Software Design Pattern End Term Report

Astana IT University

Instructor: Dr. Maksutov Akylbek

Ali Parwiz Baktash SE-2312

Student Registration System

Project Overview

Project Idea: **The idea is to create a student registration system that can efficiently manage the student registration process for various courses. The system uses different design patterns to ensure flexibility and scalability.**

Performance Goal: **The goal of this project is to create a reliable and efficient registration system that can handle a large number of student registrations and provide notifications of various events in the registration process.**

Performance Objectives:

Designing a system that can handle a large number of student registrations.

Utilizing design patterns to ensure flexibility and scalability of the system.

Providing the ability to notify various observers of events in the registration process.

Creating a system that can be easily adapted and extended to handle additional requirements in the future.

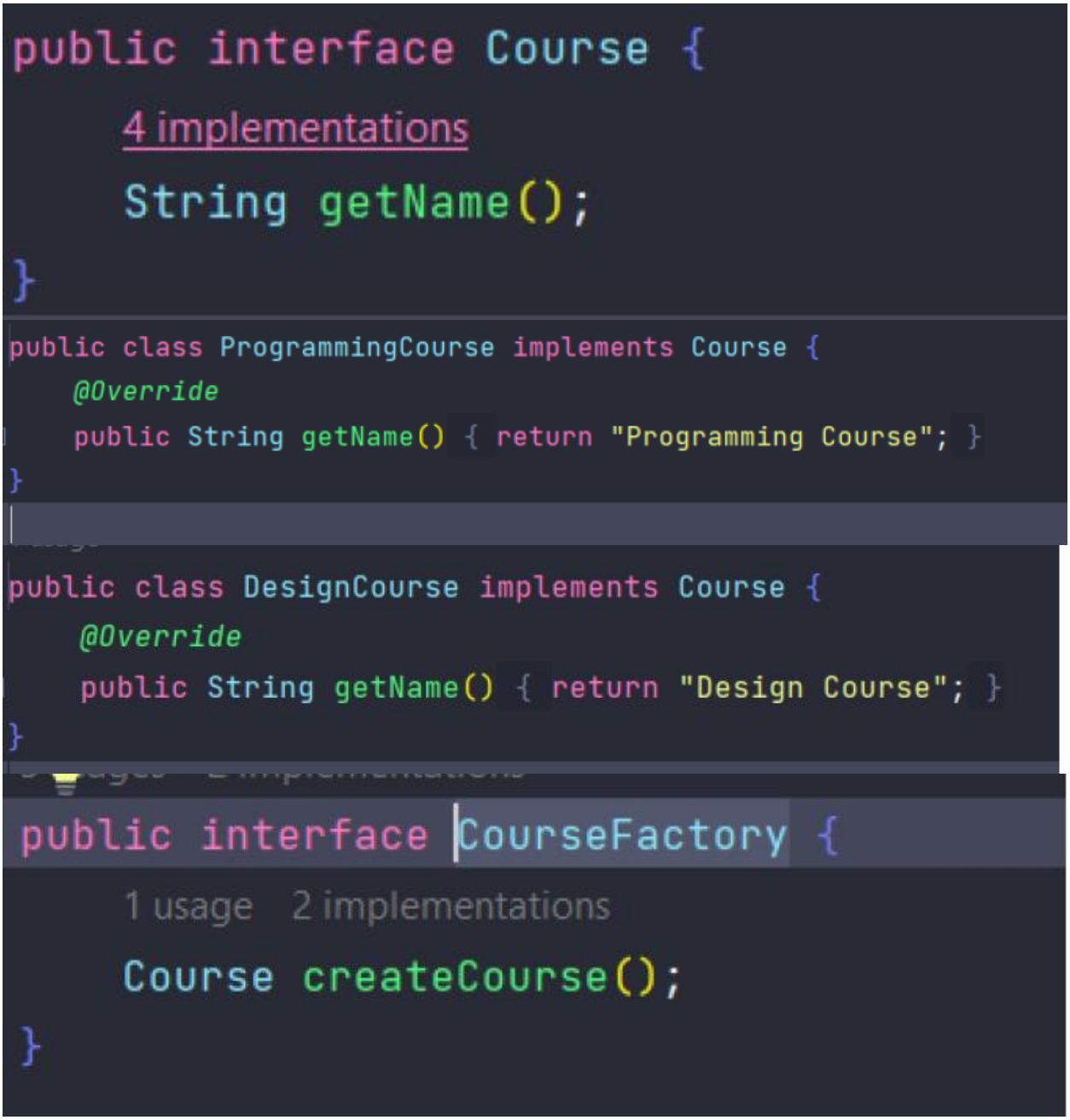
Main body

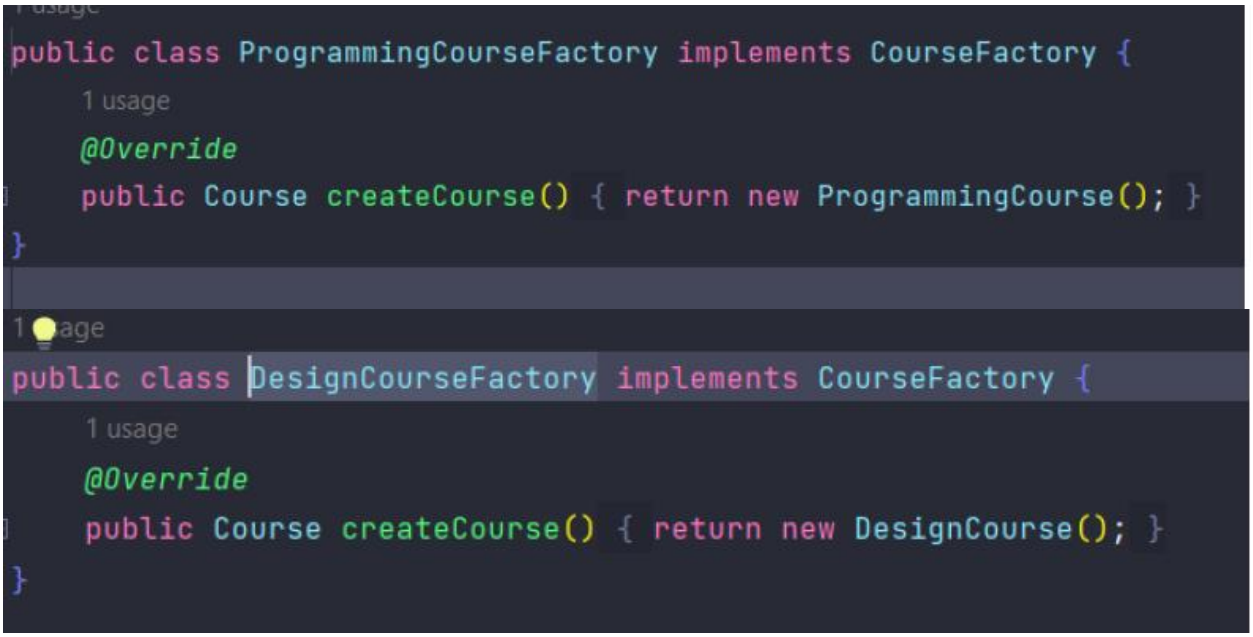
Singleton: The Singleton pattern is used for the RegistrationManager class to ensure that there is only one instance of this class. This is useful when you need exactly one object to coordinate actions in the system.

```java
public class RegistrationManager {
    3 usages
    private static RegistrationManager instance;
    3 usages
    private List<Student> students;

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

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

In this code, RegistrationManager is declared as private static, which means that it can be initialized only once. The getInstance() method checks if an instance of RegistrationManager has already been created. If not, it creates a new instance. If an instance already exists, the method returns that existing instance. This ensures that there is only one instance of the RegistrationManager class in the entire program.

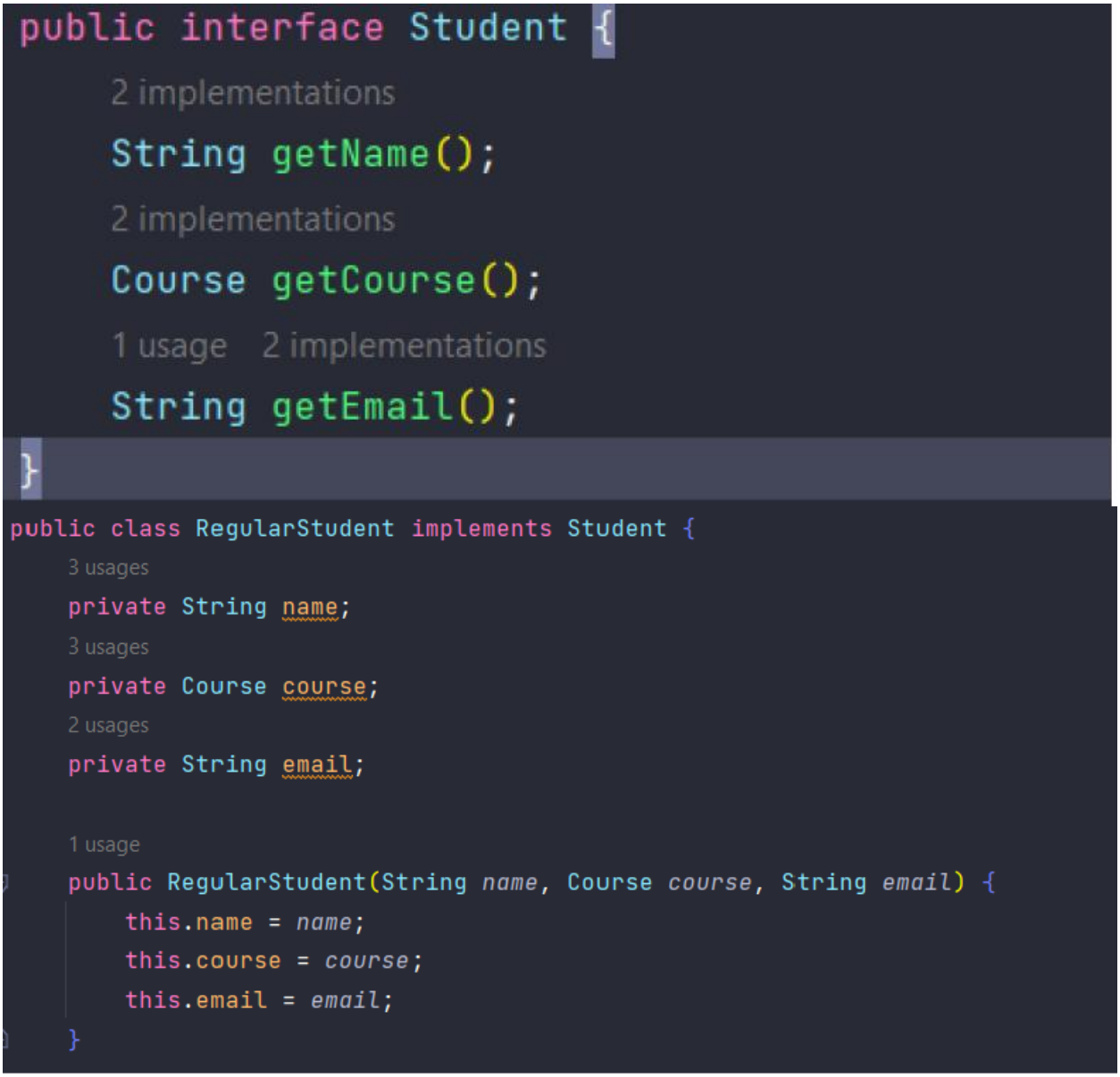
Factory Method: The Factory Method pattern is used to create Course objects. It allows you to delegate the logic of object creation to subclasses.


public interface Course {
4 implementations
String getName();
}
public class ProgrammingCourse implements Course {
@Override
public String getName() { return "Programming Course"; }
}
public class DesignCourse implements Course {
@Override
public String getName() { return "Design Course"; }
}
public interface CourseFactory {
1 usage 2 implementations
Course createCourse();
}



In this code, CourseFactory is an interface with a createCourse() method. The classes ProgrammingCourseFactory and DesignCourseFactory implement this interface and provide their own implementation of the createCourse() method, creating instances of ProgrammingCourse and DesignCourse respectively. This allows you to delegate the logic of object creation to subclasses, which is the key idea behind the Factory Method pattern.

Abstract Factory: The Abstract Factory pattern is used to create Student objects. It allows you to create families of related objects without being bound to specific classes.



```java
public interface Student {
    2 implementations
    String getName();
    2 implementations
    Course getCourse();
    1 usage   2 implementations
    String getEmail();
}
public class RegularStudent implements Student {
    3 usages
    private String name;
    3 usages
    private Course course;
    2 usages
    private String email;

    1 usage
    public RegularStudent(String name, Course course, String email) {
        this.name = name;
        this.course = course;
        this.email = email;
    }
```

```java
public class PremiumStudent implements Student {
    3 usages
    private String name;
    3 usages
    private Course course;
    3 usages
    private String email;

    1 usage
    public PremiumStudent(String name, Course course, String email) {
        this.name = name;
        this.course = course;
        this.email = email;
    }
```

In this code, StudentFactory is an interface with a createStudent() method. The RegularStudentFactory and PremiumStudentFactory classes implement this interface and provide their own implementation of the createStudent() method, creating instances of RegularStudent and PremiumStudent respectively. This allows you to create families of related objects without being bound to specific classes, which is the key idea behind the Abstract Factory pattern.

Builder: The Builder pattern is used to create Student objects with different configurations. This is useful when an object has many possible configurations.

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

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

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

In this code, StudentBuilder provides methods to set various student properties (name, course, isPremium) and then uses these properties to create a Student object using the build() method. This allows you to create Student objects with different configurations, which is the key idea behind the pattern Builder.

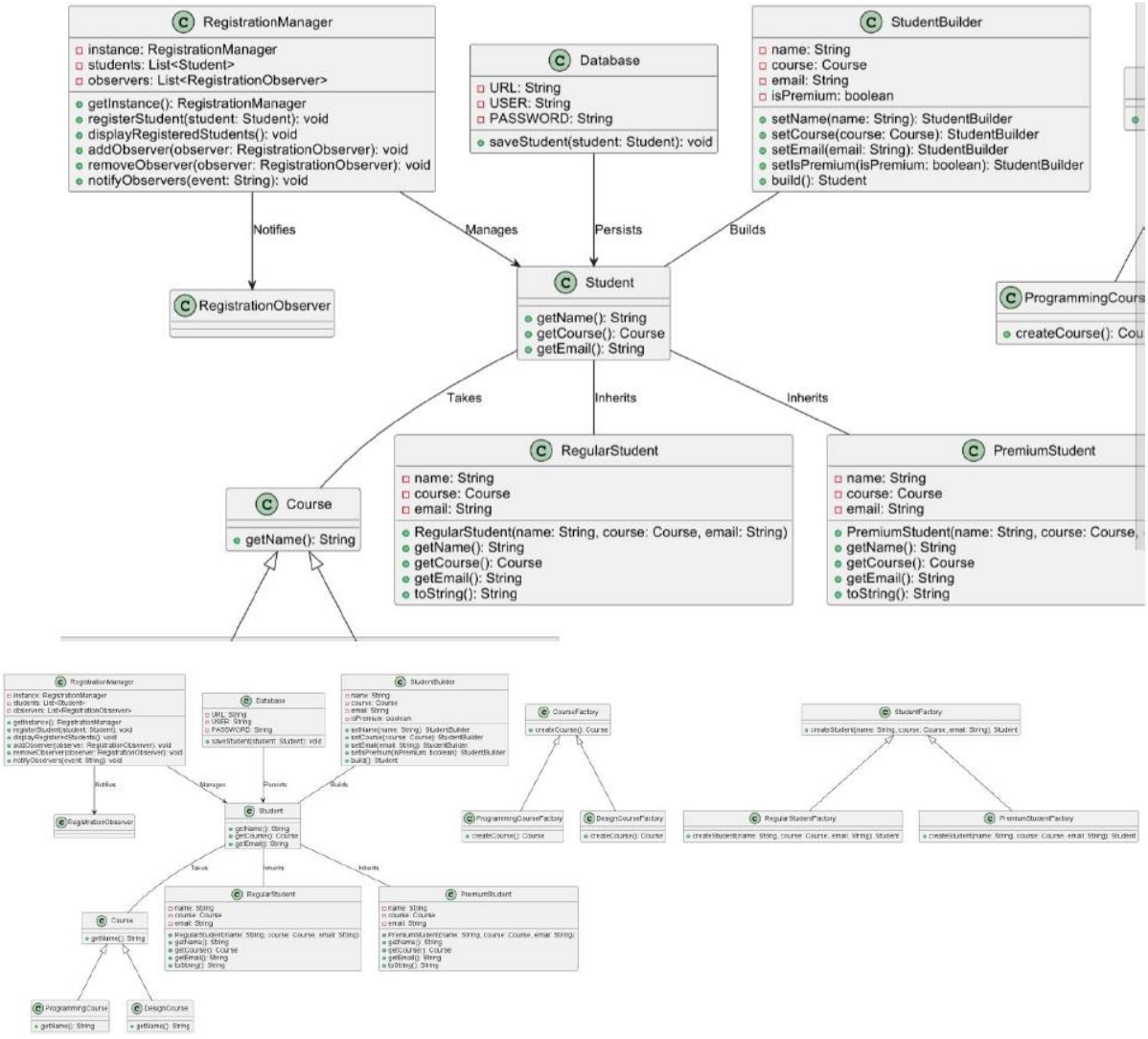
Observer: **The Observer pattern is used to notify observers of events. This is useful when the state of one object should affect other objects, but you don't want to hard link these objects together.**

RegistrationManager contains a list of observers that monitor registration events. When an event occurs (such as a new student registering), the RegistrationManager notifies all observers of the event by calling their update() method. This allows other objects to react to changes in the RegistrationManager without being hardwired into it, which is the key idea behind the Observer pattern.

Prototype: **The Prototype pattern is used to create copies of Course objects. This is evident from the use of the clone() method in the ProgrammingCourseCopy and DesignCourseCopy classes.**

```java
public class ProgrammingCourseCopy implements Course, Cloneable {
    @Override
    public String getName() { return "Programming Course Copy"; }

    @Override
    public ProgrammingCourseCopy clone() throws CloneNotSupportedException {
        return (ProgrammingCourseCopy) super.clone();
    }
}
```

UML diagram



Conclusion

This project demonstrates the use of various Java design patterns to manage student registration. Using Singleton, Observer, Factory Method, Abstract Factory, and Builder patterns, the code base is structured to provide flexibility, extensibility, and maintainability. Maven is used effectively for project management, and Postgre SQL provides easy interaction with databases. Regular study of Stack Overflow articles provided valuable insights and guidance. Working with databases, implementing design patterns, ensuring error-free execution and maintaining a clear code structure.

Challenges Faced:

Database Interaction: Establishing a connection and performing operations on the database presented a challenge. It required understanding JDBC and handling exceptions properly.

Pattern Implementation: Ensuring correct implementation of design patterns, especially in cases where multiple patterns interacted, required careful consideration and testing.

Error Handling: Properly handling exceptions and errors, both in database operations and in the application's logic, was crucial to ensure robustness.

Code Structure: Organizing the code to maintain clarity and readability, especially when dealing with multiple classes and patterns, required thoughtful planning.