Note-Manager

Test-Driven Development Project Report
Advanced Programming Technique
6-Credit Exam

Prof. Lorenzo Bettini

Muhammed Irfan Cheriya Puthan Veettil
Matriculation No: 7127908
muhammed.cheriya@edu.unifi.it
Project Repository: https://github.com/Irfancpv99/Note-Manager
January 20, 2026

Contents

1 Project Overview and Motivation

This report documents the development of Note-Manager, a desktop note management application built using Test-Driven Development (TDD) principles. The application provides a complete note management system where users can create, update, and delete notes, organize them with categories, and manage their personal notes.

1.1 What This Application Does

The application manages personal notes with an organized category system. Users can create notes, edit them, delete them, and organize them by categories (Personal, Work, Study). The entire interface runs as a desktop application built with Java Swing.

The application uses MongoDB as its persistence layer, providing document storage for notes and categories. On startup, the application automatically initializes default categories if the database is empty.

1.2 TDD Methodology

The project follows the Red-Green-Refactor cycle at every level of development. Tests were written before implementation code, ensuring that each feature was validated against explicit behavioral specifications. This approach resulted in a multi-level test suite with three distinct layers

Unit tests verify individual components in isolation using mock dependencies. These tests focus on business logic within the service layer, controller behavior, and model validation. Mockito provides the mocking framework to ensure tests remain fast and independent of external systems.

Integration tests validate interactions between application layers using real MongoDB instances provided by Docker. These tests ensure that MongoDB document mappings, repository queries, and service layer interactions function correctly.

End-to-end tests exercise exercise the complete application stack, including database layer, service layer, and the Java Swing desktop interface. AssertJ Swing tests verify user workflows through simulated GUI interactions, validating complete scenarios such as creating notes, editing them, deleting and handling error cases.

2 Technical Architecture

2.1 System Structure

The application has four layers. Each layer only talks to the layers next to it.

Presentation Layer - The GUI built with Java Swing. MainFrame has the tables, lists, text fields, and buttons. Users click buttons, type text, and see results. No business logic here, just displaying stuff and handling clicks. Uses `SwingUtilities.invokeLater()` to keep everything thread-safe.

Controller Layer -NoteController sits between GUI and business logic. When users click "Save", the controller validates the input (non-empty text, valid category), calls the service, and returns the result to the view. It never touches databases directly.

Service Layer - NoteService handles the business logic. Creates notes, updates them, and manages deletions. The DatabaseInitializer service handles automatic category creation on first startup.

Repository Layer - NoteRepository and CategoryRepository are interfaces defining CRUD operations. MongoDB implementations (NoteMongoRepository, CategoryMongoRepository) handle actual database interactions using the MongoDB Java Driver.

2.2 Domain Model

Note has a text (required), categoryId (required), and an id assigned by MongoDB. Text and categoryId cannot be null or empty - the constructor and setters validate this, throwing IllegalArgumentException for invalid input. **Category** has a name cannot be null or empty. Categories are pre-defined (PERSONAL, WORK, STUDY) and initialized automatically by DatabaseInitializer.

Both domain classes implement proper equals() and hashCode() methods. When IDs are present, equality is based on ID. When IDs are null (before persistence), equality falls back to the main attribute (text for Note, name for Category).

2.3 Repository Pattern

abstracts data access behind clean interfaces. NoteRepository and CategoryRepository define standard CRUD operations. MongoDB implementations use the MongoDB Java Driver directly, with helper methods mapping documents to domain objects.

2.4 Technology Stack

Table 1 lists the complete technology stack.

Purpose	**Technology**
Language / Build	Java 17 + Maven
Database	MongoDB 6.0
MongoDB Driver	MongoDB Java Driver 4.9.1
GUI Toolkit	Java Swing
Unit Testing	JUnit 5.9.3 3 + JUnit 4.13.2 + Mockito 4.11.0
GUI Testing	AssertJ Swing 3.24.2
Code Coverage	JaCoCo 0.8.10
Mutation Testing	PITest 1.14.1
Code Quality	SonarCloud
Coverage Tracking	Coveralls
CI/CD	GitHub Actions
Local Dev Env	Docker Compose

Table 1: Technology stack overview

3 Test-Driven Development in Practice

3.1 The Red-Green-Refactor Workflow

Every feature started with a test. Write the test first, watch it fail (red), write code to make it pass (green), then clean up the code (refactor). This cycle repeated for every method, every class.

3.2 Testing Strategy: The Testing Pyramid

The project has three test levels - lots of fast unit tests at the bottom, some integration tests in the middle, and a few slow end-to-end tests at the top.

Unit Test tests one thing at a time using mock objects. Testing NoteService uses mock repositories that simulate database behavior. If the test fails, the bug is in the service, not the database. Unit tests run instantly - fast enough to run every time you save a file.

Integration Tests verify that components work together correctly. These tests use a real MongoDB instance started by Docker Maven Plugin. Testing the NoteMongoRepository means actually saving documents and reading them back. This catches bugs that unit tests miss, like incorrect MongoDB queries or document mapping errors.

End-to-End Tests test the whole application like a user would. AssertJ Swing simulates clicks, types text, and verifies what appears on screen. A real MongoDB database runs in the background. These tests validate that all layers work together correctly.

3.3 GUI Testing

Desktop GUIs are hard to test. Swing runs on a special thread. Tests need to wait for updates and check what's on screen. AssertJ Swing handles the complexity. It knows about Swing's threading and synchronizes everything automatically.

Every GUI component has a name set in code (e.g., "noteTextArea", "saveButton"). Tests use these names to find components reliably. If the layout changes, tests still work because names don't change. The GUI tests use GuiActionRunner.execute() to run Swing code on the correct thread. After actions like clicking buttons, robot().waitForIdle() ensures the GUI has finished updating before assertions run.

3.4 Real Database Testing with Docker

Integration tests use real MongoDB instances running in Docker containers. The Docker Maven Plugin starts MongoDB before integration tests and stops it afterward.

Each test class drops and recreates collections in setUp(), ensuring test isolation. This approach proves the code works with actual MongoDB behavior - not simulated behavior that might differ from production.

3.5 Mutation Testing

PITest verifies test quality by intentionally introducing bugs (mutations) into the code. It changes conditions like if (x > 5) to if (x >= 5) and reruns tests. If tests still pass, they're weak. If tests fail, they caught the mutation - good.

Example of strong test:

```
noteService.createNote("New note", "cat1");
Note result = noteService.createNote("New note", "cat1");
assertThat(result.getId()).isNotNull();
assertThat(result.getText()).isEqualTo("New note");
```

Each assertion kills specific mutations. If PITest removes the save() call or changes return values, these assertions catch it.

4 Exclusions and Justifications

4.1 Code Coverage Exclusions

NoteManagerApp.java - Main class containing only the main() method and Swing initialization. As explained in the book, main() methods are bootstrap code that cannot be meaningfully unit tested. The class only creates MongoDB client, repositories, services, controller, view, and initializer - all tested independently. E2E tests cover the complete application startup.

4.2 Mutation Testing Exclusions

- **View Layer** – Mutation testing focuses on business logic. The View contains UI setup code (Swing components, layouts, event listeners) where mutations produce equivalent or non-meaningful changes. View correctness is verified via unit tests with mocks and E2E tests with AssertJSwing.
- **Repository Layer** – Repository classes are infrastructure code delegating to the MongoDB driver. As recommended in the book, repositories should be tested with real databases via integration tests. Mutating repositories would target MongoDB API calls, not business logic. Correctness is verified through integration tests with Docker.
- **App Class** – Bootstrap code containing only the `main()` method with no business logic to mutate.

5 Quality Metrics and Results

5.1 Code Coverage with JaCoCo

JaCoCo is used to measure code coverage across all test types, including unit, integration, and end-to-end tests. The `NoteManagerApp` class is excluded from coverage requirements, as documented in Section 4.1, since it contains only application bootstrap logic.

Table 2 summarizes the final code coverage results. All included packages achieve 100% coverage across all reported metrics.

Coverage reports are generated automatically during the Maven `verify` phase and are uploaded to Coveralls, which tracks coverage trends over time. The project repository README includes a Coveralls badge displaying the current coverage percentage.

Table 2: JaCoCo Code Coverage Summary

Package	Instructions	Branches	Lines	Methods
model	100%	100%	100%	100%
service	100%	100%	100%	100%
controller	100%	100%	100%	100%
view.swing	100%	100%	100%	100%
repository.mongo	100%	100%	100%	100%
Total	**100%**	**100%**	**100%**	**100%**

5.2 Mutation Testing with PITest

PITest was executed on the `model`, `service`, and `controller` packages, which contain the core business logic of the application. All generated mutations were successfully killed, resulting in a 100% mutation score with zero surviving mutants.

Table 3 presents the mutation testing results aggregated by package.

Table 3: PITest Mutation Coverage by Package

Package	Line Coverage	Mutation Coverage	Mutation Strength
model	100%	100%	100%
service	100%	100%	100%
controller	100%	100%	100%
Total	**100%**	**100%**	**100%**

A mutation score of 100% indicates that every injected fault caused at least one test to fail. This demonstrates that the test suite effectively validates program behavior rather than merely executing code paths.

5.3 Static Analysis with SonarCloud

SonarCloud performs automated static code analysis on every push to the repository. It evaluates code quality by detecting code smells, bugs, security vulnerabilities, and by estimating technical debt. All checks pass the SonarCloud Quality Gate with the highest possible ratings.

.

Table 4 summarizes the SonarCloud analysis results.

.

The absence of technical debt and quality issues is a direct result of adhering to clean code principles, including meaningful naming, short methods with single responsibilities, consistent formatting, and proper error handling.

Table 4: SonarCloud Quality Gate Summary

Metric	Value
Code Smells	0
Bugs	0
Vulnerabilities	0
Security Hotspots	0 (Reviewed)
Technical Debt	0 minutes
Duplications	0%
Maintainability Rating	A
Reliability Rating	A
Security Rating	A

5.4 Continuous Integration Pipeline

A continuous integration (CI) pipeline is implemented using GitHub Actions and is triggered on every push to the `main` and `develop` branches, as well as on pull requests targeting the `develop` branch. The pipeline enforces automated quality checks to ensure that only validated code is merged.

The CI pipeline consists of the following stages:

1. **Build and Test with GUI Support**
   The project is built using Maven, and all tests are executed, including unit, integration, and end-to-end tests. Swing-based GUI tests are supported using `xvfb-run` to provide a virtual display environment. A MongoDB instance is started via the Docker Maven Plugin to support repository integration tests.

2. **Mutation Testing**
   PITest is executed against the `model`, `service`, and `controller` packages to evaluate the effectiveness of the test suite.

3. **Coverage Reporting**
   Code coverage data generated by JaCoCo is uploaded to Coveralls, enabling continuous tracking of coverage metrics over time.

4. **Static Analysis**
   SonarCloud performs static code analysis, including detection of code smells, bugs, and security issues, and enforces the project quality gate.

All pipeline stages must complete successfully for the build to pass. Any failure indicates a violation of quality requirements and must be resolved before code can be merged.

6 Running the Project

The project requires Java 17 or higher, Maven 3.6 or higher, and Docker for running MongoDB during tests and development. First line.

Clone the repository:

```
git clone git clone https://github.com/Irfancpv99/Note-Manager.git
cd Note-Manager
```

6.1 Eclipse Setup

Import the project into Eclipse:

1. Open Eclipse IDE
2. Select `File` → `Import` → `Maven` → `Existing Maven Projects`
3. Browse to the cloned `Note-Manager` directory
4. Select `pom.xml` and click `Finish`

Expected Result: Project imports with no errors or warnings

Verify Tests Run:

- Right-click `src/test/java`
- Right-click `src/it/java`
- Right-click `src/e2e/java`
- Select `Run As` → `JUnit Test`
- All tests should pass

Run Application from Eclipse:

- Right-click `NoteManagerApp.java`
- Select `Run As` → `Java Application`
- GUI window should open

Running from Terminal

```
mvn package
Start databases : mvn docker:start
Run application : mvn exec:java
                : java -jar target/note-manager-1.0.0-jar-with-
                  dependencies.jar
```

6.2 Running Tests

Execute only unit tests (faster):

```
mvn clean test
```

Execute all tests (unit, integration, and end-to-end):

```
mvn clean verify
```

Generate the code coverage report:

```
mvn clean verify jacoco:report
View: target/site/jacoco/index.html
```

Run mutation testing:

```
mvn clean test org.pitest:pitest-maven:mutationCoverage
View: target/pit-reports/index.html
```

7 Project Summary

Note-Manager demonstrates how to build software with proper testing practices. The project achieves all quality targets: 100% code coverage (except the main class), 100% mutation coverage, zero technical debt from SonarCloud, and tests at three levels (unit, integration, end-to-end). GitHub Actions runs quality checks automatically on every push.

The MongoDB-based persistence works well for a desktop note management application. The repository pattern provides clean separation between business logic and data access, making the codebase easy to test and maintain.

Test-Driven Development drove every feature. Every piece of code exists because a test required it. The test suite gives confidence to change code without breaking existing functionality. Red-Green-Refactor kept development disciplined throughout the project.

GUI testing with AssertJ Swing proved valuable. These tests caught integration bugs that unit tests missed - like timing issues with the Event Dispatch Thread and state management between view and controller. Using xvfb-run in CI makes GUI tests work in headless environments.

The combination of JaCoCo (100% coverage), PITest (100% mutation coverage), and SonarCloud (zero issues) demonstrates that the tests actually verify behavior. This project shows that good TDD practices create reliable, maintainable software.

Repository: https://github.com/Irfancpv99/Note-Manager

CI/CD: https://github.com/Irfancpv99/Note-Manager/actions

Coverage: https://coveralls.io/github/Irfancpv99/Note-Manager

Quality: https://sonarcloud.io/summary/new_code?id=Irfancpv99_Note-Manager