Internship Placement Management System

SCED, Group-6

Members: **ARCHISHA** DEB BARMA, **CHARLES** LEE JUN NGAI, CHU **YI XIN**, **HARITH** BIN FA'IZAL, MAHALIK **AMISHA** ALAMA, **WAI YAN** MIN KO KO

Link to Github: https://github.com/waiyanminkoko/internship-placement-management-system-sc2002.git

Design Considerations

The design of our Internship Placement Management System is based on the **Object-Oriented Principles (OOP)** taught in our lectures. We also integrated a **Layered Architectural** style to achieve separation of concerns and manage complexity effectively.

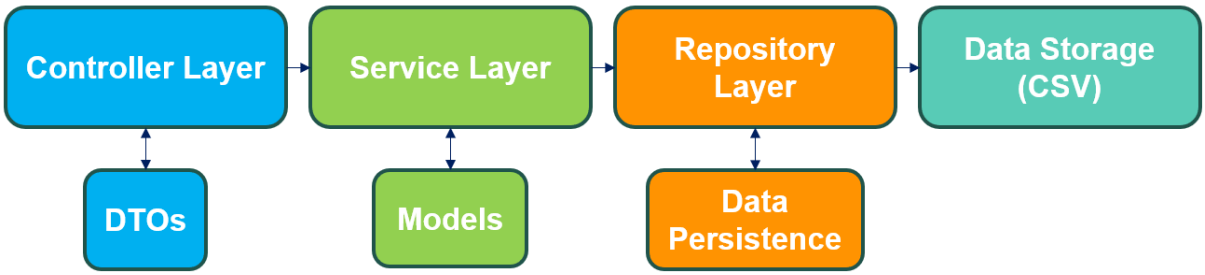


Figure 1: Layered Architectural Style

Use of Object-Oriented Concepts

Encapsulation is achieved by using private data fields, which are accessed through public getter and setter methods, while only exposing core business logic via service classes. This effectively hides internal implementation details and minimizes unintended dependencies. This design choice directly supports the design goal of minimizing the ripple effect of any necessary changes.

Inheritance is utilized to model real-world relationships in the project. A key abstract superclass, `User`, is used and is extended by specific subclasses such as `Student`, `CompanyRepresentative`, and `CareerCenterStaff`, our 3 main users of this system. This structure promotes polymorphism and fosters code reuse, which allows the system to handle different user-type objects uniformly while maintaining their unique functionalities.

Abstraction is implemented through interfaces for data repositories and services. This approach allows seamless replacement of underlying storage implementation, such as switching from CSV files to a database (which is mainly used in the industry), without requiring modifications to our pre-existing code. This significantly enhances the system's extensibility and maintainability for future upgrades.

These object-oriented concepts ensure **clear organization** and **modularization**. Classes are kept maintainable through *high cohesion,* meaning each class is focused on a single responsibility. There is *loose coupling* between classes and modules to limit the impact of changes made to the overall system, adhering to the design goals emphasized throughout the course.

Design Patterns

As the project grows in complexity, the design pattern plays a key role in ensuring maintenance and extensibility of the system. To do so, we utilized several design patterns.

The **Repository Pattern** is used to manage the project's dependency to minimize the effect of modifications made to the system. By abstracting data persistence specifics, it effectively separates the business logic from the underlying storage format used (CSV).

The **Service Layer Pattern** is key for centralizing business rules and coordinating transactions through managing operations that involve multiple entities or repositories in a single business process. This guarantees code consistency and ensures the reusability of the application's logic.

Data Transfer Objects (DTOs) are essential for maintaining a stable and secure API interface. By separating the internal domain models from the external API representations, **DTOs** enhance security, support versioning, and standardize the interface contract.

Dependency Injection is facilitated by the Spring Framework (Spring Boot), which promotes **loose coupling** and improved testability. Dependencies are injected rather than hard-coded, which adheres to the *Dependency Inversion Principle (DIP)* of the *SOLID* (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion) design principles.

Exception handling is also implemented separately to ensure system robustness and consistent error management throughout the system.

Trade-offs & Alternative Considerations

Due to assignment requirements, we are restricted to using CSV files as our data storage, which limits our ability to utilize advanced querying capabilities that can be achieved through SQL database queries. Any complex filtering, joining, or aggregating of data must be implemented manually in code, rather than using powerful SQL queries to achieve the same results easily. This is addressed by implementing in-memory caching (data from the CSV files is loaded into memory) and thread-safe concurrency control (ensures that when multiple users or processes try to access or modify the cached data at the same time, the system prevents data corruption or inconsistencies), thereby achieving a balance between system simplicity and adherence to the constraints.

The initial implementation using **Spring** and **Data Transfer Objects (DTOs)** introduces complexity and difficulty as this is the first time we are using the Spring Framework and working between multiple Java files and CSV files, but this establishes a professional-grade architecture, which we learnt from as we work on this project. This foundational structure became crucial as the project evolved, ensuring the system's long-term maintainability and scalability.

These design patterns and principles enhance the system's clarity, extensibility, and maintainability, which are extremely important as more files are created in the project.

Additional Features

Our system includes two additional features to enhance the usability of the app and help with system testing and user administration.

Account Creation for Students and Company Representatives

Career Center Staff members can create accounts for both Students and Company Representatives within the app. This streamlines user onboarding and helps with creating student and company representative accounts without having to edit within the CSV files.

Downloadable PDF Reports

Career Center Staff can generate comprehensive system reports and download them as PDF documents. Reports can be filtered by multiple criteria and provide visual summaries of internship placements, applications, and system metrics. This feature is implemented by using publicly available libraries such as ‘jsPDF’ and ‘html2canvas’ to easily generate the PDF versions of the report.

UML Class Diagram – Simplified Backend Architecture

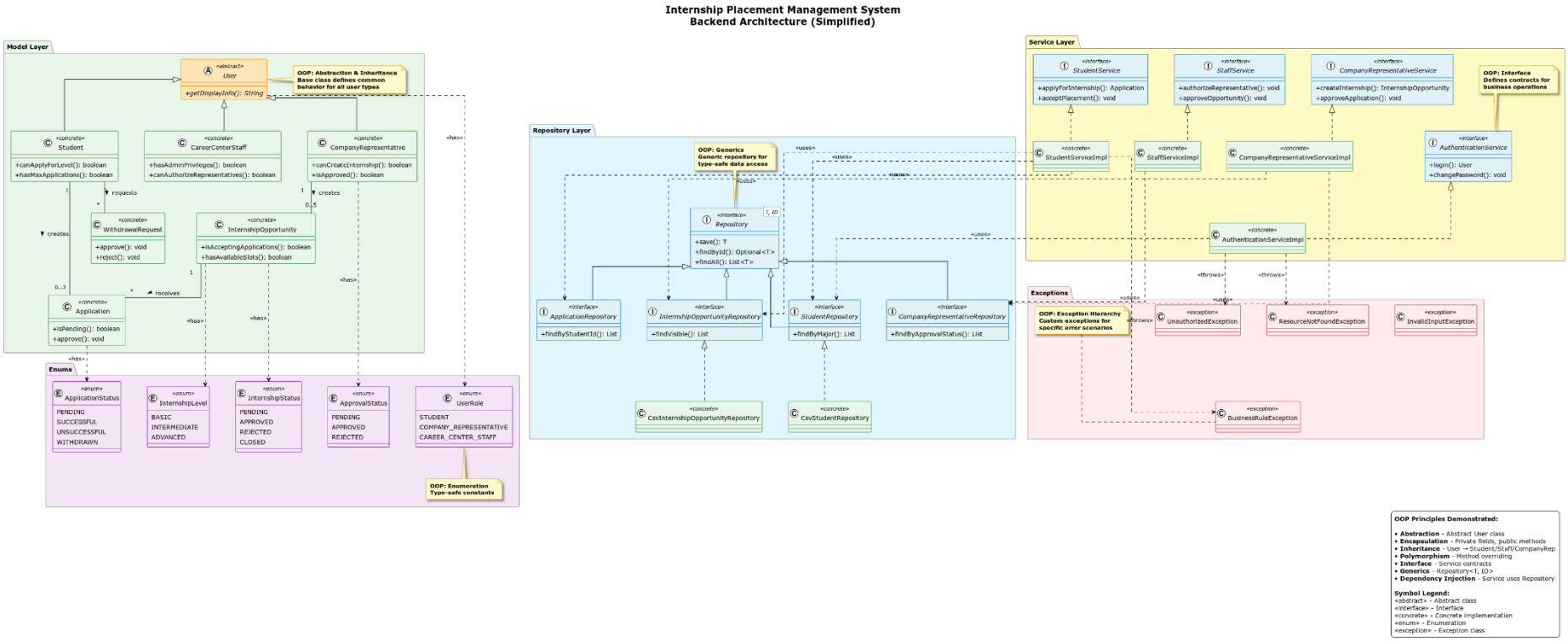


Figure 2: Simplified Backend Architecture UML Class Diagram

Complete Class Diagram can be found in the GitHub Repo (Path: docs\uml\class-diagrams):

https://github.com/waiyanminkoko/internship-placement-management-system-sc2002/tree/e35739447b4989462c593e72a10b0bf574242ee4/docs/uml/class-diagrams

Overview

The UML Class Diagram above gives an overview of how the main components in our Internship Placement Management System are structured.

We organised the diagram into clear layers, so it is easier to understand how the system works internally. The layout follows a layered architecture, which helps keep the system clean, modular, and easier to maintain:

1. Model Layer

The Model Layer contains the main classes our system uses to represent real-world objects.

User (Abstract Class)

`User` is the base class for all types of users in the system. It contains common attributes and methods like `getDisplayName()`, which are shared across all user roles. Since `User` cannot be instantiated directly and only exposes common behaviour, this demonstrates **abstraction** in our design.

`Students`, `CompanyRepresentative`, `CareerCenterStaff`

These classes extend `User` and add their own role-specific behaviour:

- Students can apply or withdraw from internship opportunities
- Company Representatives can create internships
- Career Centre Staff can approve applications and manage representatives

This part of the diagram shows **inheritance**, since all 3 roles share the same base class.

It also supports **polymorphism**, because different subclasses can provide their own implementations while still being treated as `User` objects when needed.

`InternshipOpportunity`, `Application`, `WithdrawalRequest`

These represent the core interactions in the system:

- Companies create internship opportunities
- Students submit applications or withdrawal requests
- Applications have statuses that change over time (e.g., `PENDING`, `APPROVED`)

2. Repository Layer

This layer handles all the data access operations for the system.

We use the **Repository Pattern**, which separates how data is stored from how the system uses that data.

The repositories contain the storage logic, for example:

- Opening and reading CSV files
- Searching for specific records
- Converting CSV rows into model objects

- Saving updated records back to the file

This is handled by classes such as:

- `CsvStudentRepository`
- `CsvApplicationRepository`
- `CSVInternshipOpportunityRepository`

From the UML class diagram, we can see that the **Repository Layer** is built around a generic `Repository<T, ID>` interface. Each specific repository (such as `StudentRepository`, `ApplicationRepository`, and InternshipOpportunityRepository) either implements this interface directly or has a CSV-based version that does. Some repositories also include additional query methods like `findByStudentId()` and `findVisible()`, which are shown in the diagram.

The diagram also shows that **Service Classes** depend on these repositories. This makes it clear that repositories are responsible for handling data access, while the services focus on the system's business rules. By keeping these roles separate, the diagram reflects the **Repository Pattern**, which helps make the system more modular and easier to maintain.

3. Service Layer

This layer contains the main operations that different users can perform.

Service Interface:

- `StudentService`
- `StaffService`
- `CompanyRepresentativeService`
- `AuthenticationService`

Implementations:

- `StudentServiceImpl`
- `StaffServiceImpl`
- `CompanyRepresentativeServiceImpl`
- `AuthenticationServiceImpl`

These classes contain the actual logic behind features such as:

- Applying for internships
- Approving or rejecting applications
- Creating new internship postings
- Logging in with credentials

The services call the repositories behind the scenes. This shows **encapsulation**, because the details of how data is stored are hidden from the service layer.

4. Exception Layer

To handle invalid actions or rule violations cleanly, we created a custom **Exception Hierarchy**:

- `BusinessRuleException`
- `UnauthorizedException`
- `InvalidInputException`
- `ResourceNotFoundException`

These exceptions make error handling more meaningful and help separate normal logic from error cases.

5. Enum Layer

The **enums** define fixed system values such as:

- ApplicationStatus
- ApprovalStatus
- InternshipLevel
- IntershipStatus
- UserRole

Using **enums** helps ensure that these values are consistent and type-safe, instead of relying on hardcoded strings. This reduces errors and makes the code easier to maintain, since only valid values can be used for things like application status or user roles.

Object-Oriented Principles (OOP) Demonstrated

In our system, we made use of several key OOP principles to keep the design organised and easy to maintain.

Abstraction is demonstrated through the `User` abstract class, which captures the common behaviour shared across all user types without exposing unnecessary details. This allows us to treat all users in a consistent way while still letting each role define its own behaviour.

We also use **abstraction** in the `Repository<T, ID>` interface, which outlines the general operations (like saving and retrieving data) without tying the system to a specific storage method. This helps reduce complexity, since other parts of the system only interact with what they need to know.

We made heavy use of **inheritance** in the Model Layer. Our `Student`, `CompanyRepresentative`, and `CareerCenterStaff` classes all extend the `User` class, allowing us to reuse common logic while still giving each role its own specific features. This naturally leads to **polymorphism**, because these subclasses, and even our service and repository implementations, can be used interchangeably through their shared interfaces.

Another principle we focused on was **encapsulation**. Each class in our system manages its own data and internal rules, exposing only the methods that other components need. This helps prevent accidental misuse of data and keeps each class responsible for its own behaviour. To support this, we also relied on interfaces in the Service Layer (such as `StudentService` and `AuthenticationService`) to define clear contracts for the actions each role can perform.

In the Repository Layer, we used **generics** so that one repository interface could handle multiple entity types safely and consistently.

Finally, our diagram also reflects **dependency injection**, because the service depends on repository interfaces rather than concrete CSV implementations. This reduces coupling and makes it much easier to modify or test parts of the system without affecting the rest.

UML Sequence Diagram

Company Representative – Approve Student's Internship Application

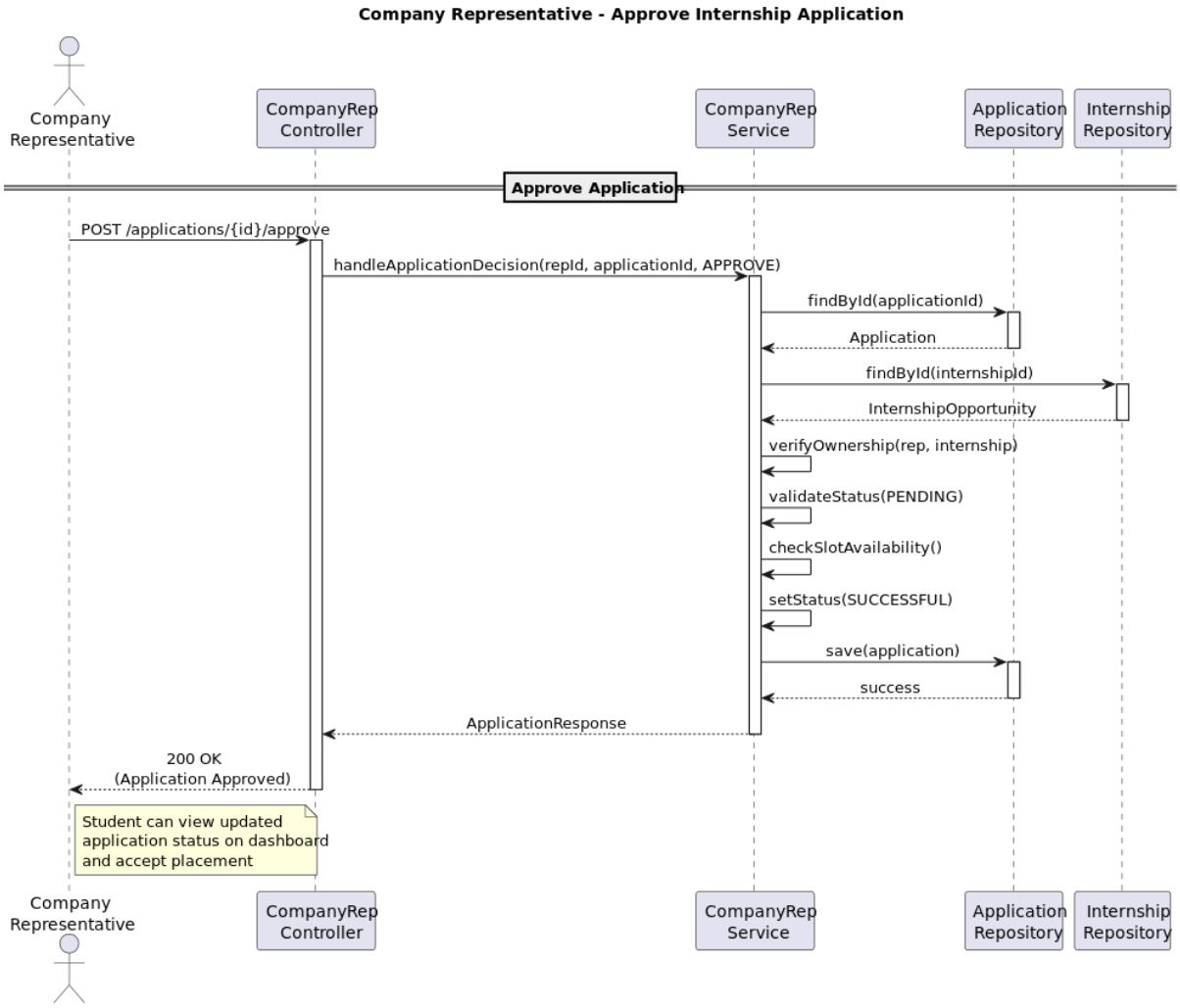


Figure 3: UML Sequence Diagram – Company Representative's Internship Application Approval

The in-depth Sequence Diagrams can be found in the GitHub Repo (Path: docs\uml\sequence-diagrams):

https://github.com/waiyanminkoko/internship-placement-management-system-sc2002/tree/d50639db095eaa38ead739df2f871228f29c86f4/docs/uml/sequence-diagrams

Sequence Diagram Explanation

This sequence diagram shows how our system handles the process when a company representative approves a student's internship application. The interaction starts when the company representative clicks the approval action in the interface, which is received by the `CompanyRepController`.

The controller then passes the request to the `CompanyRepService`, where the main business logic happens. Here, the service retrieves the relevant application from the `ApplicationRepository` and checks the corresponding internship details through the `InternshipRepository`. This is to make sure that the internship is still accepting applications and has available slots before continuing.

After all checks are done, the service updates the application status to reflect the approval and saves the changes back into the repository. The updated result then flows back through the controller to the company representative.

As shown at the end of the diagram, the student will later be able to see the updated application status on their dashboard and proceed with accepting the placement if they wish.

Overall, the diagram helps illustrate how the different layers in our system work together to support a smooth and consistent approval process.

Reflection – Archisha, Harith, Amisha

During the development of our Project System, we encountered several challenges that tested both our technical proficiency and how we work together as a team. One of the main difficulties was familiarizing ourselves with the Spring Boot framework, which was new to most of us. Unlike the standalone Java programs we had previously developed, Spring Boot required us to understand concepts such as dependency injection, layered architecture, and service-oriented design. Integrating these components while ensuring smooth interaction between controllers, repositories, and service classes was initially confusing and required a lot of experimentation.

Another major challenge was the requirement to use CSV files for data storage instead of a relational database. The lack of structured querying, transaction support, and indexing made it difficult to perform operations such as filtering, sorting, and concurrency control. This limitation forced us to work around it by using additional logic implementation to simulate database-like behaviour.

Beyond the technical aspects, we also faced issues collaborating when managing multiple contributors on GitHub. Managing conflicts, inconsistent code formatting, and differences in implementation approaches occasionally slowed progress as we had to worry about getting on the same page. In addition, updating UML diagrams to match the changing code structure required constant communication and updates within the team.

To address these challenges, we began by dividing tasks based on each member's strengths and areas of interest while keeping updated regularly via regular updates, progress tracking, and code reviews. This structure allowed us to synchronize our work effectively and ensure that integration occurred smoothly.

For the technical difficulties, we collectively explored Spring Boot documentation, online resources, and example projects to strengthen our understanding of its framework and annotations. Implementing the Repository and Service Layer patterns helped us achieve a cleaner separation between business logic and data access, reducing code coupling and improving maintainability.

To compensate for the limitations of CSV storage, we implemented in-memory caching mechanisms and thread-safe access controls. This approach enhanced performance and prevented data inconsistencies when multiple operations occurred simultaneously. We also refined our use of Git branching, pull requests, and merging strategies, which significantly reduced conflicts and improved code stability.

This project provided us with valuable exposure to real-world software engineering practices and reinforced our understanding of Object-Oriented Programming (OOP) principles such as encapsulation, inheritance, abstraction, and polymorphism. We learned how these

principles interact to produce modular, scalable, and maintainable code. Seeing them applied in an actual multi-layered system deepened our appreciation for well-structured software design.

We also gained practical experience with software architecture and design patterns, including the Repository Pattern, Service Layer Pattern, and Data Transfer Objects (DTOs). These patterns taught us how to design systems that are extensible, loosely coupled, and adaptable to future changes. Working within a Layered Architectural style emphasized the importance of separating concerns, which made debugging, testing, and future enhancements far more manageable.

In addition to technical skills, this project strengthened our collaboration, communication, and problem-solving abilities. We learned the value of version control, consistent coding standards, and constructive feedback. Most importantly, we developed a mindset of continuous learning and understanding that effective software design is not achieved instantly, but through iteration and reflection.

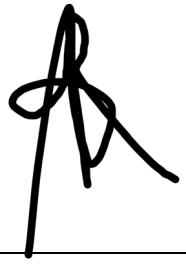
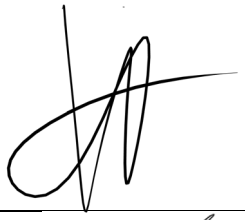
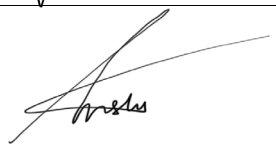
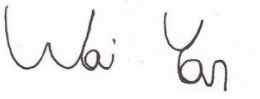
If given more time to enhance the system, we would consider integrating a relational database such as MySQL or PostgreSQL to replace CSV files. This would allow for more efficient data management since it has advanced querying and improved reliability. We would also like to implement Spring Security to provide more robust authentication and role-based authorization.

In addition, deploying the application on a web hosting platform would simulate real-world usage conditions and enable user testing in a live environment. Incorporating unit testing frameworks such as JUnit would further ensure the correctness and stability of our code. We also recognize the potential to refine the front-end interface for better usability and user engagement.

Overall, this project deepened our understanding of full-stack software design, from conceptual modelling to implementation, debugging, and testing. It taught us that well-structured design principles are essential not just for functionality, but also for long-term maintainability and scalability.

Declaration of Original Work for SC2002/CE2002/CZ2002 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 beawarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course (SC2002/CE2002 CZ2002)	Lab Group	Signature /Date
ARCHISHA DEB BARMA	SC2002	SCED	
CHARLES LEE JUN NGAI	SC2002	SCED	
CHU YI XIN	SC2002	SCED	
HARITH BIN FA'IZAL	SC2002	SCED	
MAHALIK AMISHA ALAMA	SC2002	SCED	
WAI YAN MIN KO KO	SC2002	SCED	

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.