JITET (Jurnal Informatika dan Teknik Elektro Terapan)
Vol. 14 No. 3, pISSN: 2303-0577 eISSN: 2830-7062 http://dx.doi.org/10.23960/jitet.v14i3.9784

ANALISIS PENERAPAN METODE TEST-DRIVEN DEVELOPMENT (TDD) PADA PENGEMBANGAN APLIKASI KEUANGAN ANDROID

Fabian Nur Rohman[1*], Agung Susilo Yuda Irawan, M.Kom.[2], Arip Solehudin, M.Kom[3]

[1,2,3] Program Studi Informatika, Fakultas Ilmu Komputer, Universitas Singaperbangsa Karawang;
Jl. H.S. Ronggowaluyo, Telukjambe Timur, Karawang 41361; Telp. (0267) 641177

Keywords:
Test-Driven Development; Android; Code Coverage; Unit Testing; Aplikasi Keuangan.

Corespondent Email:
2210631170118@student.unsika.ac.id

Abstrak. Penelitian ini bertujuan untuk menganalisis penerapan metode Test-Driven Development (TDD) pada pengembangan aplikasi keuangan pribadi berbasis Android bernama InsightKu. Penelitian dilatarbelakangi oleh pentingnya menjaga kualitas, stabilitas, dan keakuratan logika bisnis pada aplikasi keuangan, khususnya pada fitur pencatatan transaksi dan perhitungan saldo otomatis. Penelitian menggunakan pendekatan studi kasus eksploratif dengan metode mixed-method embedded untuk mengevaluasi proses penerapan TDD, kualitas kode, efisiensi pengembangan, serta tantangan dan manfaat yang diperoleh. Implementasi dilakukan menggunakan bahasa pemrograman Kotlin dengan arsitektur Model-View-ViewModel (MVVM), sedangkan pengujian unit dilakukan menggunakan framework JUnit dan MockK. Penelitian difokuskan pada tiga modul utama, yaitu FinanceCalculator, TransactionRepository, dan Transaction ViewModel. Proses pengembangan mengikuti siklus Red-Green-Refactor dengan total 30 test case yang disusun sebelum implementasi kode dimulai. Hasil penelitian menunjukkan seluruh test case berhasil mencapai status lulus dengan success rate sebesar 100%. Sebanyak 19 bug berhasil dideteksi pada tahap awal sebelum kode diintegrasikan. Hasil pengukuran code coverage menunjukkan rata-rata sekitar 77,8% menggunakan Kover dan sekitar 70% menggunakan JaCoCo pada modul target. Temuan penelitian menunjukkan bahwa TDD efektif dalam meningkatkan kualitas, stabilitas, maintainability, dan keandalan logika inti aplikasi keuangan Android.

Abstract. *This study aims to analyze the implementation of the Test-Driven Development (TDD) method in the development of an Android-based personal financial application called InsightKu. The research was motivated by the importance of maintaining the quality, stability, and accuracy of business logic in financial applications, particularly in transaction recording and automatic balance calculation features. The study employed an exploratory case study approach using an embedded mixed-method to evaluate the TDD implementation process, code quality, development efficiency, as well as challenges and benefits. The implementation used the Kotlin programming language with the Model-View-ViewModel (MVVM) architecture, while unit testing was conducted using JUnit and MockK frameworks. The study focused on three main modules: FinanceCalculator, TransactionRepository, and TransactionViewModel. The development followed the Red-Green-Refactor cycle with 30 test cases designed before code implementation. The results showed all test cases passed with a 100% success rate. 19 bugs were detected at the early development stage. Code coverage averaged approximately 77.8% using Kover and 70% using JaCoCo on target modules. The findings indicate that TDD is effective in improving quality, stability, maintainability, and reliability of core logic in Android financial applications.*

1. PENDAHULUAN

Kualitas kode merupakan aspek fundamental dalam pengembangan perangkat lunak modern, terutama untuk sistem yang bersentuhan langsung dengan integritas data finansial pengguna. Seiring meningkatnya kompleksitas sistem mobile, pendekatan pengembangan yang menjamin kestabilan, akurasi, dan kemudahan pemeliharaan (*maintainability*) menjadi semakin krusial [1].

Dalam pengembangan aplikasi *Android*, pengujian perangkat lunak menjadi tantangan yang persisten karena keragaman perangkat keras, versi sistem operasi, dan ketergantungan terhadap komponen *framework* yang tidak selalu tersedia di lingkungan pengujian lokal [1]. Pendekatan pengujian konvensional yang dilakukan setelah implementasi selesai sering kali terlambat mendeteksi bug logika, sehingga biaya perbaikan menjadi jauh lebih tinggi.

Test-Driven Development (TDD) adalah metode pengembangan perangkat lunak yang mewajibkan penulisan test terlebih dahulu sebelum implementasi kode dilakukan. Siklus *Red–Green–Refactor* yang menjadi inti *TDD* memastikan setiap unit kode dikembangkan dengan spesifikasi yang jelas, terverifikasi secara otomatis, dan dapat direfaktorkan dengan aman [2]. Studi Siraj & Setiani (2024) menunjukkan *TDD* mampu menurunkan jumlah *bug* hingga 45% dan mempercepat proses *debugging* sebesar 30% pada proyek *Android* [2].

Sektor *Financial Technology (Fintech)* memerlukan standar kualitas tertinggi. Bank Indonesia memproyeksikan pertumbuhan transaksi digital sebesar 52,3% pada 2025 [3], dengan nilai transaksi *QRIS* mencapai Rp659,93 triliun pada 2024 [4]. Kesalahan perhitungan sekecil apapun pada aplikasi keuangan dapat berdampak langsung pada kerugian finansial pengguna.

Meskipun beberapa penelitian telah membahas *TDD* pada *platform Android* secara umum [2][5] dan pada *backend fintech* [6], belum ada yang secara khusus menganalisis penerapan *TDD* pada *core business logic* aplikasi keuangan *Android* menggunakan pendekatan studi kasus eksploratif yang holistik. Celah inilah yang menjadi fokus penelitian ini.

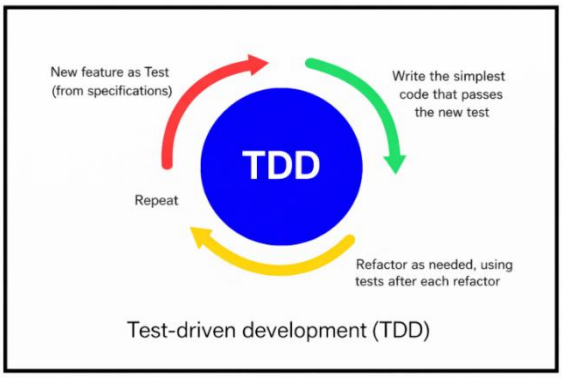
Tujuan penelitian adalah: (1) menganalisis proses penerapan TDD pada aplikasi keuangan Android; (2) menganalisis hasil metrik kualitas (code coverage, bug dini, durasi) dari penerapan TDD; dan (3) mengidentifikasi tantangan dan manfaat TDD pada konteks pengembangan Android.

2. TINJAUAN PUSTAKA

2.1 Test-Driven Development (TDD)

TDD merupakan pendekatan pengembangan iteratif yang dimulai dengan menulis tes terlebih dahulu sebelum implementasi kode *(test-first approach)* [2][7]. *TDD* beroperasi dalam siklus pendek *Red–Green–Refactor*:

- *Red* (Tes Gagal): Pengembang menulis test unit untuk fungsionalitas baru. *Test* dipastikan gagal karena implementasi belum ada, mengkonfirmasi bahwa *test* valid dan dapat mendeteksi kode yang absen.
- *Green* (Tes Lulus): Pengembang menulis kode implementasi seminimal mungkin agar *test* yang gagal menjadi lulus. Fokus pada fungsi, bukan kesempurnaan kode.
- *Refactor*: Kode dirapikan, dioptimalkan, dan disederhanakan tanpa mengubah perilaku. *Test suite* yang ada menjadi jaring pengaman yang menjamin refactoring aman.



Gambar 1. Siklus *Red-Green-Refactor* pada *TDD*

Manfaat utama *TDD* meliputi: peningkatan kualitas kode melalui desain yang

lebih modular, deteksi *bug* dini sebelum integrasi, dokumentasi otomatis berupa *test case*, dan kemudahan *maintainability* [6][8]. Studi Rahman et al. (2024) mengkonfirmasi bahwa *TDD* secara konsisten menghasilkan peningkatan metrik kualitas perangkat lunak pada berbagai konteks proyek [8].

2.2 Unit Testing dan Framework Pendukung

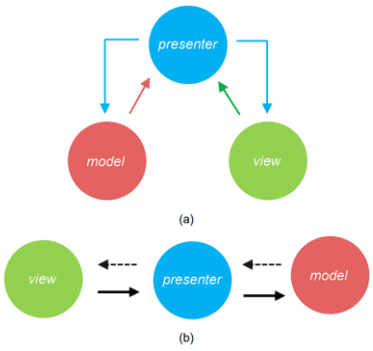
Unit Testing menguji komponen terkecil kode secara terisolasi dan independen. *JUnit 4* adalah *framework unit testing* standar untuk *Kotlin/Java* yang dijalankan di *JVM* lokal tanpa memerlukan *emulator Android*, memungkinkan eksekusi test yang sangat cepat [9]. MockK adalah *framework mocking Kotlin-first* yang memungkinkan pembuatan *mock object* untuk menggantikan dependensi eksternal seperti *database* atau *Android Context*, sehingga test benar-benar bersifat unit [9]. *InstantTaskExecutorRule* dari *androidx.arch.core:core-testing* diperlukan untuk menguji *LiveData* di *JVM* lokal dengan menggantikan *executor asinkron Android* menjadi eksekutor sinkron.

2.3 *Code Coverage* dan *Tools*

Code coverage mengukur persentase kode yang dieksekusi oleh rangkaian *test* [10]. *Kover* adalah *tool coverage* resmi *JetBrains* yang dioptimalkan untuk *Kotlin*, mengukur pada *level method call* dan mendukung *task Gradle* terintegrasi. *JaCoCo (Java Code Coverage)* versi 0.8.11 mengukur *coverage* pada level instruksi *bytecode Java*, menghasilkan metrik *line coverage, branch coverage*, dan *instruction coverage* [10]. Perbedaan metodologi keduanya menyebabkan selisih angka *coverage* yang dapat digunakan sebagai validasi silang.

2.4 Arsitektur *MVVM* pada *Android*

Model-View-ViewModel (MVVM) adalah pola arsitektur yang direkomendasikan *Google* untuk pengembangan aplikasi *Android*. Pola *MVVM* membagi sistem menjadi tiga komponen: *View* (antarmuka pengguna), *Model* (lapisan data dan logika bisnis), dan *ViewModel* (jembatan antara *Model* dan *View*). Penerapan *MVVM* memastikan bahwa *core business logic* pada *ViewModel* dapat diuji menggunakan *Unit Testing* (*JUnit* dan *MockK*) dengan cepat dan efisien, sepenuhnya mendukung siklus *Red-Green-Refactor* [11][12].



Gambar 2. Arsitektur *MVVM*

2.5 Penelitian Terdahulu

Hilmi et al. (2023) menerapkan *TDD* pada pengembangan *backend API fintech* syariah menggunakan metode eksperimen. Hasil menunjukkan seluruh *API backend* berhasil melewati seluruh skenario pengujian tanpa *error*, dan desain *endpoint API* menjadi lebih terstruktur [6]. Siraj & Setiani (2024) melakukan penelitian pada aplikasi *Android* yang menunjukkan *TDD* dapat menurunkan jumlah *bug* hingga 45%, mempercepat proses *debugging* 30%, dan meningkatkan *maintainability index* kode [2]. Rafiadly et al. (2023) mengembangkan aplikasi navigasi berbasis *Android* untuk tunanetra menggunakan metode *TDD*, menghasilkan tingkat keberhasilan 100% sesuai skenario pengujian [5]. Penelitian-penelitian tersebut menunjukkan bahwa belum ada yang secara khusus menganalisis penerapan *TDD* pada *core business logic* aplikasi keuangan *Android* dengan pendekatan studi kasus eksploratif yang holistik.

3. METODE PENELITIAN

3.1 Jenis dan Pendekatan Penelitian

Penelitian ini menggunakan jenis penelitian Studi Kasus Eksploratif (*Exploratory Case Study*) dengan pendekatan *mixed-method embedded* [13]. Pendekatan kuantitatif digunakan untuk mengumpulkan data metrik proses (durasi pengembangan, *code coverage*, jumlah *bug* awal) sebagai indikator efisiensi dan kualitas. Pendekatan kualitatif digunakan untuk menganalisis proses implementasi *TDD*, tantangan yang dihadapi (*reflexive analysis*), dan manfaat yang dirasakan dalam siklus *Red-*

Green-Refactor. Penelitian ini tidak bertujuan untuk melakukan perbandingan eksperimental antara *TDD* dan metode lain, melainkan mengeksplorasi bagaimana *TDD* bekerja dalam konteks nyata.

3.2 Objek Penelitian

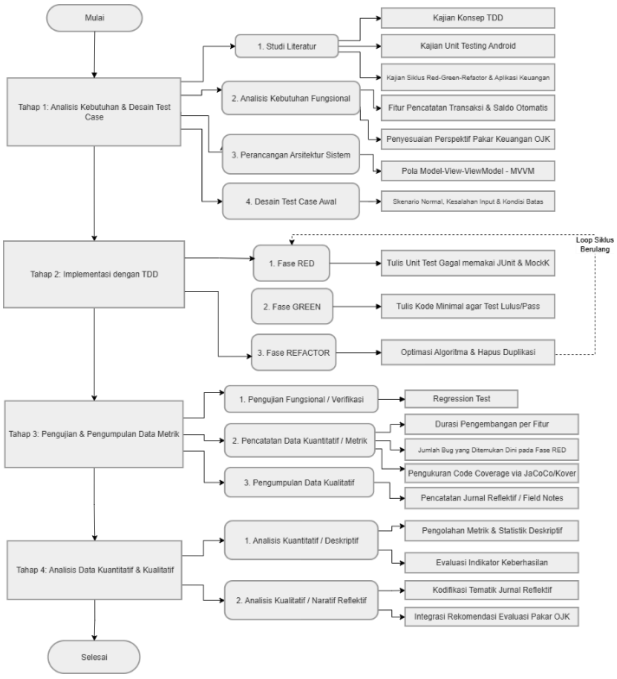
Objek penelitian adalah Aplikasi Keuangan Pribadi Berbasis Android bernama *InsightKu*. Pengembangan berfokus pada tiga modul core business logic:

Modul	***Layer***	**Deskripsi Fungsional**
Finance Calculator	*Domain Layer*	Logika perhitungan keuangan murni: *calculateBalance(), calculateTotalIncome(), calculateTotalExpense(). Pure function* tanpa dependensi eksternal.
Transaction Repository	*Data Layer*	Mengelola akses data ke *database SQLite* melalui *DBOpenHelper*. Menyediakan operasi *CRUD* dan mengekspos data melalui *LiveData*.
Transaction ViewModel	*ViewModel Layer*	Jembatan antara lapisan *UI* dan *TransactionRepository*. Mendelegasikan operasi *CRUD* ke *Repository* dan mengekspos *LiveData*.

Tabel 1. Objek Penelitian

3.3 Tahapan Penelitian

Penelitian ini terdiri dari empat tahapan utama yang terstruktur:



Gambar 3. Tahapan Metodologi Penelitian

1. Analisis Kebutuhan dan Desain *Test Case*: Studi literatur, analisis kebutuhan fungsional, perancangan arsitektur *MVVM*, dan desain *30 test case* sebelum implementasi dimulai.
2. Implementasi dengan *TDD*: Pengembangan fitur mengikuti siklus *Red-Green-Refactor* secara disiplin pada tiga modul utama.
3. Pengujian dan Pengumpulan Data Metrik: Pengujian fungsional, pencatatan data kuantitatif (durasi, *bug* dini, *code coverage* menggunakan *Kover* dan *JaCoCo*), dan pengumpulan data kualitatif melalui jurnal reflektif.
4. Analisis Data: Analisis kuantitatif menggunakan statistik deskriptif dan analisis kualitatif menggunakan teknik kodifikasi tematik untuk menjawab ketiga rumusan masalah.

3.4 Desain *Test Case*

Sebelum implementasi, 30 *test case* dirancang mencakup skenario normal, *edge case*, dan kondisi *error* pada tiga modul. Rincian *test case* disajikan pada Tabel 2.

Kode	Fungsi / *Method*	Modul	Skenario Pengujian
TC-01	*Calculate Balance (income, expense)*	*Finance Calculator*	Normal*: income > expense* → saldo positif
TC-02	*Calculate Balance (income, expense)*	*Finance Calculator*	Saldo nol: *income = expense*
TC-03	*Calculate Balance (income, expense)*	*Finance Calculator*	*Edge case*: *expense > income* (saldo negatif)
TC-04 – TC-09	*Calculate Balance(), Calculate TotalIncome(), calculate TotalExpense( )*	*Finance Calculator*	*Edge case*: nilai nol, *list* kosong, *single item*
TC-10 – TC-15	*insert(), update(), delete(), getById()*	*Transaction Repository*	Operasi *CRUD* via *MockK* – verifikasi delegasi dan *return value*
TC-16 – TC-21	*getAllTra nsactions() + LiveData state*	*Transaction Repository*	*State* dan reaktivitas *LiveData* setelah mutasi *insert/update/ delete, Observer* notifikasi
TC-22 – TC-26	*insert(), update(), delete(), getById(), getAll()*	*Transaction ViewModel*	Delegasi *ViewModel* ke *Repository – repo.method()* dipanggil tepat 1 kali
TC-27 – TC-30	*getAll(), getById(), Observer LiveData*	*Transaction ViewModel*	*State* dan reaktivitas *LiveData* via *ViewModel*, *Observer* menerima notifikasi
TOTAL	**30 *Test Case***		

Tabel 2. Desain Test Case TDD

4. HASIL DAN PEMBAHASAN

4.1 Proses Penerapan *TDD* – Siklus *Red-Green-Refactor*

4.1.1 Siklus 1: FinanceCalculator (Domain Layer)

Siklus 1 berhasil mendemonstrasikan ketiga fase *TDD* secara autentik. Pada fase *RED*, 9 *test case* (*TC-01* s.d. *TC-09*) ditulis terlebih dahulu sebelum implementasi, dan semua gagal dengan *NotImplementedError* karena *stub TODO()* belum diimplementasikan. Fase *RED* yang autentik ini mengkonfirmasi bahwa test berfungsi dengan benar.

Fase *GREEN* mengimplementasikan logika minimal: *calculateBalance()* dengan operasi *income - expense*, serta *calculateTotalIncome()* dan *calculateTotalExpense()* menggunakan *list.sum()* bawaan *Kotlin*. Implementasi ini sepenuhnya memuaskan 9 *test case*.

Fase *REFACTOR* mengubah *block body* menjadi *expression body* sesuai idiom *Kotlin* (*fun calculateBalance(...): Int = income - expense*) dan menambahkan *KDoc* formal. Seluruh 9 *test* tetap lulus setelah *refactoring*.

FinanceCalculator sebagai *pure function*—tanpa *side effect* dan dependensi eksternal—merupakan kandidat ideal untuk *TDD*. Seluruh logika dapat diverifikasi langsung melalui input-output tanpa kebutuhan *mock*.

4.1.2 Siklus 2: TransactionRepository (Data Layer)

Siklus 2 menghasilkan temuan penting berupa verifikasi retroaktif: kode *TransactionRepository* yang ada *(existing code)* ternyata sudah memenuhi spesifikasi 6 *test case* awal sehingga fase *RED* tidak terjadi natural. Ini membuktikan *TDD* tidak hanya berguna untuk kode baru, tetapi juga efektif sebagai alat audit kode *existing*.

Enam *test case LiveData* tambahan (*TC-16* s.d. *TC-21*) berhasil memverifikasi mekanisme reaktivitas: *state* awal, sinkronisasi setelah *insert/update/delete*, dan notifikasi *Observer*—komponen kritis arsitektur *MVVM*. Penggunaan *InstantTaskExecutorRule* memungkinkan pengujian *LiveData* di *JVM* lokal tanpa emulator.

Fase *REFACTOR* mengganti komentar emoji informal (// *load* awal) dengan *KDoc*

formal dan menghapus variabel sementara yang tidak diperlukan.

4.1.3 Siklus 3: TransactionViewModel (ViewModel Layer)

Siklus 3 merupakan yang paling menarik karena ditemukan *bug compile-time* sebelum *test* dapat dijalankan. Penggunaan *expression body (= TODO())* pada fungsi *non-Unit* menyebabkan *error* kompilasi: *'Nothing return type needs to be specified explicitly'*. *Bug* struktural ini terdeteksi bahkan sebelum fase *RED* dimulai, membuktikan *TDD* efektif mendeteksi kesalahan pada level kompilasi. Setelah diperbaiki ke *block body { TODO() }*, 9 *test* gagal dengan *NotImplementedError* (fase *RED* valid).

Fase *GREEN* mengimplementasikan *pure delegation pattern*—setiap *method ViewModel* hanya meneruskan pemanggilan ke *Repository*. Kesederhanaan ini bukan kelemahan; justru mengkonfirmasi prinsip *YAGNI* (*You Aren't Gonna Need It*) dan *desain thin ViewModel* yang tepat dalam *MVVM*.

Fase *REFACTOR* mengganti *unit method* menjadi *expression body* serta menambahkan *KDoc* pada semua *method*.

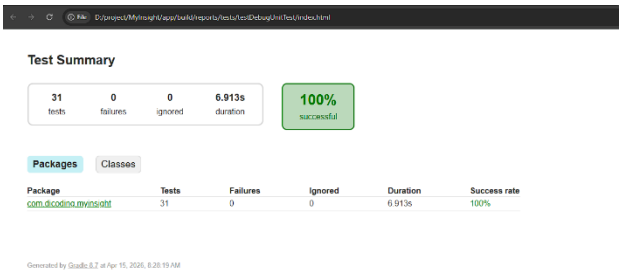
Siklus / Modul	Fase *RED*	Fase *GREEN*	Fase *REFACTOR*	*Bug* / *Coverage*
Siklus 1 ***Finance Calculator***	9/9 GAGAL – *NotImplemented Error* (autentik)	Implementasi: *income-expense, list.sum()*. 9/9 LULUS	*Block body* → *Expression body*. *KDoc* ditambahkan	*Bug*: 9 *runtime*. *Coverage*: 100% (*Kover* & *JaCoCo*)
Siklus 2 ***TransactionRepository***	Tidak terjadi natural – kode *existing* sudah valid. +6 *TC LiveData* baru	12/12 LULUS termasuk *TC LiveData* baru. *InstantTaskExecutorRule*	*Emoji comment* → *KDoc* formal. Hapus variabel sementara	*Bug*: 0 dini. *Coverage*: 58,3% *Method (Kover)* / 52% *Instr (JaCoCo)*
Siklus 3 ***TransactionViewModel***	1 *bug compile-time* + 9/9 GAGAL (*NotImplementedError*)	*Pure delegation* ke *Repository*. 9/9 LULUS	*Unit method* → *Expression body*. *KDoc* semua method	*Bug*: 10 (1 *compile* + 9 *runtime*). *Coverage*: 75% (*Kover*) / 58% (*JaCoCo*)

Tabel 3. Ringkasan Siklus *Red-Green-Refactor*

4.2 Hasil Metrik Kualitas dan Efisiensi

4.2.1 Final Run – Seluruh Test Case

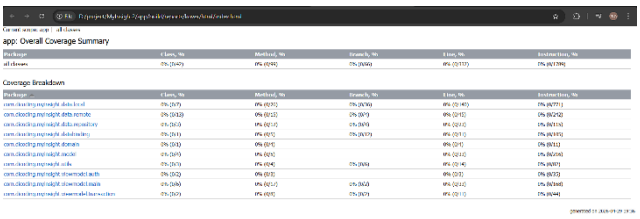
Final run dilaksanakan menggunakan perintah *gradlew testDebugUnitTest --rerun-tasks*. Seluruh 31 *test* (30 *TC* penelitian + 1 *ExampleUnitTest* bawaan *Android*) lulus dengan *success rate* 100% dalam durasi 6,913 detik—membuktikan efisiensi pengujian unit berbasis *JVM* lokal.



Gambar 4. Laporan *Final Run* – 31 *Tests*, 100% *Successful*

4.2.2 Code Coverage Sebelum TDD

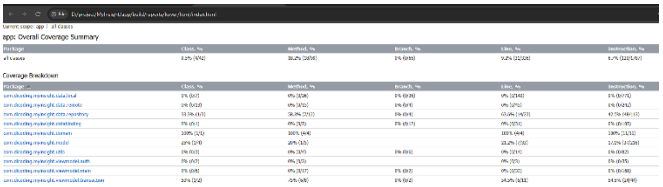
Laporan *Kover* dihasilkan pada kondisi baseline (sebelum *TDD*). Seluruh 42 kelas menunjukkan 0% *coverage* pada semua dimensi (*Class, Method, Line, Instruction*) karena tidak ada *unit test* yang berjalan.



Gambar 5. *Coverage Kover* Sebelum *TDD* (*Baseline* 0%)

4.2.3 Code Coverage Sesudah TDD – Kover

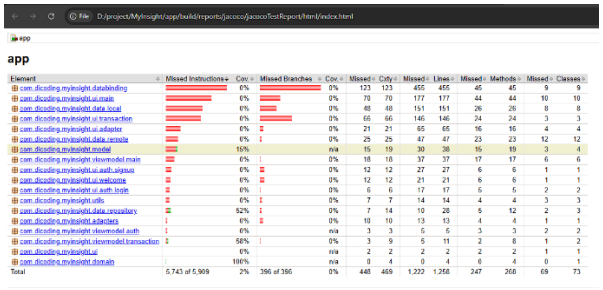
Setelah 30 *test case TDD* diimplementasikan, laporan *Kover* menunjukkan peningkatan signifikan pada ketiga modul target.



Gambar 6. *Coverage Kover* Sesudah *TDD*

4.2.4 Code Coverage Sesudah TDD – JaCoCo

JaCoCo 0.8.11 digunakan sebagai validasi silang dengan mengukur pada level instruksi *bytecode*.



Gambar 7. *Coverage JaCoCo* Sesudah *TDD*

Modul	Sebelum	*Kover*	*JaCoCo*	*Δ Kover*
Finance Calculator	0%	**100%**	**100%**	**+100%**
Transaction Repository	0%	58,3%	52%	+58,3%
Transaction ViewModel	0%	75%	58%	+75%
Rata-rata	**0%**	**~77,8%**	**~70%**	**+77,8%**

Tabel 4. Perbandingan *Coverage* Sebelum vs Sesudah *TDD*

Berikut rekap metrik yang dihasilkan pada keseluruhan penelitian yang dilakukan:

Metrik	Siklus 1 *Finance Calc*	Siklus 2 *Repository*	Siklus 3 *View Model*	TOTAL / RATA-RATA
Jumlah *Test Case*	9	12	9	**30**
Test LULUS (*Success Rate*)	9 (100%)	12 (100%)	9 (100%)	**30 (100%)**
Bug Dini *Runtime* (Fase *RED*)	9	0	9	18
Bug Compile-time	0	0	1	1
Coverage Kover (Method) – Sesudah *TDD*	100%	58,3%	75%	**~77,8%**
Coverage JaCoCo (Instruksi) – Sesudah *TDD*	100%	52%	58%	**~70%**
Coverage Kover (Method) – Sebelum *TDD*	0%	0%	0%	0%
Peningkatan *Coverage (Kover)*	**+100%**	**+58,3%**	**+75%**	**+77,8% rata-rata**
Durasi Total (Estimasi)	~14 mnt	~17 mnt	~14 mnt	~45 menit

Tabel 5. Rekap Metrik Keseluruhan Penelitian TDD

Perbandingan *code coverage* sebelum dan sesudah *TDD* menunjukkan peningkatan yang signifikan. Sebelum penerapan *TDD*, seluruh modul menunjukkan 0% *coverage* karena tidak ada *unit test* yang dijalankan. Sesudah *TDD*, *FinanceCalculator* mencapai 100% *coverage* pada kedua *tool*, mengkonfirmasi bahwa seluruh logika keuangan inti telah tercakup sepenuhnya. *Coverage* sebagian pada *Repository* (58,3%) dan *ViewModel* (75%) adalah konsekuensi yang dapat diterima dari *unit testing* berbasis *mock*: ketika *DBOpenHelper* di-*mock*, instruksi-instruksi di dalam kode *database* tidak dieksekusi secara nyata.

Overall coverage keseluruhan yang rendah (9,5% *Kover*, 2% *JaCoCo*) disebabkan cakupan pengukuran yang mencakup seluruh kelas aplikasi termasuk lapisan *UI* yang memang tidak menjadi scope penelitian. Jika dihitung hanya pada tiga paket target, rata-rata *coverage* mencapai ~77,8% (*Kover*) dan ~70% (*JaCoCo*).

4.3 Analisis *Bug* Dini

Total 19 *bug* berhasil dideteksi secara dini: 18 b*ug runtime (NotImplementedError)* pada fase *RED* Siklus 1 dan 3, serta 1 *bug compile-time* pada Siklus 3. Fakta bahwa 100% *bug* terdeteksi sebelum kode diintegrasikan merupakan bukti konkret efektivitas *TDD*. Dalam pengembangan tanpa *TDD*, *bug-bug* serupa kemungkinan baru terdeteksi saat *integration test* atau bahkan saat aplikasi dijalankan oleh pengguna.

Siklus	Jenis *Bug*	Jumlah	Fase Deteksi	Dampak Pencegahan
Siklus 1	*RuntimeError (Not Implemented Error)*	9	Fase *RED*	Mencegah *bug* kalkulasi keuangan masuk produksi
Siklus 2	Tidak ditemukan *bug* (kode *existing* valid)	0	—	Verifikasi retroaktif mengkonfirmasi kualitas kode *existing*
Siklus 3	*Compile-time ('Nothing' return type)* + *RuntimeError*	1 + 9 = 10	Sebelum *RED* + Fase *RED*	Mencegah *bug arsitektur ViewModel* dan *bug* struktural *Kotlin*
TOTAL	2 jenis *bug*	**19**	**Semua dini**	**100% sebelum diintegrasikan**

Tabel 6. Analisis *Bug* Dini yang Terdeteksi Selama Penerapan *TDD*

4.4 Pembahasan —Proses Penerapan *TDD*

Penerapan *TDD* dilaksanakan melalui *siklus Red-Green-Refactor* yang terstruktur pada tiga modul. Penggunaan *MockK* berhasil mengisolasi *TransactionRepository* dari *DBOpenHelper* yang membutuhkan *Android Context*, memungkinkan seluruh 30 *test* berjalan di *JVM* lokal tanpa *emulator* dalam kurang dari 7 detik. Ini membuktikan efektivitas pola *mocking* dalam mengatasi tantangan utama pengujian *unit Android*.

Variasi dalam kondisi fase *RED* antara siklus mencerminkan kondisi nyata pengembangan perangkat lunak: tidak selalu dimulai dari nol. Temuan pada Siklus 2 menunjukkan bahwa *TDD* tidak hanya berguna untuk pengembangan baru, tetapi juga sebagai alat audit kode *existing* (verifikasi retroaktif). Penambahan 6 *test case LiveData* membuktikan bahwa *test case* berfungsi sebagai spesifikasi yang aktif dan dapat diperluas sesuai kebutuhan.

4.5 Pembahasan — Metrik Kualitas dan Efisiensi

Coverage 100% pada *FinanceCalculator* mengkonfirmasi bahwa seluruh logika keuangan inti (*calculateBalance, calculateTotalIncome, calculateTotal Expense*) telah terverifikasi sepenuhnya. Ini sejalan dengan temuan Mylsamy & Jain (2025) yang menyatakan bahwa *TDD* menghasilkan struktur kode yang lebih modular dan memiliki tingkat *code coverage* yang lebih tinggi [14].

Perbedaan antara *Kover* (~77,8%) dan *JaCoCo* (~70%) disebabkan oleh metodologi pengukuran yang berbeda: *Kover* mengukur pada level *method call Kotlin*, sedangkan *JaCoCo* mengukur setiap instruksi *bytecode Java*. Keduanya menunjukkan tren yang konsisten, sehingga temuan dapat dianggap valid.

Meskipun *TDD* meningkatkan durasi pengembangan pada tahap awal (~45 menit untuk 3 siklus), stabilitas yang dihasilkan mampu mengurangi kebutuhan *debugging* berulang. Studi Siraj & Setiani (2024) menemukan bahwa TDD mempercepat proses *debugging* 30% lebih cepat [2], yang konsisten dengan temuan penelitian ini.

4.6 Pembahasan —Tantangan dan Manfaat *TDD*

Dari catatan jurnal reflektif, empat tantangan utama ditemukan:

- Isolasi dependensi *Android*: *DBOpenHelper* membutuhkan *Android Context* yang tidak tersedia di *JVM* lokal. Diatasi dengan *MockK*.
- *Bug compile-time Kotlin*: Penggunaan *expression body (= TODO())* pada fungsi *non-Unit* menyebabkan *compile error*, menuntut pemahaman mendalam terhadap *type system Kotlin*.
- *Coverage* layer *database*: Kode *DBOpenHelper* yang mengakses *SQLite* tidak dapat diuji melalui *unit test* berbasis *mock* (0% data.local).

- Kurva pembelajaran awal: Siklus 1 membutuhkan waktu lebih lama untuk setup infrastruktur *TDD*.

Di sisi manfaat, enam manfaat konkret diidentifikasi:

- Deteksi *bug* dini: 19 *bug* terdeteksi sebelum integrasi (100% dini).
- Peningkatan *code coverage*: Dari 0% menjadi rata-rata ~77,8% pada modul target.
- *Refactoring* aman: 3 siklus *refactoring* tanpa satu pun *test* gagal.
- *Test* sebagai dokumentasi aktif: 30 *test case* berfungsi sebagai spesifikasi yang selalu *up-to-date*.
- Desain kode lebih baik: *TDD* mendorong pemisahan *concern* dan prinsip *SOLID* secara alami.
- Verifikasi retroaktif: Kode *existing* (Siklus 2) tervalidasi kualitasnya melalui *test case*.

5. KESIMPULAN

Berdasarkan hasil penelitian dan pembahasan, dapat disimpulkan sebagai berikut:

a. Proses penerapan *TDD* berjalan secara sistematis melalui tiga layer utama (*domain, data*, dan *ViewModel*) menggunakan siklus *Red–Green–Refactor*. Penyusunan 30 *test case* sebelum implementasi terbukti berperan sebagai spesifikasi sistem yang terstruktur. Penggunaan *MockK* berhasil mengatasi tantangan isolasi dependensi *Android*. *TDD* terbukti tidak hanya berfungsi sebagai metode pengujian, tetapi juga sebagai pendekatan desain dalam pengembangan perangkat lunak.

b. Dari sisi kualitas dan efisiensi, seluruh 30 *test case* mencapai status lulus dengan *success rate* 100%. *Code coverage* rata-rata mencapai ~77,8% (*Kover*) dan ~70% (*JaCoCo*) pada tiga modul target, meningkat dari 0% sebelum *TDD*. Total 19 *bug* (18 *runtime* + 1 *compile-time*) berhasil dideteksi pada fase awal sebelum kode diintegrasikan. Meskipun *TDD* meningkatkan durasi pada tahap awal, stabilitas yang dihasilkan mampu mengurangi kebutuhan *debugging* berulang sehingga memberikan efisiensi jangka panjang.

c. Tantangan utama penerapan *TDD* adalah isolasi dependensi *Android* (diatasi dengan *MockK*), *bug compile-time Kotlin*, dan *coverage layer database* yang tidak dapat dijangkau *unit test* berbasis *mock*. Manfaat yang dominan meliputi deteksi *bug* dini, peningkatan *coverage* yang signifikan, kemudahan *refactoring* dengan jaminan *test suite*, serta ketersediaan *test case* sebagai dokumentasi sistem yang aktif.

Dengan demikian, *TDD* efektif dalam meningkatkan kualitas, stabilitas, dan keandalan pengembangan aplikasi keuangan Android, khususnya pada pengolahan logika inti seperti pencatatan transaksi dan perhitungan saldo

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Bapak Agung Susilo Yuda Irawan, M.Kom. dan Bapak Arip Solehudin, M.Kom. selaku dosen pembimbing yang telah memberikan arahan dan bimbingan selama penelitian ini. Terima kasih juga kepada Azmi Khoiri Faikar, S.Si. (*Senior Manager OJK*) atas validasi aspek keuangan dalam penelitian ini.

DAFTAR PUSTAKA

[1] T. Mahmud, M. C. Ngu, A. Yang, and G. Yang, "Why android app testing falls short: empirical insights from open-source projects and a practitioner survey," Empirical Software Engineering, vol. 30, no. 6, 2025, doi: 10.1007/s10664-025-10726-x.

[2] M. R. Siraj and N. Setiani, "Pengaruh Test-Driven Development terhadap Metrik Testabilitas Kode dalam Pengembangan Aplikasi Android," Jurnal Pendidikan dan Teknologi Indonesia, vol. 5, no. 2, 2025, doi: 10.52436/1.jpti.648.

[3] B. Kosasih, "BI Proyeksikan Transaksi Digital Indonesia Tumbuh 52,3%, Ini Sejumlah Tantangan dan Solusi di Tahun 2025," Vibiznews, Jan. 20, 2025. [Online]. Available: https://vibiznews.com.

[4] B. R. Alfathi, "Penggunaan QRIS Terus Meningkat, Nominal Transaksi Capai Rp659 Triliun," GoodStats Data, Apr. 14, 2025. [Online]. Available: https://data.goodstats.id.

[5] M. Rafiadly, R. Fauzi, and A. Musnansyah, "Perancangan Aplikasi Naviku untuk Memberikan Informasi Navigasi Kepada Tunanetra Menggunakan Metode Test Driven

Development," JOSH, vol. 4, no. 4, pp. 1455–1463, 2023, doi: 10.47065/josh.v4i4.3948.
[6] F. Hilmi, R. Fauzi, and E. N. Alam, "Penerapan Test Driven Development dalam Pengembangan Backend untuk Kebutuhan Data pada Aplikasi Peer-to-Peer Lending Syariah," JIPI, vol. 9, no. 2, pp. 568–577, 2024, doi: 10.29100/jipi.v9i2.4631.
[7] M. Marabesi et al., "Exploring TDD and Test Smells—A Systematic Literature Review," Computers, vol. 13, no. 3, 2024, doi: 10.3390/computers13030079.
[8] Md. S. Rahman et al., "Evaluating the impact of TDD on Software Quality Enhancement," IJMSC, vol. 10, no. 3, pp. 51–76, 2024, doi: 10.5815/ijmsc.2024.03.05.
[9] G. Ma, Y. Pei, L. Chen, C. Gan, H. Zhang, H. Liang, and T. Zhang, "Effective Unit Test Generation for Android Apps," in Proc. IEEE ICSME 2024, pp. 820–832, doi: 10.1109/ICSME58944.2024.00085.
[10] M. Imran, V. Cortellessa, D. Di Ruscio, R. Rubei, and L. Traini, "Is code coverage of performance tests related to source code features?," Empirical Software Engineering, vol. 30, no. 6, 2025, doi: 10.1007/s10664-025-10712-3.
[11] F. Pradana, R. I. Langundi, D. Pramono, and N. I. Iriani, "Comparative Analysis of MVVM and MVP Patterns Performance on Android Dashboard System," Jurnal Nasional Teknik Elektro dan Teknologi Informasi, vol. 14, no. 2, pp. 87–95, 2025, doi: 10.22146/jnteti.v14i2.18985.
[12] I. M. Riyadhi et al., "Penerapan MVVM pada Aplikasi Pengaduan Masyarakat Berbasis Android," INFOTECH, vol. 9, no. 1, pp. 147–158, 2023, doi: 10.31949/infotech.v9i1.5246.
[13] M. A. Storey et al., "Guiding principles for mixed methods research in software engineering," Empirical Software Engineering, vol. 30, no. 5, 2025, doi: 10.1007/s10664-025-10629-x.
[14] S. Mylsamy and A. Jain, "Test-Driven Development (TDD) and Code Quality," 2025, doi: 10.36676/URR.V12.