Student Database Program

Jannatul Ferdousi
Roll: 8

1 Introduction

This article describes how object-oriented programming, or OOP, was used in the design and implementation of a student database program. The program creates random student data, randomizes the course selection process for each student, computes given evaluation metric-based grades, and offers features for showing a comprehensive student list, overall and course based student ranks.

2 Analysis of the OOP principles

- **Encapsulation**: Encapsulation is effectively utilized throughout the codebase. Class attributes such as \`name\`, \`roll\`, \`email\`, \`totalCourses\`, \`majorCourses\`, \`optionalCourse\`, \`cgpa\`, \`total_marks\`, \`courseName\`, \`courseType\`, \`credits\`, \`students\`, \`evaluationComponents\`, \`marks\`, and \`grade\` are appropriately encapsulated within their respective classes and accessed through getter and setter methods. This ensures data integrity and facilitates modularity.

- **Abstraction** : Abstraction is achieved through the use of classes and methods, which hide implementation details and expose only essential functionalities to external components. For instance, the Student class abstracts away the complexities of student management, providing methods like addMajorCourse() and setOptionalCourse() for interacting with course enrollments.The GradeCalculator class abstracts the concept of calculating grade and sorting students based on grade, hiding its internal details.

- **Inheritance**: Classes inherit features and attributes from their parent classes through inheritance. Classes like \`Student\`, \`Course\` and \`EvaluationComponent\` do not explicitly inherit from any other class, but they utilize composition to maintain relationships with other classes.

- **Polymorphism**: The GradeCalculator() and the GradeCalculator(List<Student> students, List<Course> courses) method in the GradeCalculator class exhibits polymorphic behavior by accepting different types of parameters.

3 Key OOP Features

- **Composition**: Composition is a prominent feature in this codebase. Classes such as \`Student\`, \`Course\` and \`EvaluationComponent\` maintain lists of other objects, such as courses within a student, students within a course, and evaluation components within a course. This composition allows for flexible relationships between objects and facilitates code reuse.

- **Single Responsibility Principle (SRP)**: The code demonstrates adherence to the SRP by assigning specific responsibilities to each class. For example, the `Student` class is responsible for managing student-related data and operations, while the `Course` class handles course-related functionalities. This separation of concerns enhances maintainability and readability.

- **Open/Closed Principle (OCP)**: The code shows flexibility and extensibility, making it compliant with the OCP. For instance, the `GradeCalculator` class can be extended to incorporate additional grading algorithms modifying existing code. Similarly, the `Student_Database` class can accommodate new courses and evaluation components seamlessly.

4 Unique Contents:

- **Data Randomization:** The code incorporates randomization to generate student names, email addresses, and evaluation metrics, adding variability and realism to the simulated student database. This unique content enriches the design and demonstrates adaptability to real-world scenarios.

- **Flexibility and Scalability:** The code allows adding new courses in the 'main' method through `sCourses` ArrayList. The `evaluationComponents` in `Course` allows for the specification of assessment criteria with configurable weightage for each course for each student, adding versatility to the evaluation process. The design allows for the addition of new courses, evaluation components, and students without necessitating extensive modifications to existing code.

- **Clear Separation of Concerns:** The code demonstrates a clear separation of concerns, with each class focusing on a specific aspect of the system. This separation enhances maintainability, testability, and code reusability. For example,the code includes unique features such as the `GradeCalculator`, which computes grades and ranks students based on their performance.

- **Logic Handling:** Proper logic (constraint of total marks to be 100 and obtained mark for each assessment component cannot be greater than its weightage) are added to handle potential errors or unexpected situations.

5 Program Functionalities :

- **Student Data Generation:** The program generates random student names, rolls, and emails.
- **Course Selection:** Each student is assigned a set of 3 major courses and 1 optional one randomly.
- **Evaluation Metric Selection:** For each student a random value is assigned for each course, for each evaluation metric (Final, mid,attendance, assignment, etc. as required by the chosen metric) maintaining the constraint of having 100 in each course.
- **Grade Calculation:** Grades are calculated for each course, considering factors like midterms, regular, finals, and ct.
- **Comprehensive List**: A comprehensive list containing all the information relevant to a particular course - students, their information, and their chosen assessment criteria.
- **Overall Rank:** Students are ranked based on their overall GPA and then their total marks.
- **Course-Based Rank:** Students are ranked within each course based on their grades in that specific course and then total marks.

6 Conclusion:

In conclusion, the provided code demonstrates an object-oriented design that adheres to fundamental principles while incorporating unique features to manage student data, courses, and evaluations effectively. With encapsulation, composition, polymorphism and abstraction the codebase exhibits scalability, flexibility, and maintainability.