Test Coverage Report

Introduction:

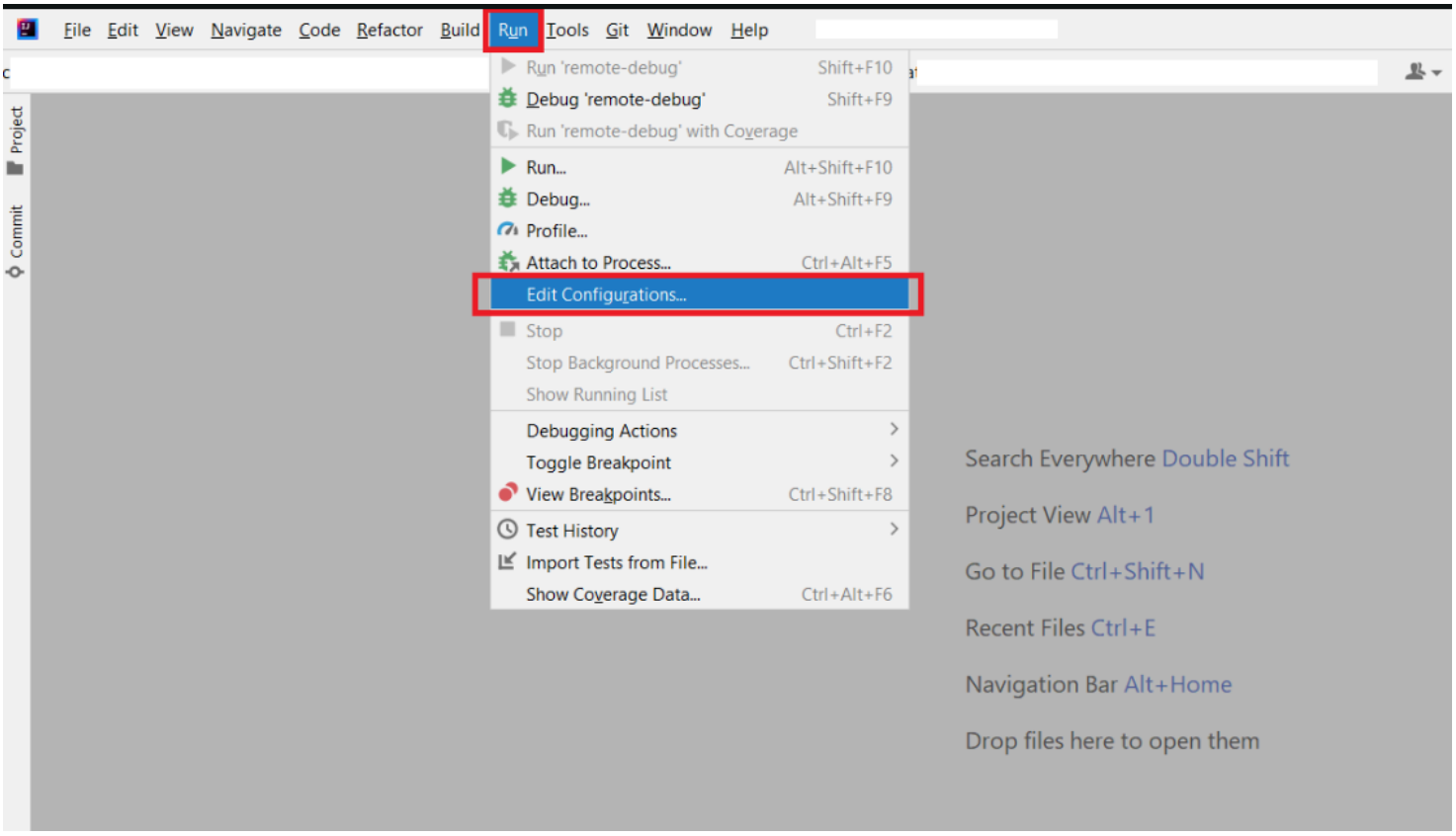
- Test Coverage for the Library Management System is the best way to ensure the validity of all critical aspects of the system through comprehensive analysis, as it helps to identify the tested, as well as the untested parts, by providing a detailed calculation of various elements : classes, methods, lines and branches .
- In this report, we used JaCoCo, an open-source tool compatible with JUnit 5; it offers detailed overview into the code set case coverage within several aspects: classes, methods, lines and branches .
- this report has different purposes :

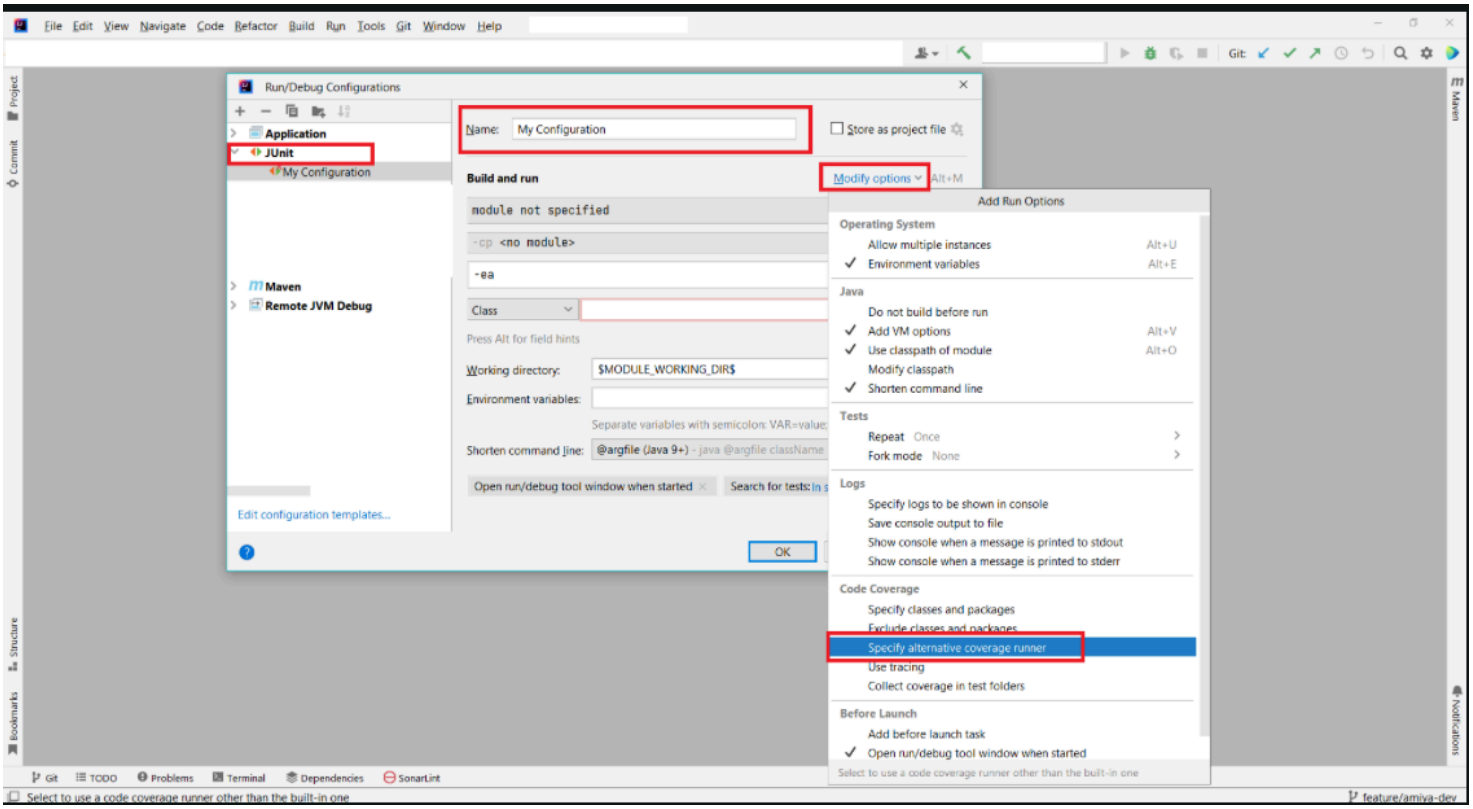
1. It helps testers spot defects early in the system life cycle .
2. It provides a specific calulation of hoe much the code we did cover .
3. It shows the test cases of low to no relevance to our project.

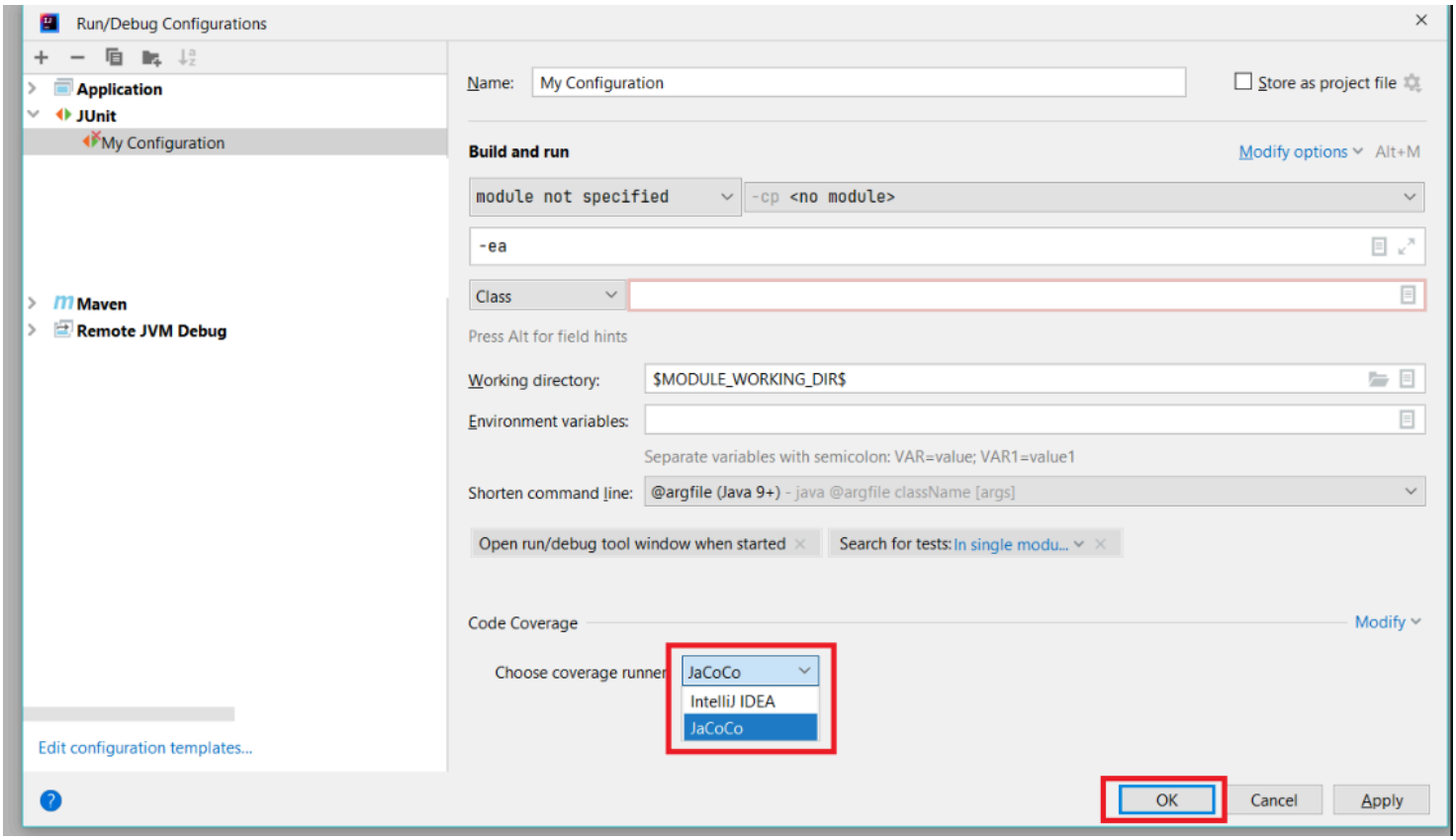
The following section will present the coverage Test cases in a more detailed way; it includes the representation of the Test Coverage, plus a structured explanation of every coverage metric .

Testing Infrastructure:

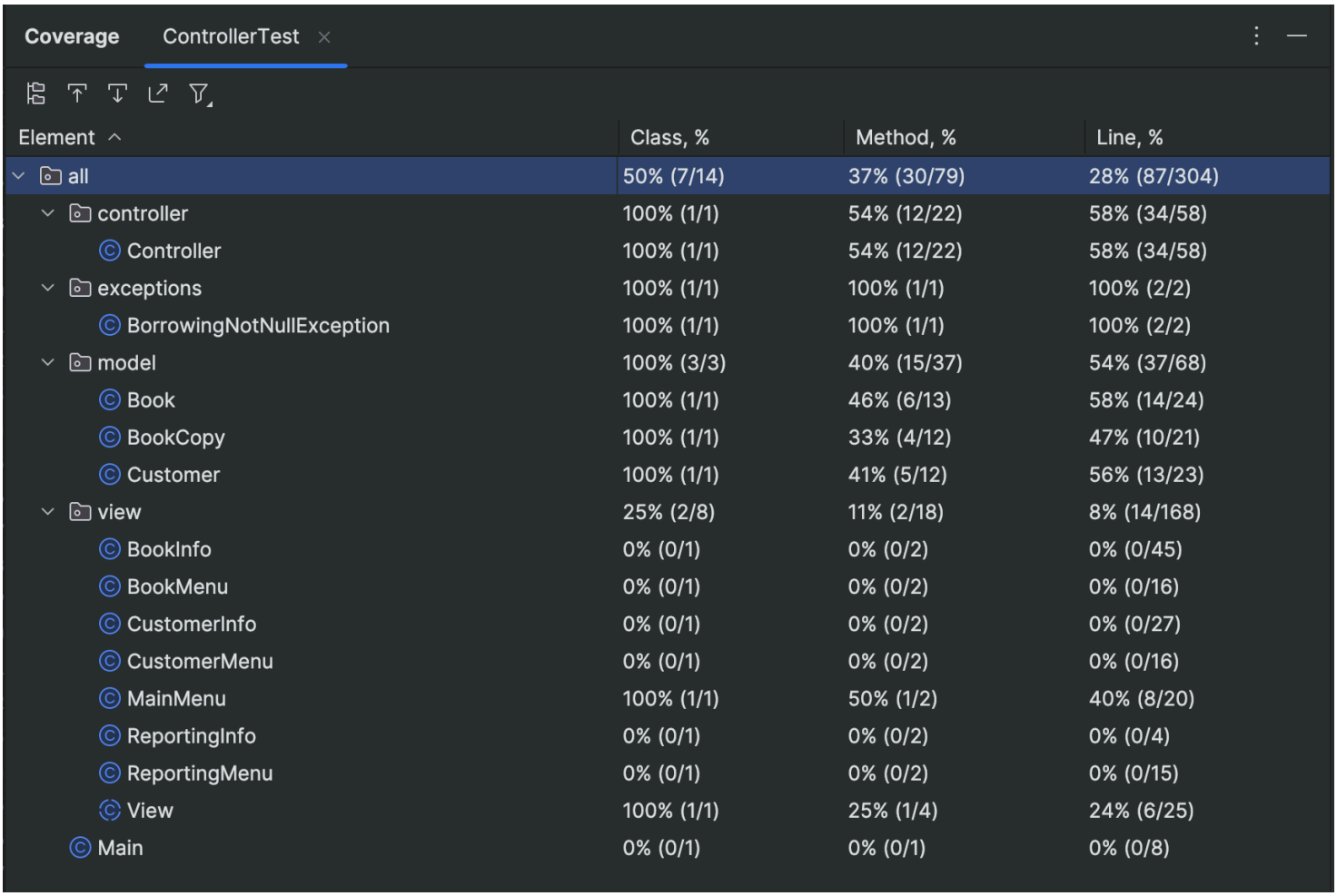
- JUnit 5 is a unit-testing framework in the java ecosystem. It contains a number of features that supports Java 8 and above, as well as many different styles of testing . However, IntelliJ supports JUnit 5 by default(it is simply done by Right Click->Run).
- As you can see in our code, we have used @BeforeEach, @Test, and assertions (assertEquals, assertFalse, assertTrue...) . These annotations are extended models for writing tests in JUnit 5 .
- JaCoCo (Java Code Coverage) is an open-source code coverage library for java . The library is often used to measure the coverage of unit or integration tests .
- Implementation : go to Run>Edit Configuration>modify options> specify alternative coverage runner>select JaCoCo from the Choose coverage runner


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Run 'remote-debug' Shift+F10
Debug 'remote-debug' Shift+F9
Run 'remote-debug' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Profile...
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Stop Ctrl+F2
Stop Background Processes... Ctrl+Shift+F2
Show Running List
Debugging Actions
Toggle Breakpoint
View Breakpoints... Ctrl+Shift+F8
Test History
Import Tests from File...
Show Coverage Data... Ctrl+Alt+F6
Search Everywhere Double Shift
Project View Alt+1
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home
Drop files here to open them


Run/Debug Configurations
Application
JUnit
My Configuration
Maven
Remote JVM Debug
Name: My Configuration
Store as project file
Build and run
Modify options Alt+M
module not specified
-cp <no module>
-ea
Class
Press Alt for field hints
Working directory: $MODULE_WORKING_DIR$
Environment variables:
Separate variables with semicolon: VAR=value;
Shorten command line: @argfile (Java 9+) - java @argfile className
Open run/debug tool window when started
Edit configuration templates...
OK
Add Run Options
Operating System
Allow multiple instances Alt+U
Environment variables Alt+E
Java
Do not build before run
Add VM options Alt+V
Use classpath of module Alt+O
Modify classpath
Shorten command line
Tests
Repeat Once
Fork mode None
Logs
Specify logs to be shown in console
Save console output to file
Show console when a message is printed to stdout
Show console when a message is printed to stderr
Code Coverage
Specify classes and packages
Exclude classes and packages
Specify alternative coverage runner
Use tracing
Collect coverage in test folders
Before Launch
Add before launch task
Open run/debug tool window when started
Select to use a code coverage runner other than the built-in one



Coverage Summary:



Coverage | ControllerTest

Element	Class, %	Method, %	Line, %
all	50% (7/14)	37% (30/79)	28% (87/304)
controller	100% (1/1)	54% (12/22)	58% (34/58)
Controller	100% (1/1)	54% (12/22)	58% (34/58)
exceptions	100% (1/1)	100% (1/1)	100% (2/2)
BorrowingNotNullException	100% (1/1)	100% (1/1)	100% (2/2)
model	100% (3/3)	40% (15/37)	54% (37/68)
Book	100% (1/1)	46% (6/13)	58% (14/24)
BookCopy	100% (1/1)	33% (4/12)	47% (10/21)
Customer	100% (1/1)	41% (5/12)	56% (13/23)
view	25% (2/8)	11% (2/18)	8% (14/168)
BookInfo	0% (0/1)	0% (0/2)	0% (0/45)
BookMenu	0% (0/1)	0% (0/2)	0% (0/16)
CustomerInfo	0% (0/1)	0% (0/2)	0% (0/27)
CustomerMenu	0% (0/1)	0% (0/2)	0% (0/16)
MainMenu	100% (1/1)	50% (1/2)	40% (8/20)
ReportingInfo	0% (0/1)	0% (0/2)	0% (0/4)
ReportingMenu	0% (0/1)	0% (0/2)	0% (0/15)
View	100% (1/1)	25% (1/4)	24% (6/25)
Main	0% (0/1)	0% (0/1)	0% (0/8)

Here is a visual representation of the coverage metrics for each module and class within the Library Management System project .
The columns represent the several test cases related to classes, methods, lines, and branches .

element	Class%	Method%	Line%	Branch%
The **packages** (e.g:view,model, exceptions, controller) the **classes of each package** (e.g:View,MainMenu,PhysicalBook, Controller...)	**Precentage** of **classes** that have been **tested** == number of tested classes / the total number of classes in that package or class	**Percentage** of **methods** that have been **tested** == Number of tested methods / The total number of methods	**Percentage** of **lines** that have been **tested** == Number of tested lines / The total number of lines	**Percentage** of **branches** that have been **tested** == Number of tested branches / The total number of branches

- **Interpretation:**
- *Package : view*

1. View:

- This class is fully covered by tests, but only 25% of methods are covered.

```
7   public abstract class View {
8       private final String PROMPT_TO_EXIT = "Press 'q' to " + (this instanceof MainMenu ? "quit." : "back.");
9       private final String CURSOR = "\n>  ";
10
11      protected Controller controller;
12      protected String name;
13      protected View prev;
14
15      public View(Controller controller, View prev) {
16          this.controller = controller;
17          this.prev = prev;
18      }
19      public abstract void show();
```

2. ReportingMenu+ReportingInfo:
   - The classes, methods, lines are entirely not covered by tests.
3. MainMenu:

- The class is fully covered, the methods are half tested, and only 40% of lines are tested.

```java
public MainMenu(Controller controller) {
    super(controller, prev: null);

    this.name = "Main Menu";

    this.options = new ArrayList<>();
    this.options.add("Book Menu");
    this.options.add("Customer Menu");
    this.options.add("Reporting Menu");
    this.options.add("Settings");
}

@Override
public void show() {
    System.out.println("Welcome to the Library Management System!\nPlease type a number or press enter.");
    String input = super.promptMenu(this.options);

    switch (input.charAt(0)) {
        case '0':
            controller.setMenu(new BookMenu(controller, prev: this));
            break;
        case '1':
            controller.setMenu(new CustomerMenu(controller, prev: this));
            break;
        case '2':
            controller.setMenu(new ReportingMenu(controller, prev: this));
            break;
        case '3':
            super.promptAndExit( s: "A setting menu will be displayed");
            this.show();
        default:
            break;
    }
}
```

- *Package: model*
- The classes, as well as the branches ares fully covered, however, averagely, 40% of the methods and 54% of the lines are covered in total.
- *Package: Controller*
- The class Controller was fully tested, 54% of methods were successfully executed, 58% of lines were tested.

——> we can say that these different percentages, concerning the Test Case Coverage are due to the use of classes when it comes to testing :

For example , in the controllerTest :

- **the Controller class:** it was fully covered and as methods: getBooks() / getPhysicalBooks() / getCustomers() / deleteBook() / getId() / addBook() , however searchBook(), searchCustomer() were not used while testing, that is why the methods and lines of the Controller class were not fully covered.
- **Book class:** the class was fully covered because it was tested and used in so many other classes such as the Controller, PhysicalBook, ControllerTest, but not all the methods were used, we can see that the constructor was primarily used .

- **Customer class:** it is also fully tested, but we only used as methods: the constructor, plus the getBorrowedList(), and this is what explains the 41%of methods tested .
- **PhysicalBook class:** 100% test covered as class, but only 41% methods tested because only the constructor, the getters and setBorrower() were used in the testing process.

Conclusion

The test Coverage Report for the Library Management System provides a structured analysis of the tests covering the code . These results highlight several areas that require some improvements to ensure an effective, yet efficient system performance .