Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 01/01/2021 | 1.0 | Started and finished the report. | Abdullah Saydemir Aleyna Ölmezcan  Burcu Arslan  Özge Yılgür  Esad Simitçioğlu  Emin Sadikhov |

**TABLE OF CONTENTS**

Revision History 1

1 Introduction 3

1.1 Document overview 3

2 Static Code Analysis 4

2.1 Tools 4

2.2 Results and Discussion 4

3 Dependency Analysis 5

3.1 Tools 5

3.2 Results and Discussion 5

4 Test Coverage Analysis 7

4.1 Tools 7

4.2 Results and Discussion 7

# Introduction

## Document overview

This document presents and interprets the code analysis results regarding the OzU-Garage software development project. Code is analyzed by 3 different tools: 1) Static Code Analysis tool 2) Dependency Analysis tool and 3) Test Coverage tool. The first tool is used to reveal potential bugs that might be overseen during the testing process. The second tool is employed for evaluating the design quality based on the amount of coupling among the software modules and to what extent the code reflects the originally envisioned design. The last tool is used for measuring the coverage of unit tests in the project. Each section below is dedicated to each of these 3 analysis.

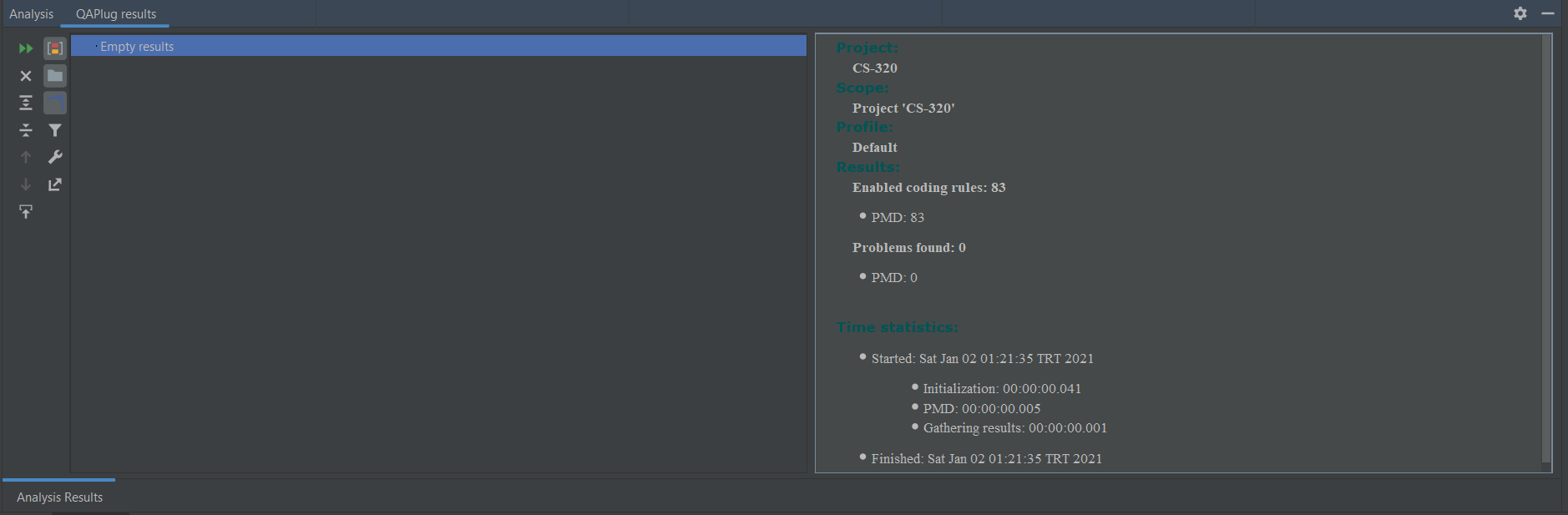
# Static Code Analysis

## Tools

We used PMD v6.21.0 (with QAPlug plugin) to do the analysis. It looks for bugs in Java code within Intellij IDEA.

## Results and Discussion

The QAPlug-PMD plugin did not find any bugs in the code.



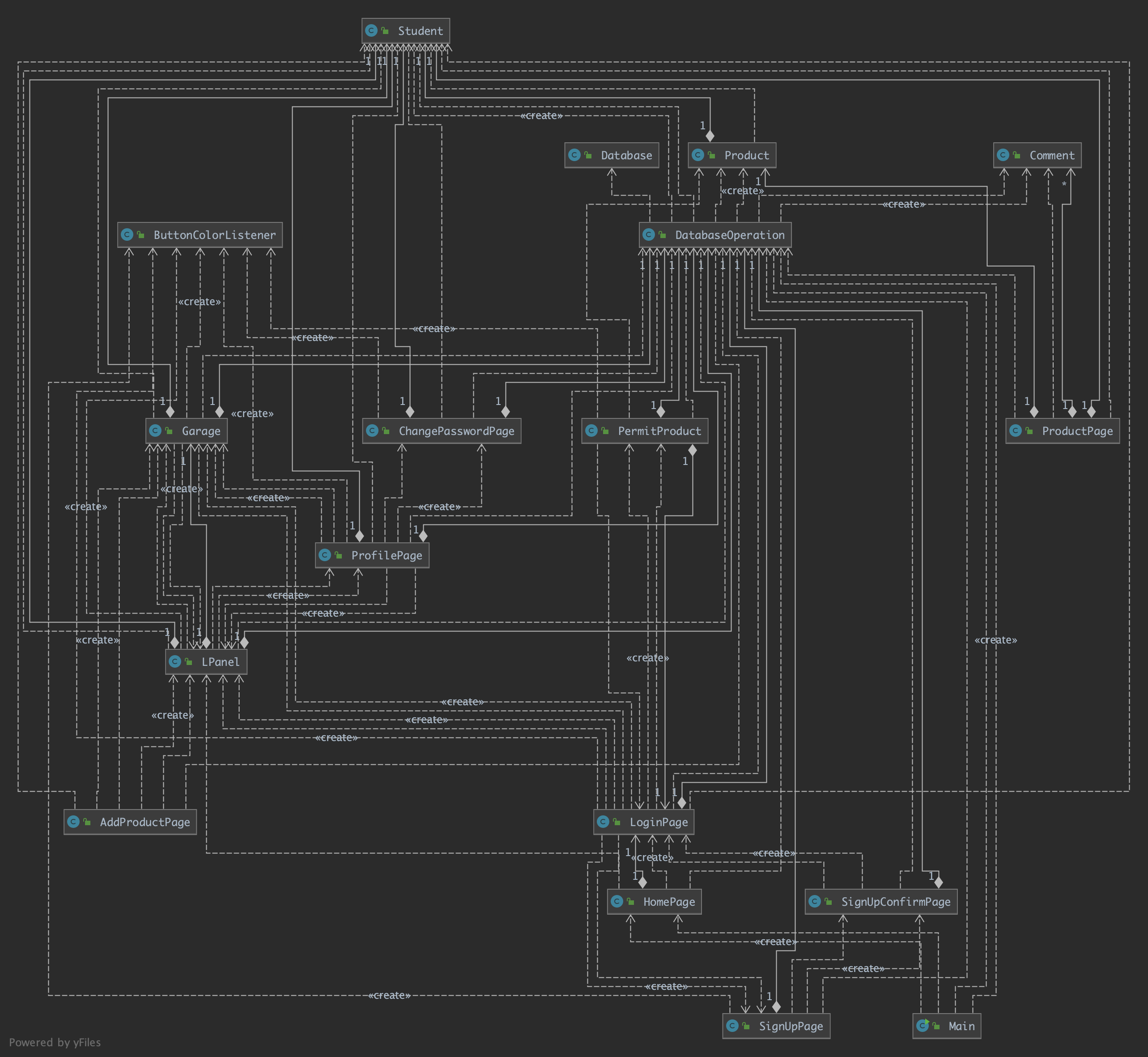
# Dependency Analysis

## Tools

IntelliJ IDEA Dependency Diagram tool is used to do dependency analysis in our project.

## Results and Discussion

Dependency Graph for OzU-Garage is as follows



The extracted dependency graph is presented above. Calculated set of metrics based on this graph is listed below:

# of Edges = 51

# of Nodes = 18

Edge-to-node Ratio = 51 / 18 = 2.83

Tree Impurity = 2 \* (51 – 18 + 1) / (18 – 1) \* (18 – 2) = 68 / 272 = 0.25

We have seen that the tree impurity turned out to be relatively small. That can be sign of a good design structure. Edge-to-node ratio is high because in our design model GUI and Controller classes are not in different packages. In contrast, we can see that our DatabaseOperation class is highly coupled as most of the edges associated with this module. This module can be refactored and separated into different packages to reduce dense coupling at this single module.

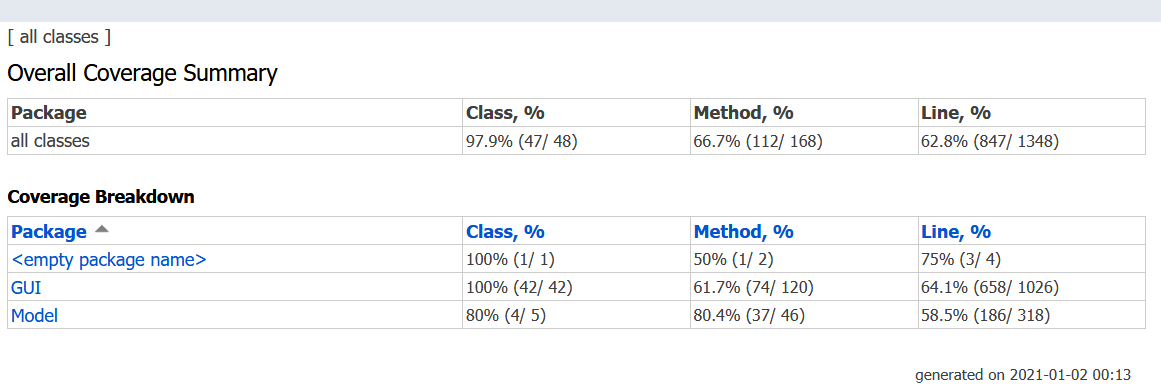
# Test Coverage Analysis

## Tools

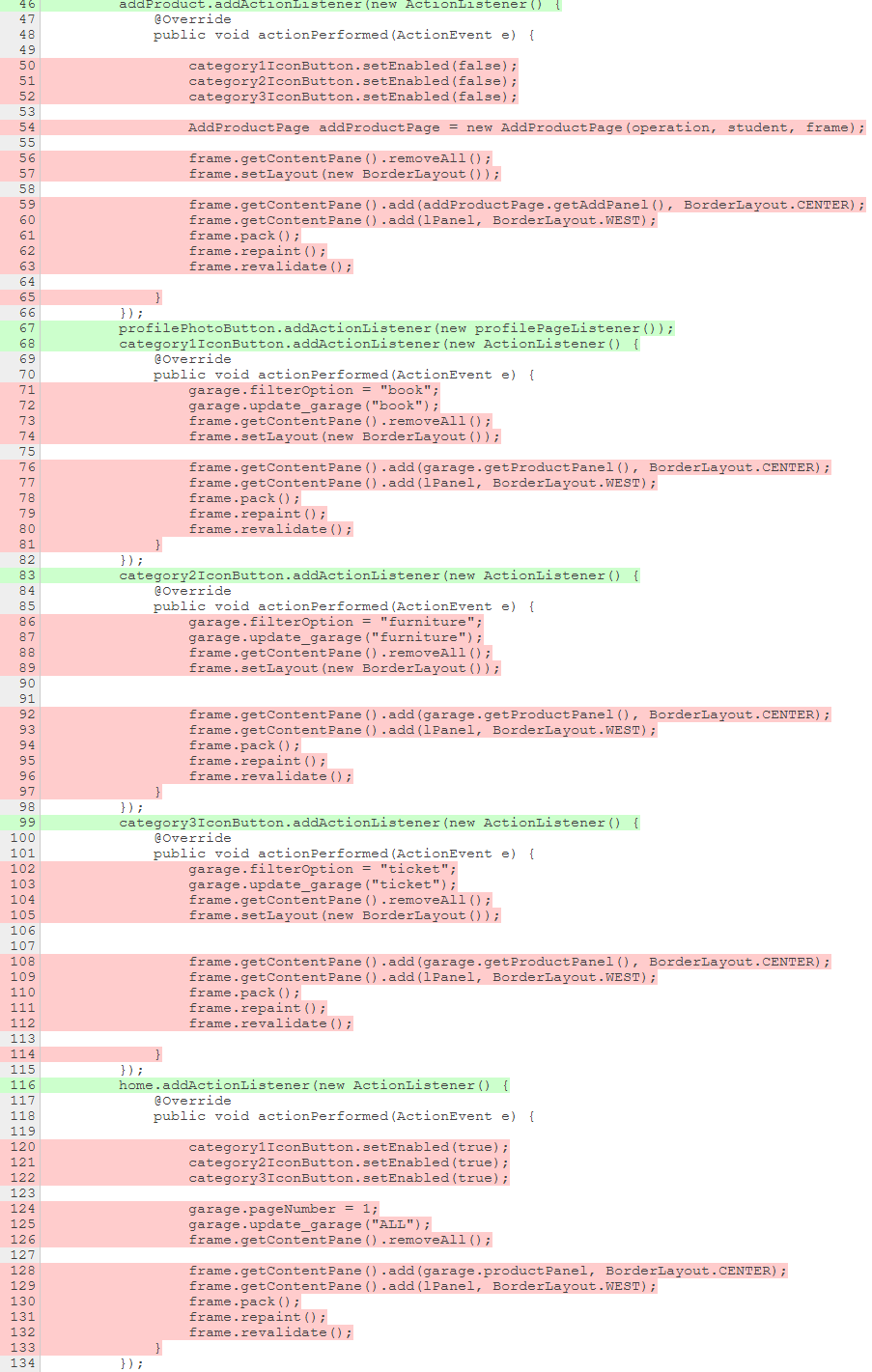
IntelliJ IDEA’s Test Coverage built-in tool is used for test coverage analysis.

## Results and Discussion

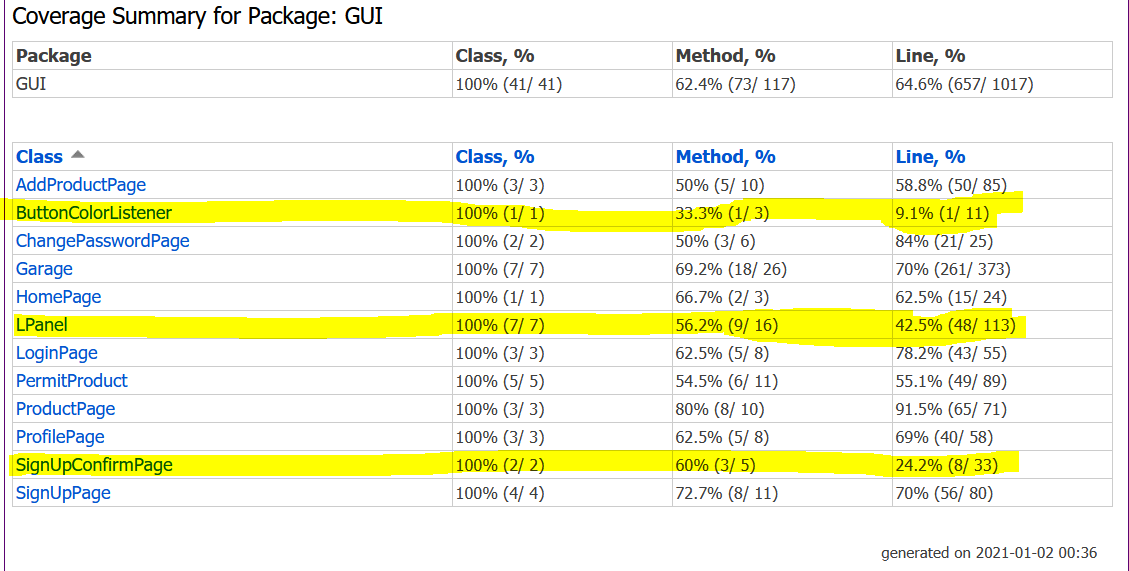
Below is the overall test coverage report. The inspection shows that with the test close to 65% of the lines, 65% of methods and most of the classes are covered.



The test does not (and cannot) cover ActionListeners that are triggered by user interaction with the interface. Since there is no GUI to test these, methods are not covered which can be seen in the example below. LPanel is instantiated most of the time since it must be available on all the pages. However, the listeners -which are the part of the constructor- are not used, decreasing the coverage rate.



This is also the case for ButtonColorListener and SignUpConfirmPage since they are instantiated but neither the class nor the methods inside are not used.



Also, some exception handling procedures need to be carried out frequently since the project depends on database commands (SQLException) and user input / output (IOException), which in turn decreases the coverage result.

