Lab 1 Report

SWE 4802: Software Maintenance Lab

Submitted To:

Lutfun Nahar Lota
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology

Submitted By:

Abrar Mahabub (200042103)
Mashrur Ahsan (200042115)
Nafisa Maliyat (200042133)
Shanta Maria (200042172)

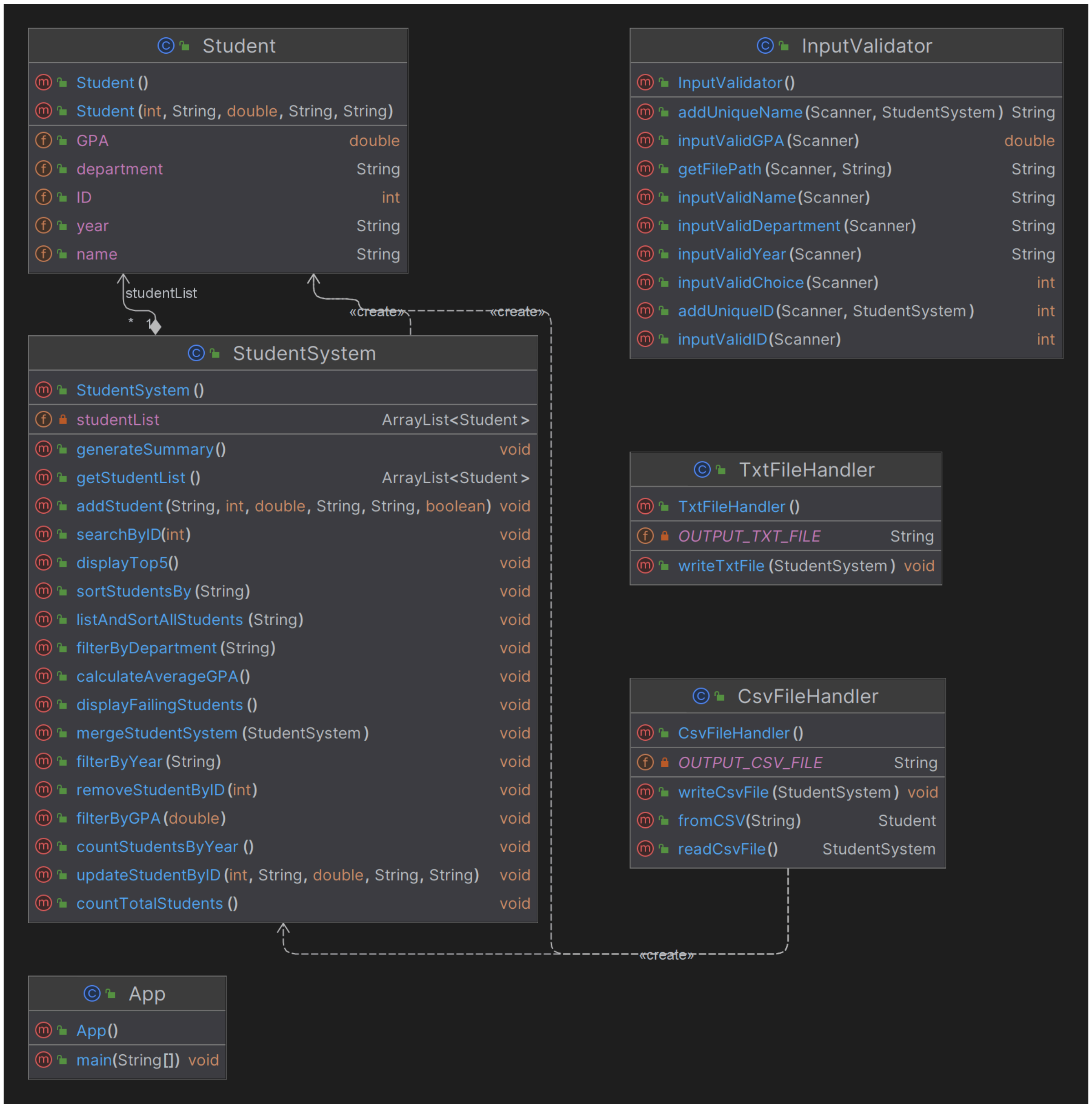
Table of Contents

1 Project

1.1 Project Description

The Student Management System is a Java-based console application designed to manage student records efficiently. It allows users to perform various operations such as adding, removing, updating, and searching for students. Additionally, it provides features like sorting, filtering, and generating performance summaries. This system is ideal for educational institutions or anyone needing to manage student data in a structured manner.

1.2 UML Diagram



Student
Student()
Student(int, String, double, String, String)
GPA double
department String
ID int
year String
name String
studentList
«create»
«create»
StudentSystem
StudentSystem()
studentList ArrayList<Student>
generateSummary() void
getStudentList() ArrayList<Student>
addStudent(String, int, double, String, String, boolean) void
searchByID(int) void
displayTop5() void
sortStudentsBy(String) void
listAndSortAllStudents(String) void
filterByDepartment(String) void
calculateAverageGPA() void
displayFailingStudents() void
mergeStudentSystem(StudentSystem) void
filterByYear(String) void
removeStudentByID(int) void
filterByGPA(double) void
countStudentsByYear() void
updateStudentByID(int, String, double, String, String) void
countTotalStudents() void
InputValidator
InputValidator()
addUniqueName(Scanner, StudentSystem) String
inputValidGPA(Scanner) double
getFilePath(Scanner, String) String
inputValidName(Scanner) String
inputValidDepartment(Scanner) String
inputValidYear(Scanner) String
inputValidChoice(Scanner) int
addUniqueID(Scanner, StudentSystem) int
inputValidID(Scanner) int
TxtFileHandler
TxtFileHandler()
OUTPUT_TXT_FILE String
writeTxtFile(StudentSystem) void
CsvFileHandler
CsvFileHandler()
OUTPUT_CSV_FILE String
writeCsvFile(StudentSystem) void
fromCSV(String) Student
readCsvFile() StudentSystem
«create»
App
App()
main(String[]) void

2 Static Analysis

2.1 SonarQube

SonarQube is a code quality tool that helps developers identify and fix problems in their software. It scans the code to detect bugs, security risks, and poorly structured sections that could make the program harder to maintain. It helps the developers by:

- Catching the bugs early by detecting potential errors that could lead to crashes or unexpected behavior
- Improving security by flagging vulnerabilities that might attack the system
- Improving maintainability by identifying code that can make future updates easier
- Providing a multitude of metrics broken down to each file to make it easier to identify where the problem is

In the context of this student management system project, SonarQube's analysis revealed critical gaps in test coverage (0%) and highlighted areas requiring security improvements and code refactoring (3.7% duplication). It also showed 211 maintainability issues that might make this application difficult to update with new features. All relevant sections are discussed in the following sections.

2.1.1 Setup and Installation Guide

Prerequisites for the setup

- Java project
- Docker installed.
- An internet connection to download the required Docker images and tools.

Step 1: The SonarQube Community Edition

This can be downloaded through Docker using the following command:

```
docker run -d --name sonarqube -p 9000:9000 sonarqube:community
```

After the image is downloaded and the container runs, SonarQube will be accessible via http://localhost:9000.

Step 2: SonarScanner CLI

The latest version of SonarScanner CLI can be downloaded from the official site: SonarScanner CLI. After the downloaded zip file is extracted to a desired location, the

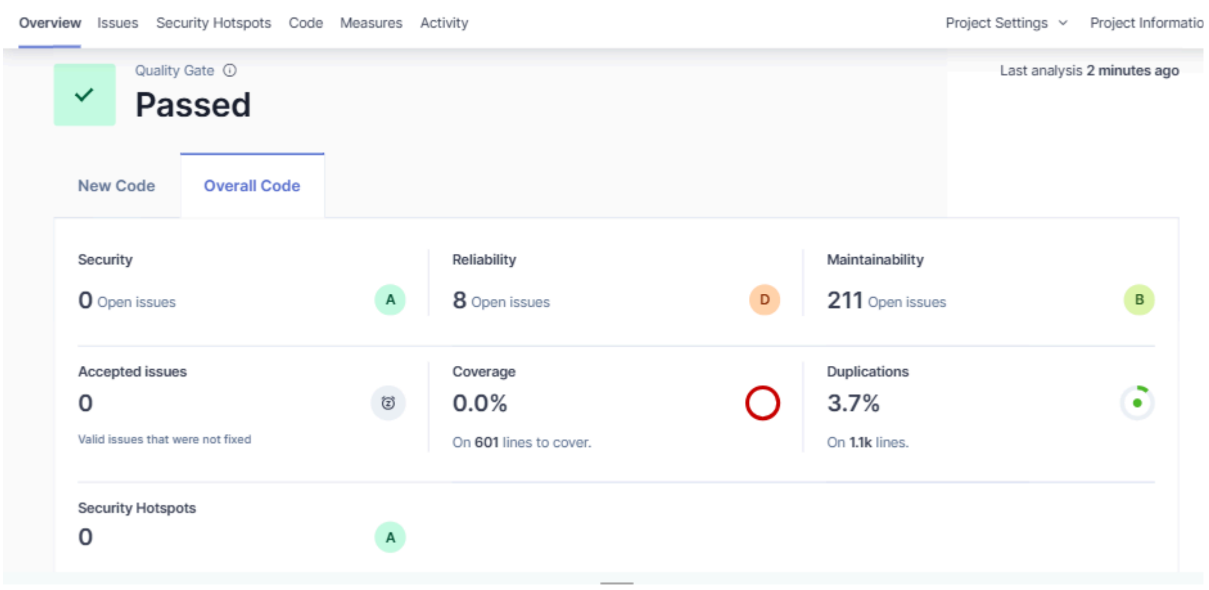
path to the bin/ folder (inside the extracted directory) has to be added to the System Environment Variables (Path).
The installation can be verified via terminal using the following command:
```
sonar-scanner -v
```

Step 3: Generating Token from SonarQube
At http://localhost:9000/account/security, the following path should be navigated:
Account → Security → Generate Tokens
This assumes that the Docker container started in *Step 1* is still running, since localhost is required.
A global token with no expiration date was selected in this case since this was an academic project. The token should be copied and stored securely since it is only provided once.

Step 4: Creating Configuration File
A sonar-project.properties file has to be created inside the root folder with basic configurations. A boilerplate template is provided at the official website SonarScanner CLI | SonarQube Server Documentation.

Step 5: Running SonarQube
Using the terminal of IntelliJ IDEA or using the command line inside the project directory, the following command will start analysis and send a full report to http://localhost:9000/projects, where different metrics can be viewed.

2.1.2 Overview
When the project is opened in SonarQube, a clear overview of key metrics—such as bugs, code smells, vulnerabilities, coverage, and duplications—is immediately displayed. Any critical issue can be selected and expanded to view detailed insights at the folder or file level.



The Quality Gate provides an overall status indicating whether the project meets the defined quality standards. If the project fails to pass, SonarQube highlights exactly which metrics fall short and what adjustments are needed to meet the acceptable thresholds.

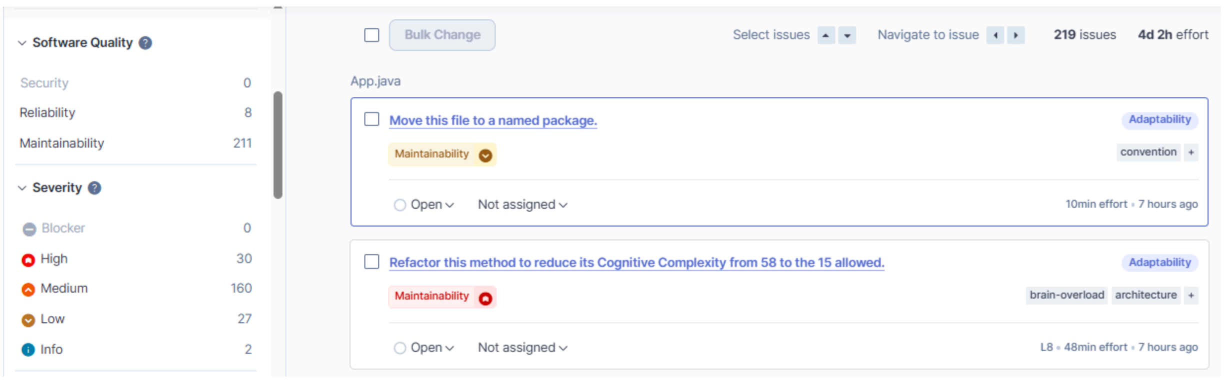
The quick snapshot of metrics reveals there are issues in the following areas:

- Reliability: 8 issues with a grading of D indicate the system might be at a high risk of showing unexpected behavior
- Maintainability: 211 issues indicate that the project might be difficult to update for new features
- Coverage: 0% coverage indicates no test cases to monitor the project

For each section, it also provides two subsections: New Code and Overall Code. New code tracks the code changes across different analyses and whether the new code has introduced issues across any areas. Since the code has not been changed for this lab, the new code metrics are not relevant for this report.

2.1.3 Metrics

For a more detailed view of the issues detected in the project, the Issues tab provides an organized breakdown of all findings, categorized by type, severity, and location.


Software Quality

Security	0
Reliability	8
Maintainability	211

Severity

Blocker	0
High	30
Medium	160
Low	27
Info	2

Bulk Change   Select issues   Navigate to issue   219 issues   4d 2h effort

App.java

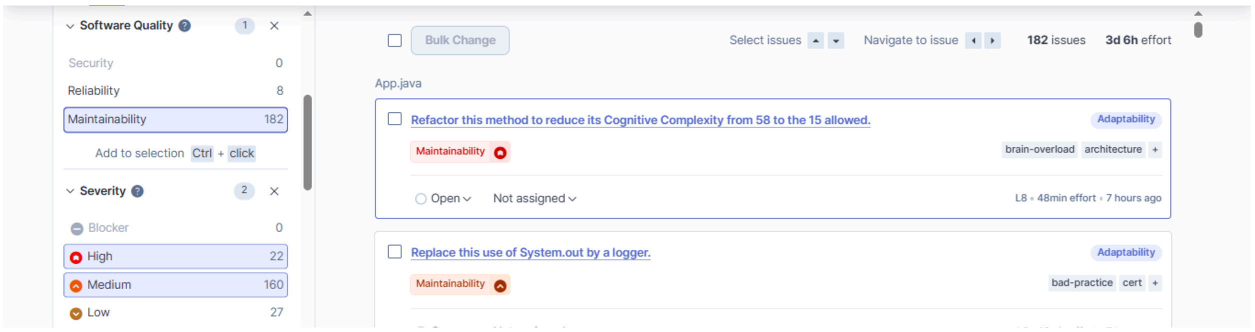
Move this file to a named package.   Adaptability
Maintainability   convention +
Open   Not assigned   10min effort · 7 hours ago

Refactor this method to reduce its Cognitive Complexity from 58 to the 15 allowed.   Adaptability
Maintainability   brain-overload   architecture +
Open   Not assigned   L8 · 48min effort · 7 hours ago

The issues page shows the following:

- Different categories of issues that can be filtered to gain specific insight
- Total number of issues and the estimated development effort to reduce them
- A list of issues, with each of them displaying detailed information
    - Instructions to resolve the issue with a link to the exact line in the codebase
    - The category of issue is tagged
    - Current status of the issue (e.g., open, resolved, false positive), which can be manually
    - Option to assign to someone in the project
    - The effort estimation for that specific issue
    - Timestamp of when this issue was introduced
    - Option to add tags to issues for good organization and tracking

Specific filters can be applied across different categories of issues to view which issues might be significant.


Software Quality 1

Security	0
Reliability	8
Maintainability	182

Add to selection Ctrl + click

Severity 2

Blocker	0
High	22
Medium	160
Low	27

Bulk Change   Select issues   Navigate to issue   182 issues   3d 6h effort

App.java

Refactor this method to reduce its Cognitive Complexity from 58 to the 15 allowed.   Adaptability
Maintainability   brain-overload   architecture +
Open   Not assigned   L8 · 48min effort · 7 hours ago

Replace this use of System.out by a logger.   Adaptability
Maintainability   bad-practice   cert +

Category		Issue Count
Software Quality	Security	0
	Reliability	8

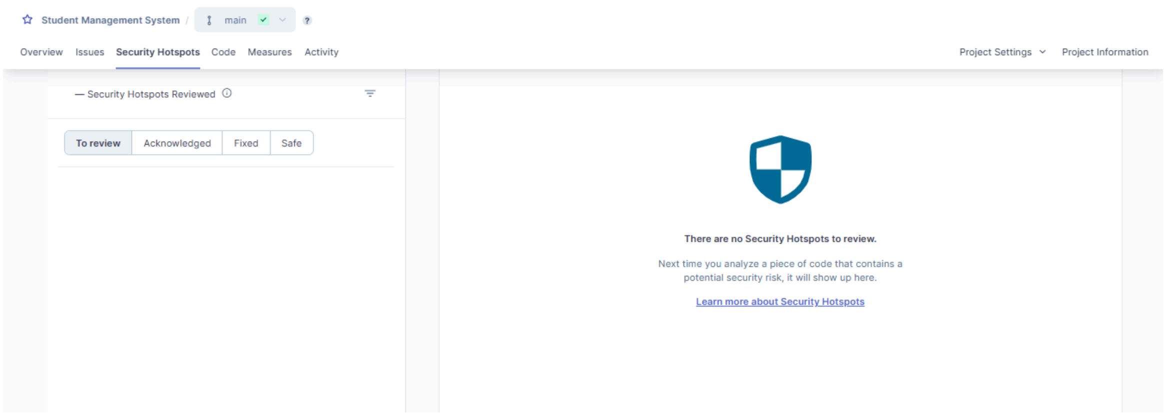
	Maintainability	211
Severity	Blocker	0
	High	30
	Medium	160
	Low	27
	Info	2
Clean Code Attribute	Consistency	19
	Intentionality	16
	Adaptability	184
	Responsibility	0

It can be seen from the data that the codebase has significant problems in maintainability and reliability that are impacting its quality. Most of the issues are medium. However, there are some high-severity issues that should be dealt with as soon as possible. Since there is no blocker, none of the issues are significant enough to stop the application from working. In the clean code attribute, the code might be difficult to adapt to any new features or changes.

Based on the metrics, the code appears to be rigid and potentially fragile, meaning it may not respond well to future modifications or enhancements. The high number of maintainability issues and the absence of test coverage suggest that introducing new features or making changes could lead to unexpected breakages.

2.1.4 Security Hotspots

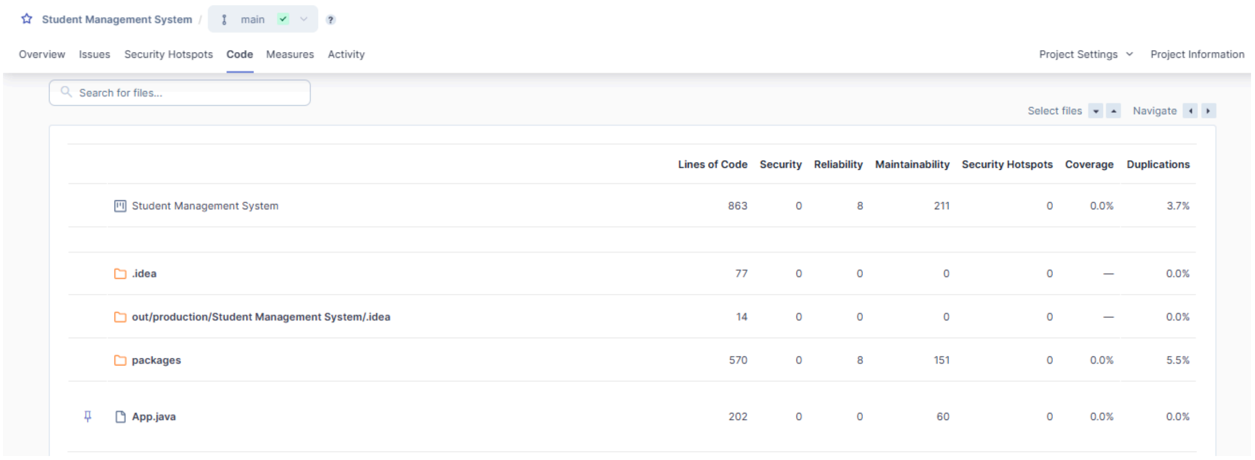
Security hotspots highlight the sensitive security issues that the developer needs to review. For this project, there were no security hotspots detected. This is expected since the project is a simple CRUD Java application that does not contain sensitive functions like authentication and input from external sources.


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information
Security Hotspots Reviewed
To review Acknowledged Fixed Safe
Learn more about Security Hotspots

There are no Security Hotspots to review.

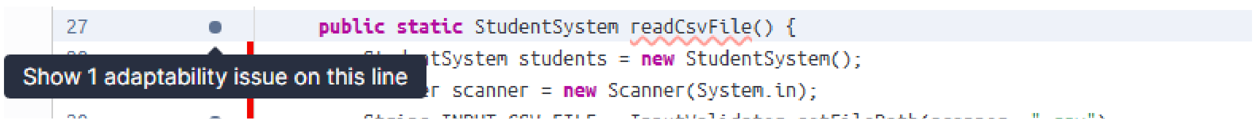
Next time you analyze a piece of code that contains a potential security risk, it will show up here.

2.1.5 Code


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information
Search for files...
Select files Navigate

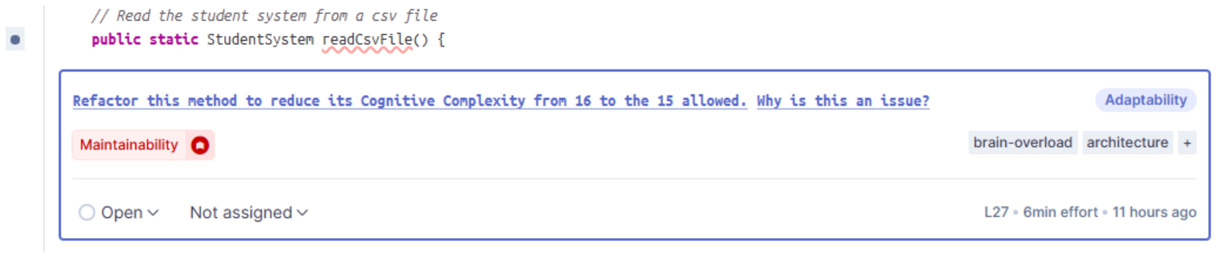
	Lines of Code	Security	Reliability	Maintainability	Security Hotspots	Coverage	Duplications
Student Management System	863	0	8	211	0	0.0%	3.7%
.idea	77	0	0	0	0	—	0.0%
out/production/Student Management System/.idea	14	0	0	0	0	—	0.0%
packages	570	0	8	151	0	0.0%	5.5%
App.java	202	0	0	60	0	0.0%	0.0%

The Code section in SonarQube displays metrics directly related to the codebase, such as lines of code (LOC), duplications, security hotspots, and complexity. These metrics can be explored at different levels of the project structure, including specific folders and files. Furthermore, it also highlights the line in the codebase where the issue lies, including the type of issue, helpful comments, and lines uncovered by test cases (as shown in diagrams below).



```
    public static StudentSystem readCsvFile() {
        tSystem students = new StudentSystem();
        r scanner = new Scanner(System.in);
```

Show 1 adaptability issue on this line



```
    // Read the student system from a csv file
    public static StudentSystem readCsvFile() {
```

Refactor this method to reduce its Cognitive Complexity from 16 to the 15 allowed. Why is this an issue?

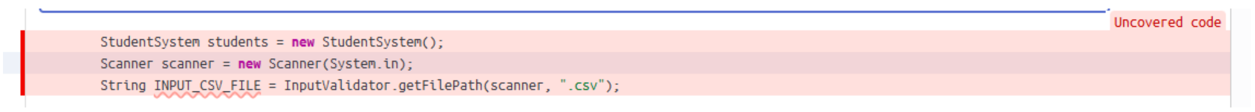
Adaptability

Maintainability

brain-overload architecture +

Open Not assigned

L27 • 6min effort • 11 hours ago



Uncovered code

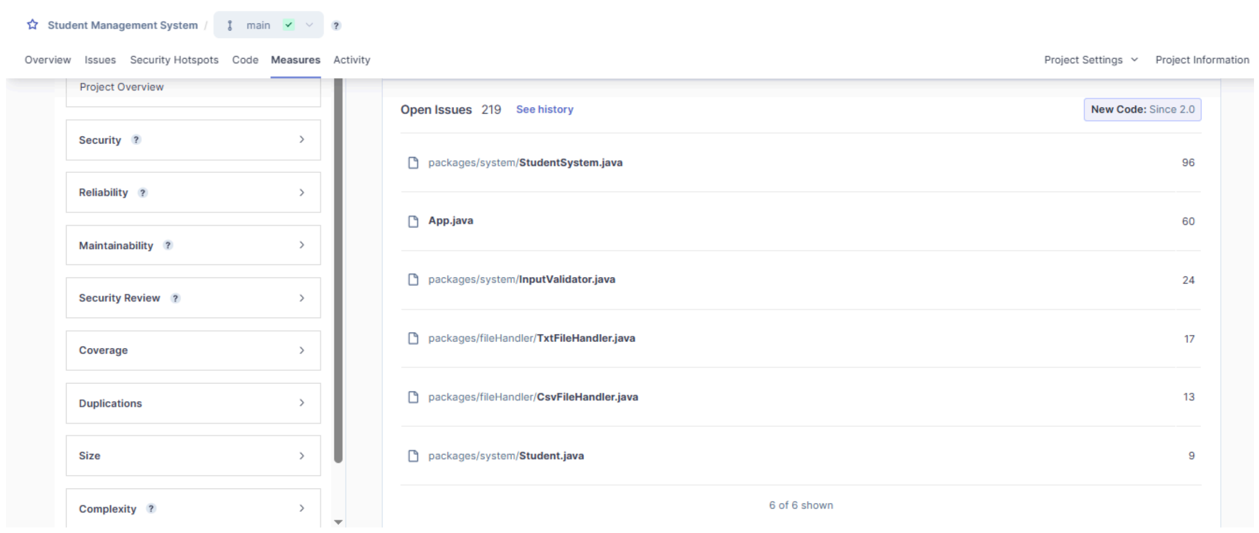
```
        StudentSystem students = new StudentSystem();
        Scanner scanner = new Scanner(System.in);
        String INPUT_CSV_FILE = InputValidator.getFilePath(scanner, ".csv");
```

Category	Count
Lines of Code	570
Security	0
Reliability	8
Maintainability	151
Security Hotspots	0
Coverage	0.0%
Duplications	5.5%

The values from the table show that the project is a small-scale application that has no detected vulnerabilities or security hotspots, likely because there is no handling of sensitive data. There are currently eight bugs in the system that might cause unexpected behavior and the application could be difficult to maintain due to a large number of maintainability issues. Since tests are not included, coverage is at a 0.0% and there is a slight number of duplications.

The highest priority would be to handle the maintainability and reliability issues in this project.

2.1.6 Measures



Open Issues 219

packages/system/StudentSystem.java	96
App.java	60
packages/system/InputValidator.java	24
packages/fileHandler/TxtFileHandler.java	17
packages/fileHandler/CsvFileHandler.java	13
packages/system/Student.java	9

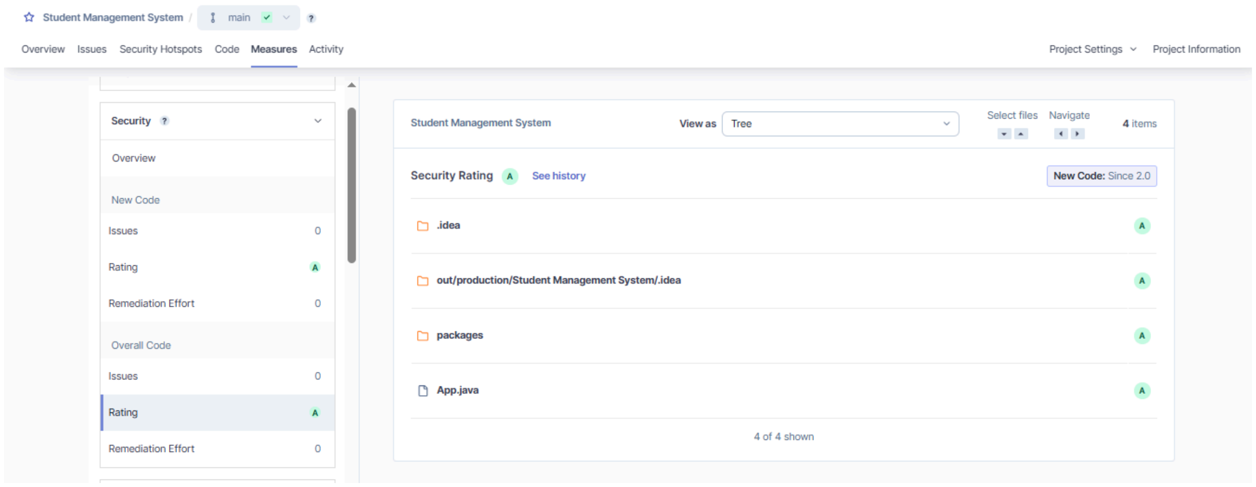
6 of 6 shown

The Measures tab provides additional details for each issue highlighted in the Code section. It includes the estimated effort required to resolve the issues and achieve an acceptable quality rating, as well as the technical debt ratio - the cost of fixing code issues relative to the time spent writing the code. Each metric can be viewed for each file for easy identification of the location where a fix is needed.

File	Issue Count
`StudentSystem`	98
`App.java`	60
`InputValidator`	24
`TxtFileHandler`	17
`CsvFileHandler`	13
`Student`	9

This table shows an overall view of individual files, further pinpointing that StudentSystem and App files will have to be optimized first, since these have the highest number of issues. The next metrics will help to identify exactly which lines might be the source.

2.1.6.1 Security


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information

Security	
Overview	
New Code	
Issues	0
Rating	A
Remediation Effort	0
Overall Code	
Issues	0
Rating	A
Remediation Effort	0

Student Management System — View as: Tree — 4 items

Security Rating A — See history — New Code: Since 2.0

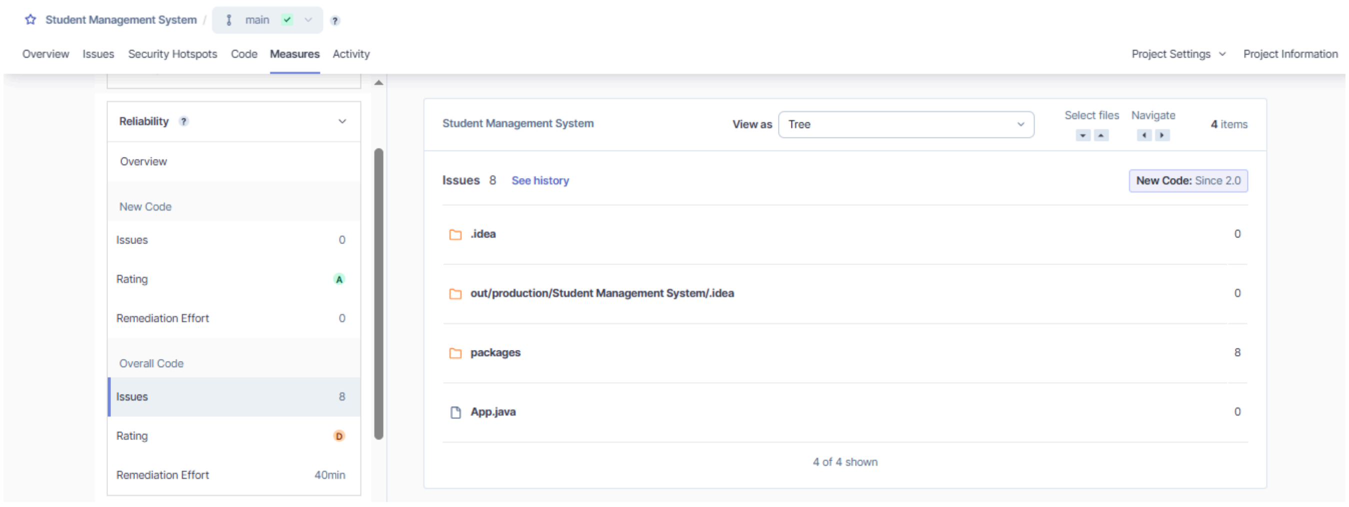
File	Security Rating
.idea	A
out/production/Student Management System/.idea	A
packages	A
App.java	A

4 of 4 shown

The security subcategory contains a grading system from A to D, where A is the best rating. None of the files in the application has any security issues that were detected in the analysis.

2.1.6.2 Reliability

The Reliability rating here is D because there is at least one issue with high impact on the reliability of the project. There's in total 8 issues regarding reliability that span across two classes.


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information

Reliability	
Overview	
New Code	
Issues	0
Rating	A
Remediation Effort	0
Overall Code	
Issues	8
Rating	D
Remediation Effort	40min

Student Management System — View as: Tree — 4 items

Issues 8 — See history — New Code: Since 2.0

File	Issues
.idea	0
out/production/Student Management System/.idea	0
packages	8
App.java	0

4 of 4 shown

File	Issues	Severity

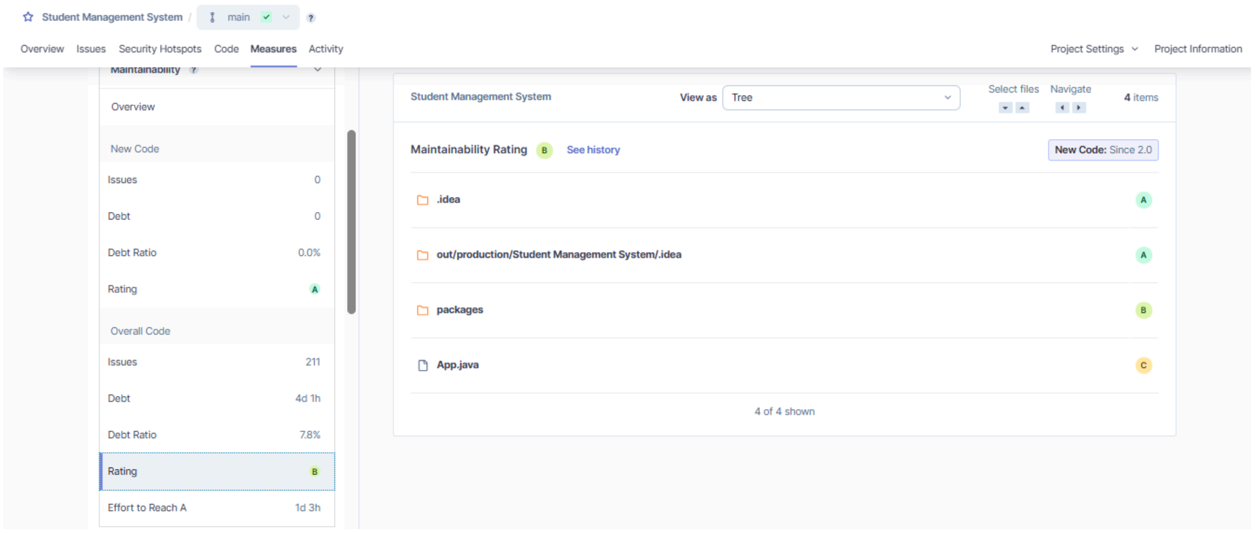
`StudentSystem.java`	4	High
`TxtFileHandler.java`	4	High

Here in both of the files suffer from ‘divide by zero’ issues. If the denominator in an integer division or remainder operation is zero, an ArithmeticException is thrown. This error will crash your program in most cases. To fix it, it has to be ensured that the denominator value in all division operations is always non-zero, or check the value against zero before performing the division.

These types of issues may interrupt the normal execution of the program, causing it to crash or putting it into an inconsistent state. Hence, this issue might impact the availability and reliability of the application.

2.1.6.3 Maintainability

Maintainability of the codebase has been rated as B because the code has a relatively higher level of technical debt when compared to the size of the codebase. Here technical debt refers to the parts of the code that may be poorly structured, lack correct documentation, or exhibit bad practices.



Student Management System

New Code	
Issues	0
Debt	0
Debt Ratio	0.0%
Rating	A

Overall Code	
Issues	211
Debt	4d 1h
Debt Ratio	7.8%
Rating	B
Effort to Reach A	1d 3h

Maintainability Rating B

New Code: Since 2.0

.idea	A
out/production/Student Management System/.idea	A
packages	B
App.java	C

4 of 4 shown

There are in total 211 maintainability issues across 6 classes. Here’s the distribution:

File	Issues	High Severity	Medium Severity	Low Severity
`StudentSystem.java`	92	14	69	9
`App.java`	60	3	54	2
`InputValidator.java`	24	0	22	2
`TxtFileHandler.java`	13	2	9	2
`CsvFileHandler.java`	13	3	6	3
`Student.java`	9	0	0	9

As we can see the larger share of maintainability issues lies in the App file and in the file that's responsible for student operations. These are the core elements of the project. To ensure the project remains adaptable to change and cost efficiency in the long term these issues need to be addressed and resolved.

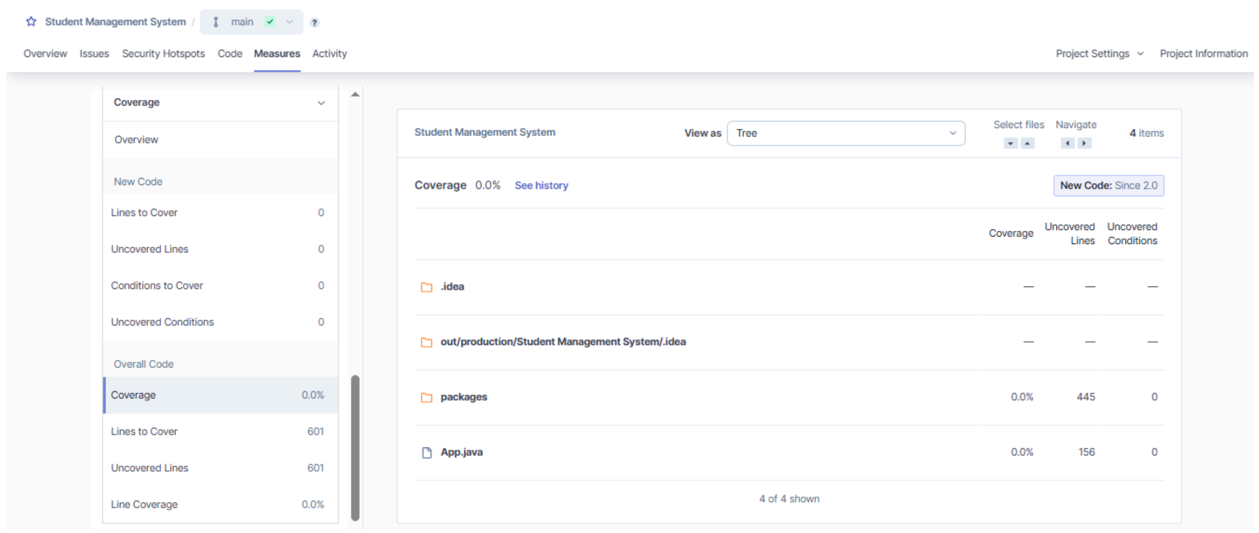
Since there were a significant amount of issues regarding maintainability we can categorise them based on the types of issues - the rules and conventions that were violated and the number of times each of them was violated. Here we're only portraying the most significant ones:

Rules violated	Occurrences	Severity
Standard outputs should not be used directly to log anything	157	Medium
String literals should not be duplicated	13	High
Local variable and method parameter names should comply with a naming convention	9	Low
Class variable fields should not have public accessibility	5	Low
Multiple variables should not be declared on the same line	5	Low
"switch" statements should have "default" clauses	4	High
Cognitive Complexity of methods should not be too high	4	High

From the violated conventions and coding rules, we can say that these patterns suggest weak encapsulation and improper logging practices which may lead to debugging issues making the code much more susceptible to change and to errors.

The code also has high levels of duplications, harming the reusability of it along with Improper flow handling. All in all, The codebase clearly shows the presence of significant technical debt, if these issues are not resolved then it can slow down future development.

2.1.6.4 Coverage


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information
Coverage
Overview
View as Tree
Select files Navigate
4 items
See history
New Code: Since 2.0
4 of 4 shown

New Code	
Lines to Cover	0
Uncovered Lines	0
Conditions to Cover	0
Uncovered Conditions	0
Overall Code	
Coverage	0.0%
Lines to Cover	601
Uncovered Lines	601
Line Coverage	0.0%

Student Management System — Coverage 0.0%

	Coverage	Uncovered Lines	Uncovered Conditions
.idea	—	—	—
out/production/Student Management System/.idea	—	—	—
packages	0.0%	445	0
App.java	0.0%	156	0

The **Coverage** metric reflects the extent to which the source code is tested by unit tests. In this project, the coverage stands at **0.0%**, meaning that no part of the codebase is currently covered by test cases. This also implies that all **601 lines** marked for testing are currently **uncovered**, which includes both logical paths and possible conditions.

SonarQube uses this data to highlight how testable and test-driven the project is. The absence of any test coverage may indicate potential risks in catching regressions or runtime bugs, especially as the system scales.

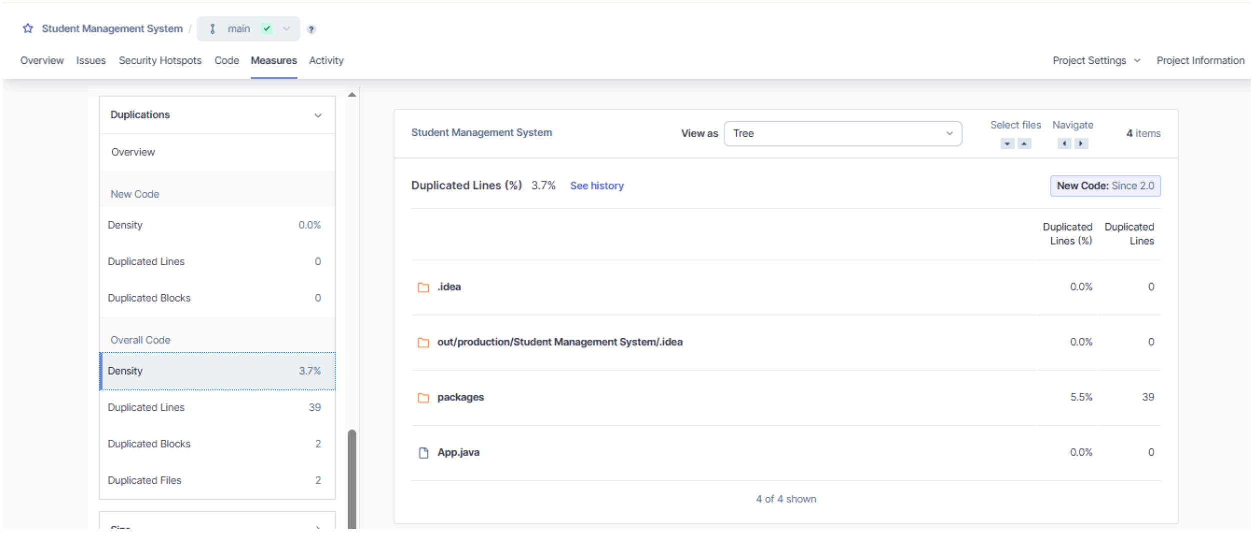
The **Coverage Overview chart** visualizes the proportion of uncovered lines across different classes. Each bubble size represents the number of uncovered lines. Larger bubbles such as those for `StudentSystem` and `App.java` show that these files are major contributors to the overall lack of coverage.

Coverage Distribution by File:

File	Coverage	Uncovered Lines	Uncovered Conditions
`StudentSystem.java`	0.0%	224	0
`App.java`	0.0%	156	0
`TxtFileHandler.java`	0.0%	81	0
`InputValidator.java`	0.0%	80	0
`CsvFileHandler.java`	0.0%	53	0
`Student.java`	0.0%	7	0

This table clearly identifies the files lacking any test coverage. Since **StudentSystem.java** and **App.java** contain the highest number of uncovered lines, it would be beneficial to prioritize adding unit tests to these files. Introducing adequate coverage can greatly enhance the overall reliability and maintainability of the project.

2.1.6.5 Duplications


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information
View as Tree
Select files Navigate
4 items
See history
New Code: Since 2.0
4 of 4 shown

Duplications

Overview

New Code

Density	0.0%
Duplicated Lines	0
Duplicated Blocks	0

Overall Code

Density	3.7%
Duplicated Lines	39
Duplicated Blocks	2
Duplicated Files	2

Student Management System

Duplicated Lines (%) 3.7%

	Duplicated Lines (%)	Duplicated Lines
.idea	0.0%	0
out/production/Student Management System/.idea	0.0%	0
packages	5.5%	39
App.java	0.0%	0

In case of duplications, we basically observe repeated parts in the codebase, either line-by-line or in slightly modified form. According to SonarQube, our project has **3.9% duplicated lines**, which translates to **39 lines** spread across 2 duplicated blocks.

Duplication Breakdown:

File	Duplicated Lines	Duplicated Lines (%)
`StudentSystem.java`	20	6.1%
`TxtFileHandler.java`	19	14.5%

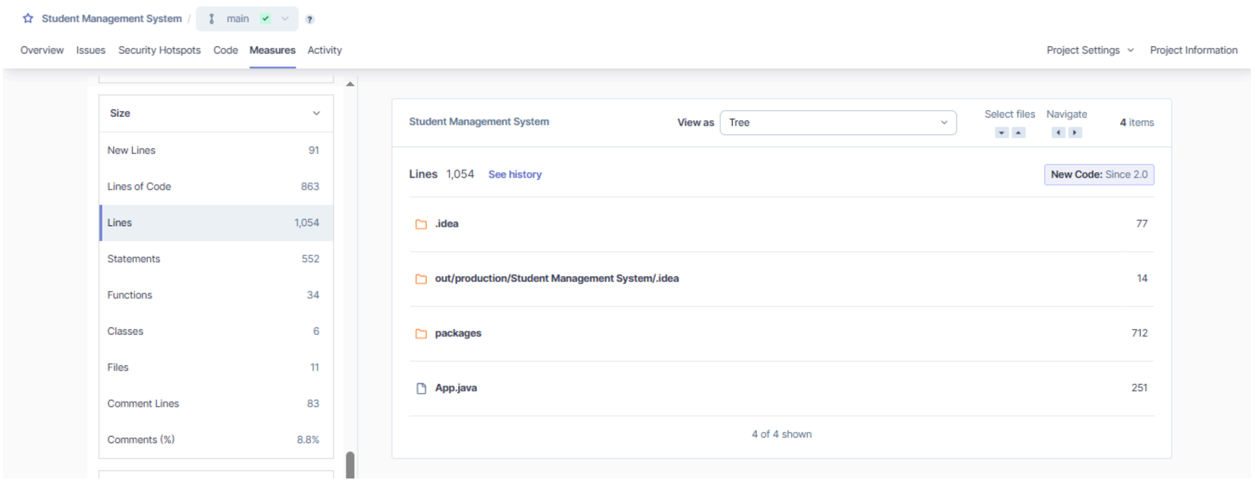
These 2 files—especially **`the second one`**–show a notable amount of repeated code. Seeing 14.5% duplication in a single file is a direct signal that some logic is being reused unnecessarily, which should ideally be refactored.

Metric	Value
Duplicated Blocks	2
Duplicated Files	2
Overall Duplication	3.9%

- In `StudentSystem.java`, it looks like some common operations are being repeated. These can be **pulled out into separate methods** to improve **readability** and reduce **redundancy**.
- Similarly, in `TxtFileHandler.java`, we may have similar patterns for reading/writing files. Instead of repeating code/operations, it's useful to have a helper method or a utility class if it fits.

Therefore, even though the duplication isn't very high overall, we shouldn't disregard it. The project will feel more organized and polished overall if the repetitive code in those files is cleaned up. It will also make things less messy and easier to manage.

2.1.6.6 Size


Student Management System / main

Overview Issues Security Hotspots Code Measures Activity — Project Settings — Project Information

Size	
New Lines	91
Lines of Code	863
Lines	1,054
Statements	552
Functions	34
Classes	6
Files	11
Comment Lines	83
Comments (%)	8.8%

Student Management System — View as Tree — Select files — Navigate — 4 items

Lines 1,054 See history — New Code: Since 2.0

.idea	77
out/production/Student Management System/.idea	14
packages	712
App.java	251

4 of 4 shown

This section gives us a quick snapshot of the overall codebase by showing various metrics.

Analysis:

Metrics	Value
Lines of Code (LoC)	812
Total Lines	1003
Statements	552
Functions	34
Classes	6
Files	12
Comment Lines	83
Comments (%)	9.3%

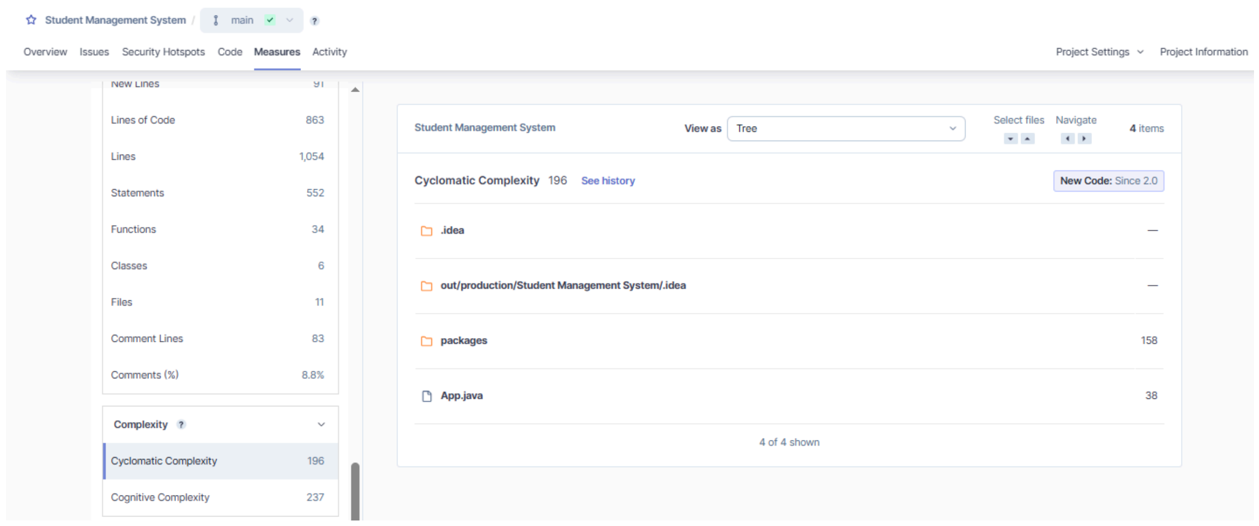
- The largest and most logically dense file is **StudentSystem.java**, which contains **273** lines of code, **199** statements, and **18** functions. This might indicate that it's taking on too much; perhaps some duties ought to be divided among different helper classes.

- Similarly, **App.java** has **202 LOC** and **151 statements**, which also suggests it handles quite a bit.

- Files like **Student.java** (16 LOC) and **CsvFileHandler.java** (75 LOC) are lightweight and modular, which is great.

We could restructure StudentSystem.java, decompose big methods, and perhaps transfer logic into smaller utility classes to make the system more streamlined and manageable. Future developers' comprehension may also be enhanced by including more insightful comments in strategic areas.

2.1.6.7 Complexity


Student Management System / main

New Lines	91
Lines of Code	863
Lines	1,054
Statements	552
Functions	34
Classes	6
Files	11
Comment Lines	83
Comments (%)	8.8%

Complexity

Cyclomatic Complexity	196
Cognitive Complexity	237

Student Management System — View as: Tree — Select files — Navigate — 4 items

Cyclomatic Complexity 196 See history — New Code: Since 2.0

.idea	—
out/production/Student Management System/.idea	—
packages	158
App.java	38

4 of 4 shown

This section explains how hard it is to read, comprehend, and maintain the code. It comes in two primary varieties:

1. **Cyclomatic Complexity:** Indicates how many independent paths—such as if, for, while, etc.—there are in the code.
2. **Cognitive Complexity:** Indicates how challenging it is to understand and follow the code mentally.

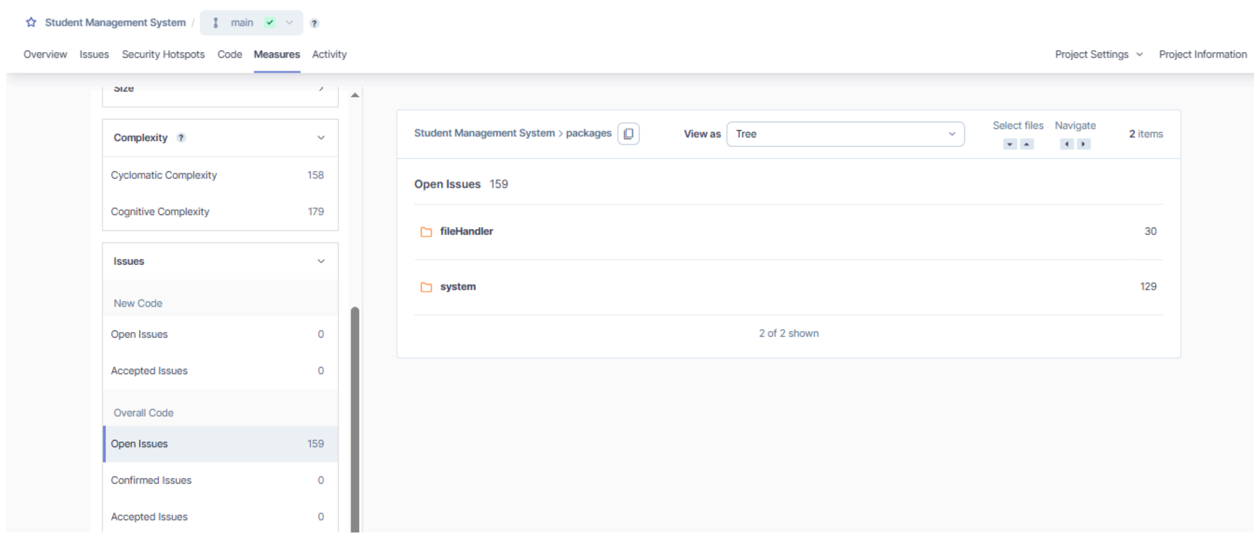
Our project has a total **Cyclomatic Complexity of 196** and a **Cognitive Complexity of 237**. These figures imply that although the logic is **not overly complicated overall**, some parts of the code (such as App.java and the files in packages/) could be made simpler to make the code easier to read.

<u>Complexity Overview Table</u>:

Metrics	Total	Packages/	App.java
Cyclomatic Complexity	196	158	38
Cognitive Complexity	237	179	58

- It's possible that App.java is managing too much logic in one location; we should think about refactoring by dividing big functions into more manageable, targeted ones.
- The high level of complexity within packages indicates that better separation of concerns and modularization of internal business logic (such as system flow or validations) would be beneficial.

2.1.6.8 Issues


Student Management System / main
Overview Issues Security Hotspots Code Measures Activity
Project Settings Project Information
Size
Select files Navigate

Complexity
Cyclomatic Complexity 158
Cognitive Complexity 179

Issues
New Code
Open Issues 0
Accepted Issues 0
Overall Code
Open Issues 159
Confirmed Issues 0
Accepted Issues 0

Student Management System > packages View as Tree 2 items

Open Issues 159

fileHandler	30
system	129

2 of 2 shown

We can quickly see where issues are in the codebase by looking at the Issues section. A total of 219 open issues were found. Fortunately, none of them fall under the security category, and the majority are related to maintainability and reliability.

The statistics make it evident that the majority of the problems are located in the packages folder, with the remaining occurring in App.java. The output and idea folders were found to be clean.

Issue Summary:

Areas	Issues
`packages/`	159
`App.java`	60
`.idea/ & build folders`	0
Total	219

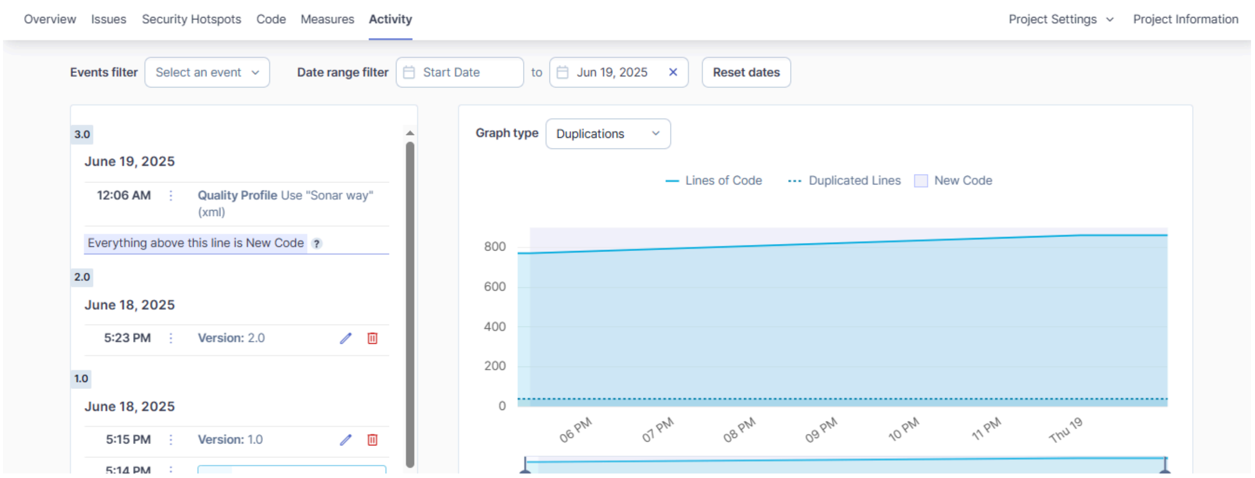
Issue Type:

Type	Count
Security	0
Reliability	8
Maintainability	211
Confirmed Issues	0
Accepted Issues	0
False Positives	0

Addressing the problems within packages, particularly in **StudentSystem.java** and related classes, should be our first priority. These most likely involve improper procedures, redundant code, or an absence of organization. We should follow good practices like modularization and appropriate naming, refactor, and eliminate redundancies.

The same is true for App.java; resolving those **60 bugs** will significantly enhance the quality of our code and lessen future difficulties.

2.1.7 Activity


Events filter: Select an event

Date range filter: Start Date to Jun 19, 2025

Graph type: Duplications

3.0
June 19, 2025
12:06 AM Quality Profile Use "Sonar way" (xml)
Everything above this line is New Code

2.0
June 18, 2025
5:23 PM Version: 2.0

1.0
June 18, 2025
5:15 PM Version: 1.0
5:14 PM

The Activity tab displays a graph of how selected attributes changed over time. There are a huge number of attributes to select from, for example: number of classes, cyclometric complexity, etc. There is also a date filter to help track changes over a specific period of time. Since the codebase has not been changed at the time of the report, there is not much change over time.

2.2 Abstract Syntax Tree (AST)

2.2.1 ASTs

An Abstract Syntax Tree (AST) is a hierarchical representation of source code, where nodes represent constructs (e.g., variables, loops, functions), and branches show relationships between them (e.g., a loop contains a condition).

ASTs abstract away unnecessary syntax like punctuation and whitespace, making them ideal for tasks such as static analysis, refactoring, and code transformations. They are commonly used in tools like linters, compilers, and IDEs for efficient code inspection and modification.

2.2.2 Static Analysis using ASTs

AST is a key tool for static analysis, which means checking code for problems without running it. It is useful for static analysis since:

- Helps find mistakes by showing exactly how the code is structured
- Makes it easier to spot problems like:
    - Missing pieces of code
    - Wrong use of variables
    - Potentially dangerous operations
- Powers tools that help write better code (like error checkers in IDEs)

Simple example - If you write if x = 5 instead of if x == 5, the AST shows this as an assignment (putting 5 into x) rather than a comparison, so the analyzer can flag it as a mistake.

2.2.3 PSI in IntelliJ IDEA

IntelliJ IDEA uses its own version of the abstract syntax tree called the Program Structure Interface (PSI). PSI is built on top of the AST but includes more detailed information, such as code meaning and structure. It also provides tools to easily work with parts of the code like classes, methods, and variables.

We used the *PSI Viewer plugin* to see this structure as a tree, similar to an AST, but with extra information that makes it easier to understand and use.

2.2.4 Setup

To view the PSI structure of Java files in IntelliJ IDEA, the PsiViewer plugin was used. By default, the View PSI Structure option may not appear in the Tools menu. The following steps enable and access it:

1. Enable Internal Mode:
    - Go to *Help > Edit* Custom Properties.
    - Add the following line:

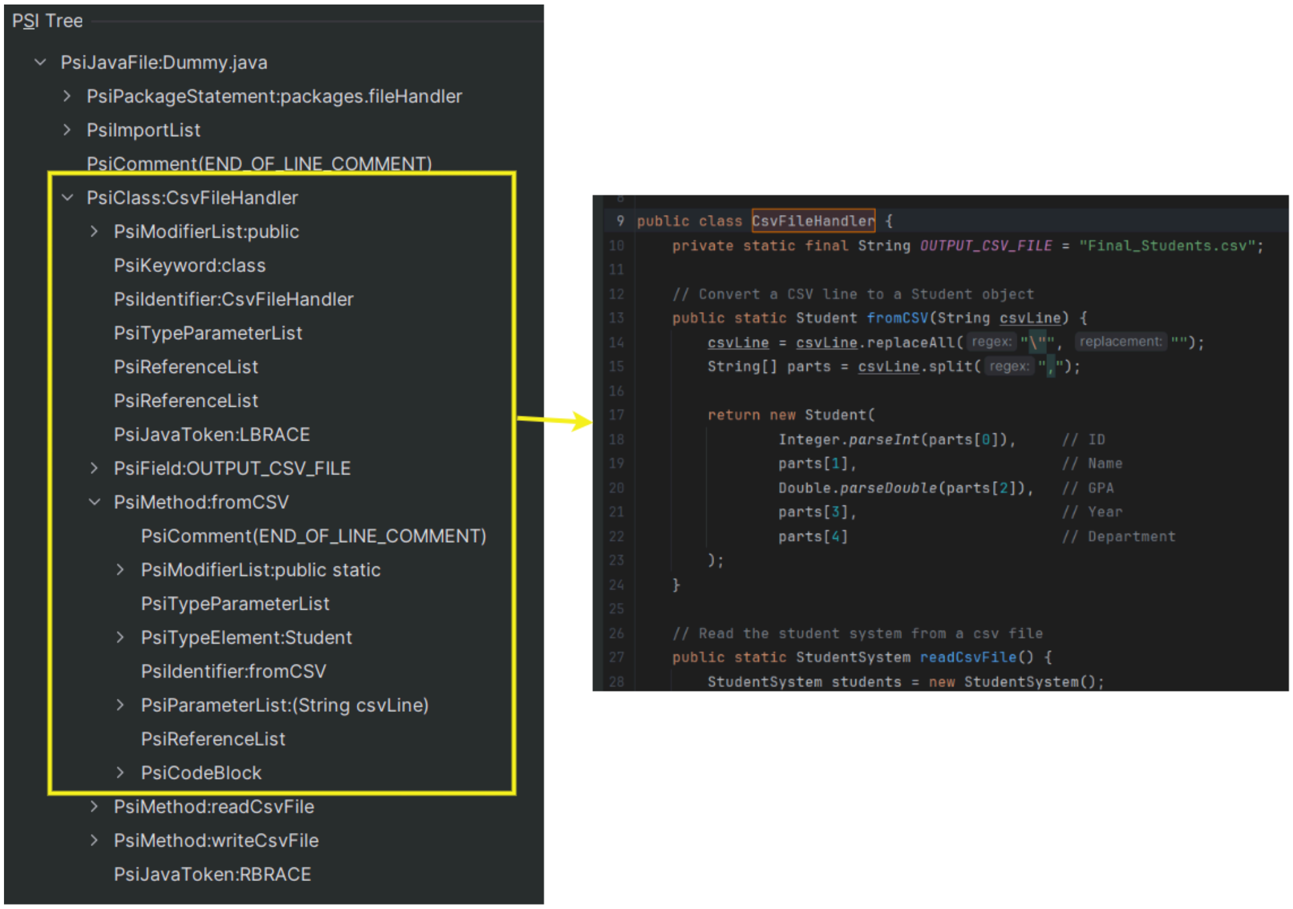
      ```
      idea.is.internal=true
      ```
    - Save the file and restart IntelliJ IDEA.
2. Install the Plugin (if not already available):
    - Navigate to *File > Settings > Plugins*, search for `PsiViewer`, and install it.
3. Access the PSI Tree:
    - Open any `.java` file.
    - Go to *Tools > View PSI Structure of Current File* (now visible after enabling internal mode)

or

- Right-click in the editor and select *View PSI Structure* (if shown)

2.2.5 Analysis

2.2.5.1 CsvFileHandler.java

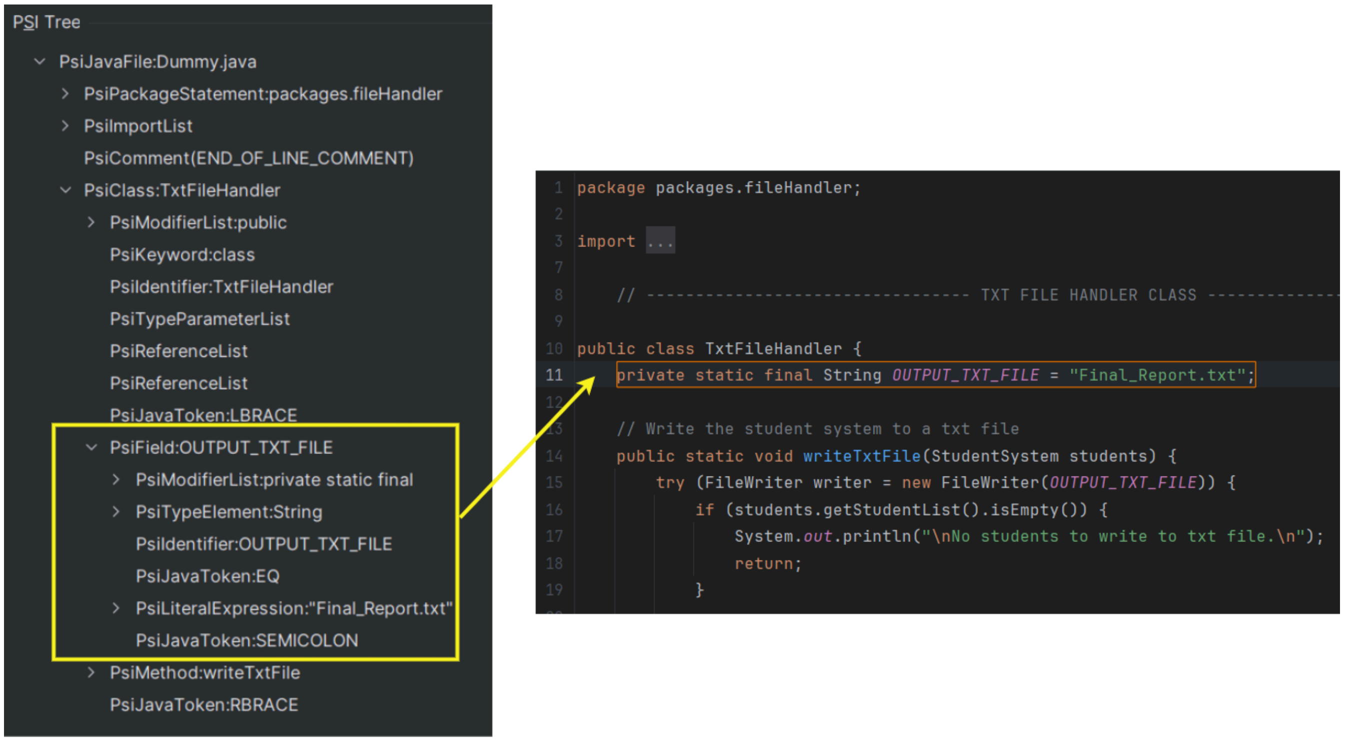


```
PSI Tree
  PsiJavaFile:Dummy.java
    PsiPackageStatement:packages.fileHandler
    PsiImportList
    PsiComment(END_OF_LINE_COMMENT)
    PsiClass:CsvFileHandler
      PsiModifierList:public
      PsiKeyword:class
      PsiIdentifier:CsvFileHandler
      PsiTypeParameterList
      PsiReferenceList
      PsiReferenceList
      PsiJavaToken:LBRACE
      PsiField:OUTPUT_CSV_FILE
      PsiMethod:fromCSV
        PsiComment(END_OF_LINE_COMMENT)
        PsiModifierList:public static
        PsiTypeParameterList
        PsiTypeElement:Student
        PsiIdentifier:fromCSV
        PsiParameterList:(String csvLine)
        PsiReferenceList
        PsiCodeBlock
      PsiMethod:readCsvFile
      PsiMethod:writeCsvFile
      PsiJavaToken:RBRACE
```

```java
public class CsvFileHandler {
    private static final String OUTPUT_CSV_FILE = "Final_Students.csv";

    // Convert a CSV line to a Student object
    public static Student fromCSV(String csvLine) {
        csvLine = csvLine.replaceAll( regex: "\"", replacement: "");
        String[] parts = csvLine.split( regex: ",");

        return new Student(
                Integer.parseInt(parts[0]),     // ID
                parts[1],                       // Name
                Double.parseDouble(parts[2]),   // GPA
                parts[3],                       // Year
                parts[4]                        // Department
        );
    }

    // Read the student system from a csv file
    public static StudentSystem readCsvFile() {
        StudentSystem students = new StudentSystem();
```

The figure above illustrates a section of the PSI tree (left) generated in IntelliJ IDEA for the `CsvFileHandler` class, alongside the corresponding highlighted source code (right). The PSI tree reveals the hierarchical structure of the class, including its modifiers, fields, and methods. Specifically, it expands the `fromCSV` method, showing detailed elements such as comments, parameter lists, return types, and code blocks. Due to the size of the codebase, only this portion is shown, demonstrating how PSI accurately reflects the internal structure of even complex Java files and supports precise code analysis.

2.2.5.2 TxtFileHandler.java

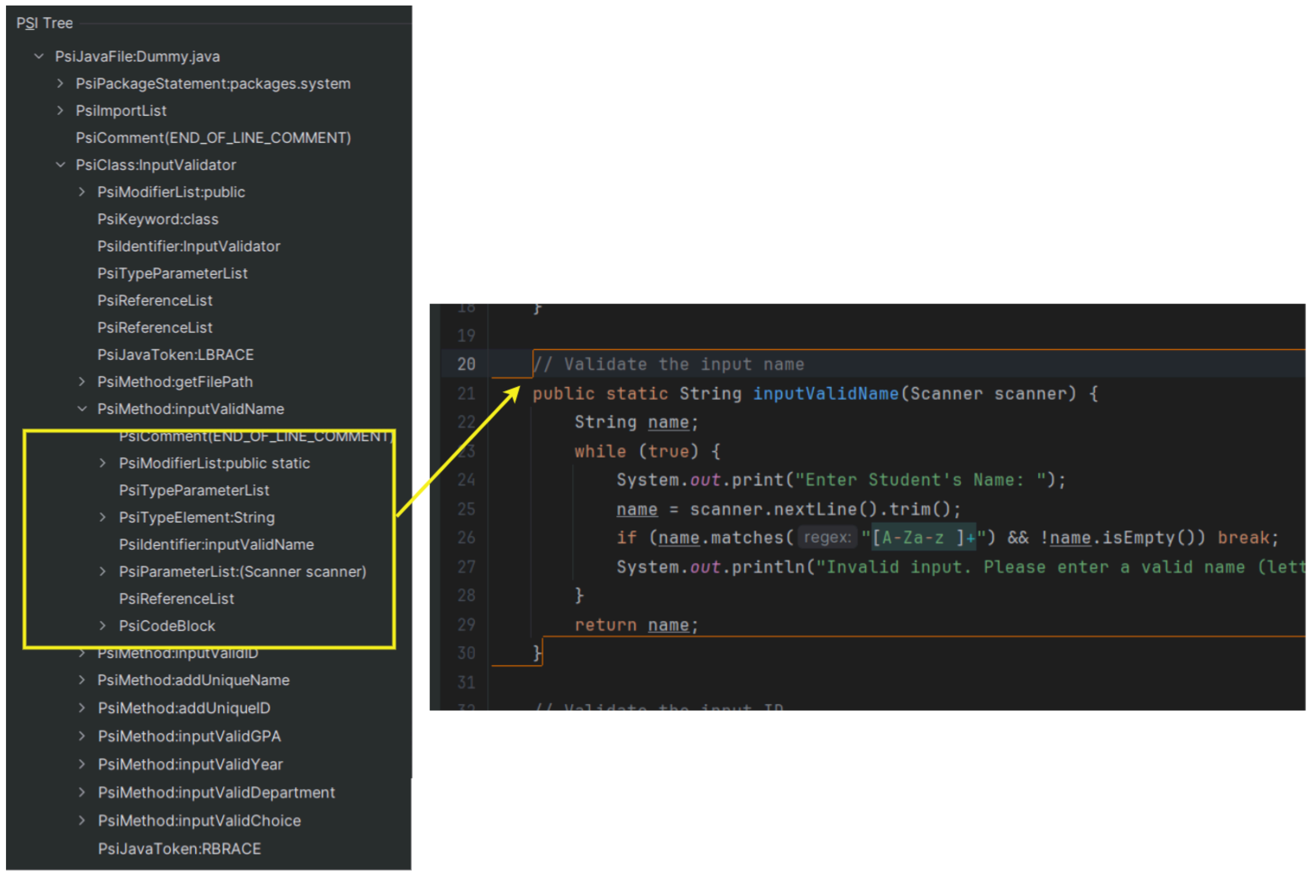


```
PSI Tree
  ⌄ PsiJavaFile:Dummy.java
    › PsiPackageStatement:packages.fileHandler
    › PsiImportList
      PsiComment(END_OF_LINE_COMMENT)
    ⌄ PsiClass:TxtFileHandler
      › PsiModifierList:public
        PsiKeyword:class
        PsiIdentifier:TxtFileHandler
        PsiTypeParameterList
        PsiReferenceList
        PsiReferenceList
        PsiJavaToken:LBRACE
      ⌄ PsiField:OUTPUT_TXT_FILE
        › PsiModifierList:private static final
        › PsiTypeElement:String
          PsiIdentifier:OUTPUT_TXT_FILE
          PsiJavaToken:EQ
        › PsiLiteralExpression:"Final_Report.txt"
          PsiJavaToken:SEMICOLON
      › PsiMethod:writeTxtFile
        PsiJavaToken:RBRACE
```

```java
package packages.fileHandler;

import ...

    // ------------------------------- TXT FILE HANDLER CLASS ------------

public class TxtFileHandler {
    private static final String OUTPUT_TXT_FILE = "Final_Report.txt";

    // Write the student system to a txt file
    public static void writeTxtFile(StudentSystem students) {
        try (FileWriter writer = new FileWriter(OUTPUT_TXT_FILE)) {
            if (students.getStudentList().isEmpty()) {
                System.out.println("\nNo students to write to txt file.\n");
                return;
            }
```

The image above presents a focused PSI tree segment for the `TxtFileHandler` class, with a highlighted portion of the corresponding source code. The PSI tree (left) breaks down the `OUTPUT_TXT_FILE` field declaration, identifying its modifiers (`private static final`), type (`String`), name (`OUTPUT_TXT_FILE`), and assigned literal value (`"Final_Report.txt"`). The code on the right confirms this structure, showing the exact line as it appears in the class. This demonstrates how PSI captures fine-grained elements of the code, enabling detailed analysis of individual declarations and their components.

2.2.5.3 InputValidator.java

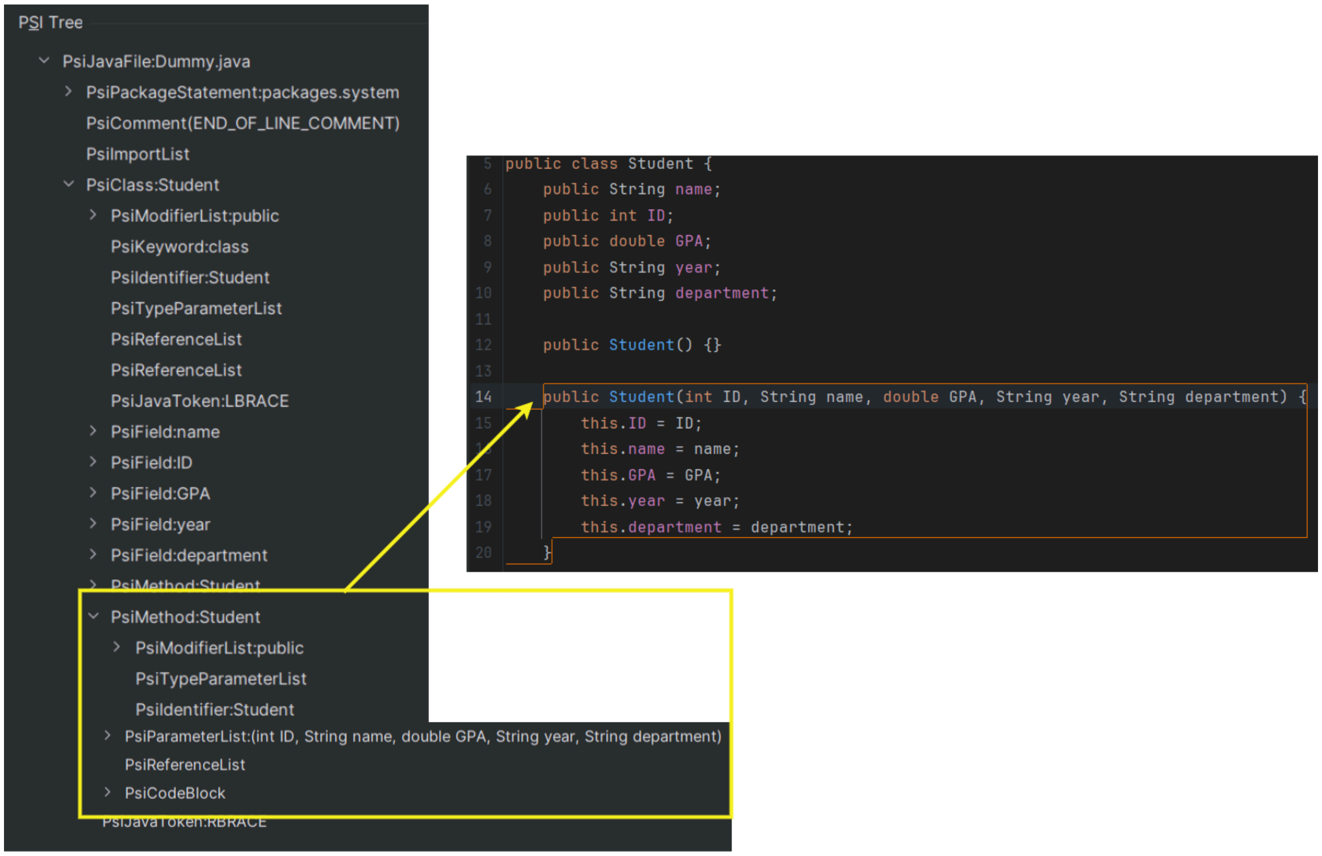


```
PSI Tree
  PsiJavaFile:Dummy.java
    PsiPackageStatement:packages.system
    PsiImportList
    PsiComment(END_OF_LINE_COMMENT)
    PsiClass:InputValidator
      PsiModifierList:public
      PsiKeyword:class
      PsiIdentifier:InputValidator
      PsiTypeParameterList
      PsiReferenceList
      PsiReferenceList
      PsiJavaToken:LBRACE
      PsiMethod:getFilePath
      PsiMethod:inputValidName
        PsiComment(END_OF_LINE_COMMENT)
        PsiModifierList:public static
        PsiTypeParameterList
        PsiTypeElement:String
        PsiIdentifier:inputValidName
        PsiParameterList:(Scanner scanner)
        PsiReferenceList
        PsiCodeBlock
      PsiMethod:inputValidID
      PsiMethod:addUniqueName
      PsiMethod:addUniqueID
      PsiMethod:inputValidGPA
      PsiMethod:inputValidYear
      PsiMethod:inputValidDepartment
      PsiMethod:inputValidChoice
      PsiJavaToken:RBRACE
```

```java
    }

    // Validate the input name
    public static String inputValidName(Scanner scanner) {
        String name;
        while (true) {
            System.out.print("Enter Student's Name: ");
            name = scanner.nextLine().trim();
            if (name.matches( regex: "[A-Za-z ]+") && !name.isEmpty()) break;
            System.out.println("Invalid input. Please enter a valid name (lett
        }
        return name;
    }

    // Validate the input ID
```

The image illustrates the PSI tree structure of the `InputValidator` class, focusing on the `inputValidName` method. On the left, the PSI tree decomposes the method into its constituent elements, including its visibility modifier (`public`), scope (`static`), return type (`String`), and parameters (`Scanner scanner`). The right side displays the corresponding source code, revealing the method's implementation, which validates user input by ensuring it contains only alphabetic characters and is non-empty.

2.2.5.4 Student.java

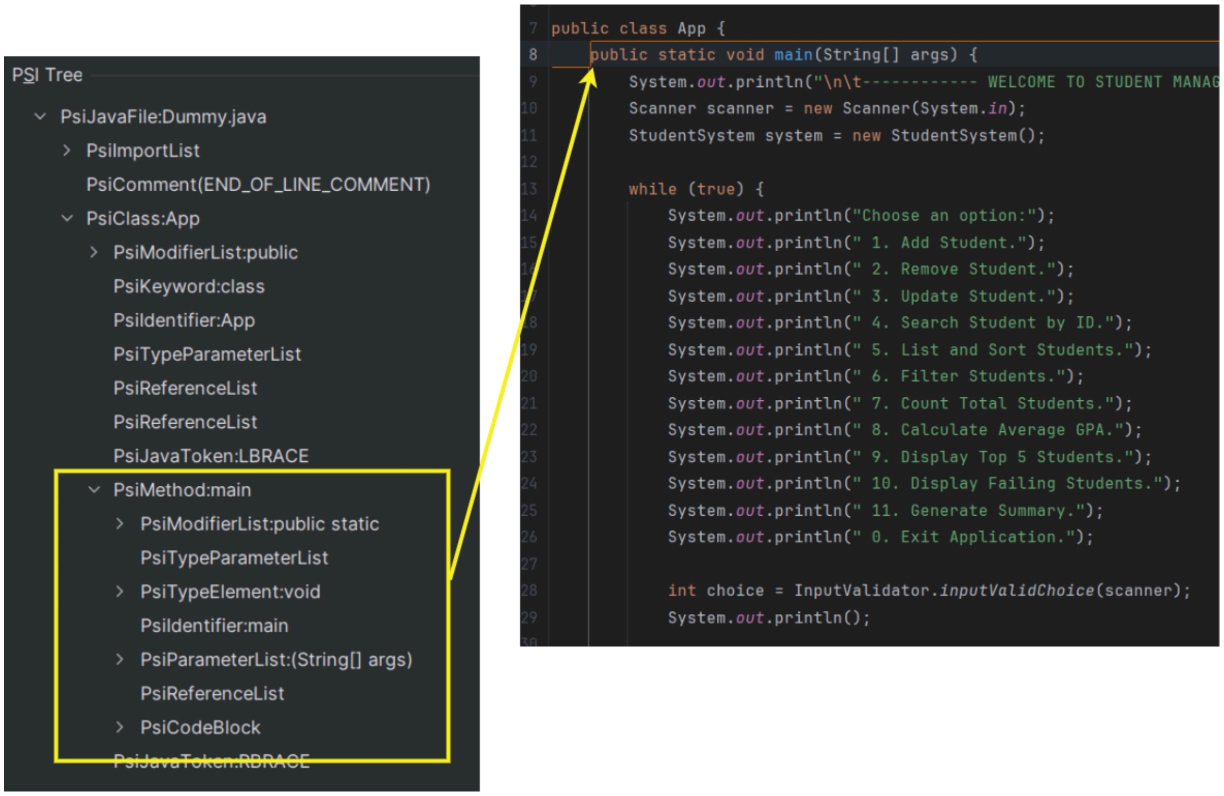


```
PSI Tree
  PsiJavaFile:Dummy.java
    PsiPackageStatement:packages.system
    PsiComment(END_OF_LINE_COMMENT)
    PsiImportList
    PsiClass:Student
      PsiModifierList:public
      PsiKeyword:class
      PsiIdentifier:Student
      PsiTypeParameterList
      PsiReferenceList
      PsiReferenceList
      PsiJavaToken:LBRACE
      PsiField:name
      PsiField:ID
      PsiField:GPA
      PsiField:year
      PsiField:department
      PsiMethod:Student
      PsiMethod:Student
        PsiModifierList:public
        PsiTypeParameterList
        PsiIdentifier:Student
        PsiParameterList:(int ID, String name, double GPA, String year, String department)
        PsiReferenceList
        PsiCodeBlock
      PsiJavaToken:RBRACE
```

```java
public class Student {
    public String name;
    public int ID;
    public double GPA;
    public String year;
    public String department;

    public Student() {}

    public Student(int ID, String name, double GPA, String year, String department) {
        this.ID = ID;
        this.name = name;
        this.GPA = GPA;
        this.year = year;
        this.department = department;
    }
```

The image presents the PSI tree structure of the `Student` class, showcasing its fields (`name, ID, GPA, year, department`) and constructors. The left side details the syntactic elements, such as modifiers (`public`), data types, and parameters, while the right side displays the corresponding code. This alignment demonstrates how PSI trees enable precise analysis of class structures, supporting tasks like code inspection and automated refactoring.

2.2.5.5 App.java



```
PSI Tree
⌄ PsiJavaFile:Dummy.java
  › PsiImportList
    PsiComment(END_OF_LINE_COMMENT)
  ⌄ PsiClass:App
    › PsiModifierList:public
      PsiKeyword:class
      PsiIdentifier:App
      PsiTypeParameterList
      PsiReferenceList
      PsiReferenceList
      PsiJavaToken:LBRACE
    ⌄ PsiMethod:main
      › PsiModifierList:public static
        PsiTypeParameterList
      › PsiTypeElement:void
        PsiIdentifier:main
      › PsiParameterList:(String[] args)
        PsiReferenceList
      › PsiCodeBlock
      PsiJavaToken:RBRACE
```

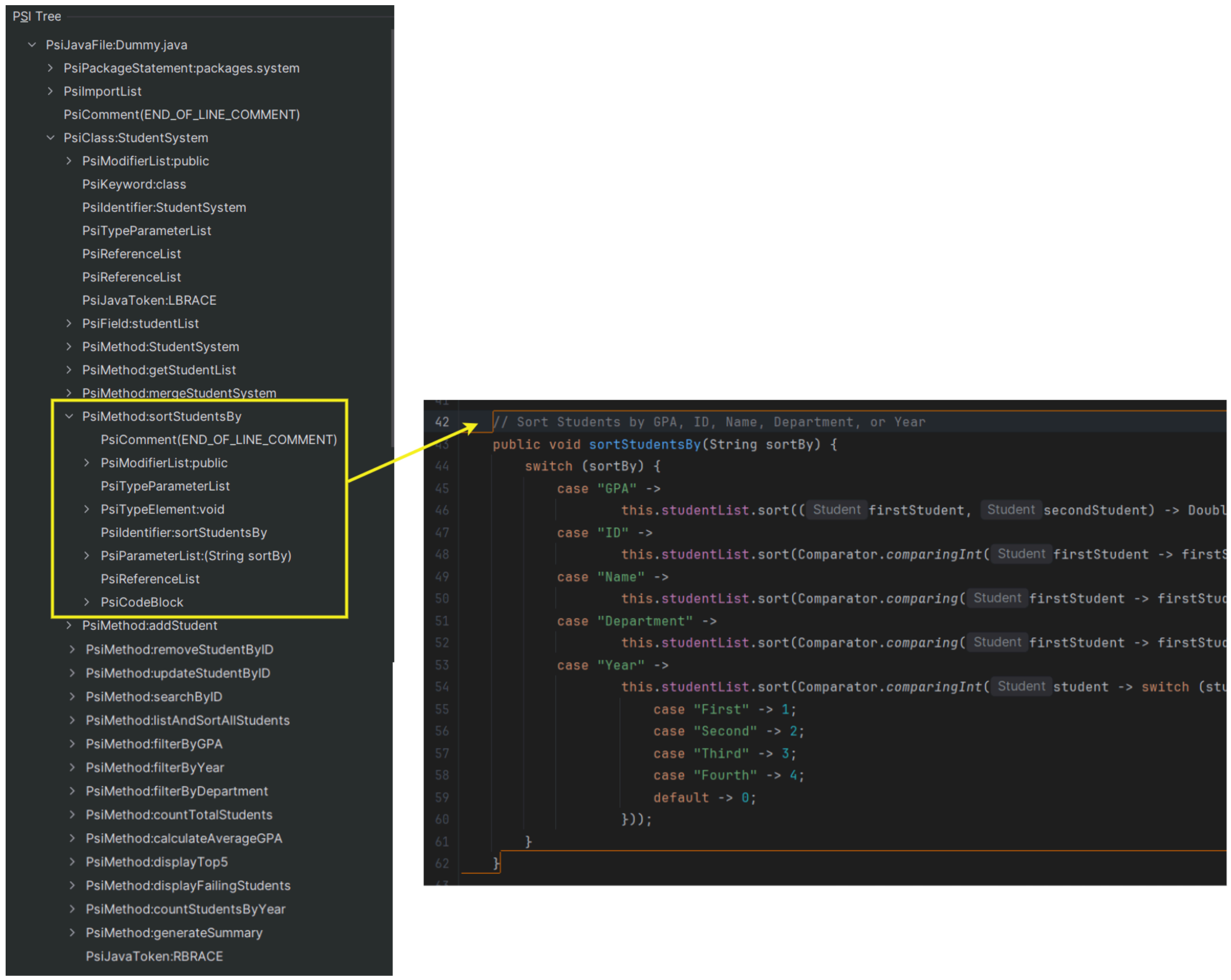
```java
public class App {
    public static void main(String[] args) {
        System.out.println("\n\t------------ WELCOME TO STUDENT MANAG
        Scanner scanner = new Scanner(System.in);
        StudentSystem system = new StudentSystem();

        while (true) {
            System.out.println("Choose an option:");
            System.out.println(" 1. Add Student.");
            System.out.println(" 2. Remove Student.");
            System.out.println(" 3. Update Student.");
            System.out.println(" 4. Search Student by ID.");
            System.out.println(" 5. List and Sort Students.");
            System.out.println(" 6. Filter Students.");
            System.out.println(" 7. Count Total Students.");
            System.out.println(" 8. Calculate Average GPA.");
            System.out.println(" 9. Display Top 5 Students.");
            System.out.println(" 10. Display Failing Students.");
            System.out.println(" 11. Generate Summary.");
            System.out.println(" 0. Exit Application.");

            int choice = InputValidator.inputValidChoice(scanner);
            System.out.println();
```

The screenshot displays the PSI tree structure of the `App` class, specifically highlighting its `main` method. On the left side, the tree decomposes the method into key elements:

- Access modifiers (`public static`)
- Return type (`void`)
- Method name (`main`)
- Input parameters (`String[] args`)

The corresponding source code on the right shows a console-based student management system featuring a menu with various operations like student addition, removal, and statistical analysis. This visualization effectively demonstrates how PSI trees provide structural insights into Java methods, supporting both code analysis and development tasks. The clear alignment between the PSI representation and actual implementation makes it particularly valuable for program comprehension and maintenance.

2.2.5.6 StudentSystem.java



```
PSI Tree
∨ PsiJavaFile:Dummy.java
    > PsiPackageStatement:packages.system
    > PsiImportList
      PsiComment(END_OF_LINE_COMMENT)
    ∨ PsiClass:StudentSystem
        > PsiModifierList:public
          PsiKeyword:class
          PsiIdentifier:StudentSystem
          PsiTypeParameterList
          PsiReferenceList
          PsiReferenceList
          PsiJavaToken:LBRACE
        > PsiField:studentList
        > PsiMethod:StudentSystem
        > PsiMethod:getStudentList
        > PsiMethod:mergeStudentSystem
        ∨ PsiMethod:sortStudentsBy
              PsiComment(END_OF_LINE_COMMENT)
            > PsiModifierList:public
              PsiTypeParameterList
            > PsiTypeElement:void
              PsiIdentifier:sortStudentsBy
            > PsiParameterList:(String sortBy)
              PsiReferenceList
            > PsiCodeBlock
        > PsiMethod:addStudent
        > PsiMethod:removeStudentByID
        > PsiMethod:updateStudentByID
        > PsiMethod:searchByID
        > PsiMethod:listAndSortAllStudents
        > PsiMethod:filterByGPA
        > PsiMethod:filterByYear
        > PsiMethod:filterByDepartment
        > PsiMethod:countTotalStudents
        > PsiMethod:calculateAverageGPA
        > PsiMethod:displayTop5
        > PsiMethod:displayFailingStudents
        > PsiMethod:countStudentsByYear
        > PsiMethod:generateSummary
          PsiJavaToken:RBRACE
```

```java
// Sort Students by GPA, ID, Name, Department, or Year
public void sortStudentsBy(String sortBy) {
    switch (sortBy) {
        case "GPA" ->
                this.studentList.sort(( Student firstStudent,  Student secondStudent) -> Doubl
        case "ID" ->
                this.studentList.sort(Comparator.comparingInt( Student firstStudent -> firstS
        case "Name" ->
                this.studentList.sort(Comparator.comparing( Student firstStudent -> firstStu
        case "Department" ->
                this.studentList.sort(Comparator.comparing( Student firstStudent -> firstStu
        case "Year" ->
                this.studentList.sort(Comparator.comparingInt( Student student -> switch (stu
                    case "First" -> 1;
                    case "Second" -> 2;
                    case "Third" -> 3;
                    case "Fourth" -> 4;
                    default -> 0;
                }));
    }
}
```

This analyzes the `StudentSystem` class through its PSI tree representation, with particular focus on the `sortStudentsBy` method. The left panel's hierarchical breakdown reveals:

- Method visibility (`public`)
- Return type (`void`)
- Parameter (`String sortBy`)

The right panel displays the corresponding implementation - a flexible sorting mechanism that organizes students by `GPA`, `ID`, `name`, `department`, or `year` using comparator logic. The PSI tree's precise mapping of method signatures to their implementations proves particularly valuable for understanding complex class behaviors, as demonstrated by this multi-criteria sorting functionality essential for student data management. This structural analysis capability makes PSI trees indispensable for both static code analysis and maintenance tasks in object-oriented systems.

3 Dynamic Analysis

3.1 VisualVM

VisualVM is a JAVA profiler tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine (JVM). VisualVM organizes JVM data that it gets from the Java Development Kit (JDK) tools and then shows the information in a way that allows data on multiple Java applications to be quickly viewed. This is possible for both locally and applications that are remotely running. VisualVM is pretty modular and easy to use extensively with the help of plugins. Here we are using the **IntelliJ IDEA** alongside the VisualVM plugin to conduct the dynamic analysis.

3.1.1 Installation and Setup

Prerequisites for the setup

- Java project
- IntelliJ IDEA Installed

Step 1: Download the VisualVM

First, we need to download the tool from their official website - VisualVM. A compressed folder will be downloaded. We then need to extract it.

Step 2: Get the VisualVM Launcher plugin for IntelliJ IDEA IDE

Based on the IDE version that we are using we would have to download the compatible plugin. It can be found on the Jetbrains Marketplace.

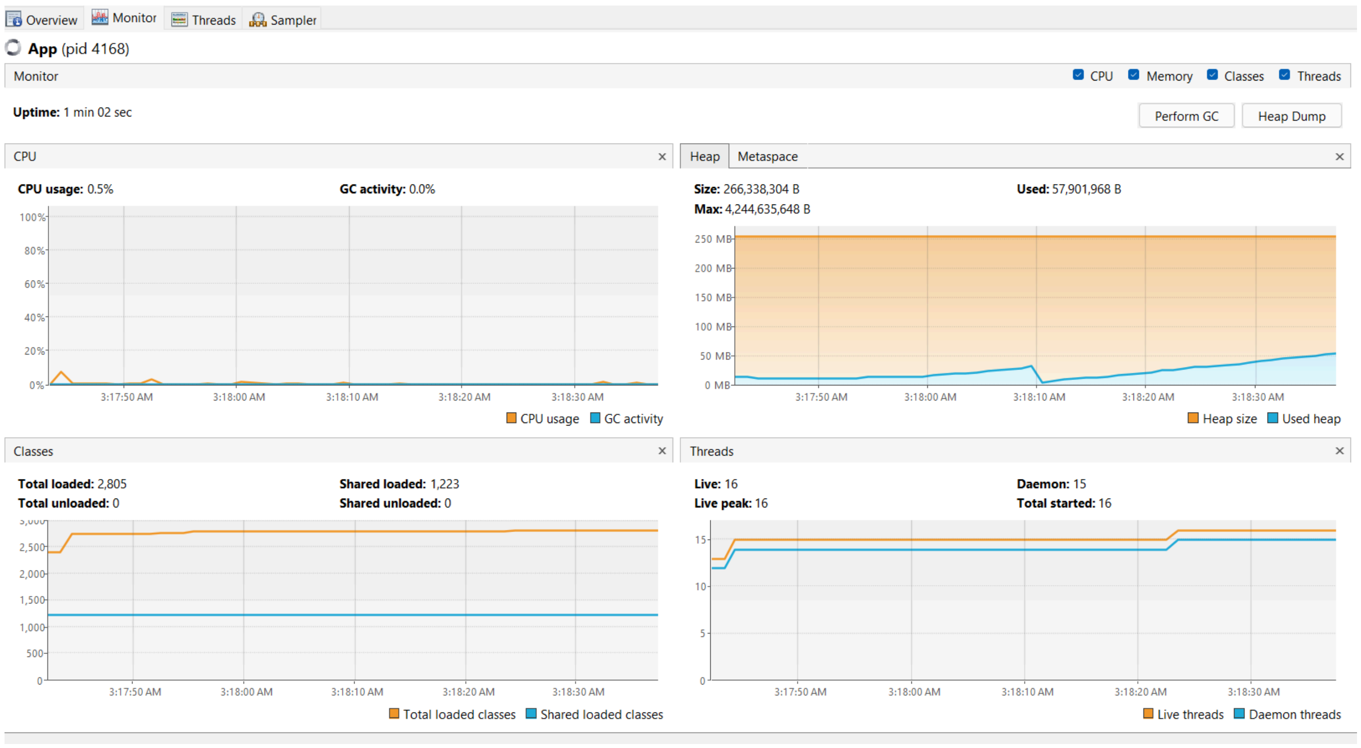
Step 3: Add the plugin to the IDE

Now we open up the Intellij Idea IDE. Then follow these steps:
Settings → Plugins → Setting icon → Install plugin from disk → Find the downloaded compressed plugin folder and then add it
After that, the IDE will ask the path of VisualVM's executable file. We would have to browse the correct path and add that as well, then press OK. Then we're good to go!

3.1.2 Overview

Dynamic analysis is the process of tracking what is happening during program execution. It is a method of testing, evaluating the program while it is running. This process helps us to identify if there's any memory leaks or not, it also helps us figure out concurrency issues in the code.

To perform dynamic analysis on the code we would need to run the code via the VisualVM plugin. The VisualVM automatically displays the application that we initiated - the application we intended to do dynamic analysis on. The page looks like this:



App (pid 4168)

Monitor

Uptime: 1 min 02 sec

CPU

CPU usage: 0.5% GC activity: 0.0%

Size: 266,338,304 B Used: 57,901,968 B
Max: 4,244,635,648 B

Classes

Total loaded: 2,805 Shared loaded: 1,223
Total unloaded: 0 Shared unloaded: 0

Threads

Live: 16 Daemon: 15
Live peak: 16 Total started: 16

3.1.3 Metrics Covered

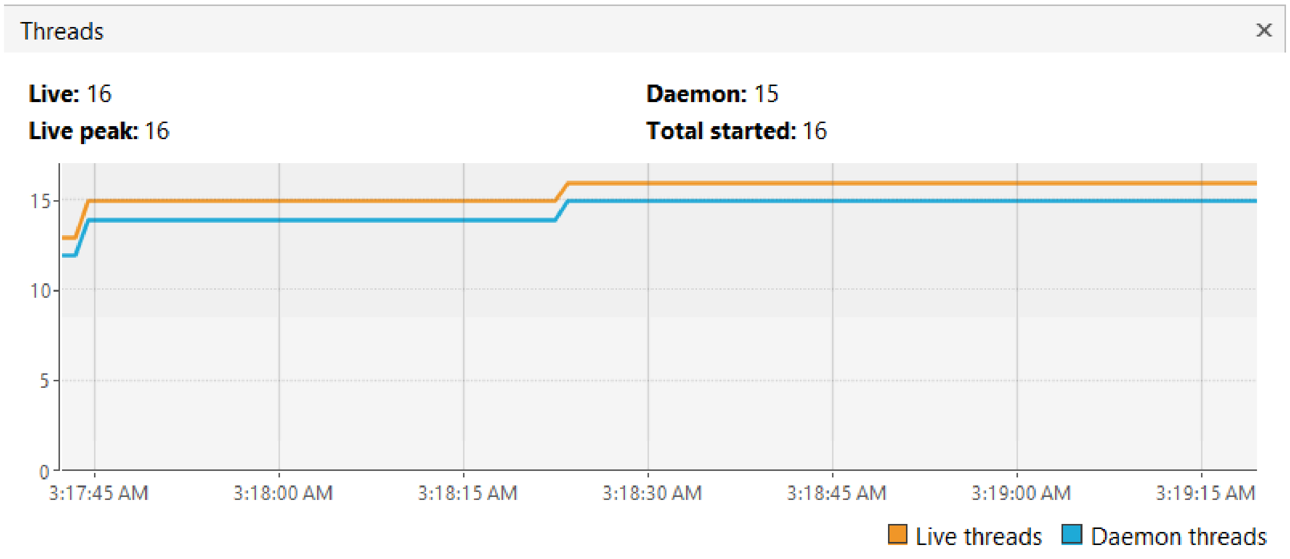
Here is the structured interpretation of the dynamic analysis that was conducted via VisualVM, with each monitored metric connected to its topic of dynamic analysis. Then a conclusion is added based on the observed behaviour.

3.1.3.1 Thread Activity

Here we observe the **Concurrency** of the program and Thread Management

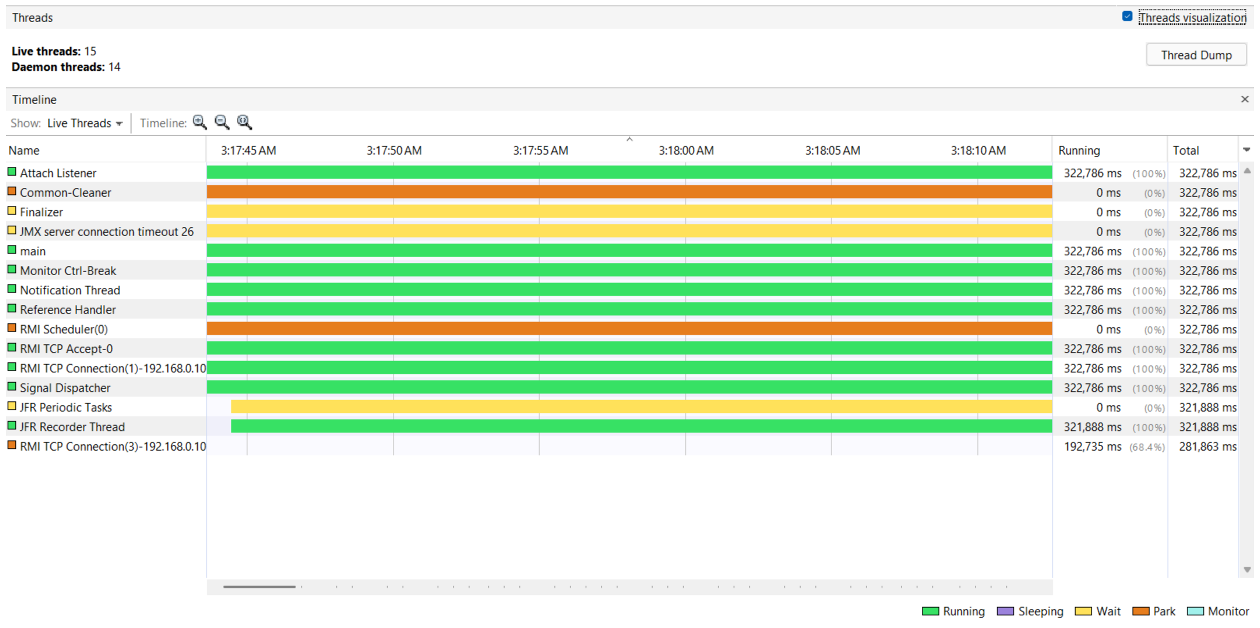
Peak Live Threads	Peak Live Daemon Threads	Total Started Threads
16	15	20

Gradual increase over time from 12 → 13 → 16 → 14



Threads

Live: 16
Live peak: 16
Daemon: 15
Total started: 16



Threads

Threads visualization

Live threads: 15
Daemon threads: 14

Thread Dump

Timeline

Show: Live Threads | Timeline:

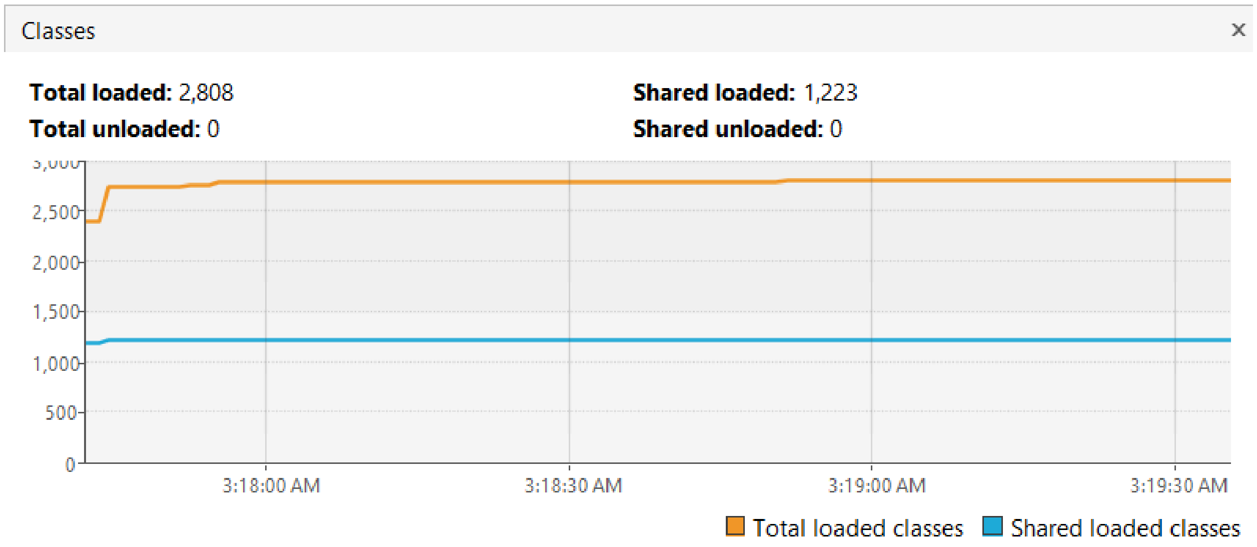
Name	Running		Total
🟩 Attach Listener	322,786 ms	(100%)	322,786 ms
🟧 Common-Cleaner	0 ms	(0%)	322,786 ms
🟨 Finalizer	0 ms	(0%)	322,786 ms
🟨 JMX server connection timeout 26	0 ms	(0%)	322,786 ms
🟩 main	322,786 ms	(100%)	322,786 ms
🟩 Monitor Ctrl-Break	322,786 ms	(100%)	322,786 ms
🟩 Notification Thread	322,786 ms	(100%)	322,786 ms
🟩 Reference Handler	322,786 ms	(100%)	322,786 ms
🟧 RMI Scheduler(0)	0 ms	(0%)	322,786 ms
🟩 RMI TCP Accept-0	322,786 ms	(100%)	322,786 ms
🟩 RMI TCP Connection(1)-192.168.0.10	322,786 ms	(100%)	322,786 ms
🟩 Signal Dispatcher	322,786 ms	(100%)	322,786 ms
🟨 JFR Periodic Tasks	0 ms	(0%)	321,888 ms
🟩 JFR Recorder Thread	321,888 ms	(100%)	321,888 ms
🟧 RMI TCP Connection(3)-192.168.0.10	192,735 ms	(68.4%)	281,863 ms

The application initializes threads as per need, peaking at 16 with almost all of them being daemon threads. Since there were no sudden spikes it can be said that it was following a stable multi-threading model with no thread leaks or runaway thread creation.

3.1.3.2 Class and Shader Loading

In this observation, we address the class loading and memory footprint phenomena.

Class Loaded	Shaders Loaded	Unloaded Classes	Unloaded Shaders
2808	1223	0	0

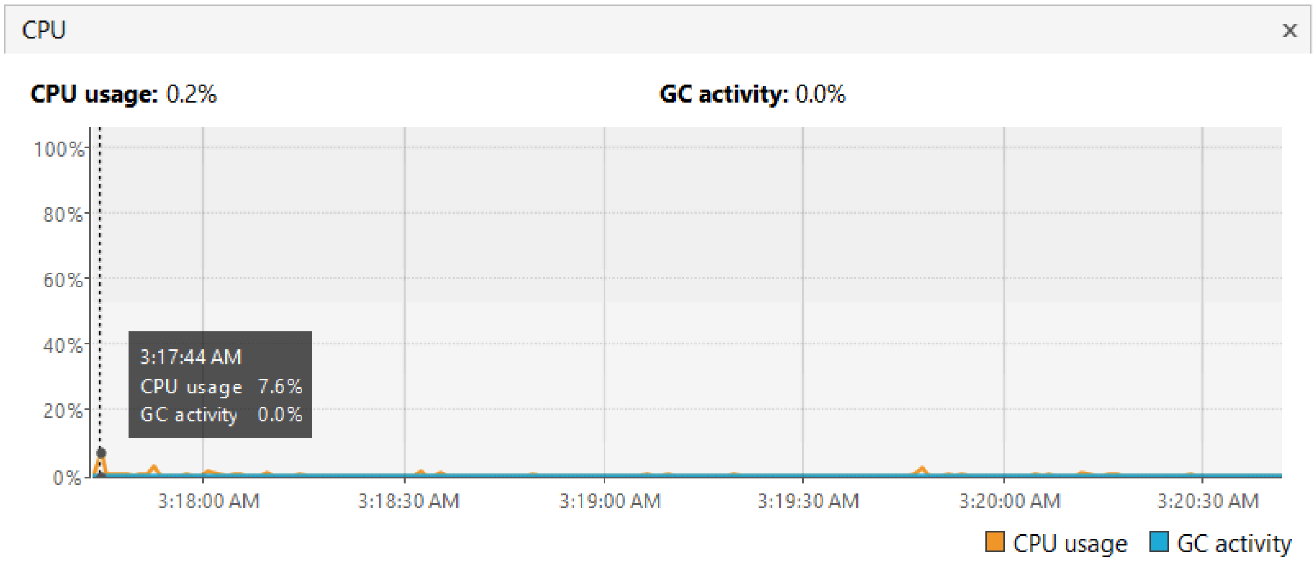


Loading of the Classes is within expected bounds for this application. The absence of class unloading shows that either minimal dynamic class loading/unloading or the classes remained in memory throughout execution.

3.1.3.3 CPU Usage

This section addresses Processor Utilization and Performance Efficiency. The observation:

- Initial Peak CPU Usage: 7.6% (first 10 seconds)
- Then stabilized CPU Usage, staying under 0.5%
- Occasional short-lived spikes



The application does some light computation at the start - during the initialization stage. Then it remains largely idle or performs minimal background processing. The short lived CPU spikes show periodic operations or event-driven triggers that are quickly handled. All in all, the CPU usage indicates efficient resource usage with no performance bottlenecks.

3.1.3.4 Garbage Collection (GC) Activity

Here we observe the Memory Management and GC Behavior. Core observation:

- GC Activity was 0% throughout runtime

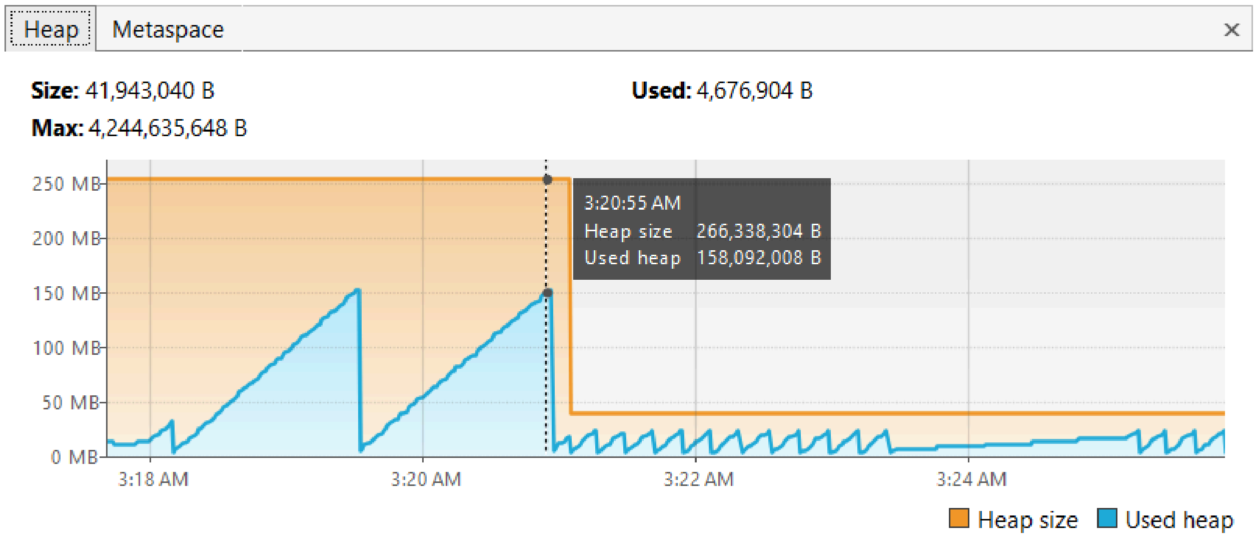
That would mean that GC was probably not triggered due to low memory pressure, or the minor GC cycles were too quick to register. This also confirms that the application maintained a small memory footprint, ensuring efficient memory use.

3.1.3.5 Heap Memory Usage

The next topic that was observed was the Memory Utilization and Object Lifecycles

Max Heap Size	Peak Used Heap	Steady Used Heap	Stead Max Head
250 MB	158 MB	20 MB	50 MB

The application initially allocates memory for the setup phase or caching, but later the memory usage drops and remains stable. This suggests effective release of memory. The heap reduction to 50MB indicates that the memory allocation was adjusted because of low demand for consistent memory.



Size: 41,943,040 B **Used:** 4,676,904 B
Max: 4,244,635,648 B

3:20:55 AM
Heap size 266,338,304 B
Used heap 158,092,008 B

3.2 Code Coverage For Java

IntelliJ IDEA has a default plugin bundled with it which analyzes which lines of code were executed during a particular run.

3.2.1 Setup Guide

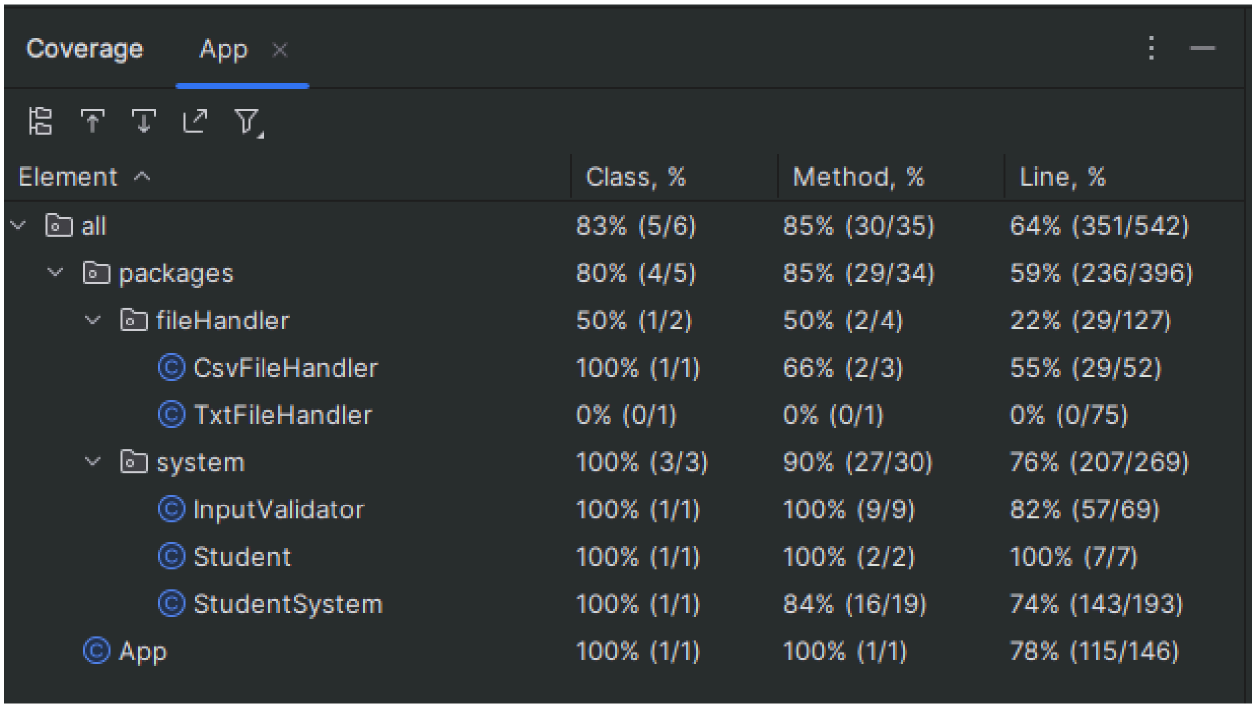
Unless disabled, it is usually enabled in the IntelliJ IDEA. This can be ensured by navigating to:
File → Settings → Plugins → Installed

By clicking on the Code Coverage, it can be checked whether it is enabled or not.

3.2.2 Execution

After ensuring that the plugin is enabled, any specific file can be run with Coverage monitoring it by clicking on the three dots on the right side of the run and then selecting "Run … with Coverage". Since this application only has one executable file App.java with a main function, it was executed and different functionalities were performed to see the code coverage. The run attempted to mimic the flow of an usual user working with the application while trying to ensure all features were explored.

3.2.3 Results



Element	Class, %	Method, %	Line, %
all	83% (5/6)	85% (30/35)	64% (351/542)
packages	80% (4/5)	85% (29/34)	59% (236/396)
fileHandler	50% (1/2)	50% (2/4)	22% (29/127)
CsvFileHandler	100% (1/1)	66% (2/3)	55% (29/52)
TxtFileHandler	0% (0/1)	0% (0/1)	0% (0/75)
system	100% (3/3)	90% (27/30)	76% (207/269)
InputValidator	100% (1/1)	100% (9/9)	82% (57/69)
Student	100% (1/1)	100% (2/2)	100% (7/7)
StudentSystem	100% (1/1)	84% (16/19)	74% (143/193)
App	100% (1/1)	100% (1/1)	78% (115/146)

The above results pops up after the run is completed, which shows:

- Percentage of classes out of total classes
- Percentage of methods out of total methods
- Percentage of lines out of total lines

that was covered during this particular run.

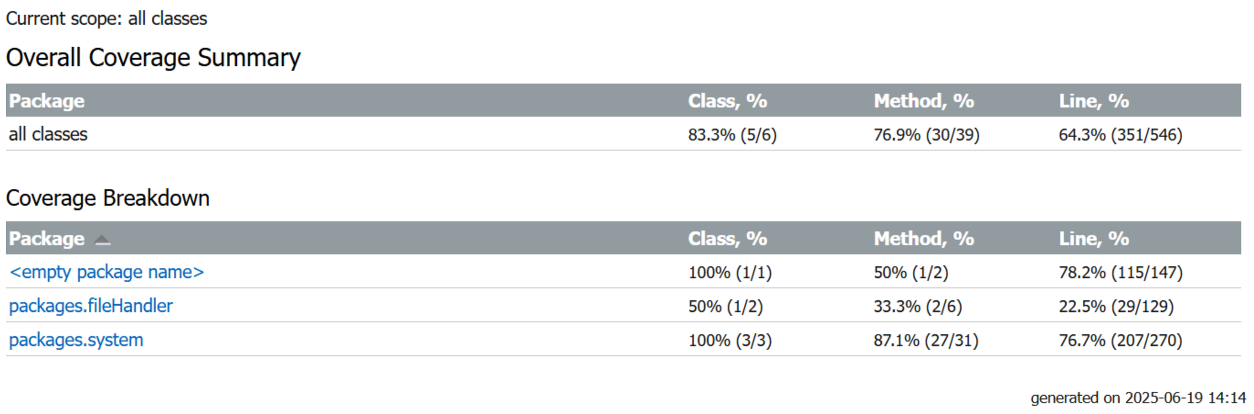
File	Class	Method	Line
`StudentSystem.java`	100% (1/1)	84% (16/19)	79% (143/193)
`App.java`	100% (1/1)	100% (1/1)	78% (115/146)
`InputValidator.java`	100% (1/1)	100% (9/9)	82% (72/69)
`TxtFileHandler.java`	0% (0/1)	0% (0/1)	0% (0/75)
`CsvFileHandler.java`	100% (1/1)	66% (2/3)	55% (29/52)
`Student.java`	100% (1/1)	100% (2/2)	100% (7/7)

From the results, it can be seen that most of the files have high coverage except for CsvFileHandler.java and TxtFileHandler.java. This is likely because during the run, instead of opting for “write to a file” in case of the features Generate Report, List and Sort Students, and Add Student, the manual input method was chosen. CsvFileHandler was used for reading the students from the list stored in CSV during the operation of generating a student list. This implies that the user prefers manual input over operating on files. The Student class was used because all of the features were tested during the run, for example: adding, removing, printing a list, etc.

Most other files show high coverage during application, but not 100%. A deeper explanation can be found using the report generated by Export Report option.

3.2.4 Report

The diagram below shows a snapshot of how the report looks. It is an HTML file containing a similar level of detail as the pop-up window. However, each of the individual files can be expanded to see which lines were not covered to gain more insight into which parts of the application might not be used. The current scope is shown at the top and the date of the generated report is attached at the bottom.



Current scope: all classes

Overall Coverage Summary

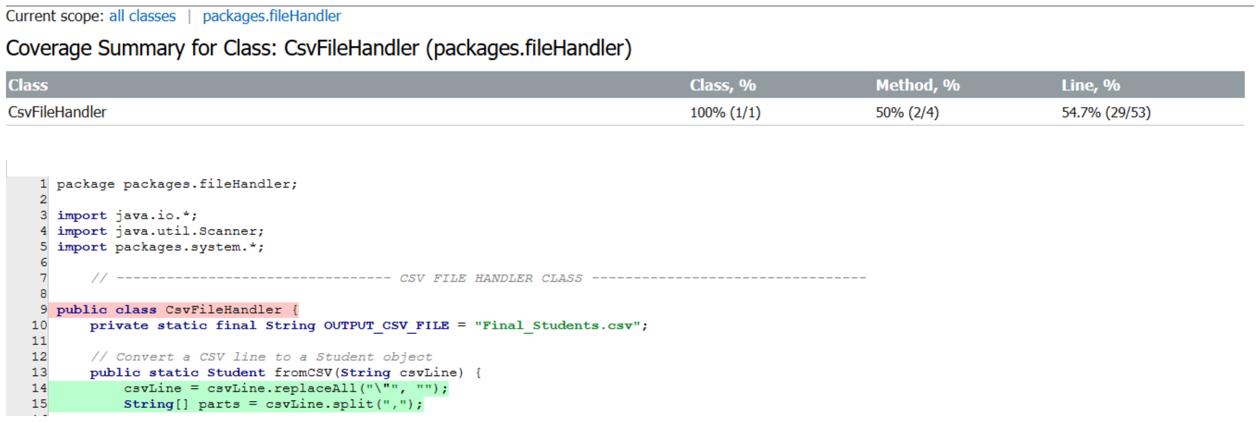
Package	Class, %	Method, %	Line, %
all classes	83.3% (5/6)	76.9% (30/39)	64.3% (351/546)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
<empty package name>	100% (1/1)	50% (1/2)	78.2% (115/147)
packages.fileHandler	50% (1/2)	33.3% (2/6)	22.5% (29/129)
packages.system	100% (3/3)	87.1% (27/31)	76.7% (207/270)

generated on 2025-06-19 14:14

When any specific file is selected, the current scope changes, and lines are marked by their usage. As the diagram below shows, the current scope is the fileHandler packer since the CsvFileHandler.java was opened. The red lines show the lines that were not executed during this particular run, and the green lines show the lines that were executed.



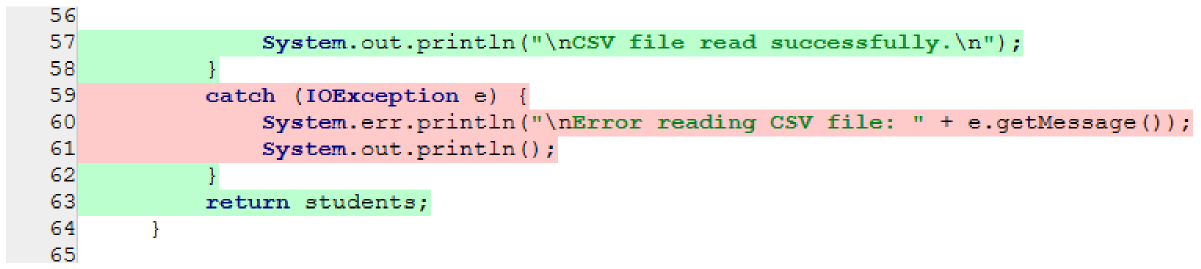
Current scope: all classes | packages.fileHandler

Coverage Summary for Class: CsvFileHandler (packages.fileHandler)

Class	Class, %	Method, %	Line, %
CsvFileHandler	100% (1/1)	50% (2/4)	54.7% (29/53)

```java
package packages.fileHandler;

import java.io.*;
import java.util.Scanner;
import packages.system.*;

    // -------------------------------- CSV FILE HANDLER CLASS --------------------------------

public class CsvFileHandler {
    private static final String OUTPUT_CSV_FILE = "Final_Students.csv";

    // Convert a CSV line to a Student object
    public static Student fromCSV(String csvLine) {
        csvLine = csvLine.replaceAll("\"", "");
        String[] parts = csvLine.split(",");
```

If looked at in detail, it provides an idea of why the code coverage of this particular file was not 100% - the IOException was not executed because the application did not face any such error.



```java

                System.out.println("\nCSV file read successfully.\n");
            }
            catch (IOException e) {
                System.err.println("\nError reading CSV file: " + e.getMessage());
                System.out.println();
            }
            return students;
        }

```

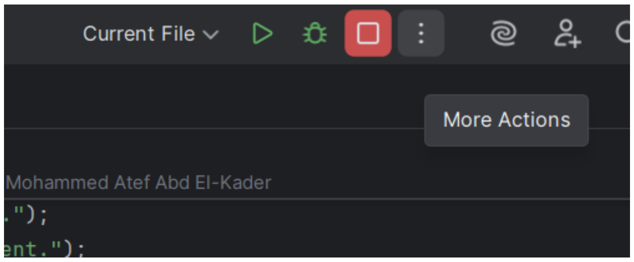
3.3 IntelliJ IDEA Profiler

The IntelliJ IDEA Profiler, available in the Ultimate edition, is a built-in tool for analyzing the runtime performance of Java applications. It helps identify performance bottlenecks such as slow methods, memory leaks, and excessive CPU usage. The profiler offers visual insights like flame graphs, call trees, and method timelines, making it easier to optimize code. It integrates seamlessly with the IDE, allowing developers to profile code directly from the editor without external tools.

3.3.1 Setup

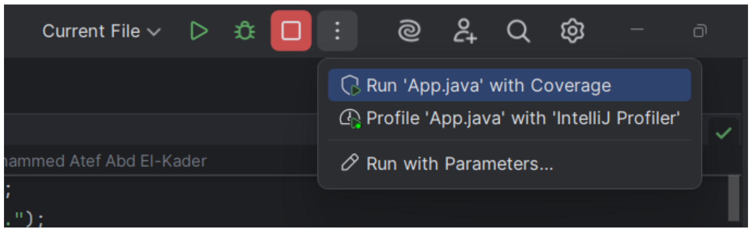
To analyze the runtime performance of a Java application using the IntelliJ IDEA Profiler (Ultimate Edition), follow these steps:

1. Open the Target Application
    - Open the Java project or application you want to analyze in IntelliJ IDEA
2. Access the Profiler Option
    - In the top-right corner of the IDE, next to the Run and Stop buttons, the vertical three-dot menu was clicked to access additional run options.

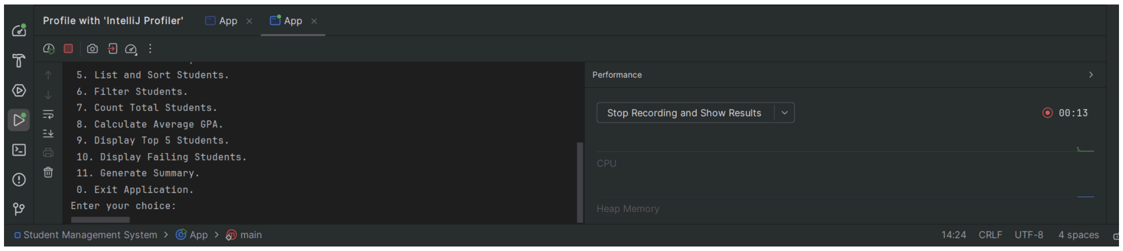


```
.");
ent.");
```

3. Select the Profiler
    - From the menu, the option *Profile 'App.java' with IntelliJ Profiler* was selected. This launched the application and began collecting profiling data.

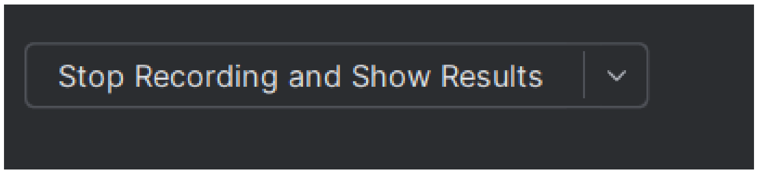

Current File
Run 'App.java' with Coverage
Profile 'App.java' with 'IntelliJ Profiler'
Run with Parameters...
ammed Atef Abd El-Kader

4. Run the Application Under Profiling
    - The application executed normally while the profiler actively recorded runtime behavior. Profiling status and related output were visible in the console.


Profile with 'IntelliJ Profiler'
App
App
Performance
Stop Recording and Show Results
00:13
CPU
Heap Memory
Student Management System > App > main
14:24 CRLF UTF-8 4 spaces

```
 5. List and Sort Students.
 6. Filter Students.
 7. Count Total Students.
 8. Calculate Average GPA.
 9. Display Top 5 Students.
 10. Display Failing Students.
 11. Generate Summary.
 0. Exit Application.
Enter your choice:
```

5. Stop and View Results
    - Once the desired profiling period was complete, the *Stop Recording and Show Results* button was clicked. This action ended the profiling session and automatically displayed the results within the IDE.



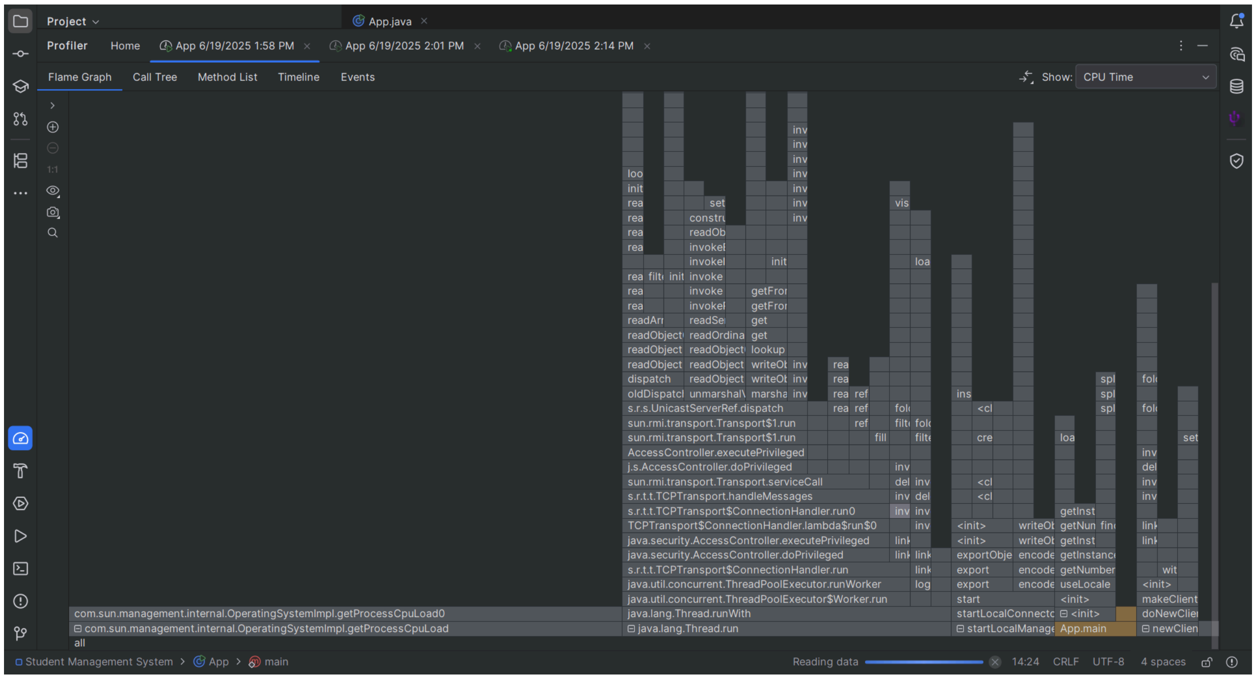
6. Analyze Profiling Data
    - The profiler presented a detailed breakdown of performance metrics, including CPU usage, method call frequencies, and execution time visualizations (e.g., flame graphs and call trees). These insights were used to identify and understand performance bottlenecks.

3.3.2 Flame Graph

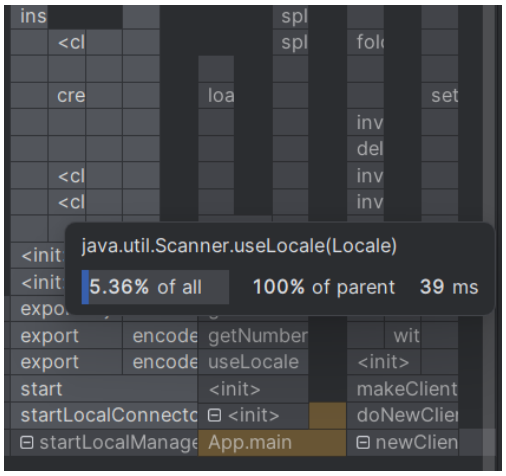
A flame graph shows how much time your application spends in each method during execution. Each block represents a method, and wider blocks mean more CPU time was used. The graph shows which methods call others, but not the order in which they run.

It helps identify performance bottlenecks. For example, if a method takes up 29% of CPU time, it may be running too often or taking too long—both are signs it could be optimized.

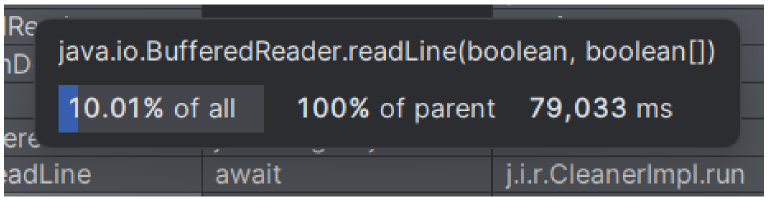
Flame graphs are useful for quickly spotting where most of the time is being spent in your code.



- **CPU Time** - From the flame graph, we observe that App.java calls the Scanner class, which in turn invokes a method that consumes 5.36% of CPU time, running for approximately 39 milliseconds. This usage is relatively low and indicates no significant performance concern in this part of the code.



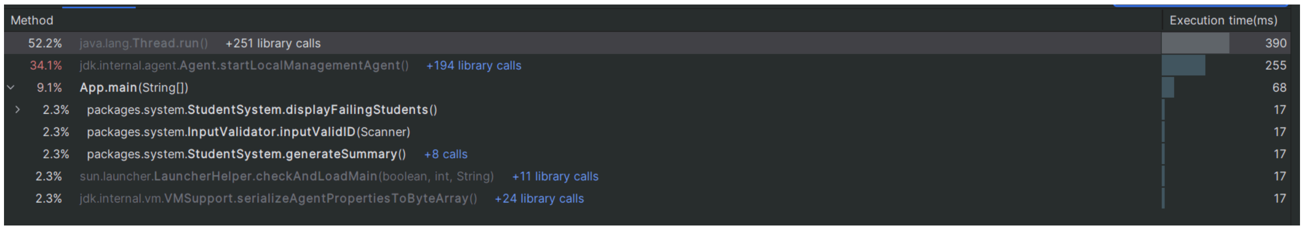
- **Total Time** - From the flame graph, we observe that App.java calls the readLine function, which in turn invokes a method that consumes 10.01% of total CPU time, running for approximately 79 milliseconds. This usage is mediocre and indicates little significant performance concern in this part of the code.



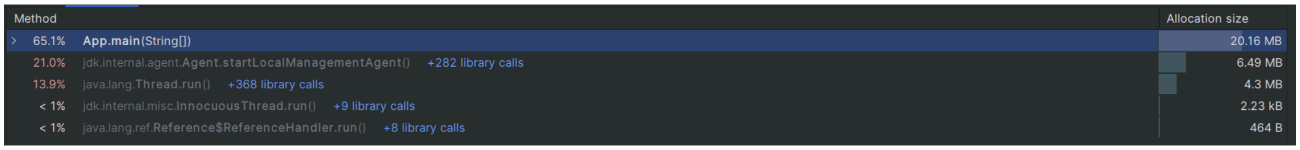
```
java.io.BufferedReader.readLine(boolean, boolean[])
10.01% of all    100% of parent    79,033 ms
```

3.3.3 Call Tree

Call tree shows us the hierarchical view of the method calls and the time it takes for them to execute them. Here we can observe which method chains consume the most resources.



```
Method                                                                                          Execution time(ms)
  52.2%  java.lang.Thread.run()  +251 library calls                                                            390
  34.1%  jdk.internal.agent.Agent.startLocalManagementAgent()  +194 library calls                              255
⌄  9.1%  App.main(String[])                                                                                     68
›   2.3%  packages.system.StudentSystem.displayFailingStudents()                                                17
    2.3%  packages.system.InputValidator.inputValidID(Scanner)                                                  17
    2.3%  packages.system.StudentSystem.generateSummary()  +8 calls                                             17
   2.3%  sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String)  +11 library calls                  17
   2.3%  jdk.internal.vm.VMSupport.serializeAgentPropertiesToByteArray()  +24 library calls                     17
```



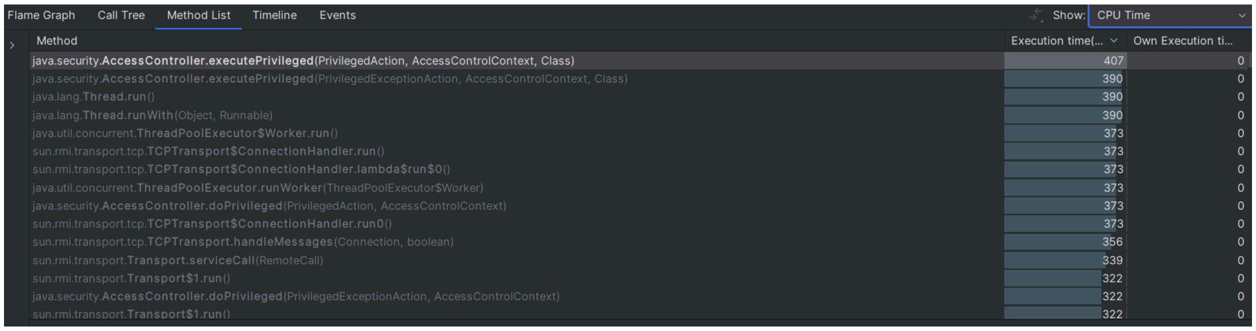
```
Method                                                                                          Allocation size
›  65.1%  App.main(String[])                                                                           20.16 MB
   21.0%  jdk.internal.agent.Agent.startLocalManagementAgent()  +282 library calls                      6.49 MB
   13.9%  java.lang.Thread.run()  +368 library calls                                                     4.3 MB
    < 1%  jdk.internal.misc.InnocuousThread.run()  +9 library calls                                     2.23 kB
    < 1%  java.lang.ref.Reference$ReferenceHandler.run()  +8 library calls                                464 B
```

CPU Time	Memory Allocation
770ms	30MB

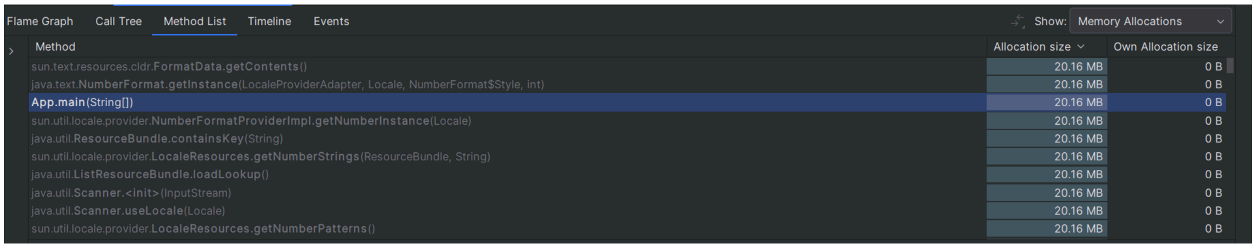
This metric helps us figure out if there are any performance bottlenecks or not. Helps us pinpoint the main chain of commands. Here we can observe whether a method is needed for an optimization or not. Abnormal use of resources is an indication that there's room for improvement.

3.3.4 Method List

This shows us the whole list of method calls made, then it shows the amount of time it took (self and total) and the amount of memory it used to carry out the execution. It also shows the number of occurrences, the number of times a particular method was invoked.



Method	Execution time(...	Own Execution ti...
java.security.AccessController.executePrivileged(PrivilegedAction, AccessControlContext, Class)	407	0
java.security.AccessController.executePrivileged(PrivilegedExceptionAction, AccessControlContext, Class)	390	0
java.lang.Thread.run()	390	0
java.lang.Thread.runWith(Object, Runnable)	390	0
java.util.concurrent.ThreadPoolExecutor$Worker.run()	373	0
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run()	373	0
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.lambda$run$0()	373	0
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor$Worker)	373	0
java.security.AccessController.doPrivileged(PrivilegedAction, AccessControlContext)	373	0
sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0()	373	0
sun.rmi.transport.tcp.TCPTransport.handleMessages(Connection, boolean)	356	0
sun.rmi.transport.Transport.serviceCall(RemoteCall)	339	0
sun.rmi.transport.Transport$1.run()	322	0
java.security.AccessController.doPrivileged(PrivilegedExceptionAction, AccessControlContext)	322	0
sun.rmi.transport.Transport$1.run()	322	0



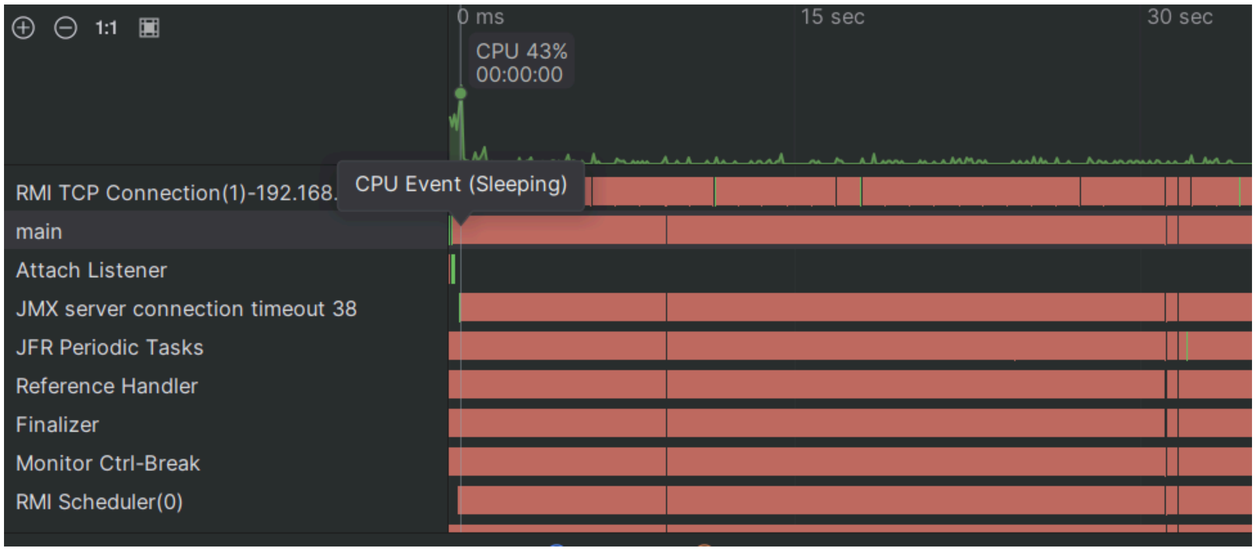
Method	Allocation size	Own Allocation size
sun.text.resources.cldr.FormatData.getContents()	20.16 MB	0 B
java.text.NumberFormat.getInstance(LocaleProviderAdapter, Locale, NumberFormat$Style, int)	20.16 MB	0 B
App.main(String[])	20.16 MB	0 B
sun.util.locale.provider.NumberFormatProviderImpl.getNumberInstance(Locale)	20.16 MB	0 B
java.util.ResourceBundle.containsKey(String)	20.16 MB	0 B
sun.util.locale.provider.LocaleResources.getNumberStrings(ResourceBundle, String)	20.16 MB	0 B
java.util.ListResourceBundle.loadLookup()	20.16 MB	0 B
java.util.Scanner.<init>(InputStream)	20.16 MB	0 B
java.util.Scanner.useLocale(Locale)	20.16 MB	0 B
sun.util.locale.provider.LocaleResources.getNumberPatterns()	20.16 MB	0 B

Max CPU Time	Max Memory Allocation
407 ms	20.15MB

This metric addresses the per-method performance cost and the ratio between self-time and total time. It identifies methods with high self-time, meaning they consume a lot of resources. Meaning that there's room for optimizations in these cases. Overall, if total time is high but self-time is low, it suggests that the sub-methods are quite expensive.

3.3.5 Timeline

- CPU 43% → Moderate CPU usage (no severe bottleneck).
- 0 ms / 15 sec / 30 sec → Shows time progression.
- CPU Event (Sleeping) → Indicates threads are mostly idle (not under heavy load).



3.5 Conclusion

Overall the dynamic analysis depicts a lightweight, well-behaved small scaled Java application with no memory leaks, low CPU and memory overhead, no performance issues, stable class loading behavior and minimal garbage collection activity.