



Automated Software Testing

Final Project

Student: Bakhauddin Ziyouddinov

ID number: 7128283

Professor: Lorenzo Bettini

Florence 2025

Content

Automated Software Testing.....	1
Introduction.....	3
Main Features.....	3
Overview.....	4
Technology Stack.....	5
Testing Approach.....	5
Testing Quality Assurance Measures.....	6
Parameterized Unit Tests.....	6
Difficulties.....	7
Continuous Integration.....	8
Future Work.....	9
Conclusion.....	10
References.....	10

Introduction

In this project, I built a desktop application to help libraries manage their book collections and author information. This application can be considered as a digital catalog system where library workers can add, view, edit, and track books and authors in their library. The application has a user-friendly interface and stores all data in a database.

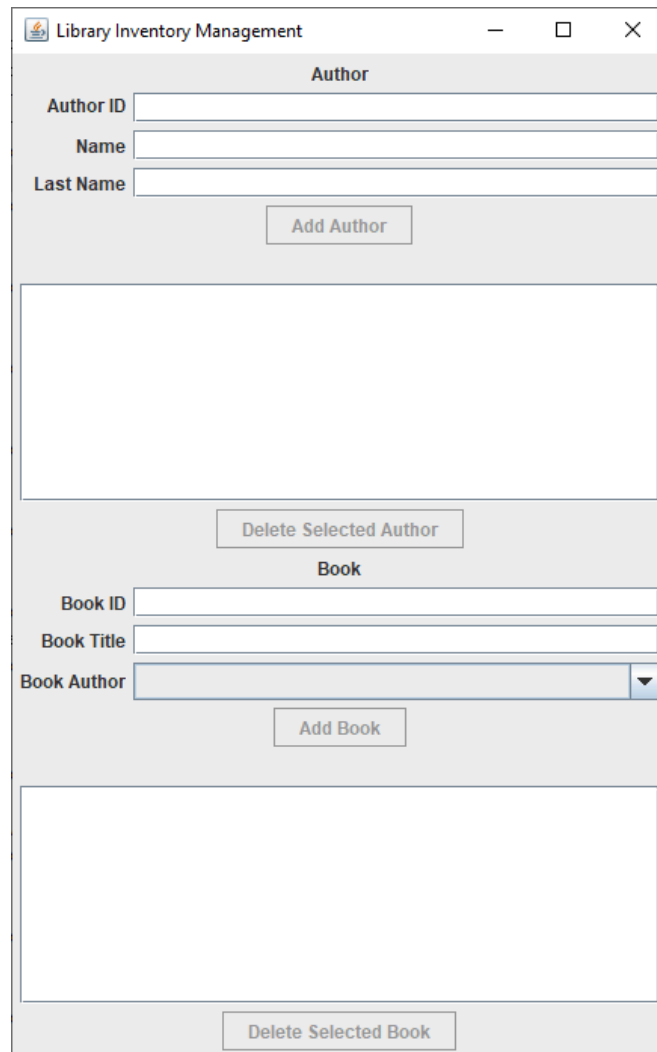
Libraries need an efficient way to organize their collections. This system solves that problem by providing:

- Easy book and author management
- Reliable data storage
- Simple and intuitive interface

Main Features

- Book Management
- Author Management
- Data Storage
- Desktop Interface
- Flexible startup options for different configurations

Overview



The screenshot shows a web application window titled "Library Inventory Management". The interface is divided into two main sections: "Author" and "Book".

Author Section:

- Fields: Author ID, Name, Last Name.
- Buttons: Add Author, Delete Selected Author.
- A large empty rectangular box is present below the fields.

Book Section:

- Fields: Book ID, Book Title, Book Author (a dropdown menu).
- Buttons: Add Book, Delete Selected Book.
- A large empty rectangular box is present below the fields.

Figure 1 - Main Interface of Application

The application is based on two main entities: Author(id, firstName, lastName) and Book(id, title, author). The Book entity requires the creation of the Author entity first. The realization of this feature through the user interface is achieved by creating a comboBox, which works as a duplicate of the author list, but where the user can select an author of the book.

The user cannot create two Author or Book entities with the same ID. If the user wants to make an author or book with the ID already in the database, the error message will be shown, and the data will not be passed to the MongoDB database.

The whole application was done by respecting the rules of Test Driven Development(TDD) and Model-View-Controller(MVC) techniques.

Technology Stack

The underlying build automation tool used is Apache Maven[1], which is also responsible for dependency management. The primary development language used was Java 8. MongoDB 5[2] was used to store data. For containerization, the Docker[3] application was used.

Testing framework includes these technologies:

- JUnit 4.12
- Mockito 4.11.0
- AssertJ 3.27.8
- Cucumber 5.5.0
- Testcontainers 1.21.0

Code Analysis:

- SonarCloud
- JaCoCo
- PIT
- Coveralls

Testing Approach

The Model-View-Controller(MVC) is my main approach while developing this application. The Book and Author models were used as a base. Each has a View that is used to present the data to the user and handles user interactions, and a controller that handles user input and updates the Model.

To develop the User Interface(UI), the Java Swing framework was used. It bridges the user and the application's logic, providing the user with appropriate input fields.

JUnit 4 was used as the main testing framework. The unit tests were performed on the User Interface(UI) logic and controllers (Both Book and Author). The Integration Tests(IT) were performed to validate the interactions and data exchange between individual application modules. The ITs were performed to validate the interactions between:

- AuthorController and a MongoDB
- BookController and a MongoDB
- User Interface with an Author and Book Controller and an Author and Book Repositories.
- Additionally, the Integration Tests that verify the user's GUI actions were performed on the database were written.

- Also, the ITs using TestContainers were performed as an additional test for the Integration Testing phase (For Book and Author separately).

The end-to-end (e2e) tests were written in two ways:

- The “traditional” way validates that the entire application works correctly from start to end. It starts a real application using a real database and simulated fundamental user interactions.
- The Behavior-Driven Development (BDD). In this way, the tests are written in plain English, which makes them more human-readable than the actual code. As a “traditional” approach, it also simulates the real interaction between the user and the application.

I decided to include BDD tests as an additional E2E test.

Testing Quality Assurance Measures

To make sure my tests are helpful and catch real problems, I implemented several quality measures:

- Jacoco is used to measure how much of the code is being tested.
- Pit: Creating mutators allows us to change the code dynamically to detect tests that will fail. In my code, I used the “Stronger” mutants. The stronger mutants can dynamically change more code than the default ones. For example, they can remove conditions when default mutants cannot.
- SonarCloud is used in the continuous integration phase as a primary tool to check code quality.

Parameterized Unit Tests

As mentioned, my application has two models (Author and Book). Because creating a new Author includes filling several input fields (from the end user perspective), I decided to create a class where I used parameterized testing to validate the user's input. The user should not be able to make an author model with empty or null fields. The author model contains three input fields, and I need to check each for null or empty conditions. We have six possible conditions, which can be grouped as shown in Figure 2.

```

@Parameterized.Parameters
public static Collection<Object[]> data() {
    return Arrays.asList(new Object[][] {
        { new Author(null, "Name", "Surname"), "Please, set a correct author id!" },
        { new Author(" ", "Name", "Surname"), "Please, set a correct author id!" },
        { new Author("1", null, "Surname"), "Please, set a correct name for author with ID: 1" },
        { new Author("1", " ", "Surname"), "Please, set a correct name for author with ID: 1" },
        { new Author("1", "Name", null), "Please, set a correct surname for author with ID: 1" },
        { new Author("1", "Name", " "), "Please, set a correct surname for author with ID: 1" }
    });
}

```

Figure 2

The way of writing parameterized tests differs from the traditional JUnit 4 tests. Parameterized tests have limitations, but when the input and output are clean, we can use them to test the application.

Difficulties

While developing the GUI, I decided to use the multithreading technique. It is known that all operations concerning GUI controls must take place in the Event Dispatcher Thread(EDT) in Swing. For this reason, it is best practice to handle user events, which are known to possibly be long-running operations that might be blocking the UI, in a parallel thread.

The local machine could be overloaded with other processes, and the thread executing the code of our GUI controls is not scheduled on time to fulfill the verification.

```

addAuthorButton.addActionListener(e -> {
    lastAddAuthorThread = new Thread(() -> {
        try {
            Thread.sleep(sleepMs);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
        authorController.newAuthor(
            new Author(
                authorIdTextBox.getText(),
                authorNameTextBox.getText(),
                authorSurnameTextBox.getText()
            )
        );
    });
    lastAddAuthorThread.start();
});

```

Figure 3 - Multithreading implementation for AddAuthorButton

To test this feature, I performed verification using Mockito's timeout mechanism. The idea is to perform verification until it succeeds (it will fail only after a given timeout)

```
@Test
public void testAddAuthorButtonShouldDelegateToAuthorControllerNewAuthor() {
    » window.textBox("authorIdTextBox").enterText("1");
    » window.textBox("authorNameTextBox").enterText("Leo");
    » window.textBox("authorSurnameTextBox").enterText("Tolstoy");
    » window.button(JButtonMatcher.withText("Add Author")).click();
    » verify(authorController, timeout(TIMEOUT)).newAuthor(new Author("1", "Leo", "Tolstoy"));
}
}
```

Figure 4 - Example of a test based on timeout

The other issue arises from here: we also need to cover the “catch” condition (Figure 3). We must ensure the controller is never called when the thread is interrupted. We must interrupt it right after the new thread is created (Figure 5). When the user clicks on the “Add Author” button (in this case), the thread is interrupted, and the controller is never called. The line “join(TIMEOUT)” needs to be checked for verification after the interrupted thread has finished. The 2000ms delay is set to ensure that the thread is sleeping and simulates a long-running operation.

```
@Test
public void testAddAuthorButtonHandlesInterruptedException() throws Exception {
    » GuiActionRunner.execute(() -> swingView.setAddSleepMs(2000));
    » window.textBox("authorIdTextBox").enterText("1");
    » window.textBox("authorNameTextBox").enterText("Test Interrupt");
    » window.textBox("authorSurnameTextBox").enterText("Test Interrupt");
    » window.button(JButtonMatcher.withText("Add Author")).click();
    »
    » assertThat(swingView.lastAddAuthorThread).isNotNull();
    » swingView.lastAddAuthorThread.interrupt();
    » swingView.lastAddAuthorThread.join(TIMEOUT);
    » verify(authorController, never()).newAuthor(new Author("1", "Test Interrupt", "Test Interrupt"));
}
}
```

Figure 5 - Test interruption of a thread

Continuous Integration

Instead of manually checking my code quality every time I made changes, I set up an automated system that does this work for me. Every time I upload new code to GitHub, the system automatically:

- Runs all my tests to make sure nothing fails
- Checks the code quality and looks for potential problems
- Make sure that the application works on different Java versions
- Generates reports showing how well my code is tested

I wanted to make sure my application works regardless of the Java application (Java 8, 11, 17)

- Java 8: Measures how much of my code is tested, sends coverage reports to Coveralls
- Java 11: Runs mutation testing, which checks if my tests cover all the “mutant” cases. Creates a detailed report showing test effectiveness
- Java 17: Uses SonarCloud to scan for potential bugs and problems, which also checks for coding standards

All the generated artifacts(Junit report, Pit report, Jacoco report) are stored for each action.

To make the process of continuous integration faster and more efficient, the system tests all three versions at the same time and saves the previously downloaded dependencies.

Future Work

There are a lot of options and ways to improve that application. The first one that comes to my mind is improving the models' relationship. Each book can have one Author, and one Author model can be assigned to many Books. The other way is also possible: when several authors write one Book. Another improvement that can be made is to increase the model parameters. For example, add a publication date to a Book Model or additional information about the Author in the author model. Another way to improve is to add security to the application by creating a user authentication component or role-based access control.

These enhancements would transform the desktop application into a modern library inventory management platform. As we all know, there is no limit to perfection.

Conclusion

I wanted to build a simple library inventory management application when I started this project. But it became a great learning experience that taught me how professional software development works.

The application I built successfully solves the initially identified problem: software to perform book management in a store/library.

Instead of just writing a few basic tests, I created a testing system that covers everything:

- Unit tests that check individual pieces of code
- Integration tests that make sure different parts of the system work together properly
- GUI test that automatically clicks buttons and fills out forms like a real user would
- End-to-end tests that test complete workflows from start to finish
- Behavior-driven tests written in plain English that anyone can understand

Ultimately, 100% of my code is tested, and I can make changes knowing that if something breaks, my tests will catch it immediately. Most importantly, I learned how to think like a real developer.

References

- [1] Apache Maven - <https://maven.apache.org/>
- [2] MongoDB - <https://www.mongodb.com/>
- [3] Docker - <https://www.docker.com/>
- [4] Java - <https://docs.oracle.com/javase/8/docs/api/>
- [5] Coveralls - <https://coveralls.io/>
- [6] SonarCloud - <https://sonarcloud.io>
- [7] Cucumber - <https://cucumber.io/>
- [8] Pitest - <https://pitest.org/>