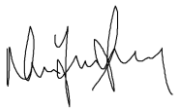
Declaration of Original Work for SC2002 Assignment

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
Neo Yan Siong	DSAI	SDDA Grp 3	 22/11/2025
Dasmond Tan Wei Jie	CE	SDDA Grp 3	 22/11/2025
Aaron Soh Jun Qi	DSAI	SDDA Grp 3	 22/11/2025
Guan Yibin	CE	SDDA Grp 3	 22/11/2025
Lee Wei Jie	CE	SDDA Grp 3	 22/11/2025

Important notes:

1. **Name must EXACTLY MATCH the one printed on your Matriculation Card.**
2. **Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.**

1. Design Considerations

Our codebase is built on strong **Object-Oriented Programming principles**. We implemented *encapsulation* to protect internal data, *inheritance* to form a clean hierarchy for different user types, and *polymorphism* to allow menus and managers to operate generically across Students, Company Representatives, and Staff. We also use *abstraction* to hide complex logic inside manager classes and apply *composition* and *aggregation* to structure relationships cleanly—for example, a Student 'has' applications, and menus 'use' the SystemManager without owning it.

Our system also follows the **SOLID design principles** closely. Each class has a single responsibility, the system is open for extension while closed for modification, covering Single Responsibility Principle (SRP) and Open Closed Principle (OCP). Subclasses follow the contracts defined in the User base class, and classes depend only on the interfaces they need, reflecting the Liskov Substitution Principle (LSP) and the Interface Segregation Principle (ISP). Finally, for Dependency Inversion Principle (DIP), we ensure that high-level logic interacts with repositories and managers through abstraction, not concrete implementations.

Lastly, our architecture aligns with the **Model-View-Controller pattern**. The *Model* consists of our entities and repositories. The *View* is represented by the console-based menu interfaces. And the *Controller* is implemented through manager classes such as UserManager, InternshipManager, and ApplicationManager. This separation keeps the system modular, testable, and easy to maintain.

These design foundations ensured that our code remained clean, scalable, and easy to extend as the project grew.

2. UML Class Diagram

2.1. Overview

The UML class diagram for the Internship Placement Management System provides a comprehensive visual representation of the system's architecture, illustrating the relationships between classes, their attributes, methods, and the overall design patterns employed. The system follows a three-tier architecture pattern consisting of Entity (domain), Control (business logic), and Boundary (presentation) layers, promoting separation of concerns and maintainability.

2.2. Architectural Layers

2.2.1. Entity Layer

User Hierarchy

- User (Abstract Class): Serves as the base class for all user types in the system, implementing common functionality such as authentication and profile management. This abstract class defines protected attributes (userID, name, password) and abstract methods (getUserType(), getRole()) that must be implemented by subclasses.
- Student: Represents student users who can apply for internships. Key constraints include a maximum of 3 concurrent applications, acceptance of only 1 internship placement, and year-based eligibility rules (Year 1-2 students can only apply for Basic-level internships, while Year 3-4 students can apply for all levels). The class maintains lists of applied internships and tracks the accepted internship ID.
- CompanyRepresentative: Represents company employees who create and manage internship opportunities. Each representative can create up to 5 internships and requires approval from Career Center Staff before gaining system access.
- CareerCenterStaff: Represents administrative users who oversee the entire system. They have elevated privileges to approve company representatives, approve or reject internship postings, and handle students with drawal requests.

Core Entities

- InternshipOpportunity: Represents an internship posting with comprehensive details including title, description, level, preferred major, dates, and slot management.
- Application: Tracks student applications to internships with full lifecycle management.

2.2.2. Enumeration Types

The system uses four enumeration classes to ensure type safety and constrain valid values:

- Major: Defines academic majors (CSC, EEE, MAE, CEE, MSE, CBE, OTHER) with display names and utility methods for string conversion.
- ApplicationStatus: Defines application lifecycle states (PENDING, SUCCESSFUL, UNSUCCESSFUL, WITHDRAWN) with display names for user interface rendering.
- InternshipStatus: Defines internship lifecycle states (PENDING, APPROVED, REJECTED, FILLED) representing the approval and completion workflow.
- InternshipLevel: Defines internship difficulty levels (BASIC, INTERMEDIATE, ADVANCED) which determine student eligibility based on year of study.

2.2.3. Control Layer

Manager Classes

- UserManager: Handles user authentication, registration, and profile management.
- InternshipManager: Manages the complete internship lifecycle including creation, filtering, approval/rejection, visibility toggling, and statistics generation.
- ApplicationManager: Coordinates the application process between students, internships, and company representatives.
- SystemManager (Singleton): Implements the Singleton design pattern to provide a centralised access point for all system components. It manages dependency injection and session-based filter settings. The singleton pattern ensures a single instance manages all system-wide states. It serves as an
- DataManager: Handles all file I/O operations for data persistence.
- InternshipRepository: A helper manager to manage data storage and retrieval for each InternshipOpportunity
- ApplicationRepository: A helper manager to manage data storage and retrieval for each Application
- UserManager: A helper manager to manage data storage and retrieval for each User (includes Student, CareerCenterStaff, CompanyRepresentative)

Utility Classes

- IdGenerator: Generates unique identifiers for internships (INT000001 format) and applications (APP000001 format) using sequential numbering based on repository size.
- InternshipFilterSettings: Encapsulates filter criteria for internship searches, supporting multiple filter dimensions (status, major, level, dates, slots, application status). Persists for the user's entire session.

2.2.4. Boundary Layer

- BaseMenu (Abstract Class): Provides common functionality for all menu classes including password management, input validation, user information display, and logout handling.
- MainMenu: Serves as the application entry point, presenting login and registration options.
- StudentMenu: Implements the student-specific interface with options for viewing internships (with filtering), applying for internships, viewing application status, accepting placements, and requesting withdrawals.
- CompanyRepresentativeMenu: Provides company representative functionality including internship creation, listing management, application review, and visibility control.
- CareerCenterStaffMenu: Implements administrative functions including approval workflows for representatives and internships, withdrawal request management, and comprehensive reporting capabilities.
- FilterUIHelper: Utility class that provides reusable filter configuration dialogs for consistent user experience across different menu classes
- Main: The application entry point that initialises the SystemManager singleton, loads persisted data, and launches the MainMenu.

2.3. Object-Oriented Principles Applied

2.3.1. Encapsulation

Example of Application in the System:

Student Class: Encapsulates application logic by keeping appliedInternships and acceptedInternshipID as private fields, providing controlled access through methods like canApplyForMore(), applyForInternship(), and acceptInternship() that enforce business rules (max 3 applications, only 1 acceptance).

```
- appliedInternships: List (private)
- acceptedInternshipID: String (private)
+ canApplyForMore(): boolean (enforces business rule)
```

Benefits Demonstrated: Data integrity is maintained through controlled access, implementation details are hidden allowing future changes.

2.3.2. Polymorphism

Example of Application in the System:

User Hierarchy (Runtime Polymorphism): The abstract User class defines abstract methods getUserType() and getRole() that are implemented differently by each subclass. This allows MainMenu.handleLogin() to work with User references without knowing the concrete type, routing to appropriate menus based on runtime 3 type.

Benefits Demonstrated: Code reusability through common interfaces, flexibility in adding new user types without modifying existing code, and simplified routing logic.

2.3.3. Inheritance

Example of Application in the System:

```
User (abstract)
     Student
     CompanyRepresentative
     CareerCenterStaff
```

- Common inherited attributes: userID, name, password (protected access)
- Common inherited methods: getUserID(), getName(), changePassword(), validatePassword()

Specialized attributes:

- Student adds: yearOfStudy, major, appliedInternships, acceptedInternshipID
- CompanyRepresentative adds: companyName, department, position, isApproved, createdInternships
- CareerCenterStaff adds: staffDepartment

Abstract method implementation: Each subclass provides concrete implementation of getUserType() and getRole()

Benefits Demonstrated: Elimination of code duplication across similar classes, establishment of clear hierarchical relationships (IS-A), and centralized modification of common behavior in parent classes.

2.3.4. Abstraction

Example of Application in the System:

Abstract User Class: Defines the essential concept of a "user" without specifying concrete user types. Forces subclasses to implement getUser Type() and getRole() while providing common authentication logic:

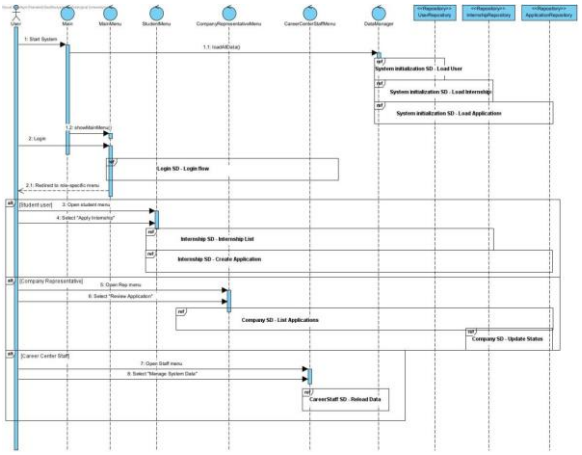
```
abstract class User {
     +  validatePassword(inputPassword:  String):  boolean  //
     Concrete implementation
     + getUserType(): String // Abstract - must be implemented
     by subclasses
     + getRole(): String // Abstract - must be implemented by
     subclasses
}
```

Benefits Demonstrated: Simplified interfaces for complex operations, separation of "what" from "how", flexibility to change implementations without affecting clients, and reduced cognitive load on developers using the classes.

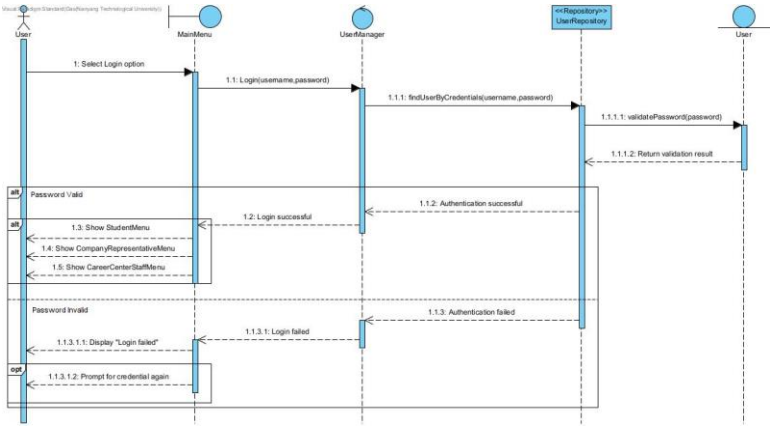
3. UML Sequence Diagram

(Please refer to the sequence diagram folder for full-resolution images)

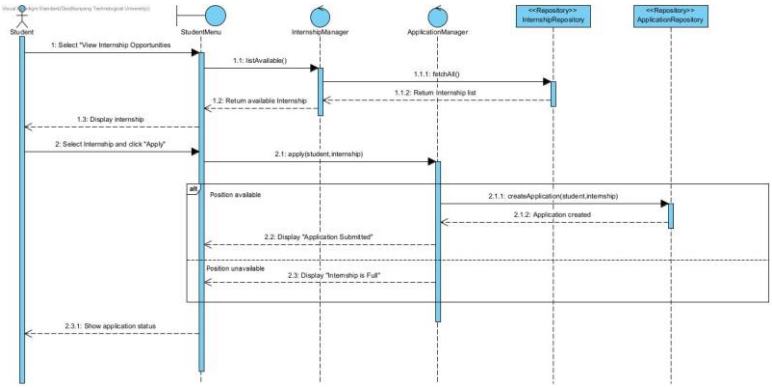
1. Master Sequence Diagram



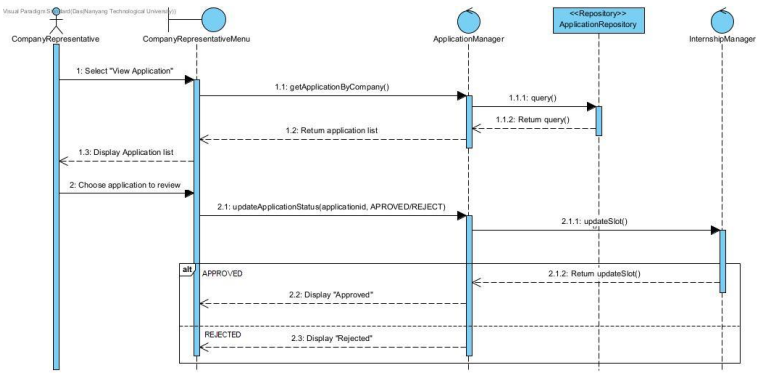
2. Login Sequence Diagram



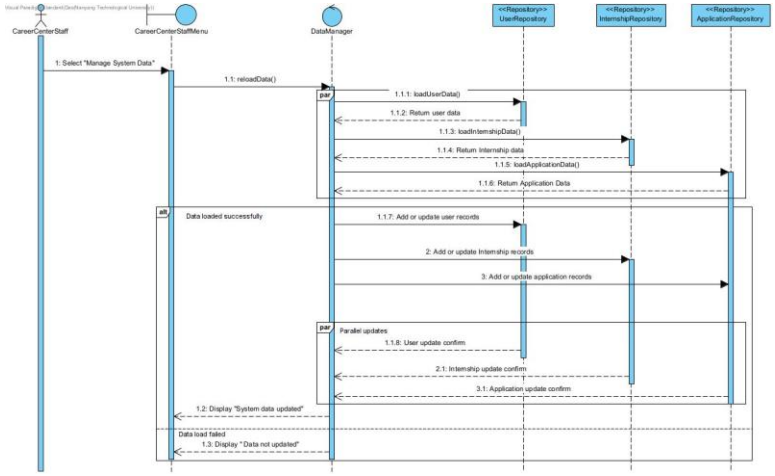
3. Internship Sequence Diagram



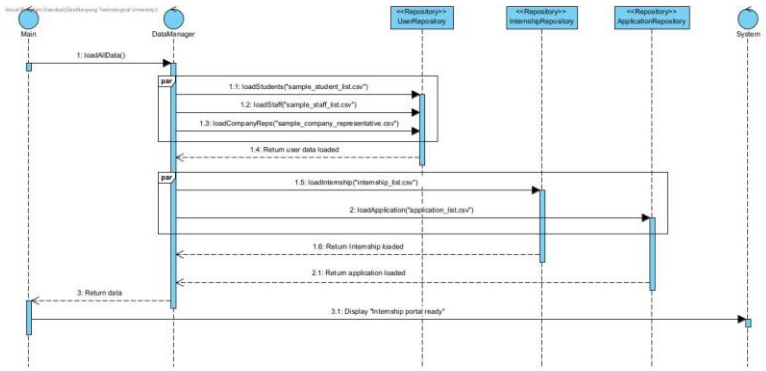
4. Company Representative Diagram



5. Career Staff Sequence Diagram



6. System initialisation Sequence Diagram



4. Test Cases and Results

Test Cases	Expected Behavior	Failure Indictors	Pass/Fail
All users have user ID, Name and password	user.txt has relevant information	missing information	Pass
All user valid login	User able to log in successfully	unable to login despite correct information / able to login despite incorrect information	Pass
All user Invalid ID	User receives "Login failed. Invalid credentials or account not approved."	able to login with invalid ID/Failure to print message	Pass
All user logout	User able to log out	Unable to logout despite choosing log out option	Pass
All user password change	User able to change password	password not change / updated	Pass
Student user registration is automatic from user.txt	Student user are able to log in without registration	student user unable to log in despite student information inside of the user.txt file	Pass
Student user for Y1/Y2 students only can view basic internships with visibility toggled ON	year 1 and year 2 only able to view basic level internships	internship that are not basic level are being shown as well.	Pass
Student user for Y3/Y4 students can view all internships opportunity with visibility toggled ON	year 3 and year 4 able to view all level internships	Internships hidden from year 3 and year 4 students	Pass
Student user cannot view internship when visibility toggled OFF	with visibility toggled off, student should not see any internship that has been toggled OFF	Internships are still being listed even though visibility toggled OFF	Pass
Student user should be able to still be able to see internship applied for even after visibility is toggled OFF ("PENDING", "SUCCESSFUL", UNSUCCESSFUL")	internships applied for still able to be viewed with application status being shown	unable to view internships	Pass
Student user should be able to see status change for internship applied (PENDING -> SUCCESSSFUL, PENDING -> UNSUCCESSFUL) even when visibility OFF	internship application status changed according to company decision	changes not reflected	Pass
Student user can accept the internship placement if application status is "SUCCESSFUL".	students able to accept the internship placement	students unable to accept the internship	Pass
Student user can only accept one internship placement	student shouldn't be able to accept more than 1 internship, all others shall be withdrawn	students able to accept more than 1 or internship application not withdrawn.	Pass
Student unable to apply for more internships once he/she has accept an internship placement	student unable to apply for more internships. "You cannot apply for more internships. You have already accepted an internship placement. "	student still able to apply for new internship opportunites	Pass
Student user allow to request withdrawal for their internship application BEFORE placement confirmation	students able to request withdrawal before placement confirmation	students unable to request withdrawal	Pass
Student user allow to request withdrawal for their internship application AFTER placement confirmation	students able to request withdrawal after placement confirmation	students unable to request withdrawal	Pass
Company representatives able to register as a representative of specific company	successfully registered as a company representative user	unable to register	Pass
Company representative able to login after approval	successfully login	unable to login despite approval	Pass
Company representative able to create internship opportunities	successfully created internship opportunity (capped at 5)	unable to create internships / creating over the limit of 5	Pass
Company representative's internship FILLED, students will not be able to apply anymore	unable to view internship that has already been filled up	still able to see / apply for the internship	Pass
Company representative able to view applicants details for each of their internship opportunities	representative able to view the applicant details, major , year of study etc	unable to view details	Pass
Company representative able to approve or reject internship applications	representative able to approve or reject applicants	unable to approve / reject	Pass
Company representative's internship opportunity status becomes "Filled" only when all available slots are confirmed by students	Internship opportunity becomes filled and unable to view when all available slots are filled	still able to view and not filled	Pass
Company representative able to toggle visibility.	able to toggle each internship individually	unable to toggle / toggling does not actually change visibility	Pass
Career Center Staff registration via user.txt file	registration is automatic via user.txt file	unable to log in despite valid credentials and information is logged in user.txt	Pass
Career Center Staff able to authorize the account creation of Company Representative	Successful creation with company representative able to log in	company representative unable to login with right credentials after creation / failed creation	Pass
Career Center Staff able to approve or reject the internship opportunities submitted by company representatives	Successful approval/rejection	students able to view the internships that are rejected / unsuccessful approval / rejection	Pass
Career Center Staff able to approve or reject student withdrawal requests before placement confirmation	Succesful approval / rejection	unable to approve or reject the withdrawal request	Pass
Career Center Staff able to approve or reject student withdrawal requests after placement confirmation	Succesful approval / rejection	unable to approve or reject the withdrawal request	Pass
Career Center Staff able to generate comprehension reports regarding internship opportunities created	successful creation with correct details displayed	incorrect display, unable to display	Pass

(Please refer to the excel notebook)

5. Reflection on Design Choices

5.1. What Worked Well

What worked well for our project is the adoption of the Model-View-Controller architecture which helps us separate our code cleanly, making the development process much easier when working in a team. In the MVC architecture, we separate the user interface from data logic, making it easy to locate and modify specific functionality and implement business logic. Additionally, this design makes it easily testable since we can perform unit testing on each individual component of our application separately.

An additional benefit of this design pattern is that our code is modular. The use of specialised Repository classes with a fixed template makes it easy to swap out for a concrete database implementation later on.

Using inheritance hierarchy, our use of a User abstract base class allows us to extend our system easily when we want to add new user types in future.

5.2. Challenges and Improvement

Our initial draft consisted of tightly coupled code as many classes were dependent on a singleton instance of the SystemManager class. After we completed our initial draft, we refactored our code to adopt good design practices. This included decoupling all data logic from the SystemManager to better align our design with the MVC architecture and the Single Responsibility Principle in SOLID design principles. Instead of having SystemManager act as a data store that other classes directly accessed, we changed this into a layered dependency injection architecture. We created dedicated repository classes to handle all data access operations, leaving SystemManager to serve as an application context and dependency container. Each manager class declares its dependencies through constructor parameters which are instantiated once during application startup. Now, managers depend on repositories rather than on each other or a central data store. Our new approach provided benefits for testing and maintainability since we can perform unit tests on each component in isolation.

An improvement for our application would be to implement proper error handling. Currently, some methods just return null to indicate failure. We can solve this by designing a proper class

with generics, implementing an exception handling strategy so that it would help to improve clarity and robustness for both developers and users. As the inconsistent handling of nulls led to unexpected crashes, this is a possible future improvement for our project.

As for data persistence, since we are explicitly forbidden from using database applications, we tried to implement a workaround using file-based storage with text files. However, there are limitations to this since we compromise data security by storing passwords and sensitive user details in plaintext. For future improvements, we can use a proper database with hashed passwords. For validation, we can use centralised validation rules so that it would improve code reusability and maintainability as validation logic is usually duplicated across layers.

5.3. Key Lessons Learnt

Through this project, we learnt that design patterns should address real architectural challenges rather than being applied for the sake of doing so. Sometimes, strictly implementing design patterns without analysing the project requirements thoroughly results in unnecessary complexity. We adopted for a singleton application context instead of more complex patterns like the Strategy pattern, since we are only building a CLI application. Our takeaway is that design patterns should make the code clearer, not more complex.

Implementing a clear separation between the boundary, control and entity layers saved a lot of time with our team's development and testing. The MVC architecture reduced the load of each team member since we focused on implementing one layer at a time. This meant that making changes in one layer rarely caused unintended effects for other team members, improving maintainability and the speed at which we could complete our code. Through this project, we encountered many trade-offs that we made after careful discussion. Our centralised use of dependency injection with SystemManager introduced some coupling to this central class. The inheritance for the User hierarchy is an intuitive design but offers less flexibility compared to other approaches that use composition. We learnt to balance both benefits and limitations as it is impossible to achieve a perfect design.

Lastly, we learnt that building an extensible system needs upfront planning. While the use of abstract classes, interfaces, enums and polymorphism allows for the addition of new features and types, it could violate open/closed principles on top of other SOLID design principles

which was what we experienced, resulting in costly time spent to refactor the code. Hence, planning for extensibility keeps our codebase readable and adaptable to future requirements without requiring extensive refactoring or breaking existing functionality.