REVISION HISTORY

Date	Version	Description	Author
11.05.2025	0.1	Dependency Analysis section (Section 3) has been completed.	Emre Tekin
11.05.2025	0.2	Static Code Analysis section (Section 2) has been completed.	Emre Tekin, Tahsin Karcı
11.05.2025	0.3	Tools for Test Coverage Analysis (Section 4.1) have been determined and written.	İris Akdemir, İrem Akova
11.05.2025	0.4	Corrected the description of the tool used in Section 4.1	Elifnur Boncuk
11.05.2025	0.5	Results and Discussion Part (Section 4.2) has been completed.	Mert Turan
11.05.2025	1.0	The document has been transferred from Google Docs to Word. The last touch has been made to the document.	Tahsin Karcı

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 **7**
3.1 Tools *7*
3.2 Results and Discussion *7*
4 Test Coverage Analysis **9**
4.1 Tools *9*
4.2 Results and Discussion *9*

1 Introduction

1.1 Document overview

This document presents and interprets the code analysis results regarding the RMS 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 analyses.

2 Static Code Analysis

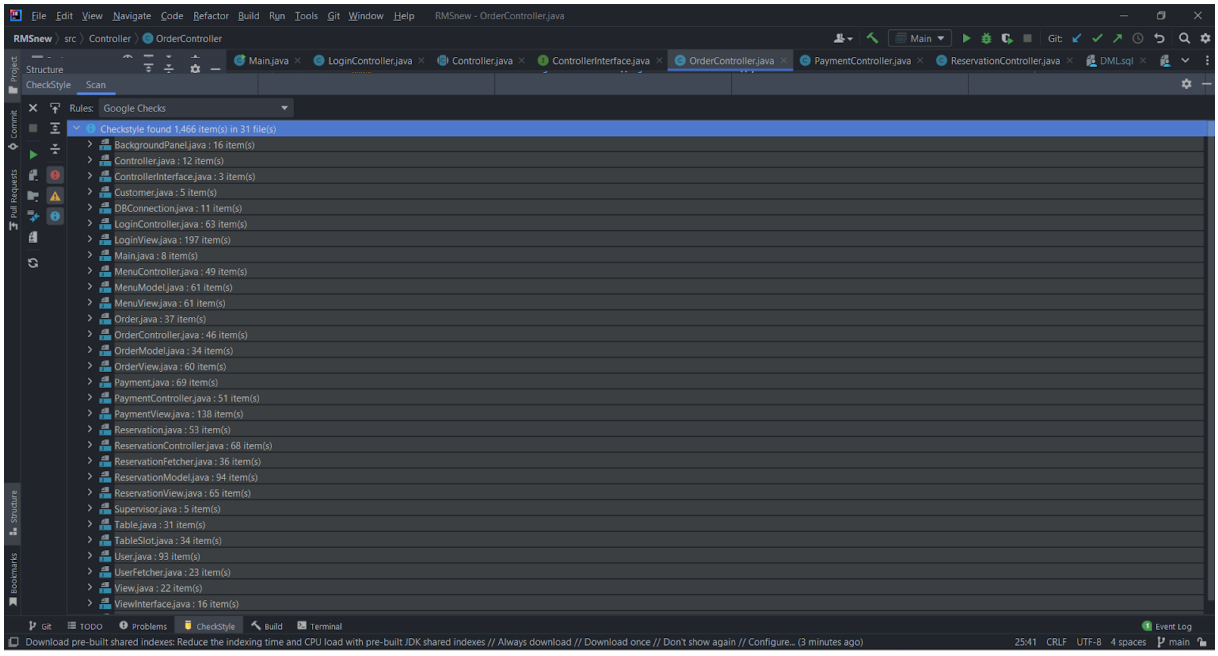
2.1 Tools

Checkstyle is a static code analysis tool for Java source code. Checks code against established code standards like Sun Checks and Google Checks and style guides. This tool is also available as a plugin integrated with IntelliJ IDEA.

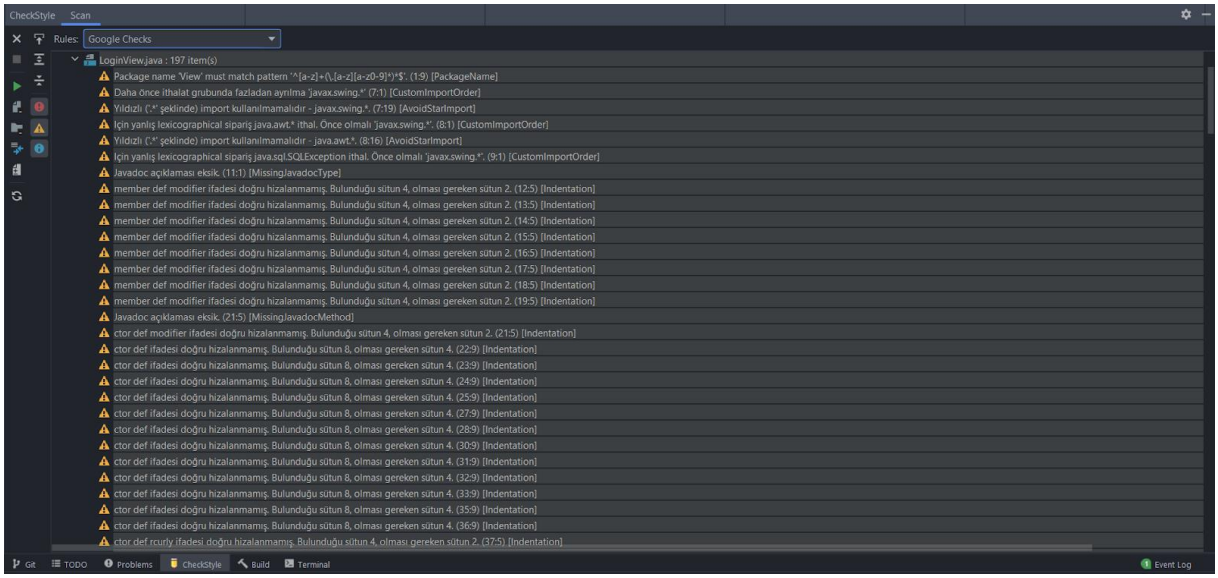
- CheckStyle, https://checkstyle.sourceforge.io/

2.2 Results and Discussion

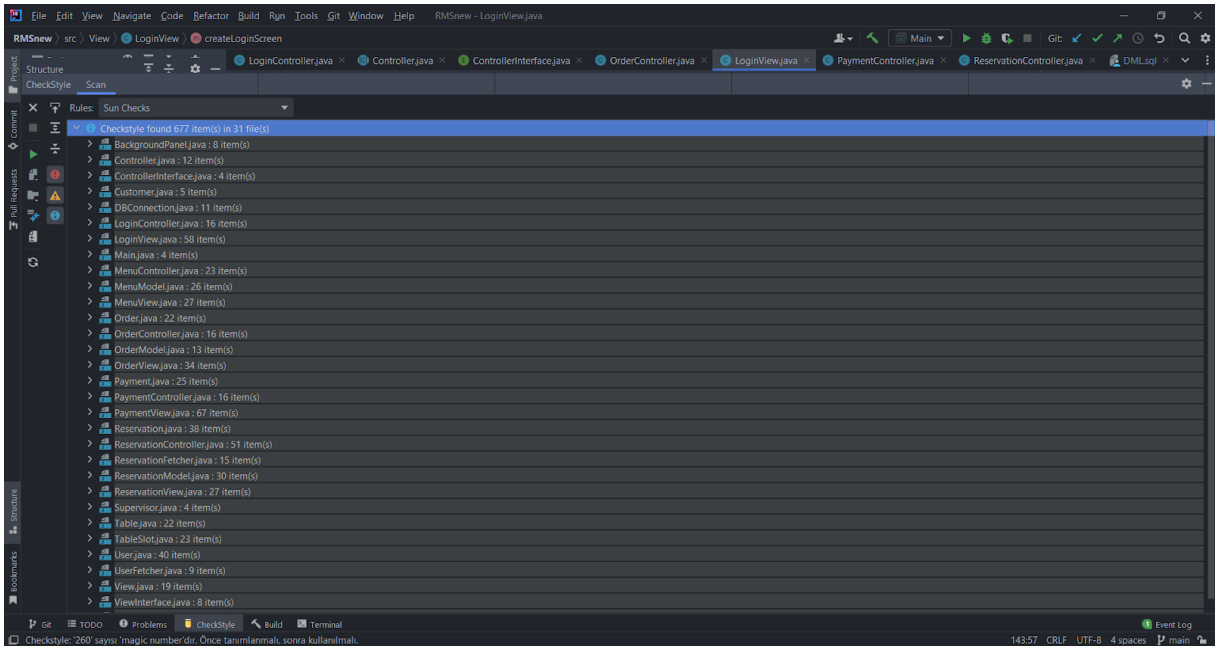
In the CheckStyle analysis, we compared our code against both of the built-in code standards.
Google Checks analysis has returned a total of 1466 suggestions. None of these were determined as critical; however, a lot of alignment, line length and import order warnings were noted. Such style and format issues can significantly affect the readability and maintainability of the code, potentially affecting team collaboration.



As can be seen in the screenshot above, the class with the most warnings is `LoginView` class. Below, a part of the warnings of `LoginView` class can be seen.



Also, we compared our code against **Sun Checks** configuration too. Sun Checks analysis has returned a total of 677 suggestions. Again, none of these were determined as critical. Below, output of the Sun Checks analysis can be found.



Analysis of alerts that are reported by the tool (Google Checks):

Type	Count of Instances	Severity	Count of Real Bugs
Critical Error	0	High	0
Missing Javadoc	89	High	89
Import Corrections	112	Low	112
Styling Suggestions	1024	Low	1024

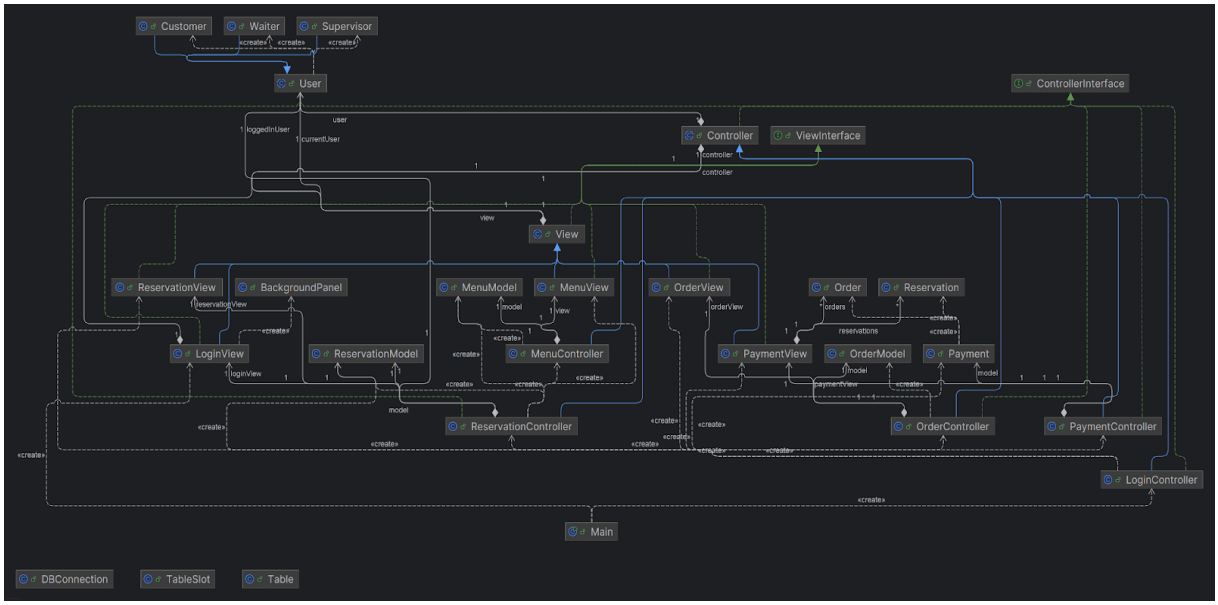
Note: In this table, only the most common ones are included.

3 Dependency Analysis

3.1 Tools

Intellij Ultimate Dependency Diagram Tool has been used in this section.

3.2 Results and Discussion



The extracted dependency graph is presented above. We calculated a set of metrics based on this graph as listed below:

# of Edges = 63
# of Nodes = 29
Edge-to-node Ratio = 63 / 29 = 2.17
Tree Impurity = 2 * (63 – 29 + 1) / (29 – 1) * (29 – 2) = 70 / 756 = 0.09

The dependency graph shows a moderately complex structure with 29 nodes and 63 connections, and a low density of 0.08. The system is not fully connected, meaning some helper classes or parts don't directly interact with others, which is okay in a layered or modular design. However, the fact that there are repeated connections and cycles means that some parts might depend on each other in a loop, which can make the system harder to test and maintain. The MVC pattern is clearly used, with separate roles for Model, View, and Controller. Notably, the most tightly connected classes are `LoginController`, `ReservationController`, and `PaymentController`, each interacting heavily with both the `View` and `Model` layers. These connections, such as `LoginController`'s calls to `view.updateView()` and creation of new `Controller` instances, or `PaymentController` calling `model.takePayment()` and `model.closeReservation()` while coordinating `PaymentView` events, increase coupling and reduce modularity. Since the graph is not tree-like and has overlapping parts, these interwoven connections make components more tightly coupled than ideal. This can make it

harder to expand or update the system later. Making clearer separations between parts—especially reducing how much controllers directly handle view or model logic—and minimizing redundant dependencies would help make the system easier to manage and improve.

4 Test Coverage Analysis

4.1 Tools

This test coverage analysis used *IntelliJ IDEA's built-in code coverage tool.* This tool allows developers to measure which parts of the application code are executed during runtime or while unit tests are being run.

IntelliJ IDEA highlights covered (executed) and uncovered (skipped) lines directly in the source code editor using color-coded indicators (green for covered, red for uncovered, and yellow for partially covered). This makes it easy to visually inspect and evaluate the completeness of the test.

Since the tool is integrated into IntelliJ IDEA by default, no external plugins or installations are required. The coverage analysis was conducted by running the Main.java class using the "Run with Coverage" option and manually interacting with the graphical user interface (GUI) to simulate real user workflows in different roles in the system (Customer, Waiter, Supervisor).

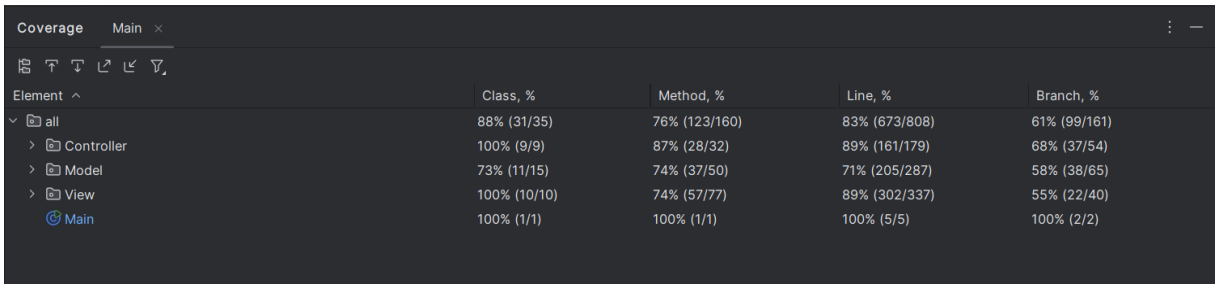
4.2 Results and Discussion

This is the code we run in the main to observe the coverage of the application.

```
public class Main {  Emre +1 *
    public static void main(String[] args) {  Emre *
        for (int i = 0; i < 3; i++) {
            LoginView loginView = new LoginView();
            LoginController controller = new LoginController(loginView);
            loginView.setController(controller);

            loginView.show();
        }
    }
}
```

Here is the overall coverage summary of our application:



Here is the coverage of the Model component which has the lowest coverage with 73% class coverage.

Model	73% (11/15)	74% (37/50)	71% (205/287)	58% (38/65)
Customer	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
DBConnection	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
MenuModel	100% (1/1)	100% (4/4)	78% (26/33)	66% (4/6)
Order	100% (1/1)	100% (5/5)	100% (9/9)	100% (0/0)
OrderModel	100% (1/1)	100% (2/2)	85% (17/20)	75% (3/4)
Payment	100% (1/1)	100% (5/5)	84% (39/46)	70% (7/10)
Reservation	100% (1/1)	100% (8/8)	100% (15/15)	100% (0/0)
ReservationFetcher	0% (0/1)	0% (0/1)	0% (0/19)	0% (0/10)
ReservationModel	100% (1/1)	100% (4/4)	84% (49/58)	62% (10/16)
Supervisor	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
Table	0% (0/1)	0% (0/5)	0% (0/9)	100% (0/0)
TableSlot	0% (0/1)	0% (0/5)	0% (0/9)	100% (0/0)
User	100% (1/1)	83% (5/6)	88% (46/52)	82% (14/17)
UserFetcher	0% (0/1)	0% (0/1)	0% (0/13)	0% (0/2)
Waiter	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)

ReservationFetcher and UserFetcher are not covered as they were written with the purpose of testing whether the connection with the database functions correctly. After the testing, we didn't delete those functions as they don't affect the code.

Table and TableSlot are also not covered as we never used them in the code. They were mainly written as a part of the project but as we advanced in the code, we never needed them again.

Below, there is an example of a code snippet that shows that not all the lines marked with red are incorrect codes. Some of them are not covered because of the exceptions.

```
89          } catch (Exception e) {
90              e.printStackTrace();
91          }
92      }
```

Coverage Analysis

The average coverage of the application is very sufficient with 88% class coverage, 76% method coverage, 83% line coverage and 62% branch coverage. These results show a strong overall coverage, in terms of class, line and method coverage, majority of the application's structure and the logic is being effectively tested. However, relatively low branch coverage may indicate that while most of the code paths are exercised, not all possible ones are observed.