Group 7

Hilda Hermunen, Veera Ruotsalainen, Patrick Scott

Sprint 6

Statistical Code Review

1. Introduction

   The goal of the statistical code review is to find mistakes or bad practices in the code by using tools that automatically check for common problems. These tools help developers clean up and optimize their code, and make sure that best practices are being followed.

2. Static Code Analysis Tools Run

   PMD is a static code analysis tool that scans source code and categorizes issues into five priority levels, with level one being the most critical and level five the least. PMD doesn't analyze security, unlike SonarQube which can detect vulnerabilities and security risks in the code. PMD has been integrated using a Maven dependency in the pom.xml file.

   SonarQube is another static code analysis platform that provides a broader overview of code quality. It analyzes code for bugs, vulnerabilities, code smells and test coverage, and separates the issues into different categories. SonarQube was set up locally and connected to the project using the SonarScanner and a configuration file (sonar-project.properties).

3. Findings

   3.1. SonarQube (SonarScanner)

   In the first scan the Sonar Scanner found 157 different issues, 6,8% coverage and 7,3% duplications. The following information is gathered from the Sonar Scanner analysis report. It contains the main issues in different categories

   Security (1 Issues)

Issue	Impact	Severity	Amount
Make sure this database password gets changed and	If a database password leaks to an unintended audience, it can have serious consequences for the security of your database instance,	Blocker	1

removed from the code.	the data stored within it, and the applications that rely on it.		

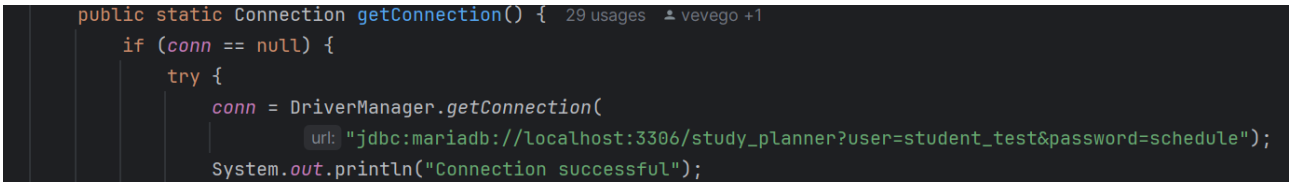


Figure 1: Code snippet of a security issue in the code.

Reliability (33 Issues)

Issue	Impact	Severity	Amount
Either re-interrupt this method or rethrow the "InterruptedException" that can be caught here.	If an InterruptedException or a ThreadDeath error is not handled properly, the information that the thread was interrupted will be lost.	Medium	4
Use try-with-resources or close this "PreparedStatement" in a "finally" clause.	Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box the application is on to their knees.	High	28

Figure 2: A code snippet demonstrating a problem where an InterruptedException is caught but not properly rethrown or the thread is not re-interrupted.

```
try {
    PreparedStatement st = conn.prepareStatement(sql);
    st.setInt( parameterIndex: 1,  id);
    ResultSet rs = st.executeQuery();
    if (rs.next()) {
        return new Assignment(
                rs.getInt( columnLabel: "course_id"),
                rs.getString( columnLabel: "title"),
                rs.getString( columnLabel: "description"),
                rs.getTimestamp( columnLabel: "deadline").toLocalDateTime(),
                rs.getString( columnLabel: "status")
        );
    }
} catch (SQLException e) {
    e.printStackTrace();
}
return null;
```

Figure 3: A code snippet demonstrating the need to use try-with-resources or close the PreparedStatement in a finally block.

Maintainablility (119 Issues)

Issue	Impact	Severity	Amount
Remove the "variable" field and declare it as a local variable in the relevant methods.	When the value of a private field is always assigned to in a class' methods before being read, then it is not being used to store class information. Therefore, it should become a local variable in the relevant methods to prevent any misunderstanding.	Low	9
Declare "variable" and all following declarations on a separate line.	Declaring multiple variables on one line is difficult to read.	Low	19
Use concise character class syntax '\\D' instead of '[^0-9]'.	The latter is not only shorter but easier to read and thus to maintain.	Low	4
Replace this use of System.out by a logger.	In software development, logs serve as a record of events within an application, providing crucial	Medium	31

	insights for debugging. That is why defining and using a dedicated logger is highly recommended.		
Remove this unused private method.	A method that is never called is dead code, and should be removed. Cleaning out dead code decreases the size of the maintained codebase, making it easier to understand the program and preventing bugs from being introduced.	Medium	4
Refactor this method to reduce its Cognitive Complexity to the 15 allowed.	During code reading, the deeper you go through nested layers, the harder it becomes to keep the context in mind.	High	4
Define a constant instead of duplicating this literal “String”	Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.	High	6

Figure 4: Code snippet demonstrating the need to define a constant instead of duplicating literal “String”.

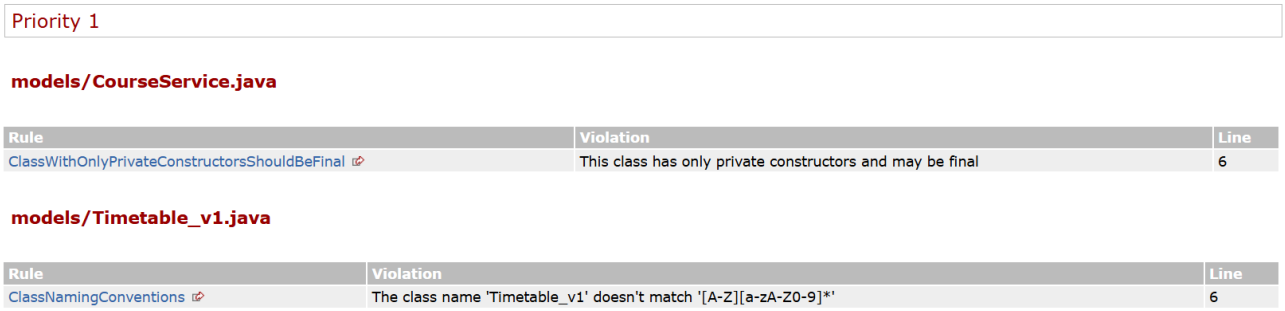
```
if (date != null && deadlineTimeString != null) {
    DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("H:mm");
    LocalDateTime assignmentDeadline = LocalDateTime.of(date, LocalTime.parse(deadlineTimeString, timeFormatter));
    System.out.println(assignmentDeadline);
    System.out.println(date);
```

Figure 5: Code snippet demonstrating the need to replace a use of System.out by a logger.

3.2. PMD

Initial PMD report found 108 violations. Two violations are priority level 1, 81 violations were priority level 3, and 25 violations were priority level 4. The PMD report displays the class where each issue is found, along with the specific rule that is violated and a detailed description of the violation.

The following figure is a screenshot of the initial PMD report that shows the priority level 1 violations. Priority level 1 violations are critical issues that are likely to cause bugs or crashes, and should be fixed as soon as possible.



Priority 1

models/CourseService.java

Rule	Violation	Line
ClassWithOnlyPrivateConstructorsShouldBeFinal	This class has only private constructors and may be final	6

models/Timetable_v1.java

Rule	Violation	Line
ClassNamingConventions	The class name 'Timetable_v1' doesn't match '[A-Z][a-zA-Z0-9]*'	6

Figure 6: A screenshot of priority level 1 violations of initial PMD report.

As shown in Figure 6, the report indicates that the class CourseService has only private constructors and could be declared as final. Additionally, the class Timetable_v1 violates naming convention best practices.

The following figure shows the private constructor on the CourseService class.

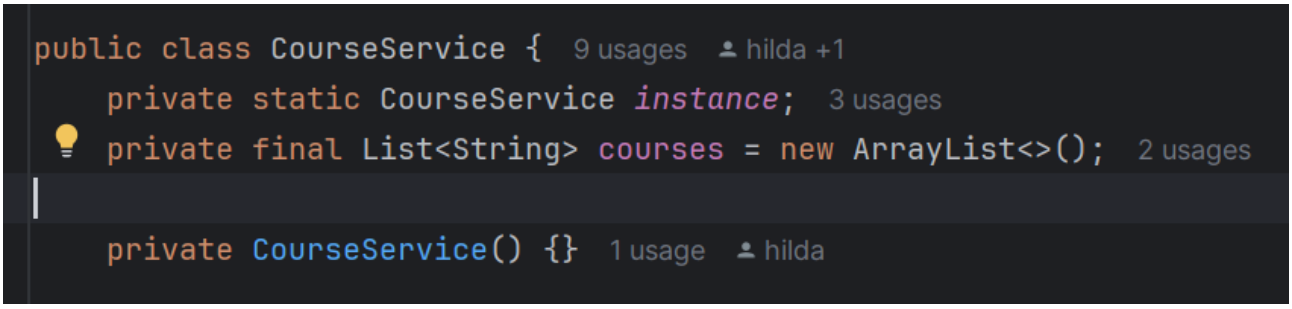


Figure 7: A screenshot of a private constructor in a class that is not final.

As seen on figure 7, the public class CourseService is not declared final. Declaring the class as final would prevent it being subclassed and clarify its purpose as a utility class.

The following figure is a screenshot of the initial PMD report that shows an example of priority level 3 violations. Priority level 3 violations are code quality issues that might not break the program but could make it harder to maintain or extend.

Priority 3

Main.java

Rule	Violation	Line
NoPackage	All classes, interfaces, enums and annotations must belong to a named package	3
UseUtilityClass	This utility class has a non-private constructor	3

config/MariaDbConnection.java

Rule	Violation	Line
UseUtilityClass	This utility class has a non-private constructor	11
NonThreadSafeSingleton	Singleton is not thread safe	19–28

controllers/AddAssignmentController.java

Rule	Violation	Line
SingularField	Perhaps 'timeTableDAO' could be replaced by a local variable.	19
SingularField	Perhaps 'courses' could be replaced by a local variable.	21
LiteralsFirstInComparisons	Position literals first in String comparisons	163

Figure 8: Screenshot of a PMD report showing examples of priority 3 violations.

As shown in Figure 8, an example of a priority level 3 violation is that the main class isn't declared within a package and has a constructor that should be private. These are issues that impact code quality and maintainability.

Figure 9 shows a screenshot of the initial PMD report that shows an example of priority level 4 violations. Priority level 4 violations are minor violations of coding conventions or style guidelines.

Priority 4

controllers/AddAssignmentController.java

Rule	Violation	Line
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	42
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	45
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	48
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	51

controllers/AddClassScheduleController.java

Rule	Violation	Line
UnnecessaryImport	Unused import 'java.util.List'	16
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	57
OneDeclarationPerLine	Use one line for each declaration, it enhances code readability.	60

Figure 9: Screenshot of a PMD report showing examples of priority level 4 violations.

Figure 9 illustrates a priority level 4 violation, where several variables are declared on the same line. According to best practices, each variable should be declared on its own line.

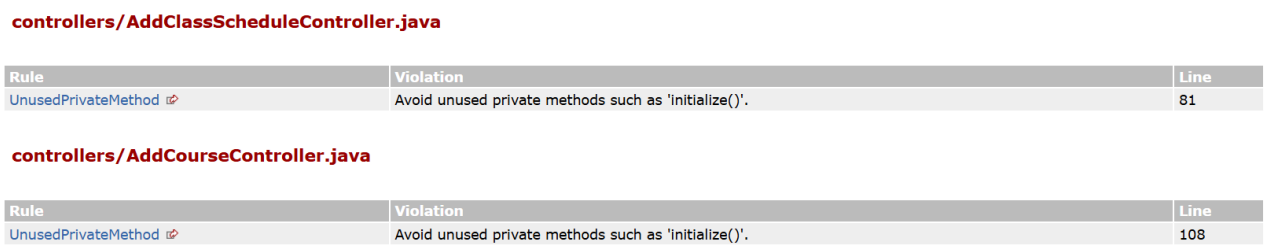
Figure 10 shows an example of the previously mentioned priority level 4 violation.

```
@FXML
private Button backButton, assignmentSaveButton;
```

Figure 10: A screenshot of a multiple variable declaration.

As seen in Figure 10, declaring multiple variables on a single line can reduce code readability.

PMD analysis is not flawless. Figure 11 shows a priority level 3 violation reported by PMD that cannot be resolved manually.



controllers/AddClassScheduleController.java

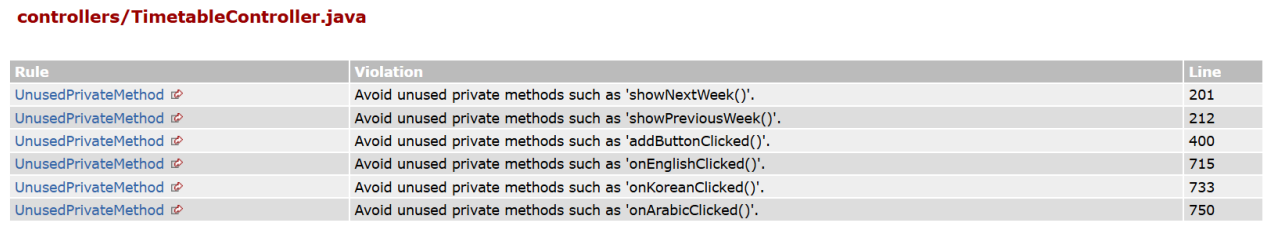
Rule	Violation	Line
UnusedPrivateMethod	Avoid unused private methods such as 'initialize()'.	81

controllers/AddCourseController.java

Rule	Violation	Line
UnusedPrivateMethod	Avoid unused private methods such as 'initialize()'.	108

Figure 11: A screenshot of the initial PMD report showing violations related to unused private methods.

In this case, PMD marks the “initialize()” method as unused. However, in JavaFX, this method is not called explicitly in the code. Instead, it is automatically invoked by the JavaFX framework when the FXML file is loaded. The same false violations can be seen in figure 12.



controllers/TimetableController.java

Rule	Violation	Line
UnusedPrivateMethod	Avoid unused private methods such as 'showNextWeek()'.	201
UnusedPrivateMethod	Avoid unused private methods such as 'showPreviousWeek()'.	212
UnusedPrivateMethod	Avoid unused private methods such as 'addButtonClicked()'.	400
UnusedPrivateMethod	Avoid unused private methods such as 'onEnglishClicked()'.	715
UnusedPrivateMethod	Avoid unused private methods such as 'onKoreanClicked()'.	733
UnusedPrivateMethod	Avoid unused private methods such as 'onArabicClicked()'.	750

Figure 12: A screenshot of the initial PMD report showing violations related to unused private methods.

The violations shown in Figure 12 cannot be fixed manually. However, PMD can be configured to ignore specific rule violations. This approach would be risky if the product is to be further developed, as ignoring a rule violation may cause valid violations to be overlooked in the future.

3.3. Manual Evaluation

Reviewing the code itself, there were logical structures comprising multiple “if-else” blocks of similar conditions within various methods of the “TimetableController” class. Refactoring these methods to instead use a single line reduces code length and improves readability.

```
private <T> String getEventStartTime(T event) {  1 usage   hilda
    if (event instanceof ClassSchedule classSchedule) {
        return classSchedule.getStartTime().toLocalTime().toString();
    } else if (event instanceof StudySession studySession) {
        return studySession.getStartTime().toLocalTime().toString();
    } else if (event instanceof Exam exam) {
        return exam.getExamDate().toLocalTime().toString();
    } else if (event instanceof Assignment assignment) {
        return assignment.getDeadline().toLocalTime().toString();
    }
    return "00:00"; // Default
}
```

Figure 13: A screenshot of the unrefactored method “getEventStartTime”, having four if-else blocks.

```
private String getEventStartTime(MyEvent myEvent) {
    return myEvent.getEventStartTime().toString();
}
```

Figure 14: A screenshot of the refactored method “getEventStartTime”, using a single line to return an inherited method of the parameter.

4. Code Clean-Up

4.1. SonarQube

After code clean-up that followed the results from the SonarQube scan, the issues decreased from 157 to 43. The security issues dropped to 0, reliability issues dropped to 0, and Maintainability issues dropped to 43.

The following tasks were done to improve the quality of the code:

- Made sure that the database password was changed and removed from the code.
- Re-interrupt threads in a few methods for thread safety.
- Used try-with-resources to handle a preparedStatement.

- Removed variable fields and declared them as a local variable in the relevant methods.
- Use concise character class syntax '\\D' instead of '[^0-9]'.
- Remove this unused private method.
- Refactor this method to reduce its Cognitive Complexity to the 15 allowed.
- Define a constant instead of duplicating the literal.

Issues with severity Blocker or High were all solved. The duplications dropped by 0,3%. The tests coverage stayed the same. The test coverage will be higher after we get Mockito to work in the project as the missing tests depend on it.

4.2. PMD

After the code clean-up, the number of violations in the PMD report dropped from 108 to 15. Nine of the remaining violations are the previously mentioned UnusedPrivateMethod violations, which cannot be fixed manually. Both priority level 1 violations were resolved, along with all 25 priority level 4 violations. The remaining violations are all at priority level 3, meaning that 66 level 3 violations were fixed.

Both priority level 1 violations are shown in figure 6. The violations were fixed by making the ”CourseService” class final and renaming ”TimetableController_v1” to ”TimetableController”. These changes improve code clarity and enforce better design principles.

All priority level 4 violations were fixed. Figure 9 shows examples of these violations. Most of the violations were related to multiple FXML variables declared in the same line, rather than each variable having its own line. Separating each variable to its own line enhances code readability.

Another example of a priority level 3 violation that was not fixed is shown in Figure 8, where the ”Main” class is flagged as a utility class with a non-private constructor. However, this is not a valid violation, since the ”Main” class is not a utility class.

4.3. Code Refactoring

4.3.1. Creating the “MyEvent” class

To address the issue in 3.3, a parent class “MyEvent” was made to reduce method content.

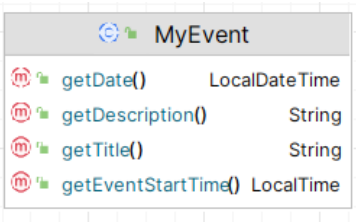


Figure 15: A screenshot of the ”MyEvent” class diagram, showing its methods.

This provides each event type with methods for retrieving shared data types. When the “TimetableController” class refers to an event, it can now do so through this abstract class.

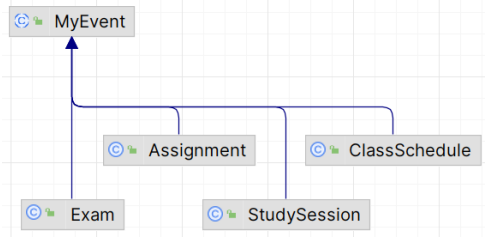


Figure 16: The inheritance diagram, showing the relation of each event type under the ”MyEvent” class.

5. Summary

After reviewing the code manually and with two statistical code analysis tools, the quality of the code has overall improved. Though not flawless, the statistical code analysis tool helps implement best practices and identify possible issues early in development. As shown in this report, the tools helped eliminate high-priority issues and improve code readability and maintainability. The most beneficial would be to implement a statistical code analysis tool quite early in the development process.