TodoTag-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/TodoTag-Manager
December 5, 2025

Contents

1 Project Overview and Motivation

This report documents the development of TodoTag-Manager, a desktop todo management application built by Test-Driven Development (TDD) principles. The application provides a simple complete todo management system where users can create, update, and delete tasks, organize them with tags, and search through their task description. The Project is the dual database support - the same application can work with either MongoDB or MySQL, decided at runtime through configuration.

1.1 What This Application Does

The application manages daily tasks (todos) with an organized system based on tags. Users can create tasks, mark them complete, attach multiple tags for todo, and search through their description list. The entire interface runs as a desktop application built with Java Swing.

The main technical aspect is the dual database architecture. The same application can store data in either MongoDB (a document database) or MySQL (a relational database), chosen through a simple configuration change.

1.2 TDD Methodology

TDD Methodology Applied 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 spanning three distinct layers:

Unit tests verify individual components in isolation using mock dependencies. These tests focus on business logic within the service layer, GUI controller behavior, repository operations with both MongoDB and MySQL implementations, and configuration management. Mockito provides the mocking framework to ensure tests remain fast and independent of external systems.

Integration tests validate interactions between application layers using real database instances provided by Testcontainers. These tests ensure that MongoDB document mappings, JPA entity relationships, repository queries, and transaction boundaries function correctly across both database implementations. The dual-database architecture requires comprehensive integration testing to guarantee feature parity between MongoDB and MySQL backends.

End-to-end tests exercise the complete application stack, including database persistence, service layer, and the Java Swing desktop interface. AssertJ Swing tests verify user workflows through simulated GUI interactions, validating complete scenarios such as creating todos, managing tags, performing searches, and handling todo-tag relationships. These tests execute in headless mode via Xvfb in the CI environment, ensuring consistent test execution across development and automated build pipelines.

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 - MainFrameController sits between GUI and business logic. When users click "Add Todo", the controller checks the input isn't empty, trims whitespace, calls the service, and returns the result. It never touches databases directly.

Service Layer - TodoService does the real work. Creates todos, marks them complete, manages tags, searches. Had to solve a problem: MySQL needs transactions (start, do work, commit or rollback), MongoDB doesn't. Solution: wrap operations in a helper method that handles transactions for MySQL but just runs MongoDB operations normally.

Repository Layer - Talks to databases. TodoRepository and TagRepository are interfaces defining operations like save, find, delete. Actual implementations (MongoDB or MySQL) get created by RepositoryFactory based on what's in `application.properties`. This makes databases swappable without changing business code.

2.2 Domain Model

Todo has a description (required), done status (starts false), and tags. Description can't be null - the constructor checks. Methods mark todos complete/incomplete and manage tags. **Tag** just has a name (also required). Todos and tags connect both ways - todos know their tags, tags know their todos. Many-to-many relationship.

The hard part: When adding a tag to a todo, both sides need updating. If only one updates, data breaks. The `addTag()` method updates both at once:

```
public void addTag(Tag tag) {
    tags.add(tag);                     // Update todo
    tag.getTodosInternal().add(this); // Update tag
}
```

Defensive copying protects collections. `getTags()` returns a copy, not the original. External code can't mess with internal state by accident.

Equality uses database IDs when they exist. Two todos with the same ID are the same todo. But `hashCode()` uses description (or tag name) because objects need to work in HashSets before they have IDs.

2.3 Database Abstraction

One codebase, two databases.

MongoDB - Uses MongoDB Java Driver. Todos are documents with fields like `_id`, `description`, `done`, `tagIds`. MongoDB doesn't auto-increment IDs like MySQL, so repositories maintain a counter. Find highest ID on startup, set counter to one higher. Tags are separate documents. Todo-tag relationships stored as arrays of tag IDs in each todo.

MySQL - Uses JPA and Hibernate. Annotations like `@Entity`, `@Table`, `@ManyToMany` tell Hibernate how to map objects to tables. The many-to-many relationship creates a join table `todo_tags` automatically. `@GeneratedValue` handles auto-increment IDs. `fetch = FetchType.EAGER` loads tags immediately with todos.

Key idea: Repository interfaces only define `findAll()`, `save()`, `findById()`. Don't care how they work. MongoDB uses documents and queries. MySQL uses SQL through

JPA. Service layer just calls `repository.findAll()` and gets todos - doesn't know which database.

Transactions work differently. MongoDB operations are atomic by default - either everything saves or nothing does. MySQL needs explicit steps: start transaction, do work, commit if good, rollback if error. The `executeWithTransaction()` method handles MySQL transactions automatically but just runs MongoDB operations directly.

2.4 Configuration and Dependencies

AppConfig reads `application.properties`. Main setting: `database.type` (MONGODB or MYSQL). Also stores connection details - host, port, database name for MongoDB; JDBC URL, username, password for MySQL. Has defaults if file is missing.

RepositoryFactory looks at config and creates the right repositories. MongoDB, Creates `MongoTodoRepository` and `MongoTagRepository`. MySQL, Creates `MySqlTodoRepository` and `MySqlTagRepository` with an EntityManager. Also provides transaction methods (`beginTransaction()`, `commitTransaction()`, `rollbackTransaction()`). For MySQL these do real work, for MongoDB they do nothing.

DatabaseManager handles MySQL's JPA setup. Creates entity manager factory, manages connections, coordinates transactions. MongoDB doesn't need this - its driver handles connections itself.

Dependencies flow one way: GUI → Controller → Service → Repository. Makes testing easy. Test GUI with fake controller, controller with fake service, service with fake repositories. For integration tests, use real repositories connected to Docker containers via Testcontainers.

2.5 Technology Stack

Table 1 lists the complete technology stack.

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 Tests test one thing at a time using fake objects (mocks). Testing `TodoService`, Use fake repositories that pretend to be databases. If the test fails, the bug is in the service, not the database.

Example: Testing "add tag to todo". Mock repositories return a fake todo and tag. Call the service method. Check it called `save()` with the right data. Done in milliseconds, no real database needed. Unit tests run instantly - fast enough to run every time you save a file.

Purpose	Technology
Language / Build	Java 17 + Maven
Databases	MongoDB 6.0 | MySQL 8.0
ORM (MySQL)	Hibernate 6.2.2 + Jakarta Persistence 3.1.0
MongoDB Driver	MongoDB Java Driver 4.9.1
GUI Toolkit	Java Swing
Unit Testing	JUnit 5.9.3 + Mockito 4.11.0
GUI Testing	AssertJ Swing 3.17.1
Integration Tests	Testcontainers 1.18.0
Code Coverage	JaCoCo 0.8.8
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

Integration Tests check that parts work together. These use real databases in Docker containers via Testcontainers.

Testing MongoDB repository, Start a MongoDB container, save a todo, read it back, check the data matches. This catches bugs that unit tests miss, like wrong database queries or mapping problems. Integration tests are slower but catch different bugs - bad JPA annotations, wrong queries, transaction problems.

End-to-End Tests test the whole app like a user would. AssertJ Swing clicks buttons, types text, and checks what appears on screen. A real database runs in the background.

Example: Type "Buy milk" in the text box, click "Add Todo", check it shows up in the table, double-click to mark it done, verify status changes, click delete, verify it's gone.

E2E tests are slowest but catch problems between all layers - GUI bugs, timing issues, anything that breaks the full workflow.

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.

Tests look like this: Find the window, find components by name (`button("addTodoButton")`), click them, wait for changes, check the result.

Waiting is critical. After clicking "Add Todo", the app saves to the database then updates the screen. Can't check immediately - nothing's happened yet. Instead: "wait until table has 1 row" with a 10-second timeout.

Every GUI component has a name in code. Tests use these names to find components. If I change the layout, tests still work because names don't change.

3.4 Real Database Testing with Testcontainers

Instead of fake databases, tests use real MongoDB and MySQL in Docker containers.

Setup: Add `@Testcontainers` to the test class, declare `@Container` fields, done. Testcontainers starts containers before tests and stops them after.

For MongoDB: Create `MongoDBContainer`, get the connection URL, point repository at it, run tests. Uses real MongoDB.

For MySQL: Create `MySQLContainer`, get JDBC URL, configure JPA, run tests. Uses real MySQL.

Fake databases lie. H2 (in-memory database) might work fine, then real MySQL fails because they're different. Testcontainers proves code works with actual databases.

Trade-off: Containers are slow to start. Fixed by making containers static - one container per test class, not per test. Clean data between tests instead of restarting.

3.5 Mutation Testing

PITest checks this by breaking your code on purpose. It changes `if (x > 5)` to `if (x >= 5)` and reruns tests. If tests still pass, your tests are weak. If tests fail, good - they caught the change.

Example weak test:

```
todoService.addTag(todoId, tagId);
// No check, PITest can break addTag and test still passes
```

Fixed:

```
todoService.addTag(todoId, tagId);
Todo updated = todoService.getTodoById(todoId).get();
assertEquals(1, updated.getTags().size()); // Now it checks
```

Final result: PITest generated 257 mutations, all 257 killed. Zero survived. Combined with 100% code coverage, this gives that tests actually work.

4 Implementation Challenges and Solutions

4.1 GUI Testing in CI

Tests worked on my laptop but crashed on GitHub Actions with "No display" errors. AssertJ Swing needs a screen to show GUI components, but CI servers don't have screens.

Fixed it with Xvfb - a fake display for servers. The CI workflow installs Xvfb, starts it, and tests think there's a real screen. For PITest, I added `-Djava.awt.headless=true` to make it work in headless mode.

4.2 Auto-Increment IDs in MongoDB

MySQL gives you 1, 2, 3... automatically. MongoDB gives random IDs like "507f1f77bcf86cd799439011". I needed both databases to use simple numbers.

Solution: MongoDB repositories keep a counter. When they start, they find the highest ID in the database and add 1. When saving new items, they use that counter and increment it. Works fine for a single-user desktop app.

4.3 Slow Integration Tests

Tests took 5+ minutes. Because every test started fresh database containers, ran for 2 seconds, then threw them away.

Changed containers from regular to static fields. Now one container runs for all tests in a class instead of one per test. Tests clean data between runs instead of restarting containers. Time dropped to under 2 minutes.

4.4 Todo-Tag Relationships

Todos have tags. Tags know their todos. Keeping both sides synced is tricky.

Can't just do `todo.getTags().add(tag)` - only updates one side. Made helper methods that update both:

```
public void addTag(Tag tag) {
    tags.add(tag);                      // Update todo side
    tag.getTodosInternal().add(this); // Update tag side
}
```

4.5 Handling Transactions

MySQL needs explicit transactions. MongoDB doesn't. Didn't want to write transaction code twice.

Made the service layer call `beginTransaction()` and `commitTransaction()` always. For MySQL, these actually do something. For MongoDB, they're empty methods. Service layer doesn't care which database is running.

4.6 Getting 100% Mutation Coverage

PITest kept finding weak tests. It would remove a `save()` call and tests still passed. Made tests check everything:

```
// Weak - doesn't verify anything
todoService.createTodo("Task");

// Strong - checks it actually saved
Todo created = todoService.createTodo("Task");
assertNotNull(created.getId());
Optional<Todo> found = todoService.getTodoById(created.getId());
assertTrue(found.isPresent());
```

Each assertion kills a specific mutation.

5 Quality Metrics and Results

5.1 Code Coverage with JaCoCo

The project uses JaCoCo to measure test coverage across all test types. Two classes are excluded from coverage requirements:

- `TodoApplication` - The main method wires dependencies together. E2E tests already verify the application launches correctly.
- `MainFrame` - Contains mostly Swing GUI layout code (button positioning, panel sizing, colors). E2E tests verify this through actual GUI interaction.

Table 2 shows the final coverage results. Every other class achieves 100% coverage across all metrics.

Table 2: JaCoCo Code Coverage Summary

Package	**Instr.**	**Branch**	**Lines**	**Meth.**
config	100% (309/309)	100% (26/26)	100% (85/85)	100% (21/21)
gui	100% (187/187)	100% (40/40)	100% (53/53)	100% (12/12)
model	100% (296/296)	100% (44/44)	100% (71/71)	100% (28/28)
repository	100% (190/190)	100% (24/24)	100% (49/49)	100% (10/10)
repository.mongo	100% (562/562)	100% (38/38)	100% (119/119)	100% (26/26)
repository.mysql	100% (224/224)	100% (16/16)	100% (57/57)	100% (16/16)
service	100% (369/369)	100% (26/26)	100% (86/86)	100% (30/30)
Total 100%	**(2 137/2 137)**	**(214/214)**	**(520/520)**	**(143/143)**

Coverage reports generate automatically during the Maven test phase and upload to Coveralls, which tracks coverage history over time. The repository README displays a Coveralls badge showing the current percentage.

5.2 Mutation Testing with PITest

PITest generated 257 mutations across 11 classes. All 257 were killed, achieving 100% mutation coverage with zero survivors. Table 3 shows the mutation testing breakdown by package.

Table 3: PITest Mutation Coverage by Package

Package	**Line**	**Mut.**	**Strength**
config	100% (83/83)	100% (33/33)	100% (33/33)
gui	100% (53/53)	100% (44/44)	100% (44/44)
model	100% (71/71)	100% (39/39)	100% (39/39)
repository	100% (49/49)	100% (21/21)	100% (21/21)
repository.mongo	100% (119/119)	100% (45/45)	100% (45/45)
repository.mysql	100% (57/57)	100% (28/28)	100% (28/28)
service	100% (86/86)	100% (47/47)	100% (47/47)
Total 100%	**(518/518)**	**(257/257)**	**(257/257)**

A 100% mutation score means every single mutation caused at least one test to fail. This proves the tests actually verify behavior instead of just executing code. For example, when PITest changed `todo.setDone(true)` to `todo.setDone(false)`, tests caught it because they explicitly verify the status. When PITest removed `repository.save()` calls, tests failed because they check data persistence.

Mutation testing runs as a separate GitHub Actions job only on pushes to main branch since it's time-consuming. Reports upload as build artifacts for review.

5.3 Static Analysis with SonarCloud

SonarCloud performs automated code quality analysis on every pull request, checking for code smells, bugs, security vulnerabilities, and calculating technical debt.As shown in Table 4, all classes except the one explicitly discussed attain full (100 %) coverage on every metric.

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

Table 4: SonarQube Quality Gate Summary

Zero technical debt results from following clean code practices: meaningful variable names, short methods with single responsibilities, proper error handling, and consistent formatting. SonarCloud's quality gate must pass before merging pull requests, enforcing these standards throughout development.

5.4 Continuous Integration Pipeline

GitHub Actions runs three jobs on every push:

Test & Coverage - Starts MongoDB and MySQL services, runs all tests with Xvfb for GUI support, generates JaCoCo coverage reports, uploads to Coveralls, and runs SonarCloud analysis. This job fails if any test fails or if coverage drops below 100%.

Mutation Testing - Runs only on main branch pushes. Executes PITest with headless Swing support and uploads mutation reports as artifacts. This job fails if mutation score drops below 100%.

Build - Packages the application into a JAR file and uploads it as an artifact. This verifies the application builds successfully.

All three jobs must pass for the build to succeed. Failed builds prevent merging pull requests, automatically enforcing quality standards.

6 Running the Project

6.1 Development Environment Setup

The project requires Java 17 or higher, Maven 3.8 or higher, and Docker for running databases and enabling Testcontainers in tests.

Clone the repository:

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

Maven automatically downloads all dependencies on first build.

6.2 Database Configuration

The application can use MongoDB or MySQL. Select the database by editing src/-main/resources/application.properties. Change `database.type` to either MONGODB or MYSQL, then configure the connection details for your chosen database. For local development, start databases using Docker Compose:

```
docker-compose up -d. This starts MongoDB on port 27017 and MySQL
   on port 3307.
```

6.3 Running Tests

Execute all tests (unit + integration + E2E):

```
mvn clean verify
```

Execute only unit tests (faster, for development):

```
mvn clean test
```

Docker must be running - Testcontainers automatically starts database containers as needed.

Generate coverage report:

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

Run mutation tests:

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

Eclipse: Right-click `src/test/java` and select `Run As` → `JUnit Test`

6.4 Eclipse Setup

Import the project into Eclipse:

1. Open Eclipse IDE
2. Select `File` → `Import` → `Maven` → `Existing Maven Projects`
3. Browse to the cloned `TodoTag-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`
- Select `Run As` → `JUnit Test`

- All tests should pass

Run Application from Eclipse:

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

Running from Terminal

```
Start databases : docker-compose up -d
Run application : mvn exec:java
```

The GUI window opens and connects to the configured database. Create todos, add tags, and manage your tasks through the interface.

7 Project Summary

TodoTag-Manager shows how to build software application with proper testing. The project hits all quality targets: 100% code coverage (except GUI layout code), 100% mutation coverage (257 mutations killed, zero survivors), 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 dual database setup works well. Same code runs MongoDB or MySQL - just change a config file. This proves the repository pattern and abstraction actually work in practice.

Test-Driven Development drove everything. Every feature started with a test that failed, then code to make it pass, then cleanup. Red-Green-Refactor kept development disciplined. The test suite now gives confidence to change code without breaking things.

GUI testing with AssertJ Swing was worth the effort. These tests caught bugs between layers that unit tests missed - like timing issues and state management problems. Xvfb makes GUI tests run in CI without a real display.

The reports confirm this: 100% coverage, 257/257 mutations killed, zero code smells. The tests actually verify behavior, not just run code. This project shows that good TDD practices create software applications.

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

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

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

Quality: https://sonarcloud.io/summary/new_code?id=Irfancpv99_TodoTag-Manager