SC2002 – OBJECT ORIENTED DESIGN AND PROGRAMMING

**Project Title: Final Year Project Management System (FYPMS)**

**Declaration of Original Work for CE/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 (CE2002 or CZ2002) | Lab Group | Signature /Date |
|---|---|---|---|
| Toh Jing Qiang | SC2002 | A33 | Toh Jing Qiang / 15-04-23 |
| Toh Jing Hua | SC2002 | A33 | Toh Jing Hua / 15-04-23 |
| Ong Xin Ning, Trini | SC2002 | A33 | Ong Xin Ning, Trini / 15-04-23 |

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card.

## 0. Executive Summary

This report focuses on explaining the design considerations and OOP concepts used in the development of the FYPMS, a Java CLI application for managing Final Year Projects in tertiary institutions. The report includes a detailed UML Class Diagram, SOLID design principles, the use of OOP concepts, and other design considerations like Repository Pattern. The report also includes a section on testing, with test cases and results, as well as a reflection section that details the difficulties encountered, knowledge gained, and further improvements that can be made. Overall, this report provides a comprehensive analysis of the design principles and OOP concepts used in the development of the FYPMS.

# 1. Design Considerations

## 1.1 Overview of Our Approach

The Final Year Project Management System (FYPMS) was designed with a **modular approach** while adhering to **Object-Oriented Concepts** and **SOLID design principles**. This ensures that the FYPMS application is **extensible**, **maintainable**, and **easy to test**.

To achieve this, we first separated the FYPMS application into **different packages** with distinct responsibilities. Within each package, each class is designed, while adhering to **Object-Oriented Concepts**, to tackle **one specific task** related to the package's responsibility. As a whole and between each class, **SOLID design principles** were taken into account when constructing their relationships with each other. Hence, with this approach, the FYPMS was designed to follow a **Model-View-Controller (MVC)** architecture, with additional packages like **Services**, **Stores**, **Interfaces**, and **Enums**. This ensures that there is **separation of concerns** between different classes of FYPMS, as well as make it easy for new developers to understand and maintain.

Additionally, we also employed other design patterns and considerations like a **Simplified Repository Pattern** to address specific design challenges when developing the FYPMS.

## 1.2 Assumptions Made

Firstly, we assumed that the data stored in the data files are well-formed and will not require extensive data checking when importing the data during the initialisation of FYPMS. This assumption is only made when the data files are used for the first time during the initialisation of FYPMS since subsequent updates to the same data files will be well-formed as intended.

Secondly, we assumed that the users know what FYPMS is and understand what they can do in the FYPMS according to their roles – Student, Supervisor, FYP Coordinator. Thus, the CLI descriptions and instructions that explain what each feature does were less descriptive and less prevalent.

Thirdly, we assume that the users will be running the FYPMS on a system with the required resources to run the FYPMS.

## 1.3 Use of Object-Oriented Concepts

### 1.3.1 Abstraction

Abstraction was used to simplify the complexity of the FYPMS by hiding unnecessary details and exposing only the essential features. Firstly, **Interfaces** were used to abstract away the implementation details of the Services and Views. Secondly, an **Abstract Model Class** (Request Class) was used to abstract away the implementation details of the concrete models (Sub-classes of Requests Class). Hence, interactions with Services, Views, and Sub-classes of Request Class are through Interfaces or the Abstract Model Class (Request Class). *Fig. 1.3.1a* shows an example.

```
11 public class AuthController {
12     private static final Scanner sc = new Scanner(System.in);
13     private static IAuthService authService;
```

Fig 1.3.1a. AuthController uses IAuthService, which is an interface for the different IAuthService implementations.

### 1.3.2 Encapsulation

Encapsulation was used to protect the internal state of the objects and to prevent unwanted access of data by other classes. Thus, **attributes** of classes were made **private** and can only be accessed and modified via **public getters** and **setters methods**.

### 1.3.3 Inheritance

Inheritance was used to create **hierarchical relationships** between classes, with child classes inheriting behaviors and properties from parent classes. For example, the Student and Supervisor Classes inherit from the User Class, and the FYPCoordinator Class inherits from the Supervisor Class. Inheritance was also used for the Controller Classes, where StudentController and SupervisorController classes inherits from UserController, and the FYPCoordinatorController Class inherits from the SupervisorController. Inheritance was also used in the services and interfaces for Project and Request Services related to the FYPCoordinator Class. This Inheritance concept is illustrated in the UML Class Diagram in *Section 2*.

### 1.3.4 Polymorphism

Polymorphism was used to allow objects to take on multiple forms and behaviors. For example, the Request Abstract Model Class can have different behaviors depending on its Subclass due to method overriding of the `` `approve()` `` and `` `reject()` `` methods. Additionally, Interfaces were used to allow the Controllers Class to interact with different types of Services and Views in a polymorphic way.

## 1.4 Design Principles

### 1.4.1 Single Responsibility Principle (SRP)

The FYPMS is split into several packages with each package performing a single specific responsibility. They include Stores, Models, Views, Controllers, Services, Interfaces, and Enums. Within each package, each class is designed to tackle one specific task related to the package's responsibility.

(a) Stores

Classes that manage the global state of the application with direct interaction with the database. For example, `AuthStore` that manages the global authentication state, and `DataStore` that manages the global object data state via direct interaction with the database.

(b) Models

Entity Classes and their relationships with each other. Some examples include `User`, `Student`, `Supervisor`, `FYPCoordinator`,`Project`, `Request`.

(c) Views

Classes that display information to the user on the CLI. The concrete classes implemented each display a specific information about a particular object. For example, `ProjectAvailableView` class displays information about a project object that is available, while the `ProjectAllocatedView` class displays information about a project object that is allocated. This approach is similar for Request related View classes. For example, `RequestAllocateProjectView`.

(d) Controllers

Classes that invoke Services and Views (via Interfaces) based on user inputs to control the user experience and flow. A different controller class is used for different control tasks. For example, `AuthController` controls user authentication experience, while `StudentController`, `SupervisorController`, and `FYPCoordinatorController` controls the FYPMS experience for students, supervisors, and fyp coordinator respectively.

(e) Services

Classes that perform specific business logic functionality. For example, `ProjectStudentService` that contains business logic about students' interactions with projects, `CsvDataService` that contains business logic about how data from CSV files are read into FYPMS and updated, and etc.

(f) Interfaces

Classes that define a set of related methods that must be implemented by a concrete class. Some examples include `IFileDataService`, `IProjectStudentService`, and `IRequestView`.

(g) Enums

Classes that define a set of constant values that are used throughout the application. For example `UserRole`, `ProjectStatus`, `RequestStatus`, and `RequestType`.

### 1.4.2 Open-Closed Principle (OCP)

To ensure the extensibility of FYPMS, OCP was followed by utilizing interfaces and abstract classes to promote loose coupling between classes and ensure that each class can be swapped out easily or extended without affecting the rest of the application too much.

(a) Interfaces

Each Services and Views implement a specific Interface. For example, `CsvDataService` implements the `IFileDataService` interface. When adding a new file data service, a new class, for instance `TxtDataService` which implements the same interface can be created without much effect on the rest of the application. Hence, the `DataStore` class that uses this service can easily swap out `CsvDataService` with `TxtDataService` since both implement the same interface.

This applies to the other interfaces like `IProjectView`, `IRequestView`, `IProjectStudentService`, `IRequestSupervisorService`, and etc.

(b) Abstract Classes

`Request` class is an abstract class that is currently extended by several request subclasses like `AllocateProjectRequest`. FYPMS can be extended to include more request types by creating a new subclass that extends the `Request` class. For example, `IncreaseMaxProjectCapRequest` to increase the maximum number of projects a supervisor can supervise.

`AuthService` is also an abstract class that is currently extended by several subclasses like `AuthStudentService` to provide specific authentication service to students. FYPMS can be easily extended to authentication service to a user with a new role. For example, `AuthAdminService`.

### 1.4.3 Liskov Substitution Principle (LSP)

To ensure that FYPMS follows LSP, we designed derived classes to have pre-conditions no stronger than the base class methods, and post-conditions no weaker than the base class method. Thus, derived classes are designed to never introduce new requirements that are not present in the base class, as well as to never reduce the guarantees provided by the base class. This is achieved via polymorphism and is applicable to all generalization and realization relationships (E.g.: `Request` base class and `AllocateProjectRequest` derived class).

### 1.4.4 Interface Segregation Principle (ISP)

The Interfaces in the FYPMS application are specific in the sense that classes that implement the Interfaces use all the methods defined in the Interfaces.

Thus, for Project Service related Interfaces, 3 different interfaces are created for the 3 different User Roles – Student, Supervisor, and FYPCoordinator. Each interface only defines methods representing business logic for projects related to the specific role, thus ensuring that each role can only perform operations that they are allowed to. The 3 interfaces are `IProjectStudentService`, `IProjectSupervisorService`, and `IProjectFYPCoordinatorService`. Similarly for Request Service related Interfaces, the 3 different interfaces are `IReqeustStudentService`, `IReqeustSupervisorService`, and `IReqeustFYPCoorinatorService`.

### 1.4.5 Dependency Inversion Principle (DIP)

FYPMS was designed to follow the DIP as high-level classes depend on abstractions rather than concrete implementations. This is achieved via utilizing interfaces and abstract classes.

#### (a) Services

All Services implement an Interface, which Controller classes and Store classes interact with via their Interfaces. For example, `StudentController` class depends on `IProjectStudentService` interface instead of `ProjectStudentService` concrete implementation.

#### (b) Views

All Views implement an Interface, which Controller classes interact with via their Interfaces. For example, `StudentController` class depends on the `IProjectView` interface instead of the `ProjectAllocatedView` concrete implementation.

(c) Abstract Classes

The other parts of the FYPMS depend on the `Request` abstract class instead of its derived class like `AllocateProjectRequest`, `DeregisterProjectRequest`, and etc.

## 1.5 Other Design Considerations

### 1.5.1 Foreign Key Association

To ensure data integrity in the whole application and database, foreign keys are used to link related objects together instead of using object composition/aggregation. This ensures that when details about an object changes, the other objects that contain the updated object will still have the updated information about the object. This is because there is only one instance of the updated object in the whole FYPMS application. Thus, ensuring data integrity. For example, `Project` class is related to the `Student` and `Supervisor` classes. To ensure data integrity, the private attributes of `Project` class related to the student and supervisor classes are `studentID` and `supervisorID` strings instead of `Student` and `Supervisor` objects *(Fig. 1.5.1a)*. To retrieve the student and supervisor objects, `public getter methods` are used to directly access the DataStore and return the required objects *(Fig. 1.5.1a)*.

```
public class Project {
    private static int lastProjectID = 0;
    private int projectID;
    private String title;
    private String studentID;
    private String supervisorID;
    private ProjectStatus status;
```

```
public Supervisor getSupervisor() {
public boolean setSupervisor(String supervisorID) {
public Student getStudent() {
public boolean setStudent(String studentID) {
```

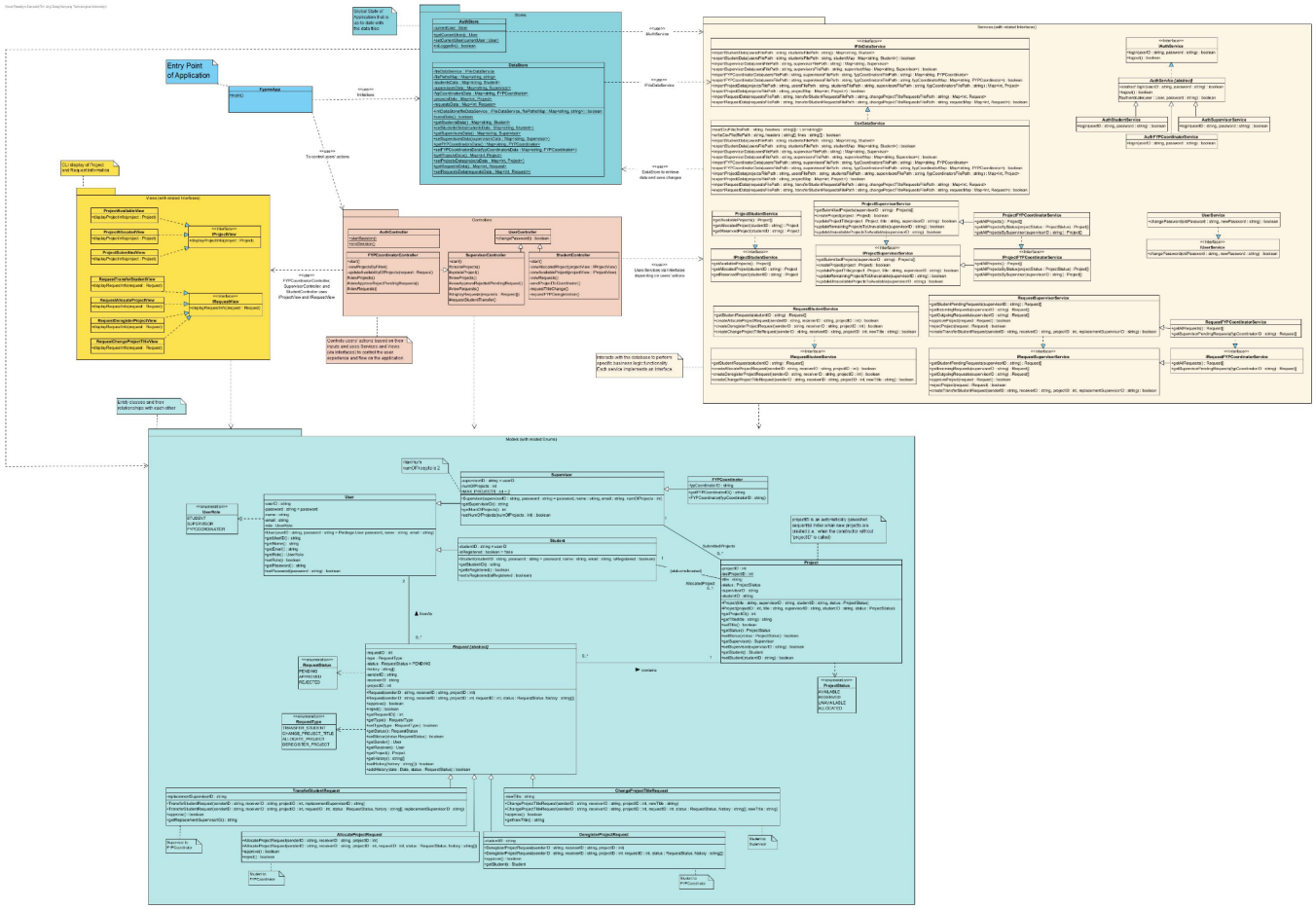
*Fig. 1.5.1a: Project's studentID and supervisorID string attributes (left) and getter methods (right)*

### 1.5.2 Simplified Repository Pattern

A simplified Repository Pattern is used to design the FYPMS. This pattern is implemented via the `DataStore` class. The DataStore class acted as a repository for the data stored in the data files, and the IFileDataService interface defined the methods used to access this data. This pattern allows the data layer to be isolated from the rest of the application and the details of the data storage implementation to be abstracted away. Thus, making it easy to switch to a different data storage mechanism in the future without affecting the rest of the application too much.

## 2. UML Class Diagram

The following image depicts a UML Class Diagram with UML package notation to show the class relationships and dependencies between different classes and packages. The notes in the diagram provide additional explanation about some of the design choices and patterns for the FYPMS. A clearer diagram image is available in a separate file named `uml-class-diagram.jpg`.



## 3. Functional Tests and Results

Note: Additional End-to-End Tests can be found in the attached pdf named `e2e-tests.pdf`.

### 3.1 Authentication

| Test Case | Test Description | Expected Result | Pass /Fail |
|---|---|---|---|
| 1 | **[Cannot Login]**<br>(a) Entering wrong userID for the selected role. | For both (a) and (b), the user fails to login and is prompted to re-enter their credentials. | Pass |

| | | | |
|---|---|---|---|
| | (b) Entering correct userID but wrong password for the selected role. | | |
| 2 | **[Successful Login]**<br>(a) User enters the correct userID and password for the selected role. | User logins successfully, and a different menu list is displayed for different user roles – student, supervisor, fyp coordinator. | Pass |
| 3 | **[Change Password]**<br>(a) After login, the user changes password by entering his old password.<br>(b) After login, the user changes password by entering a new password. | (a) User fails to change the password, and an error message is shown.<br>(b) Password is changed and the user is signed out of the application. Re-login with a new password is successful. Relevant data files are updated. | Pass |

## 3.2 Student Functionality

| Test Case | Test Description | Expected Result | Pass /Fail |
|---|---|---|---|
| 1 | **[Change Password]**<br>Same as section 3.1.1 test case 3 | Same as section 3.1.1 test case 3 | Pass |
| 2 | **[View Available Projects]** | A list of projects that are available is shown. | Pass |
| 3 | **[Select the project to send to the coordinator]**<br>Select a project and send a request to the coordinator to allocate the project to the student. | Project status will change to "RESERVED", a request to allocate the project is sent to the coordinator, and relevant data files are updated. | Pass |
| 4 | **[View his/her own project]**<br>(a) Viewing his/her own project before the coordinator approves the project allocation.<br>(b) Viewing his/her own project after the coordinator approves the project. | (a) Error message "You have not registered a project" is shown.<br>(b) Student's own project is displayed and the project status is "ALLOCATED" and the project has the student's information displayed. | Pass |
| 5 | **[View requests status and history]** | Shows all the requests with the request information, status and history displayed. | Pass |
| 6 | **[Request to change title]**<br>Input the new title and the supervisor should get the request. | A request to change project title is created and the receiver is the supervisor that submitted the project. Relevant data files are updated. | Pass |
| 7 | **[Request coordinator to deregister FYP]** | A request to deregister FYP is sent to the coordinator. Relevant data files are updated. | Pass |

## 3.3 Supervisor Functionality

| Test Case | Test Description | Expected Result | Pass/ Fail |
|---|---|---|---|
| 1 | **[Change Password]**<br>Same as section 3.1.1 test case 3 | Same as section 3.1.1 test case 3 | Pass |
| 2 | **[Create/Update/View Projects]**<br>(a) Create 2 new projects by inputting the required data.<br>(b) Update 1 project by choosing a project by projectID, then input a new title.<br>(c) View submitted projects. | (a) Project is created, supervisor's number of supervising projects increase by 1 and relevant data files are updated.<br>(b) Project title is updated.<br>(c) Necessary data about the projects submitted by the supervisor is printed. | Pass |
| 3 | **[View Student Pending Requests]** | Indicate 'NEW' if there is pending request and supervisor can choose student request by RequestID | Pass |
| 4 | **[Approve/Reject Requests]**<br>(a) Approve "Change Project Title" requests to update the project title.<br>(b) Reject "Change Project Title" requests. | (a) Project Title for the project associated with the request is changed. Request status changes to "APPROVED". Relevant data files are updated.<br>(b) Request status changes to "REJECTED". | Pass |
| 5 | **[View Request History]** | All requests with detailed history are displayed. | Pass |
| 6 | **[Request to Transfer Student]**<br>Create a request to the transfer student by inputting necessary input. | Request created, the FYP Coordinator receives the request, and relevant data files are updated. | Pass |

## 3.4 FYP Coordinator Functionality

| Test Case | Test Description | Expected Result | Pass/ Fail |
|---|---|---|---|
| 1 | **[Can perform all functions that supervisor can perform]**<br>Same as all the test cases in section 3.1.3. | Same as all the test cases in section 3.1.3. | Pass |
| 2 | **[View Requests]** | A list of all requests are shown. FYP coordinator can then select a request for approval or rejection. | Pass |
| 3 | **[Choose Request from** | If the replacement supervisor has already hit the cap, | Pass |

| | | | |
|---|---|---|---|
| | **Supervisors to Approve/Reject]** | the hint message should be displayed and up to the coordinator's decision to approve or reject the request. | |
| 4 | **[Allocate a Project to Student via Student Request]**<br>(a) Approve request<br>(b) Reject request | (a) Project is updated with Student's information, project's status is changed to "ALLOCATED", and the request's status is changed to "APPROVED".<br>(b) Request's status is changed to "REJECTED", and the project's status is changed to "AVAILABLE". | Pass |
| 5 | **[Deallocate a Project from a Student via Student Request]**<br>(a) Approve request<br>(b) Reject request | (a) Student's information is removed from the Project, Project status is changed to "AVAILABLE", and the request status is changed to "APPROVED".<br>(b) Request status is changed to "REJECTED". | Pass |
| 6 | **[View projects according to different filters]**<br>(a) Filter by SupervisorID<br>(b) Filter by Project Status | (a) A list of projects with selected supervisorID is displayed.<br>(b) A list of projects with selected Project Status is displayed. | Pass |

### 3.5 Error Handling

| Test Case | Test Description | Expected Result | Pass/ Fail |
|---|---|---|---|
| 1 | [Invalid Inputs]<br>(a) Entering an invalid input when selecting userID, requestID, and projectID | A message that prompts the user to input a valid input is displayed. | Pass |

## 4. Reflection

### 4.1 Difficulties Encountered

Firstly, we face difficulties in ensuring that the FYPMS follows the Single Responsibility Principle (SRP) and has Separation of Concerns. We overcame this challenge by implementing an MVC architecture with additional packages like services, and stores, which enabled us to separate the application's concerns and made collaboration and changes to the application easier.

Secondly, we face difficulties in trying to abstract away the data layer and maintain a global state for the application. We overcame this challenge by using the DataStore with access to the data files via the `IFileDataService` interface, which enabled us to store the global state of the application and interact with the database directly.

Thirdly, we face difficulties in trying to maintain data integrity in the FYPMS and the database. We overcame this challenge by using foreign key associations to store the relationships between entity classes. This ensures that changes to one entity class would be consistent throughout the application and database since there is only one instance of the updated entity class.

Lastly, we faced difficulties in trying to implement the requests functionality with different approve and reject operations while adhering to SOLID principles. We overcame this challenge by creating an abstract Request class with `approve()` and `reject()` methods that will be overridden in the derived class to operations specific to the derived class via dynamic binding.

## 4.2 Knowledge Learnt

Throughout the development of the application, we gained a deeper understanding of OOP and SOLID principles and best practices in software design and implementation. We learnt that using packages to separate concerns and for easier collaboration and updates is important. Thus, we learnt the importance of having a well structured architecture (MVC architecture) for both the code implementation and UML class diagrams. Additionally, we learnt that using SOLID principles enabled us to extend and maintain our application easily. We also learnt the importance of naming conventions, like using prefixing Interfaces with an "I" (e.g.: IProjectView), and using Domain-driven naming for classes (e.g.; ProjectStudentService and ProjectSupervisorService) to increase the ease of collaboration. Furthermore, we learnt the need to abstract the Data Layer so that the rest of the application can access and update the data files easily anywhere in the application. This enhances maintainability of the application

## 4.3 Further Improvements

Firstly, the Data Layer which includes a DataStore for all entity classes, can follow the SRP more by utilizing the Repository Pattern – A Repository for each entity class.

Secondly, security in the personal information of users can be improved. In particular, passwords should be hashed before storing into the data files.

Thirdly, the CLI interface can be more user-friendly and intuitive. For example, CLI features like autocomplete, as well as more detailed descriptions to explain the FYPMS can be added.

Lastly, additional features could be added to the FYPMS, for example, sign up feature for new users.