

**SC2002 – Object Oriented Design & Programming**

Group Report

**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 | Lab Group | Signature /Date |
| NG WEE KIAT | SC2002 | A27 | 16 April 2023 |
| NG YUEN HERNG | SC2002 | A27 | 16 April 2023 |
| NG ZI XUAN | SC2002 | A27 | 16 April 2023 |
| PEARLINA TAN QINLIN | SC2002 | A27 | 16 April 2023 |
| TAN WEI YIN | SC2002 | A27 | 16 April 2023 |

# Design Considerations

**1.1 Software design**

Our software design is predominantly based on the Model-View-Controller (MVC) concept. The Model component manages the relevant data required to run the program. Data is stored from an Excel database *FYP\_database.xlsx*. The View component handles the user interface, which is shown to the user. Finally, the Controller component is the intermediate component between the Model and View component, taking user input and data manipulation.

OOP concepts of Abstraction, Inheritance, Encapsulation, Polymorphism and Composition are integrated into our design. Additionally, we apply the SOLID object-oriented principles to our program to be easily maintainable and modifiable. These concepts are applied to minimise impacts of change via code organisation, ultimately improving the modularity and reusability of our program.

**1.2 Data management**

A LinkedHashmap is used for data management in our project due to the following benefits:

|  |  |
| --- | --- |
| Maintains Order of insertion | Order of when user data was added to our LinkedHashMap is maintained. This enables chronological sorting of requests between user types of our FYP management system. |
| Unique Keys | Keys act as unique identifiers to enable linking for objects |
| Iteration | Iteration in a LinkedHashmap enables us to display object details such as in viewing all available projects |
| Easy Object Manipulation | Entries in a LinkedHashmap can be removed by their key  This enables quick access in looking up user information |

On program initialization, data is read from the given Excel workbooks with the ***Apache POI*** library. This data is then written into LinkedHashmaps in Java for subsequent manipulation by the Model-View-Controller based system. Upon program termination, data is written into an Excel Database *FYP\_database.xlsx* according to its entity. Each entity is written into a specific worksheet, with worksheets *student, FYPSupervisor, FYPCoordinator, Project, StuToFYPReq, StuToSupReq, SupToFYPReq*. Subsequently, when a new user initializes the system, the program checks if a *FYP\_database.xlsx* Excel workbook exists, and initializes the data.

**1.3 Applications to OOP principles**

**Single Responsibility Principle (SRP)**

SRP is applied via separation of contents principle. The program is divided into specific classes and components that are responsible for its specific tasks. Detailed below are a few examples of encapsulation and inheritance used in our project.

|  |  |
| --- | --- |
| **Class Name** | **Class Description** |
| *RequestStatus.java*  *RequestType.java*  *ProjectStatus.java*  *Request.java (Superclass)* | Each class is responsible for managing either the status or types of requests/projects. I.e., there is no one single “Status” class to store all types of requests/projects statuses. |
| *Student.java*  *Supervisor.java*  *User.java (Superclass)*  *FYP.java* | These classes are responsible for managing student, supervisor, FYP coordinator and user information, respectively. Each class has a single responsibility to manage information specific to its corresponding entity. |

This extends to our Excel database, with each entity’s data separated into individual worksheets for compartmentalised access. This ensures each class has a clear and specific responsibility, simplifying code and making it easier to maintain and modify over time. Assigning a unique responsibility to each class ensures that there is only at most one reason for modification each time.

**Open-Closed Principle (OCP)**

OCP is used in extending the behaviour of our classes without modifying its source code. Defining interfaces such as *Request.java,* with accessor and mutator methods for request status, adds functionality without modifying additional classes. In the table below, we introduce the Request interface and create independent implementations of different user-to-user requests. Therefore, we do not need to modify *Request.java* whenever we add unique features.

|  |  |
| --- | --- |
| Interface | Implementations |
| Request.java | StuToSupReq.java  StuToFYPReq.java  SupToFYPReq.java |

Dependency injection was also incorporated to decouple classes for easy testing and maintenance. Specifically, setter injection was used in the ExcelData class whereby the Student class was passed to the ExcelData through a mutator method rather than the ExcelData creating the student itself.

**Liskov Substitution Principle (LSP)**

LSP was applied using Inheritance and Polymorphism for objects of the superclass to be replaced by objects of the subclass. Illustrated below are the Superclasses with their corresponding subclasses:

|  |  |
| --- | --- |
| Superclass | Subclass |
| User.java | Student.java  FYP.java  Supervisor.Java |
| Request.java | StuToSupReq.java  StuToFYPReq.java  SupToFYPReq.java |

Since the Student class is a subclass of the User class then any instance of the Student class can be used in place of the User class without errors. In other words, they are substitutable for their base types.

**Interface Segregation Principle (ISP)**

ISP is applied in our project by ensuring users are only required to implement necessary methods in our Interface classes. This was integrated into designing our StudentView and StudentController class. The StudentView class provides a set of methods for displaying options and handling user input, while the student controller class provides methods such as changing the password, selecting projects and viewing request history. Hence, the StudentView class adheres to ISP by depending only on methods it needs and avoiding dependencies on methods it does not use.

**Dependency Inversion Principle (DIP)**

DIP was used in our Project, ProjectController and ProjectStatus classes. The Project Controller class depends on the ExcelData class to get the project database. However, the ProjectController class does not depend on the ExcelData class *directly* but instead depends on an abstraction of the ExcelData class. This was similarly reflected in our Project class depending on abstractions of the Supervisor and Student class instead of concrete implementations of these classes directly.

**1.4 Design Extensibility and Maintainability**

Using the MVC pattern separates concerns and makes it easy to modify and extend each component without affecting the others. This facilitates maintainability by providing a clear structure for the codebase. Easy data retrieval with LinkedHashmaps and Apache POI library makes data retrieval and manipulation easy, facilitating extensibility and maintainability. Integrating the SOLID principles alongside Abstraction and Encapsulation ensures system’s components are loosely coupled allowing them to be modified or extended without causing ripple effects on other components in the system. Inheritance and polymorphism facilitate code reuse, likewise making it easier to extend the system’s functionality.

# Detailed UML Class Diagram

(Kindly refer to UML Diagram attached along as image file in zip folder if unclear.)

# Testing

|  |
| --- |
| **Student #1** |
| **STUDENT** - View registered project fails. Error message: *You have not registered a project.* |
| **STUDENT** -View all available projects |
| **STUDENT** - Select a project by specifying the projectID. |
| **FYPCOORDINATOR** – Approve Request |
| **STUDENT** - View all available projects fails. Error message: *You are currently allocated to a FYP and do not have access to available project list.* |
| **STUDENT** - View registered project. |
| **STUDENT** - Request to change title by providing a new title. |
| **FYPSUPERVISOR** – Approve Change of title request |
| **STUDENT** – View registered project to verify the title change |
| **STUDENT** – Request to deregister the project |
| **FYPCOORDINATOR** – Approve deregistering request |

|  |  |
| --- | --- |
| **STUDENT** – View registered project (Fail with error message) | **STUDENT** – View all available projects (Fail with error message) |

|  |
| --- |
| **Supervisor’s cap** |
| **Student #2** selects Bo An’s project (and FYP coordinator approves) |
| **Student #2** changes title (and Bo An approves) |
| **Student #3** selects Bo An’s project (and FYP coordinator approves) |
| **Student #4** view all available projects, Bo An’s remaining projects are NOT included in the list. |
| **FYPSupervisor(Bo An)** transfers a project to FYPSupervisor(Dusit Niyato) and FYP coordinator approves. |
| **Student #4** view all available projects, Bo An’s remaining projects are included in the list. |
| **Student #4** selects Dusit Niyato’s project (and FYP coordinator approves). |

|  |
| --- |
| **Dusit Niyato** submits a new project. |
| **Student #5** view all available projects, Dusit Niyato’s remaining projects are NOT included in the list. |
| **Student #4** deregisters FYP (and FYP approves). |
| **Student #5** view all available projects, Dusit Niyato’s remaining projects including the  deregistered project will be displayed in the available project list. |

|  |
| --- |
| **Student #5** selects the recycled project. |

# Reflection

Over the course of our project, most of the difficulties encountered revolved around the design of our software. Specifically, starting out with the UML diagram, we realised our classes were too interdependent, with the diagram becoming too well-connected. This resulted in relationships represented by lines overlapping with each other. To circumvent this issue, we had to create well-defined interfaces such that there would be loose coupling between the classes. This also implied that we had to apply OO principles such as inheritance to create subclasses from abstract classes and define them as required for their purposes. This ensured we do not have a one-size-fits-all super class which performs all functions.

Additionally, data management was another difficulty encountered, with the key challenge of writing and reading data in and out of our Excel database. To overcome this difficulty, we utilised OOP principles such as encapsulation for this process. Data was written into specific classes to help to maintain data compartmentalization.

One recommendation we would integrate to improve our current design is to assign the respective controllers, models, etc. into different modules for better reference and access. Another recommendation would be to create classes for the purpose of exception handling. This can assist us to catch more general exceptions and improves code readability.

Overall, we learnt about the multiple OOP design principles for creating a system, as well as techniques for design extensibility and maintainability, which are crucial for successful OOP systems.