat donor, artikel kesehatan, *chatbot*, pengingat kesehatan, dan sistem *check-in*. Semua fitur tersebut saling terhubung dan memanfaatkan berbagai layanan pihak ketiga seperti Firebase, API lokasi, dan notifikasi yang membuat kompleksitas aplikasi ini tinggi. Karena dikembangkan dalam konteks proyek akademik, kode sumber Sedonor dapat diakses secara penuh oleh penulis.

Untuk mendalami perbedaan karakteristik keduanya, disusun Tabel 3.3 berikut sebagai landasan analisis. Tabel ini tidak hanya menyajikan deskripsi teknis masing-masing aplikasi, tetapi juga digunakan sebagai dasar untuk membandingkan kesesuaian keduanya terhadap pendekatan *Test-Driven Development* (TDD). Dengan melihat aspek-aspek seperti bahasa pemrograman, paradigma, arsitektur, serta potensi *refactoring* dan dukungan terhadap *unit test*, penulis dapat mengevaluasi sejauh mana masing-masing aplikasi dapat mendukung proses penerapan TDD secara optimal.

Tabel 3.3 Karakteristik Teknik ThinkMate dan Sedonor

| **Aspek** | **Thinkmate** | **Sedonor** |
|---|---|---|
| Bahasa Pemrograman | Go | Kotlin |
| Paradigma Pemrograman | Prosedural | *Object-Oriented Programming* (OOP) |
| Platform | *Multi-platform* (Web dan Android) | *Mobile* (Android) |
| Kompleksitas Fitur | Tinggi (JWT, *Remote DB*, autentikasi) | Tinggi (Firebase, API lokasi, notifikasi) |
| Arsitektur | *Layered architecture* | Tidak menggunakan arsitektur spesifik |
| Peluang *Refactoring* | Terbatas (struktur sudah stabil) | Tinggi (fleksibel dan mudah dikembangkan) |

| | | |
|---|---|---|
| Dukungan terhadap TDD | Terbatas (tidak menggunakan paradigma OOP) | Sangat mendukung (menggunakan paradigma OOP) |

Berdasarkan Tabel 3.3, dapat dilihat bahwa secara teknis, kedua aplikasi memiliki tingkat kompleksitas fitur yang tinggi. ThinkMate memiliki integrasi autentikasi JWT dan *remote database*, sedangkan Sedonor memanfaatkan Firebase dan API layanan lokasi, yang juga menambah kompleksitas sistem. Namun, kompleksitas semata tidak cukup untuk mendukung praktik TDD jika tidak dibarengi dengan struktur dan paradigma yang sesuai.

Dalam hal bahasa dan paradigma pemrograman, ThinkMate menggunakan Go yang bersifat prosedural, sementara Sedonor menggunakan Kotlin yang berbasis OOP. Hal ini menjadi faktor penting, karena TDD secara prinsip lebih cocok diterapkan pada sistem yang modular dan berbasis OOP. Struktur OOP memungkinkan pembuatan *unit test* yang lebih terisolasi, terpisah antar kelas, dan mudah untuk diuji secara otomatis.

Pada aspek arsitektur, ThinkMate telah menerapkan *layered architecture* yang stabil dan tersusun rapi. Namun kondisi ini justru membuat fleksibilitas perubahan dan *refactoring* menjadi terbatas, karena struktur kode telah terbentuk dengan jelas dan terikat antar lapisan. Di sisi lain, meskipun Sedonor tidak menerapkan arsitektur khusus secara eksplisit, fleksibilitas struktur dan pendekatan pengkodean yang digunakan memberikan ruang lebih luas untuk *refactoring*.

Dari sisi dukungan terhadap TDD, ThinkMate tidak menggunakan pendekatan OOP dan tidak dilengkapi dengan *framework unit testing*, sehingga kurang mendukung pengujian di level *unit*. Sebaliknya, Sedonor sangat mendukung penerapan TDD karena sejak awal dibangun dengan prinsip OOP dan dapat dikombinasikan dengan alat uji bawaan Kotlin seperti JUnit.

Oleh karena itu, meskipun keduanya memenuhi kriteria awal berupa kompleksitas dan aksesibilitas kode sumber, Sedonor dipilih sebagai studi kasus utama dalam penelitian ini karena lebih sesuai dengan pendekatan TDD, baik dari segi arsitektur, paradigma pemrograman, maupun potensi eksplorasi pengujian dan *refactoring*.

## B. Penentuan Fitur Sedonor yang Dianalisis

Aplikasi Sedonor yang digunakan sebagai objek studi kasus dalam penelitian ini merupakan aplikasi donor darah berbasis Android yang dikembangkan menggunakan bahasa pemrograman Kotlin. Aplikasi ini memiliki total delapan fitur utama, yaitu:

1. Info tempat donor
2. Artikel kesehatan
3. *Chatbot* Sedonor
4. Riwayat donor darah
5. Pengingat kesehatan
6. *Check-in* donor
7. Kalender donor
8. Autentikasi donor

Seluruh fitur tersebut berkontribusi dalam mendukung proses bisnis dari dua aktor utama, yaitu pendonor dan petugas PMI. Alur kerja dari kedua aktor ini tercermin pada *usecase* diagram Sedonor.

Namun, dalam penelitian ini, tidak seluruh fitur dianalisis. Penulis memilih enam dari delapan fitur untuk dianalisis dan diterapkan pengembangan ulang menggunakan metode *Test-Driven Development* (TDD). Enam fitur yang dipilih adalah:

1. Info tempat donor
2. Artikel kesehatan
3. *Chatbot* Sedonor
4. Riwayat donor darah
5. Pengingat kesehatan
6. *Check-in* donor

Pemilihan fitur-fitur tersebut didasarkan pada tingkat kompleksitas dan keterkaitannya dengan layanan pihak ketiga. Lima fitur awal mengandalkan Firebase sebagai *remote database* untuk keperluan penyimpanan dan sinkronisasi data, sementara fitur *chatbot* terhubung secara langsung dengan API milik OpenAI. Integrasi dengan layanan eksternal ini memberikan tantangan teknis tambahan, terutama dalam hal pengolahan data, komunikasi jaringan, serta implementasi logika bisnis. Seluruh aspek tersebut berada dalam area yang sangat relevan untuk diuji menggunakan pendekatan *Test-Driven Development* (TDD) dan *unit testing*, mengingat pentingnya isolasi dan keterpisahan logika dalam proses pengujian.

Sementara itu, dua fitur lainnya, yaitu kalender donor dan autentikasi donor, hanya berkaitan dengan tampilan antarmuka (UI) dan tidak memiliki logika kompleks atau interaksi yang berarti dengan *backend* maupun dependensi eksternal. Mengingat bahwa fokus utama TDD berada pada pengujian logika bisnis melalui *unit test* dan bukan pada pengujian UI, maka dua fitur tersebut tidak disertakan dalam ruang lingkup analisis.

Dengan demikian, keenam fitur yang dipilih dianggap mewakili bagian-bagian penting dari sistem Sedonor. Hal ini dianggap mampu menunjukkan pengaruh TDD secara signifikan terhadap *testability* kode, terutama pada area yang berkaitan dengan logika aplikasi dan dependensi eksternal.

### 3.2.4 Perhitungan Metrik Kode Lama

Pengukuran nilai metrik *testability* pada kode program aplikasi Sedonor versi awal (sebelum penerapan TDD) dilakukan dengan memanfaatkan alat bantu analisis statis otomatis bernama *CK Library*. *CK Library* merupakan sebuah *tools open source* berbasis Java yang dikembangkan oleh Mauricio Aniche. Alat ini dirancang untuk menghitung berbagai metrik perangkat lunak berorientasi objek secara otomatis dengan melakukan analisis statis terhadap kode sumber. *CK Library* mampu menghasilkan laporan dalam bentuk *file* CSV yang berisi informasi detail terkait metrik pada level kelas, metode, dan atribut dalam sebuah proyek perangkat lunak (Aniche, 2019).

Pemilihan *CK Library* dilakukan setelah penulis melakukan penelusuran terhadap berbagai *tools* otomasi yang dapat digunakan untuk menghitung metrik-metrik *testability*. Alat ini dipilih karena mendukung bahasa pemrograman Kotlin, yang menjadi dasar pengembangan aplikasi Sedonor.

Namun, dari hasil pencarian tersebut, tidak ditemukan *tools* yang secara langsung mendukung perhitungan metrik *testability* berbasis kode sumber Kotlin. Untuk mengatasi hal ini, peneliti memilih pendekatan alternatif dengan menggunakan bahasa Java, yang memiliki struktur sintaksis yang serupa dan kompatibel dengan Kotlin. Selain itu, Android Studio menyediakan fitur dekompilasi *bytecode* Kotlin menjadi kode Java, sehingga proses konversi dapat dilakukan dengan praktis tanpa mengubah struktur logika program.

Setelah seluruh proses dekompilasi selesai, kode program dalam bentuk Java dianalisis menggunakan *CK Library*. Metrik yang dihitung meliputi *Response for a Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), dan *Number of Method Calls* (NMC), sebagaimana telah dijelaskan pada subbab sebelumnya.

Untuk menganalisis hasil metrik tersebut, peneliti mengacu pada ambang batas yang telah ditetapkan oleh Chidamber dan Kemerer (1994). Ambang batas ini digunakan sebagai standar evaluasi untuk menentukan apakah suatu kelas tergolong kompleks, terdokumentasi dengan baik, atau memiliki tingkat keterujian yang rendah. Tabel 3.4 berikut menunjukkan ambang batas yang digunakan:

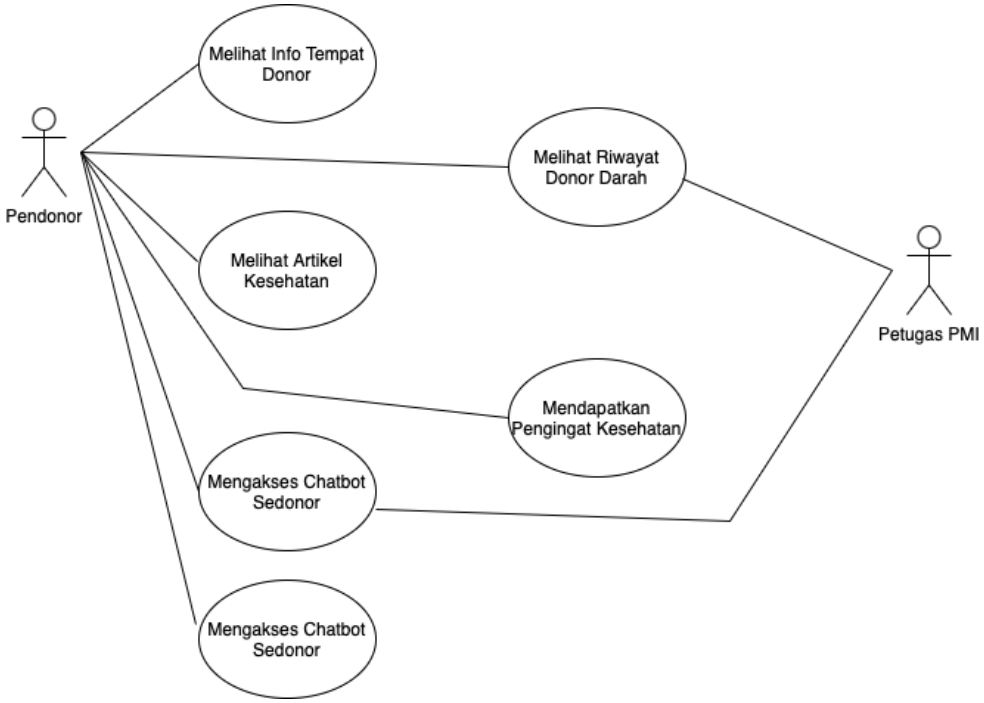
Tabel 3.4 Ambang Batas Metrik

| Metrik | Ambang Batas | Interpretasi |
|---|---|---|
| RFC | Lebih kecil dari atau sama dengan 100. | Jika lebih dari 100, kode dianggap terlalu kompleks karena terlalu banyak metode yang dipanggil. |
| LOC | Lebih kecil dari 200 per fitur. | Lebih dari 200 menunjukkan potensi *code smell* dan rendahnya modularitas. |
| LOCCOM | Tidak ada ambang batas, tapi idealnya 20-40% dari LOC. | Komentar rendah menunjukkan dokumentasi buruk, terlalu banyak dapat berarti kode sulit dipahami. |
| WMC | Lebih kecil dari atau sama dengan 50. | Nilai di atas 50 menandakan kompleksitas tinggi dan sulit diuji. |
| NMC | Lebih kecil dari 100. | Semakin tinggi nilainya, *testability* semakin rendah. |

Dengan pendekatan ini, peneliti memperoleh *baseline* kuantitatif dari struktur kode sebelum penerapan TDD. Hasil ini selanjutnya akan dibandingkan dengan kode baru setelah TDD untuk menilai perubahan *testability* yang terjadi.

### 3.3 Pelaksanaan TDD

Pelaksanaan TDD, seperti yang telah ditentukan sebelumnya, akan diterapkan pada aplikasi Sedonor. Proses pengembangan ulang akan dilakukan terhadap enam fitur Sedonor yang telah ditentukan sebelumnya. Dalam enam fitur ini, terdapat dua aktor yang melaksanakan proses bisnis, yaitu pendonor dan petugas PMI. Diagram *usecase* Sedonor dapat dilihat pada Gambar 3.3.




Gambar 3.3 Diagram *Usecase*

Masing-masing *actor* memiliki peran dan tujuan berbeda dalam menggunakan aplikasi. Peran masing-masing *actor* dapat dilihat lebih jelas di Tabel 3.5.

Tabel 3.5 Pemetaan *Usecase*

| Aktor | Peran |
|---|---|
| Pendonor | 1. Melihat artikel kesehatan<br>2. Melihat riwayat donor darah<br>3. Melihat info tempat donor<br>4. Mengakses *chatbot* sedonor<br>5. Melakukan *check-in* donor<br>6. Mendapatkan pengingat kesehatan |
| Petugas PMI | 1. Melihat riwayat donor darah pendonor<br>2. Mengakses *chatbot* sedonor |

### 3.3.1 Pembuatan *Unit test*

Pada tahap awal penerapan TDD, penulis harus membuat *unit test* yang sesuai dengan kebutuhan aplikasi. Kebutuhan aplikasi ini diperoleh dari diagram *usecase* serta analisis kode lama aplikasi.

Selanjutnya, pada penelitian ini, tahap pembuatan *unit test* menjadi langkah penting karena memastikan bahwa setiap fitur yang dikembangkan memiliki pengujian yang memadai sebelum implementasi lebih lanjut. *Unit test* dirancang untuk menguji fungsionalitas terkecil

dari aplikasi, sehingga dapat mengurangi masalah seminimal mungkin. Hasil dari tahap ini adalah keluaran eror pada *terminal* karena belum ada kode yang ditulis untuk memenuhi *unit test* yang telah dibuat.

Kode *unit test* akan dibuat dengan menggunakan *plugin* JUnit, yang merupakan alat pengujian bawaan dalam Android Kotlin. *Plugin* ini memungkinkan pengujian unit dilakukan secara otomatis dan efisien.

### 3.3.2 Implementasi Kode Program

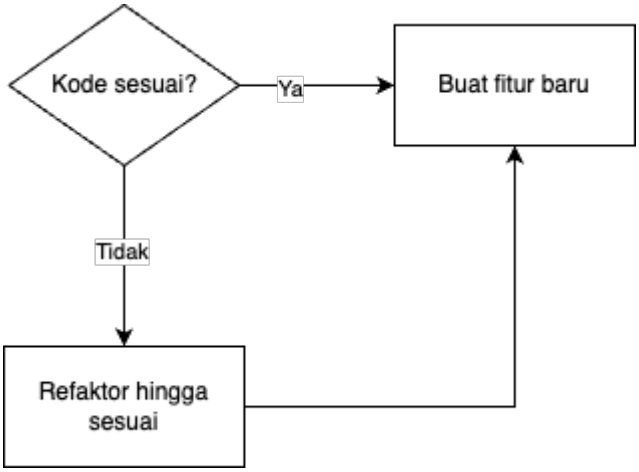
Tahap selanjutnya adalah implementasi kode program. Tahap ini bertujuan untuk memastikan bahwa *unit test* yang telah dibuat sebelumnya dapat terpenuhi, sehingga tidak ada lagi pesan eror yang muncul di *terminal*. Fokus utama dalam tahap ini adalah membuat kode yang dapat melewati semua *unit test*. Oleh karena itu, pada tahapan ini mungkin akan banyak penulisan kode secara cepat dan kurang terstruktur, tanpa memperhatikan aspek-aspek seperti *clean coding* atau prinsip-prinsip lain yang biasanya diterapkan dalam penulisan kode (Naufal et al., 2023). Meskipun demikian, pendekatan ini diperlukan untuk memastikan bahwa fungsionalitas inti dari aplikasi berfungsi dengan baik sebelum melakukan *refactoring*.

Implementasi kode program ini akan dilakukan menggunakan bahasa pemrograman Kotlin, sesuai dengan bahasa utama yang digunakan dalam pengembangan aplikasi sebelumnya. Penggunaan Kotlin memastikan kesesuaian dengan struktur dan arsitektur aplikasi yang sudah ada.

### 3.3.3 *Refactor*

Tahap terakhir dalam metode TDD adalah *refactoring*. Pada tahap ini, tujuan utamanya adalah "membersihkan" kode agar menjadi lebih teroptimasi dan mudah untuk dipelihara. Di sinilah pengembang mulai menerapkan prinsip-prinsip *clean coding* untuk merapikan kode.

Tahap-tahap dalam proses refactoring kode dapat dilihat pada Gambar 3.4. Gambar ini menggambarkan langkah-langkah yang diambil dalam melakukan refactoring pada kode aplikasi Sedonor.




Gambar 3.4 Diagram *Refactor*

Tahap pertama adalah menilai apakah kode yang telah dibuat sudah terlihat baik. Jika kode tersebut dianggap baik, proses dapat dilanjutkan ke fitur berikutnya. Namun, jika kode dianggap kurang baik, maka perlu dilakukan *refactoring* sebelum melanjutkan ke fitur berikutnya. Pengecekan kualitas kode ini dilakukan melalui dua cara: *code review* dan penggunaan alat bantu. Dalam *code review*, pengembang akan menganalisis kode secara mendalam. Alat bantu yang digunakan adalah SonarLint, sebuah ekstensi yang berfungsi untuk membantu mengoptimalkan kode (SonarSource, 2017). SonarLint bekerja dengan menganalisis keseluruhan kode dan memberikan saran apabila terdapat kode yang dapat dioptimalkan lebih lanjut sesuai dengan prinsip *clean architecture* (Martin, 2009). Hal ini sangat berguna sebagai alat bantu dalam tahap *refactor*.

## 3.4 Analisis Pengaruh TDD

Pada tahap ini, akan dilakukan pengukuran nilai metrik terhadap kode baru, yang selanjutnya akan dianalisis dengan membandingkannya terhadap hasil pengukuran kode lama. Setelah itu, proses evaluasi akan dilengkapi melalui *expert review* untuk menilai kualitas kode secara kualitatif.

### 3.4.1 Perhitungan Metrik Terhadap Kode Baru

Penghitungan metrik terhadap kode baru akan dilakukan dengan pendekatan yang sama seperti pada kode lama. Aplikasi Sedonor berbasis Android tetap menjadi objek pengukuran, dengan tujuan menilai performa sistem yang baru dalam hal *testability* kode. Pengujian ini kembali dilakukan menggunakan *CK Library*. Hasil pengukuran ini akan dibandingkan dengan

data pengukuran dari kode lama untuk menganalisis perbedaan performa dan efektivitas *testability* kode yang diterapkan.

### 3.4.2 Perbandingan Hasil Metrik Kode Lama dan Baru

Setelah hasil pengukuran dari kode baru diperoleh, data tersebut akan dianalisis dan dibandingkan dengan hasil pengukuran kode lama. Tujuan dari perbandingan ini adalah untuk mengevaluasi sejauh mana perbaikan yang telah dilakukan meningkatkan performa dan *testability* kode. Analisis akan mencakup perbedaan hasil dari masing-masing metrik LOC, RFC, LOCCOM, WMC, dan NMC antara kode lama dan baru. Dengan demikian, perbedaan performa antara kode lama dan kode baru dapat diidentifikasi, serta efektivitas *refactoring* dalam meningkatkan kualitas kode dapat diukur secara objektif. Hasil akhir dari perbandingan ini akan memberikan wawasan penting untuk pengembangan lebih lanjut dan memastikan bahwa perubahan yang dilakukan berdampak positif pada kualitas keseluruhan kode.

### 3.4.3 *Expert review* Terhadap Kode Lama dan Baru

Selain perbandingan secara kuantitatif, penelitian ini juga akan melibatkan *expert review* untuk mengevaluasi kode lama dan kode baru berdasarkan aspek *clean architecture*, modularitas, dan *testability*. *Expert review* merupakan metode evaluasi yang melibatkan individu dengan keahlian teknis khusus untuk menilai kualitas suatu artefak, seperti kode program, antarmuka, atau dokumen desain. Proses ini banyak digunakan dalam rekayasa perangkat lunak maupun *usability engineering* untuk mengidentifikasi kelemahan dan memberikan saran perbaikan berdasarkan standar atau prinsip yang telah disepakati. Dalam konteks penelitian ini, pendekatan *expert review* mengikuti prinsip-prinsip serupa seperti *heuristic evaluation* yang diperkenalkan oleh (Nielsen, 1994), namun diaplikasikan pada tinjauan kualitas kode.

Pada tahap ini, wawancara akan dilakukan terhadap tiga responden yang telah dipilih berdasarkan kriteria tertentu, yaitu: memiliki pengalaman minimal lima tahun dalam pengembangan perangkat lunak berbasis *mobile*, khususnya dalam penerapan *clean architecture* dan modularitas, serta pernah terlibat dalam evaluasi dan *review* kode. Pertanyaan yang diajukan akan mencakup konsep *clean architecture*, modularitas kode, dan *testability* kode.

Beberapa pertanyaan yang diajukan yaitu:

1. Bagaimana responden menilai penerapan clean architecture pada kode ini.

2. Apakah struktur kode sudah memisahkan komponen sesuai dengan tanggung jawabnya.
3. Sejauh mana kode terstruktur secara modular.
4. Bagaimana nilai keterbacaan kode tersebut.
5. Apakah setiap modul memiliki tanggung jawab yang jelas dan terdefinisi.
6. Apakah kode ini mudah diuji secara unit.
7. Serta apa perbedaan utama antara kode lama dan baru dalam hal *clean architecture, modularitas,* dan *testability*.

Responden juga akan diminta untuk menilai apakah kode baru menunjukkan perbaikan yang signifikan dibandingkan kode lama. Dari hasil wawancara ini, akan disimpulkan perbandingan kualitas antara kode lama dan kode baru.

# BAB IV
# HASIL DAN PEMBAHASAN

Pada bab ini menjelaskan tahap dari alur proses, yaitu pemilihan metrik dan analisis kode lama, penerapan TDD, dan analisis pengaruh TDD. Tahapan ini perhitungan metrik pada kode lama sesuai dengan metrik yang telah ditentukan di bab sebelumnya, serta implementasi TDD yang melibatkan pembuatan *unit test*, pengembangan kode program, dan *refactoring*. Selanjutnya, dilakukan analisis dampak penerapan TDD dengan menghitung metrik kualitas pada kode baru. Setelah itu, hasil perbandingan metrik antara kode lama dan kode baru akan dievaluasi melalui *expert review* guna mendapatkan perspektif lebih mendalam terhadap perubahan yang terjadi.

## 4.1 Perhitungan Metrik dan Analisis Kode Lama

Pada subbab ini, penulis akan melakukan perhitungan terhadap metrik-metrik yang telah dipilih, yaitu RFC, LOC, LOCCOM, NMC, dan WMC, untuk mengevaluasi kualitas dan kompleksitas kode lama. Setiap metrik akan dihitung untuk mengidentifikasi aspek-aspek seperti ukuran kode, jumlah metode, dan kompleksitas respon yang dimiliki oleh kelas-kelas dalam kode tersebut. Setelah itu, penulis akan menganalisis hasil perhitungan ini untuk menentukan area yang memerlukan perbaikan atau *refactoring* guna meningkatkan kualitas kode dan memudahkan pemeliharaan di masa depan.

### 4.1.1 Perhitungan Metrik Kode Lama

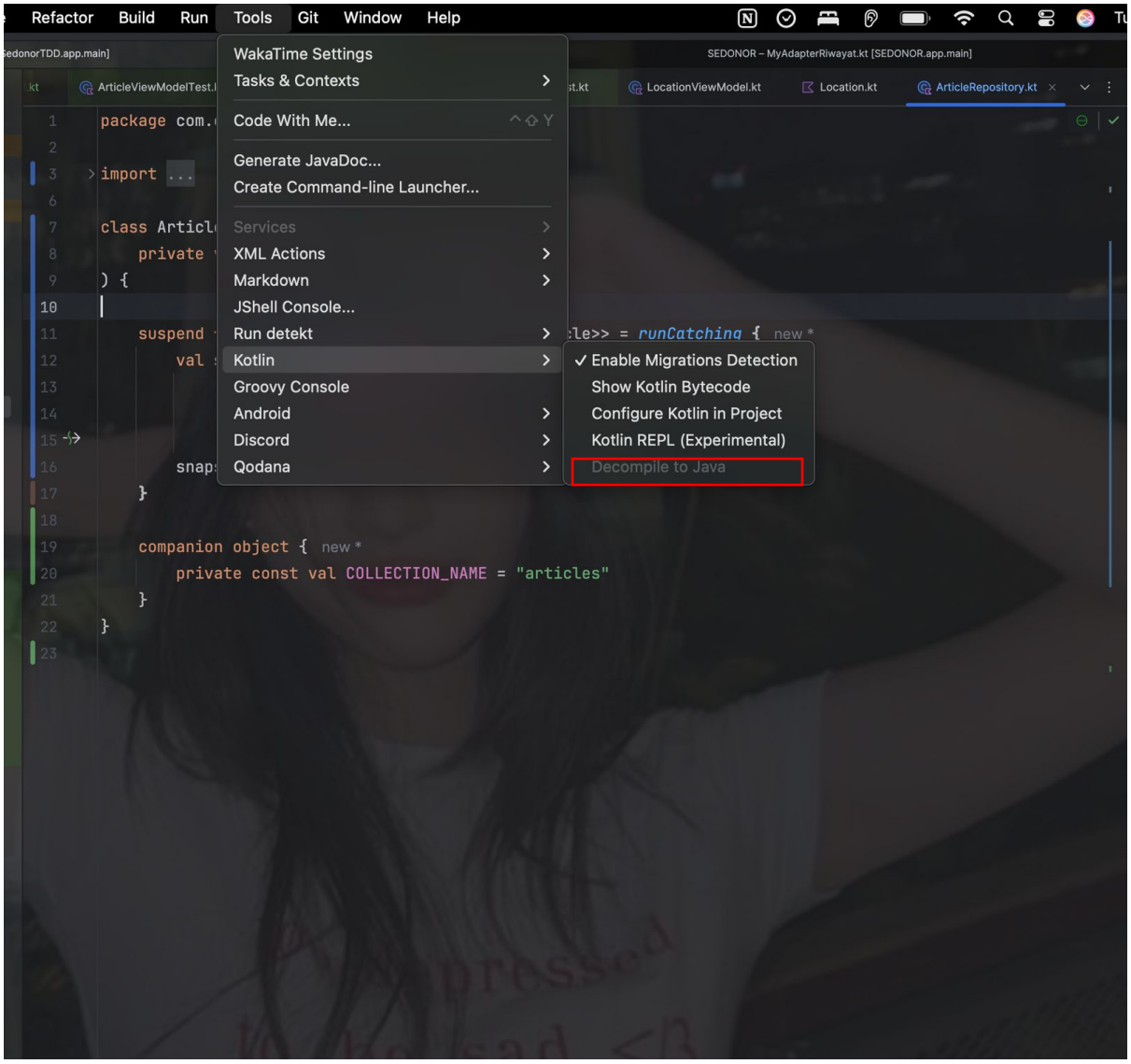
Pada tahap ini, dilakukan perhitungan terhadap kode lama untuk memperoleh nilai yang mencerminkan kualitasnya jika diukur menggunakan metrik *testability*. Proses ini bertujuan untuk memahami sejauh mana kode lama memenuhi aspek *testability* berdasarkan metrik yang digunakan. Hasil dari perhitungan ini nantinya akan menjadi acuan untuk dibandingkan dengan kode setelah penerapan TDD pada tahap akhir, sehingga dapat memberikan gambaran mengenai perubahan atau perbedaan yang terjadi.

Untuk melakukan perhitungan metrik *testability*, ada beberapa langkah yang perlu dilakukan. Pertama, kode yang ditulis dalam bahasa Kotlin akan diubah menjadi Java agar kompatibel dengan proses perhitungan. Selanjutnya, perhitungan metrik dilakukan

menggunakan *CK Library*, yang dirancang khusus untuk menghitung metrik Chidamber dan Kemerer.

**A. Pengubahan Kode dari Bahasa Kotlin menjadi Bahasa Java**

Sebelum melakukan perhitungan metrik, kode terlebih dahulu dikonversi dari Kotlin ke Java. Setiap kode yang termasuk dalam enam fitur yang telah ditentukan akan dikonversi tanpa terkecuali. Hal ini dilakukan karena *library* yang digunakan untuk perhitungan metrik ditulis dalam bahasa Java. Proses konversi dapat dilakukan dengan mudah menggunakan fitur bawaan Android Studio, yaitu *Decompile to Java*.




Gambar 4.1 Melakukan *Decompiling*

Kode program yang semula ditulis dalam bahasa Kotlin telah dikonversi ke dalam bahasa Java. Proses perubahan ini dapat dilihat pada Gambar 4.2 dan Gambar 4.3 dengan contoh kode untuk fitur melihat artikel.

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

}
```

Gambar 4.2 Kode sebelum di-*decompile* ke Java

```
public final class ArticleRepository {
   @NotNull
   private final FirebaseFirestore firestore;

   public ArticleRepository(@NotNull FirebaseFirestore firestore) {
      Intrinsics.checkNotNullParameter(firestore, "firestore");
      super();
      this.firestore = firestore;
   }

   @Nullable
   public final Object fetchArticles_IoAF18A/* $FF was: fetchArticles-
IoAF18A*/(@NotNull Continuation $completion) {
      Object $continuation;
      label46: {
         if ($completion instanceof <undefinedtype>) {
            $continuation = (<undefinedtype>)$completion;
            if         ((((<undefinedtype>)$continuation).label         &
Integer.MIN_VALUE) != 0) {
               ((<undefinedtype>)$continuation).label                  -=
Integer.MIN_VALUE;
               break label46;
            }
         }

         $continuation = new ContinuationImpl($completion) {
            // $FF: synthetic field
            Object result;
            int label;

            @Nullable
            public final Object invokeSuspend(@NotNull Object $result) {
               this.result = $result;
               this.label |= Integer.MIN_VALUE;
               Object  var10000  =  ArticleRepository.this.fetchArticles-
IoAF18A((Continuation)this);
               return var10000 == IntrinsicsKt.getCOROUTINE_SUSPENDED() ?
var10000 : Result.box-impl(var10000);
            }
         };
      }
```

Gambar 4.3 Kode setelah di-*decompile* ke Java

Seperti yang terlihat pada Gambar 4.2 dan Gambar 4.3, terdapat perbedaan jumlah baris kode setelah proses *decompile*. Namun, sebagaimana telah dijelaskan sebelumnya, perbedaan ini tidak memengaruhi kompleksitas kode secara signifikan sehingga tidak berdampak besar terhadap tingkat *testability*-nya.

## B. Perhitungan Metrik dengan *CK Library*

Setelah semua *file* di-*decompile* ke Java, perhitungan dilakukan terhadap metrik-metrik yang telah dipilih dengan mengetik *command "java -jar ck-0.7.1-SNAPSHOT-jar-with-dependencies.jar ./"* di *terminal* seperti yang dapat dilihat di Gambar 4.4. Langkah ini dilakukan untuk memberi tahu *CK Library* untuk menghitung nilai-nilai metrik di dalam *folder*.




Gambar 4.4 Menjalankan *Library* Lewat *Terminal*

Perhitungan metrik yang dilakukan menggunakan *CK Libray* secara otomatis menghasilkan *output file "class.csv"*. *File* ini berisi hasil analisis metrik dari masing-masing kelas dalam bentuk tabel. Data mentah tersebut kemudian difilter berdasarkan fitur yang relevan. Awalnya data terbagi per *file* kelas, lalu disusun ulang menjadi per fitur aplikasi seperti artikel kesehatan, riwayat donor, dan lainnya. Keluaran dari pengukuran *CK Library* dilihat lebih lanjut pada Lampiran A. Setelah proses penggabungan dan klasifikasi selesai, hasil akhirnya disajikan dalam bentuk ringkasan metrik per fitur yang ditampilkan pada Tabel 4.1.

Tabel 4.1 Hasil Perhitungan Metrik Kode Lama

| **Metrik** | **Hasil Pengukuran Info Tempat Donor** | **Hasil Pengukuran Artikel Kesehatan** | **Hasil Pengukuran Riwayat Donor Darah** | **Hasil Pengukuran *Chatbot* Sedonor** | **Hasil Pengukuran *Check-in* Donor** | **Hasil Pengukuran Pengingat Kesehatan** |
|---|---|---|---|---|---|---|
| RFC | 91 | 78 | 103 | 61 | 88 | 41 |
| LOC | 191 | 297 | 306 | 146 | 115 | 106 |
| LOCCOM | 20 | 17 | 6 | 6 | 24 | 3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| NMC | 50 | 35 | 56 | 75 | 35 | 8 |
| WMC | 113 | 55 | 149 | 139 | 59 | 12 |

Berdasarkan Tabel 4.1, hasil pengukuran metrik terhadap kode lama (sebelum penerapan TDD) menunjukkan bahwa sejumlah metrik pada beberapa fitur memiliki nilai yang melebihi ambang batas *best practice* yang disarankan oleh Chidamber dan Kemerer (1994), yang ditandai dengan warna merah pada tabel. Pada metrik *Response for class* (RFC), diperoleh nilai 91 (info tempat donor), 78 (artikel kesehatan), 103 (riwayat donor darah), 61 (*chatbot* sedonor), 88 (*check-in* donor), dan 41 (pengingat kesehatan). Secara umum, nilai RFC pada sebagian besar fitur masih berada di bawah ambang batas 100, yang menandakan bahwa tingkat interaksi antar metode dalam kelas masih tergolong wajar dan *testability* tidak begitu buruk. Namun, terdapat satu pengecualian pada fitur riwayat donor darah, yang memiliki nilai RFC sebesar 103, melebihi ambang batas yang disarankan. Nilai yang tinggi ini mengindikasikan bahwa kelas tersebut memiliki kompleksitas interaksi metode yang cukup besar. Hal ini disebabkan oleh struktur kode lama yang belum menerapkan pemisahan tanggung jawab secara tepat, sehingga berbagai fungsi dan logika digabungkan dalam satu kelas. Jika struktur ini dipertahankan, penambahan fitur di masa depan dapat menjadi sulit dan rawan menimbulkan kerusakan pada bagian lain.

Untuk *Lines of Code* (LOC), hasil pengukuran menunjukkan angka yang masih di bawah ambang batas 200 yang mengindikasikan bahwa sebagian besar fitur minim *code smells*. Namun, terlihat bahwa dua fitur, yaitu artikel kesehatan dan riwayat donor darah, memiliki nilai LOC yang melebihi ambang batas *best practice* sebesar 200 baris. Hal ini mengindikasikan bahwa ukuran kelas pada fitur-fitur tersebut sudah terlalu besar dan berpotensi mengandung kode-kode yang tidak dibutuhkan, seperti logika yang tumpang tindih, duplikasi, atau kode yang tidak modular. Jika dibiarkan, kondisi ini dapat menambah kompleksitas, menyulitkan pemeliharaan, dan memperlambat proses pengembangan.

Untuk metrik *Lines of Comment* (LOCCOM), hasil pengukuran menunjukkan tingkat dokumentasi yang bervariasi antar fitur, dengan beberapa fitur memiliki jumlah komentar yang relatif minim, seperti pada fitur riwayat donor darah dan pengingat kesehatan yang hanya mencatat enam baris komentar atau kurang. Secara keseluruhan, nilai-nilai ini masih berada dalam rentang yang dianggap wajar berdasarkan acuan dari Chidamber dan Kemerer (1994).

Sementara itu, hasil pengukuran pada metrik *Number of Method Calls* (NMC) memperlihatkan frekuensi pemanggilan metode yang beragam di setiap fitur. Nilai-nilai

tersebut umumnya masih berada dalam batas yang dapat diterima, namun fitur seperti riwayat donor darah dan *chatbot* menunjukkan angka yang relatif tinggi, yakni masing-masing mencapai 56 dan 75 pemanggilan. Kondisi ini perlu diperhatikan agar tidak menimbulkan ketergantungan berlebihan antar komponen.

Untuk metrik *Weighted Methods per Class* (WMC), hasil pengukuran menunjukkan bahwa sebagian besar fitur memiliki nilai yang cukup tinggi, bahkan melebihi ambang batas *best practice* sebesar 100, seperti pada fitur riwayat donor darah (149), *chatbot* Sedonor (139), dan info tempat donor (113). Nilai ini mencerminkan banyaknya metode serta kompleksitas logika yang terkonsentrasi dalam satu kelas, yang dapat mempersulit proses pengujian maupun pemeliharaan. Hal ini terjadi karena struktur kode lama belum menerapkan prinsip pemisahan tanggung jawab dengan baik, sehingga banyak fungsi dikonsolidasikan dalam satu kelas. Sebaliknya, nilai WMC yang sangat rendah terlihat pada fitur pengingat kesehatan, yakni hanya 12, karena fungsionalitas fitur ini relatif sederhana dan hanya terbatas pada penampilan notifikasi tanpa logika pemrosesan yang kompleks.

Secara keseluruhan, meskipun nilai-nilai metrik yang diperoleh masih berada dalam rentang yang dapat ditoleransi untuk tahap awal pengembangan, beberapa indikator seperti RFC, LOC, dan WMC menunjukkan angka yang cukup tinggi pada sejumlah fitur tertentu. Hal ini mengindikasikan adanya potensi kompleksitas berlebih, ukuran kelas yang membesar, serta tingkat interaksi metode yang tinggi dalam satu kelas. Kondisi tersebut umumnya disebabkan oleh struktur kode lama yang belum menerapkan prinsip pemisahan tanggung jawab secara optimal, sehingga logika dan fungsionalitas masih terpusat dalam satu kelas. Jika dibiarkan, hal ini dapat menyulitkan proses pengujian, pemeliharaan, serta pengembangan fitur di masa depan. Oleh karena itu, diperlukan tindakan *refactoring* dan penerapan prinsip desain modular guna meningkatkan keterbacaan, keterujian, serta skalabilitas kode seiring bertambahnya kompleksitas aplikasi.

## 4.2 Pelaksanaan TDD

Untuk mendukung penerapan TDD secara efektif, setiap *usecase* dalam aplikasi telah dipetakan dengan kelas-kelas yang berkaitan, baik pada lapisan *repository*, *view model*, maupun entitas data yang digunakan. Pemetaan ini berfungsi sebagai fondasi awal dalam menyusun skenario pengujian dan implementasi kode secara bertahap. Berikut adalah daftar *usecase* beserta kelas-kelas yang mendukungnya dalam pengembangan aplikasi Sedonor.

Tabel 4.2 Pemetaan Kelas yang Berkaitan Terhadap *use case*

| ***Usecase*** | **Kelas yang Berkaitan** |
|---|---|
| Info tempat donor | 1. *LocationRepository*<br>2. *LocationViewModel*<br>3. *Location* |
| Artikel kesehatan | 1. *ArticleRepository*<br>2. *ArticleViewModel*<br>3. *Article* |
| Riwayat donor darah | 1. *DonorHistoryRepository*<br>2. *DonorHistory* |
| Pengingat kesehatan | 1. *Reminder* |
| *Check-in* donor | 1. *CheckInRepository*<br>2. *DonorHistory*<br>3. *CheckInViewModel* |
| *Chatbot* Sedonor | 1. *OpenAI*<br>2. *OpenAIRepository*<br>3. *ChatbotViewModel* |

Desain *test*case di atas merupakan desain *test case* umum yang akan diimplementasikan dalam kode. Desain ini dapat diterapkan pada berbagai kelas karena struktur kode mengikuti arsitektur berlapis. Implementasi *test case* ini dapat dilakukan secara menyeluruh pada setiap lapisan aplikasi.

Pada subbab selanjutnya, akan disajikan contoh penerapan *Test-Driven Development* (TDD) pada beberapa fitur, yaitu melihat artikel, mengakses *chatbot*, dan memasukkan data ke riwayat donor. Pemilihan fitur-fitur ini dilakukan untuk menghindari penulisan yang *redundant*, mengingat beberapa fitur memiliki alur pengembangan yang serupa. Hasil unit test dari fitur-fitur yang tidak dipilih dapat dilihat pada Lampiran B dan Lampiran C.

### 4.2.1 Siklus TDD Pada Fitur Melihat Artikel

Pada subbab ini akan dilakukan penerapan TDD pada fitur melihat artikel. Seperti yang tertera di bab sebelumnya, siklus TDD akan diawali pembuatan *unit test*, implementasi kode program dan *refactor*.

## A. Pembuatan Unit *test*

Sebelum melakukan penulisan kode *unit test*, alur fitur akan lebih dahulu ditentukan. Dalam hal ini, layaknya penerapan pada sistem sebelumnya, Melihat artikel bekerja dengan mengambil data dari remote database dan menyimpannya di sistem. *Remote database* yang digunakan di sini adalah Firebase Firestore dan penerapannya dalam bahasa Kotlin.

Dengan demikian, dapat dihasilkan kasus uji sesuai dengan kebutuhan asli. Kasus uji tersebut disajikan pada Tabel 4.3 dan Tabel 4.4.

Tabel 4.3 Desain *Test Case* Melihat Artikel Berhasil

| **Judul** | TC-01: Verifikasi bahwa sistem dapat memuat artikel dengan benar |
|---|---|
| ***Requirement*** | Koneksi Internet. |
| ***Preconditions*** | *Remote* Database yang telah terisi data. |
| **Langkah** | Kirim permintaan pengambilan data ke database. |
| ***Expected Results*** | Sistem berhasil mendapatkan daftar artikel dari database. |

Tabel 4.4 Desain *Test Case* Melihat Artikel Menghasilkan Eror

| **Judul** | TC-02: Verifikasi bahwa sistem menampilkan pesan eror ketika gagal memuat artikel |
|---|---|
| ***Requirement*** | Koneksi Internet. |
| ***Preconditions*** | *Remote* Database yang tidak terisi data. |
| **Langkah** | Kirim permintaan pengambilan data ke *database.* |
| ***Expected Results*** | Sistem menampilkan pesan eror *"Firestore fetch eror".* |

Dalam proses penulisan *test case*, alat yang digunakan adalah JUnit sebagai kerangka pengujian unit, serta MockK sebagai *library* untuk melakukan *mocking* terhadap dependensi atau objek-objek eksternal yang diperlukan selama pengujian. Penggunaan alat ini membantu memastikan bahwa setiap fungsi dapat diuji secara terisolasi tanpa bergantung pada implementasi nyata dari bagian lain. Selanjutnya kode pengujian dibuat dan dapat dilihat di Gambar 4.5 dan Gambar 4.6.

```
    @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())
        }
```

Gambar 4.5 Kode *Test ArticleRepository*

```
    @Test
        fun `fetch articles should throw exception`() = runTest {

            val Repository = ArticleRepository(mockFirestore)

            every { mockFirestore.collection("articles") } returns
mockCollectionReference
            every { mockCollectionReference.get() } throws
RuntimeException("Firestore fetch error")

            val result = Repository.fetchArticles()

            assertTrue(result.isFailure)
            assertEquals("Firestore fetch error",
result.exceptionOrNull()?.message)
        }
```

Gambar 4.6 Kode *Test ArticleRepository*

Pada Gambar 4.5 dan Gambar 4.6 dapat dilihat bahwa terdapat dua fungsi pengujian dalam kelas *ArticleRepository*. Fungsi pertama bernama *"fetch articles successfully from firestore"* merupakan kasus uji yang dirancang untuk memastikan bahwa, ketika data diambil dari Firebase, fungsi yang diuji harus mengembalikan objek *Result* dalam bentuk success dan memuat daftar artikel yang telah ditentukan sebelumnya. Dalam pengujian ini, digunakan beberapa objek *mock* seperti *FirebaseFirestore*, *CollectionReference*, *Task*, dan *QuerySnapshot* untuk menyimulasikan proses pengambilan data tanpa benar-benar terhubung ke Firebase. Fungsi ini juga memverifikasi bahwa hasil dari pemanggilan fungsi *"fetchArticles"* sesuai dengan data *dummy* yang telah dibuat. Sementara itu, fungsi *"fetch articles should throw exception"* digunakan untuk menguji skenario ketika proses pengambilan data gagal. Pada kasus ini, *get* dari *CollectionReference* akan melemparkan *RuntimeException*, dan hasil dari fungsi *"fetchArticles"* diharapkan menghasilkan objek *Result* dalam bentuk *Failure* dengan pesan kesalahan yang sesuai, yaitu *"Firestore fetch error"*. Kedua pengujian

ini memastikan bahwa aplikasi dapat menangani baik kondisi berhasil maupun gagal dalam pengambilan data artikel dari Firestore. Hasil tes dari kode uji ini dapat dilihat di Gambar 4.7.




Gambar 4.7 Hasil *Test ArticleRepository* Gagal

Sesuai pada Gambar 4.7, pengujian ini termasuk dalam tahap awal pengembangan, di mana hasil yang diharapkan adalah status *red* atau gagal sesuai dengan penelitian dari (Naufal et al., 2023). Hal ini wajar terjadi karena logika dari fungsi yang akan diuji belum diimplementasikan, sehingga kode masih dalam tahap membangun kerangka dasar pengujian.

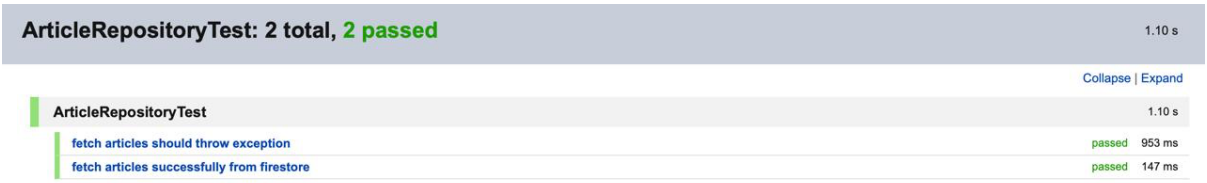
## B. Implementasi Kode Program

Setelah penulisan kode pengujian, kode program lalu diimplementasikan dengan harapan dapat memenuhi seluruh kasus uji. Kode juga dibuat sedemikian rupa dari lapisan model agar, jika nantinya basis data jarak jauh berubah, selain Firestore, perubahannya tidak mengganggu banyak berkas, sesuai dengan prinsip Liskov (Martin, 2009). Pada tahap ini, kode pertama ditulis untuk memenuhi kasus uji dasar, yaitu memastikan data dapat diambil.

```
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)
}
}
}
```

Gambar 4.8 Kode Implementasi *Refactoring*

Pada Gambar 4.8 ditampilkan implementasi fungsi *"fetchArticles"* dalam kelas *ArticleRepository*, yang berfungsi untuk mengambil data dari koleksi *articles* di Firebase Firestore. Fungsi ini bersifat *suspend*, sehingga dapat dijalankan secara asinkron menggunakan *Coroutine*. Di dalam blok *try*, koleksi diakses melalui *"firestore.collection("articles")"*, lalu datanya diambil dengan fungsi *get* dan dikonversi menjadi daftar objek *Article*. Hasilnya dikembalikan dalam bentuk *Result.success*. Jika terjadi exception selama proses pengambilan data, maka akan ditangani oleh blok *catch* dan dikembalikan sebagai *Result.failure*.

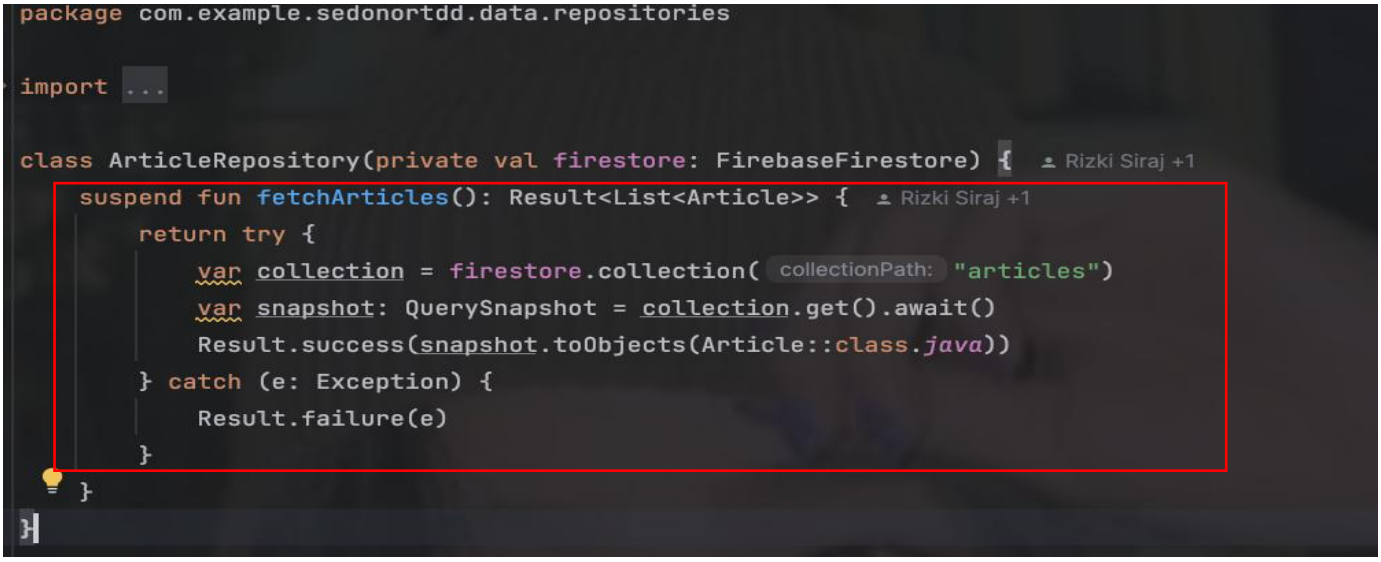



Gambar 4.9 Hasil *Test ArticleRepository* Berhasil

Seperti yang tertera di Gambar 4.9, hasil dari tahap ini adalah kode pengujian yang berhasil. Hal ini ditandai dengan status *"green"* pada hasil pengujian.

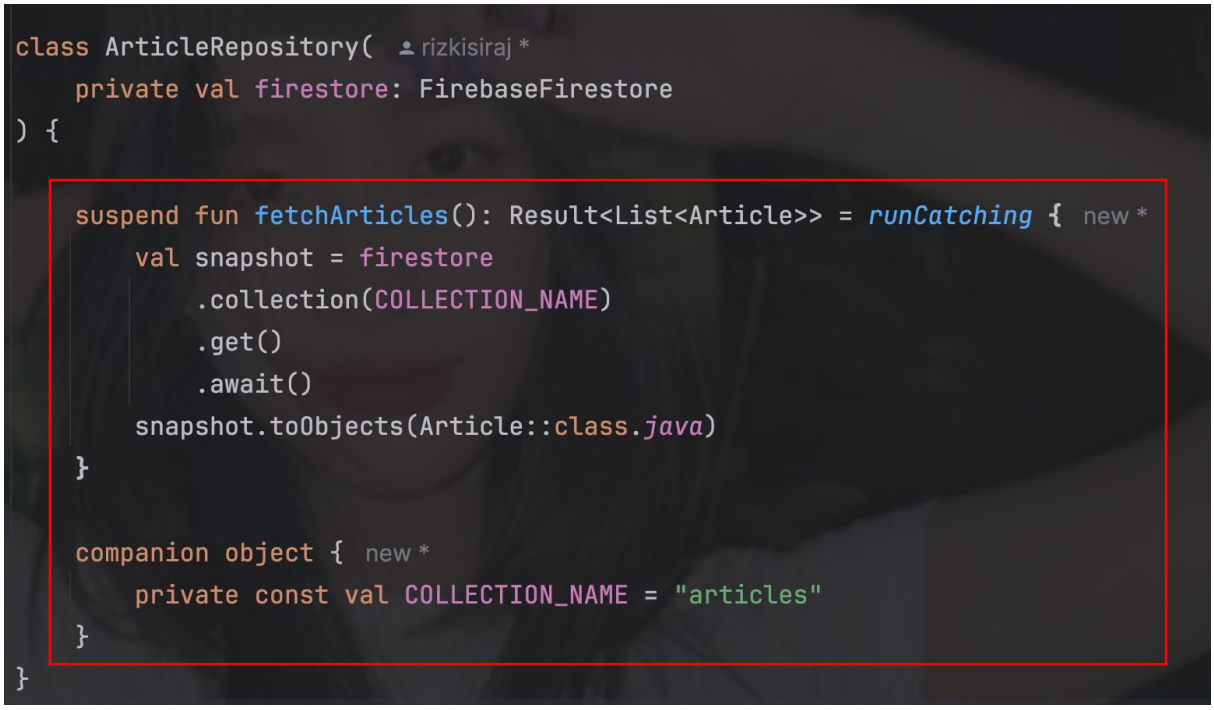
### C. *Refactor* Kode

Setelah implementasi kode selesai dilakukan, langkah selanjutnya adalah melakukan *refactor* kode, salah satunya adalah di *file ArticleRepository*. *Refactor* dilakukan agar kode lebih mudah dibaca, dikelola, dan sesuai dengan prinsip *clean code* menurut (Martin, 2009). Sebagai contoh, pada *ArticleRepository*, dilakukan penyederhanaan logika pengambilan data dari Firestore serta perbaikan struktur penulisan kode agar lebih konsisten dan jelas. Proses *refactor* ini juga dibantu oleh alat seperti *tool linting* dari IntelliJ IDEA dan *plugin* SonarLint, yang memberikan saran perbaikan terhadap potensi masalah seperti kompleksitas kode, penamaan variabel, serta penggunaan struktur yang kurang efisien. Dengan *refactor* ini, setiap fungsi dalam *ArticleRepository* menjadi lebih terstruktur dan siap diuji serta dikembangkan lebih lanjut secara berkelanjutan.




Gambar 4.10 Kode *ArticleRepository* Sebelum Di*refactor*

Terlihat pada Gambar 4.10, fungsi *"fetchArticles"* masih memiliki struktur yang kurang efisien dan belum sepenuhnya mengikuti prinsip *clean code* (Martin, 2009). Penggunaan blok *try-catch* secara manual membuat kode terlihat lebih panjang dan repetitif, sementara variabel sementara yang tidak diperlukan justru menambah beban visual saat membaca kode yang dapat terlihat dari garis bawah yang diberikan Intellij IDEA di bagian bawah *var*. Selain itu, penggunaan *string literal* pada nama *collection* meningkatkan risiko kesalahan dan menyulitkan proses pemeliharaan apabila terjadi perubahan di masa mendatang.




Gambar 4.11 Kode *Refactoring* Setelah Di*refactor*

Pada Gambar 4.11 ditunjukkan hasil *refactor ArticleRepository* yang telah mengikuti prinsip *clean code* dan SOLID (Martin, 2009). Penggunaan *runCatching* membuat penanganan eror lebih ringkas dan idiomatis. Konstanta *COLLECTION_NAME* menghindari penggunaan *string literal* sehingga kode lebih terstruktur. Secara SOLID, kelas ini memenuhi prinsip *Single Responsibility* karena hanya menangani logika pengambilan artikel, serta prinsip *Dependency Inversion* melalui penyuntikan dependensi Firebase Firestore.

### 4.2.2 Siklus TDD Pada Fitur Memasukkan Data ke Riwayat Donor

Pada subbab ini akan dilakukan penerapan TDD pada fitur memasukkan data ke riwayat donor. Seperti yang tertera di bab sebelumnya, siklus TDD akan diawali pembuatan *unit test*, implementasi kode program dan *refactor*.

**A. Pembuatan *Unit test***

Pada fitur ini, *remote database* masih menggunakan Firestore seperti fitur sebelumnya. Desain *Test Cases* pada fitur ini dapat dilihat di Tabel 4.5 dan Tabel 4.6.

Tabel 4.5 *Test Case* Memasukkan Data ke Riwayat Berhasil

| | |
|---|---|
| **Judul** | TC-03: Verifikasi bahwa aplikasi dapat menyimpan riwayat donor darah baru |
| ***Requirement*** | Koneksi Internet |
| ***Preconditions*** | *Remote database* belum terisi data |
| **Langkah** | 1. Membuat data *pre-generated*<br>2. Simpan ke Database |
| ***Expected Results*** | Data riwayat donor berhasil disimpan ke database |

Tabel 4.6 *Test Case* Memasukkan Data ke Riwayat Gagal

| | |
|---|---|
| **Judul** | TC-04: Verifikasi bahwa aplikasi mengirim pesan eror jika gagal menyimpan riwayat donor darah baru. |
| ***Requirement*** | Koneksi Internet. |
| ***Preconditions*** | *Remote database* belum terisi data. |
| **Langkah** | 1. Membuat data *pre-generated* |

| | 2. Simpan ke Database |
|---|---|
| ***Expected Results*** | Data riwayat donor mengirim eror. |

```
    @Test
        fun `create DonorHistory successfully in firestore`() = 
runBlocking {
            val history = DonorHistory(
                weight = 80,
                heartRate = 120,
                hemoglobinLevel = 54,
                status = true,
                temperature = 35,
                diastolicPressure = 120,
                systolicPressure = 8,
                donorLocation = "Dutamas"
            )
            every { mockFirestore.collection("donorHistories") } returns 
mockCollectionReference
            every { mockCollectionReference.add(history) } returns 
mockDocumentReferenceTask

            mockkStatic("kotlinx.coroutines.tasks.TasksKt")
            coEvery { mockDocumentReferenceTask.await() } returns 
mockDocumentReference

            val result = Repository.createHistory(history)

            assertTrue(result.isSuccess)
            assertEquals(null, result.getOrNull())
            verify { mockCollectionReference.add(history) }
        }
```

Gambar 4.12 Kode *Test DonorHistoryRepository*

Pada Gambar 4.12 ditampilkan salah satu pengujian unit untuk fitur membuat riwayat donor dengan judul fungsi "c*reate DonorHistory successfully in firestore"*. Pengujian ini bertujuan untuk memastikan bahwa data riwayat donor yang dibuat dapat berhasil disimpan ke dalam koleksi *donorHistories* di Firestore. Pada bagian awal, objek *DonorHistory* dibuat dengan data *dummy*. Selanjutnya, dependensi Firestore di-*mock* menggunakan *library* MockK, termasuk pemanggilan *add* serta proses asynchronous dengan *await*. Fungsi *createHistory* kemudian dipanggil, dan hasilnya divalidasi menggunakan *assertTrue* dan *assertEquals* untuk memastikan bahwa fungsi mengembalikan *Result.success* dengan nilai *null*, sesuai dengan tipe pengembalian *void*. Selain itu, dilakukan verifikasi bahwa fungsi *add* memang dipanggil sesuai ekspektasi.

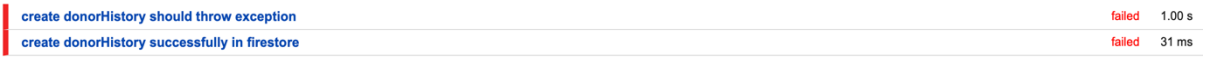
```
    @Test
        fun `create DonorHistory should throw exception`() = runTest {
        val history = DonorHistory(
                weight = 80,
                heartRate = 120,
                hemoglobinLevel = 54,
                status = true,
                temperature = 35,
                diastolicPressure = 120,
                systolicPressure = 8,
                donorLocation = "Dutamas"
            )
            every { mockFirestore.collection("donorHistories") } returns
mockCollectionReference
            every { mockCollectionReference.add(history) } throws
RuntimeException("Firestore write error")

            val result = Repository.createHistory(history)

            assertTrue(result.isFailure)
            assertEquals("Firestore write error",
result.exceptionOrNull()?.message)
        }
```

Gambar 4.13 Kode *Test DonorHistoryRepository*

Gambar 4.13 memperlihatkan pengujian unit untuk fungsi *createHistory* dengan skenario kegagalan saat menyimpan data ke Firestore. Pada pengujian ini, objek *DonorHistory* dibuat sebagai data contoh, kemudian dependensi Firestore di-*mock* sehingga ketika fungsi *add* dipanggil pada koleksi *donorHistories*, akan melemparkan *RuntimeException* dengan pesan *"Firestore write error"*. Setelah fungsi *createHistory* dijalankan, hasilnya diuji menggunakan *assertTrue* untuk memastikan bahwa operasi berakhir dengan *Result.failure*, serta *assertEquals* untuk memverifikasi bahwa pesan kesalahan yang dihasilkan sesuai dengan yang diharapkan. Hasil dari tahap ini adalah kode pengujian yang gagal seperti Gambar 4.14.




Gambar 4.14 Hasil *Test DonorHistoryRepository* Gagal

**B. Implementasi Kode**

Setelah penulisan kode pengujian selesai, implementasi kode program dilakukan dengan tujuan untuk memenuhi semua skenario pengujian yang telah dirancang. Struktur kode disusun mulai dari lapisan model, dengan mempertimbangkan kemungkinan perubahan basis data jarak jauh di masa depan. Dengan desain ini, apabila terjadi perubahan dari Firestore ke layanan lain, dampaknya terhadap berkas lain akan minimal, sejalan dengan prinsip Liskov

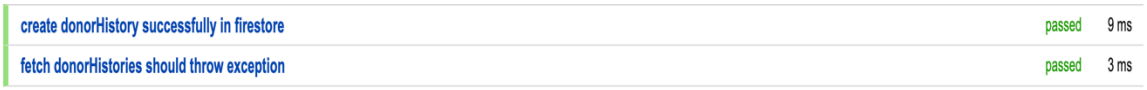
(Martins et al., 2023). Pada tahap ini, fokus utama adalah menulis kode dasar yang dapat memastikan proses pengambilan data berjalan dengan benar.

```
suspend fun createHistory(history: DonorHistory): Result<Nothing?> {
        return try {
            firestore.collection("donorHistories")
                .add(history)
                .await()
            Result.success(null)
        } catch (e: Exception) {
            Result.failure(e)
        }
    }
```

Gambar 4.15 Kode Implementasi *DonorHistoryRepository*

Seperti yang terlihat dari Gambar 4.15, implementasi fungsi *createHistory* bertujuan untuk menambahkan data riwayat donor ke dalam koleksi *donorHistories* pada Firestore. Fungsi ini dideklarasikan sebagai *suspend*, yang berarti dijalankan dalam konteks *coroutine* agar proses komunikasi dengan basis data dapat dilakukan secara *asynchronous* tanpa memblokir *thread* utama. Pada bagian *try*, fungsi *add* dipanggil untuk menambahkan data, lalu *await* digunakan untuk menunggu proses penyimpanan selesai. Jika operasi berhasil, fungsi akan mengembalikan *Result.success(null)* karena tidak ada data spesifik yang dikembalikan. Namun, jika terjadi kesalahan selama proses penyimpanan, pengecualian akan ditangkap di blok *catch*, dan fungsi akan mengembalikan *Result.failure(e)*. Dengan pendekatan ini, fungsi dapat mengelola keberhasilan maupun kegagalan operasi secara eksplisit dan aman.

Seperti halnya pada siklus sebelumnya, hasil akhir dari tahap ini menunjukkan hasil pengujian fitur yang berhasil. Hal ini terlihat dengan status *"green*" pada Gambar 4.16.




Gambar 4.16 Hasil *Test DonorHistoryRepository* Berhasil

### C. *Refactor* Kode

Pada tahap ini, *refactor* akan dilakukan terhadap fungsi *createHistory* karena ada peluang untuk perbaikan dari segi *clean architecture.* Pada Gambar 4.15 terlihat bahwa fungsi *createHistory* masih menggunakan *Result<Nothing?>* sebagai tipe data hasil pengembalian. Penggunaan *Nothing?* di sini kurang tepat, karena fungsi tersebut hanya bertugas menyimpan data tanpa mengembalikan nilai apa pun selain status berhasil atau gagal. Selain itu, struktur

kode masih dapat diperbaiki agar lebih konsisten dan mudah dipahami. Meskipun penanganan kesalahan sudah diterapkan dengan menggunakan blok *try-catch*, penggunaan tipe pengembalian yang kurang representatif dapat menimbulkan kebingungan dalam pengembangan selanjutnya. Perbaikan selanjutnya dilakukan pada Gambar 4.17.

```
suspend fun createHistory(history: DonorHistory): Result<Unit> {
    return try {
        firestore.collection("donorHistories")
            .add(history)
            .await()
        Result.success(Unit)
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

Gambar 4.17 Kode *DonorHistoryRepository* Setelah Di*refactor*

Pada Gambar 4.17, fungsi *createHistory* telah menggunakan *Result<Unit>* sebagai tipe data hasil pengembalian, yang secara semantik lebih sesuai karena *Unit* merepresentasikan aksi yang tidak mengembalikan data. Selain itu, penyusunan kode disusun lebih rapi tanpa melakukan perubahan struktur besar seperti menambahkan kelas baru, sehingga tetap menjaga kesederhanaan implementasi. Dengan perbaikan ini, fungsi menjadi lebih bersih, lebih mudah dibaca, dan lebih sesuai dengan prinsip *clean architecture* yang mengutamakan keterpisahan antara logika domain dengan detail teknis tanpa memperumit arsitektur.

### 4.2.3 Siklus TDD Pada Fitur Mengakses *Chatbot*

Dalam subbab ini, akan dilakukan penerapan TDD pada fitur mengakses *chatbot*. Seperti yang tertera di bab sebelumnya, siklus TDD akan diawali pembuatan *unit test*, implementasi kode program dan *refactor*.

#### A. Pembuatan *Unit test*

Pada fitur ini, *chatbot* yang dibuat akan memanfaatkan layanan pihak ketiga, yaitu OpenAI. Skema dari fitur ini adalah pengguna akan mengirimkan *prompt*, kemudian OpenAI API akan mengirimkan respon.

Dengan demikian, dapat dihasilkan *test cases* yang sesuai dengan kebutuhan aplikasi. *Test cases* tersebut disajikan pada Tabel 4.7 dan Tabel 4.8.

Tabel 4.7 Desain *Test Case* Mengakses *Chatbot* Berhasil

| **Judul** | TC-05: Verifikasi bahwa aplikasi dapat berkomunikasi dengan *chatbot* |
|---|---|
| ***Requirement*** | Koneksi Internet. |
| ***Preconditions*** | Aplikasi sudah berjalan dengan baik. |
| **Langkah** | 1. Akses fitur *chatbot*<br>2. Kirim pesan "Halo |
| ***Expected Results*** | *Chatbot* merespons pesan pengguna dengan benar |

Tabel 4.8 *Test Case* Mengakses *Chatbot* Gagal

| **Judul** | TC-06: Verifikasi bahwa aplikasi dapat berkomunikasi dengan *chatbot* |
|---|---|
| ***Requirement*** | Koneksi Internet. |
| ***Preconditions*** | Aplikasi sudah berjalan dengan baik. |
| **Langkah** | 1. Akses fitur *chatbot*<br>2. Kirim pesan “Halo” |
| ***Expected Results*** | *Chatbot* merespons pesan pengguna dengan mengeluarkan pesan eror. |

```
@Test
    fun `getGptResponse should make a successful API call`() {
        val fakeResponse = OpenAiResponse(
            id = "test123",
            `object` = "chat.completion",
            created = 0,
            model = "gpt-4.1",
            usage = Usage(1, 1, 2),
            choices = listOf(
                Choice(
                    message = Message("assistant", "Hello!"),
                    finish_reason = "stop",
                    index = 0
                )
            )
        )

        `when`(openAiApi.getGptResponse(any(),
any())).thenReturn(mockCall)
        `when`(mockCall.enqueue(any(Callback::class.java)              as
Callback<OpenAiResponse>)).thenAnswer {
```

```
            val callback = it.getArgument<Callback<OpenAiResponse>>(0)
            val response = Response.success(fakeResponse)
            callback.onResponse(mockCall, response)
        }

        val request = OpenAiRequest(model = "gpt-4.1", messages =
listOf(Message("user", "Hi")), temperature = 0.7)
        val authorizationHeader = "APIKEY"
        Repository.getGptResponse(authorizationHeader, request, object :
Callback<OpenAiResponse> {
            override fun onResponse(call: Call<OpenAiResponse>, response:
Response<OpenAiResponse>) {
                assertTrue(response.isSuccessful)
                val responseBody = response.body()
                assertNotNull(responseBody)
                assertEquals("test123", responseBody?.id)
            }

            override fun onFailure(call: Call<OpenAiResponse>, t:
Throwable) {
                fail("API call failed: ${t.message}")
            }
        })
    }
```

Gambar 4.18 Kode *ChatbotRepository*

Gambar 4.18 menunjukkan pengujian unit untuk fungsi *getGptResponse* yang bertujuan memastikan bahwa pemanggilan API ke OpenAI dapat berjalan dengan sukses. Pada pengujian ini, respon palsu disiapkan, kemudian objek *openAiApi* dan *mockCall* di-*mock* untuk meniru perilaku sukses saat melakukan *enqueue*. Fungsi *getGptResponse* kemudian dipanggil dengan data *dummy*, dan hasilnya diverifikasi menggunakan *assertTrue* untuk memastikan respon berhasil, *assertNotNull* untuk memastikan respon tidak kosong, serta *assertEquals* untuk memeriksa kesesuaian ID dari data yang diterima.

```
        @Test
        fun `getGptResponse should handle API failure`() {
            // Mock the API response failure
            `when`(openAiApi.getGptResponse(any(),
any())).thenReturn(mockCall)
            `when`(mockCall.enqueue(any(Callback::class.java) as
Callback<OpenAiResponse>)).thenAnswer {
                val callback =
it.getArgument<Callback<OpenAiResponse>>(0)
                callback.onFailure(mockCall, Throwable("Network error"))
            }


            val request = OpenAiRequest(model = "gpt-4.1", messages =
listOf(Message("user", "Hi")), temperature = 0.7)
            val authorizationHeader = "APIKEY"
            Repository.getGptResponse(authorizationHeader, request,
object : Callback<OpenAiResponse> {
```

```
                override fun onResponse(call: Call<OpenAiResponse>,
response: Response<OpenAiResponse>) {
                    fail("API call should have failed")
                }

                override fun onFailure(call: Call<OpenAiResponse>, t:
Throwable) {
                    assertEquals("Network error", t.message)
                }
            })
        }
```

Gambar 4.19 Kode *Test ChatbotRepository*

Selanjutnya, pada Gambar 4.19 ditunjukkan pengujian unit untuk fungsi *getGptResponse* dengan skenario kegagalan saat melakukan pemanggilan API ke OpenAI. Dalam pengujian ini, objek *openAiApi* dan *mockCall* di-*mock* untuk meniru kondisi kegagalan dengan memicu *onFailure* dan menghasilkan pesan kesalahan *"Network error"*. Setelah fungsi *getGptResponse* dipanggil, dilakukan verifikasi dengan *assertEquals* untuk memastikan bahwa pesan kesalahan yang diterima sesuai dengan yang diharapkan. Pengujian ini memastikan bahwa fungsi mampu menangani kegagalan jaringan dengan baik dan memberikan respon kesalahan yang sesuai. Hasil pengujian pertama dari kode uji ini dapat dilihat di Gambar 4.20.




Gambar 4.20 Hasil *Test Chatbot* Repostiory

## B. Implementasi Kode

Setelah proses penulisan kode pengujian diselesaikan, implementasi kode program dilanjutkan dengan tujuan memenuhi seluruh skenario uji yang telah dirancang. Penyusunan struktur kode dimulai dari lapisan model, dengan memperhatikan kemungkinan perubahan basis data jarak jauh di masa mendatang.

```
class ChatbotRepository {

    private val openAiApi: OpenAiApi

    init {
        val retrofit = Retrofit.Builder()
            .baseUrl("https://api.openai.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .client(
                OkHttpClient.Builder()
                    .connectTimeout(60, TimeUnit.SECONDS)
                    .readTimeout(60, TimeUnit.SECONDS)
```

```
                    .build()
            )
            .build()

        openAiApi = retrofit.create(OpenAiApi::class.java)
    }

    fun getGptResponse(authorizationHeader: String, request:
OpenAiRequest, callback: Callback<OpenAiResponse>) {
        openAiApi.getGptResponse(authorizationHeader,
request).enqueue(callback)
    }
}
```

Gambar 4.21 Kode Implementasi *ChatbotRepository*

Pada Gambar 4.21 ditampilkan implementasi kelas *ChatbotRepository* yang berfungsi sebagai perantara antara aplikasi dengan layanan OpenAI API. Pada bagian *init*, objek Retrofit dikonfigurasi dengan menetapkan *baseUrl* menuju *endpoint OpenAI*, menambahkan *GsonConverterFactory* untuk mengonversi data JSON, serta mengatur *OkHttpClient* dengan batas waktu koneksi dan baca selama 60 detik. Setelah konfigurasi selesai, antarmuka *OpenAiApi* diinisialisasi dari instance *Retrofit* tersebut. Kelas ini menyediakan fungsi *getGptResponse* yang digunakan untuk mengirim permintaan ke OpenAI API secara asinkron melalui metode *enqueue*, serta mengelola respon atau kegagalan permintaan melalui mekanisme *callback*.

Setelah implementasi kode dilakukan, pengujian unit akan dilanjutkan untuk memastikan fungsionalitasnya. Hasil pengujian tersebut dapat dilihat di Gambar 4.22.




Gambar 4.22 Hasil *Test ChatbotRepository* Berhasil

### C. *Refactor* Kode

Pada tahap ini terdapat peluang untuk *refactor* kode pada kelas *ChatbotRepository*. Seperti yang terlihat di Gambar 4.18, terdapat beberapa perbaikan yang dapat dilakukan untuk meningkatkan kualitas kode berdasarkan prinsip *clean architecture*. Pada kode awal, proses inisialisasi Retrofit dan *OkHttpClient* dilakukan langsung di dalam blok *init*, sehingga membuat kode menjadi panjang dan sulit dibaca. Sesuai prinsip *Single Responsibility*, proses pembuatan *OkHttpClient* dan *Retrofit* sebaiknya dipisahkan ke dalam fungsi khusus agar kode menjadi lebih modular, mudah diuji, dan lebih fleksibel terhadap perubahan. Selain itu, penggunaan callback dalam fungsi *getGptResponse* masih berbasis *callback-style*, yang kurang

optimal dibandingkan penggunaan suspend function berbasis coroutine yang lebih bersih dan mengurangi potensi *callback hell*. Dengan *refactoring* ini, kode menjadi lebih bersih, terstruktur, dan sesuai dengan prinsip *Separation of Concerns* (Martin, 2009). Ada peluang perbaikan yang dapat dilihat di Gambar 4.23.

```
    class ChatbotRepository {

        private val openAiApi: OpenAiApi

        init {
            val retrofit = createRetrofit()
            openAiApi = retrofit.create(OpenAiApi::class.java)
        }

        private fun createRetrofit(): Retrofit {
            return Retrofit.Builder()
                .baseUrl("https://api.openai.com/")
                .addConverterFactory(GsonConverterFactory.create())
                .client(createOkHttpClient())
                .build()
        }

        private fun createOkHttpClient(): OkHttpClient {
            return OkHttpClient.Builder()
                .connectTimeout(60, TimeUnit.SECONDS)
                .readTimeout(60, TimeUnit.SECONDS)
                .build()
        }

        fun getGptResponse(
            authorizationHeader: String,
            request: OpenAiRequest,
            callback: Callback<OpenAiResponse>
        ) {
            openAiApi.getGptResponse(authorizationHeader,
request).enqueue(callback)
        }
    }
```

Gambar 4.23 Kode *ChatbotRepository* Setelah Di*refactor*

Pada Gambar 4.23 terlihat bahwa kode *ChatbotRepository* telah di*refactoring* agar lebih modular dan sesuai dengan prinsip *clean architecture*. Sebelumnya, seluruh proses inisialisasi Retrofit dan *OkHttpClient* ditulis langsung dalam blok *init*, sehingga membuat kode menjadi panjang dan kurang terstruktur. Setelah *refactoring*, pembuatan Retrofit dan *OkHttpClient* dipisahkan ke dalam fungsi *createRetrofit* dan *createOkHttpClient*, sehingga kode menjadi lebih modular, mudah dibaca, serta lebih mudah dipelihara. Blok *init* kini lebih bersih dengan hanya satu baris inisialisasi *openAiApi*, sehingga memperjelas alur pembentukan objek.

## 4.3 Pengukuran Metrik Kode Baru

Setelah implementasi TDD selesai, pengukuran metrik pada kode baru dilakukan dengan langkah yang sama seperti pada pengukuran metrik kode lama. Hasil pengukuran metrik tersebut dapat dilihat pada Tabel 4.9.

Tabel 4.9 Hasil Pengukuran Metrik Kode Baru

| **Metrik** | **Hasil Pengukuran Info Tempat Donor** | | **Hasil Pengukuran Artikel Kesehatan** | | **Hasil Pengukuran Riwayat Donor Darah** | | **Hasil Pengukuran *Chatbot* Sedonor** | | **Hasil Pengukuran *Check-in* Donor** | | **Hasil Pengukuran Pengingat Kesehatan** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Sebe-lum** | **Sesu-dah** | **Sebelum** | **Sesudah** | **Sebelum** | **Sesudah** | **Sebelum** | **Sesudah** | **Sebelum** | **Sesudah** | **Sebelum** | **Sesudah** |
| RFC | 91 | 72 | 78 | 95 | **103** | **17** | 61 | 73 | 88 | 77 | 41 | 25 |
| LOC | 191 | 212 | 297 | 258 | **306** | **53** | 146 | 226 | 115 | 88 | **106** | **57** |
| LOCCOM | 20 | 4 | 17 | 17 | 6 | 0 | 6 | 6 | 24 | 11 | 3 | 0 |
| NMC | 50 | 28 | 35 | 31 | **56** | **15** | **75** | **27** | 35 | 12 | 8 | 4 |
| WMC | **113** | **36** | 55 | 40 | **149** | **18** | **139** | **35** | **59** | **21** | 12 | 6 |

Berdasarkan hasil perbandingan metrik pada Tabel 4.9, tampak adanya perbedaan signifikan yang tercermin dalam berbagai aspek struktural kode. Tabel pengukuran metrik pada enam fitur utama menunjukkan bahwa sebagian besar nilai metrik mengalami penurunan setelah penerapan TDD. Penurunan ini menjadi indikator penting terhadap peningkatan *testability*, khususnya dalam hal modularitas, keterpisahan logika, serta pengurangan kompleksitas metode.

Penurunan paling mencolok terlihat pada fitur riwayat donor darah. Pada fitur ini, WMC turun drastis dari 149 menjadi 18 (turun 88%), disertai penurunan RFC dari 103 menjadi 17 dan NMC dari 56 menjadi 15. LOC juga menurun signifikan dari 306 menjadi 53, mencerminkan bahwa refaktorisasi telah menyederhanakan struktur kelas secara menyeluruh. Perubahan ini terjadi karena pendekatan TDD mendorong pengembangan kode dalam unit-unit kecil dan terpisah, sehingga setiap fungsi ditulis hanya untuk memenuhi satu kebutuhan spesifik. Ini merupakan contoh ideal dari keberhasilan siklus *Red-Green-Refactor* dalam TDD yang menghasilkan kelas-kelas kecil dengan logika yang terisolasi dan mudah diuji.

Fitur *chatbot* sedonor juga menunjukkan hasil serupa. WMC menurun dari 139 menjadi 35, dan NMC dari 75 menjadi 27. Walaupun LOC pada fitur ini meningkat dari 146 menjadi 226, peningkatan tersebut dapat dijustifikasi karena arsitektur yang lebih berlapis dan pembentukan dependensi eksplisit melalui *repository* dan antarmuka API. Dalam hal ini, peningkatan LOC bukan merupakan pertanda meningkatnya kompleksitas, karena metrik-metrik lainnya yang mencerminkan logika internal justru mengalami penurunan.

Penurunan yang cukup stabil juga terjadi pada fitur info tempat donor, *check-in* donor, dan pengingat kesehatan. Pada ketiga fitur ini, WMC menurun rata-rata sebesar 50–70%, sedangkan NMC juga mengalami penyusutan antara 40–65%. Beberapa fitur, seperti *check-in* donor, mencatatkan penurunan LOC dari 115 menjadi 88, dan RFC dari 88 menjadi 77. Penurunan ini mencerminkan bahwa penerapan TDD berhasil mendorong penyusunan ulang struktur kode menjadi lebih modular dan terfokus. Artinya, logika yang sebelumnya terpusat dalam satu kelas besar dipecah menjadi beberapa bagian yang lebih kecil dengan tanggung jawab spesifik. Hasilnya adalah kode yang lebih terorganisir, mudah diuji secara unit, dan lebih tahan terhadap perubahan karena dependensinya telah dikelola dengan lebih baik. Selain itu, karena refaktorisasi difokuskan pada struktur dan bukan penambahan fitur, ukuran kelas tetap terkendali tanpa terjadi pembengkakan LOC yang signifikan.

Fitur artikel kesehatan merupakan salah satu pengecualian yang menarik untuk dicermati dalam hasil pengukuran metrik. Meskipun nilai *Weight Method per Class* (WMC) dan *Number of Method Calls* (NMC) mengalami penurunan, masing-masing dari 55 menjadi 40 dan dari 35 menjadi 31, nilai *Response for Class* (RFC) justru mengalami peningkatan dari 78 menjadi 95. Selain itu, nilai *Lines of Code* (LOC) pada fitur ini tetap tinggi, yaitu 258 setelah sebelumnya sebesar 297. Kondisi ini menunjukkan bahwa meskipun logika dalam metode telah dipisahkan dan disederhanakan, jumlah pemanggilan metode antarkelas meningkat akibat fragmentasi fungsional yang dilakukan dalam proses refaktorisasi. Dengan kata lain, kenaikan nilai RFC merupakan konsekuensi dari penerapan arsitektur modular yang lebih eksplisit, bukan karena meningkatnya kompleksitas logika internal. Oleh sebab itu, peningkatan nilai LOC pada fitur ini dapat dianggap wajar dan tidak mengurangi tingkat *testability* kode, sebab kompleksitas setiap kelas justru lebih kecil dan tetap dapat diuji dengan mudah.

Jika dilihat dari perspektif ambang batas Chidamber & Kemerer (1994) jumlah metrik yang melampaui ambang batas mengalami penurunan yang signifikan. Sebelum TDD, terdapat enam kasus metrik yang melampaui batas: tiga pada WMC (info tempat donor, riwayat donor, dan *chatbot*), dua pada LOC (artikel kesehatan dan riwayat donor), serta satu pada RFC (riwayat donor). Setelah penerapan TDD, hanya tersisa tiga kasus, seluruhnya berada pada LOC (info tempat donor, artikel kesehatan, dan *chatbot* sedonor). Tidak ada lagi fitur dengan nilai RFC di atas 100 ataupun WMC di atas 50, menandakan bahwa kompleksitas logika dalam kelas berhasil ditekan secara sistematis.

Tabel 4.10 Perbandingan Rata-rata Metrik Kode Lama dan Baru

| **Metrik** | **Rata-rata Lama** | **Rata-rata Baru** | **Persentase Penurunan** |
|---|---:|---:|---:|
| RFC | 77,0 | 59,8 | 22,3% |
| LOC | 193,5 | 136,1 | 29,6% |
| LOCCOM | 12,6 | 6,3 | 50,0% |
| NMC | 43,1 | 19,5 | 54,8% |
| WMC | 87,8 | 26,0 | 70,4% |

Tabel 4.10 menunjukkan bahwa secara agregat, semua metrik mengalami penurunan rata-rata yang signifikan. WMC menurun paling besar dengan penurunan 70,4%, diikuti oleh NMC yang turun 54,8%. Penurunan WMC ini merupakan sinyal positif karena mencerminkan

bahwa logika di dalam metode menjadi lebih sederhana dan lebih terbatas, yang merupakan hasil dari pemisahan fungsi secara sistematis. LOC dan LOCCOM juga mengalami penurunan, yang menunjukkan adanya efisiensi struktur kode serta perbaikan dalam keterbacaan dan kejelasan tanggung jawab kelas. Bahkan RFC, meskipun pada beberapa fitur mengalami peningkatan, secara keseluruhan tetap menurun 22,3%. Penurunan rata-rata ini menunjukkan bahwa *testability* kode semakin baik, karena metrik yang lebih kecil mencerminkan struktur kode yang lebih mudah diuji, lebih modular, dan lebih efisien.

Meskipun demikian, terdapat beberapa hasil yang belum sepenuhnya sesuai ekspektasi. Sebagai contoh, nilai RFC pada fitur Artikel Kesehatan meningkat dari 78 menjadi 95. Kenaikan ini bukan disebabkan oleh peningkatan kompleksitas internal, melainkan karena pemisahan logika ke dalam berbagai komponen yang mendukung pengujian terisolasi, seperti *repository, viewmodel, dan service helper*. Akibatnya, kelas tersebut menjadi memiliki lebih banyak relasi terhadap metode lain. Hal ini merupakan konsekuensi dari praktik clean architecture, yang walaupun menambah pemanggilan metode, justru menurunkan kompleksitas logika di dalam metode itu sendiri, sebagaimana dibuktikan oleh turunnya nilai WMC dan NMC.

Demikian pula, masih terdapat beberapa fitur dengan nilai LOC yang tinggi, melebihi 200 baris, seperti pada *chatbot* sedonor (226), artikel kesehatan (258), dan info tempat donor (212). Nilai LOC yang tinggi ini merupakan hasil dari bertambahnya jumlah kelas dan fungsi untuk memisahkan tanggung jawab dan mempermudah pengujian unit. Hal ini sejalan dengan filosofi TDD, yang menuntut pemecahan logika ke dalam unit-unit kecil dan teruji. Oleh karena itu, meskipun nilai LOC tinggi secara absolut, nilai tersebut dapat diterima karena disertai dengan penurunan metrik kompleksitas lainnya, serta mendukung peningkatan *testability* secara struktural.

Secara keseluruhan, penerapan TDD telah mendorong transformasi signifikan terhadap struktur kode aplikasi Sedonor, baik secara metrik individual per fitur maupun secara keseluruhan dalam *codebase*. Penurunan pada metrik kompleksitas (WMC, NMC), interaksi (RFC), dan ukuran kelas (LOC) mengindikasikan bahwa kode hasil refaktorisasi TDD lebih modular, mudah diuji, serta memiliki struktur yang lebih terkontrol dan dapat dipelihara secara jangka panjang jika ditinjau dari segi kuantitatif.

## 4.4 *Expert review* Terhadap Kode Lama dan Kode Baru

Pada subbab ini, dilakukan *expert review* untuk membandingkan aspek *testability*, modularitas, dan penerapan *clean architecture* antara kode lama sebelum penerapan TDD dan kode baru setelah penerapan TDD. Tujuan dari *expert review* ini adalah untuk mengevaluasi apakah penerapan TDD memberikan perbaikan yang signifikan terhadap kualitas arsitektur, struktur modular, serta kemudahan dalam pengujian kode. Proses pelaksanaannya dilakukan dengan meminta responden untuk terlebih dahulu membaca dan memahami kode lama dan kode baru, kemudian dilanjutkan dengan sesi wawancara untuk memperoleh penilaian dan masukan berdasarkan aspek-aspek yang telah ditentukan.

### 4.4.1 Identitas Responden

Wawancara dilakukan terhadap seorang profesional di bidang pengembangan perangkat lunak *mobile* yang memenuhi kriteria yang telah ditetapkan pada Bab 3. Responden memiliki pengalaman lebih dari tiga tahun dalam pengembangan aplikasi *mobile* berbasis *clean architecture* dan modularitas, serta memiliki pengalaman dalam melakukan evaluasi dan review kode.

Adapun identitas responden secara umum dapat dilihat di Tabel 4.11. Tabel ini menyajikan informasi mengenai nama, jabatan, dan spesialisasi responden yang terlibat dalam penelitian.

Tabel 4.11 Tabel Identitas Responden

| No | Responden | Jabatan | Pengalaman | Spesialisasi |
|---|---|---|---|---|
| 1 | Responden 1 | *Senior Developer* | Mantan *Tech Lead* di salah satu *software house* di Yogyakarta, lebih dari 3 tahun di pengembangan *software*. | *Full stack & mobile apps* |
| 2 | Responden 2 | *Senior Developer* | Programmer di Yogyakarta, lebih dari 3 tahun di pengembangan *software*. | *Full stack & Mobile apps* |
| 3 | Responden 3 | *Achievement Support* | Mantan Head of Software Development di salah satu *software house* di Yogyakarta, lebih dari 10 tahun di pengembangan *software*. | *Full stack* & Rekayasa perangkat lunak |

### 4.4.2 Hasil Wawancara dan Evaluasi Kode

Evaluasi dilakukan dengan membandingkan kode lama dan kode baru berdasarkan beberapa aspek, yaitu *clean architecture*, modularitas, *testability*, dan keterbacaan kode. Berikut merupakan ringkasan hasil wawancara dan penilaian responden terhadap kedua versi kode:

Tabel 4.12 Hasil Wawancara Responden

| Aspek | Kode Lama | Kode Baru | Kutipan Wawancara | Narasumber |
|---|---|---|---|---|
| *Clean architecture* | Kurang menerapkan konsep *clean architecture* karena struktur folder masih belum dibuat terpisah dan masih banyak campuran antara bahasa Java dan Kotlin. | Sudah menerapkan konsep *clean architecture* dengan baik karena setiap *file* sudah dibuat secara terpisah sesuai tanggung jawabnya melalui *data layer*, *ui layer*, *Repository*, *viewmodel* dan seterusnya. | "*Clean architecture* buat kode yang baru jadinya orang baru jadi lebih bisa membaca, soalnya kan sudah dipisah-pisah nih antara *data*, *UI*, dan *viewmodel*-nya. Sementara itu, kalau yang lama hanya halaman-halaman tertentu saja yang dipisah. Dan untuk bahasanya pun masih campur-campur." | Responden 1 |
| | Tidak ada pemisahan domain, data, dan UI. Business logic masih bercampur dengan UI. | *Clean architecture* sudah rapi, alur data jelas, dan sudah pakai *Repository Pattern*. | “Kode baru lebih *maintainable* karena struktur *domain* dan data sudah jelas.” | Responden 2 |

| | | | | |
|---|---|---|---|---|
| | Sudah menerapkan *clean architecture*, namun struktur foldernya masih belum terorganisir. | Prinsip *clean architecture* lebih terorganisir dan kelihatan karena pembagiannya jelas. | “Kode baru lebih terorganisir karena pembagiannya jelas” | Responden 3 |
| Modularitas Kode | Sudah menerapkan dasar modularitas kode namun masih belum sempurna. | Sudah menerapkan modularitas kode dengan baik di mana masing-masing modul seperti artikel, lokasi, *chatbot*, dan *scanner* sudah dipisahkan sesuai dengan tanggung jawabnya. | "Untuk yang baru, ternyata di dalam *folder data* ada *folder* lagi yang memisahkan model dan *repositories*, dan sudah jelas kegunaannya sesuai nama *folder*, beda sama yang sebelumnya." | Responden 1 |
| | Modularitas rendah, menyulitkan kolaborasi. | Modularitas sangat baik, memudahkan pengembangan paralel dan skalabilitas. | “Struktur modular bikin pengembangan lebih terfokus dan *scalable*.” | Responden 2 |
| | Modularitas sedikit lebih baik dari segi keterbacaan, karena folder dipisahkan per fitur. | Sudah baik, namun sedikit tidak familiar karena dipisahkan per *layer* bukan per fitur. | “Sempat bingung di awal untuk menilik kode yang *refactor* karena harus melihat di masing-masing *folder* untuk mencari masing-masing *file* yang berhubungan” | Responden 3 |

| | | | | |
|---|---|---|---|---|
| *Testability* Kode | Lebih sulit untuk dilakukan *unit test* karena logika dari kode masih bertumpuk. | Lebih mudah untuk dilakukan *unit test* karena struktur kode sudah lebih modular dan terpisah-pisah. | "Nah, kalau untuk yang lama, kalau mau dilakukan *unit test* akan lebih butuh *effort* karena struktur kodenya masih numpuk-numpuk, beda sama yang baru yang lebih *compact* dan lebih mudah buat dilakukan *unit test*." | Responden 1 |
| | Tidak ada penerapan *test*ing sama sekali. | *Test* menjadi terpusat karena *Repository Pattern.* | "Penerapan *Repository Pattern* bikin *unit test* bisa dilakukan dengan lebih efisien." | Responden 2 |
| | Lebih sulit untuk dilakukan *test*ing karena kode yang campur dan Panjang. | Lebih mudah untuk dilakukan *test*ing karena kode telah dibagi per layer atau unit yang kecil. | "Di kode yang baru *test*ing lebih mudah untuk dilakukan karena struktur yang sudah dipisah-pisah per *layer* sesuai dengan prinsip kita untuk melakukan tes yaitu lebih baik kode yang pendek" | Responden 3 |

### 4.4.3 Kesimpulan Wawancara

Dari wawancara yang dilakukan, dapat disimpulkan bahwa kode baru sudah menerapkan prinsip *clean architecture* dengan baik, di mana setiap komponen seperti *data layer*, *UI layer*, dan *viewmodel* dipisahkan dengan jelas, memudahkan pengembang baru untuk memahami struktur kode. Selain itu, modularitas kode pada kode baru juga jauh lebih baik, dengan pemisahan modul yang jelas sesuai dengan tanggung jawab masing-masing, seperti artikel,

lokasi, dan *chatbot*. Hal ini membuat kode lebih mudah dikelola dan dikembangkan. Sementara itu, kode lama cenderung masih terstruktur secara tumpuk dan tidak memisahkan komponen-komponen tersebut secara jelas, yang membuatnya sulit untuk dipahami dan diuji, membutuhkan lebih banyak usaha saat melakukan *unit test*.

Dengan penerapan modularitas yang lebih baik pada kode baru, *testability* juga meningkat. *Unit test* menjadi lebih mudah dilakukan karena struktur kode yang lebih compact dan terpisah-pisah. Sebaliknya, pada kode lama, *unit test* membutuhkan lebih banyak *effort* karena kode yang masih bertumpuk-tumpuk dan belum terorganisir dengan baik. Secara keseluruhan, kode baru memiliki struktur yang lebih jelas, memudahkan pemeliharaan dan pengembangan lebih lanjut, serta lebih siap untuk diuji. Dokumentasi wawancara dapat dilihat di Lampiran D.

## 4.5 Kesimpulan Hasil Penelitian

Secara kuantitatif, penerapan *Test-Driven Development* (TDD) terbukti menurunkan nilai metrik-metrik yang berkontribusi terhadap kompleksitas dan rendahnya *testability* kode. Hal ini terlihat dari hasil perhitungan terhadap lima metrik utama, yaitu RFC, LOC, LOCCOM, WMC, dan NMC. Penurunan signifikan pada metrik WMC dan NMC menunjukkan bahwa logika metode menjadi lebih sederhana dan interaksi antarkomponen lebih terkendali. Selain itu, nilai RFC yang lebih rendah mencerminkan penurunan tingkat *coupling* dalam kelas, sedangkan LOC yang mengecil menunjukkan ukuran kelas yang lebih ramping dan efisien. Hasil ini menunjukkan bahwa struktur kode menjadi lebih modular, ringan, dan mudah diuji.

Secara kualitatif, peningkatan *testability* juga diperkuat melalui hasil wawancara dengan para *expert*. Mereka menyatakan bahwa kode hasil penerapan TDD telah memenuhi prinsip *clean architecture* dengan pemisahan tanggung jawab yang jelas antara *data layer, UI layer,* dan *viewmodel*. Modularitas kode juga meningkat, dengan pembagian fungsi dan kelas sesuai domain fitur, sehingga memudahkan proses pengujian unit secara terisolasi. Struktur kode baru dinilai lebih mudah dipahami, dipelihara, dan dikembangkan, yang secara langsung menunjukkan peningkatan tingkat keterujian (*testability*) dibandingkan dengan versi kode sebelumnya.

Dengan demikian, baik dari sisi perhitungan metrik maupun evaluasi para ahli, penerapan TDD secara konsisten memberikan dampak positif terhadap peningkatan *testability* kode pada aplikasi Android kompleks seperti Sedonor.

# BAB V
# KESIMPULAN

## 5.1 Kesimpulan

Penelitian ini dilakukan untuk menjawab rumusan masalah mengenai bagaimana pengaruh penerapan *Test-Driven Development* (TDD) terhadap *testability* kode pada aplikasi Android kompleks, serta apakah kode yang dikembangkan dengan TDD memiliki *testability* yang lebih baik dibandingkan dengan kode non-TDD, ditinjau dari metrik-metrik seperti *Response for Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), dan *Number of Method Calls* (NMC). Penelitian ini menggunakan studi kasus pada aplikasi Sedonor, yang dikembangkan ulang dengan pendekatan TDD dan kemudian dibandingkan terhadap versi awalnya menggunakan metode kuantitatif (pengukuran metrik) dan kualitatif (*expert review*).

Hasil dari penelitian menunjukkan bahwa penerapan *Test-Driven Development* (TDD) pada aplikasi Android kompleks memberikan pengaruh positif terhadap *testability* kode. Penerapan TDD yang dimulai dengan penulisan *unit test* sebelum implementasi dan dilanjutkan dengan *refactoring* terbukti mampu menghasilkan struktur kode yang lebih modular, terpisah, dan mudah diuji. Hal ini tercermin dari penurunan rata-rata nilai metrik *testability*, yaitu RFC turun 22,3%, LOC turun 29,6%, WMC turun 70,4%, dan NMC turun 54,8%. Selain itu, hasil *expert review* juga mendukung temuan tersebut, dengan pernyataan bahwa kode hasil TDD lebih bersih, mudah dipahami, serta memenuhi prinsip *clean architecture*, sehingga lebih mudah untuk dilakukan pengujian unit secara terisolasi.

Dibandingkan dengan kode non-TDD, kode yang dikembangkan menggunakan prinsip TDD terbukti memiliki *testability* yang lebih baik jika ditinjau dari lima metrik utama, yaitu *Response for Class* (RFC), *Lines of Code* (LOC), *Lines of Comment* (LOCCOM), *Weight Method per Class* (WMC), dan *Number of Method Calls* (NMC). Penurunan signifikan pada sebagian besar metrik menunjukkan perbaikan kualitas struktural kode, seperti pada fitur riwayat donor yang mengalami penurunan RFC sebesar 83%, WMC sebesar 88%, dan LOC sebesar 82%, serta fitur *chatbot* yang mencatat penurunan WMC sebesar 75% dan NMC sebesar 64%. Walaupun terdapat sedikit peningkatan pada metrik RFC di fitur artikel kesehatan, hal tersebut tetap diimbangi dengan penurunan WMC dan NMC, yang menandakan peningkatan modularitas.

Secara keseluruhan, penerapan TDD terbukti memberikan pengaruh positif terhadap *testability* kode pada aplikasi Android kompleks. Kode hasil TDD lebih modular, ringan, dan mudah diuji dibandingkan kode non-TDD, sebagaimana dibuktikan melalui penurunan metrik dan dukungan hasil *expert review*. Temuan ini menunjukkan bahwa TDD adalah pendekatan yang efektif untuk meningkatkan *testability* kode.

Bagi pihak pengembang aplikasi Sedonor, hasil penelitian ini memberikan *insight* langsung terhadap kualitas dan struktur kode yang telah digunakan. Dengan diterapkannya pendekatan TDD, pengembangan ulang menunjukkan perbaikan yang signifikan dalam hal *testability* kode. Temuan ini dapat dijadikan dasar untuk mengadopsi praktik TDD secara lebih luas dalam proses pengembangan aplikasi Sedonor ke depan, guna memastikan keberlanjutan sistem, efisiensi pemeliharaan, dan kesiapan integrasi terhadap layanan pihak ketiga. Hasil yang lebih komprehensif dapat dilihat di *Documentation Guide* seperti pada Lampiran E.

## 5.2 Saran

Dalam subbab ini, penulis menyampaikan saran untuk pengembangan sistem di masa mendatang serta saran untuk penelitian lanjutan. Kedua saran tersebut disusun berdasarkan temuan dan keterbatasan yang diperoleh selama proses penelitian.

### 5.2.1 Saran Terhadap Pengembangan

Berdasarkan temuan selama proses penerapan *Test-Driven Development* (TDD), disarankan agar pengembangan fitur yang berinteraksi dengan sumber data eksternal, seperti *remote database* maupun *third-party* API, menggunakan pendekatan arsitektur berlapis (*layered architecture*). Pendekatan ini terbukti efektif dalam meningkatkan aspek *testability*, memisahkan tanggung jawab secara lebih jelas, serta memudahkan proses *refactoring* dan *mocking* selama pengujian unit dilakukan. Dalam pengembangan ulang aplikasi Sedonor, pola ini telah diimplementasikan secara konsisten pada lima fitur yang memiliki integrasi dengan layanan eksternal, antara lain fitur artikel kesehatan, *chatbot* sedonor, riwayat donor darah, *check-in* donor, dan pengingat kesehatan. Seluruh fitur tersebut terhubung dengan layanan seperti Firebase Firestore dan OpenAI API. Dengan menerapkan struktur lapisan yang terdiri atas *repository, data source,* dan *viewmodel*, integrasi terhadap pihak ketiga menjadi lebih modular serta tidak mengganggu logika utama aplikasi. Struktur ini tidak hanya mempermudah pengujian, tetapi juga memungkinkan kode untuk digunakan kembali (*reusable*) serta lebih

mudah dikembangkan dalam jangka panjang. Oleh karena itu, pola arsitektur berlapis ini direkomendasikan sebagai acuan pengembangan untuk fitur serupa di masa yang akan datang.

### 5.2.2 Saran Terhadap Penelitian Selanjutnya

Untuk penelitian selanjutnya, disarankan agar pendekatan TDD tidak hanya diterapkan pada proyek berskala kecil atau menengah, tetapi juga diuji pada proyek perangkat lunak yang lebih besar dan tim yang lebih kompleks guna melihat skalabilitas manfaatnya. Selain itu, integrasi dengan *automated testing framework* lain seperti Espresso atau Mockito juga dapat dieksplorasi lebih dalam untuk mendukung pengujian yang lebih menyeluruh. Penelitian ini juga terbatas pada pengukuran aspek *testability* saja; maka dari itu, studi lebih lanjut dapat meneliti pengaruh TDD terhadap aspek lain seperti *maintainability, performance, atau time-to-market* dari sebuah aplikasi.

# DAFTAR PUSTAKA


Abran, A., Bourque, P., Dupuis, R., & Moore, J. W. (2001). *Guide to the software engineering body of knowledge-SWEBOK.*

Agha, D., Sohail, R., Meghji, A., Qaboolio, R., & Bhatti, S. (2023). Test Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review. *VAWKUM Transactions on Computer Sciences*, *11*, 268–280. https://doi.org/10.21015/vtcs.v11i1.1494

Alenezi, M. (2022). *Investigating Software Testability and Test cases Effectiveness*.

Aniche, M. (2019). *CK: Code metrics for Java code by means of static analysis*. Https://Github.Com/Mauricioaniche/Ck. https://github.com/mauricioaniche/ck

Beck, K. (2003). *Test-driven Development: By Example*. Addison-Wesley. https://books.google.co.id/books?id=CUlsAQAAQBAJ

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Efatmaneshnik, M., & Ryan, M. (2017). A STUDY OF THE RELATIONSHIP BETWEEN SYSTEM TESTABILITY AND MODULARITY. *INSIGHT*, *20*, 20–24. https://doi.org/10.1002/inst.12140

Garousi, V., Felderer, M., & Kılıçaslan, F. N. (2019). A survey on software testability. In *Information and Software Technology* (Vol. 108, pp. 35–64). Elsevier B.V. https://doi.org/10.1016/j.infsof.2018.12.003

Gəruslu, V., Felderer, M., & Kilicaslan, F. (2018). *What we know about software testability: a survey*.

Grano, G., De Iaco, C., Palomba, F., & Gall, H. C. (2020). Pizza versus Pinsa: On the Perception and Measurability of Unit Test Code Quality. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 336–347. https://doi.org/10.1109/ICSME46990.2020.00040

Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall. https://books.google.co.id/books?id=hjEFCAAAQBAJ

Martins, L., Pontillo, V., Costa, H., Ferrucci, F., Palomba, F., & Machado, I. (2023). *Test Code Refactoring Unveiled: Where and How Does It Affect Test Code Quality and Effectiveness?* https://doi.org/10.48550/arXiv.2308.09547

Mona, J. (2022). Software Testability (Its Benefits, Limitations, and Facilitation). *Next Generation of Internet of Things*.

Naufal, M. I., #1, I., #2, A., & Sari, Z. (2023). Dampak Test-Driven Development pada Kualitas Kode. *JEPIN (Jurnal Edukasi Dan Penelitian Informatika)*.

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann Publishers Inc.

Rahman, M., Saha, A., Chakraborty, U., Sujana, H., & Shafi, S. M. A. (2024). Evaluating the impact of Test-Driven Development on Software Quality Enhancement. *International Journal of Mathematical Sciences and Computing*, *10*, 51–76. https://doi.org/10.5815/ijmsc.2024.03.05

SonarSource. (2017). *SonarQube for IDE*. Https://Github.Com/SonarSource/Sonarlint-Vscode.

Venkatesan, P., Rozario, R., & Fiaidhi, J. (2020). *Junit framework for unit testing.pdf*. https://doi.org/10.36227/techrxiv.12092259.v1

Vijaywargi, A., & Boddapati, U. K. (2024). EMBRACING TEST-DRIVEN DEVELOPMENT (TDD) IN MODERN ANDROID APPLICATIONS. *International Research Journal of Modernization in Engineering Technology and Science*, *6*, 11557. https://doi.org/10.56726/IRJMETS53809

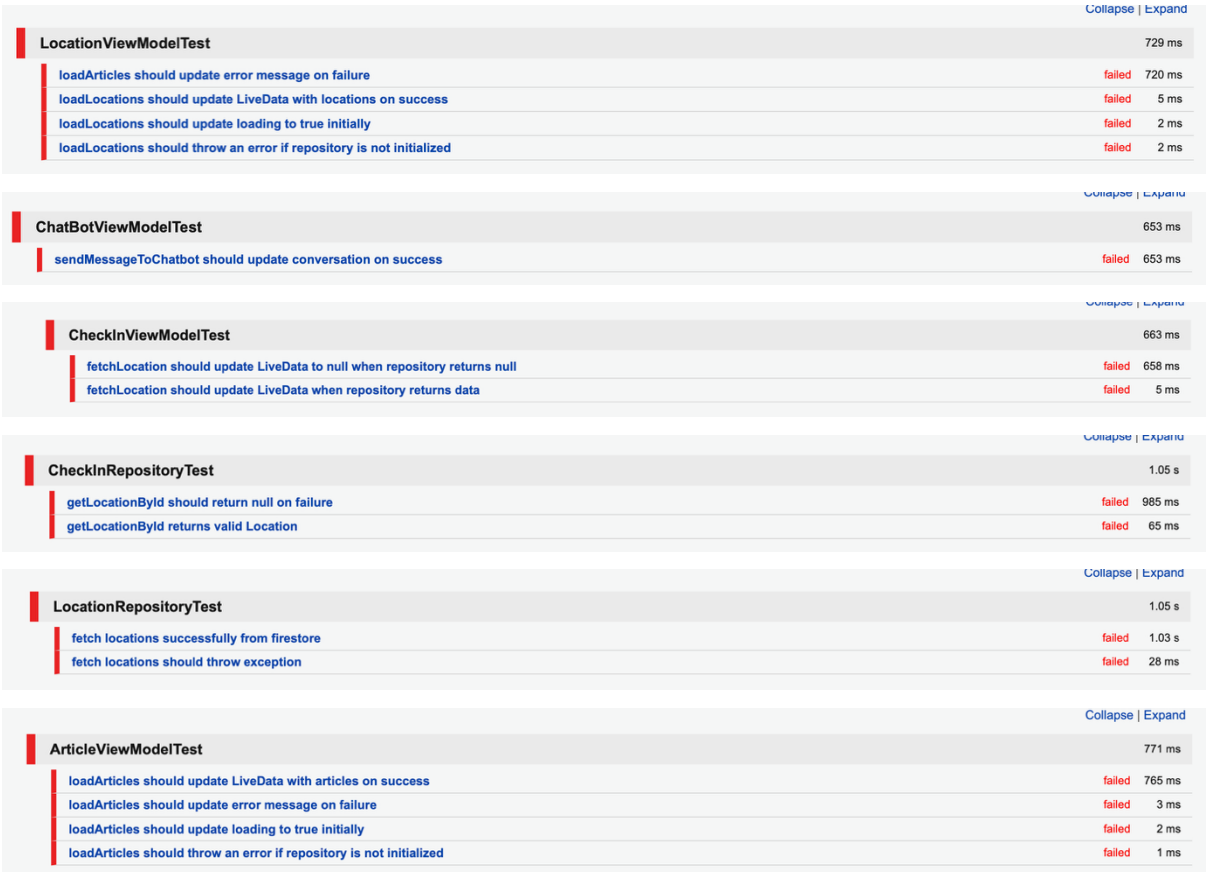
Zhou, Y., Leung, H., Song, Q., Zhao, J., Lu, H., Chen, L., & Xu, B. (2012). An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. *Science China Information Sciences*, *55*(12), 2800–2815. https://doi.org/10.1007/s11432-012-4745-x

# LAMPIRAN

## Lampiran A. Hasil Pengukuran Metrik dari *library* CK

| class | type | cbo | cboModified | fanin | fanout | wmc | dit | noc | rfc | lcom |
|---|---|---|---|---|---|---|---|---|---|---|
| com.example.sedonortdd.data.models.Article | class | 5 | 9 | 4 | 5 | 41 | 1 | 0 | 6 | 0 |
| com.example.sedonortdd.ui.lokasi.LokasiAdapter$MyV | innerclass | 12 | 12 | 0 | 12 | 5 | 2 | 0 | 13 | 1 |
| org.gradle.accessors.dm.LibrariesForLibs$AndroidxNa | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 1 | 1 |
| org.gradle.accessors.dm.LibrariesForLibs$AndroidxLife | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 1 | 1 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 5 | 5 | 0 | 5 | 3 | 2 | 0 | 0 | 3 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 6 | 1 |
| VersionAccessors | innerclass | 2 | 4 | 2 | 2 | 26 | 1 | 0 | 1 | 325 |
| org.gradle.accessors.dm.LibrariesForLibs$KotlinPlugin | innerclass | 5 | 5 | 0 | 5 | 2 | 2 | 0 | 1 | 1 |
| org.gradle.accessors.dm.LibrariesForLibs$FirebaseFire | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 1 | 1 |
| com.example.sedonortdd.data.repositories.DonorHisto | class | 6 | 6 | 0 | 6 | 3 | 1 | 0 | 1 | 3 |
| com.example.sedonortdd.data.models.DonorHistory | class | 6 | 8 | 2 | 6 | 73 | 1 | 0 | 8 | 61 |
| org.gradle.accessors.dm.LibrariesForLibs$FirebaseCra | innerclass | 5 | 5 | 0 | 5 | 3 | 2 | 0 | 1 | 3 |
| com.example.sedonortdd.ui.lokasi.LokasiAdapter$OnIt | interface | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| com.example.sedonortdd.viewModel.CheckInViewMod | innerclass | 6 | 6 | 0 | 6 | 3 | 1 | 0 | 2 | 0 |
| AndroidPluginAccessors | innerclass | 2 | 3 | 1 | 2 | 2 | 1 | 0 | 1 | 1 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 5 | 5 | 0 | 5 | 3 | 2 | 0 | 6 | 3 |
| org.gradle.accessors.dm.LibrariesForLibs$BundleAcce | innerclass | 6 | 6 | 0 | 6 | 1 | 2 | 0 | 0 | 0 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 6 | 1 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 6 | 1 |
| com.example.sedonortdd.viewModel.ChatBotViewMod | class | 6 | 7 | 1 | 6 | 8 | 2 | 0 | 4 | 9 |
| ArticleAdapter | class | 13 | 16 | 3 | 13 | 8 | 3 | 0 | 12 | 1 |
| org.gradle.accessors.dm.LibrariesForLibs$AndroidxEsp | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 1 | 1 |
| com.example.sedonortdd.viewModel.ArticleViewModel | class | 6 | 7 | 1 | 6 | 7 | 2 | 0 | 2 | 13 |
| org.gradle.accessors.dm.LibrariesForLibs | class | 24 | 24 | 0 | 24 | 15 | 30 | 0 | 1 | 105 |
| com.example.sedonortdd.viewModel.LocationViewMod | class | 6 | 7 | 1 | 6 | 7 | 2 | 0 | 2 | 13 |
| AndroidxCameraLibraryAccessors | innerclass | 2 | 4 | 1 | 3 | 6 | 1 | 0 | 1 | 15 |
| com.example.sedonortdd.data.models.Location | class | 7 | 12 | 5 | 7 | 60 | 1 | 0 | 6 | 19 |
| org.gradle.accessors.dm.LibrariesForLibsInPluginsBloc | innerclass | 4 | 4 | 0 | 4 | 2 | 2 | 0 | 6 | 1 |
| com.example.sedonortdd.ui.chatbot.ChatBotActivity | class | 27 | 30 | 3 | 27 | 21 | 2 | 0 | 34 | 11 |
| KotlinxLibraryAccessors | innerclass | 2 | 5 | 2 | 3 | 2 | 1 | 0 | 0 | 1 |

## Lampiran B. Hasil *Test* Sebelum Di-*refactor*



Lampiran C. Hasil *Test* Setelah Di-*refactor*

**LocationViewModelTest** 753 ms

| | | |
|---|---|---|
| loadArticles should update error message on failure | passed | 729 ms |
| loadLocations should update LiveData with locations on success | passed | 16 ms |
| loadLocations should update loading to true initially | passed | 6 ms |
| loadLocations should throw an error if repository is not initialized | passed | 2 ms |

Collapse | Expand

**ArticleViewModelTest** 762 ms

| | | |
|---|---|---|
| loadArticles should update LiveData with articles on success | passed | 748 ms |
| loadArticles should update error message on failure | passed | 7 ms |
| loadArticles should update loading to true initially | passed | 6 ms |
| loadArticles should throw an error if repository is not initialized | passed | 1 ms |

**ChatBotViewModelTest** 617 ms

| | | |
|---|---|---|
| sendMessageToChatbot should update conversation on success | passed | 617 ms |

Collapse | Expand

**CheckInViewModelTest** 664 ms

| | | |
|---|---|---|
| fetchLocation should update LiveData to null when repository returns null | passed | 659 ms |
| fetchLocation should update LiveData when repository returns data | passed | 5 ms |

Collapse | Expand

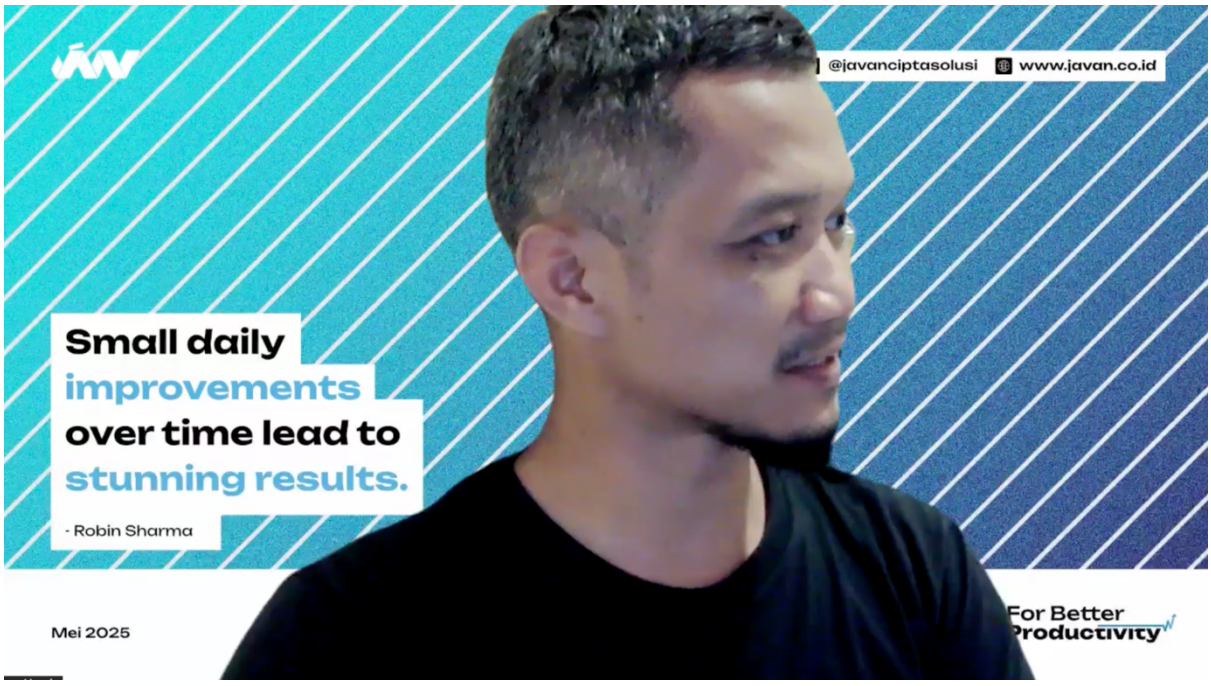
**LocationRepositoryTest** 1.08 s

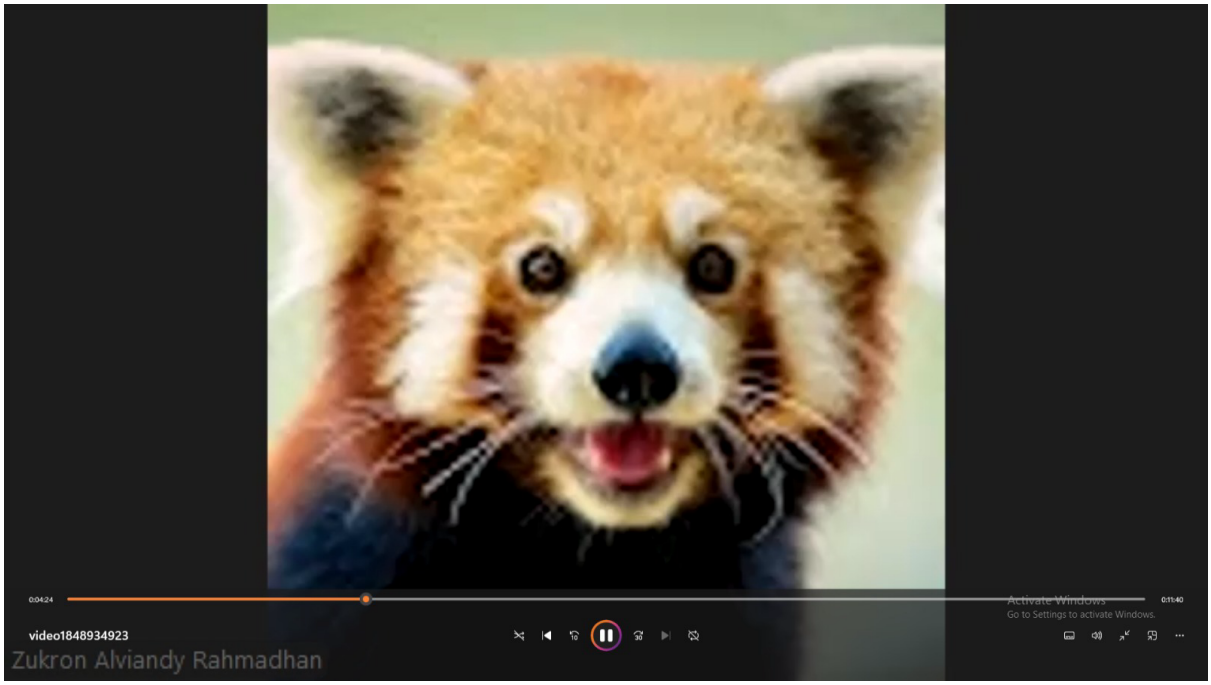
| | | |
|---|---|---|
| fetch locations successfully from firestore | passed | 1.05 s |
| fetch locations should throw exception | passed | 28 ms |

**CheckInRepositoryTest** 1.06 s

| | | |
|---|---|---|
| getLocationById should return null on failure | passed | 1.00 s |
| getLocationById returns valid Location | passed | 58 ms |

Lampiran D. Dokumentasi Wawancara






Lampiran E. *Documentation Guide* untuk Pihak Sedonor

# 🩸 SedonorRefactor

**SedonorRefactor** adalah aplikasi donor darah berbasis Android yang dikembangkan menggunakan Kotlin. Proyek ini merupakan hasil refaktorisasi dari aplikasi Sedonor sebelumnya, dengan fokus pada peningkatan struktur arsitektur, modularitas, dan testability.

## 🚀 Pendahuluan

Sedonor adalah aplikasi yang mempermudah masyarakat untuk melakukan donor darah, melihat lokasi donor terdekat, membaca artikel kesehatan, hingga menerima pengingat untuk donor berikutnya.
Refactor ini bertujuan untuk menyusun ulang kode agar lebih terstruktur dan mudah dikembangkan ke depannya, terutama dengan penerapan arsitektur bersih (clean architecture) dan praktik pengembangan perangkat lunak yang baik.

## 🛠️ Perubahan dan Peningkatan

### ✅ Modularisasi dan Clean Architecture

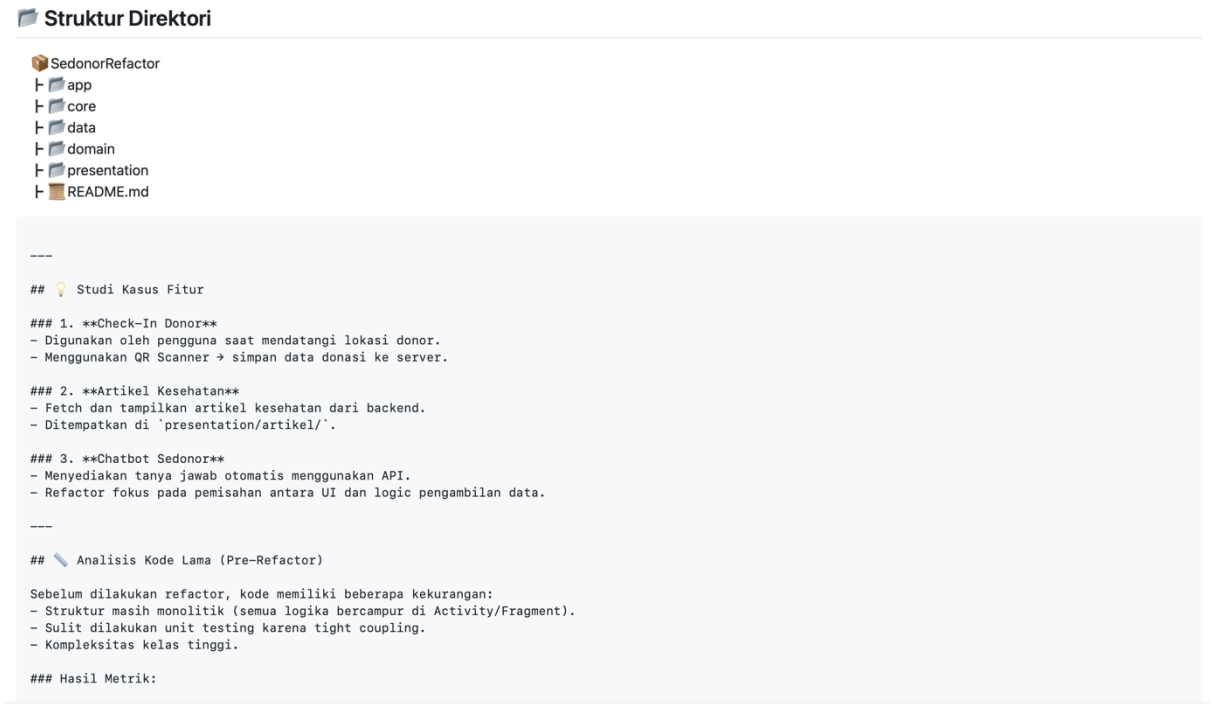
- `data/` : Berisi repository, implementasi API, dan data source.
- `domain/` : Berisi entitas dan use case (interactor).
- `presentation/` : Berisi UI (Activity, Fragment) dan ViewModel.
- `core/` : Utilitas umum, wrapper, dan kelas shared lainnya.

### ✅ MVVM Pattern

Menggunakan ViewModel dan LiveData untuk memisahkan logika UI dan data.

### ✅ Kode Lebih Testable

Setiap fitur dipisah ke dalam domain dan dapat diuji dengan lebih mudah menggunakan unit test.



📁 **Struktur Direktori**

📦SedonorRefactor
├ 📁app
├ 📁core
├ 📁data
├ 📁domain
├ 📁presentation
├ 📄README.md

```
---

## 💡 Studi Kasus Fitur

### 1. **Check-In Donor**
- Digunakan oleh pengguna saat mendatangi lokasi donor.
- Menggunakan QR Scanner → simpan data donasi ke server.

### 2. **Artikel Kesehatan**
- Fetch dan tampilkan artikel kesehatan dari backend.
- Ditempatkan di `presentation/artikel/`.

### 3. **Chatbot Sedonor**
- Menyediakan tanya jawab otomatis menggunakan API.
- Refactor fokus pada pemisahan antara UI dan logic pengambilan data.

---

## 🔪 Analisis Kode Lama (Pre-Refactor)

Sebelum dilakukan refactor, kode memiliki beberapa kekurangan:
- Struktur masih monolitik (semua logika bercampur di Activity/Fragment).
- Sulit dilakukan unit testing karena tight coupling.
- Kompleksitas kelas tinggi.

### Hasil Metrik:
```

Lampiran F. Surat Pengajuan Penelitian ke Sedonor

UNIVERSITAS ISLAM INDONESIA

FAKULTAS
TEKNOLOGI INDUSTRI

Gedung KH. Mas Mansur
Kampus Terpadu Universitas Islam Indonesia
Jl. Kaliurang km 14,5 Yogyakarta 55584
T. (0274) 898444 ext. 4100, 4101
F. (0274) 895007
E. fti@uii.ac.id
W. fti.uii.ac.id

Nomor : 118/Kaprodi.IF.S1/20/IF/VI/2025
Lamp : -
Hal : Permohonan Penelitian Tugas Akhir

Kepada Yth.
Pihak Pengembang Sedonor
di Jalan Kaliurang KM 14

*ikum Warahmatullahi Wabarokatuh*

Melalui surat ini Program Studi Informatika Program Sarjana Universitas Islam Indonesia memohon kepada Bapak/Ibu agar dapat mengizinkan mahasiswa kami untuk melaksanakan penelitian Tugas Akhir (TA) di institusi yang Bapak/Ibu pimpin. Pelaksanaan penelitian TA ini merupakan salah satu syarat lulus di program studi kami.

Mahasiswa yang akan melaksanakan penelitian adalah :

Nama : Muhammad Rizki Siraj
No. Mhs : 21523278

Kami berharap mahasiswa kami dapat melaksanakan penelitian di institusi Bapak/Ibu antara bulan Juni 2025 sampai dengan Juli 2025. Namun demikian, kepastian jadwal pelaksanaan kami serahkan sepenuhnya kepada institusi Bapak/Ibu.

Demikian permohonan kami, atas perhatian serta terkabulnya permohonan ini kami ucapkan terima kasih.

*ykum Warahmatullahi Wabarokatuh*

Yogyakarta, 24 Juni 2025
Ketua Program Studi Informatika - Program Sarjana, Fakultas Teknologi Industri

Ir. Dhomas Hatta Fudholi, S.T., M.Eng., Ph.D.