1 Introduction

To make our project more maintainable and modular, we are refactoring it into the Model-View-Controller (MVC) design pattern. This document outlines the new structure, explains the responsibilities of each component, and provides an updated package organization and project priorities for demo day.

2 MVC Design Pattern Overview

The MVC pattern separates the application into three main components:

- **Model**: Manages the data, logic, and rules of the application.
- **View**: Displays data to the user and handles the user interface.
- **Controller**: Acts as an intermediary between Model and View, handling user input, updating the Model, and refreshing the View.

3 Project Structure

The updated package organization is as follows:

```
src/
|-- main/
|   |-- java/
|       |-- controller/
|       |   |-- CommandAction.java
|       |   |-- CommandFactory.java
|       |   |-- Functions.java
|       |   `-- ParsingInputs.java
|       |-- model/
|       |   |-- UMLClass.java
|       |   |-- UMLRelationship.java
|       |   |-- UMLRelationshipType.java
|       |   `-- Storage.java
|       `-- view/
|           `-- Display.java
```

4 Component Descriptions

4.1 Model

The **Model** is responsible for managing the data and business logic. It includes:

- `UMLClass.java`: Represents each UML class with its attributes and methods.
- `UMLRelationship.java`: Represents the relationships between UML classes.
- `UMLRelationshipType.java`: Defines the types of UML relationships (e.g., association, inheritance).
- `Storage.java`: Manages the storage and retrieval of UML entities, handling persistence of data.

4.2 View

The **View** is responsible for displaying information to the user. It includes:

- `Display.java`: Manages output formatting and displays information to the user. This class interacts with the Controller to receive updated data.

4.3 Controller

The **Controller** acts as an intermediary between Model and View, interpreting user input and modifying the Model. It includes:

- `CommandAction.java`: Processes actions based on commands received from the user and modifies the Model.
- `CommandFactory.java`: Creates command instances based on user inputs.
- `Functions.java`: Contains methods for handling command execution logic.
- `ParsingInputs.java`: Parses user inputs and routes commands to the appropriate methods.

5 Class Refactoring to Adhere to MVC Principles

Each class must be refactored to fully conform to MVC principles, separating responsibilities and improving maintainability. Below are specific guidelines:

5.1 Model Refactoring

Model classes (`UMLClass`, `UMLRelationship`, `UMLRelationshipType`, and `Storage`) should:

- Only contain logic for managing and accessing data.
- Avoid including any input/output code. For instance, `Storage` should handle data storage without printing or formatting data.
- Use methods to encapsulate data, enabling controlled access for the Controller and View.

5.2 View Refactoring

The `Display` class in the View should:

- Exclusively handle displaying information, without processing or altering data directly.
- Rely on formatted data provided by the Controller, which coordinates updates to the View.
- Contain methods for rendering error messages, success notifications, and data summaries as directed by the Controller.

5.3 Controller Refactoring

Controller classes (`CommandAction`, `CommandFactory`, `Functions`, and `ParsingInputs`) should:

- Interpret user input, update the Model, and coordinate updates to the View.
- Use the Model's methods for data manipulation rather than directly modifying data structures.
- Move all command processing logic to be encapsulated within the Controller classes.

5.4 Example Refactoring in `CommandAction`

Original code with tightly coupled Model and View logic:

```
public void executeCommand(String command) {
        if (command.equals("addClass")) {
                UMLClass newClass = new UMLClass("NewClass");
                Storage.addClass(newClass);
                System.out.println("Class added: " + newClass.getName());
        }
}
```

Refactored code following MVC principles:

```
// Controller Class - CommandAction.java
public void executeCommand(String command) {
        if (command.equals("addClass")) {
                UMLClass newClass = new UMLClass("NewClass");
                Storage.addClass(newClass);
                Display.showClassAdded(newClass.getName());
        }
}

// View Class - Display.java
public static void showClassAdded(String className) {
        System.out.println("Class added: " + className);
}
```

The refactored `executeCommand` method in `CommandAction` only coordinates actions, delegating output to `Display` and data storage to `Storage`.

6 Project Priorities for Demo Day

To prepare for demo day, the following tasks are prioritized:

[1] **Priority High -** Bugfix the logic from Sprint 1

[2] **Priority High -** Post all user scenarios on the GitHub board

[3] **Priority High -** Build GUI (highly recommended to peer-program for efficiency)

[4] **Priority Medium -** Log and post all bug fixes to GitHub board

[5] **Priority Low -** Fix JavaFX .jar launch issue

[6] Priority Low - Document the new build for clarity and future maintenance

[7] Priority Low - Begin planning for new features, such as auto-complete functionality

7 Conclusion

This refactored structure and priority list for demo day ensure that:

- The **Model** manages data effectively.
- The **View** displays information accurately.
- The **Controller** handles logic and user inputs.

By adhering to this MVC pattern and focusing on key priorities, each component will operate independently, simplifying testing, debugging, and future expansion. The priority list will help allocate resources effectively to ensure a successful demo.