كلية علوم وهندسة الحاسب
College of Computer Science & Engineering

جامعة جدة
University of Jeddah

Software Security

Project

Part 2: Static and Dynamic analysis

Lara Sami Alofi

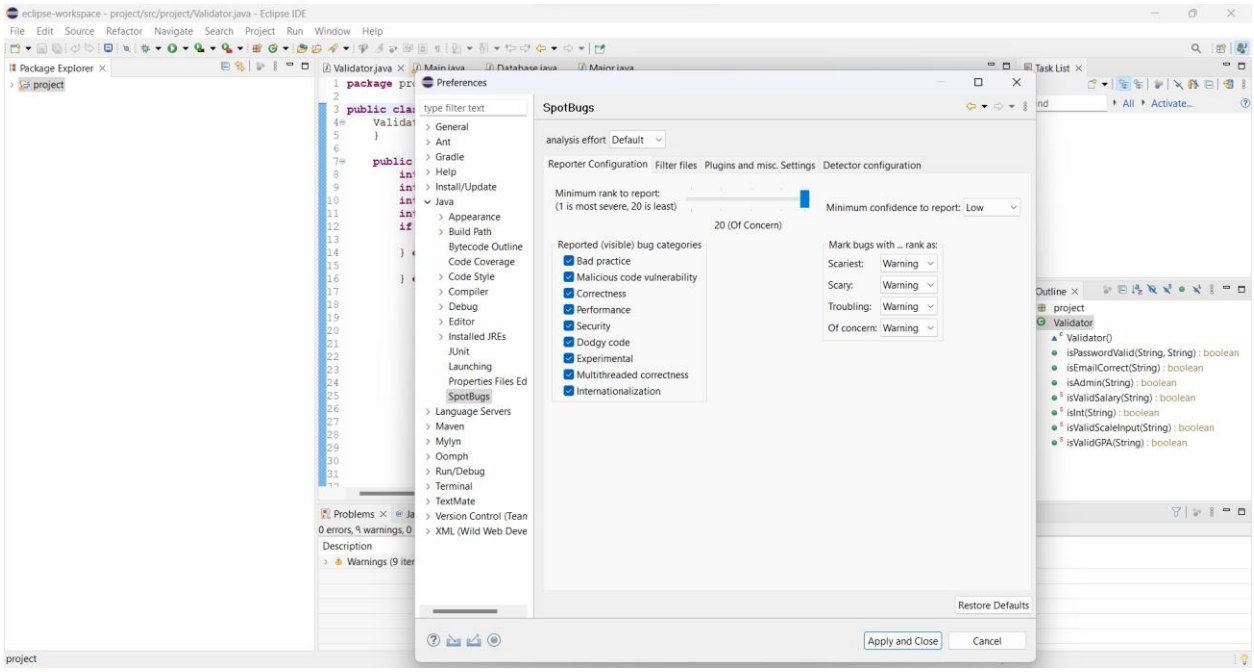
2110886

CY8

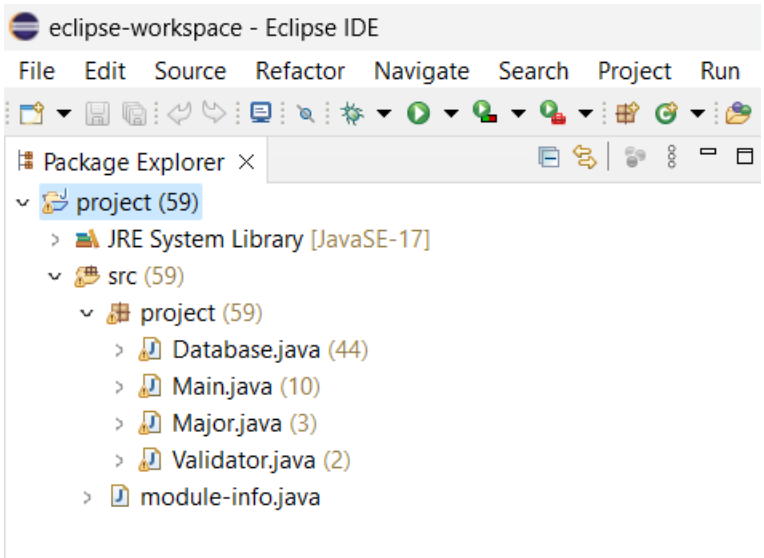
`

Static analysis using SpotBugs in eclipse.

- After installing spotbugs to analyze the code, I set the configuration as follows:

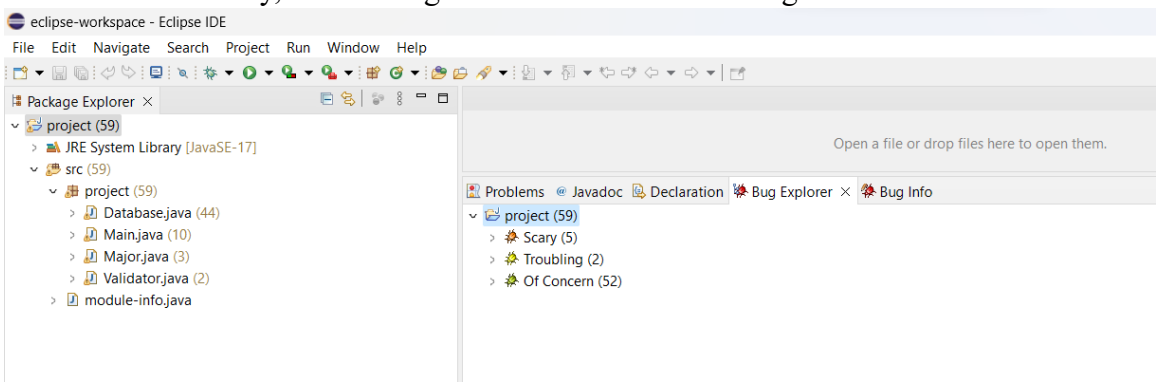


- Here is an overview of the project, including its classes and the number of bugs found in each class:

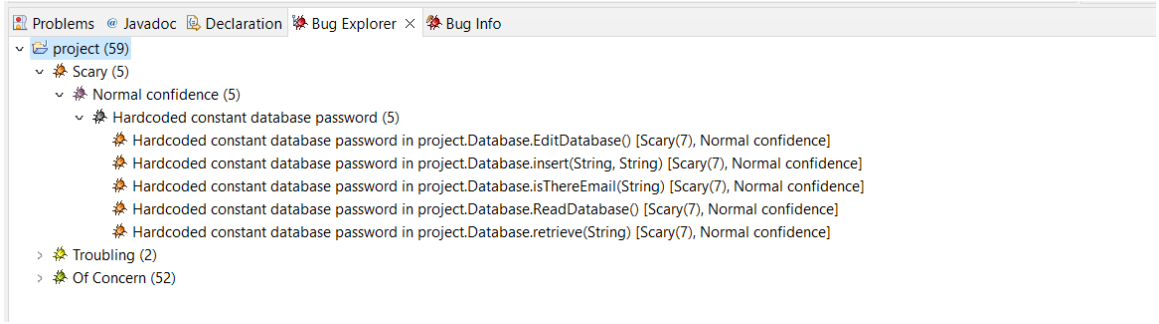


`

- Here we find 5 Scary, 2 Troubling and 52 Of Concern ranking:



- **The first bugs I will explain it is Scary:**

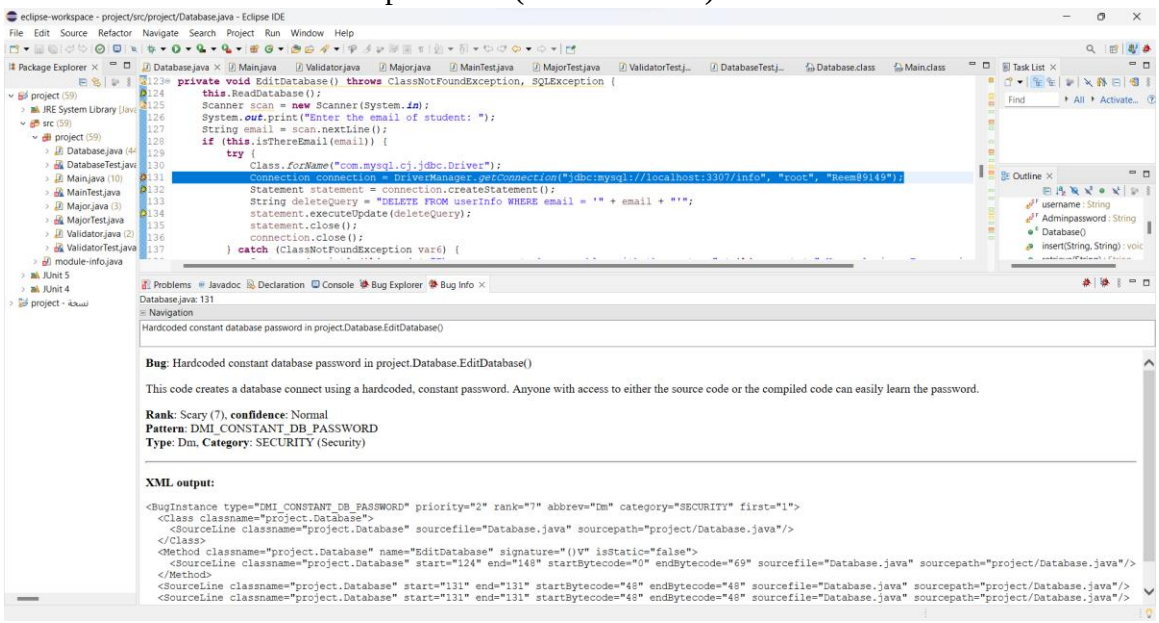


1. Scary (Confidence: 5)
   Description: This bug refers to a hardcoded constant database password that is present in the project.
   Details: The presence of a hardcoded constant database password poses a significant security risk. It means that the password is directly embedded in the code, making it easier for unauthorized individuals to access the database. This can lead to unauthorized data access, potential data breaches, and compromise of sensitive information. To address this issue, it is crucial to remove the hardcoded password and implement secure authentication mechanisms that do not rely on fixed credentials.

2. Normal Confidence (Confidence: 5)
   Description: This bug indicates a concern of normal confidence level, but specific details about the bug are not provided.
   Details: Without specific details, it is challenging to provide further information about the bug. To effectively address and resolve the concern, it is necessary to investigate and provide more specific information about its nature and potential impact.

`

3. Hardcoded constant database password (Confidence: 5)



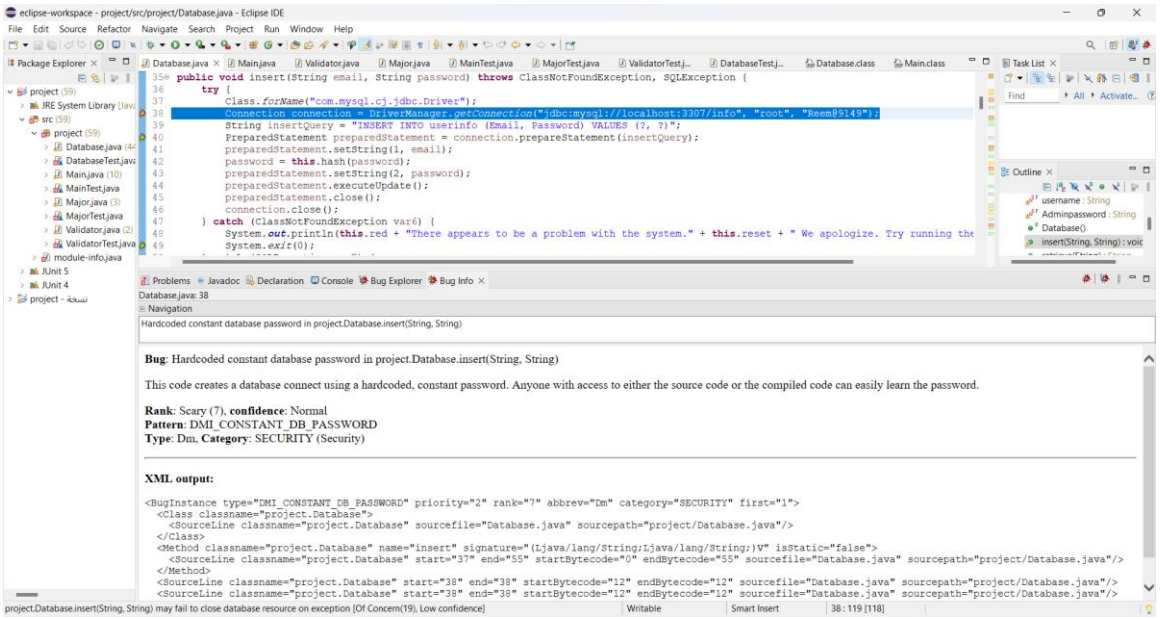

```
<BugInstance type="DMI_CONSTANT_DB_PASSWORD" priority="2" rank="7" abbrev="Dm" category="SECURITY" first="1">
  <Class classname="project.Database">
    <SourceLine classname="project.Database" sourcefile="Database.java" sourcepath="project/Database.java"/>
  </Class>
  <Method classname="project.Database" name="EditDatabase" signature="()V" isStatic="false">
    <SourceLine classname="project.Database" start="124" end="148" startBytecode="0" endBytecode="69" sourcefile="Database.java" sourcepath="project/Database.java"/>
  </Method>
  <SourceLine classname="project.Database" start="131" end="131" startBytecode="48" endBytecode="48" sourcefile="Database.java" sourcepath="project/Database.java"/>
  <SourceLine classname="project.Database" start="131" end="131" startBytecode="48" endBytecode="48" sourcefile="Database.java" sourcepath="project/Database.java"/>
```

Description: This bug points to the presence of a hardcoded constant database password in the project.

Details: Similar to the first bug, having a hardcoded constant database password is problematic from a security perspective. It means that the password is directly written in the code, making it easier for unauthorized individuals to gain access to the database. This can lead to unauthorized data manipulation, data leakage, and other security breaches. To mitigate this risk, it is important to remove the hardcoded password and implement secure password management practices, such as using encryption or secure credential storage mechanisms.

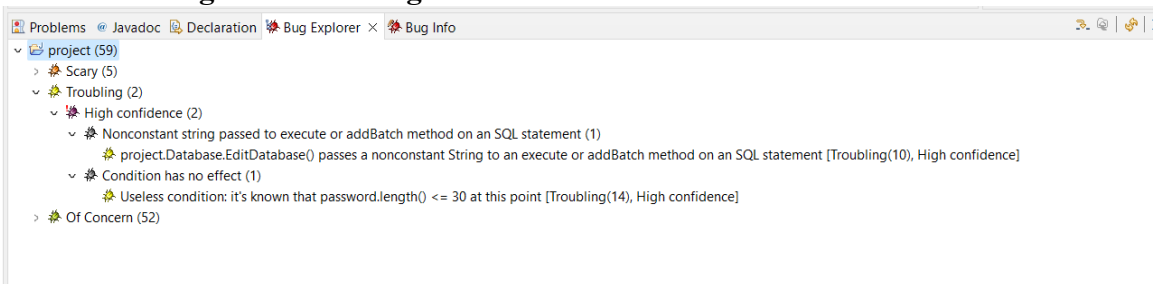
`

4. Hardcoded constant database password in project Database EditDatabase() [Scary (7), Normal confidence]



Description: This bug specifically refers to a hardcoded constant database password in the 'EditDatabase()' function of the 'Database' project.
Details: The presence of a hardcoded constant password in the 'EditDatabase()' function poses a serious security vulnerability. It means that anyone with access to the code can easily retrieve the password and potentially gain unauthorized access to the database. This can lead to unauthorized data modifications, data loss, and other security incidents. To address this issue, it is necessary to eliminate the hardcoded password and implement secure authentication mechanisms that protect the database with strong, dynamically generated passwords.

- **The second bugs are Troubling:**



1. Troubling (Confidence: 2)

Description: This bug is classified as "Troubling" and has a confidence level of 2.

Details: Unfortunately, no specific details about the bug are provided. It would be helpful to gather more information about the bug, such as the context in which it occurs or any error messages or unexpected behaviors observed. This will enable a more accurate and effective resolution of the issue.

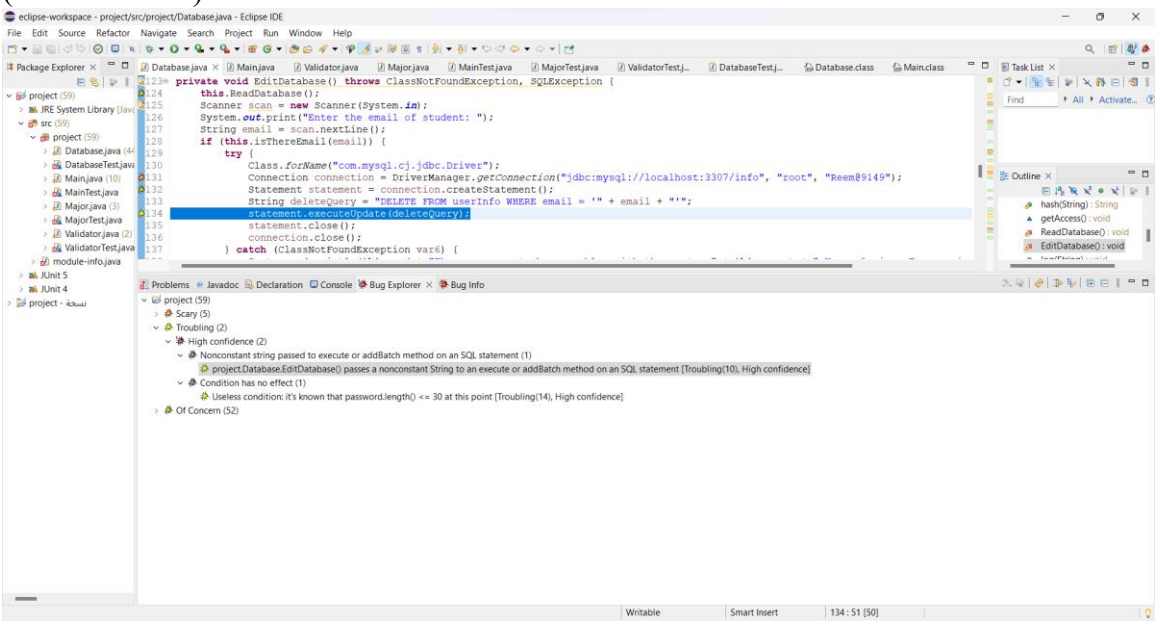
`

2. High Confidence (Confidence: 2)

Description: This bug is classified as having "High Confidence" and has a confidence level of 2.

Details: Similar to the previous bug, no specific details are provided regarding the bug. It is essential to investigate further and provide more information about the bug, including any error messages, logs, or steps to reproduce the issue. This will help in identifying the root cause and resolving the bug effectively.

3. Nonconstant string passed to execute or addBatch method on an SQL statement (Confidence: 1)

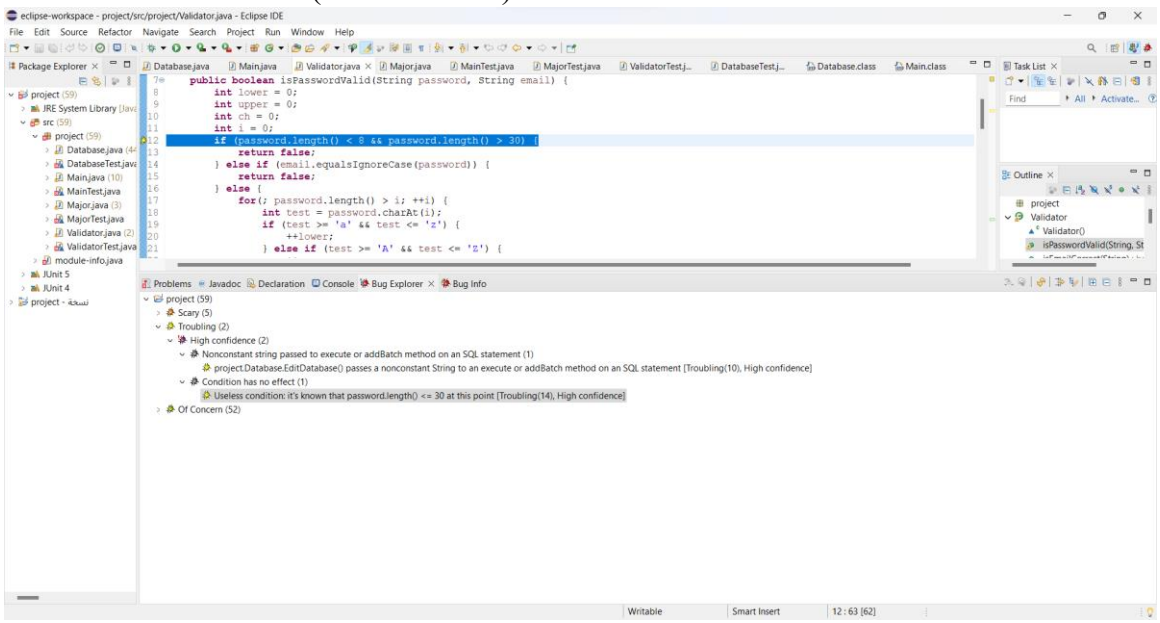


Description: This bug refers to a situation where a nonconstant string is passed to the `execute` or `addBatch` method on an SQL statement.

Details: When constructing SQL statements, it is generally recommended to use parameterized queries or prepared statements to prevent SQL injection attacks and ensure the safety and integrity of the database. Passing nonconstant strings directly to the `execute` or `addBatch` method can introduce vulnerabilities and potential security risks. It is advisable to review the code and ensure that appropriate sanitization and parameterization techniques are implemented to handle user input securely.

`

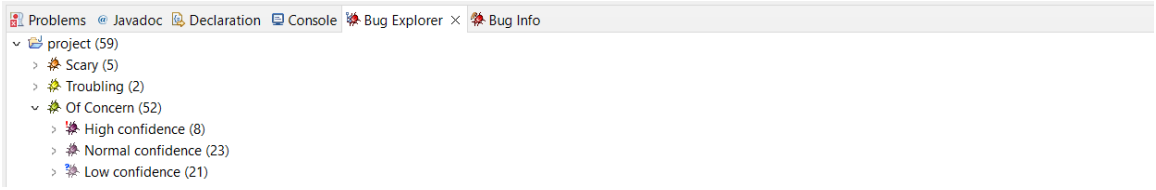
4. Condition has no effect (Confidence: 1)



Description: This bug indicates that a condition in the code has no effect or does not impact the program's logic or execution flow.

Details: When a condition has no effect, it means that the condition does not change the behavior or outcome of the program. This could be due to incorrect or redundant logic, a missing or mismatched variable, or an unintended coding error. It is important to review the condition carefully, consider the intended logic, and make necessary adjustments to ensure the condition is correctly implemented and has the desired effect on the program's behavior.

1. **The third bugs is Of Concern:**

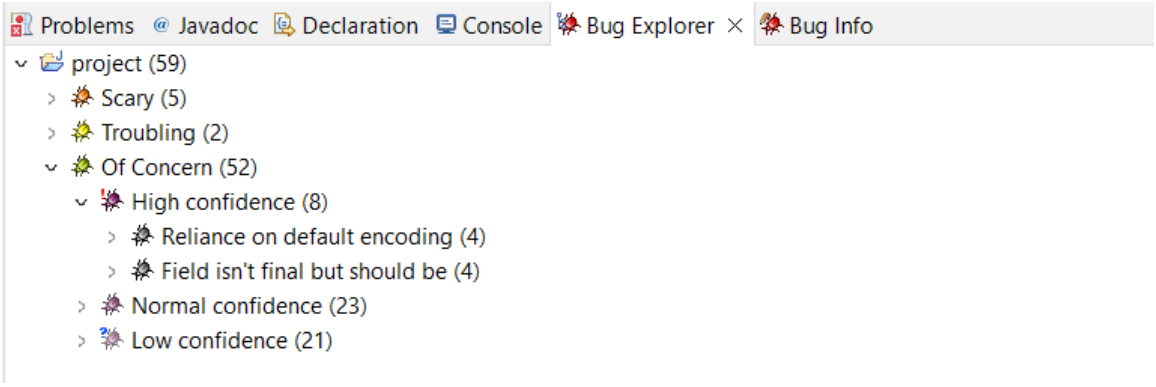


Of Concern (Confidence: 52)

Description: This bug is classified as "Of Concern" and has a high confidence level of 52.
Details: Unfortunately, no specific details about the bug are provided. To effectively address this bug, it would be helpful to gather more information, such as the context in which it occurs, any observed issues or errors, and any steps to reproduce the problem. This additional information will allow for a more accurate identification and resolution of the issue.

`

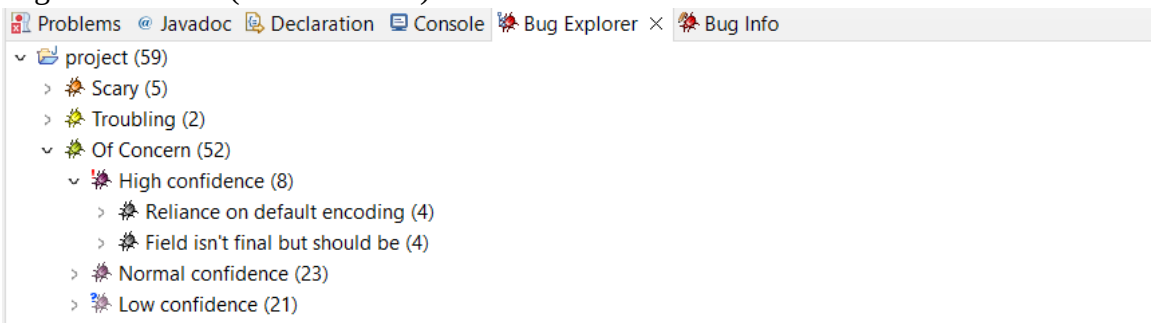
2. Of Concern (Confidence: 52)



Description: This bug is classified as "Of Concern" and has a high confidence level of 52.
Details: Unfortunately, no specific details about the bug are provided. To effectively address this bug, it would be helpful to gather more information, such as the context in which it occurs, any observed issues or errors, and any steps to reproduce the problem. This additional information will allow for a more accurate identification and resolution of the issue.

We have 3 types of Concerns high, normal and low:
High confidence = 8
Normal confidence = 23
Low confidence = 21

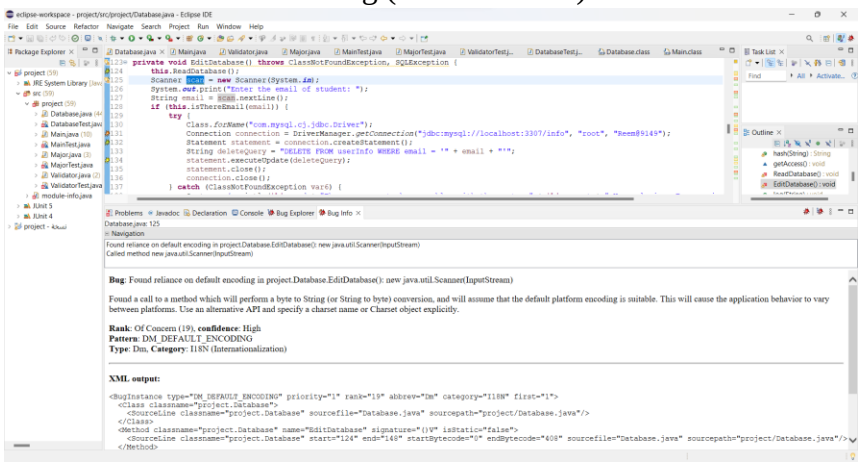
1. High Confidence (Confidence: 8)



Description: This bug is classified as having "High Confidence" and has a confidence level of 8.
Details: Similar to the previous bug, no specific details about the bug are provided. It is important to gather more information, such as error messages, logs, or any observed behaviors, to investigate and resolve the issue effectively. This additional information will assist in identifying the root cause and taking appropriate corrective actions.

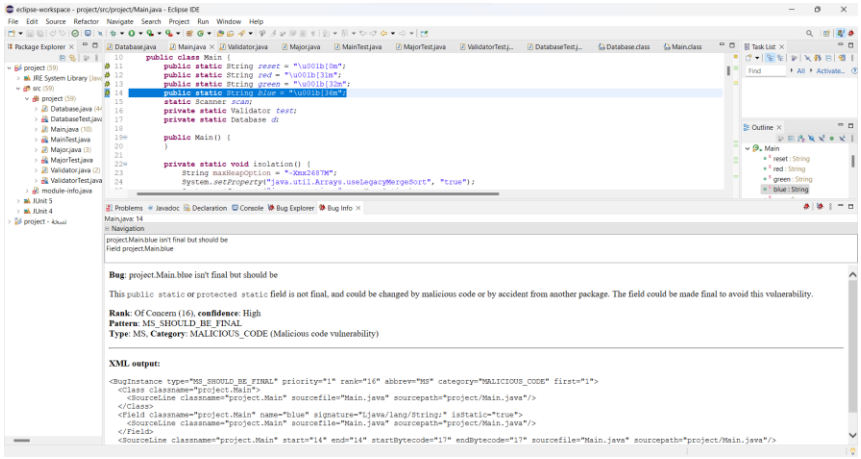
`

2. Reliance on default encoding (Confidence: 4)



Description: This bug refers to a situation where a software application relies on the default encoding without explicitly specifying it.
Details: Relying on the default encoding without explicitly specifying it can lead to various issues. It can result in data corruption, loss of information, and difficulties in interoperability. When the default encoding is used, there is a risk of misinterpreting or incorrectly processing text data. This can cause text to appear garbled or distorted, leading to inconsistencies or inaccuracies in the data. To mitigate this bug, it is essential to explicitly specify the appropriate encoding for data processing and handling to ensure consistent and accurate interpretation of text data.

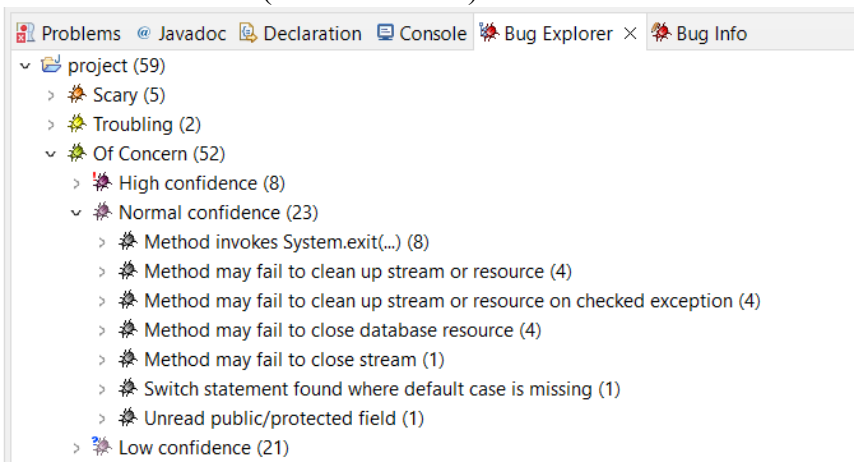
3. Field isn't final but should be (Confidence: 4)



Description: This bug indicates that a field in the code is not declared as "final" but should be.
Details: When a field is not declared as "final" but should be, it means that the field's value can be modified, even though it should remain constant. Declaring a field as "final" is important for immutability, ensuring that its value cannot be changed once assigned. Failing to declare a field as "final" when it should be can lead to unintended modifications, potential bugs, and difficulties in maintaining code consistency. To address this bug, it is necessary to review the code and identify fields that should be declared as "final" to prevent accidental modifications and ensure code stability.

`

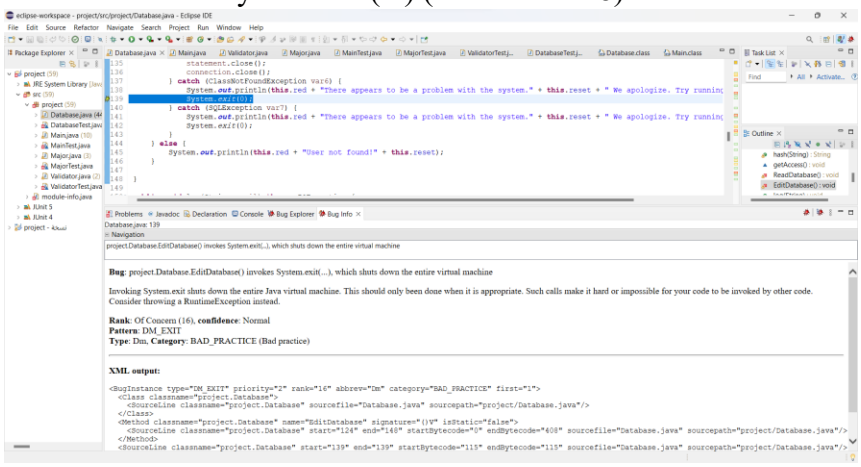
1. Normal Confidence (Confidence: 23)



Description: This bug is classified as having "Normal Confidence" and has a confidence level of 23.

Details: Similar to the previous bug, no specific details about the bug are provided. It is important to gather more information, such as error messages, logs, or any observed behaviors, to investigate and resolve the issue effectively. This additional information will assist in identifying the root cause and taking appropriate corrective actions.

2. Method invokes System.exit(...) (Confidence: 8)



Description: This bug indicates that a method in the code is invoking the System.exit(...) method.

Details: Invoking the System.exit(...) method forcefully terminates the Java Virtual Machine (JVM) and abruptly stops the program. This can be problematic in certain scenarios where a graceful termination is desired, such as when running multiple components or modules within a larger system. To address this bug, it is recommended to review the code and consider alternative approaches to handle program termination, such as returning specific values or throwing appropriate exceptions, to allow for better control and handling of program flow.

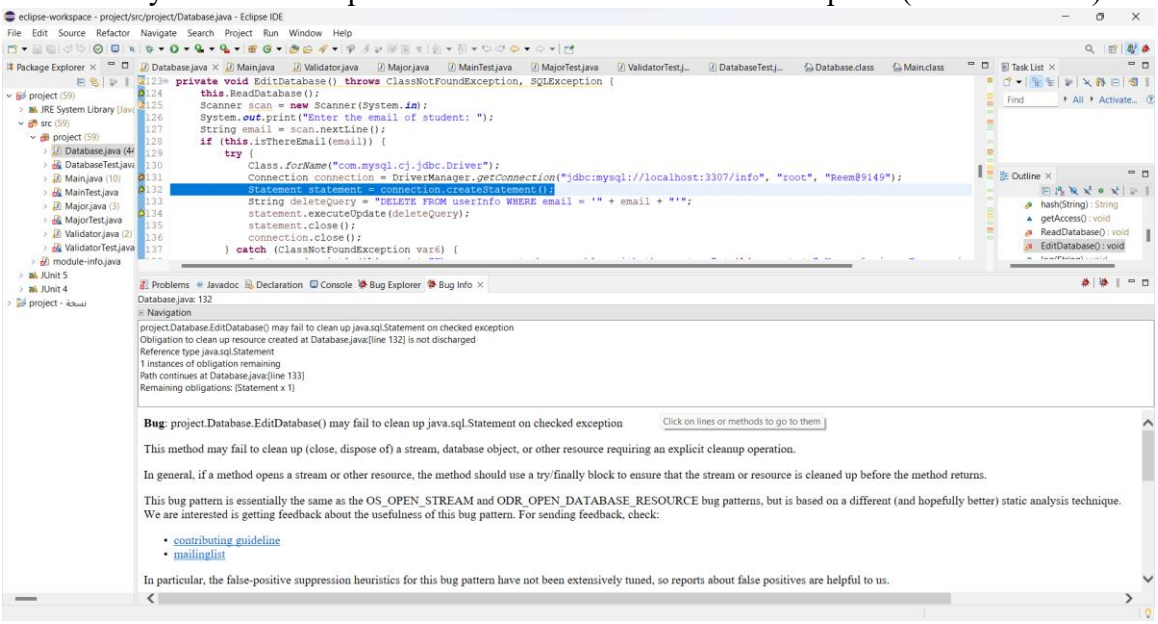
`

3. Method may fail to clean up stream or resource (Confidence: 4)

Description: This bug suggests that a method in the code may fail to properly clean up a stream or resource.

Details: Failing to clean up streams or resources can lead to resource leaks, potential memory issues, or other undesirable consequences. It is important to ensure that streams or resources, such as file handles or network connections, are properly closed or released after use. This can be achieved by using try-with-resources blocks, finally blocks, or appropriate cleanup mechanisms. Reviewing the code and implementing proper resource management techniques will help mitigate this bug.

4. Method may fail to clean up stream or resource on checked exception (Confidence: 4)



Description: This bug suggests that a method in the code may fail to properly clean up a stream or resource in the presence of a checked exception.

Details: When a method encounters a checked exception, proper handling and cleanup of resources should be ensured. Failing to do so can lead to resource leaks or other issues. It is important to review the code and implement appropriate exception handling mechanisms, such as try-catch-finally blocks, to ensure proper cleanup of resources even in the presence of exceptions.

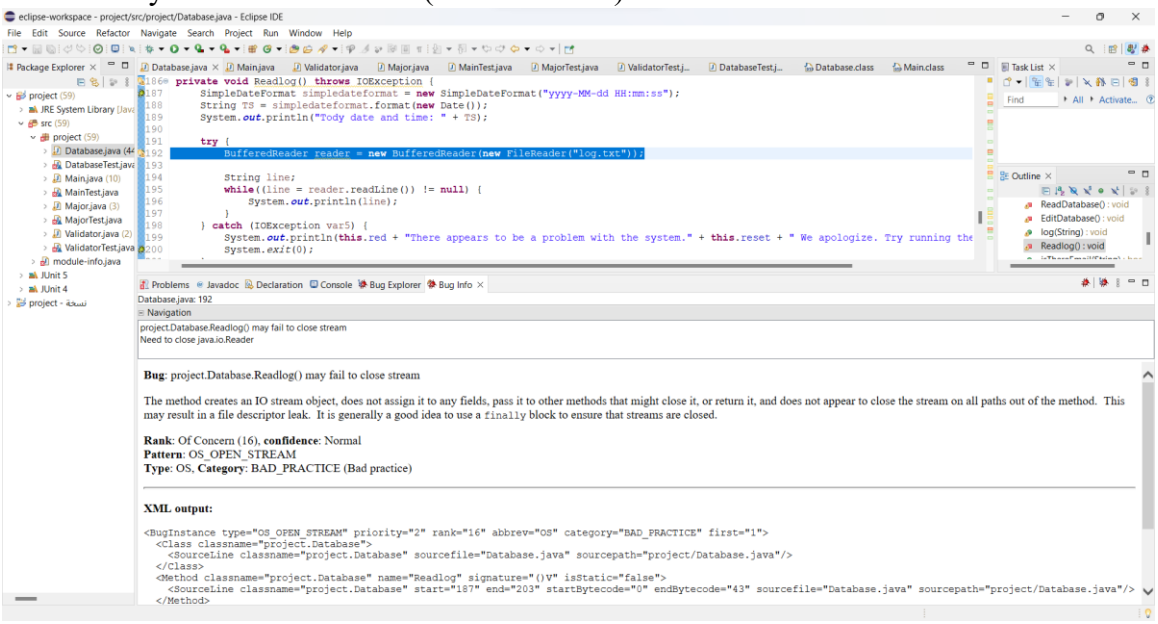
`

5. Method may fail to close database resource (Confidence: 4)

Description: This bug indicates that a method in the code may fail to properly close a database resource.

Details: Failing to close database resources, such as connections, statements, or result sets, can lead to resource exhaustion, connection leaks, or other database-related issues. It is important to review the code and ensure that database resources are properly closed or released after use. This can be achieved using try-with-resources blocks or by explicitly calling the appropriate close or release methods. Implementing proper resource management techniques will help address this bug.

6. Method may fail to close stream (Confidence: 1)



Description: This bug suggests that a method in the code may fail to properly close a stream.

Details: Failing to close streams, such as input streams or output streams, can result in resource leaks, potential memory issues, or unexpected behavior. It is essential to review the code and ensure that streams are properly closed after use. This can be done using try-with-resources blocks or by invoking the close method explicitly. Proper stream management will help prevent this bug and ensure efficient resource utilization.

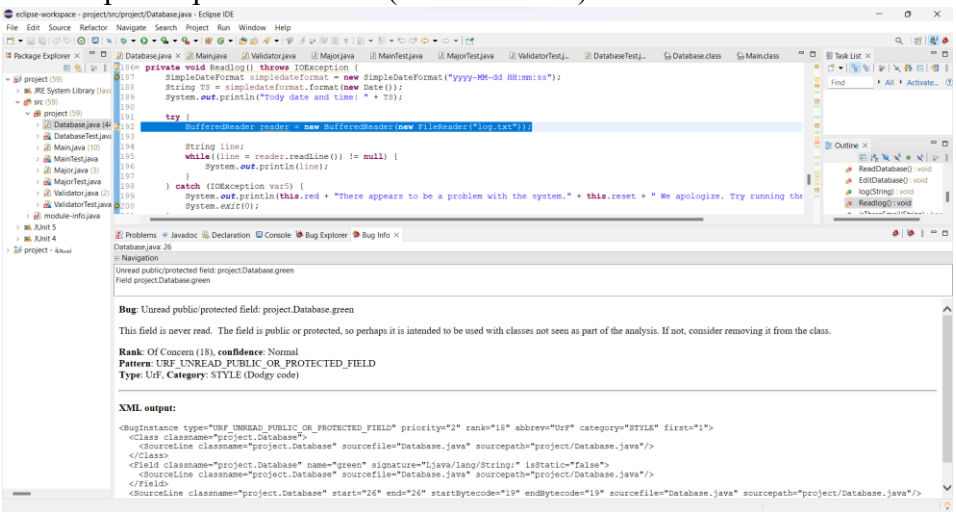
`

7. Switch statement found where default case is missing (Confidence: 1)

Description: This bug indicates that a switch statement is present in the code but lacks a default case.

Details: When using a switch statement, it is generally recommended to include a default case to handle unexpected or unmatched cases. Failing to include a default case can lead to unhandled situations, potential bugs, or unintended behavior. To address this bug, it is advisable to review the code and add a default case that covers any unmatched cases, providing appropriate handling or error reporting as needed.

8. Unread public/protected field (Confidence: 1)

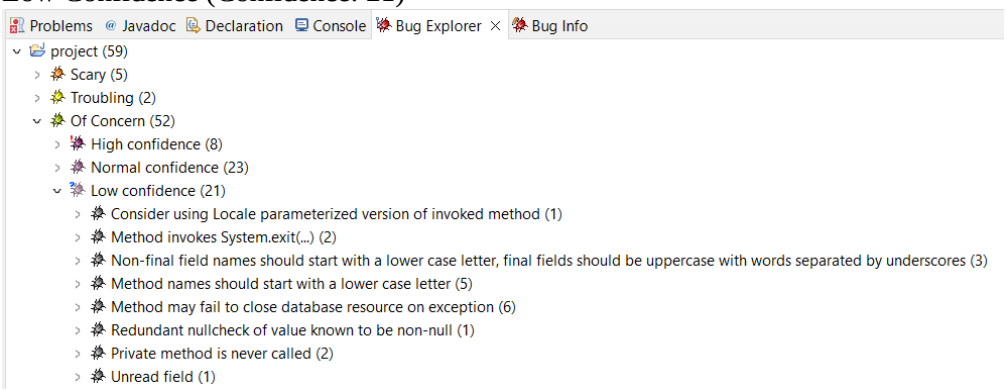


Description: This bug suggests that a public or protected field is declared but remains unused or unread in the code.

Details: Declaring fields that are not utilized or accessed can indicate redundant or unnecessary code. It is important to review the code and ensure that all declared fields areutilized appropriately. If a public or protected field is not needed, it can be removed to improve code clarity and maintainability. However, if the field serves a specific purpose and is intended to be accessed externally, it should be used as intended or evaluated for potential refactoring to ensure its proper utilization.

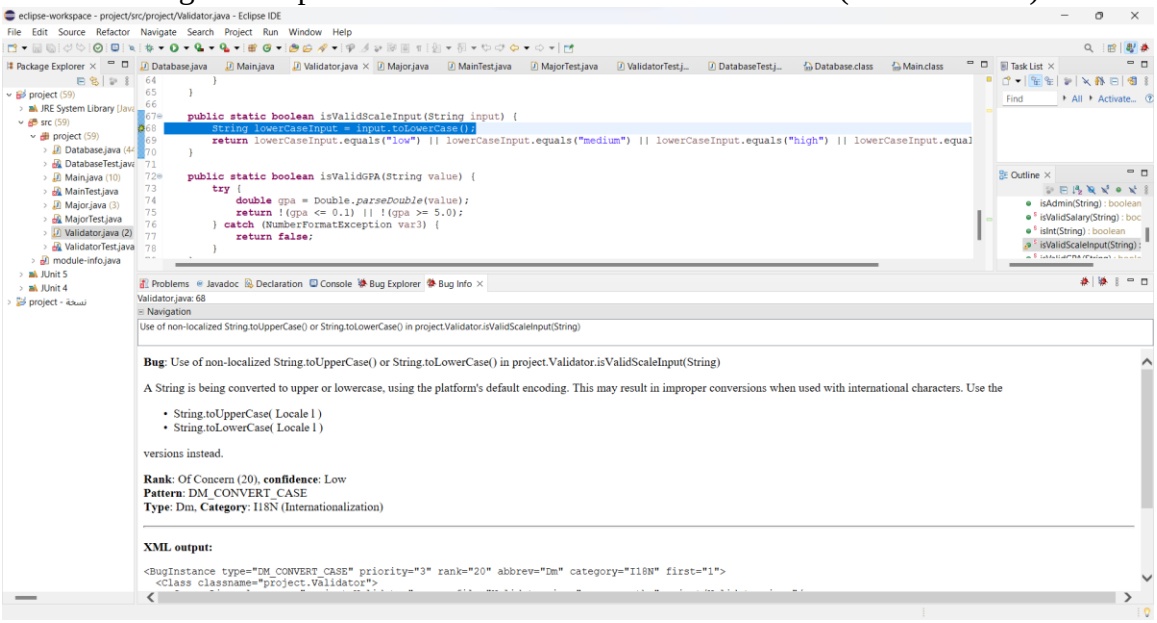
`

1. Low Confidence (Confidence: 21)



Description: This bug is classified as having "Low Confidence" and has a confidence level of 21.
Details: Similar to the previous bug, no specific details about the bug are provided. It is important to gather more information, such as error messages, logs, or any observed behaviors, to investigate and resolve the issue effectively. This additional information will assist in identifying the root cause and taking appropriate corrective actions.

2. Consider using Locale parameterized version of invoked method (Confidence: 1)

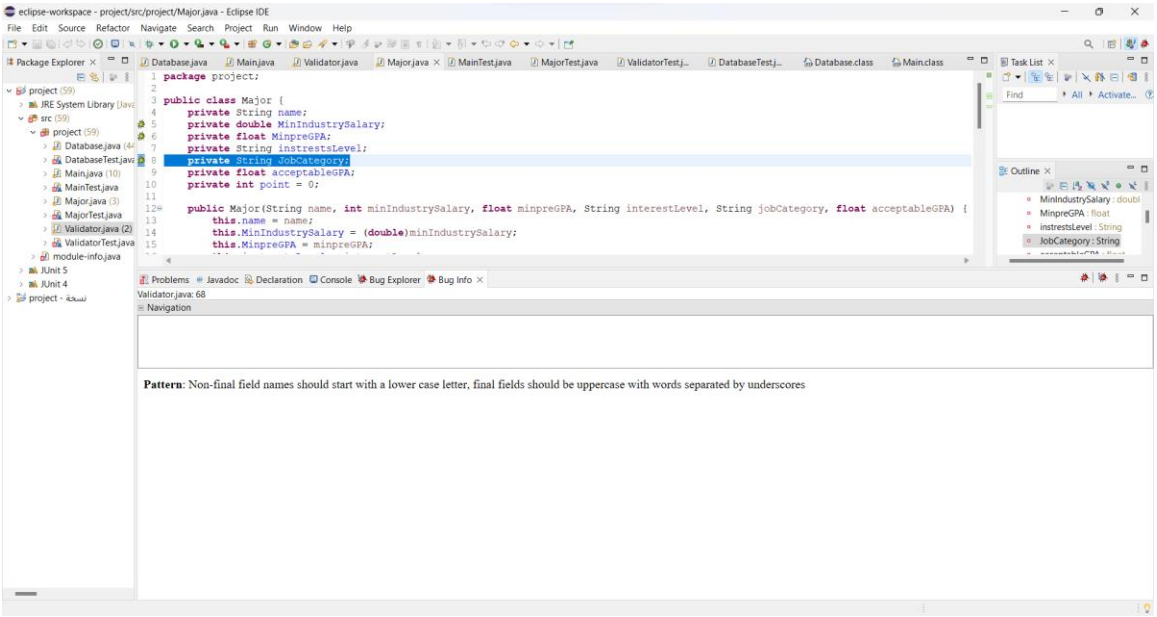


Description: This bug suggests using the parameterized version of a method that supports localization by specifying a Locale.
Details: When working with methods that involve localization, it is recommended to use the parameterized versions of those methods that accept a Locale as a parameter. This allows for better language and cultural customization. To address this bug, review the code and replace the method invocation with the appropriate parameterized version, specifying the desired Locale.

`

3. Method invokes System.exit() (Confidence: 2)
   Description: This bug indicates that a method in the code is invoking the System.exit() method.
   Details: Invoking the System.exit() method forcefully terminates the Java Virtual Machine (JVM) and abruptly stops the program. This can be problematic in certain scenarios where a graceful termination is desired, such as when running multiple components or modules within a larger system. To address this bug, it is recommended to review the code and consider alternative approaches to handle program termination, such as returning specific values or throwing appropriate exceptions, to allow for better control and handling of program flow.

4. Non-final field names should start with a lowercase letter, final fields should be uppercase with words separated by underscores (Confidence: 3)



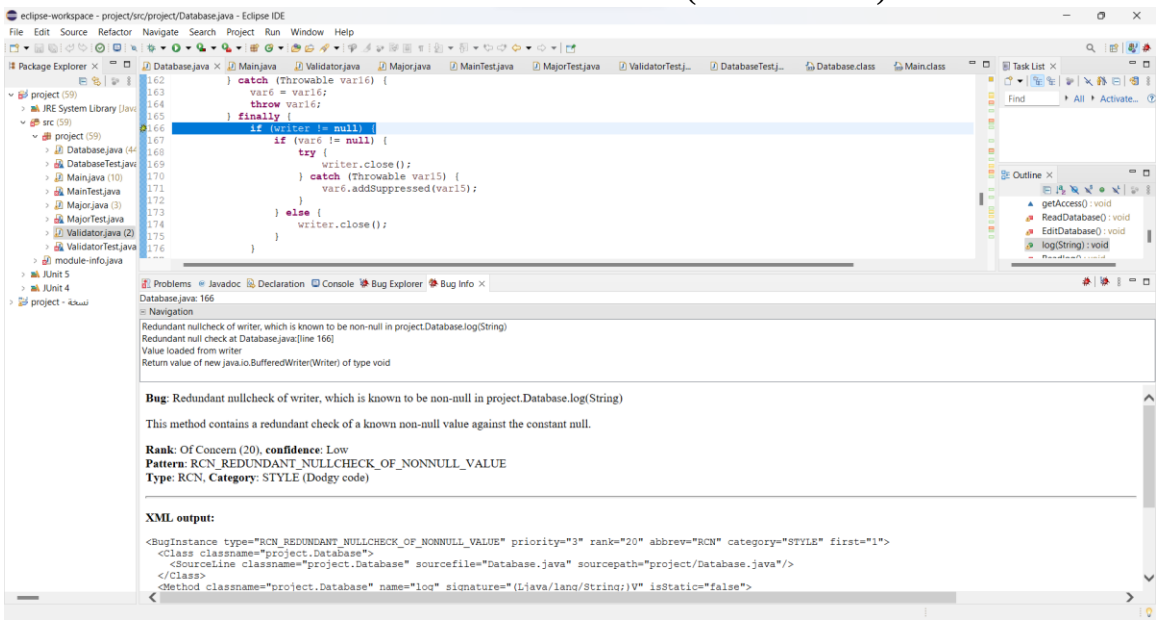
   Description: This bug suggests following a naming convention for fields where non-final fields start with a lowercase letter, and final fields are written in uppercase with words separated by underscores.
   Details: Adhering to naming conventions improves code readability and maintainability. In general, non-final fields should be named using camel case starting with a lowercase letter, while final fields should be named using uppercase letters with words separated by underscores. Review the code and update the field names to follow the appropriate naming convention.

5. Method names should start with a lowercase letter (Confidence: 5)
   Description: This bug suggests that method names should start with a lowercase letter.
   Details: In Java, it is a convention for method names to start with a lowercase letter. This convention enhances code readability and consistency. Review the code and update the method names to start with a lowercase letter, ensuring that they adhere to the accepted naming convention.

`

6. Method may fail to close a database resource on an exception (Confidence: 6)
   Description: This bug indicates that a method in the code may fail to properly close a database resource in the event of an exception.
   Details: When working with database resources, it is crucial to ensure proper resource management, especially when exceptions occur. Failing to handle exceptions and properly close database resources can lead to resource leaks or other issues. Review the code and implement appropriate exception handling mechanisms, such as try-catch-finally blocks, to ensure proper cleanup of database resources, even in the presence of exceptions.

7. Redundant null check of value known to be non-null (Confidence: 1)

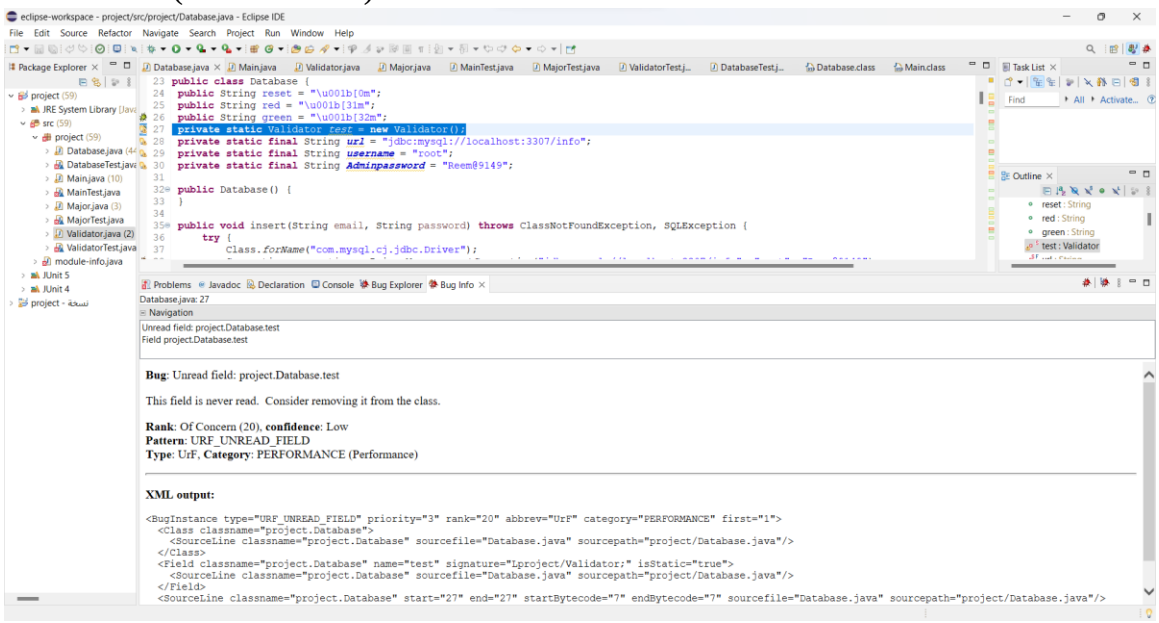


   Description: This bug suggests that there is a redundant null check in the code for a value that is known to be non-null.
   Details: Performing null checks for values that are guaranteed to be non-null can introduce unnecessary code complexity and reduce readability. Review the code and remove the redundant null check for the known non-null value to simplify the code and improve its clarity.

8. Private method is never called (Confidence: 2)
   Description: This bug indicates that a private method in the code is never called.
   Details: A private method that is never called serves no purpose and can be considered dead code. Review the code and ensure that the private method is either called from within the class or remove it if it is no longer needed. Removing unused private methods improves code cleanliness and reduces potential confusion.

`

9. Unread field (Confidence: 1)

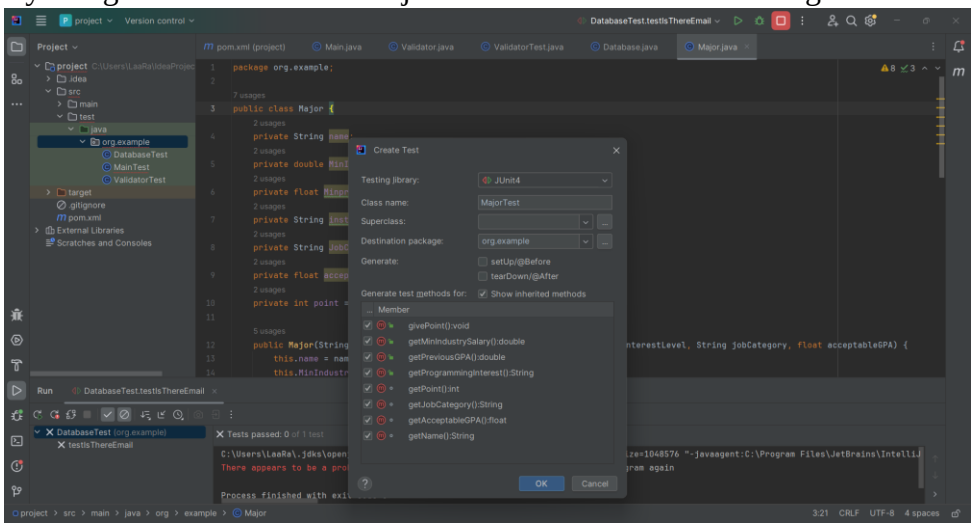


Description: This bug suggests that a field in the code is declared but remains unread or unused.
Details: Declaring fields that are not utilized or accessed can indicate redundant or unnecessary

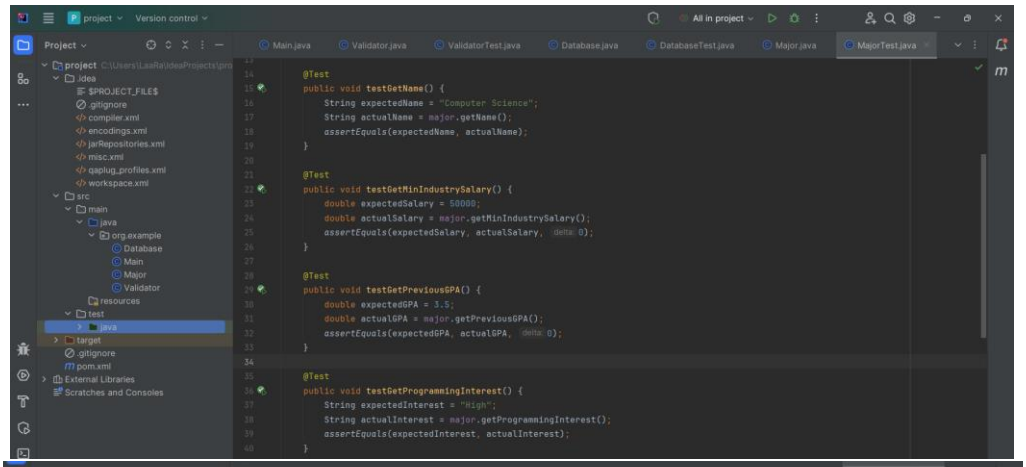
`

Dynamic analysis result

By using Junit in IntelliJ in Major class I chose all methods to generate:



the methods test I used:

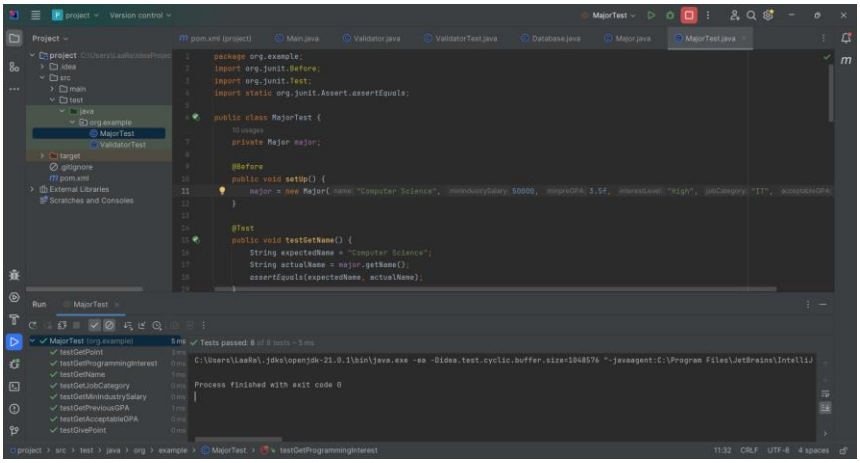


```java
@Test
public void testGetName() {
    String expectedName = "Computer Science";
    String actualName = major.getName();
    assertEquals(expectedName, actualName);
}

@Test
public void testGetMinIndustrySalary() {
    double expectedSalary = 50000;
    double actualSalary = major.getMinIndustrySalary();
    assertEquals(expectedSalary, actualSalary, delta: 0);
}

@Test
public void testGetPreviousGPA() {
    double expectedGPA = 3.5;
    double actualGPA = major.getPreviousGPA();
    assertEquals(expectedGPA, actualGPA, delta: 0);
}

@Test
public void testGetProgrammingInterest() {
    String expectedInterest = "High";
    String actualInterest = major.getProgrammingInterest();
    assertEquals(expectedInterest, actualInterest);
}
```

```java
@Test
public void testGetPoint() {
    int expectedPoint = 0;
    int actualPoint = major.getPoint();
    assertEquals(expectedPoint, actualPoint);
}

@Test
public void testGetJobCategory() {
    String expectedCategory = "IT";
    String actualCategory = major.getJobCategory();
    assertEquals(expectedCategory, actualCategory);
}

@Test
public void testGetAcceptableGPA() {
    float expectedGPA = 3.0f;
    float actualGPA = major.getAcceptableGPA();
    assertEquals(expectedGPA, actualGPA, delta: 0);
}

@Test
public void testGivePoint() {
    int expectedPoint = 1;
    major.givePoint();
    int actualPoint = major.getPoint();
    assertEquals(expectedPoint, actualPoint);
}
}
```

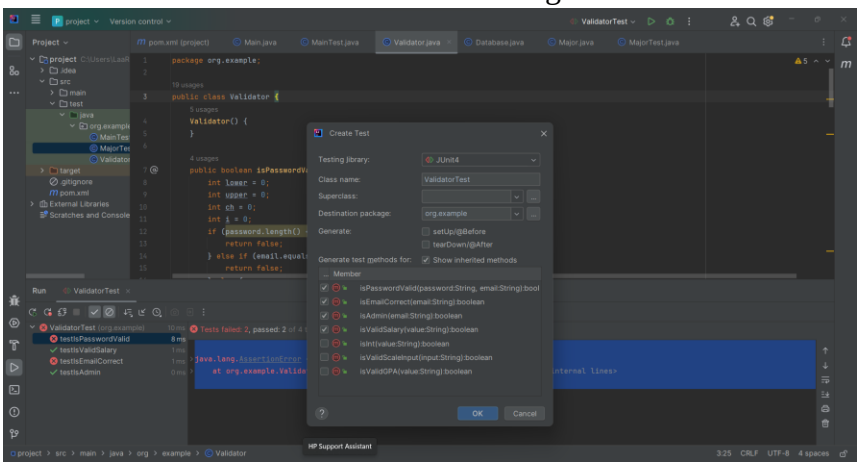
`

The result:

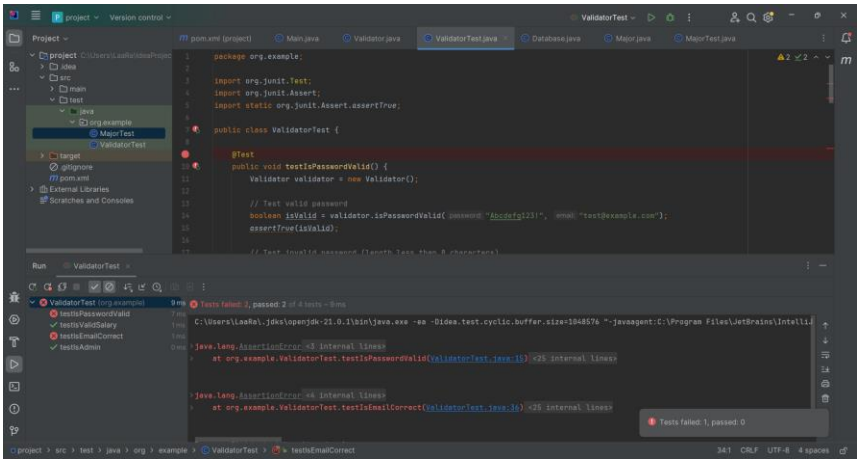


The test case is passed.

in Validator class I chose four methods to generate the test:



The result:



as we see two test is passed and two test is failed.

`

The first method test is testIsPasswordValid_ValidPassword:

```
9       @Test
10      public void testIsPasswordValid_ValidPassword() {
11          String password = "Abc123!@#";
12          String email = "test@example.com";
13          boolean isValid = Validator.isPasswordValid(password, email);
14          assertTrue(isValid);
15      }
```

is failing because the assertion assertTrue(isValid) is not evaluating to true. This means that the isPasswordValid method is returning false for the given inputs, which indicates that the password is not considered valid according to the validation criteria.

The second method test is testIsEmailCorrect:

```
17      @Test
18      public void testIsEmailCorrect() {
19          Validator validator = new Validator();
20
21          // Test valid email
22          boolean isCorrect = validator.isEmailCorrect("test@example.com");
23          assertTrue(isCorrect);
24
25          // Test invalid email (length less than minimum)
26          isCorrect = validator.isEmailCorrect("a@b.c");
27          Assert.assertFalse(isCorrect);
28      }
```

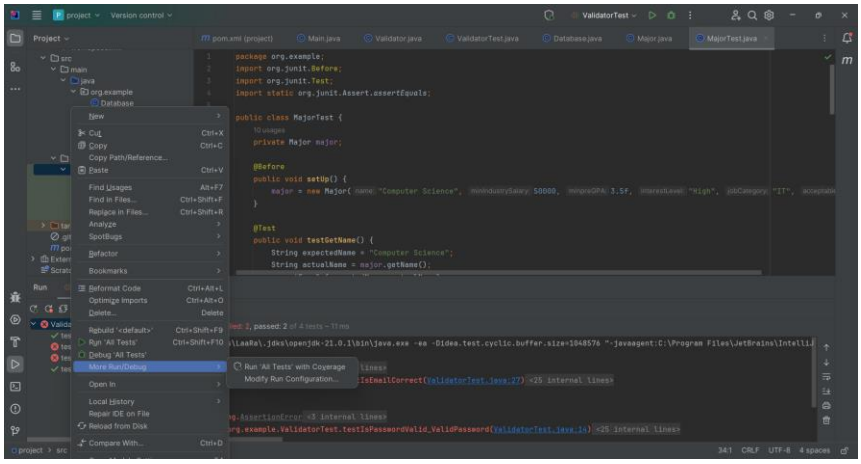
The test method testIsEmailCorrect is failing because the assertion assertTrue(isCorrect) or Assert.assertFalse(isCorrect) is not evaluating to the expected value. This means that the isEmailCorrect method is returning a different result than what is being asserted.

`

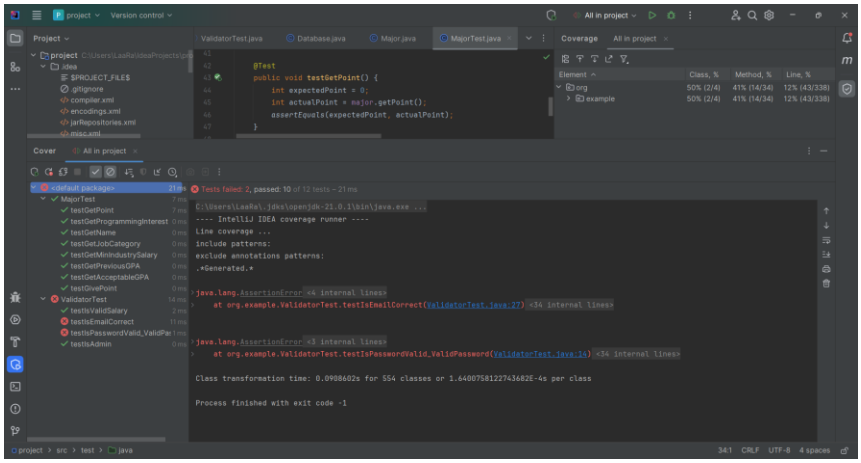
Coverage Report

I use IntelliJ

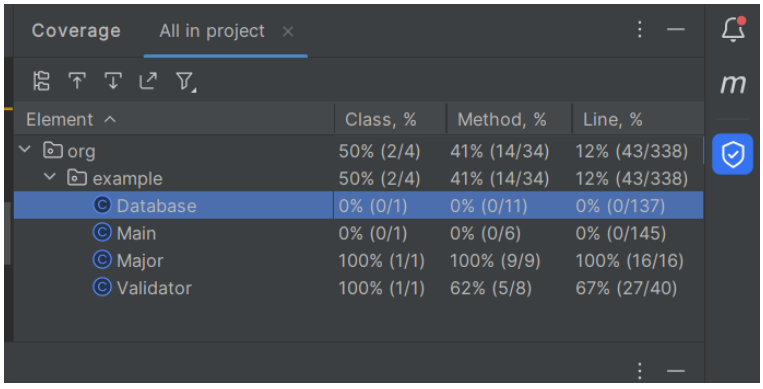
First step I chose run "All Tests" with coverage:



Here as we see all test methods is passed except two methods in validator class:

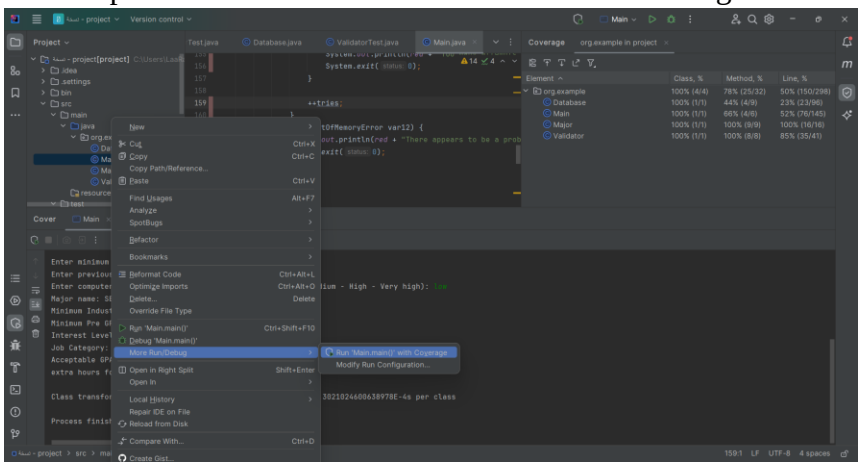


The result:

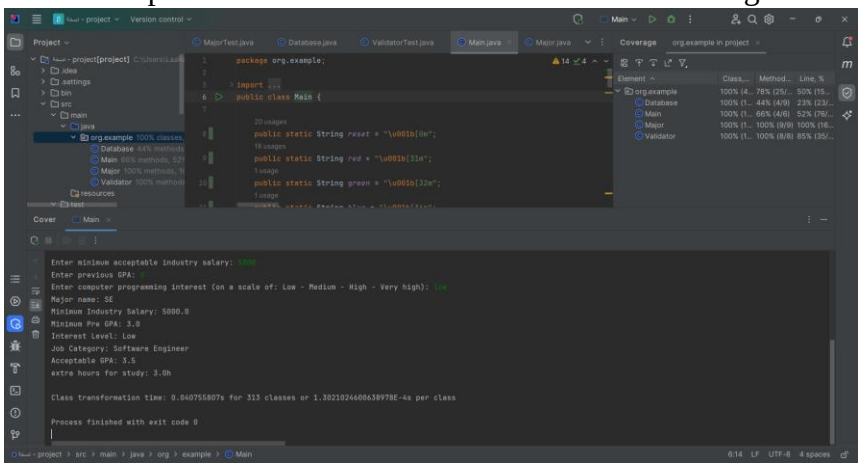


`

Next step I use main class and run the code with coverage:

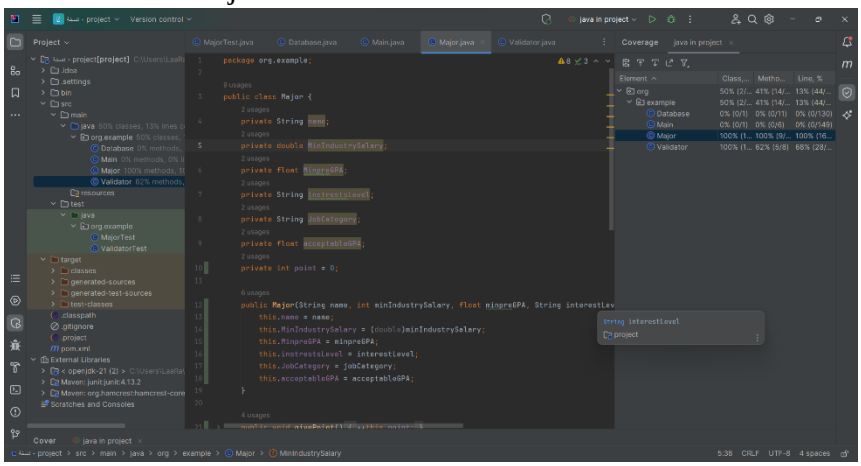


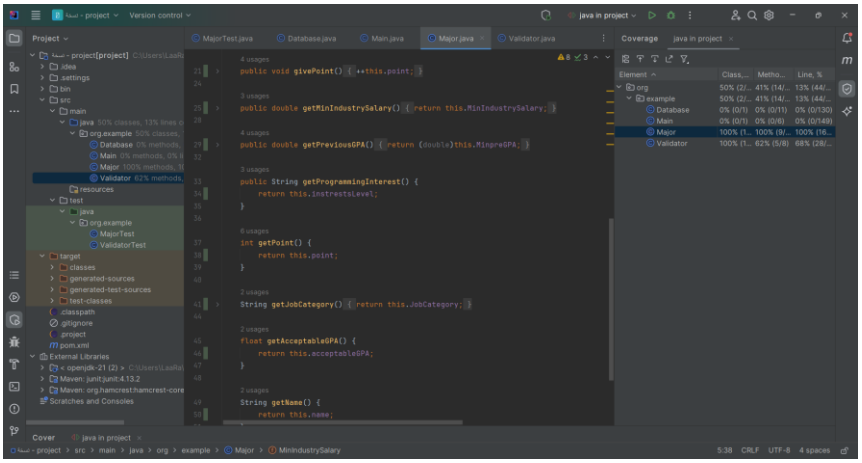
enter the inputs here and then see the results of the coverage:



`

The result for Major class:

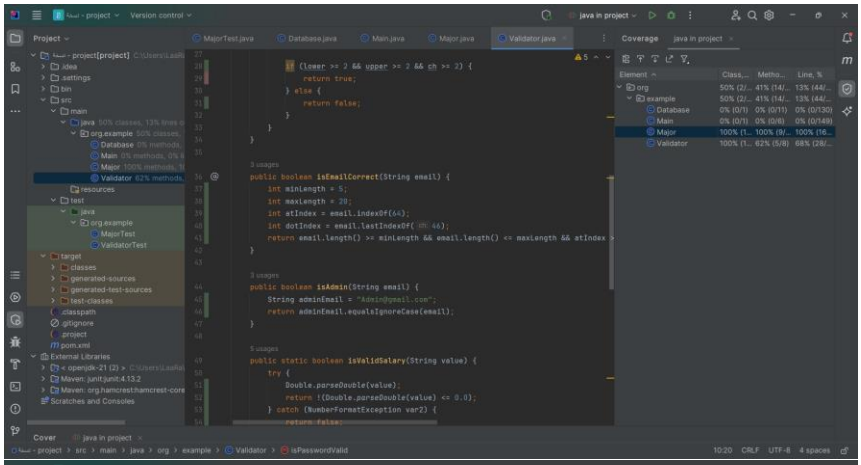


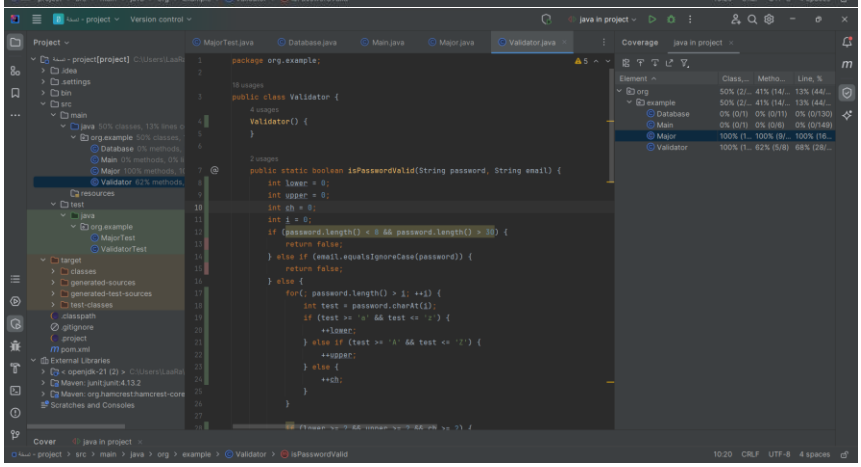


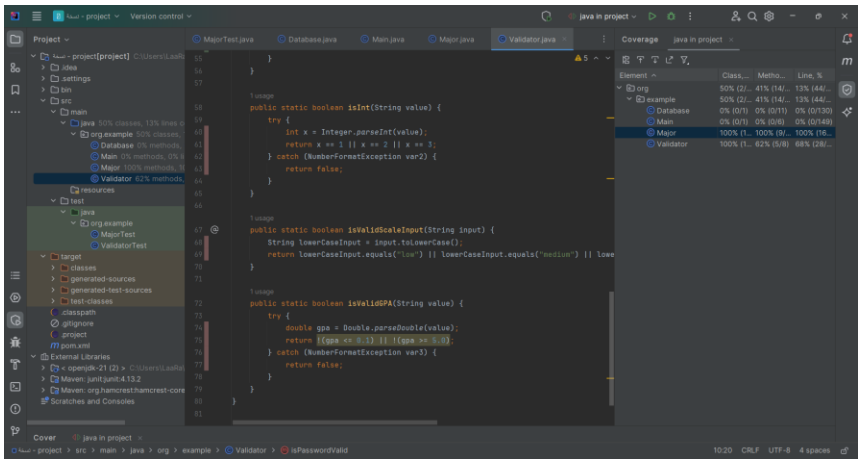
achieving 100% coverage at the class, method, and line levels means that all parts of the Major class code have been tested and executed during testing. This provides a high level of confidence in the correctness and reliability of the software.

`

The result for Validator class:





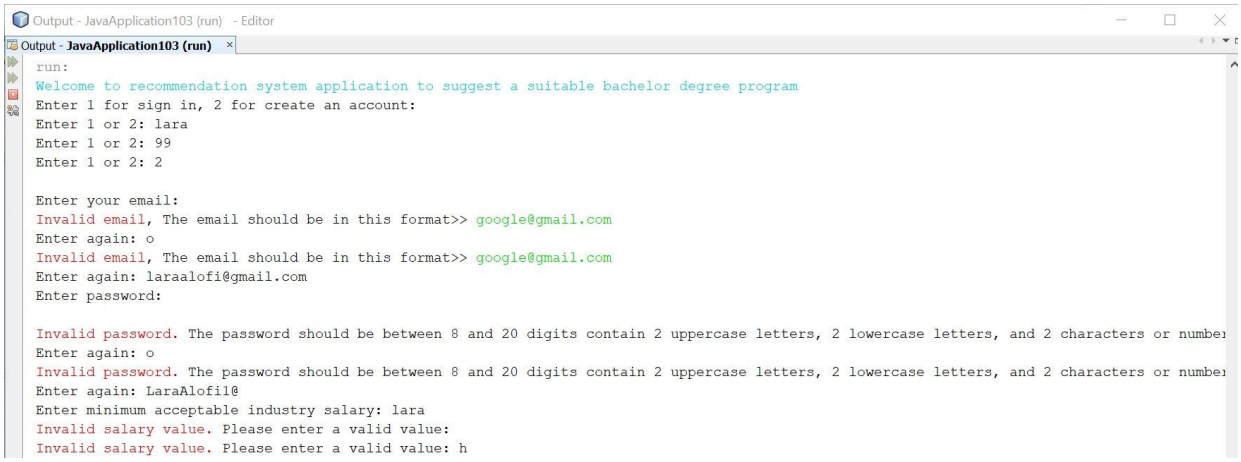


The class has achieved 100% coverage, meaning that all parts of the code within the class have been executed during testing. However, the individual methods and lines of code within those methods have lower coverage rates of 62% and 68% respectively, indicating that some parts of the code have not been thoroughly tested. It's important to focus on increasing coverage in those specific areas to improve confidence in the correctness of the software.

`

Black box testing:

Black box testing is a software testing technique where the tester evaluates the functionality of a system without knowledge of its internal structure. The tester treats the system as a "black box" and focuses on inputs and outputs to validate that the system behaves as expected from an end user's perspective. It is used to uncover functional defects and ensure that the system meets specified requirements. Black box testing can be performed at different levels and is advantageous as it doesn't require knowledge of internal implementation details. However, it may not uncover certain types of defects and is often complemented by other testing techniques.



Test Case 1: Enter an invalid option (whitespace)

- Input: ' '

- Expected Output: Display an error message asking for a valid salary value

- Result: Invalid

Test Case 2: Enter an invalid option -

Input: 'lara'

- Expected Output: Display an error message asking for a valid input

- Result: Invalid

Test Case 3: Enter an invalid option

- Input: 99

- Expected Output: Display an error message asking for a valid input

- Result: Invalid

Test Case 4: Enter an invalid option

-Input: 2

`

-Expected Output: Display an error message asking for a valid input

-Result: Invalid

5. Test Case 5: Enter an invalid email format (whitespace)

Input: ' '

Expected Output: Display an error message asking for a valid email format

Result: Invalid

6. Test Case 6: Enter an invalid email format

input: 'o'

Expected Output: Display an error message asking for a valid email format

Result: Invalid

7. Test Case 7: Enter a valid email address

Input: 'laraalofi@gmail.com'

Expected Output: Proceed to the password entry step

Result: Pass

Test Case 8: Enter an invalid password

Input: " "

Expected Output: Display an error message indicating the password requirements

Result: Invalid

Test Case 9: Enter an invalid password

input: 'o'

Expected Output: Display an error message indicating the password requirements

Result: Invalid

Test Case 10: Enter a valid password

- Input: 'LaraAlofi1@'

- Expected Output: Proceed to the minimum acceptable industry salary entry step

- Result: Pass

`

Test Case 11: Enter an invalid salary value

Input: 'lara'

- Expected Output: Display an error message asking for a valid salary value

Result: Invalid

Test Case 12: Enter an invalid salary value (whitespace)

Input: ' '

Expected Output: Display an error message asking for a valid salary value

Result: Invalid

Test Case 13: Enter a valid salary value

Input: 'h'

Expected Output: Display an error message asking for a valid salary value

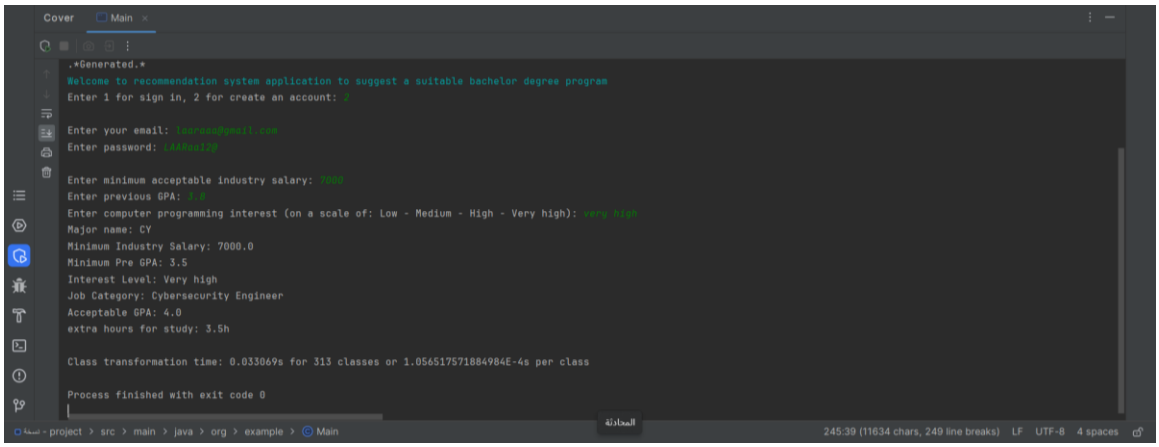
Result: Invalid

`

Test case 14: As a user, I want to test the overall functionality of the code without considering its internal implementation.

Input:

- I choose to create an account by entering '2'
- I set the minimum acceptable industry salary to '7000'
- I set my previous GPA to '3.8'
- I indicate my computer programming interest as 'Very high'



Explanation:

- I choose to create an account and proceed to enter my preferred criteria for a bachelor degree program.
- I set the minimum acceptable industry salary to '7000' because I'm looking for a program that offers a minimum salary of 7000 or higher.
- My previous GPA is '3.8', indicating that I want a program that accepts students with a GPA of 3.8 or higher.
- I specify my computer programming interest as 'Very high' to highlight my intense passion for computer programming.

Expected Output:

- After creating my account, I expect the program to recommend a suitable bachelor degree program that matches my criteria.
- The program should display the details of the recommended program(s), including the major name, minimum industry salary, minimum previous GPA, interest level, and job category.

`

White box testing:

White box testing is a software testing technique that focuses on examining the internal structure, code paths, and logic of a system. Testers with knowledge of the internal implementation design test cases to ensure that all components and interactions are thoroughly tested. White box testing techniques include statement coverage, branch coverage, path coverage, and condition coverage.

Teset cases in the `Validator` class:

1. isPasswordValid(String password, String email)`:

```
2 usages
public static boolean isPasswordValid(String password, String email) {
    int lower = 0;
    int upper = 0;
    int ch = 0;
    int i = 0;
    if (password.length() < 8 && password.length() > 30) {
        return false;
    } else if (email.equalsIgnoreCase(password)) {
        return false;
    } else {
        for(; password.length() > i; ++i) {
            int test = password.charAt(i);
            if (test >= 'a' && test <= 'z') {
                ++lower;
            } else if (test >= 'A' && test <= 'Z') {
                ++upper;
            } else {
                ++ch;
            }
        }

        if (lower >= 2 && upper >= 2 && ch >= 2) {
            return true;
        } else {
            return false;
        }
    }
}
```

This method checks if a password is valid based on specific criteria. The password must be between 8 and 30 characters long. It should not be the same as the email address provided. Additionally, the password must contain at least two lowercase letters, two uppercase letters, and two other characters (such as digits or symbols).

2. isEmailCorrect(String email):

```
3 usages
public boolean isEmailCorrect(String email) {
    int minLength = 5;
    int maxLength = 20;
    int atIndex = email.indexOf(64);
    int dotIndex = email.lastIndexOf( ch: 46);
    return email.length() >= minLength && email.length() <= maxLength && atIndex > 0 && dotIndex > atIndex && dotIndex < email.length() - 1;
}
```

This method verifies if an email address is correct based on certain criteria. The email address should be between 5 and 20 characters long. It must contain the '@' symbol with at least one character before it, and it should have at least one '.' character after the '@' symbol.

`

3. isAdmin(String email):

```
3 usages
public boolean isAdmin(String email) {
    String adminEmail = "Admin@gmail.com";
    return adminEmail.equalsIgnoreCase(email);
}
```

This method checks if an email address belongs to an admin. It compares the provided email address with a predefined admin email address ("Admin@gmail.com") in a case-insensitive manner. If the email matches the admin email, it returns `true`; otherwise, it returns `false`.

4. isValidSalary(String value):

```
5 usages
public static boolean isValidSalary(String value) {
    try {
        Double.parseDouble(value);
        return !(Double.parseDouble(value) <= 0.0);
    } catch (NumberFormatException var2) {
        return false;
    }
}
```

This method determines if a value represents a valid salary. It tries to convert the value into a number and checks if the resulting number is greater than zero. If the conversion is successful and the number is positive, it returns `true`; otherwise, it returns `false`.

5. isInt(String value):

```
1 usage
public static boolean isInt(String value) {
    try {
        int x = Integer.parseInt(value);
        return x == 1 || x == 2 || x == 3;
    } catch (NumberFormatException var2) {
        return false;
    }
}
```

This method checks if a value represents a valid integer. It attempts to convert the value into an integer and checks if the resulting number is either 1, 2, or 3. If the conversion is successful and the number matches one of the specified values, it returns `true`; otherwise, it returns `false`.

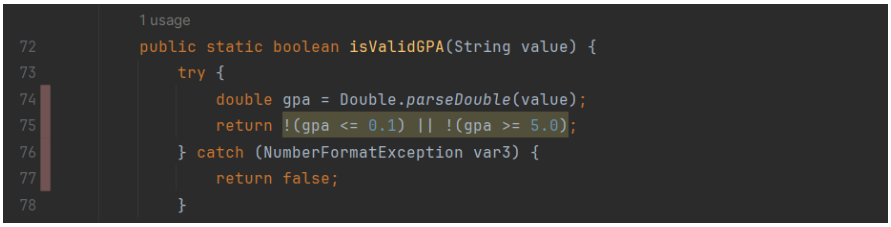
6. `isValidScaleInput(String input):

```
1 usage
public static boolean isValidScaleInput(String input) {
    String lowerCaseInput = input.toLowerCase();
    return lowerCaseInput.equals("low") || lowerCaseInput.equals("medium") || lowerCaseInput.equals("high") || lowerCaseInput.equals("very high");
}
```

This method verifies if an input represents a valid scale input. It converts the input to lowercase and checks if it matches one of the predefined valid options: "low", "medium", "high", or "very high". If the input matches any of these options, it returns `true`; otherwise, it returns `false`.

`

7. isValidGPA(String value):



This method checks if a value represents a valid GPA (Grade Point Average). It attempts to convert the value into a decimal number and verifies if it falls within a specific range. The GPA should be greater than 0.1 and less than 5.0 to be considered valid. If the conversion is successful and the GPA falls within the specified range, it returns `true`; otherwise, it returns `false`.

White Box Testing:

Input:

- I choose to sign in by entering '1'
- I set the minimum acceptable industry salary to '5000'
- I set my previous GPA to '3.2'
- I indicate my computer programming interest as 'High'

Explanation:

- I choose to sign in and proceed to enter my preferred criteria for a bachelor degree program.
- I set the minimum acceptable industry salary to '5000' because I'm looking for a program that offers a minimum salary of 5000 or higher.
- My previous GPA is '3.2', which means I want a program that accepts students with a GPA of 3.2 or higher.
- I specify my computer programming interest as 'High' to indicate that I have a strong interest in computer programming.

Expected Output:

- I expect the program to recommend a suitable bachelor degree program that matches my criteria.
- The program should display the details of the recommended program(s), including the major name, minimum industry salary, minimum previous GPA, interest level, and job category.