



UNIVERSITÀ
DEGLI STUDI
FIRENZE

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>

February 10, 2026

Contents

| | | |
|-----------|--|-----------|
| 1 | Project Overview and Motivation | 2 |
| 1.1 | What This Application Does | 2 |
| 1.2 | TDD Methodology | 2 |
| 2 | Technical Architecture | 2 |
| 2.1 | System Structure | 2 |
| 2.2 | Domain Model | 3 |
| 2.2.1 | Entity Relationship | 3 |
| 2.3 | Repository Pattern | 3 |
| 2.4 | Technology Stack | 3 |
| 3 | Design and Implementation Choices | 4 |
| 3.1 | Architecture Overview | 4 |
| 4 | Test-Driven Development in Practice | 5 |
| 4.1 | The Red-Green-Refactor Workflow | 5 |
| 4.2 | Testing Strategy: The Testing Pyramid | 5 |
| 4.3 | GUI Testing | 5 |
| 4.4 | Docker Configuration for Testing | 5 |
| 4.5 | Mutation Testing | 6 |
| 5 | Exclusions and Justifications | 6 |
| 5.1 | Code Coverage Exclusions | 6 |
| 5.2 | Mutation Testing Exclusions | 7 |
| 6 | Problem Encountered | 7 |
| 6.1 | Known Limitation: Java 17 JPMS Compatibility Warning | 7 |
| 6.2 | Docker port conflicts | 7 |
| 6.3 | JUnit 4 and JUnit 5 coexistence | 7 |
| 6.4 | Coveralls with Java 17 | 8 |
| 7 | Quality Metrics and Results | 8 |
| 7.1 | Code Coverage with JaCoCo | 8 |
| 7.2 | Mutation Testing with PITest | 8 |
| 7.3 | Static Analysis with SonarCloud | 9 |
| 7.4 | Continuous Integration Pipeline | 9 |
| 8 | Branching Strategy and CI Workflow | 10 |
| 9 | Running the Project | 10 |
| 9.1 | Eclipse Setup | 11 |
| 9.2 | Running Tests | 11 |
| 10 | Project Summary | 12 |

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 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 handling user interaction. Uses `EventQueue.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`).

2.2.1 Entity Relationship

The domain model consists of two entities: `Note` and `Category`, with a many-to-one relationship in which each `Note` belongs to exactly one `Category`.

In the MongoDB implementation, each `Note` document stores a `categoryId` field containing the string representation of the corresponding `Category`'s MongoDB `ObjectId`. The `NoteMongoRepository` provides a `findByCategoryId(String categoryId)` method, which queries notes using `Filters.eq("categoryId", categoryId)`.

This approach implements the relationship at the application level, following MongoDB's document-oriented model, where references between collections are stored as field values rather than enforced through foreign key constraints as in relational databases.

For presentation, the `NoteSwingView` resolves category names by maintaining a `categoryIdToName` map, which is populated when `showAllCategories()` is executed. The list cell renderer uses this mapping to display each entry in the format:

Note: [text] | Category: [name]

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 + JUnit 4.13.2 + Mockito 4.11.0 |
| GUI Testing | AssertJ Swing 3.17.1 |
| 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 Design and Implementation Choices

3.1 Architecture Overview

The application follows the Model–View–Controller (MVC) pattern with an additional service layer between the controller and repository. This structure matches the approach used in the course book, where the controller handles user input, the service contains business logic, and repositories manage data access.

- **View abstraction:** The view is defined through the `NoteView` interface and implemented by `NoteSwingView`. This allows the controller to be tested with a mocked view, without creating Swing components. If the UI changes, only a new `NoteView` implementation is required.
- **Service layer:** The controller does not access repositories directly. Business operations such as creating or deleting notes are handled by `NoteService`, keeping responsibilities clear and enabling independent testing with mocks.
- **Database initialization:** Default categories (PERSONAL, WORK, STUDY) are initialized in a separate `DatabaseInitializer` class. This improves readability and allows the initialization logic to be tested in isolation.
- **Repository design:** Repositories are defined as interfaces, with MongoDB-specific implementations using the MongoDB Java Driver. Unit tests use mocked repositories, while integration tests run against a real MongoDB instance.
- **Docker for testing:** MongoDB is started for integration tests using the `fabric8 docker-maven-plugin`. The container is started before integration tests and stopped afterward as part of the Maven lifecycle, following the approach described in the course book.

4 Test-Driven Development in Practice

4.1 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.

4.2 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.

4.3 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.

4.4 Docker Configuration for Testing

Docker is used to provide a real MongoDB 6.0 instance for integration and end-to-end tests. The configuration ensures that the database container starts automatically during Maven builds and is available when running tests from Eclipse.

Automatic startup with Maven

The `fabric8 docker-maven-plugin` is configured in `pom.xml` with two executions bound to Maven lifecycle phases:

- **docker-start** bound to `pre-integration-test`: pulls and starts a `mongo:6.0` container on port 27017, waiting up to 30 seconds for the log message "Waiting for connections" before proceeding.

- **docker-stop** bound to **post-integration-test**: stops and removes the container after tests complete.

When running `mvn clean verify`, Maven automatically starts MongoDB before integration tests (executed by the Failsafe plugin) and stops it afterward. Unit tests (executed by the Surefire plugin) do not require Docker since they rely on mocked dependencies.

Running tests from Eclipse

When running integration or end-to-end tests from Eclipse using *Run As* → *JUnit Test*, MongoDB must be started manually beforehand:

```
docker run -p 27017:27017 mongo:6.0
```

This follows the approach described in the course book, where Docker containers for Eclipse-based testing are started manually or via Docker Compose.

CI environment

In GitHub Actions, the Docker Maven Plugin manages the container lifecycle automatically during `mvn clean verify`. GUI tests are executed using `xvfb-run` to provide a virtual display for Swing-based components.

4.5 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.

5 Exclusions and Justifications

5.1 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.

5.2 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.

6 Problem Encountered

6.1 Known Limitation: Java 17 JPMS Compatibility Warning

During GUI testing with AssertJ Swing 3.17.1, the Java 17 module system generates `InaccessibleObjectException` warnings. These warnings occur because AssertJ Swing performs reflective access to `java.util.TimerTask.state` for test synchronization, which is blocked by the Java Platform Module System (JPMS).

Solutions were implemented but the warning still exist:

- **JVM Module Access Configuration**
Added `-add-opens java.base/java.util=ALL-UNNAMED` to both the Maven Surefire and Failsafe plugins via the `argLine` parameter.
- **Cache Invalidation**
Cleared Maven's local plugin cache to force a re-download of plugins with the updated configuration applied.

6.2 Docker port conflicts

during development. During local development, running `mvn clean verify` occasionally failed because port 27017 was already occupied by a previously running MongoDB container that was not properly stopped (e.g., after an interrupted build). The solution was to ensure no MongoDB instance is running before starting the build (`docker ps` and `docker stop` if needed), and to rely on the fabric8 plugin's automatic cleanup during `post-integration-test`.

6.3 JUnit 4 and JUnit 5 coexistence

. AssertJ Swing requires JUnit 4, while the rest of the test suite uses JUnit 5. To run both in a single build, the JUnit Vintage Engine (`junit-vintage-engine`) was added as a dependency, allowing Maven's Surefire and Failsafe plugins to execute both JUnit 4 and JUnit 5 tests transparently.

6.4 Coveralls with Java 17

The coveralls-maven-plugin (version 4.3.0) failed with Java 17 because it depends on javax.xml.bind (JAXB), which was removed from the JDK after Java 8. The solution was to add an explicit jaxb-api dependency (version 2.3.1) to the plugin configuration in pom.xml.

7 Quality Metrics and Results

7.1 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.

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% |

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.

7.2 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.

7.3 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.

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 |

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.

7.4 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.

8 Branching Strategy and CI Workflow

The project follows a branching workflow based on GitHub Pull Requests to manage feature development and ensure code quality.

The main development branches used were:

- **develop** — the integration branch where features are merged before reaching **main**
- **feature** — used for implementing new functionality (model, repository, service, controller, view)
- **refactor** — dedicated to code improvements and restructuring
- **e2e** — used for developing end-to-end tests

Each feature or task was developed in its own dedicated branch and merged into the **develop** branch through a GitHub Pull Request. Once the **develop** branch was considered stable and all CI checks had passed, it was merged into the **main** branch. After merging, feature branches were deleted to keep the repository clean and maintain an organized workflow.

The GitHub Actions CI pipeline is configured to run on pushes to the **main** and **develop** branches, as well as on pull requests targeting **develop**. This setup ensures that every proposed change is automatically validated before being merged into the main codebase.

9 Running the Project

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

Clone the repository:

```
git clone https://github.com/Irfanpv99/Note-Manager.git
cd Note-Manager
```

9.1 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
mvn docker:start
java -jar target/note-manager-1.0.0-jar-with-dependencies.jar
```

Alternatively, the application can be started using `mvn exec:java`

9.2 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
```

10 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/Irfanpv99/Note-Manager>

CI/CD: <https://github.com/Irfanpv99/Note-Manager/actions>

Coverage: <https://coveralls.io/github/Irfanpv99/Note-Manager>

Quality: https://sonarcloud.io/summary/new_code?id=Irfanpv99_Note-Manager