REVISION HISTORY

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 31/12/2023 | 1.0 | General details added and Static code analysis implemented | Arda Onur |
| 02/01/2024 | 1.1 | Test Coverage implemented | Arda Onur, Buğra Bedir, Kaan İşmen |
| 02/01/2024 | 1.2 | Dependency Analysis implemented | Arda Onur, Meriç Okçu, Ömer Özmeteler |

**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 AVEC 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

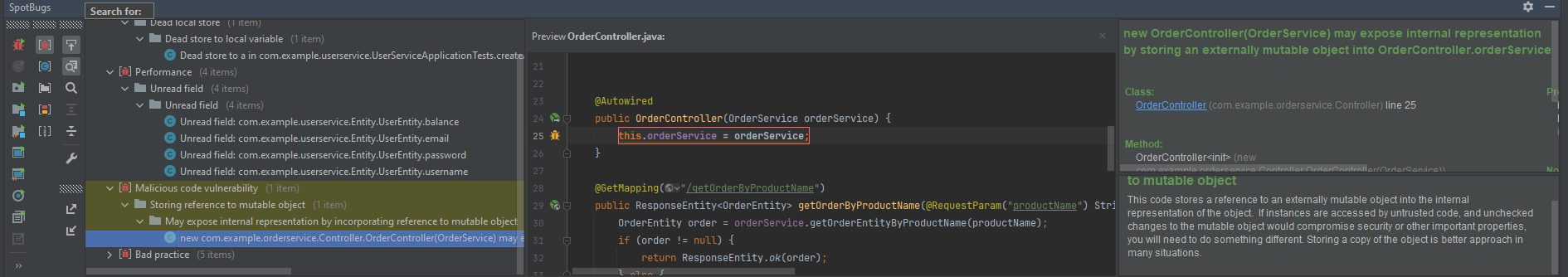
SpotBugs plugin is used to see the results of Static Code Analysis

SpotBugs is a plugin that used to find bugs in java code.

* SpotBugs : https://plugins.jetbrains.com/plugin/14014-spotbugs

## Results and Discussion

We used an Intelij plugin of the tool, which provided as a GUI like the following:



Analysis of alerts that are reported by the tool:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Count of Instances** | **Severity** | **Count of Real Bugs** |
| Dodgy Code  DLS: Dead Local Store | 1 | Medium | 0 |
| Performance  UF:Unread Files | 4 | Medium | 0 |
| Malicious Code Vulnerability  SRMO: Storing Reference Multiple Object | 1 | Medium | 0 |
| Bad Practices  CSE: Checking String Equality using == or != | 4 | High | 0 |
| Bad Practice  CMN:Confusing Method Name | 1 | Medium | 0 |

As can be seen in the table above, as a result of the analysis, "1 DLS, 4 UF, 1 SRMO, 4 CSE AND 1 CMN warnings" appeared. Except for CSE, all of these are medium seriousness, but CSE is high seriousness warnings. As a result of the analysis, a report was created that the operators used would cause slowness and fragility in the system.

# Dependency Analysis

## Tools

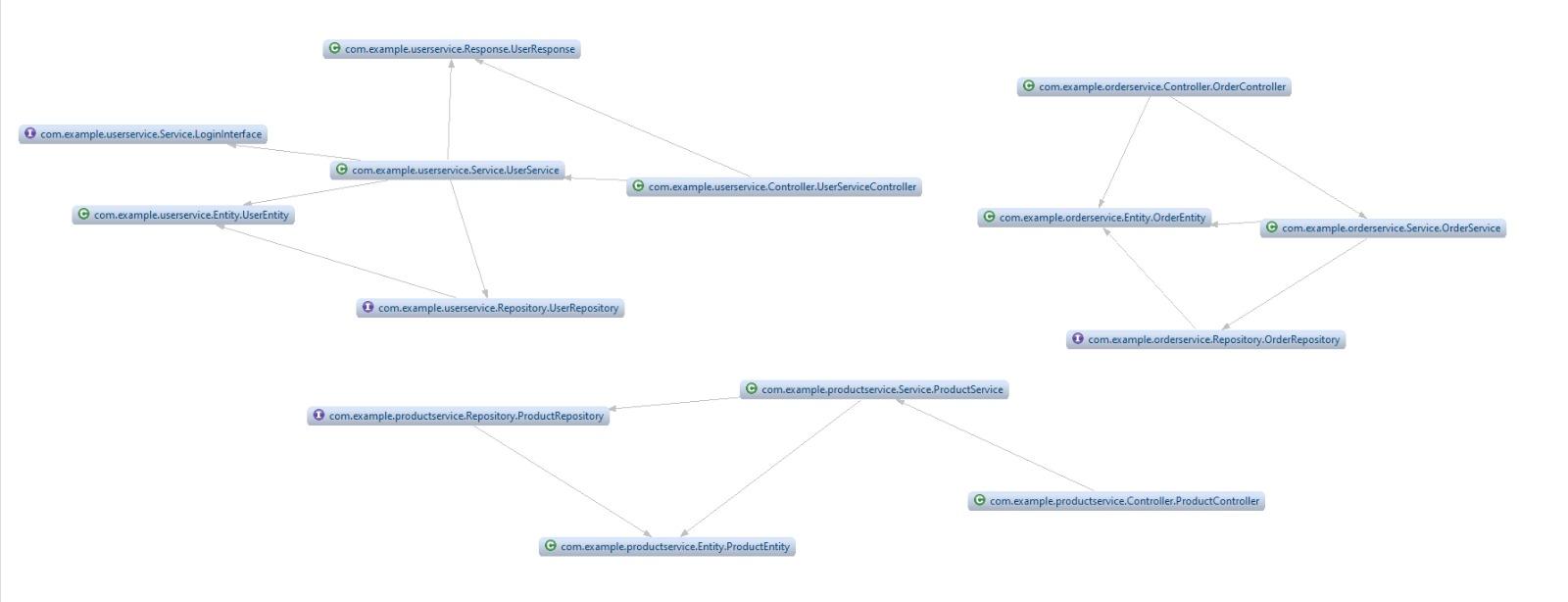
We have used the Java Dependency Viewer from Eclipse since it is easy to use and we had problems with some of the other tools.

The Eclipse Java Dependency Viewer generates a visual map revealing how different parts of a Java project depend on each other. It displays a complicated web of relationships between external libraries, packages, and classes.

This model helps in identifying any potential flaws or areas where the project can struggle. It also helps in better decision-making during development or updates by clarifying the effects of changes to one section of the code on others.

To put it simply, it provides the viewer a clear image of the project's dependencies and structure, helping developers to predict effects, optimize the code, and make wise decisions.

## Results and Discussion



# of Edges = 16

# of Nodes = 14

Edge-to-node Ratio = 16 / 14 = 1.14

Tree Impurity = 2 \* (16 - 14 + 1) / ((14 – 1) \* (14 – 2)) = 6 / 156 = 0.038

It can be observed that the overall dependency structure shows small tree impurity, which can be an indicator for a good design structure. Edge-to-node ratio is also not so high. We can also see that some of the modules are highly connected with some of the other modules. Such as, com.example.userservice.Service.UserService. The reason for that is, the other modules are dependent on that module because it is the core component of that service. In conclusion, the dependency analysis looks good, we have 3 different services and they are all dependent within their own services and they aren’t dependent on any other structure out of their service.

# Test Coverage Analysis

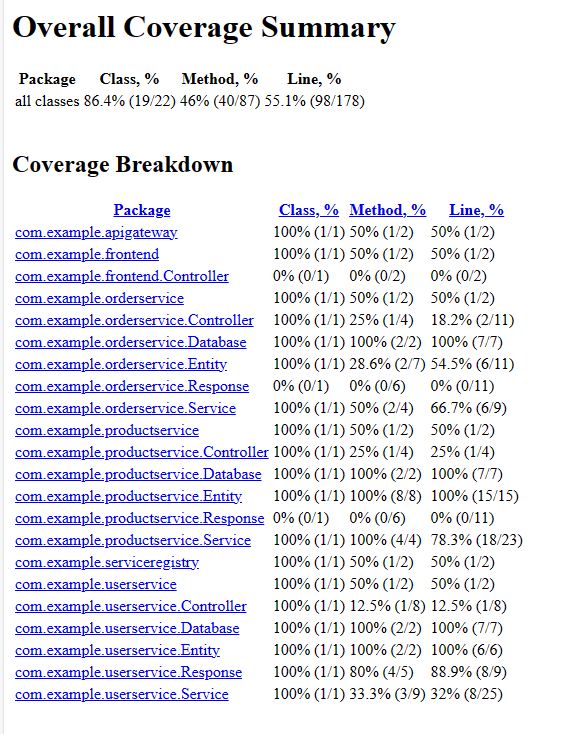
## Tools

The IntelliJ’s test coverage plugin was used for test coverage… This plugin allows you to view results, configure and run tests, and pinpoint specific code sections that require more testing.

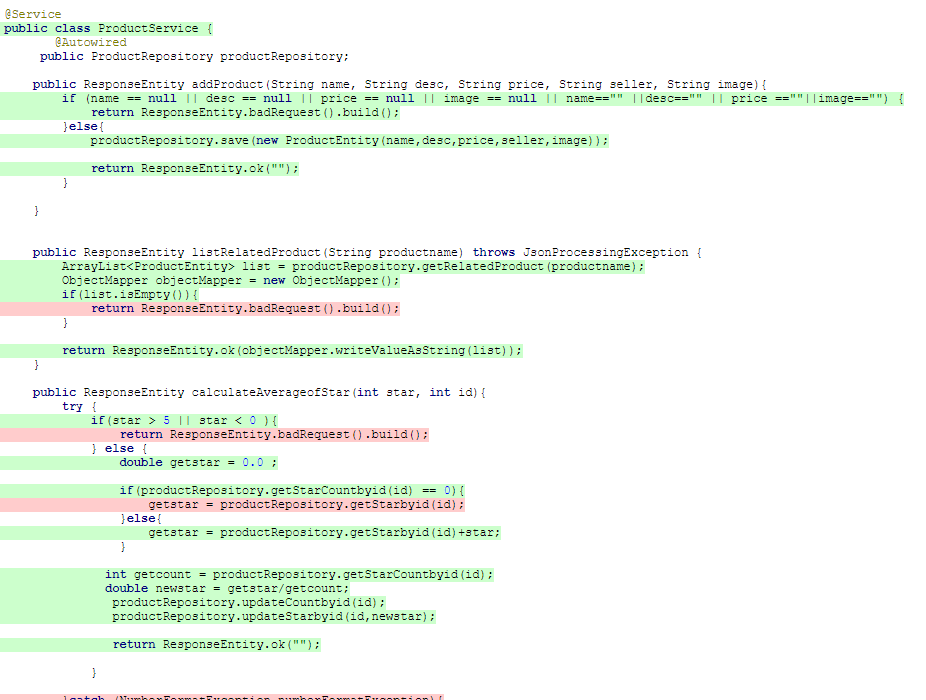
* Test coverage: <https://www.jetbrains.com/help/idea/code-coverage.html>

## Results and Discussion

According to test coverage analysis table below, 86.4% class, 46% method and 55.1% line was covered.



In general, the reason why some method cover tags are low is that our system adheres to the MVC technique, and most of the work is done in the service package, and controller methods were not subject to testing. At the same time, we see that the Response class is at 0%. The reason for this is that the Response class is used to receive information from the frontend and it is 0% because we tested the service package.



The table above, as an example, shows where the operations performed during the test were processed by the system and where they were not. It seems that most transactions have worked successfully and passed the test, but the reason why some processes do not work is related to not meeting certain conditions.