Declaration of Original Work for SC2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature / Date
BEH WEI XUAN	SC2002	SCMA	 22/11/2025
CHEYENNE SOO TZE CHENG	SC2002	SCMA	 22/11/2025
DARREN WEY YAN HENG	SC2002	SCMA	 22/11/2025
LAURENSIUS MARIO KISWANDHI	SC2002	SCMA	 22/11/2025
TAN SOON PING	SC2002	SCMA	22/11/2025

GitHub Link:
https://github.com/BehXuan/SC2002_LabProject

1. Requirement Analysis and Feature Selection

1.1 The Problem Statement

The goal of this project is to design and develop an Internship Placement Management System. This system will act as a centralised hub for students, company representatives, and career centre staff. Each of these roles will have basic user management features, e.g., login, logout, and change password, and have its own set of capabilities. The data required for the operation of the system (user list, internship list, etc.) will be stored in a Comma Seperated Value (CSV) format, and the whole system is operated through a Command Line Interface (CLI). Object-oriented programming (OOP) and object-oriented design (OOD) principles like encapsulation, polymorphism, inheritance and the design principles were seen as a goal during our coding and software design phase.

1.2 Features

All users have basic user management features such as login, logout, and updating their passwords. There are three types of users: Student, Company Representatives, and Career Centre Staff. Each of these user roles has its own associated capabilities.

Students are able to apply for internships they are eligible for based on the internship level. They can then accept the internship placement after being approved by the Company Representative, or withdraw their application anytime they want, subject to the Career Centre Staff's approval.

Company Representatives must first register an account to use the system. Once approved by the Career Centre Staff, they can create internship listings, toggle their visibility, and approve or reject students' applications.

The role of Career Centre Staff, aside from the aforementioned ones, is to generate reports regarding the internship opportunities. The report must be customisable based on several filters.

1.3 Assumptions Made

- Due to the System using a CLI, the system does not support having concurrent writes
- When a Student Object accepts an internship offer, they can no longer withdraw their accepted offer (Assume accepting an internship is binding)

2. System Architecture & Structural Planning

2.1 Architecture Used

The system adopts a Model-View-Controller (MVC) architecture to ensure clear separation of concerns. The Model stores and manages data, such as the User superclass and its subclasses, the View handles all user interaction through text-based menus, and the Controller contains the application logic, such as login, application processing, and report generation. This separation improves maintainability, as changes to the interface do not affect the core logic, and enhances scalability by allowing new features to be added without disrupting existing components. MVC is also well-suited for a CLI system, where the View simply prints menus and reads input, while the Controller processes commands sequentially. Overall, MVC keeps the system modular, organised, and easier to extend.

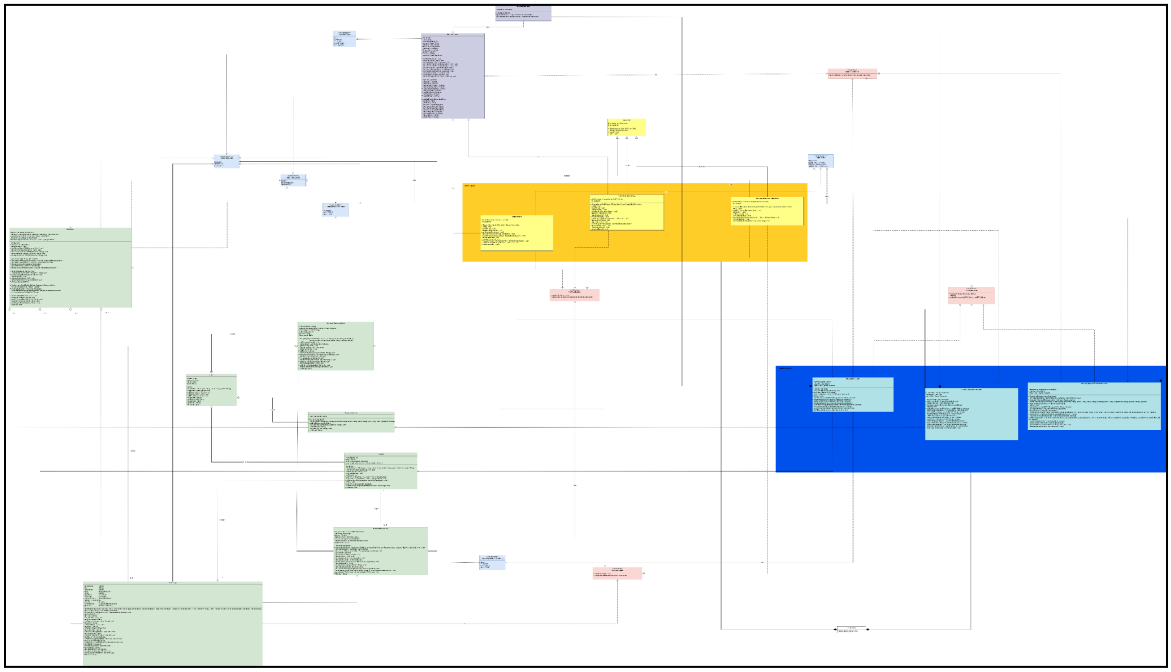
For storing data, we initially wanted to use arrays. However, using arrays meant that we would have to constantly update the memory space allocated to it, and arrays do not allow for efficient searching, filtering and retrieval, all of which were crucial to our system, especially for storage of login and internship details.

Arrays would have forced each controller to maintain its own separate collections of Users, Internships, and Applications. This decentralised storage makes it difficult to share information across controllers, causes data inconsistencies, and violates key SOLID principles.

To solve this issue, our group decided to utilize a single DataStore for storing data instead, as it was scalable, allowing for reusability of our code in the future when more users were added to the database. Moreover, it allows for the centralisation of all management of data, preventing scattered data or storage logic across the controller classes. Due to the centralisation of data management via the DataStore, any bugs that occur regarding data storage would only require inspection of DataStore, allowing for increased maintainability. By using a centralized DataStore instead, all controllers access a single shared source of truth, improving maintainability and extensibility.

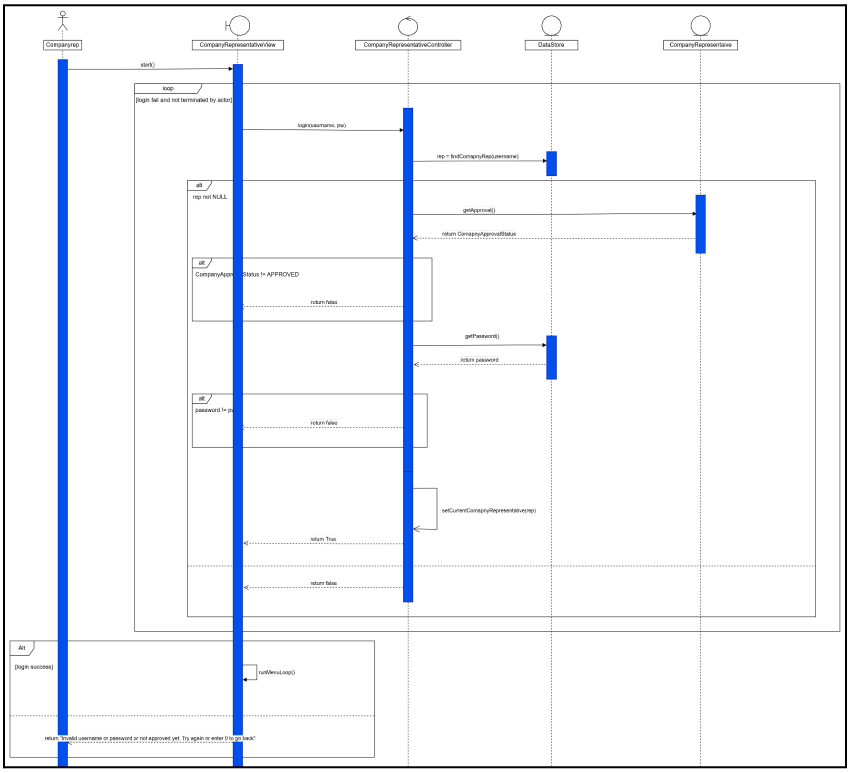
3. Object Oriented Design

3.1 Class Diagram



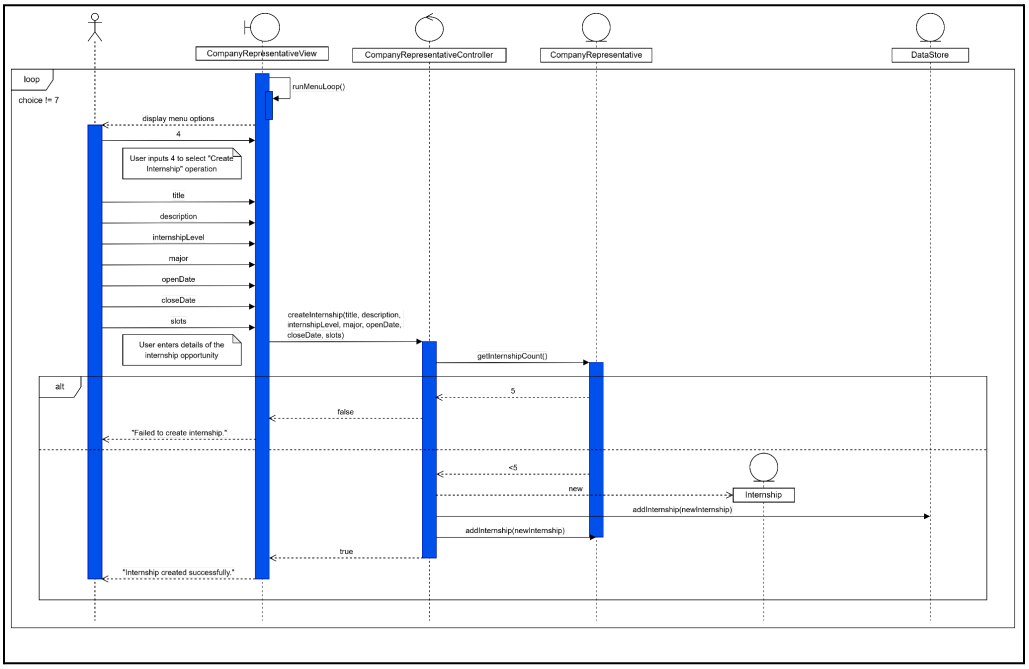
3.2 Sequence Diagram

3.2.1 Company Representative Login



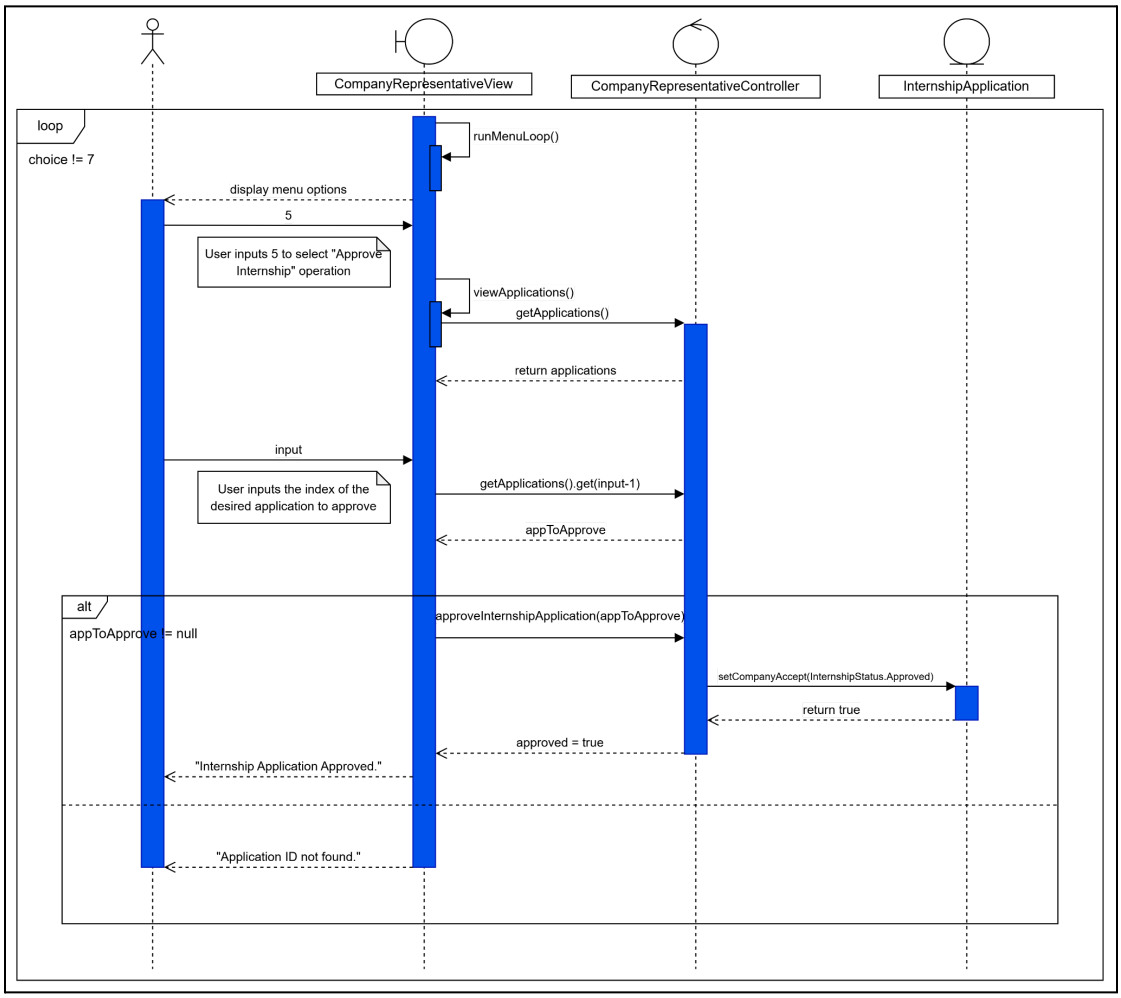
When a Company Representative attempts to log in, the system continuously prompts for credentials until a valid login is achieved or the user exits. The controller then checks the username and password against stored data in the DataStore. If both match, the representative successfully enters the main menu; otherwise, the system displays an error and repeats the login prompt.

3.2.2 Company Representative Creates Internship



The create-internship process begins when the CompanyRepresentativeView displays the menu and the user selects the create-internship option. The view collects all required inputs and passes them to the CompanyRepresentativeController, which first checks whether the representative has already reached the maximum of five internship listings. If the limit is exceeded, it returns a failure message; otherwise, it creates a new Internship object, stores it in the DataStore, and adds it to the representative's own internship list. A confirmation message is shown, and the menu is displayed again for further actions.

3.2.3 Company Representative Approves Internship



During the menu loop, the CompanyRepresentative selects option “5” to view and approve internship applications. After choosing an application index, the controller retrieves the corresponding application. If the index is invalid, the system prints “Application ID not found”. If the index is valid, the application is approved, and the system displays “Internship Application Approved.”.

4. OOP Concepts

Modifier and Type	Method	Description
`boolean`	`approveInternship(String internshipId)`	Approves the internship with the provided id.
`boolean`	`approveWithdrawal (InternshipApplication app)`	Approves a student's withdrawal request for a given application.
`boolean`	`authoriseCompany(String companyRepId)`	Authorises (approves) a company representative by id.
`List<Internship>`	`generateReport (ReportCriteria criteria)`	Generates a report of internships according to the provided criteria.

4.1 Abstraction

An example of abstraction in this system is the usage of controllers that abstract the user interface workflows from the actual data operations. The controllers reduce complexity and hide unnecessary details from other components.

4.2 Encapsulation

Our team encapsulated the data fields in the respective classes, making them private to ensure that the main application has no access to the fields. In order to get the data, controllers of the respective classes (Internship, CompanyRepresentative, Student and CareerStaff) would have to call their methods on the database. This ensures data integrity and prevents accidental modifications.

4.3 Inheritance

The CompanyRepresentative, Student and CareerStaff classes inherit from the base class, User. This prevents duplication of code with respect to attributes for login, UserID and name.

4.4 Polymorphism

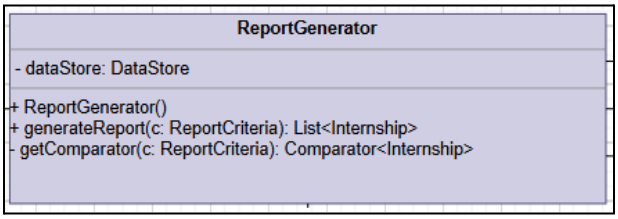
An example of polymorphism used in this system is the Starting logic. The classes CompanyRepresentativeView, CareerCenterStaffView and StudentView implement their own unique Start method from UserView.

4.5 SOLID Design Principles

A focus of this project is to apply good design practices. This enables the resulting code to have great reusability, extensibility, and maintainability. Achieving these aspects means ensuring that the code has low coupling, yet high cohesion.

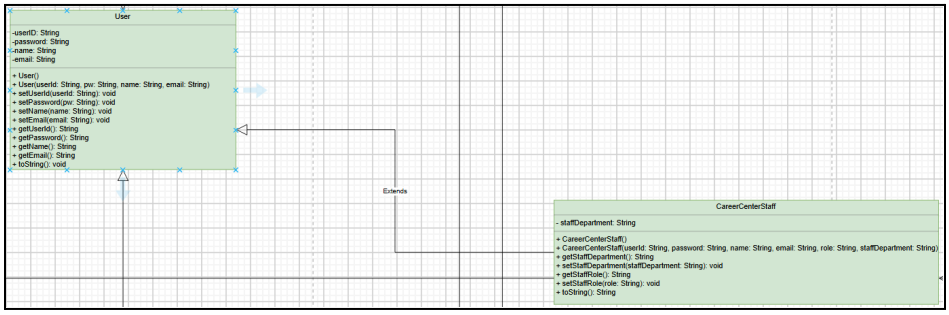
4.5.1 Single Responsibility Principle (SRP)

According to this principle, a class should only have a single clear responsibility, which has been demonstrated in this system. For example, *ReportGenerator* only generates reports and applies filters for each class.



4.5.2 Open-Closed Principle (OCP)

This principle dictates that classes should be closed to modification, but open to extension. A common strategy to enforce this is to create several specialisations of a base class. In our implementation, the different user types inherit the same *User* class, which enhances extensibility. If a new user type is introduced, the new class can simply be made to inherit the *User* class.

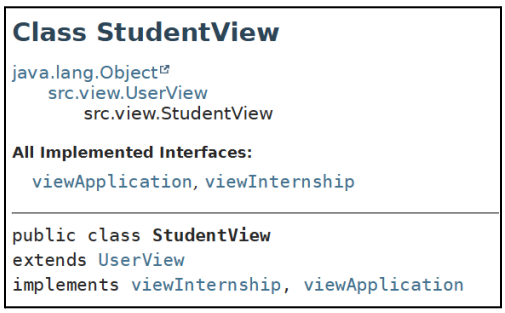


4.5.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that a child class is expected to be able to provide all the same services as its parent class, such that the parent class is substitutable by its child class. This is shown in our code, where the Student, CompanyRepresentative and the CareerStaff all inherit from the User class, allowing them to be used wherever the system expects a User. Controllers receive the correct user type and the system does not break when a subclass replaces the superclass.

4.5.4 Interface Segregation Principle (ISP)

The Interface Segregation Principle states that an interface should be more client-specific, meaning a class should not implement a general interface with unused methods. This helps improve the code by reducing coupling.



For example, our StudentView class implements two small, specialised interfaces: viewInternship and viewApplication. Each interface only contains the methods required for a specific behaviour. This ensures that StudentView includes only the functionality a student actually needs, such as viewing internships, without inheriting irrelevant methods intended for other user types.

4.5.5 Dependency Injection Principle (DIP)

Dependency Injection Principle helps overall design by decoupling high-level modules from low-level modules. Instead of a class creating its own dependencies, those dependencies are provided to it from an external source, typically through a constructor, method, or property.

In this system, controller classes depend on the DataStore for data storage, instead of implementing their own arrays. This removes the need for controller classes to manage concrete details, such as how data is stored and how data is retrieved. This thus satisfies DIP as the controller classes being high level modules, does not need to depend on low-level details.

5. Testing

Testing is done by manually operating the software for it to perform different use cases. The test case table below lists the action required and the corresponding expected output.

5.1 Test Case Table

No.	Test Cases	Resulting Behaviour	Pass/Fail
1	Valid User Login	User logs into their respective menu.	Pass
2	Invalid ID	User will be prompted with an error message: “username does not exist” and be asked to re-login.	Pass
3	Incorrect Password	User will be prompted with an error message: “Password is incorrect” and asked to re-login.	Pass
4	Password Change functionality	System updates password, redirects User to login page, and accepts only login with new password.	Pass
5	Company Representative Account Creation	Without approval, Company Representative will receive “User not approved” error message. After approval, Company Representative will be able to login.	Pass
6	Internship Opportunity Visibility Based on User Profile and Toggle	Internship opportunities are visible to students based on their year of study, major, internship level eligibility, and the visibility setting.	Pass
7	Internship Application Eligibility	Students can only apply for internship opportunities relevant to their profile (correct major preference, appropriate level for their year of study) and when	Pass

		visibility is on.	
8	Viewing Application Status after Visibility Toggle Off	Students continue to have access to their application details regardless of internship opportunities' visibility in "view my applications" menu.	Pass
9	Single Internship Placement Acceptance per Student	Internship is marked as accepted, all other internship applications that the Student has sent out are withdrawn automatically.	Pass
10	Company Representative Internship Opportunity Creation	System does not allow Company Representative to create sixth internship, gives error message "Failed to create internship".	Pass
11	Internship Opportunity Approval Status	Shows status of internships "PENDING, APPROVED, REJECTED".	Pass
12	Internship Detail Access for Company Representative	System still displays all full information on Company Representative's internships.	Pass
13	Restriction on Editing Approved Opportunities	System displays an error message "Failed to edit internship. Internship already approved", prevents Company Representative from editing.	Pass
14	Student Application Management and Placement Confirmation	Company Representative can view all student applications. After Company Representative accepts an application, and student accepts the internship offer, System reduces number of slots for that internship.	Pass
15	Internship Placement Confirmation Status Update	Placement confirmation status is updated to reflect the actual confirmation condition.	Pass
16	Create, Edit, and Delete Internship Opportunity Listings	System allows Company Representative to create, edit and delete internship opportunity listings as long as Career Center Staff has not approved.	Pass

17	Career Center Staff Internship Opportunity Approval	Career Center Staff can review and approve/reject internship opportunities submitted by Company Representatives.	Pass
18	Toggle Internship Opportunity Visibility	The internship opening with visibility turned off should be hidden from Students.	Pass

6. Reflection

6.1 Challenges & Knowledge Gained

Prior to doing this project, the mindset that the group had was a *code-first* mindset. Once we began working collaboratively, and delegating each member with a different portion or part of the code, our group faced code integration difficulties. Such difficulties include merge conflicts and inconsistent naming for functions, which caused runtime errors. Our code was also tightly coupled, meaning that the classes had many dependencies between them. When a team member changed their code in a core class, it could unexpectedly break another member's code.

However, issues with code duplication and merging conflicts were resolved after implementing a proper UML diagram. It became a shared blueprint that helped to clarify which class had what responsibility and methods, enabling each teammate to code in parallel, creating a more efficient workflow.

Furthermore, fully adopting the *design-first* mindset involves an iterative process of moving from the class diagram, sequence diagram, and code. This enabled the team to identify any missing methods that may be useful for certain operations, and to restructure the classes if necessary, to create a more maintainable and scalable application.

The group also learnt more about good architecture, such as the importance of encapsulation and restricted visibility, as visible data would have created more coupling between the respective classes. Another lesson on good architecture was the usage of DataStore, instead of the controller classes managing the data and persistence. For example, modifying the Internship class required the modification of all the other classes, until the group introduced the DataStore, which heavily improved modularity.

The group also tested the program manually, which was effective but time-consuming and error-prone. Moreover, manual testing could lead to accidentally avoiding any edge cases and usually required the testing of the whole program. Instead of testing the whole program, to ensure that there were no errors in each class, the group could have used unit testing instead,

where individual components of the program could be tested in isolation to ensure that they were functioning correctly. This would allow the group to isolate the problem at the root, allowing for faster debugging.

6.2 Limitations of Current Design

6.2.1 Bottleneck with DataStore

As all controllers rely on the singular instance of DataStore and CSV files are read to and written from the DataStore, it could cause a bottleneck in the system. Currently, our team's system has no caching. This would cause scalability issues when more internships or users are added into the CSV or DataStore, as functions like retrieving user data and filtering for internships become significantly more time-consuming. Any bug or data change would impact the whole system, and due to the tight coupling between the DataStore and the controllers and controllers all sharing a singular DataStore, the controllers cannot be independently tested.

6.2.2 No Data Safety

Currently, our system is reading once from the CSV file upon program start up, and writing once to CSV upon program shutdown. However, if the program terminates or crashes midway, any potential changes made to the DataStore are not saved to the CSV file.

6.3 Improvement Suggestions

6.3.1 Reducing Responsibility of DataStore

Lowering the responsibility of DataStore would help lower the coupling between the DataStore and the controller classes. Currently the DataStore handles the writing and reading to and from the CSV files, and handles all the data related to Internship, CompanyRepresentative, CareerStaff and Student. This potentially made the DataStore into a "God Object", which breaches the SOLID principle of Single Responsibility Principle (SRP). The group can instead employ Interface Segregation Principle (ISP), where we can divide storage responsibilities into smaller, specialised interfaces such as StudentDataAccess for storing Student data, InternshipDataAccess for storing Internship data, and InternshipApplicationDataAccess for storing InternshipApplication data. The controllers can then use the respective DataAccess classes for executing business logic.

6.3.2 Improving Data Safety

To save any data changes during the duration of the program, the application or program could instead use a SQLite database instead of a CSV to support concurrent writing and reading, allowing the application to write to the database every time the data in DataStore is changed.