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 be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course (SC2002/CE2002 CZ2002)	Lab Group	Signature /Date
Justin So Wei Lun	SC2002	SCE2	[illegible] 20 Nov 2025
Liang Yu Ting, Joie	SC2002	SCE2	[illegible] 20 Nov 2025
Sheryl Ng Wenhui	SC2002	SCE2	[illegible] 20 Nov 2025
Tong Khanh Van	SC2002	SCE2	Van 20 Nov 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

This Internship Placement Management System has 3 types of users: Students, Company Representatives, and Career Center Staff. Students can use this system to apply for internships that Company Representatives post, while the Career Center Staff oversees this process. We applied our knowledge of OODP and SOLID design principles to develop this system.

1.1 BCE Architecture

We decomposed the problem using the BCE architecture and decided which classes each stereotype consisted of. This helps to provide a structured way to separate the responsibilities in the system and make the program easier to understand, maintain, and extend.

- **Boundary:** Classes that interact between the system and external. For example, StudentUI, CompanyRepUI, and CareerStaffUI.
- **Control:** Classes that manage the logic to coordinate and realize use cases. For example, StudentController, CompanyRepController, and CareerStaffController.
- **Entity:** Classes that hold the data and business rules of the system. For example, Student, CompanyRep, and CareerStaff.

1.2 Class Relationships

We determined the relationships between the classes. Some examples include:

MainApplication to IDataRepo is a use-a relationship. This allows the MainApplication to use the information from IDataRepo without the need to store the data permanently. It can also be loosely coupled from the specific implementation of data storage.

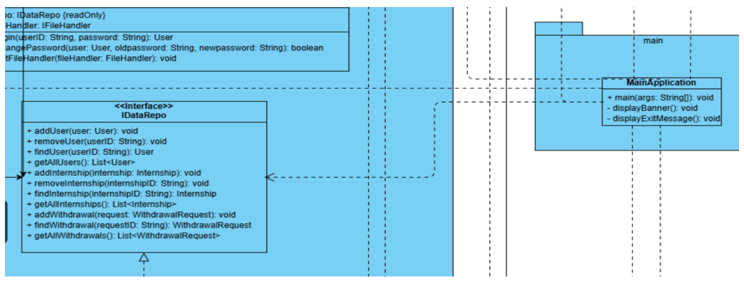

po: IDataRepo {readOnly}
Handler: IFileHandler
gin(userID: String, password: String): User
angePassword(user: User, oldpassword: String, newpassword: String): boolean
tFileHandler(fileHandler: FileHandler): void
<<Interface>>
IDataRepo
+ addUser(user: User): void
+ removeUser(userID: String): void
+ findUser(userID: String): User
+ getAllUsers(): List<User>
+ addInternship(internship: Internship): void
+ removeInternship(internshipID: String): void
+ findInternship(internshipID: String): Internship
+ getAllInternships(): List<Internship>
+ addWithdrawal(request: WithdrawalRequest): void
+ findWithdrawal(requestID: String): WithdrawalRequest
+ getAllWithdrawals(): List<WithdrawalRequest>
main
MainApplication
+ main(args: String[]): void
- displayBanner(): void
- displayExitMessage(): void

Figure 1: MainApplication and IDataRepo: use-a relationship

LoginUI to IFileHandler is a has-a relationship, where LoginUI contains IFileHandler as its data member. The LoginUI contains a handler to read from and write to CSV files. It focuses on user interaction, while the handler encapsulates the file manipulation logic. This makes maintenance simpler. This also makes replacing the file handling mechanism easier as the core logic of LoginUI will not be affected.

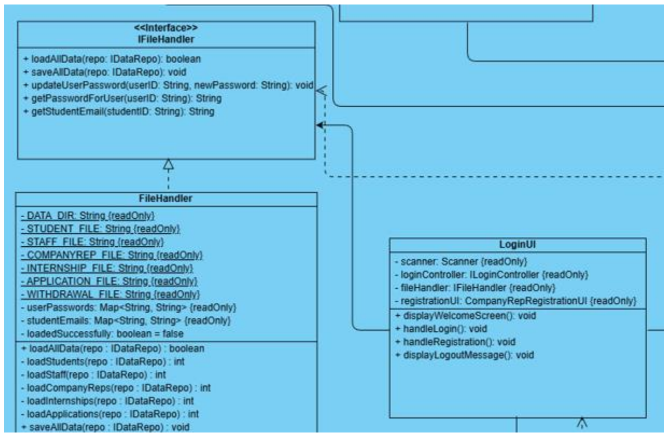

<<Interface>>
IFileHandler
+ loadAllData(repo: IDataRepo): boolean
+ saveAllData(repo: IDataRepo): void
+ updateUserPassword(userID: String, newPassword: String): void
+ getPasswordForUser(userID: String): String
+ getStudentEmail(studentID: String): String
FileHandler
- DATA_DIR: String {readOnly}
- STUDENT_FILE: String {readOnly}
- STAFF_FILE: String {readOnly}
- COMPANYREP_FILE: String {readOnly}
- INTERNSHIP_FILE: String {readOnly}
- APPLICATION_FILE: String {readOnly}
- WITHDRAWAL_FILE: String {readOnly}
- userPasswords: Map<String, String> {readOnly}
- studentEmails: Map<String, String> {readOnly}
- loadedSuccessfully: boolean = false
+ loadAllData(repo : IDataRepo) : boolean
- loadStudents(repo : IDataRepo) : int
- loadStaff(repo : IDataRepo) : int
- loadCompanyReps(repo : IDataRepo) : int
- loadInternships(repo : IDataRepo) : int
- loadApplications(repo : IDataRepo) : int
+ saveAllData(repo : IDataRepo) : void
LoginUI
- scanner: Scanner {readOnly}
- loginController: ILoginController {readOnly}
- fileHandler: IFileHandler {readOnly}
- registrationUI: CompanyRepRegistrationUI {readOnly}
+ displayWelcomeScreen(): void
+ handleLogin(): void
+ handleRegistration(): void
+ displayLogoutMessage(): void

Figure 2: LoginUI and IFileHandler: has-a relationship

Student, CompanyRep, CareerStaff to User is an is-a relationship. This promotes code reusability as the 3 user classes inherit all the attributes and methods from User. It also simplifies maintenance where any changes made to the User will also be automatically updated to the subclasses.

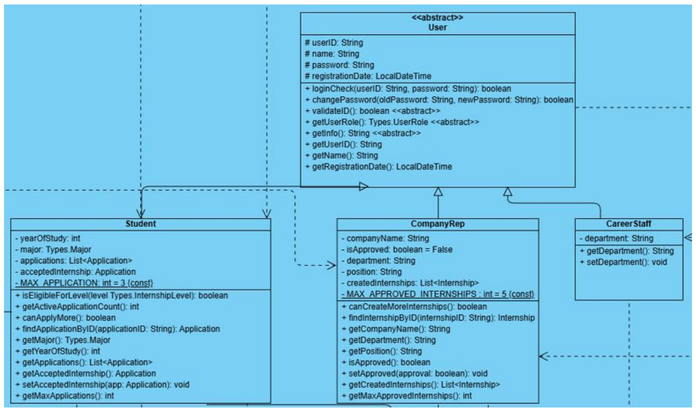

<<abstract>>
User
# userID: String
# name: String
# password: String
# registrationDate: LocalDateTime
+ loginCheck(userID: String, password: String): boolean
+ changePassword(oldPassword: String, newPassword: String): boolean
+ validateID(): boolean <<abstract>>
+ getUserRole(): Types.UserRole <<abstract>>
+ getInfo(): String <<abstract>>
+ getUserID(): String
+ getName(): String
+ getRegistrationDate(): LocalDateTime
Student
- yearOfStudy: int
- major: Types.Major
- applications: List<Application>
- acceptedInternship: Application
- MAX_APPLICATION: int = 3 {const}
+ isEligibleForLevel(level Types.InternshipLevel): boolean
+ getActiveApplicationCount(): int
+ canApplyMore(): boolean
+ findApplicationByID(applicationID: String): Application
+ getMajor(): Types.Major
+ getYearOfStudy(): int
+ getApplications(): List<Application>
+ getAcceptedInternship(): Application
+ setAcceptedInternship(app: Application): void
+ getMaxApplications(): int
CompanyRep
- companyName: String
- isApproved: boolean = False
- department: String
- position: String
- createdInternships: List<Internship>
- MAX_APPROVED_INTERNSHIPS : int = 5 {const}
+ canCreateMoreInternships(): boolean
+ findInternshipByID(internshipID: String): Internship
+ getCompanyName(): String
+ getDepartment(): String
+ getPosition(): String
+ isApproved(): boolean
+ setApproved(approval: boolean): void
+ getCreatedInternships(): List<Internship>
+ getMaxApprovedInternships(): int
CareerStaff
- department: String
+ getDepartment(): String
+ setDepartment(): void

Figure 3: Student, CompanyRep, CareerStaff and User: is-a relationship

1.3 Assumptions

1. All users have unique user IDs.
2. Company Representative ID is always a valid email address.
3. Career Center Staff IDs have the same format (3-10 characters using letters and digits).
4. Only Company Representatives register through the UI. Other users' information will be created and maintained using CSV files externally.
5. Each internship belongs to exactly one Company Representative, and its ownership is fixed. There are no shared or transferred internships.
6. CSV files are not edited concurrently when users make updates, but only after logging out.

1.4 Design Trade-Offs

For better extensibility, we traded simplicity. One area was using interfaces (e.g. IDataRepo, IFileHandler, controller interfaces), which increases the number of files and complexity. However, we felt that this abstraction allowed new functionalities to be added easily without modifying existing code. Separating responsibilities using the BCE architecture also led to more classes and lines of code to achieve a simple task as compared to writing all the logic in one class. However, this ensured that our system was organized, and each class had a single, clear responsibility, allowing for better maintainability, testability, and scalability.

1.5 Additional Features/Functionalities

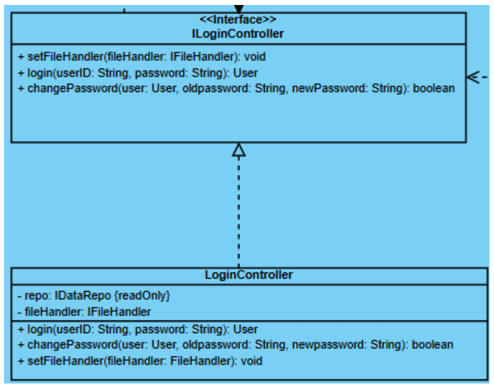
- For all users, there is a viewProfile() method for users to easily retrieve and display their personal information. This makes the system more user-friendly.
- For Career Center Staff, there is a viewStatistics() method to help them get a quick overview of the system (e.g. how many users, how many pending internships). They also have a viewAllUsers() method to display a consolidated view of user information. These make administrative processes more efficient for the staff as they can obtain data instantly.

2. Object Oriented Design Principles

2.1 Abstraction

Abstraction refers to showing only the essential features of an object to simplify its complexity. In our system, this is seen in the entity classes through the interfaces and the abstract User class.

Example: Interface Abstraction of ILoginController

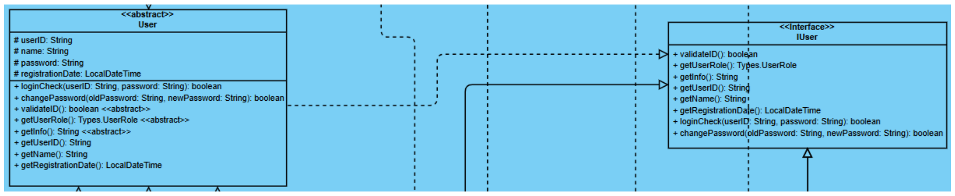


<<Interface>>
ILoginController
+ setFileHandler(fileHandler: IFileHandler): void
+ login(userID: String, password: String): User
+ changePassword(user: User, oldpassword: String, newPassword: String): boolean
LoginController
- repo: IDataRepo {readOnly}
- fileHandler: IFileHandler
+ login(userID: String, password: String): User
+ changePassword(user: User, oldpassword: String, newpassword: String): boolean
+ setFileHandler(fileHandler: FileHandler): void

Figure 4: Interface Abstraction of ILoginController

The ILoginController interface uses abstraction by defining what operations must exist but does not specify how they work, creating a contract for the class that implements it. With this, we can fix the essential methods that the LoginController needs and enable flexibility in the implementation of code without changing any dependent code.

Example: Abstract Class Abstraction in User



<<abstract>>
User
# userID: String
# name: String
# password: String
# registrationDate: LocalDateTime
+ loginCheck(userID: String, password: String): boolean
+ changePassword(oldPassword: String, newPassword: String): boolean
+ validateID(): boolean <<abstract>>
+ getUserRole(): Types.UserRole <<abstract>>
+ getInfo(): String <<abstract>>
+ getUserID(): String
+ getName(): String
+ getRegistrationDate(): LocalDateTime
<<Interface>>
IUser
+ validateID(): boolean
+ getUserRole(): Types.UserRole
+ getInfo(): String
+ getUserID(): String
+ getName(): String
+ getRegistrationDate(): LocalDateTime
+ loginCheck(userID: String, password: String): boolean
+ changePassword(oldPassword: String, newPassword: String): boolean

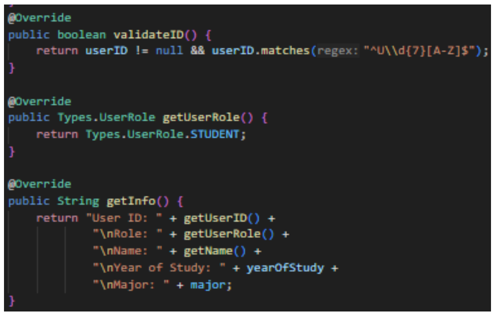
Figure 5: Interface Abstraction of IUser

The abstract class User implements the IUser interface, thus it has all the methods stated in IUser. The difference is that User implements some common functions that all roles have (i.e. loginCheck() and changePassword()) but continues to keep role-specific methods abstract for specific implementations within their classes.

2.2 Polymorphism

Polymorphism allows objects to behave differently depending on their class. In our system, one way this was demonstrated was through method overriding where subclasses provide their own implementations of the abstract methods defined in the parent class.

Example: Method Overriding of Abstract Methods in User

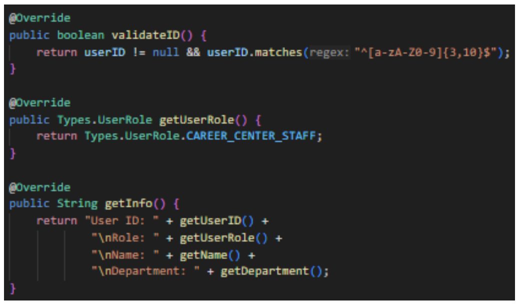


```java
@Override
public boolean validateID() {
    return userID != null && userID.matches(regex: "^U\\d{7}[A-Z]$");
}

@Override
public Types.UserRole getUserRole() {
    return Types.UserRole.STUDENT;
}

@Override
public String getInfo() {
    return "User ID: " + getUserID() +
            "\nRole: " + getUserRole() +
            "\nName: " + getName() +
            "\nYear of Study: " + yearOfStudy +
            "\nMajor: " + major;
}
```

Figure 6: Method Overriding in Student Class

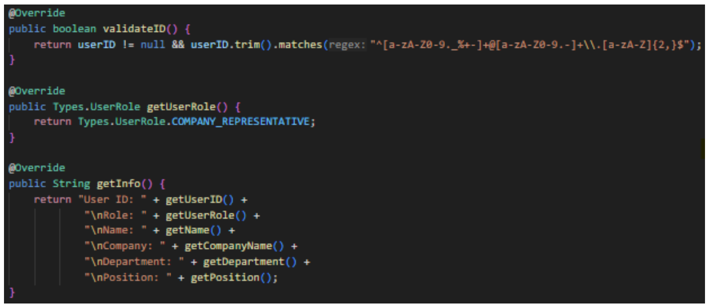


```java
@Override
public boolean validateID() {
    return userID != null && userID.matches(regex: "^[a-zA-Z0-9]{3,10}$");
}

@Override
public Types.UserRole getUserRole() {
    return Types.UserRole.CAREER_CENTER_STAFF;
}

@Override
public String getInfo() {
    return "User ID: " + getUserID() +
            "\nRole: " + getUserRole() +
            "\nName: " + getName() +
            "\nDepartment: " + getDepartment();
}
```

Figure 7: Method Overriding in CareerStaff Class



```java
@Override
public boolean validateID() {
    return userID != null && userID.trim().matches(regex: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$");
}

@Override
public Types.UserRole getUserRole() {
    return Types.UserRole.COMPANY_REPRESENTATIVE;
}

@Override
public String getInfo() {
    return "User ID: " + getUserID() +
            "\nRole: " + getUserRole() +
            "\nName: " + getName() +
            "\nCompany: " + getCompanyName() +
            "\nDepartment: " + getDepartment() +
            "\nPosition: " + getPosition();
}
```

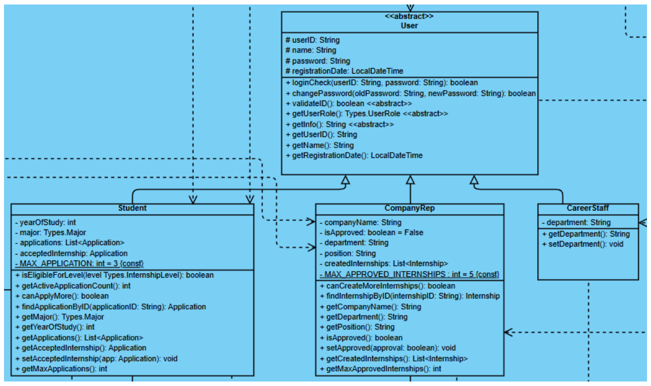
Figure 8: Method Overriding in CompanyRep Class

Each user type overrides the abstract methods from the User class, providing role-specific implementations with the same method call. The validateID() method ensures that each user has its own validation rules for object creation, getUserRole() displays the corresponding user type, and getInfo() enables individual ways of displaying the different details of each user. This use of polymorphism makes the system easier to extend in the future, allowing minimal changes to be made when adding new user types or modifying existing ones.

2.3 Inheritance

Inheritance is an is-a relationship, enabling code reuse and hierarchy. In our system, inheritance can be seen in the extension of user classes from User.

Example: Inheritance from User



<<abstract>>
User
# userID: String
# name: String
# password: String
# registrationDate: LocalDateTime
+ loginCheck(userID: String, password: String): boolean
+ changePassword(oldPassword: String, newPassword: String): boolean
+ validateID(): boolean <<abstract>>
+ getUserRole(): Types.UserRole <<abstract>>
+ getInfo(): String <<abstract>>
+ getUserID(): String
+ getName(): String
+ getRegistrationDate(): LocalDateTime
Student
- yearOfStudy: int
- major: Types.Major
- applications: List<Application>
- acceptedInternship: Application
- MAX_APPLICATION: int = 3 {const}
+ isEligibleForLevel(level Types.InternshipLevel): boolean
+ getActiveApplicationCount(): int
+ canApplyMore(): boolean
+ findApplicationByID(applicationID: String): Application
+ getMajor(): Types.Major
+ getYearOfStudy(): int
+ getApplications(): List<Application>
+ getAcceptedInternship(): Application
+ setAcceptedInternship(app: Application): void
+ getMaxApplications(): int
CompanyRep
- companyName: String
- isApproved: boolean = False
- department: String
- position: String
- createdInternships: List<Internship>
- MAX_APPROVED_INTERNSHIPS : int = 5 {const}
+ canCreateMoreInternships(): boolean
+ findInternshipByID(internshipID: String): Internship
+ getCompanyName(): String
+ getDepartment(): String
+ getPosition(): String
+ isApproved(): boolean
+ setApproved(approval: boolean): void
+ getCreatedInternships(): List<Internship>
+ getMaxApprovedInternships(): int
CareerStaff
- department: String
+ getDepartment(): String
+ setDepartment(): void

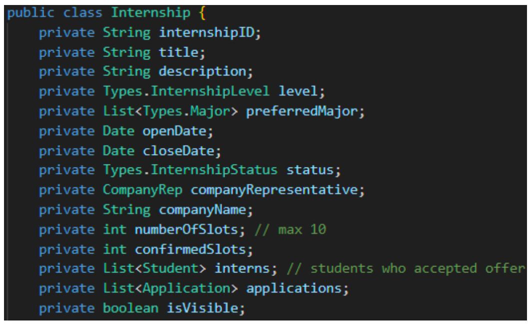
Figure 9: Inheritance of User in Student, CompanyRep and CareerStaff

As User attributes are protected, the subclasses (i.e. Student, CompanyRep, CareerStaff) can access them, alongside the concrete and abstract methods. This enables code reusability of the concrete methods (e.g. loginCheck() only has to be written once in the User class) and ensures that the essential abstract methods are implemented in a role-specific way.

2.4 Encapsulation

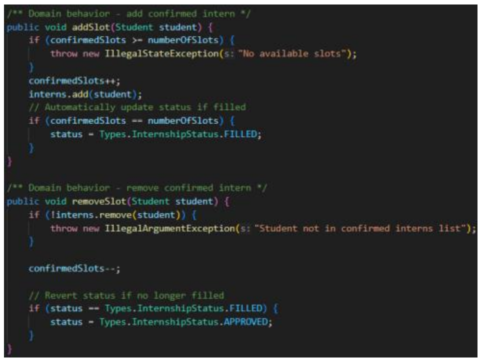
Encapsulation is the restriction of access to some of an object's components to ensure data integrity. In our system, encapsulation is demonstrated through private fields.

Example: Slot Management in Internship



```java
public class Internship {
    private String internshipID;
    private String title;
    private String description;
    private Types.InternshipLevel level;
    private List<Types.Major> preferredMajor;
    private Date openDate;
    private Date closeDate;
    private Types.InternshipStatus status;
    private CompanyRep companyRepresentative;
    private String companyName;
    private int numberOfSlots; // max 10
    private int confirmedSlots;
    private List<Student> interns; // students who accepted offer
    private List<Application> applications;
    private boolean isVisible;
```

Figure 10: Private Attributes of Internship Class



```java
/** Domain behavior - add confirmed intern */
public void addSlot(Student student) {
    if (confirmedSlots >= numberOfSlots) {
        throw new IllegalStateException(s: "No available slots");
    }
    confirmedSlots++;
    interns.add(student);
    // Automatically update status if filled
    if (confirmedSlots == numberOfSlots) {
        status = Types.InternshipStatus.FILLED;
    }
}

/** Domain behavior - remove confirmed intern */
public void removeSlot(Student student) {
    if (!interns.remove(student)) {
        throw new IllegalArgumentException(s: "Student not in confirmed interns list");
    }

    confirmedSlots--;

    // Revert status if no longer filled
    if (status == Types.InternshipStatus.FILLED) {
        status = Types.InternshipStatus.APPROVED;
    }
}
```

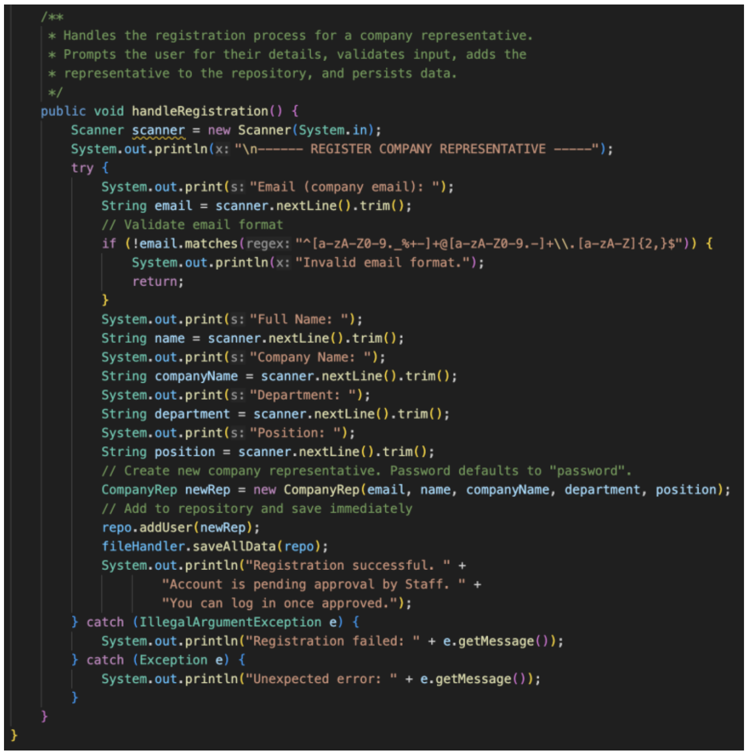
Figure 11: Slot Modification example

The example of slot modification shows how the attribute confirmedSlots is private and can only be modified through the methods addSlot() and removeSlot(). This ensures data integrity and security to keep any private data hidden. Non-authorized users cannot add or remove the students' internship slot placement.

3. SOLID Design Principles

3.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should only have one reason to change, meaning it should have one key responsibility. Some examples would be: CompanyRepRegistrationUI class only displays the UI for registration of new company representatives, FileHandler only deals with reading and writing CSV files (if the file format changes, only this class should be modified), DataRepo focuses on in-memory storage and retrieval, and StudentController encapsulates the rules specific to student actions.



```java
    /**
     * Handles the registration process for a company representative.
     * Prompts the user for their details, validates input, adds the
     * representative to the repository, and persists data.
     */
    public void handleRegistration() {
        Scanner scanner = new Scanner(System.in);
        System.out.println(x: "\n------ REGISTER COMPANY REPRESENTATIVE -----");
        try {
            System.out.print(s: "Email (company email): ");
            String email = scanner.nextLine().trim();
            // Validate email format
            if (!email.matches(regex: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$")) {
                System.out.println(x: "Invalid email format.");
                return;
            }
            System.out.print(s: "Full Name: ");
            String name = scanner.nextLine().trim();
            System.out.print(s: "Company Name: ");
            String companyName = scanner.nextLine().trim();
            System.out.print(s: "Department: ");
            String department = scanner.nextLine().trim();
            System.out.print(s: "Position: ");
            String position = scanner.nextLine().trim();
            // Create new company representative. Password defaults to "password".
            CompanyRep newRep = new CompanyRep(email, name, companyName, department, position);
            // Add to repository and save immediately
            repo.addUser(newRep);
            fileHandler.saveAllData(repo);
            System.out.println("Registration successful. " +
                    "Account is pending approval by Staff. " +
                    "You can log in once approved.");
        } catch (IllegalArgumentException e) {
            System.out.println("Registration failed: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
        }
    }
}
```

Figure 12: SRP: CompanyRep Registration

3.2 Open-Closed Principle (OCP)

The Open-Closed Principle states that a module should be open for extension but closed for modification. This means that we should be able to allow the addition of new functionalities without changing the existing code. We apply this principle by having the Student, CompanyRep and CareerStaff classes extend the abstract class User as seen in Figure 9 with interfaces and

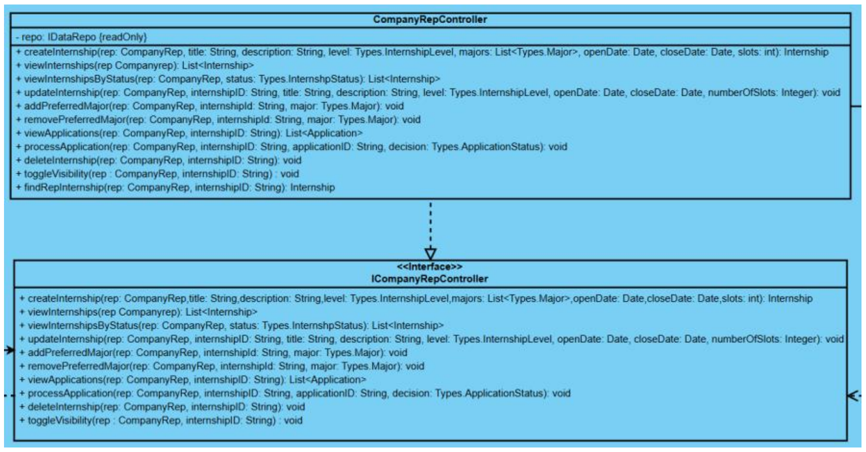
Enums. Zooming in on CompanyRep, it extends the functionality of User by implementing more specific methods related to the internship application process such as canCreateMoreInternship().

3.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states that a base class can be substituted by its subtypes without causing errors in the base class. This is done through inheritance and interface implementation. In our project, anywhere a User is expected, we can safely pass a Student, CompanyRep, or CareerStaff. Controllers and repository methods operate on User references. We also implement IStudent, ICompanyRep and ICareerStaff respectively. When IUser or User is expected, the subtypes can be used without causing errors.

3.4 Interface Segregation Principle (ISP)

The Interface Segregation Principle states that larger interfaces should be split into smaller ones such that classes should only depend on interfaces they use. In our program we achieve this by implementing specialized interfaces for the User subclasses in IStudent, ICompanyRep and ICareerStaff. There are also specialized interfaces for our controller classes as well, ILoginController, IStudentController, ICompanyRepController and ICareerStaffController. This ensures that changes to one interface do not impact unrelated classes and encourages code reusability across different contexts without unnecessary dependencies.



```
CompanyRepController
- repo: IDataRepo {readOnly}
+ createInternship(rep: CompanyRep, title: String, description: String, level: Types.InternshipLevel, majors: List<Types.Major>, openDate: Date, closeDate: Date, slots: int): Internship
+ viewInternships(rep Companyrep): List<Internship>
+ viewInternshipsByStatus(rep: CompanyRep, status: Types.InternshpStatus): List<Internship>
+ updateInternship(rep: CompanyRep, internshipID: String, title: String, description: String, level: Types.InternshipLevel, openDate: Date, closeDate: Date, numberOfSlots: Integer): void
+ addPreferredMajor(rep: CompanyRep, internshipId: String, major: Types.Major): void
+ removePreferredMajor(rep: CompanyRep, internshipId: String, major: Types.Major): void
+ viewApplications(rep: CompanyRep, internshipID: String): List<Application>
+ processApplication(rep: CompanyRep, internshipID: String, applicationID: String, decision: Types.ApplicationStatus): void
+ deleteInternship(rep: CompanyRep, internshipID: String): void
+ toggleVisibility(rep : CompanyRep, internshipID: String) : void
+ findRepInternship(rep: CompanyRep, internshipID: String): Internship

<<Interface>>
ICompanyRepController
+ createInternship(rep: CompanyRep,title: String,description: String,level: Types.InternshipLevel,majors: List<Types.Major>,openDate: Date,closeDate: Date,slots: int): Internship
+ viewInternships(rep Companyrep): List<Internship>
+ viewInternshipsByStatus(rep: CompanyRep, status: Types.InternshpStatus): List<Internship>
+ updateInternship(rep: CompanyRep, internshipID: String, title: String, description: String, level: Types.InternshipLevel, openDate: Date, closeDate: Date, numberOfSlots: Integer): void
+ addPreferredMajor(rep: CompanyRep, internshipId: String, major: Types.Major): void
+ removePreferredMajor(rep: CompanyRep, internshipId: String, major: Types.Major): void
+ viewApplications(rep: CompanyRep, internshipID: String): List<Application>
+ processApplication(rep: CompanyRep, internshipID: String, applicationID: String, decision: Types.ApplicationStatus): void
+ deleteInternship(rep: CompanyRep, internshipID: String): void
+ toggleVisibility(rep : CompanyRep, internshipID: String) : void
```

Figure 13: Interface Abstraction of CompanyRepController

3.5 Dependency Injection Principle (DIP)

The Dependency Injection Principle states that high level modules should not depend on low level modules, instead both should depend on abstractions. In our project, high level modules such as LoginController, StudentController, CompanyRepController and CareerStaffController depend on IDataRepo and IFileHandler instead of concrete classes. In MainApplication, it also depends on the controllers' interfaces versions, which shows that the program mostly remains independent of concrete implementations and makes it easier to replace.

4. UML Class Diagram

https://github.com/Shoterjust/NTU-SC2002-Group-Project-Final/blob/main/diagrams/Class%20Diagram%20Annotated.png

5. UML Sequence Diagram

https://github.com/Shoterjust/NTU-SC2002-Group-Project-Final/blob/main/diagrams/SequenceDiagram_CompanyRep.png

6. Test Cases

https://docs.google.com/spreadsheets/d/1aNPFb4aVypfzWYU0yXTe99nTEp7lS4Dpx4lxCvVLUgI/edit?usp=sharing

7. GitHub Repository Link

https://github.com/Shoterjust/NTU-SC2002-Group-Project-Final.git

8. Reflection

Difficulties encountered and how we conquered:

Misunderstanding the BCE flow:

- Put most (if not all) logic and flow directly inside entity classes (e.g. Student entity class would interact with Internship, Application, set new status, etc.) with almost no interfaces. After revising, we realized that the controller classes should manage most of the control logic, while entities should remain inside their domains only.
- We then added controller interfaces (IStudentController, ICompanyRepController, ICareerStaffController) to describe what operations are available, made the design extendable, and created looser coupling between layers of BCE. For example, boundary classes depend only on interfaces, not concrete controllers, which means each layer can change internally without forcing changes in the others.
- Entities now focus mainly on representing states and providing small helpers methods, while controllers coordinate interactions.

Selecting entities and assigning methods

- It was a challenge to keep relationships consistent. For example, the Student-Internship relationship: where to place eligibility checks, how to track applications, how to update both sides when an application is accepted or withdrawn.

Data persistence

- During one testing, FileHandler did not align with a slight change in CSV schema, so the system loaded incomplete data but still saved on exit (overwrote CSV files), resulting in original inputs being wiped out.
- We then modified to validate data loading success, print error messages when a line cannot be parsed, and only save back to CSV when the load is successful.
- Initially, we did not have DataRepo. Instead, the global HashMaps for users and internships were attributes of the CareerStaff class, with public static methods to add or

retrieve data. This design tightly coupled system-wide storage to a specific entity type (career staff), mixed persistence concerns into an entity, and relied on public static state.

Knowledge learnt

Design a system from scratch in layers:

- Think in terms of data flow before writing code
- Use BCE model to keep logic, data, and presentation separated.
- Application of SOLID principles and Interfaces:
  - Coding against interfaces makes the system more flexible and easier to change
  - SRP helped to decide which class owns which responsibility instead of putting everything into one.
- Learn to translate textual requirements into diagrams and code implementations.

Further Improvements suggestion

Extend registration: A more realistic version would support registration and approval for students and staff as well, with proper validation.

Strengthen authentication and security: Replace plain naked passwords in CSV files with hashed passwords or another encryption.