Assignment on:

Program Comprehension

SWE 4802: Software Maintenance Lab

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)

Project Link: Student Management System

Table of Contents

1 Introduction

Program comprehension is the process of understanding the whole software program. Understanding a program is a **cognitive** process and for that, a lot of processes have been developed. This is a very common challenge if one wants to perform **maintenance** on a program and to fuel its **evolution**.

To extend a program by adding features, functionalities, maintaining the software, and ensuring a successful evolution, one must thoroughly **understand** the whole system properly. This includes the source code, components, modules and their relationships with one another. To properly grasp them is an essential skill to have for any programmer. To overcome this issue, many models have been proposed, these are known as **comprehension models**.

These comprehension models typically have *three basic elements*: [2]

(1) Goal of cognition: A programmer usually sets a goal in mind while trying to understand the program code. When there's a **goal in mind**, it helps guide the understanding process of the program and the scope of the program comprehension process. While skimming through the program code with a specific goal, one can acquire new knowledge about the program and better understand the knowledge that went into the code in the first place, aided by the programmer's existing knowledge.

(2) Knowledge base: The **knowledge base** refers to the **accumulation** of the knowledge that the programmer possesses or has at their disposal — the knowledge the programmer uses to understand the program. It could be the knowledge that is already known to the programmer through prior experience, the documentation, or any specifications provided with the program code, information from the internet, domain-specific knowledge, and more.

(3) Mental model. When a programmer tries to understand a program, they try to build a mental **picture** of it in their mind; this picture is called a **mental model**. This model helps them grasp what a code fragment does, how different parts of the program work together, and what different parts of the program are trying to accomplish.

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.

The main **objective** here is to explain how software tools and IDEs can help programmers understand the program code during the program comprehension process. To carry out program comprehension activities on the Student Management System, we are using **IntelliJ IDEA** as the IDE.

2 Activities

There are different methods for programmers to understand the program codebase written by someone else. All programmers follow their technique when looking at a codebase, no matter their experience. While less experienced programmers might try to move around in files to understand how it works, experienced programmers usually have a technique they have adapted through the years that makes the process more efficient. Additionally, code editors also have features integrated which support code comprehension. Some of them are:

- Call hierarchy views
- Code Browsing
- Find all references
- Go to definition
- Intelligent code completion
- Split view
- UML diagrams

2.1 Call Graphs/Call Hierarchy

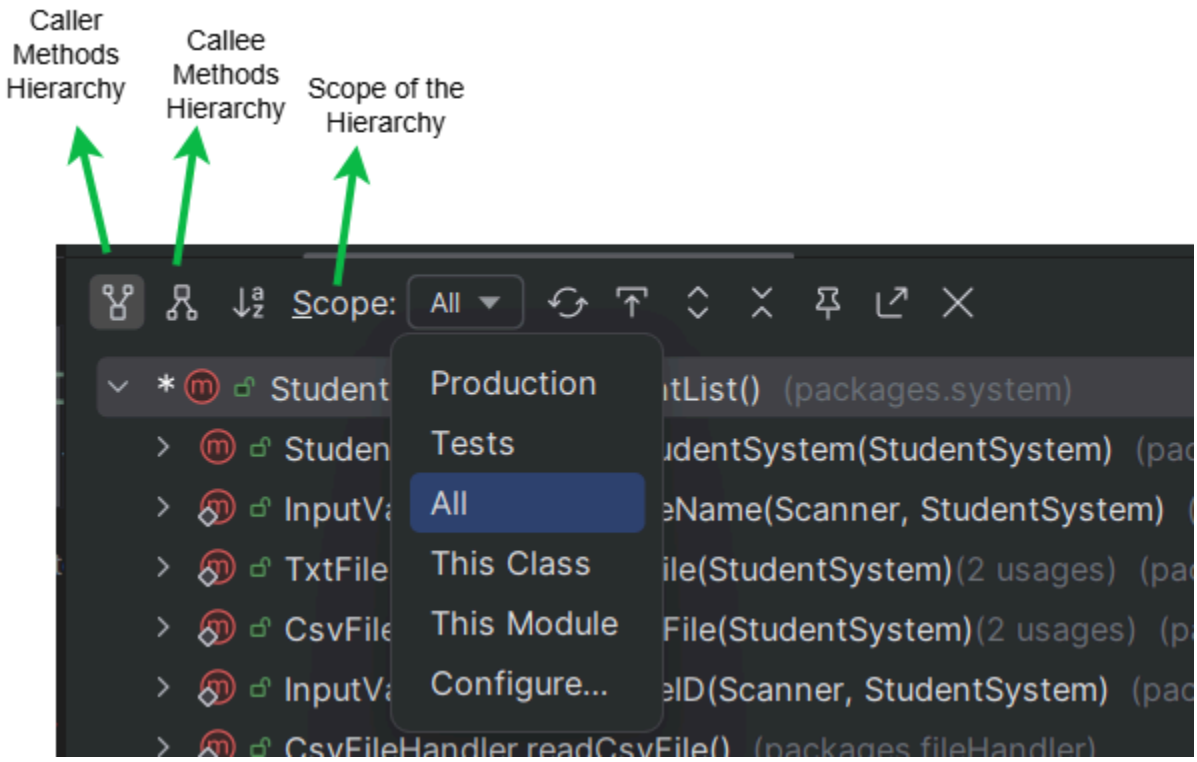
A call graph or call hierarchy view is one of the supporting tools used in the program comprehension process. It follows the top-down comprehension approach, as it allows developers to gain an understanding of the control flow of a specific function or method. It provides a visual tree or graph representation where each node represents a function, and the directed edges indicate the direction of the calls. The tree illustrates which function calls which, as well as the interdependencies among functions, either

within the scope of the entire project or confined to a single class. This tool is particularly useful during debugging and refactoring processes.

This feature is built into IntelliJ IDEA, and the analysis for this report will be conducted using it. IntelliJ IDEA offers the following options:

- **Caller Methods Hierarchy** - Displays the methods that call or use the function being analyzed—that is, the functions that call this function, as well as those that call the caller functions, and so on.
- **Callee Methods Hierarchy** - Displays the methods that are called by the function being analyzed—that is, the functions invoked within this function.

IntelliJ IDEA also allows customization of the hierarchy scope, enabling developers to limit the analysis to specific classes or expand it to cover the entire project, depending on the focus of the comprehension task.



To view the call graph of any method in IntelliJ IDEA, simply place the cursor on the desired method and press `Ctrl + Alt + H`. A side panel will appear, displaying the

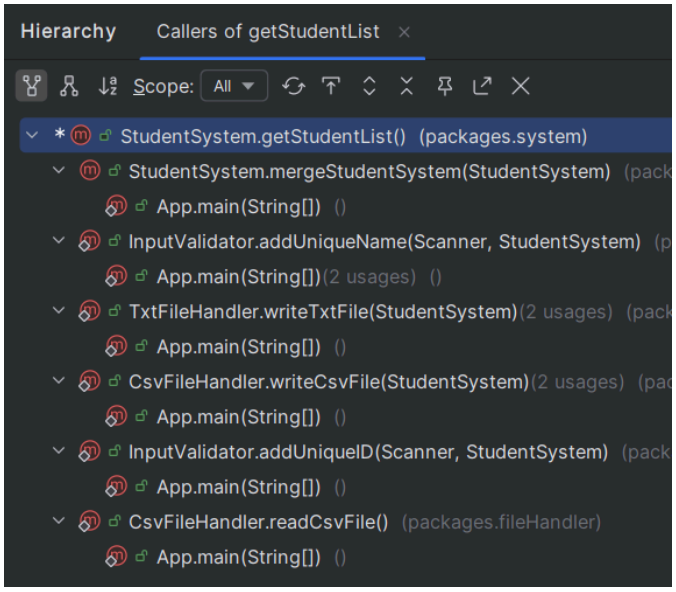
Caller Methods Hierarchy view by default, along with various customization options to adjust the scope and depth of the hierarchy.

```java
public class StudentSystem {  10 usages  Mohammed Atef Abd El-Kader
    private final ArrayList<Student> studentList;  29 usages

    public StudentSystem() {this.studentList = new ArrayList<>();}  14 usages  Moham
    public ArrayList<Student> getStudentList() {return studentList;}  8 usages  Moha

    // Method to merge another StudentSystem into this one
    public void mergeStudentSystem(StudentSystem otherSystem) {  1 usage  Mohammed A
        int nonUniqueID = 0, nonUniqueName = 0, oldSize = this.studentList.size();
        for (Student student : otherSystem.getStudentList()) {
            boolean isFound = false;
            for (Student student1 : this.studentList) {
                if (student1.ID == student.ID) {
                    nonUniqueID++;
                    isFound = true;
                    break;
                }
                if (student1.name.equals(student.name)) {
                    nonUniqueName++;
                    isFound = true;
                    break;
                }
            }
            if (!isFound) this.studentList.add(student);
        }
```

For tool analysis, the `StudentSystem.java` class from the *Student Management System* is shown. As you can see, the code snippet above shows the implementation of a method named `mergeStudentSystem()`. Inside the function, `getStudentList()` is invoked. The call graph for the method `getStudentList()` is shown below.



If we focus on the first collapsed branch of the call tree, we can observe that the `getStudentList()` method has a direct caller within the `StudentSystem.java`

class. This caller, in turn, has a parent caller in the `App.java` class. From this section of the tree, a developer can infer that both the `StudentSystem` class and, indirectly, the `App` class are dependent on the `getStudentList()` method. As a result, any error in this method is likely to propagate and cause issues in those two classes as well.

Consequently, the call graph displays other methods that directly invoke or utilize the `getStudentList()` method, along with their respective parent callers. From this visualization, it becomes evident that the method is used across multiple classes, indicating a high level of interdependence. Therefore, modifying this method may have significant effects on all associated classes.

2.2 Code Browsing

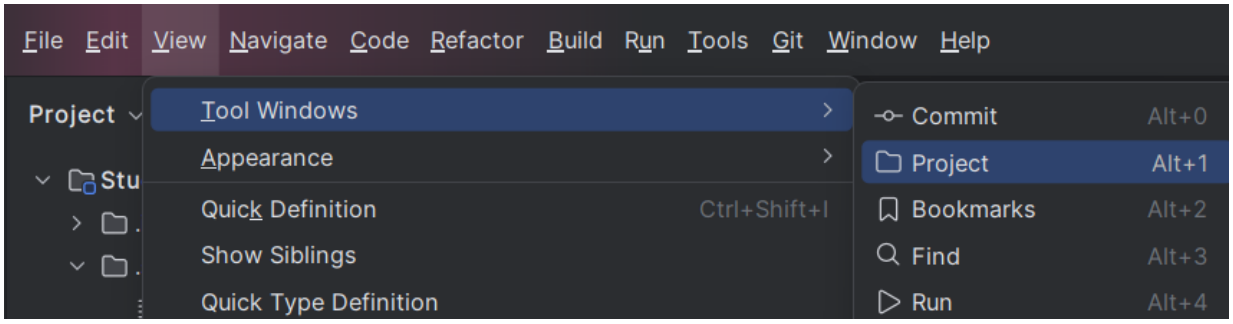
Code browsing means going through the codebase to understand each part of the program properly. One would need to jump between files, modules, classes and methods to fully understand how the program is built and how they are all connected.

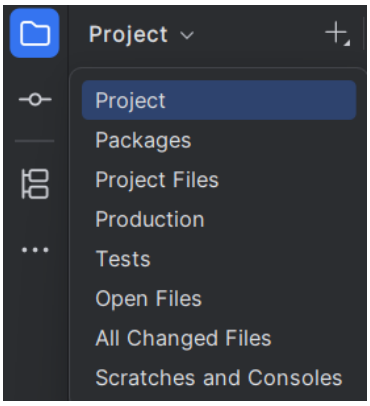
2.2.1 Top-down Approach:

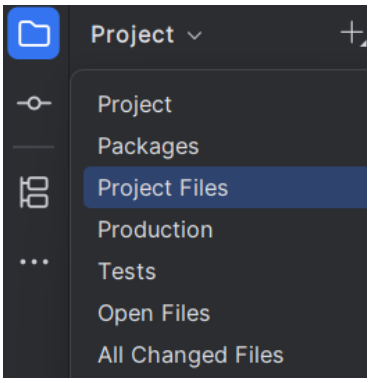
In the top-down approach, we start by looking at the bigger parts first so we can mentally imagine the big picture. This would include going through all the modules, files, classes then methods, in that order. Meaning we start from a higher-level structure, then go to the lower ones.

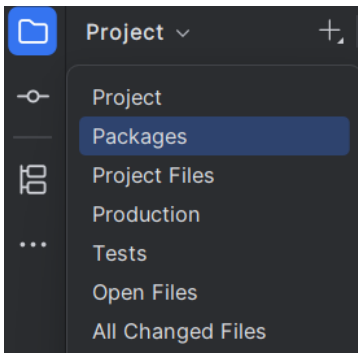
Project Structure, Package, Project Files View:

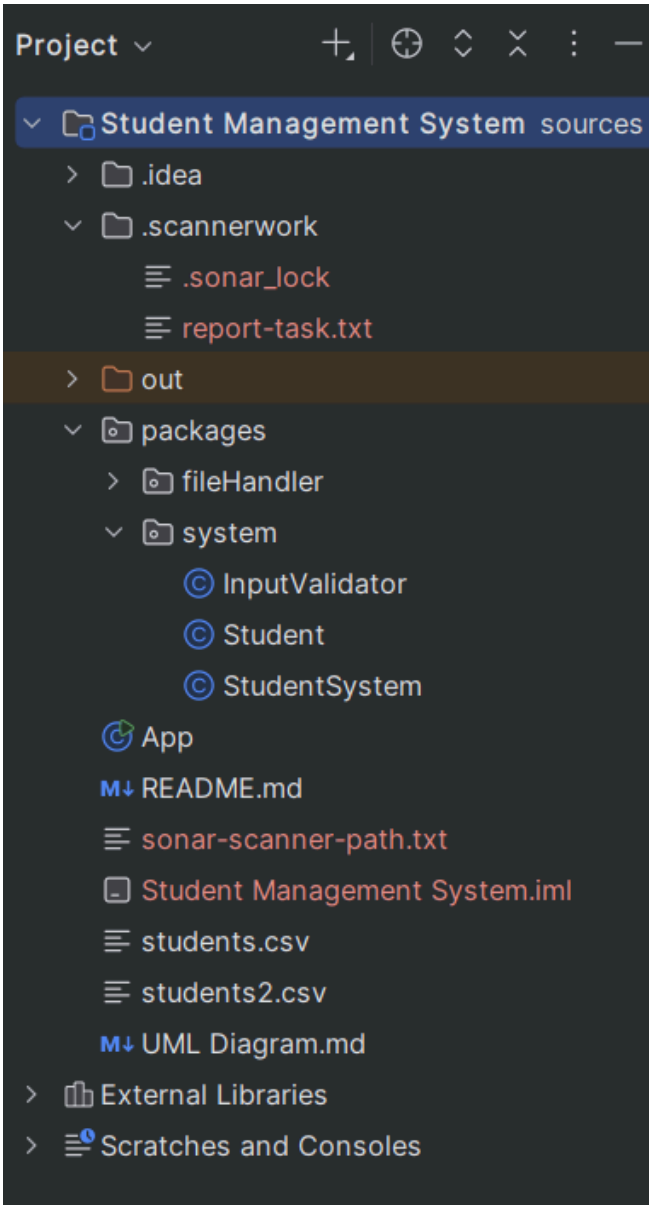
- Navigation: View → Tool Windows → Project, Packages, Open Files, etc
- If we look at the figure on the right, we can see the whole project hierarchy, we can browse the packages, files, classes, helping us understand the overall structure of the project.





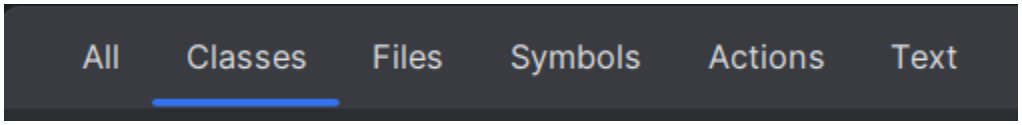






File, Class View:

- File view shortcut: `Ctrl + Shift + N` This allows searching for specific files
- Class view shortcut: `Ctrl + N` This allows searching for specific classes
- If we take a look at the figure down below, we can see the scope of the search. Here, while skimming through the modules, we can search for a specific file or class, making our comprehension process much easier



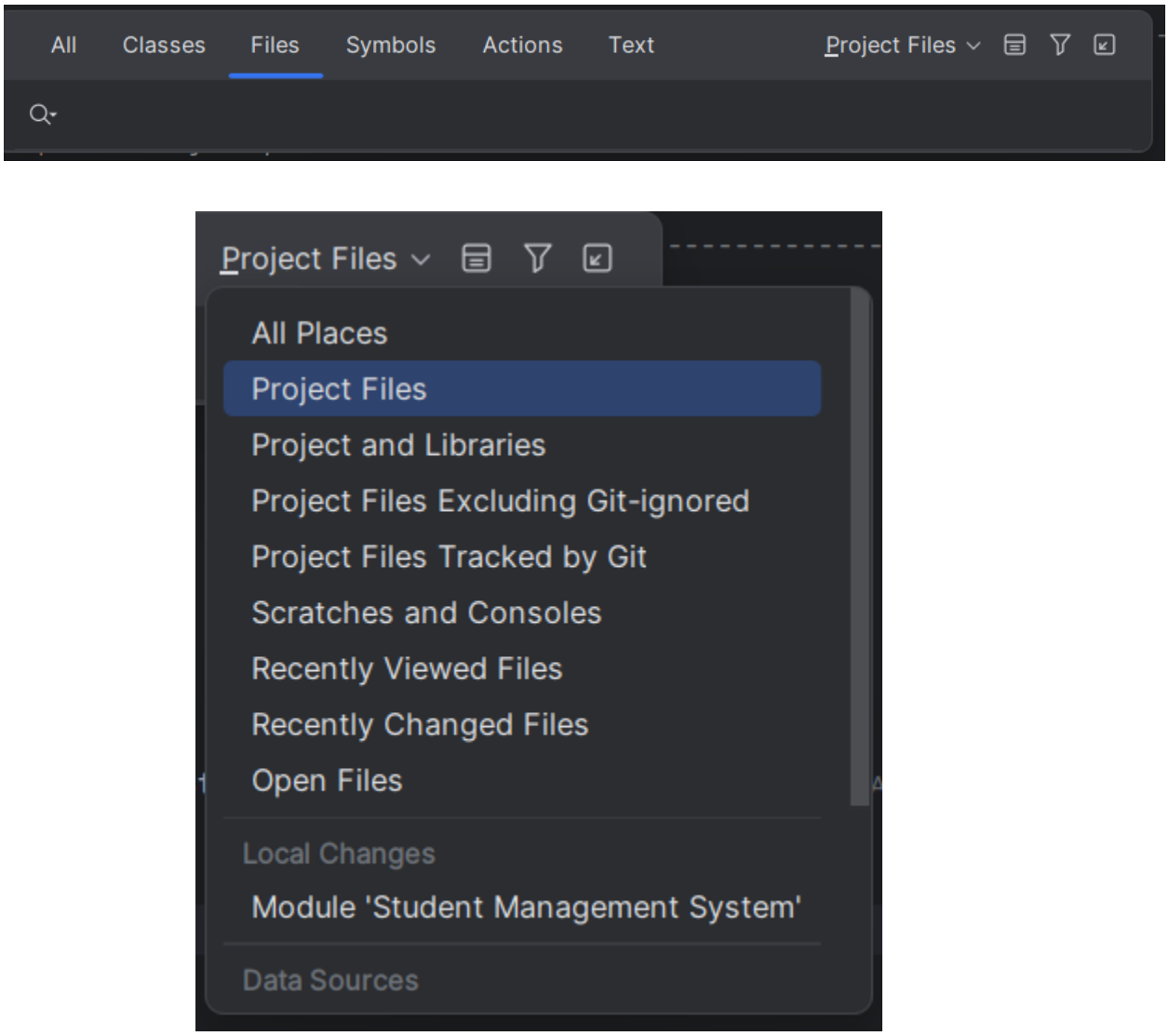


Figure: Scope of File, Class, Symbol Search Tool

2.2.2 Bottom-up Approach:

In the bottom-up approach we start from the small specifications. Then we accumulate all the small parts and come up with the bigger picture. This would include going through all the lines, variables, signatures and methods, in that order. Meaning we start from lower-level structures then gradually go to the higher ones.

Symbols View:

- File view shortcut: `Ctrl + Shift + Alt + N` This allows searching for specific symbol
- Quite similar to the files and classes viewer, this helps us understand the code, relationships between the classes and modules on a small scale. For example, we can search for methods, variables, constructors, lists and even classes, interfaces and enumerators.

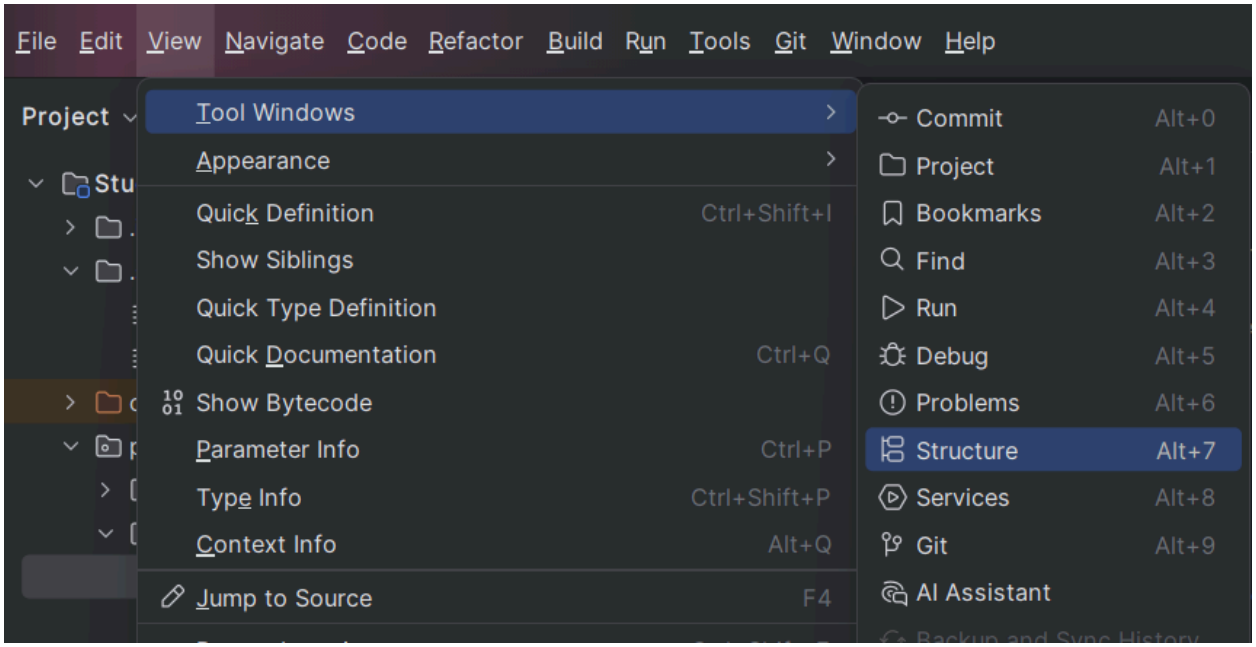


Figure: Navigation to the Structure Panel

Structure Panel Tool:

- Shortcut: `Alt + 7` or Navigation: `View → Tool Windows → Structure`
- This panel allows us to see and jump between methods, fields, variables, and lists of a class or an interface in an efficient way. The figure below shows the structure of a particular class.

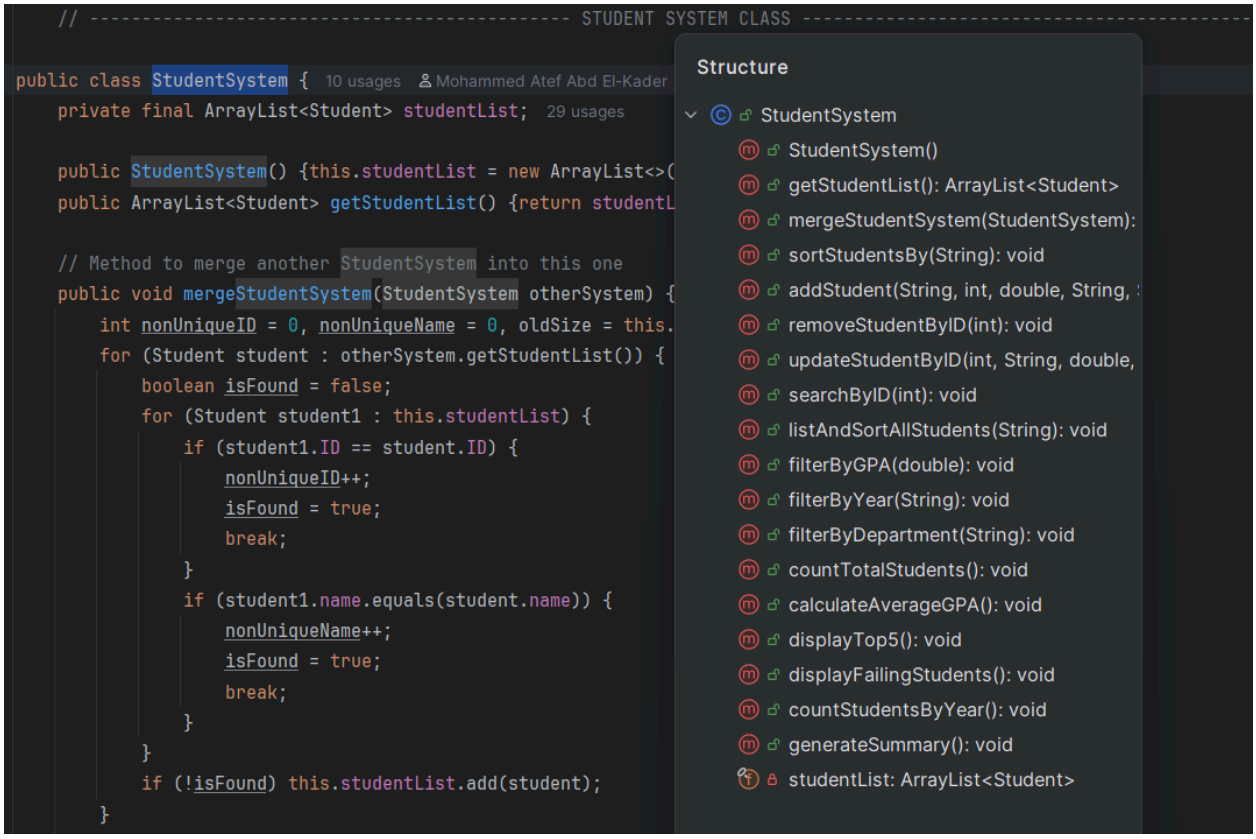


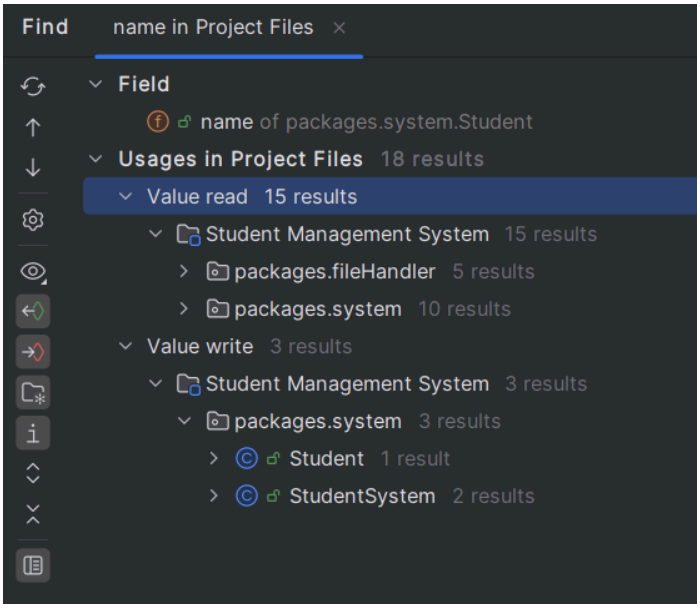
Figure: Structure Panel

2.3 Find All References

Find All References is a fairly self-explanatory tool that most developers have used at some point in their careers. Since it allows analysis starting from the granular elements of the code, this tool supports the **bottom-up approach** to program comprehension. Examining the usage of a single variable can help a developer assess its importance and contribute to the chunking process.

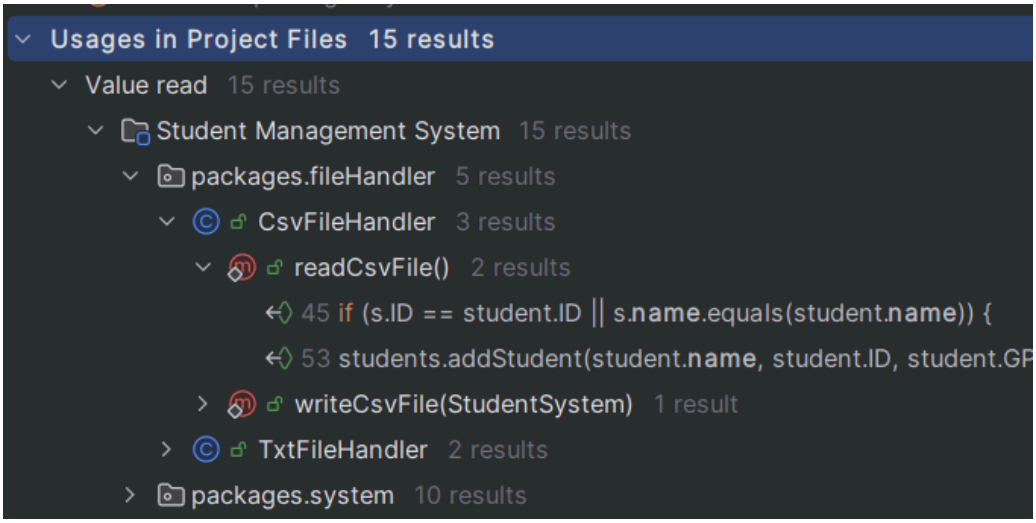
For example, if a declared variable has no references anywhere in the codebase, it may be identified as dead code and should be removed. Also, finding all references helps in understanding the control flow and functional dependencies within the program.

To view the *Find All References* panel of any variable in IntelliJ IDEA, simply place the cursor on the desired variable and press `Alt + F7`. Alternatively, we can also right-click on the variable and select the *Find Usage* option. A bottom panel will appear, displaying all the usages of the variable, along with various customization options to adjust the access and details of the usages.



From the screenshot, we can see that the references are divided into two categories:

- **Value read** - Indicates usages where the variable's value is accessed or used
- **Value write** - Indicates usages where the variable's value is modified or assigned



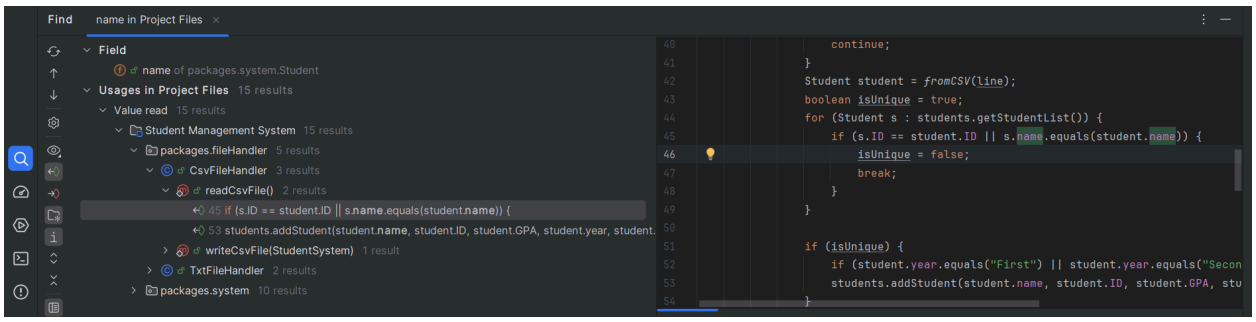
For analysis purposes, this is the fully expanded view of the variable `name` for the first method that uses it.

At the top of the panel, we can see the total number of usages of the variable across the entire codebase—15 usages, as indicated beside *Usage in Project Files*. Since the method in question falls under the Value Read category, that tab is expanded.

The first branch in the hierarchy indicates the main project file, *Student Management System*, followed by the package structure. As you can see the `packages.fileHandler` consists 5 out of the total 15 usages. Inside this package, two classes are identified: `CsvFileHandler.java` and `TxtFileHandler.java`. The `CsvFileHandler` class which contains 3 out of the 5 usages in this package is further expanded to show the specific methods that reference the variable `name`.

Among the two methods found – `readCsvFile()` and `writeCsvFile()` – the `readCsvFile()` method is expanded again. This specifies the exact line of code where the variable is accessed along with the corresponding line number. Adding to it, an icon is displayed at the beginning of the line to indicate that only read permissions are being used here.

Clicking on the code line opens another side panel. This displays the code snippet from the source file highlighting the line where the variable is accessed. This navigation feature makes it extremely convenient to locate any use of any variable in the entire codebase regardless of the project's size.



2.4 Go To Definition

The top-down approach begins by outlining the main application logic. This is broken down further with increasingly detailed dives into methods and objects to get a cleaner picture of how the system works in its entirety.

Step 1: The Main method in App.java as the starting point

- The system begins execution from the main() method.
- A StudentSystem object is created:

```
StudentSystem system = new StudentSystem();
```

- The structure permits the user to interact with the system through the menu-driven interface based on their input selections.

Step 2: Choosing an operation from the menu

- For instance, selecting option 1 leads to "Add Student," which has 2 sub-options:
    1. From CSV file:
       system.mergeStudentSystem(CsvFileHandler.readCsvFile());
    2. Manual input:

       system.addStudent(...);

Step 3: Navigating to StudentSystem.java methods

- addStudent() instantiates a new student object and adds it to the internal ArrayList<Student>.
- Other methods from StudentSystem.java include:
    1. removeStudentByID(int id)
    2. updateStudentByID(...)
    3. listAndSortAllStudents(String sortBy)
    4. filterByGPA(), filterByYear(), filterByDepartment()
    5. displayTop5() and displayFailingStudents()
    6. generateSummary(), etc.

Step 4: Inside Student.java

- Each student object stores:

  String name, int ID, double GPA, String year, String department.

Following the method calls starting from app.java and traversing down into class definitions and object manipulations illustrates this top-down approach.

2.5 Code Completion

Code Completion is a built-in feature in IntelliJ IDEA that improves program comprehension and, additionally, development speed. For large applications with a large number of components, code completion is a valuable tool for navigating and understanding the codebase faster.

IntelliJ IDEA provides intelligent, context-aware suggestions as a developer is typing. These suggestions are as follows:

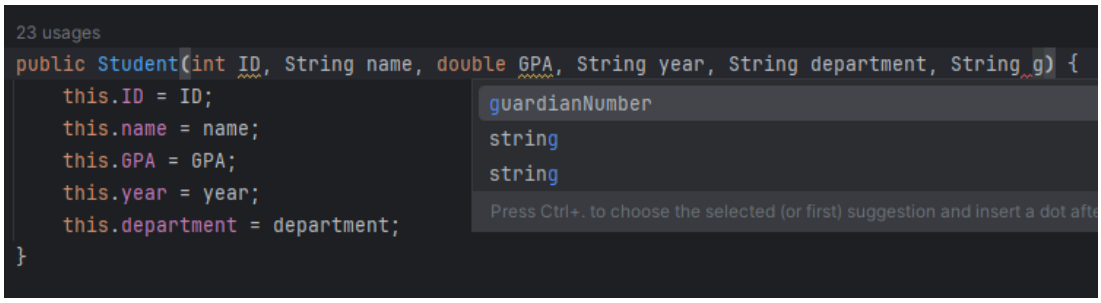
- **Basic Completion** shows the names of the classes, methods, fields, and variables that match what is being typed, limited to the current scope. For example, typing G in the `Student` class would provide suggestions for the `GPA` variable.
- **Smart Completion** offers suggestions based on the expected type of variable.
- **Statement Completion** makes the current statement syntactically correct by adding necessary syntax, like adding a semicolon or balancing parentheses.
- **Cyclic Word Completion** suggests words that are already in the current file or other open files.
- **Live Templates** allows use of shortcuts like `psvm,` which expands to `public static void main(String [] args)`.

Code completion follows a top-down approach to program comprehension. Developers can start from higher-level interactions, such as method calls, and then look into the lower-level implementation details like their definitions, as needed. This is useful in this Student Management System, where developers can view and use methods when using any instance of a class and then see how the method works internally if needed.

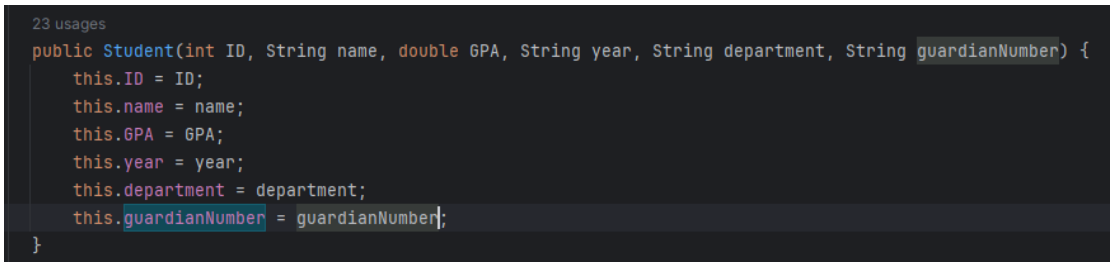
For example, if there is a new feature to be added that stores a guardian's number, code completion can help to identify the fields and methods available in the `Student` class and add the field. With real-time suggestions from the **Basic Completion** feature, the developer can easily update the logic within minutes. Once updated, the code edition will then suggest the necessary changes required to be made to the `StudentSystem` class as well as other related classes.

For basic demonstration of this, the feature addition is simulated in the IntelliJ IDEA. First the field `guardianNumber` is added to the `Student` class and while typing, the IDE provides a list of possible keywords and methods that might be added.

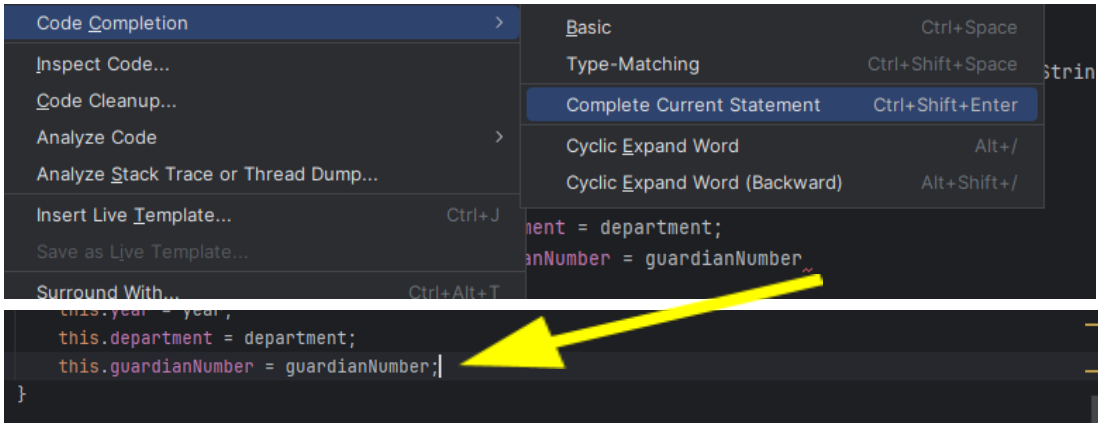
When updating the constructor, it shows the variable name that matches the typed letter ‘g’.



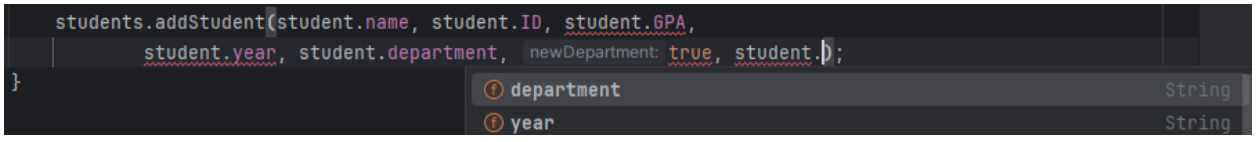
Using the **Cyclic Completion**, guardianNumber can be added without even searching for it by cycling through the words.



Afterwards, the **Statement Completion** can be used to add a semicolon to the end of the statement to balance it.



When making changes to other classes, the suggestions also make it easier and faster to navigate to the appropriate field without typing the entire word. No spell-check is required here either.



All of these have easy keyboard shortcuts that make the process even faster. There was very little need to look into the low level implementation of any methods or look through the entire codebase.

Without code completion, developers would have to read a significant amount of the codebase to find out where the changes are needed, double-checking would be required to make sure that the method names are not wrong, any additional packages needed would have to be typed fully to be imported for use, and so on. Instead of a few minutes, it might have taken hours, thus slowing down the development productivity.

2.6 Split View

The Split View feature in IntelliJ IDEA allows a developer to open and view any set of two files, whether side by side or one on top of the other. It boosts understanding during both forms of development, i.e., top-down and bottom-up.

2.6.1 Top-down Approach

In a top-down approach, a split view is handy as the user can repeatedly switch between the high-level components and low-level classes like App.java, StudentSystem.java, and Student.java.

For example:

- While analyzing the system, addStudent(…) in App.java, users can directly go to the declaration of the addStudent method from the class StudentSystem.
- This reduces the need to constantly switch tabs or use Go To Definition, offering a bird's-eye view across the codebase.

2.6.2 Bottom-up Approach

From the lower-level perspective, let’s assume that they were trying to improve or debug an upper-level class such as Student.java. The split view would enable them to:

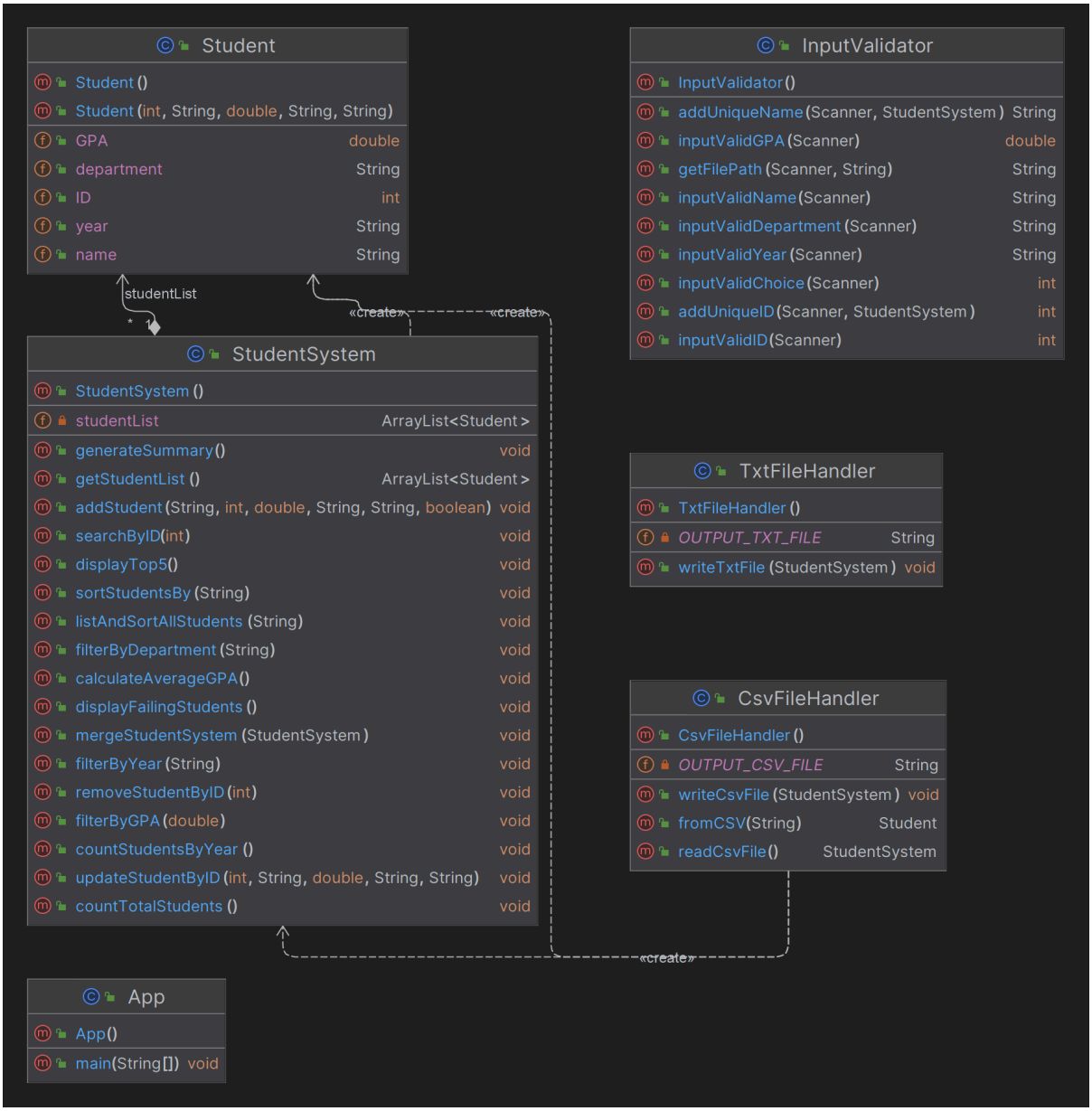
- View documents together to see how functions that alter fields such as name, GPA are defined and invoked in StudentSystem.java, for example.
- This is particularly useful when refactoring, as developers can instantly verify where and how each class or field is used.

The split view feature in IntelliJ supports visually understanding, debugging, and modifying code. In conjunction with Go To Definition, it offers a context-sensitive way to

navigate through the code, which helps understand the logical connections needed throughout multiple files without losing your train of thought.

2.7 UML Diagram

A UML (Unified Modeling Language) Diagram is a top-down approach that aids in understanding the program. Its top-down approach nature means it provides a clear, high-level view of the system's structure, the relationships among classes, and the functionality of each class. In large or collaborative projects, UML is one of the most effective tools to support program comprehension since it is independent of any programming language and has a unified notation easily understood by developers.



In the Student Management System, the class diagram allows a good visualization of the interactions and features. Without viewing the codebase, it can be seen from the UML diagram that there are six classes in the application that have the following responsibilities:

1. `Student` class, which represents all the information required for the management of a student.
2. `Student System` class that holds an array of `Students` and handles all related operations to manage students.
3. `Input Validator` class to handle all validations related to input values for department, GPA, year, etc.
4. A CSV and A Text File Handler that provides support for saving and fetching student information in both CSV and TXT formats.
5. `App` class that provides an interface to the classes for the users to interact with and acts as a bridge between the UI and the backend.

The reason that the classes provide such clear information to this extent is due to the intuitive variable and function names across classes, which are self-explanatory, which significantly enhances program comprehension. Developers intending to introduce any new features or modify existing ones can easily see which classes would require changes as opposed to reading the entire codebase to understand the system logic. For adding logic that the guardian's number has to be stored as a new feature, the developer can easily trace it to its specific class, where this information is being stored.

This also makes the development simplified due to easier onboarding processes, but also makes the maintenance process less time-consuming. Developers can pinpoint the location of changes or fixes and read the relevant classes to better understand as opposed to diving directly into the source code.

The relationships between the classes (like associations and dependencies) are also illustrated in the diagram. This visibility makes it easy to predict the effect any modification might have on other related classes. For example, if the structure of the `Student` class changes to store the guardian's number, it is easy to see that the `Student System` class will also be subject to changes, as well as the file handler classes and input validator function that depend on it.

The diagram also shows where the data is being passed, validated, stored, or manipulated which makes it easier to follow the life cycle of information. This can be rather helpful for the debugging process or future enhancements. For example, if the data for the GPA of a student is not being updated properly, the UML diagram can help developers to identify exactly where the GPA is being handled. If the update of this GPA is focused on, it can be traced to `UpdateStudentById` in the `StudentSystem` class, validation of GPA input in the `InputValidator` class and a function in the `App` class taking the input.

References

1. Fekete, Anett, and Zoltán Porkoláb. "A comprehensive review on software comprehension models." *Annales Mathematicae et Informaticae*. Vol. 51. Liceum University Press, 2020.
2. Amal A. Shargabi, Syed Ahmad Aljunid, Muthukkaruppan Annamalai, and Abdullah Mohd Zin. "*Performing Tasks Can Improve Program Comprehension Mental Model of Novice Developers: An Empirical Approach*" In Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)