DUE DATE: 06.05.2024

Software Engineering - CEN 302

GROUP 3: Kledia Boka, Sonia Sotiri, Sorina Hastoci, Jada Mece, Friona Pocari

OUR GITHUB REPOSITORY LINK: https://github.com/kboka/SE_Project_Phase1_Team3

PHASE IV:

1. INTRODUCTION TO TESTING

Software testing is a critical phase in the software development lifecycle aimed at uncovering defects or bugs that could potentially impact the performance, reliability, and usability of an application. It involves a systematic and methodical examination of the software to ensure that it behaves as expected and meets the requirements outlined during the development process. Testing is not just about finding bugs; it's also about validating that the software functions correctly under various scenarios and conditions, including normal usage, edge cases, and unexpected inputs.

There are various types of software testing, ranging from unit testing, where individual components or modules are tested in isolation, to system testing, where the entire application is evaluated as a whole. Testing methodologies can also vary, including manual testing, where testers manually execute test cases, and automated testing, which involves using tools and scripts to automate the testing process. Each type of testing serves a specific purpose and contributes to ensuring the overall quality of the software.

Effective software testing requires a combination of technical expertise, domain knowledge, and thorough understanding of the project requirements. It is an iterative process that starts early in the development cycle and continues throughout the software's lifecycle. By identifying and fixing defects early on, testing helps reduce the cost and time involved in software development, enhances user satisfaction, and ultimately contributes to the success of the software product in the market.

2. PURPOSE OF TESTING

Testing the code for the Ilir Subashi Management System is crucial for guaranteeing the software's quality, dependability, and functioning. Here are the main reasons why testing is necessary:

Early Detection of Faults: Developers can identify faults and problems in code through testing. By performing tests on multiple portions of the system in a systematic manner, errors may be found and resolved quickly, minimizing the risk of more serious problems arising later in the development lifecycle or after deployment.

Verification of Software Components: Testing ensures that functions, modules, and features meet requirements and standards. This procedure guarantees that the developed code matches the intended behavior described in the project's design and functional requirements.
Ensuring functional correctness:

During testing, engineers ensure that the product works properly under all situations and user interactions. This comprises checking for boundary conditions, error handling, input validations, and intended outputs. It verifies that the software fits the functional criteria specified for inventory management, financial transactions, and administrative operations.
Testing improves software quality by identifying and addressing flaws early on. Developers may enhance and optimize software by testing it on a continual basis, resulting in a more robust and dependable solution.

Thorough testing ensures the Ilir Subashi Management System satisfies user expectations and delivers a smooth experience. By discovering and resolving issues before deployment, the system is less likely to have interruptions or usability difficulties that might affect user satisfaction.

Testing reduces hazards connected with software deployment and usage. Identifying and fixing problems during the development phase reduces the chance of significant failures or security vulnerabilities in the production environment, increasing the system's overall dependability.

Testing promotes continual improvement through its iterative nature. Testing feedback allows developers to improve their code, follow best practices, and optimize performance, resulting in a more efficient and maintainable software solution over time.

3. FOCUS ON TESTING A SINGLE COMPONENT

1. DATABASE CONNECTION

- **Component:** The database connection class/module/function is crucial as it handles the interaction between the software application and the underlying database. It's responsible for establishing, maintaining, and closing connections to the database, as well as executing queries and handling results.
- **Importance of Testing:** Testing this component is vital because any issues with the database connection can lead to data loss, corruption, or security vulnerabilities. Moreover, errors in database interaction can cause the entire application to malfunction, affecting its usability and reliability.
- **Complexity:** The complexity of this component lies in handling various database management systems (e.g., MySQL, PostgreSQL, MongoDB), connection pooling, transaction management, and error handling.
- **Impact on the System:** A faulty database connection can result in data inconsistency, application crashes, or even security breaches. Therefore, thoroughly testing this component ensures the stability, performance, and security of the entire system.

2. LOGIN WINDOW

- Component: The login window class/module/function is responsible for authenticating users and granting access to the system. It validates user credentials, handles session management, and controls user permissions.

- Importance of Testing: Testing this component is essential as it forms the first line of defense against unauthorized access to the system. Any vulnerabilities or flaws in the login process can compromise the system's security and integrity.

- Complexity: The complexity of this component lies in handling various authentication methods (e.g., username/password, OAuth), implementing security features (e.g., CAPTCHA, two-factor authentication), and managing user sessions securely.

- Impact on the System: A weak or flawed login mechanism can lead to unauthorized access, data breaches, and loss of sensitive information. Thorough testing ensures that the login process is robust, reliable, and resistant to security threats.

3. ECONOMIST FRONT PAGE EXIT LABEL

- Component: The economist front page exit label represents a specific feature or functionality on the front page of an application, possibly related to navigation or user interaction.

- Importance of Testing: Testing this component ensures that users can navigate or interact with the front page of the application seamlessly. Any issues with this component could result in a poor user experience or hinder users from accessing important features or content.

- Complexity: The complexity of this component depends on its functionality, such as handling user inputs, triggering actions, or navigating to different sections of the application. It may also involve integration with other parts of the application.

- Impact on the System: A malfunctioning or poorly implemented exit label can frustrate users and lead to dissatisfaction with the application. Thorough testing helps identify and address any usability issues or bugs, ensuring a smooth user experience.

4. INSERTING NEW ADMIN

- Component: The functionality responsible for inserting a new admin user into the system's database.

- Importance of Testing: Testing this component is crucial as it involves handling sensitive user data and granting administrative privileges. Any vulnerabilities or flaws in this process could lead to unauthorized access or privilege escalation.

- Complexity: The complexity of this component lies in validating user inputs, enforcing security measures (e.g., password hashing, input sanitization), and ensuring proper authorization for administrative actions.

- Impact on the System: A security breach or unauthorized access resulting from flaws in the new admin insertion process can have severe consequences, including data breaches, system compromise, and legal liabilities. Thorough testing helps mitigate these risks by identifying and addressing security vulnerabilities.

5. INSERTING NEW PRODUCT

- **Component:** The functionality responsible for adding a new product to the system's inventory or database.
- **Importance of Testing:** Testing this component is important as it directly impacts the accuracy and reliability of the system's inventory management. Any errors or inconsistencies in adding new products could lead to inventory discrepancies or disruptions in supply chain operations.
- **Complexity:** The complexity of this component lies in validating product information, handling inventory updates, and ensuring data integrity across the system.
- **Impact on the System:** Inaccurate or incomplete product data resulting from flaws in the insertion process can lead to mismanagement of inventory, stockouts, or overstock situations. Thorough testing helps ensure that the system's inventory remains accurate and up-to-date, facilitating smooth business operations.

4. PREPARING TEST CASES

1. DATABASE CONNECTION

- **Normal Input:** Test connecting to the database using valid credentials and ensure that the connection is successful.
- **Edge Cases:** Test connecting to the database with maximum allowed connections and verify that it handles connection pooling efficiently. Test connecting to the database with incorrect but recoverable credentials and ensure appropriate error handling.
- **Invalid Input:** Test connecting to the database with incorrect credentials and ensure that it fails gracefully with an informative error message. Test connecting to a non-existent database and verify that it handles the error appropriately.

2. LOGIN WINDOW

- **Normal Input:** Test logging in with valid username and password and ensure that the user gains access to the system.
- **Edge Cases:** Test logging in with a username containing special characters and verify that it handles them correctly. Test logging in with a password exceeding the maximum length and ensure that it truncates or rejects the input appropriately.
- **Invalid Input:** Test logging in with an incorrect password and ensure that it fails with an appropriate error message. Test logging in with a non-existent username and verify that it handles the error gracefully.

3. ECONOMIST FRONT PAGE EXIT LABEL

- **Normal Input:** Test clicking on the exit label and ensure that it navigates the user to the intended page or performs the intended action.
- **Edge Cases:** Test clicking on the exit label rapidly multiple times and verify that it doesn't cause any unexpected behavior or errors. Test clicking on the exit label while other asynchronous processes are ongoing and ensure that it doesn't interfere with them.

- **Invalid Input:** Test clicking on the exit label when the application is in a state where navigation is not allowed and verify that it handles the action appropriately without crashing or freezing.

4. INSERTING NEW ADMIN

- **Normal Input:** Test inserting a new admin with valid username, password, and permissions and ensure that the new admin is added to the system successfully.
- **Edge Cases:** Test inserting a new admin with the maximum allowed username length and ensure that it handles it correctly. Test inserting a new admin with special characters in the password and verify that it handles them securely.
- **Invalid Input:** Test inserting a new admin with a username that already exists in the system and ensure that it fails with an appropriate error message. Test inserting a new admin with a blank password and verify that it rejects the input and prompts for a valid password.

5. INSERTING NEW PRODUCT

- **Normal Input:** Test inserting a new product with valid information (e.g., name, price, quantity) and ensure that it is added to the inventory correctly.
- **Edge Cases:** Test inserting a new product with the maximum allowed length for the product name and ensure that it handles it correctly. Test inserting a new product with a price of zero or a negative value and verify that it rejects the input.
- **Invalid Input:** Test inserting a new product with a blank name or missing required fields and ensure that it fails with an appropriate error message. Test inserting a new product with a quantity exceeding the maximum allowed value and verify that it handles the input appropriately.

5. CHOOSING TESTING FRAMEWORKS

For Java development in NetBeans, JUnit is a highly recommended testing framework for unit testing. It provides a simple and effective way to write and execute tests for your Java classes. Here's how you can set up your testing environment with JUnit in NetBeans:

1. Install JUnit:

- JUnit is typically bundled with most Java development environments, including NetBeans. However, if you need to install it manually, you can download the JUnit JAR file from the official website (https://junit.org/junit5/) and add it to your project's classpath.

2. Create Test Classes:

- In NetBeans, right-click on your project in the Project Explorer.
- Go to New > Other.
- Choose the "JUnit" category and select "JUnit Test".
- Click "Next" and follow the prompts to create a new test class.
- You can also manually create test classes by creating Java classes and annotating them with `@Test` annotations from the JUnit framework.

3. Write Test Methods:
- Inside your test class, write methods to test different aspects of your code.
- Use JUnit's assertion methods like `assertEquals()`, `assertTrue()`, `assertFalse()`, etc., to validate the expected behavior of your code.

4. Run Tests:
- Right-click on your test class or individual test methods.
- Choose "Test File" or "Test Method" from the context menu.
- Alternatively, you can run tests using the keyboard shortcut (usually Shift+F6).

5. View Test Results:
- The results of the test execution will be displayed in the NetBeans output console.
- You'll see a summary of the tests executed, along with any failures or errors encountered.

6. Analyze Results:
- NetBeans provides detailed information about test failures, including stack traces and assertion failures.
- Use this information to diagnose and fix issues in your code.

7. Repeat and Refactor:
- Continuously write new tests as you develop new features or refactor existing ones.
- Run tests frequently to ensure that your code remains functional and regression-free.

6. WRITING TEST CODE

For the `testInsertProduct` method in the `ProduktiTest` class, we're testing the functionality of inserting a product. Here's how we can structure the test code:

1. Test Environment Setup: Begin by importing necessary libraries, including JUnit and `java.lang.reflect.Method`.

2. Instantiate the Class Under Test: Create an instance of the `Produkti` class, which represents the GUI frame for managing products.

3. Set Sample Data: Set up sample data within the text fields of the `Produkti` frame. This data simulates user input for inserting a product.

4. Invoke the Private Method: Use reflection to access the private method `jButton1ActionPerformed`, which represents the action performed when the insert button is clicked. By setting its accessibility to true and invoking it, we simulate the button click event programmatically.

5. Assertions: Add assertions to verify the expected behavior. For example, after insertion, we can assert that the text fields are cleared. Additionally, we might want to check if the table displaying products is updated correctly.

6. Exception Handling: Wrap the test logic in a try-catch block to handle any exceptions that might occur during the test execution.

7. RUNNING TESTS

1. Run Tests:

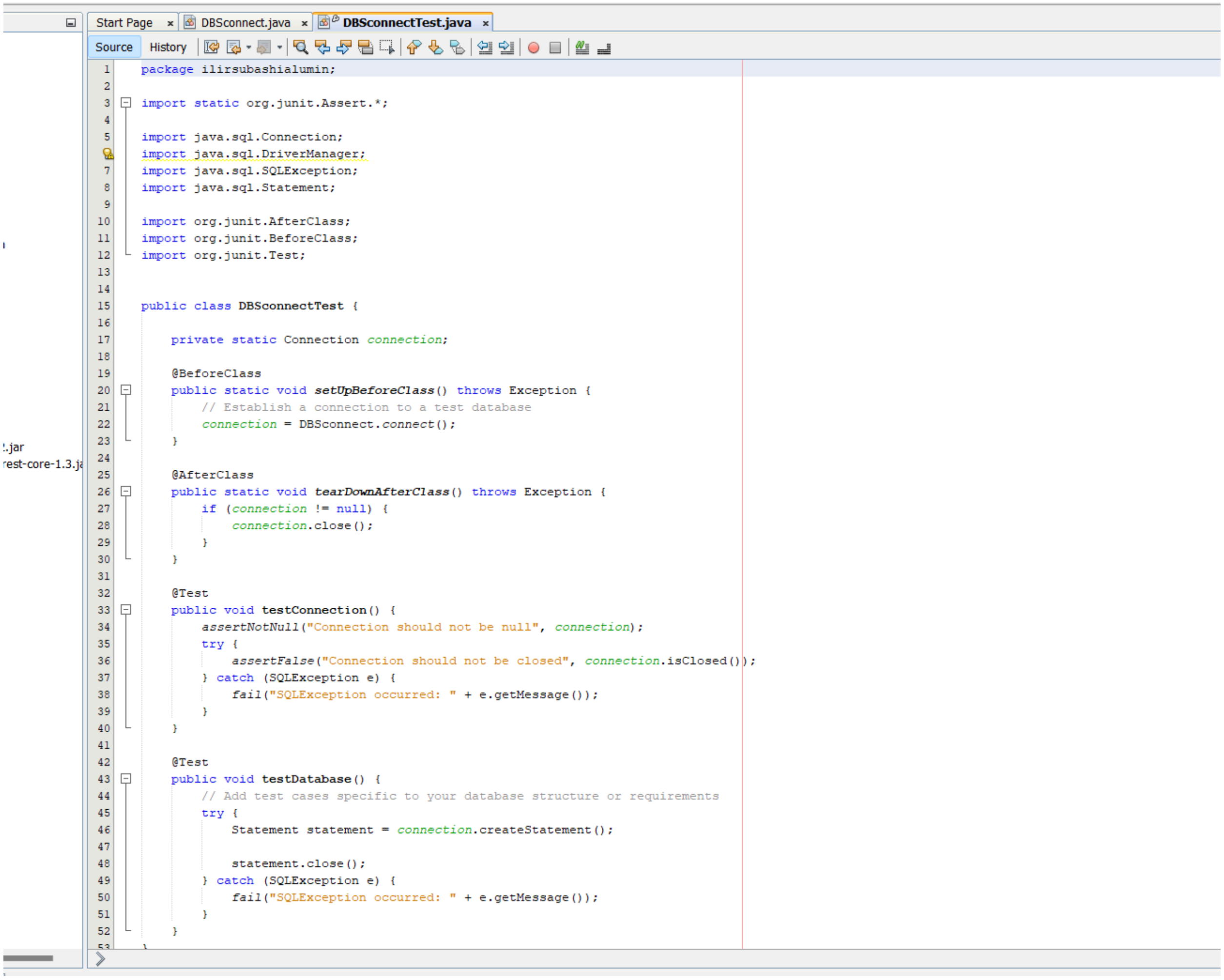
- Open your project in NetBeans.
- Navigate to the test class or test suite you want to execute.
- Right-click on the test class or suite file.
- Choose the "Run File" option from the context menu.
- NetBeans will execute the selected tests and display the results in the Output window.

2. Interpret Results:

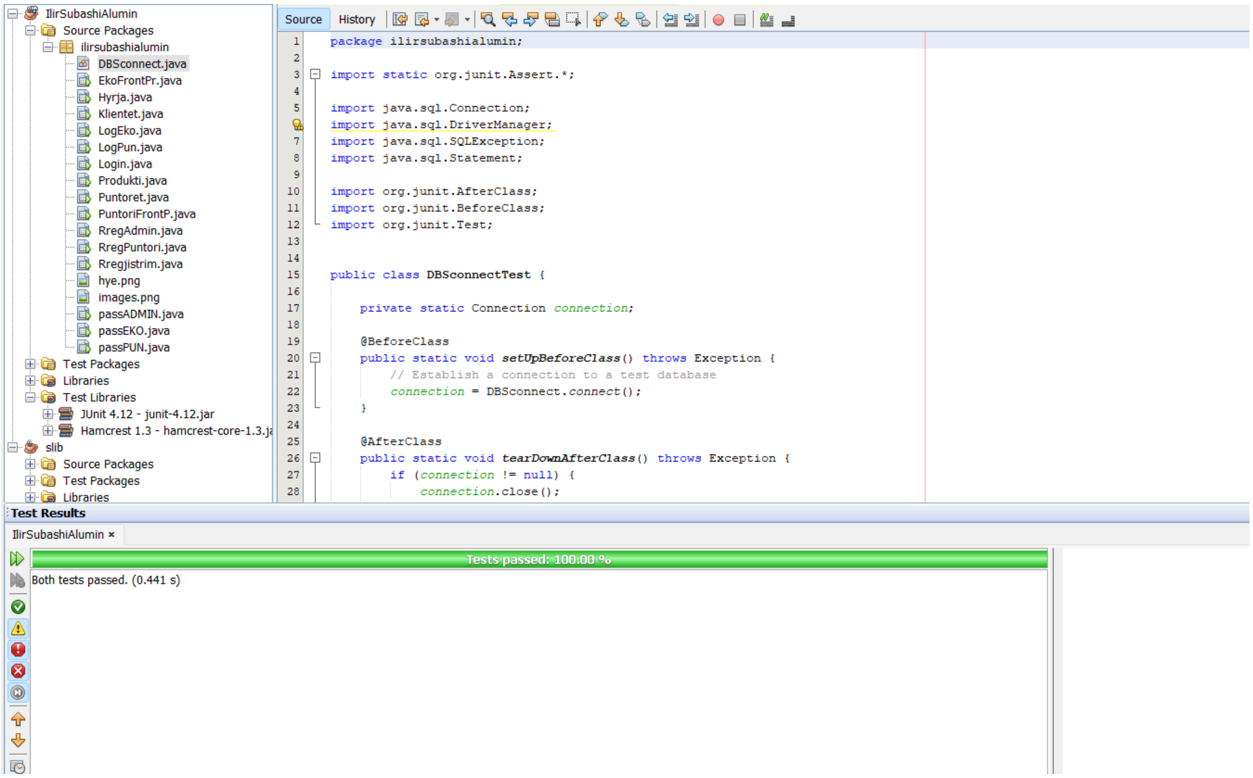
- Passing Tests: Successful tests indicate that the functionality behaves as expected.
- Failing Tests: Tests encountering unexpected behavior fail, highlighting discrepancies between expected and actual behavior.most of the problems happened with assertions
- Error Scenarios: Errors occur due to unexpected exceptions or runtime issues during test execution.This happened because the GUI components have private access to the main code.
- Debugging: Use NetBeans' built-in debugging tools to diagnose test failures and errors.
- Fixing Issues: Modify application or test code as necessary to address identified issues.
- Regression Testing: Re-run affected tests and broader regression test suites to ensure fixes don't introduce regressions.

3. Reporting:

- NetBeans provides detailed reports summarizing test execution results, including passed, failed, and skipped tests, along with any errors encountered.
- Utilize these reports to communicate testing outcomes to stakeholders.

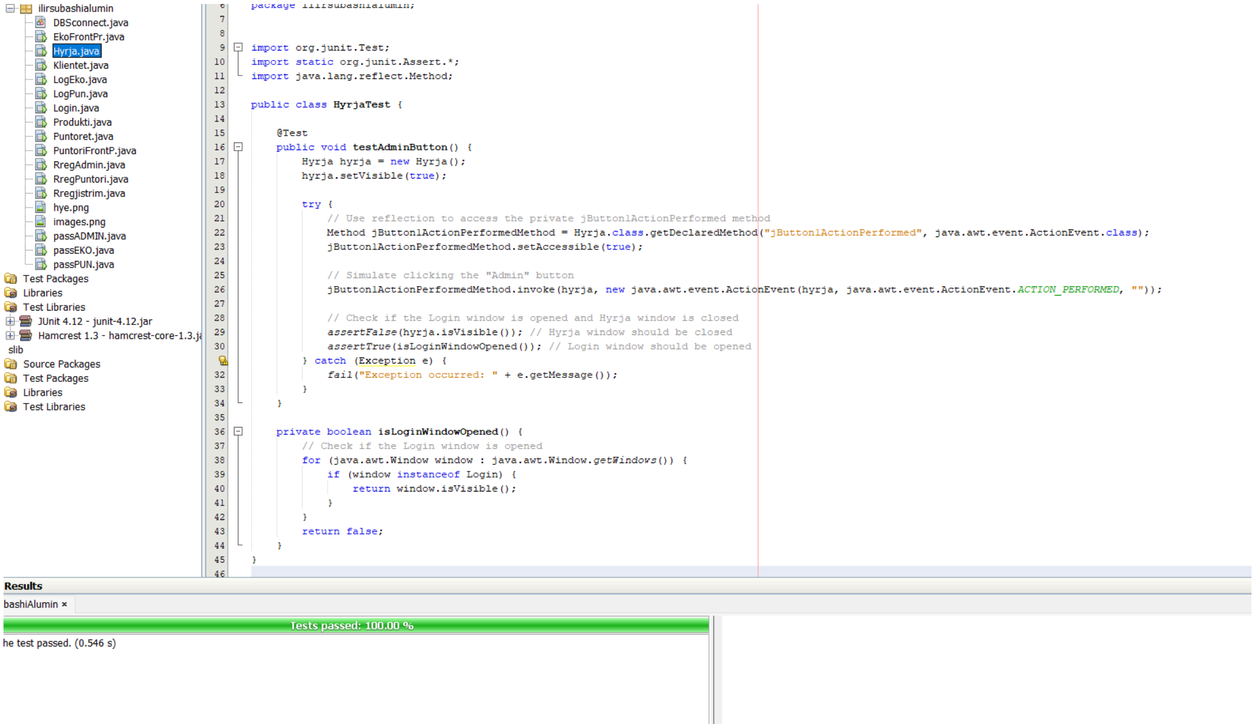


```
package ilirsubashialumin;

import static org.junit.Assert.*;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;


public class DBSconnectTest {

    private static Connection connection;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        // Establish a connection to a test database
        connection = DBSconnect.connect();
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        if (connection != null) {
            connection.close();
        }
    }

    @Test
    public void testConnection() {
        assertNotNull("Connection should not be null", connection);
        try {
            assertFalse("Connection should not be closed", connection.isClosed());
        } catch (SQLException e) {
            fail("SQLException occurred: " + e.getMessage());
        }
    }

    @Test
    public void testDatabase() {
        // Add test cases specific to your database structure or requirements
        try {
            Statement statement = connection.createStatement();

            statement.close();
        } catch (SQLException e) {
            fail("SQLException occurred: " + e.getMessage());
        }
    }
}
```



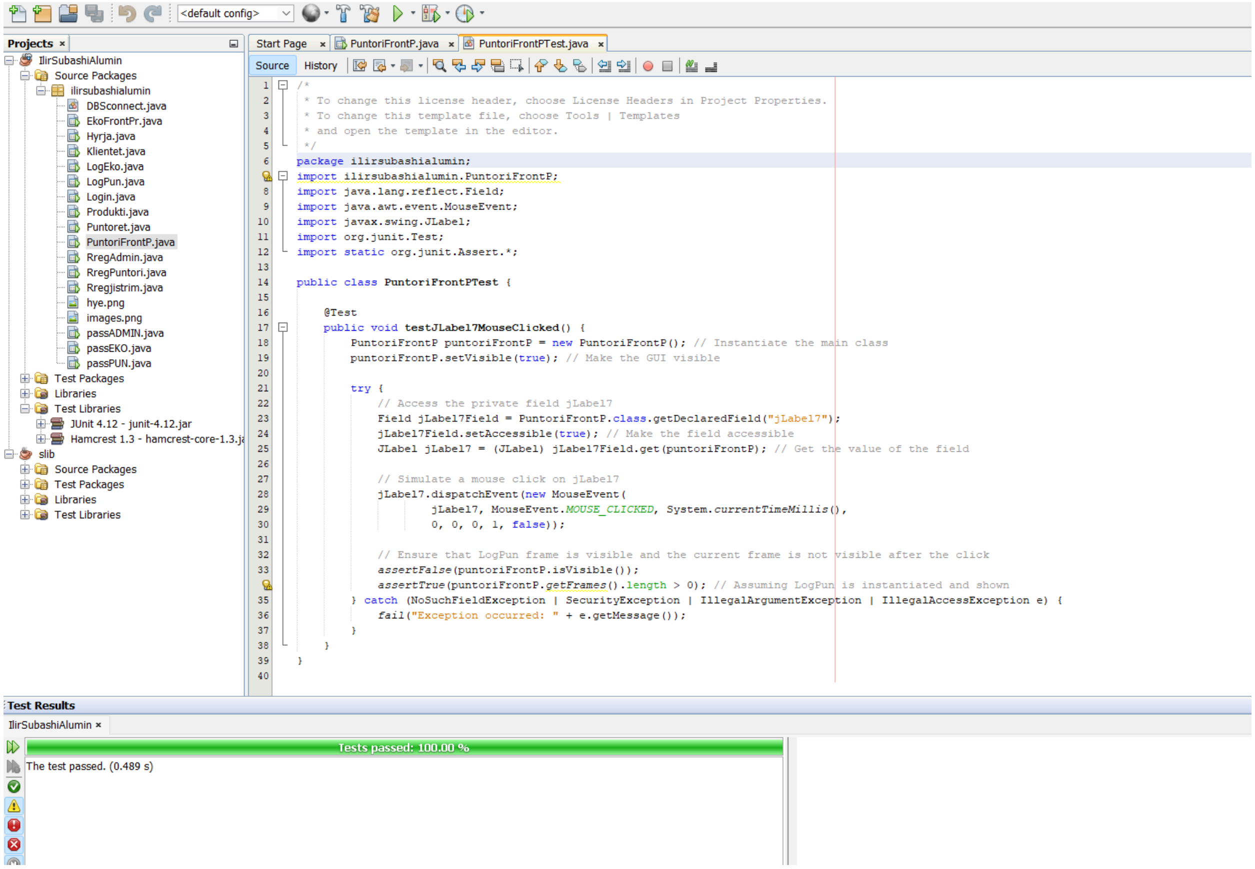
```
package ilirsubashialumin;

import static org.junit.Assert.*;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;


public class DBSconnectTest {

    private static Connection connection;

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        // Establish a connection to a test database
        connection = DBSconnect.connect();
    }

    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        if (connection != null) {
            connection.close();
```

```
Tests passed: 100.00 %
Both tests passed. (0.441 s)
```



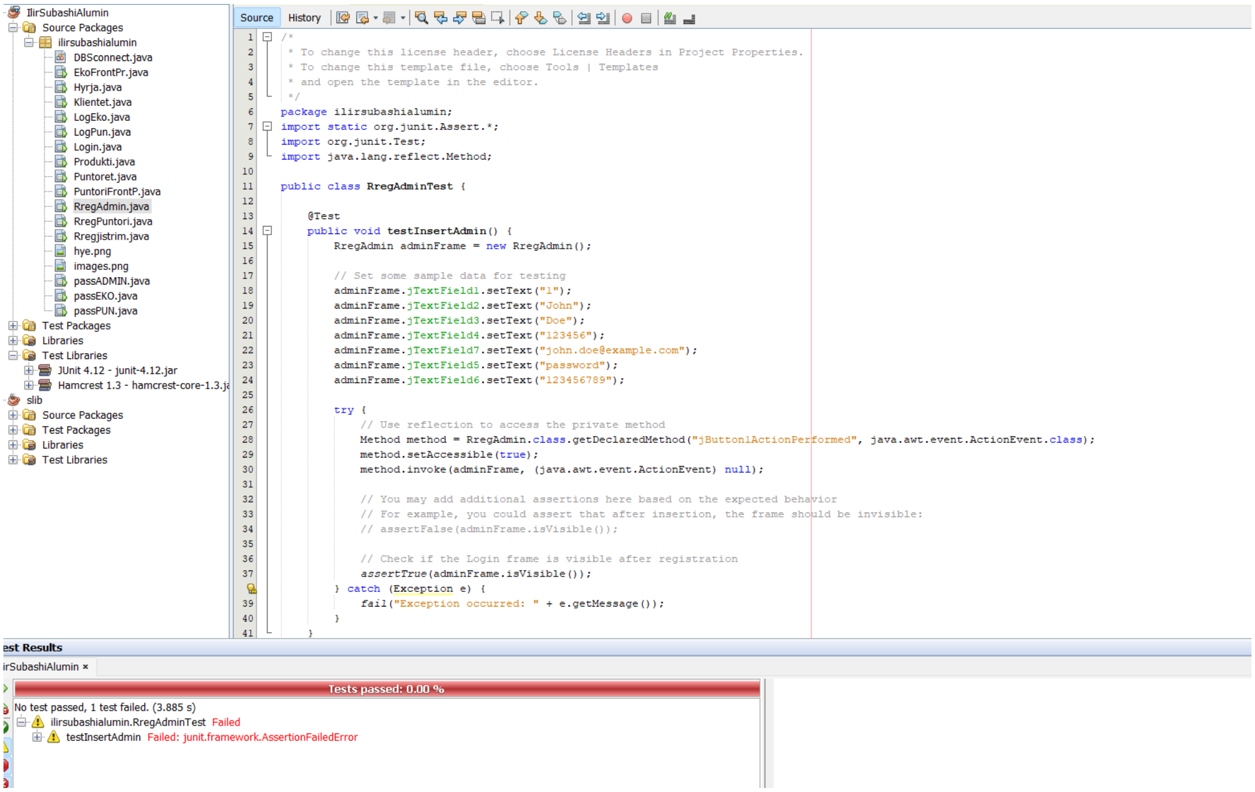
```java
package ilirsubashialumin;


import org.junit.Test;
import static org.junit.Assert.*;
import java.lang.reflect.Method;

public class HyrjaTest {

    @Test
    public void testAdminButton() {
        Hyrja hyrja = new Hyrja();
        hyrja.setVisible(true);

        try {
            // Use reflection to access the private jButton1ActionPerformed method
            Method jButton1ActionPerformedMethod = Hyrja.class.getDeclaredMethod("jButton1ActionPerformed", java.awt.event.ActionEvent.class);
            jButton1ActionPerformedMethod.setAccessible(true);

            // Simulate clicking the "Admin" button
            jButton1ActionPerformedMethod.invoke(hyrja, new java.awt.event.ActionEvent(hyrja, java.awt.event.ActionEvent.ACTION_PERFORMED, ""));

            // Check if the Login window is opened and Hyrja window is closed
            assertFalse(hyrja.isVisible()); // Hyrja window should be closed
            assertTrue(isLoginWindowOpened()); // Login window should be opened
        } catch (Exception e) {
            fail("Exception occurred: " + e.getMessage());
        }
    }

    private boolean isLoginWindowOpened() {
        // Check if the Login window is opened
        for (java.awt.Window window : java.awt.Window.getWindows()) {
            if (window instanceof Login) {
                return window.isVisible();
            }
        }
        return false;
    }
}
```

```
Tests passed: 100.00 %
he test passed. (0.546 s)
```



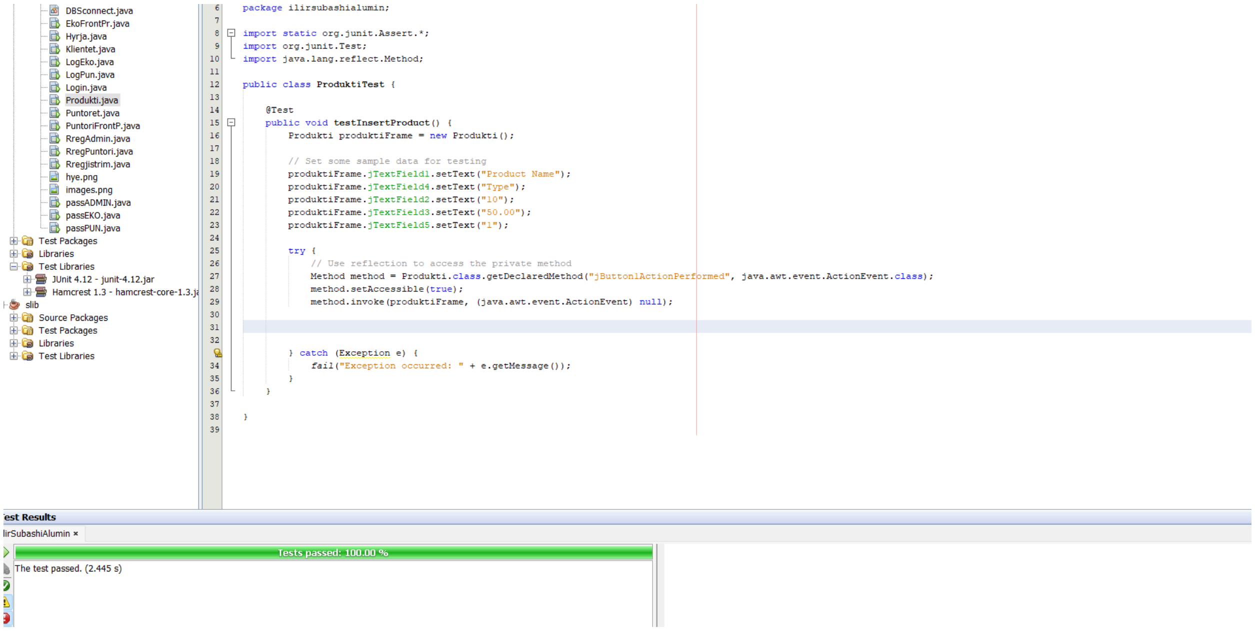
```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package ilirsubashialumin;
import ilirsubashialumin.PuntoriFrontP;
import java.lang.reflect.Field;
import java.awt.event.MouseEvent;
import javax.swing.JLabel;
import org.junit.Test;
import static org.junit.Assert.*;

public class PuntoriFrontPTest {

    @Test
    public void testJLabel7MouseClicked() {
        PuntoriFrontP puntoriFrontP = new PuntoriFrontP(); // Instantiate the main class
        puntoriFrontP.setVisible(true); // Make the GUI visible

        try {
            // Access the private field jLabel7
            Field jLabel7Field = PuntoriFrontP.class.getDeclaredField("jLabel7");
            jLabel7Field.setAccessible(true); // Make the field accessible
            JLabel jLabel7 = (JLabel) jLabel7Field.get(puntoriFrontP); // Get the value of the field

            // Simulate a mouse click on jLabel7
            jLabel7.dispatchEvent(new MouseEvent(
                    jLabel7, MouseEvent.MOUSE_CLICKED, System.currentTimeMillis(),
                    0, 0, 0, 1, false));

            // Ensure that LogPun frame is visible and the current frame is not visible after the click
            assertFalse(puntoriFrontP.isVisible());
            assertTrue(puntoriFrontP.getFrames().length > 0); // Assuming LogPun is instantiated and shown
        } catch (NoSuchFieldException | SecurityException | IllegalArgumentException | IllegalAccessException e) {
            fail("Exception occurred: " + e.getMessage());
        }
    }
}
```

```
Tests passed: 100.00 %
The test passed. (0.489 s)
```



```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package ilirsubashialumin;
import static org.junit.Assert.*;
import org.junit.Test;
import java.lang.reflect.Method;

public class RregAdminTest {

    @Test
    public void testInsertAdmin() {
        RregAdmin adminFrame = new RregAdmin();

        // Set some sample data for testing
        adminFrame.jTextField1.setText("1");
        adminFrame.jTextField2.setText("John");
        adminFrame.jTextField3.setText("Doe");
        adminFrame.jTextField4.setText("123456");
        adminFrame.jTextField7.setText("john.doe@example.com");
        adminFrame.jTextField5.setText("password");
        adminFrame.jTextField6.setText("123456789");

        try {
            // Use reflection to access the private method
            Method method = RregAdmin.class.getDeclaredMethod("jButton1ActionPerformed", java.awt.event.ActionEvent.class);
            method.setAccessible(true);
            method.invoke(adminFrame, (java.awt.event.ActionEvent) null);

            // You may add additional assertions here based on the expected behavior
            // For example, you could assert that after insertion, the frame should be invisible:
            // assertFalse(adminFrame.isVisible());

            // Check if the Login frame is visible after registration
            assertTrue(adminFrame.isVisible());
        } catch (Exception e) {
            fail("Exception occurred: " + e.getMessage());
        }
    }
```

```
Tests passed: 0.00 %
No test passed, 1 test failed. (3.885 s)
ilirsubashialumin.RregAdminTest Failed
    testInsertAdmin Failed: junit.framework.AssertionFailedError
```



```java
package ilirsubashialumin;

import static org.junit.Assert.*;
import org.junit.Test;
import java.lang.reflect.Method;

public class ProduktiTest {

    @Test
    public void testInsertProduct() {
        Produkti produktiFrame = new Produkti();

        // Set some sample data for testing
        produktiFrame.jTextField1.setText("Product Name");
        produktiFrame.jTextField4.setText("Type");
        produktiFrame.jTextField2.setText("10");
        produktiFrame.jTextField3.setText("50.00");
        produktiFrame.jTextField5.setText("1");

        try {
            // Use reflection to access the private method
            Method method = Produkti.class.getDeclaredMethod("jButton1ActionPerformed", java.awt.event.ActionEvent.class);
            method.setAccessible(true);
            method.invoke(produktiFrame, (java.awt.event.ActionEvent) null);



        } catch (Exception e) {
            fail("Exception occurred: " + e.getMessage());
        }
    }

}
```

```
Tests passed: 100.00 %
The test passed. (2.445 s)
```

8. TEST COVERAGE

Achieving high test coverage is crucial for ensuring thorough testing of software. Test coverage refers to the extent to which the code and functionalities of a software application are exercised by test cases. Here's why high test coverage is important:

1. **Identification of Uncovered Areas**: High test coverage helps in identifying areas of the codebase that have not been adequately tested. Uncovered areas are potential breeding grounds for bugs and defects that may remain undetected until the software is in use. By striving for comprehensive coverage, developers and testers can ensure that all parts of the application are rigorously tested, reducing the likelihood of critical issues slipping through.

2. **Improved Software Quality**: Thorough testing contributes to higher software quality. When a significant portion of the codebase is covered by tests, it increases confidence in the software's reliability and functionality. High test coverage means that more scenarios and edge cases are considered during testing, leading to fewer bugs and defects in the final product. This, in turn, enhances the overall user experience and satisfaction with the software.

3. **Effective Maintenance and Refactoring:** High test coverage makes software maintenance and refactoring easier and safer. When developers need to make changes to the codebase, whether it's adding new features or fixing bugs, having a comprehensive suite of tests provides a safety net. Tests act as a form of documentation, helping developers understand the intended behavior of the code and ensuring that modifications do not inadvertently introduce regressions or break existing functionality.

4. **Facilitates Continuous Integration and Delivery**: In modern software development practices such as continuous integration and delivery (CI/CD), having high test coverage is essential. Automated testing is a cornerstone of CI/CD pipelines, allowing teams to rapidly deploy changes with confidence. High test coverage ensures that automated tests provide meaningful feedback on the quality of each code change, enabling teams to release software more frequently and reliably.

In summary, achieving high test coverage is vital for thorough testing of software because it helps identify uncovered areas, improves software quality, facilitates maintenance and refactoring, and enables effective CI/CD practices. By prioritizing comprehensive testing, software development teams can deliver more reliable and robust applications to their users.