Ryan Cheung & Francisco Ortega, Lab 2 Group 12

MAIN OBJECTIVE: **create a program for a student grade management system. Must use at least 3 classes with attributes and methods. Use all of the OOP principles to connect these classes together.**

MAIN FUNCTIONS:

1. Add students to the system
2. Add courses to the system
3. Assign grades to students for specific courses
    a. just represent 1 grade for the whole class (assume that this student has completed this course, final exam and all)
4. Calculate student grade avg
5. Display student details including grades

Classes

- Person
    - Attributes
        - Name (str)
    - Method
        - _ _ str_ _ ()

- Student
    - INHERITS PERSON
    - Attributes
        - Name (str, from Person)
        - ID (str)
        - Courses taken with grades (dict)
            - {“course” (Course class): “Grade” (Grade Class), etc….. }
    - Methods
        - Enroll() - enroll in new class, add to student courses dictionary
        - __ str __ ()
- Teacher
    - INHERITS PERSON
    - Attributes:
        - Name (str, from Person)
        - ID (str)
        - courses teaching (list)
    - Methods:
        - __ str __()

- Grades
    - Dependency w Student
    - Attributes
        - Score (float, raw score)
        - Grade (string, letter grade)

    - Methods
        - __ str __ ()
            - Just to represent grade printed out
- Course
    - Aggregates Teacher for attribute
    - Attributes
        - Code (str)
        - Title (str)
        - Credits (int)
        - Teacher (Teacher)
    - Methods
        - _ _ str_ _ ()
- StudentRecord
    - Aggregates Student
    - Attributes
        - Student (from custom student class / obj)
        - Semester
        - Year
    - Methods
        - add_grade()
            - Adds grade to a student's course completed
        - calculate_gpa()
            - Calculates GPA of student's courses taken so far
        - print_record()
            - Prints out record of student's performance
- GradeSystem
    - Aggregates Course and Student
    - Attributes
        - Courses: list
        - Students: list
    - Methods:
        - add_course() - does as implied, adds course to student course dictionary
        - remove_course() - does as implied
        - add_student() - does as implied, creates another student
        - remove_student() - does as implied

DESIGN CHOICES:

To demonstrate inheritance, we created a parent class, Person, which serves as a generic template for storing a name. Both Student and Teacher inherit the name attribute from Person and extend it with their own unique attributes. The Student class includes a dictionary (courses_and_grades) to track courses taken and corresponding grades, while the Teacher class maintains a list of courses they teach. This design avoids redundancy and promotes code reuse.

The Grade and Course classes encapsulate essential attributes for the system. The Grade class holds a score (float) and a grade (string) to represent percentage and letter grades, respectively. The Course class includes a code (string), title (string), credits (int), and a Teacher object to represent the instructor. These classes abstract away the complexity of managing grades and courses, allowing Student and Teacher to focus on their core responsibilities.

The StudentRecord and GradeSystem classes handle higher-level operations. StudentRecord aggregates a Student object, a semester (string), and a year (int). It provides methods like add_grade() to update grades, calculate_gpa() to compute the student's GPA, and print_record() to display the student's academic record. The GradeSystem class manages lists of courses and students, offering methods to add or remove courses and students. These classes encapsulate the logic for managing academic records and system-wide operations.

Application of OOP Principles

1. Encapsulation

The Person class encapsulates the name attribute and provides get_name() and set_name() methods. The Student class encapsulates its courses_and_grades dictionary, allowing grades to be added or retrieved only through methods like enroll() and add_grade(). The GradeSystem class encapsulates lists of courses and students, ensuring they are modified only through its methods.

2. Abstraction

The Grade class abstracts the representation of a grade, allowing other classes to work with grades without worrying about how they are stored or calculated. The Course class abstracts the details of a course, such as its code, title, and instructor, enabling Student and Teacher to interact with courses at a higher level. The StudentRecord class abstracts the process of managing a student's academic record, providing methods like calculate_gpa() without exposing the underlying data structure.

3. Inheritance

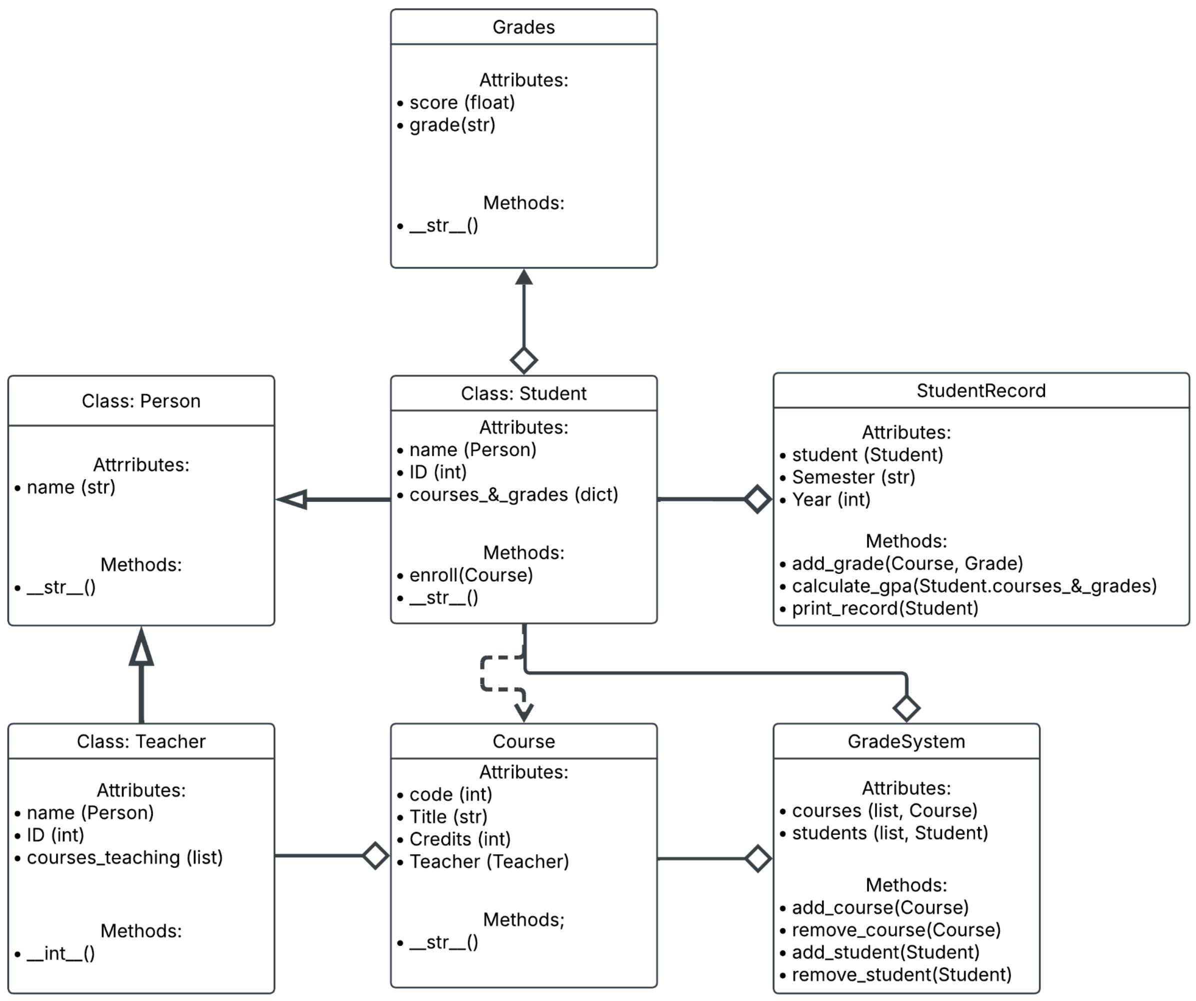
The Person class serves as the parent class for Student and Teacher, providing a common name attribute and reducing redundancy. Both Student and Teacher extend Person with their own attributes (ID, courses_and_grades for Student; ID, courses_teaching for Teacher).

4. Polymorphism

The __str__() method is overridden in multiple classes (Person, Student, Teacher, Course, Grade, etc.) to provide a consistent way of representing objects as strings. The GradeSystem class can interact with any Student or Course object, regardless of their specific implementations, demonstrating polymorphism through shared interfaces.

Ryan Cheung & Francisco Ortega, Lab 2 Group 12

UML REPRESENTATION (also included as txt file in python code package):



Difficulties:

It was a little difficult to decide what kind of relationships that we wanted to have for our different classes. What classes and members / functions were redundant or unnecessary? We had to constantly ask that question to hone in on the real design of the program.

It was also the first time for me (Ryan) to make a UML Diagram. I had to constantly look over the textbook material to ensure that I was making it correctly. I verified with my partner to make sure that it was ok before proceeding to the implementation phase. I used LucidChart to create the diagram in an easier workflow so that we could get started quickly.

Teamwork online is always a bit of a hassle, but Francisco was very cooperative and helpful the whole time. The problems mentioned above were solved relatively quickly thanks to his input and timely work.