Software Design Pattern End Term Report

Astana IT University

Instructor: Dr. Maksutov Akylbek

Ali Parwiz Baktash

Student Registration System

Project Overview

Project Idea: The Student Registration System is designed to streamline the registration process for students enrolling in different courses. To make the system modular, scalable, and maintainable, several design patterns have been integrated, with the Model-View-Controller (MVC) architecture pattern serving as the foundational structure.

Performance Goal: The goal of this project is to build a robust, flexible registration system that handles multiple students, supports database persistence with PostgreSQL, and dynamically extends functionality for premium students. By using MVC and various design patterns, the project achieves a high degree of modularity and scalability.

Performance Objectives

1. Structure the system using an architecture pattern that cleanly separates concerns (MVC).
2. Implement creational patterns to enhance the flexibility of object creation.
3. Implement structural patterns to enable dynamic functionality extensions.
4. Apply behavioral patterns to manage interactions and notifications.

Main Body

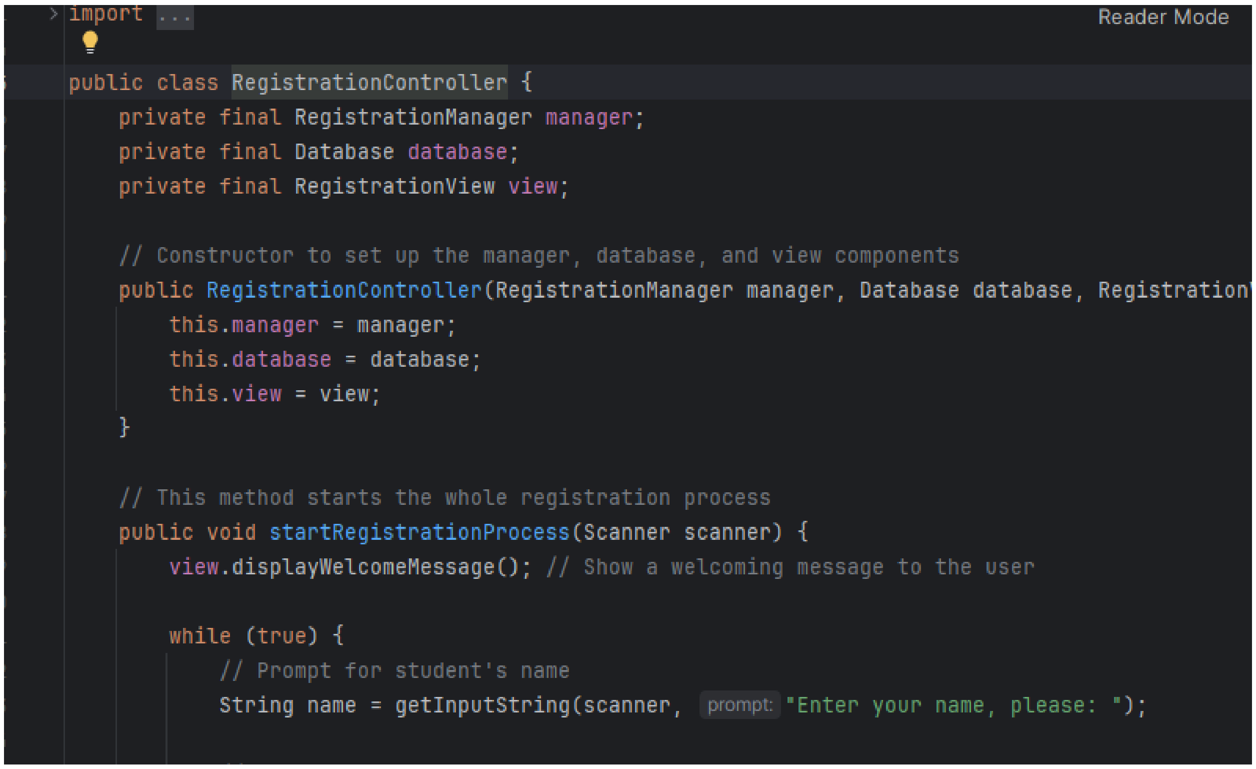
A UML diagram has been included to provide a visual overview of the system architecture, illustrating the relationships between the core components and design patterns (MVC, Singleton, Factory, Decorator, Observer, and Builder). This diagram helps convey the structure and interactions within the Student Registration System.s

1. Architecture Pattern: Model-View-Controller (MVC)

The Model-View-Controller (MVC) architecture pattern is the backbone of the Student Registration System. It organizes the application into three primary components, each with a specific responsibility, ensuring a clear separation of concerns:

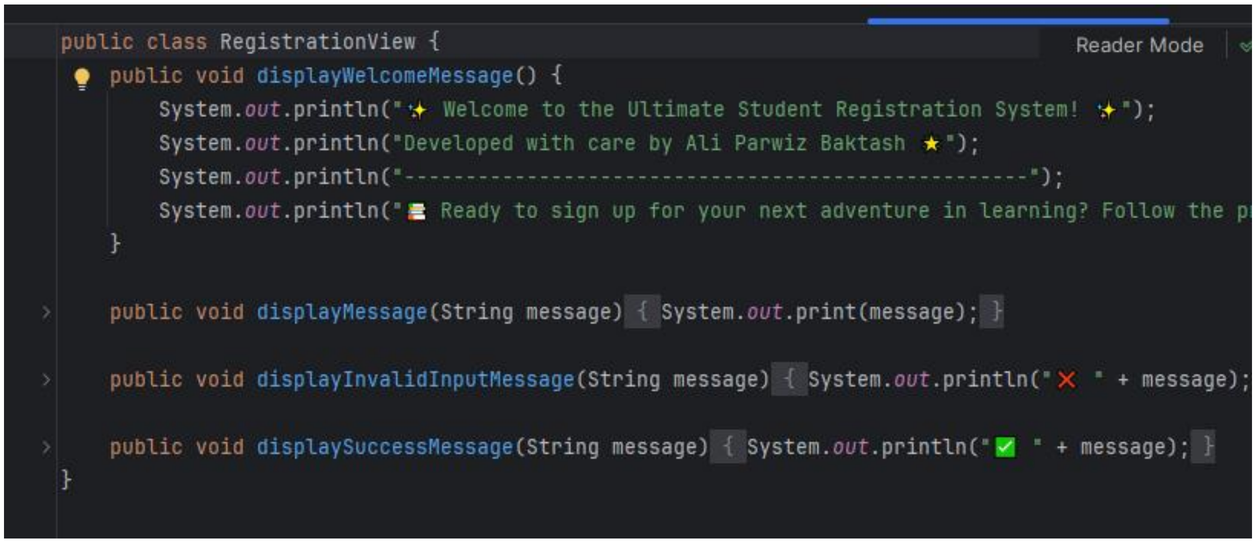
- **Model: Manages data and core logic (e.g., Student, Course, and Database classes). The Database class interacts with PostgreSQL to store and retrieve student information.**
- **View: Handles user interaction and displays messages through the RegistrationView class.**
- **Controller: Coordinates between the Model and View, processing user input and managing the registration flow in RegistrationController.**

Significance: Using MVC makes the code modular and adaptable to future changes. Each component can be modified independently, facilitating easier maintenance and debugging.



Reader Mode

```java
import ...

public class RegistrationController {
    private final RegistrationManager manager;
    private final Database database;
    private final RegistrationView view;

    // Constructor to set up the manager, database, and view components
    public RegistrationController(RegistrationManager manager, Database database, Registration
        this.manager = manager;
        this.database = database;
        this.view = view;
    }

    // This method starts the whole registration process
    public void startRegistrationProcess(Scanner scanner) {
        view.displayWelcomeMessage(); // Show a welcoming message to the user

        while (true) {
            // Prompt for student's name
            String name = getInputString(scanner, prompt: "Enter your name, please: ");
```



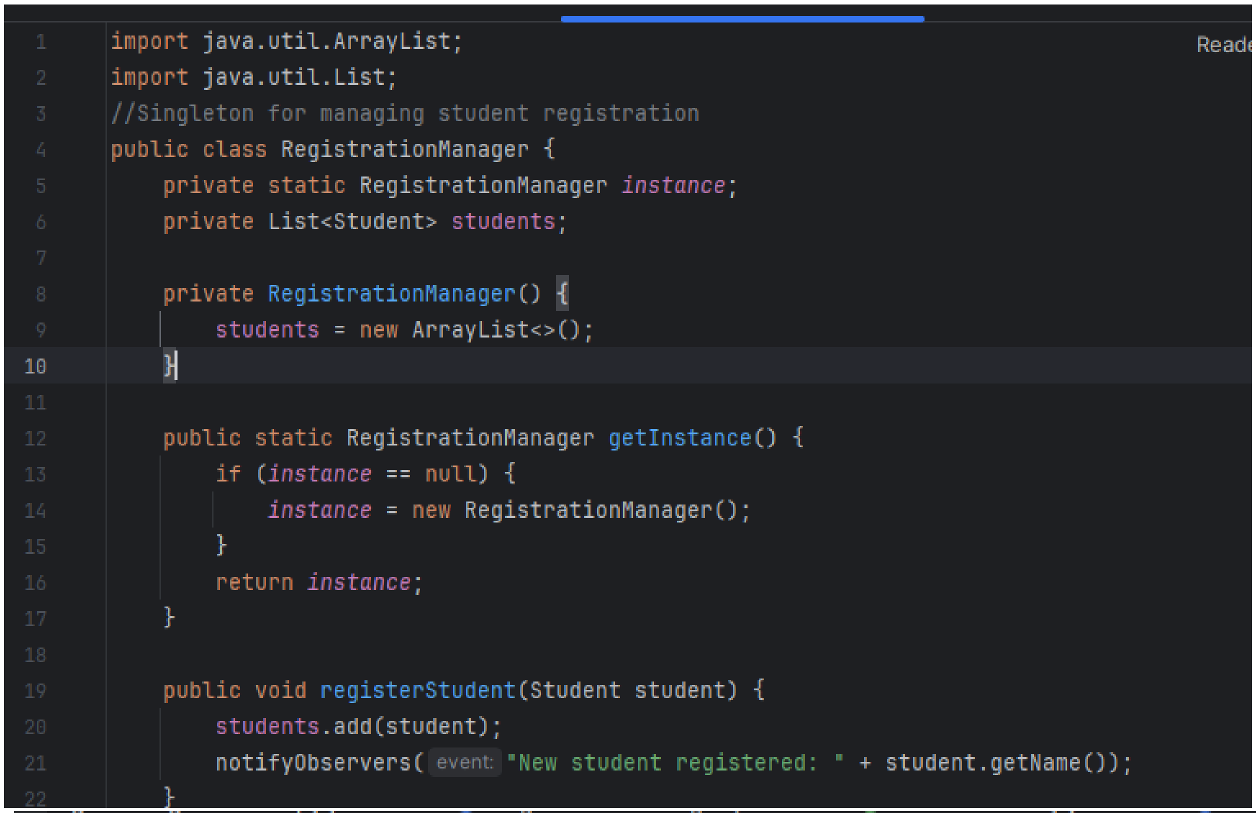
```java
public class RegistrationView {
    public void displayWelcomeMessage() {
        System.out.println("✨ Welcome to the Ultimate Student Registration System! ✨");
        System.out.println("Developed with care by Ali Parwiz Baktash ⭐");
        System.out.println("--------------------------------------------------");
        System.out.println("📚 Ready to sign up for your next adventure in learning? Follow the p
    }

    public void displayMessage(String message) { System.out.print(message); }

    public void displayInvalidInputMessage(String message) { System.out.println("❌ " + message);

    public void displaySuccessMessage(String message) { System.out.println("✅ " + message); }
}
```

2. Creational Patterns

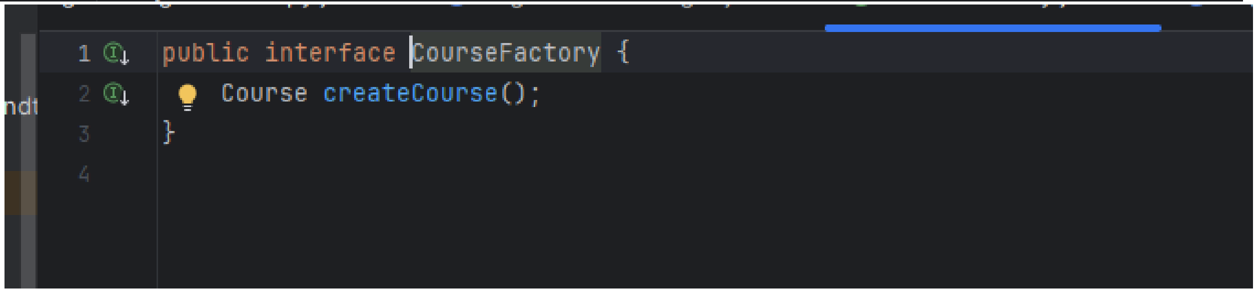
Singleton Pattern: The Singleton pattern is applied to the RegistrationManager class to ensure only one instance exists across the system. This instance centralizes student registrations and notifies observers about registration events.

Factory Method Pattern: This pattern is used to create Course objects, allowing dynamic instantiation of specific course types (ProgrammingCourse and DesignCourse) based on user input.

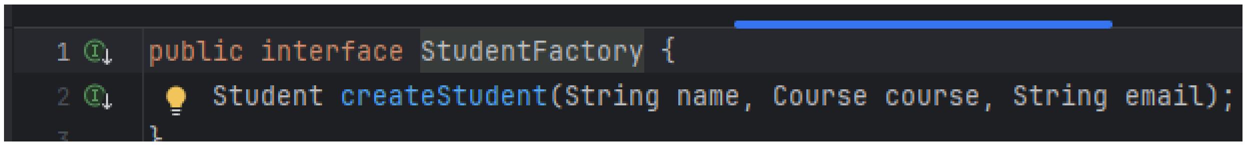
Abstract Factory Pattern: The Abstract Factory pattern is implemented to create different Student objects (e.g., RegularStudent and PremiumStudent). The StudentFactory interface provides a consistent way to create related objects without binding the code to specific classes.



```
import java.util.ArrayList;
import java.util.List;
//Singleton for managing student registration
public class RegistrationManager {
    private static RegistrationManager instance;
    private List<Student> students;

    private RegistrationManager() {
        students = new ArrayList<>();
    }

    public static RegistrationManager getInstance() {
        if (instance == null) {
            instance = new RegistrationManager();
        }
        return instance;
    }

    public void registerStudent(Student student) {
        students.add(student);
        notifyObservers( event: "New student registered: " + student.getName());
    }
```



```
public interface CourseFactory {
    Course createCourse();
}
```



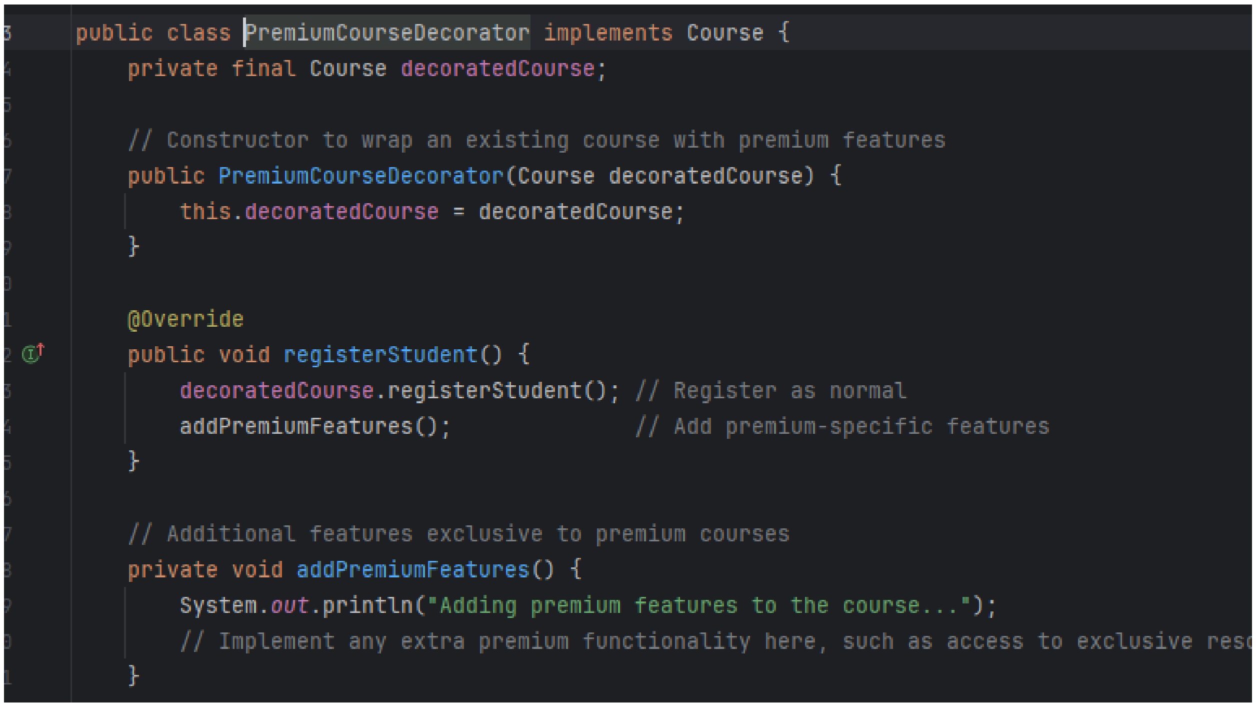
```
public interface StudentFactory {
    Student createStudent(String name, Course course, String email);
}
```

3. Structural Patterns

Decorator Pattern: The Decorator pattern was added to allow dynamic extension of Course functionalities. PremiumCourseDecorator wraps an existing Course object to add premium-specific features without modifying the original Course classes.

- **PremiumCourseDecorator: This class implements the Course interface and decorates a Course object with additional methods, like exclusive access for premium students.**

Example Usage: In the registration process, if a student is marked as premium, their selected Course is wrapped in a PremiumCourseDecorator to provide extra benefits.



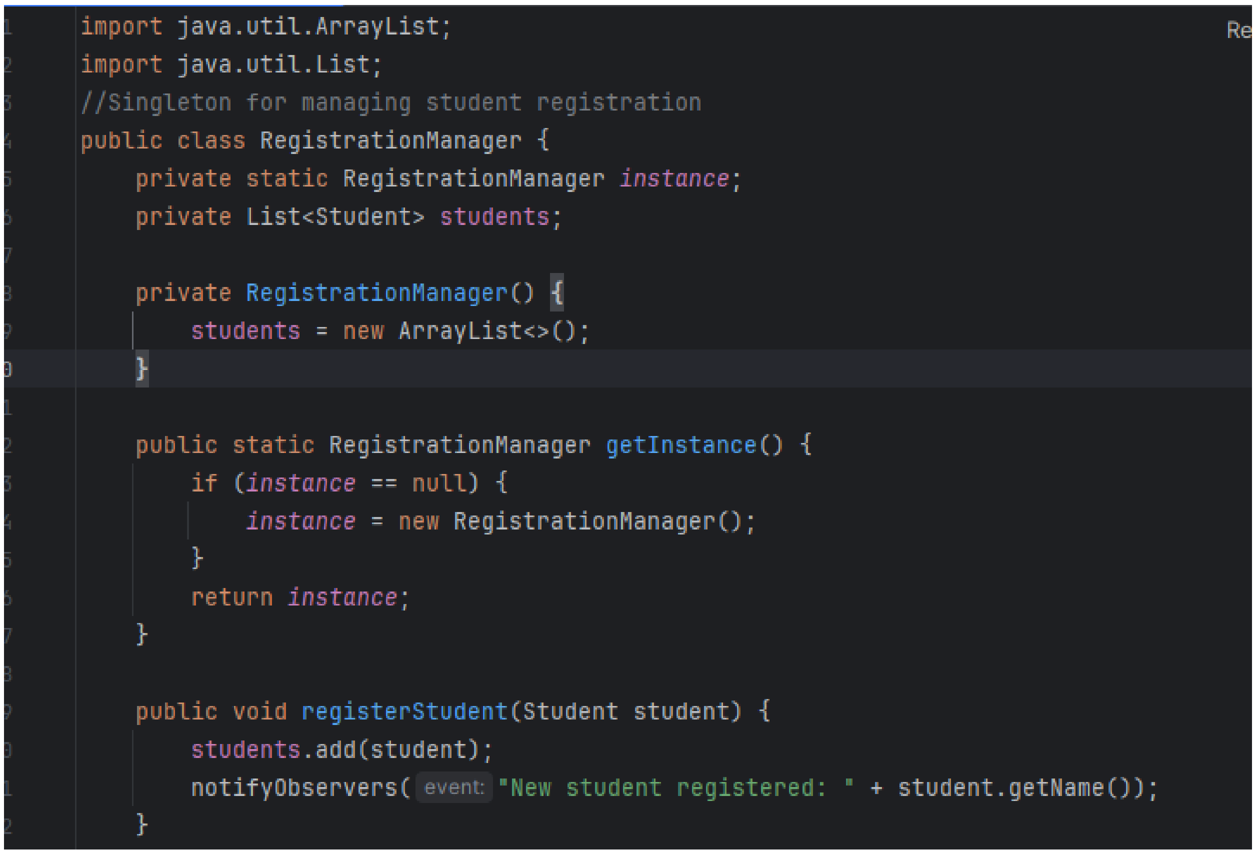
```java
public class PremiumCourseDecorator implements Course {
    private final Course decoratedCourse;

    // Constructor to wrap an existing course with premium features
    public PremiumCourseDecorator(Course decoratedCourse) {
        this.decoratedCourse = decoratedCourse;
    }

    @Override
    public void registerStudent() {
        decoratedCourse.registerStudent(); // Register as normal
        addPremiumFeatures();              // Add premium-specific features
    }

    // Additional features exclusive to premium courses
    private void addPremiumFeatures() {
        System.out.println("Adding premium features to the course...");
        // Implement any extra premium functionality here, such as access to exclusive resc
    }
```

Facade Pattern (optional): If included, the Facade pattern would provide a simplified interface to complex operations, like managing course registration and notifying students, by encapsulating these actions in a RegistrationFacade class.

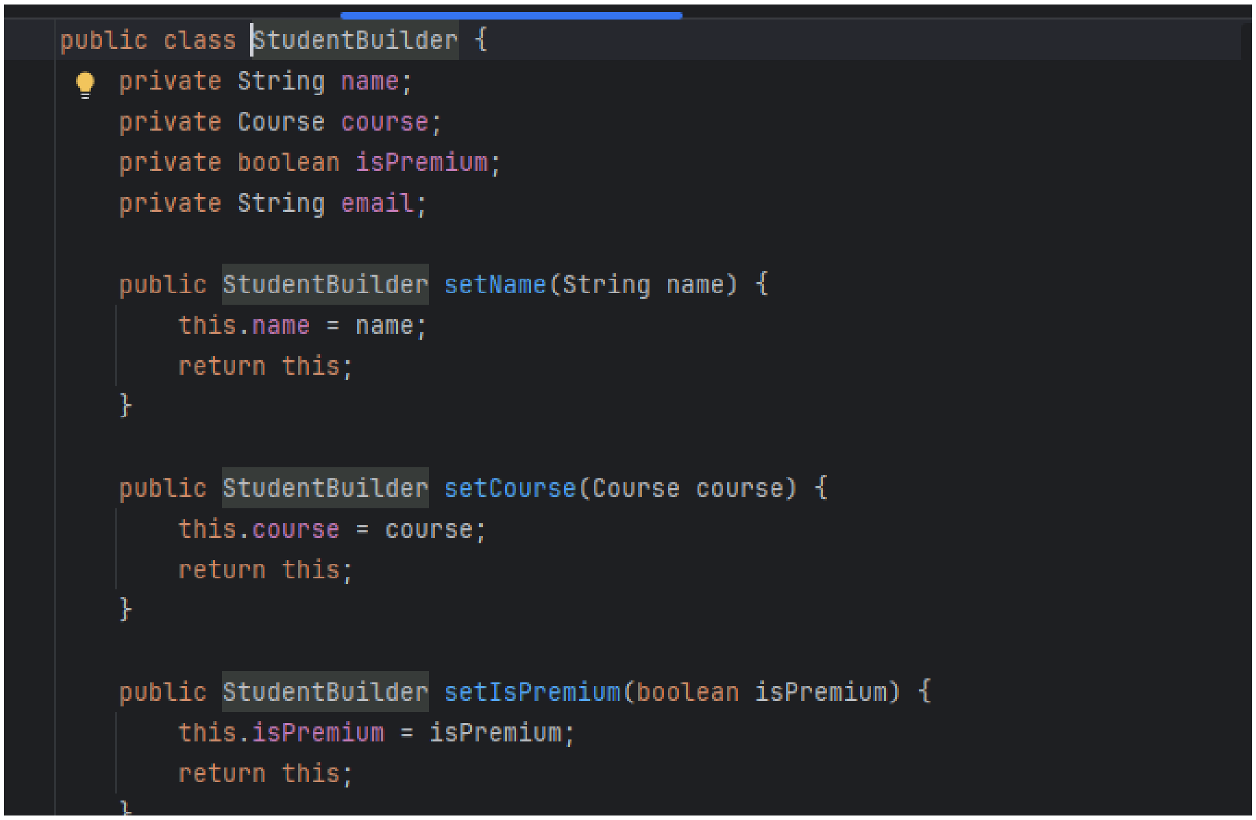
4. Behavioral Patterns

Observer Pattern: The Observer pattern is used to notify components of significant registration events. Observers, such as LoggingObserver and EmailNotificationObserver, are registered with RegistrationManager, allowing them to react to new student registrations independently.

Builder Pattern: The Builder pattern provides a flexible approach for constructing Student objects based on user input. The StudentBuilder class allows setting specific properties and then builds a fully configured Student object.

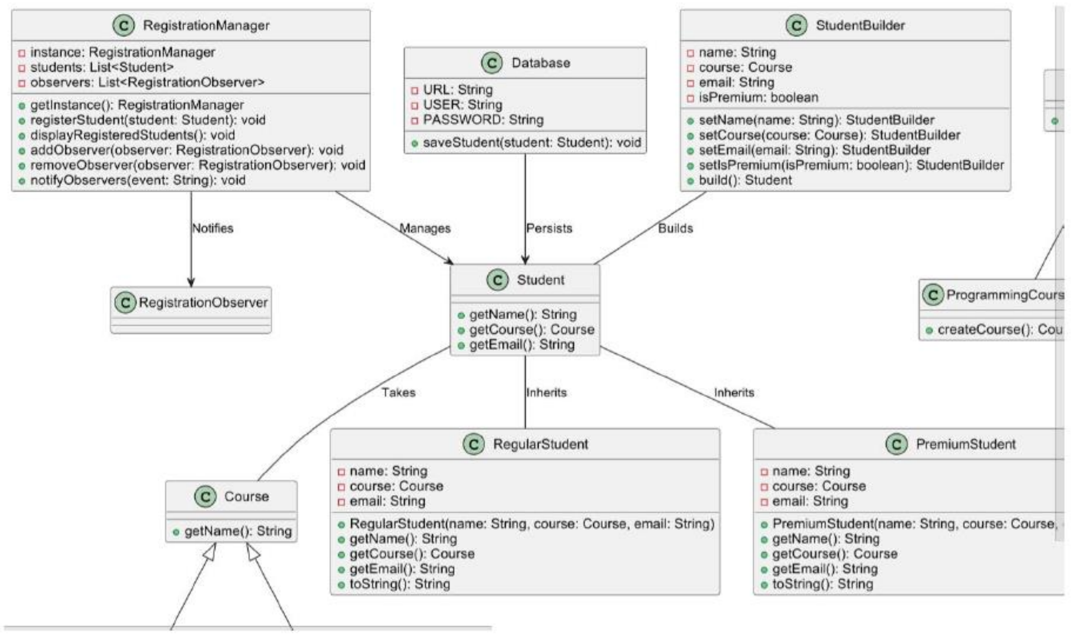


```java
import java.util.ArrayList;
import java.util.List;
//Singleton for managing student registration
public class RegistrationManager {
    private static RegistrationManager instance;
    private List<Student> students;

    private RegistrationManager() {
        students = new ArrayList<>();
    }

    public static RegistrationManager getInstance() {
        if (instance == null) {
            instance = new RegistrationManager();
        }
        return instance;
    }

    public void registerStudent(Student student) {
        students.add(student);
        notifyObservers( event: "New student registered: " + student.getName());
    }
```



```java
public class StudentBuilder {
    private String name;
    private Course course;
    private boolean isPremium;
    private String email;

    public StudentBuilder setName(String name) {
        this.name = name;
        return this;
    }

    public StudentBuilder setCourse(Course course) {
        this.course = course;
        return this;
    }

    public StudentBuilder setIsPremium(boolean isPremium) {
        this.isPremium = isPremium;
        return this;
    }
```

UML diagram



Conclusion

This project demonstrates the effective use of MVC architecture along with various design patterns to create a flexible and maintainable Student Registration System. Each design pattern plays a specific role:

- **MVC Architecture: Provides a structured, modular approach to organizing the system.**
- **Creational Patterns: Enhance flexibility in object creation (e.g., Singleton, Factory Method, Abstract Factory).**
- **Structural Patterns: Enable dynamic behavior extension (Decorator for premium features).**
- **Behavioral Patterns: Manage complex interactions (Observer for notifications, Builder for object construction).**

The project leverages Maven for dependency management and PostgreSQL for data storage, with JDBC handling database interactions. The structured use of design patterns ensures that the system is well-organized, adaptable to future changes, and easy to maintain.

Challenges Faced

1. **Database Integration: Configuring PostgreSQL and managing data connections required careful exception handling.**
2. **Pattern Integration: Combining multiple design patterns within the MVC framework required strategic planning to ensure smooth interactions.**
3. **Error Handling: Managing robust error handling for database operations and input validation was crucial for system stability.**
4. **Code Organization: Structuring the code to accommodate various patterns within the MVC framework was essential for clarity and future extensibility.**