E - Learning Platform Documentation

Pe, Jiovany

Ramos, Princess Nicole R.

Duque, Zyron

College of Engineering And Information Technology

University Of Southern Mindanao

December 16, 2024

1. Introduction

We developed a Python-based Learning Management System featuring classes for Student, Instructor, and Admin, each with role-specific functionalities like course enrollment, grading, and schedule management. Core entities such as Course, Assignment, and Schedule encapsulate educational details, while a centralized PlatformAdmin handles data storage and user management. The system's modular design ensures scalability and efficient educational interactions.

2. Requirements

Encapsulation

Encapsulation is implemented in the system using private, protected, and public attributes within the classes:

- **Private Attributes:** Attributes like _id and _password in the User class are private, ensuring secure data handling.
- **Protected Attributes:** Attributes such as _name and _email in the Person class provide controlled subclass access.
- **Public Attributes:** Elements like courses and grades in the Student class are public, enabling safe usability.

Polymorphism

Polymorphism is demonstrated in the system through method overriding in derived classes, allowing different behaviors for the same method. For instance, the get_details() method in the Person base class is overridden in subclasses like Student and Instructor to provide specific details relevant to each role. This ensures that the same method adapts dynamically based on the subclass, enhancing the system's flexibility and maintainability.

Access Modifiers

The code uses different access modifiers to control how attributes are accessed within the classes:

- **Private (__):** Attributes like __password in the User class are fully hidden from outside the class, ensuring sensitive information is securely managed. Access is controlled via class methods, maintaining data integrity.
- **Protected (_):** Attributes such as _name and _email in the Person class are partially hidden, allowing access within the class and its subclasses (Student and Instructor). This enables controlled inheritance and extension of functionality while safeguarding internal details.
- **Public:** Methods like get_details() in the Person and User classes are accessible from anywhere, enabling safe interaction with user data. These methods allow external code to retrieve or update information like email or name while preserving encapsulation for critical attributes.

Abstract Class

The Person class is implemented as an abstract class using the ABC module, enforcing a consistent structure and behavior across its subclasses like Student and Instructor. This ensures that all subclasses implement key methods such as get_details(), promoting uniformity and adherence to the system's design principles.

Inheritance

The system demonstrates inheritance through various classes:

Person is the base class.

- **Admin** and **Student** inherit from Person.
- **Instructor** also inherits from Person and extends functionality for course management.

User is the base class for user-related operations.

- **Admin** and **Student** inherit from User to manage specific roles and responsibilities.

Student and **Instructor** interact with courses:

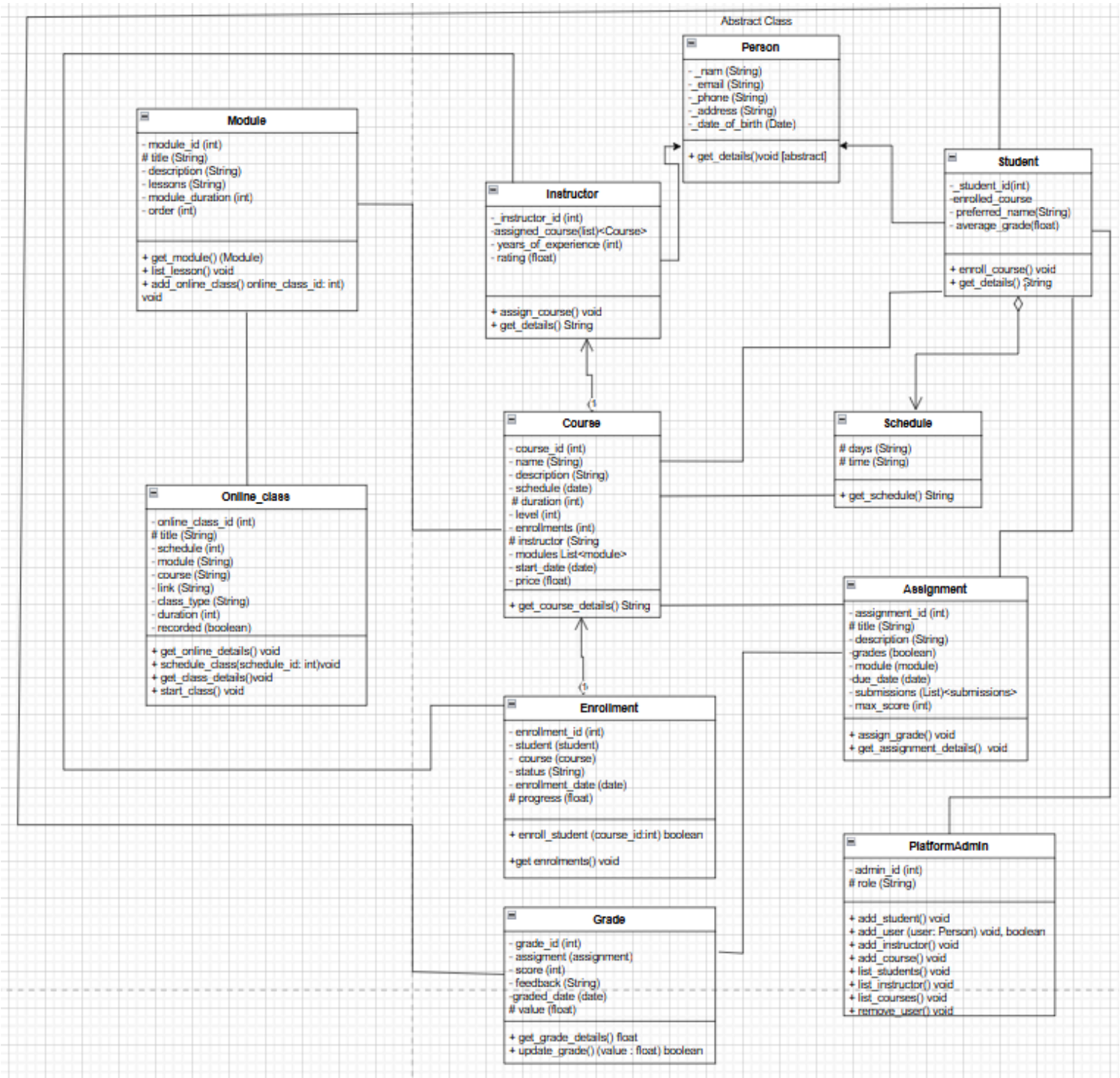
- **Student** can enroll in courses, assign grades, and view schedules.
- **Instructor** is responsible for assigning courses to themselves and managing schedules.

Class Structure

The implementation includes:

- **13 Classes**
- **16 Class Attributes**
- **12 Instance Attributes**
- **13 Instance Methods**
- **8 Static Methods**
- **5 Class Methods**

3. Class Diagram



Person (Abstract Base Class)

The Person class serves as an abstract base class for managing general person-related functionalities in the system. It defines attributes such as name, phone, address, and date of birth, along with methods for:

- Validating user inputs such as name, email, and phone number.
- Formatting the address and calculating age based on the date of birth.
- Providing a structure for subclasses (Admin, Student, and Instructor) by enforcing the implementation of the get_details method.

The class includes staticmethod and classmethod features to validate and manage data efficiently.

```python
class Person(ABC):
    def __init__(self, name, phone, address, date_of_birth):
        self._name = name
        self._phone = phone
        self._address = address
        self._date_of_birth = date_of_birth

    @abstractmethod
    def get_details(self):
        pass

    @staticmethod
    def validate_email(email):
        return "@" in email

    @staticmethod
    def validate_phone(phone):
        return phone.isdigit() and len(phone) >= 10
    @staticmethod
    def calculate_age(date_of_birth, current_date):
        return current_date.year - date_of_birth.year

    @staticmethod
    def format_address(address):
        return address.title()

    @staticmethod
    def validate_name(name):
        return name.isalpha()

    @classmethod
    def create_from_dict(cls, data):
        return cls(data['name'], data['phone'], data['address'], data['date_of_birth'])

    @classmethod
    def create_default(cls):
        return cls("Default Name", "0000000000", "Default Address", None)
```

User

The User class handles user account details such as ID, email, password, role, and a reference to the associated Person instance. It provides methods for:

- Retrieving user details.
- Placeholder methods (create, update, delete) for managing user accounts.

```python
class User:
    def __init__(self, id, email, password, role, person):
        self._id = id
        self._password = password
        self._email = email
        self._person = person
        self._role = role

    def get_details(self):
        return {
            "email": self._email,
            "password": self._password,
            "role": self._role,
            "person": self._person.get_details()
        }

    def create(self):
        # TODO
        pass

    def update(self):
        # TODO
        pass

    def delete(self):
        # TODO
        pass
```

Admin (Subclass of Person)

The Admin class specializes the Person class to represent administrators in the platform. It provides methods to:

- Retrieve admin-specific details such as name, phone, and address.
- Manage platform-related tasks (facilitated through the PlatformAdmin class).

```python
class Admin(Person):
    def __init__(self, name, phone, address, date_of_birth):
        super().__init__(name, phone, address, date_of_birth)

    def get_details(self):
        return {
            "name": self._name,
            "phone": self._phone,
            "address": self._address,
            "date_of_birth": self._date_of_birth
        }
```

Student (Subclass of Person)

The Student class extends the Person class to include attributes and methods specific to students:

- **Attributes:**
  - **student_id: Unique identifier for the student.**
  - **courses: List of enrolled courses.**
  - **grades: Dictionary for storing course grades.**
- **Methods:**
  - **Enroll in courses, view courses, and manage grades.**
  - **View and manage schedules for enrolled courses.**

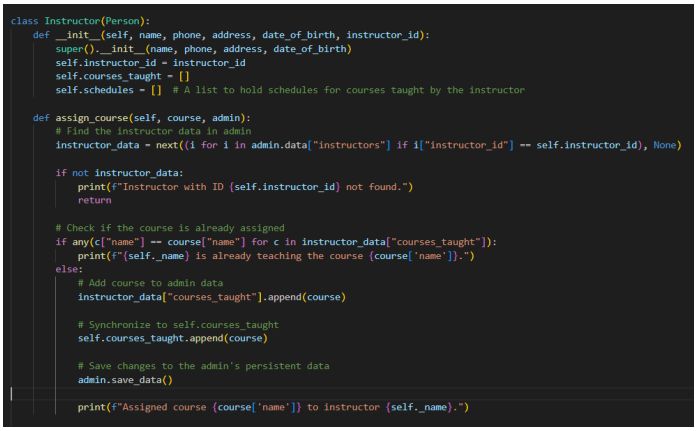
The class ensures validation of enrollment and prevents duplicate entries.

```python
class Student(Person):
    def __init__(self, name, phone, address, date_of_birth, student_id):
        super().__init__(name, phone, address, date_of_birth)
        self.student_id = student_id
        self.courses = []  # List to hold enrolled courses
        self.grades = {}  # Dictionary to hold grades for courses

    def enroll_course(self, course):
        # Check if the course is already enrolled based on course_id
        if course['course_id'] not in [c['course_id'] for c in self.courses]:
            self.courses.append(course)
            print(f"{self._name} enrolled in {course['name']}.")
        else:
            print(f"{self._name} is already enrolled in {course['name']}.")

    def assign_grade(self, course_name, grade):
        if course_name in self.grades:
            print(f"{self._name} already has a grade for {course_name}.")
        else:
            self.grades[course_name] = grade  # Make sure this is a dictionary update
            print(f"Assigned grade {grade} to {self._name} for course {course_name}.")


    def get_details(self):
        return {
            "name": self._name,
            "phone": self._phone,
            "address": self._address,
            "date_of_birth": self._date_of_birth,
            "student_id": self.student_id,
            "courses": self.courses,
            "grades": self.grades,
        }

    def view_courses(self):
        if not self.courses:
            print(f"{self._name} is not enrolled in any courses yet.")
        else:
            print(f"Courses enrolled by {self._name}:")
            for course in self.courses:
                print(f"- {course['name']}")

    def view_grades(self):
        if not self.grades:
            print(f"{self._name} has no grades assigned.")
        else:
            print(f"Grades for {self._name}:")
            for grade_info in self.grades:  # Iterate over the list of grades
                course_name = grade_info['course_name']
                grade = grade_info['grade']
                print(f"Course: {course_name}, Grade: {grade}")
```

Instructor (Subclass of Person)

The Instructor class builds on Person to include instructor-specific attributes:

- **Attributes:**
  - **instructor_id: Unique identifier for the instructor.**
  - **courses_taught: List of courses the instructor teaches.**
  - **schedules: A list to manage schedules for each course.**
- **Methods:**
  - **Assign courses, create assignments, and assign grades to students.**
  - **Manage course schedules and retrieve details about courses taught.**

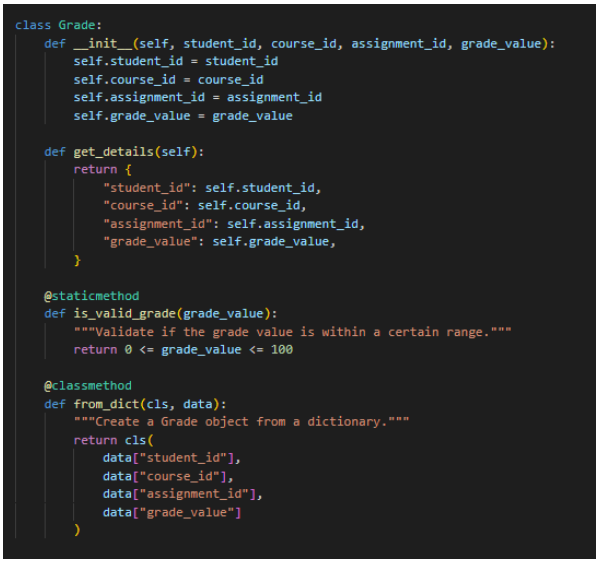


```python
class Instructor(Person):
    def __init__(self, name, phone, address, date_of_birth, instructor_id):
        super().__init__(name, phone, address, date_of_birth)
        self.instructor_id = instructor_id
        self.courses_taught = []
        self.schedules = []  # A list to hold schedules for courses taught by the instructor

    def assign_course(self, course, admin):
        # Find the instructor data in admin
        instructor_data = next((i for i in admin.data["instructors"] if i["instructor_id"] == self.instructor_id), None)

        if not instructor_data:
            print(f"Instructor with ID {self.instructor_id} not found.")
            return

        # Check if the course is already assigned
        if any(c["name"] == course["name"] for c in instructor_data["courses_taught"]):
            print(f"{self._name} is already teaching the course {course['name']}.")
        else:
            # Add course to admin data
            instructor_data["courses_taught"].append(course)

            # Synchronize to self.courses_taught
            self.courses_taught.append(course)

            # Save changes to the admin's persistent data
            admin.save_data()

            print(f"Assigned course {course['name']} to instructor {self._name}.")
```

Grade

The Grade class represents a grade entry for a specific student, course, and assignment. It includes methods for:

- **Validating grade values.**
- **Creating instances from dictionaries for flexibility.**



```python
class Grade:
    def __init__(self, student_id, course_id, assignment_id, grade_value):
        self.student_id = student_id
        self.course_id = course_id
        self.assignment_id = assignment_id
        self.grade_value = grade_value

    def get_details(self):
        return {
            "student_id": self.student_id,
            "course_id": self.course_id,
            "assignment_id": self.assignment_id,
            "grade_value": self.grade_value,
        }

    @staticmethod
    def is_valid_grade(grade_value):
        """Validate if the grade value is within a certain range."""
        return 0 <= grade_value <= 100

    @classmethod
    def from_dict(cls, data):
        """Create a Grade object from a dictionary."""
        return cls(
            data["student_id"],
            data["course_id"],
            data["assignment_id"],
            data["grade_value"]
        )
```

Assignment

The Assignment class captures assignment details such as title, description, due date, and associated course. It provides methods to:

- **Retrieve assignment details.**
- **Validate deadlines and manage assignment creation.**

```python
class Assignment:
    def __init__(self, assignment_id, title, description, due_date, course_id):
        self.assignment_id = assignment_id
        self.title = title
        self.description = description
        self.due_date = due_date
        self.course_id = course_id

    def get_details(self):
        return {
            "assignment_id": self.assignment_id,
            "title": self.title,
            "description": self.description,
            "due_date": self.due_date,
            "course_id": self.course_id,
        }
    @staticmethod
    def is_due_soon(due_date, current_date):
        """Check if the assignment is due soon given the current date."""
        return (due_date - current_date).days <= 3

    @classmethod
    def from_dict(cls, data):
        """Create an Assignment object from a dictionary."""
        return cls(
            data["assignment_id"],
            data["title"],
            data["description"],
            data["due_date"],
            data["course_id"]
        )
```

Schedule

The Schedule class represents course schedules and provides attributes such as start date, end date, class time, and days of the week. Methods include:

- **Checking if a schedule is active.**
- **Retrieving schedule details.**

```python
class Schedule:
    def __init__(self, course_name, start_date, end_date, class_time, days):
        self.course_name = course_name
        self.start_date = start_date
        self.end_date = end_date
        self.class_time = class_time
        self.days = days

    def get_details(self):
        return {
            "course_id": self.course_id,
            "start_date": self.start_date,
            "end_date": self.end_date,
            "class_time": self.class_time,
            "days": self.days,
        }
    @staticmethod
    def is_active(start_date, end_date, current_date):
        """Check if the schedule is currently active."""
        return start_date <= current_date <= end_date

    @classmethod
    def from_dict(cls, data):
        """Create a Schedule object from a dictionary."""
        return cls(
            data["course_id"],
            data["start_date"],
            data["end_date"],
            data["class_time"],
            data["days"]
        )
```

Course

The Course class encapsulates details about courses, such as their name, description, schedule, duration, level, instructor, and price. It provides functionality to manage course-related information.

```python
class Course:
    def __init__(self, course_id, name, description, schedule, duration, level, enrollments, instructor, module_list, start_date,price):
        self.course_id = course_id
        self.name = name
        self.description = description
        self.schedule = schedule
        self.duration = duration
        self.level = level
        self.enrollments = enrollments
        self.instructor = instructor
        self.module_list = module_list
        self.start_date = start_date
        self.price = price

    def get_course_details(self):
        return{

        }
```

Module

The Module class represents individual modules within a course, including their titles, descriptions, lessons, and order within the course. It facilitates managing course content in smaller units.

```
class Module:
    def __init_(self, module_id, title, description, lessons, module_duration, order):
        self.module_id = module_id
        self.title = title
        self.description = description
        self.lessons = lessons
        self.module_duration = module_duration
        self.order = order

    def get_module():
        return{

        }
```

Online Class

The OnlineClass class models online sessions with attributes such as the class title, schedule, type (live or recorded), and duration.

```
class Online_class:
    def __init__(self, online_class_id, title, schedule, module, link, class_type, duration, recorded):
        self.online_class_id = online_class_id
        self.title = title
        self.schedule = schedule
        self.module = module
        self.link = link
        self.class_type = class_type
        self.duraton = duration
        self.recorded = recorded

    def get_online_details():
        return{

        }
```

Enrollment

The Enrollment class represents the link between students and courses. It manages enrollment details such as status, date, and progress in the course.

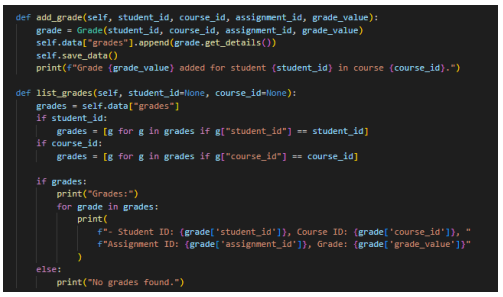
```
class Enrollment:
    def __init__(self, enrollment_id, student, course, status, enrollment_date, progress):
        self.enrollment_id = enrollment_id
        self.student = student
        self.course = course
        self.status = status
        self.enrollment_date = enrollment_date
        self.progress = progress

    def get_enroll_student():
        return{

        }
```

PlatformAdmin

The PlatformAdmin class acts as the central controller for managing the system's data. It provides methods for:

- **Managing users, courses, assignments, grades, and schedules.**
- **Persisting data to a JSON file for storage and retrieval.**
- **Handling user authentication and system workflows.**

```
class PlatformAdmin:
    def __init__(self, data_file="data2.json"):
        self.data_file = data_file
        self.data = {
            "users": [],
            "students": [],
            "instructors": [],
            "courses": [],
            "assignments": [],
            "grades": [],
            "schedules": [],
        }
        self.load_data()

    def add_assignment(self, assignment_id, title, description, due_date, course_id):
        assignment = Assignment(assignment_id, title, description, due_date, course_id)
        self.data["assignments"].append(assignment.get_details())
        self.save_data()
        print(f"Assignment {title} added successfully!")

    def list_assignments(self, course_id=None):
        if course_id:
            assignments = [a for a in self.data["assignments"] if a["course_id"] == course_id]
        else:
            assignments = self.data["assignments"]

        if assignments:
            print("Assignments:")
            for assignment in assignments:
                print(f"- {assignment['title']} (ID: {assignment['assignment_id']})")
        else:
            print("No assignments found.")

    def add_schedule(self, course_id, start_date, end_date, class_time, days):
        schedule = Schedule(course_id, start_date, end_date, class_time, days)
        self.data["schedules"].append(schedule.get_details())
        self.save_data()
        print(f"Schedule for course {course_id} added successfully!")

    def list_schedules(self):
        if self.data["schedules"]:
            print("Course Schedules:")
            for schedule in self.data["schedules"]:
                print(
                    f"- Course ID: {schedule['course_id']}, Start: {schedule['start_date']}, "
                    f"End: {schedule['end_date']}, Time: {schedule['class_time']}, Days: {schedule['days']}"
                )
        else:
            print("No schedules found.")
```



```
def add_grade(self, student_id, course_id, assignment_id, grade_value):
    grade = Grade(student_id, course_id, assignment_id, grade_value)
    self.data["grades"].append(grade.get_details())
    self.save_data()
    print(f"Grade {grade_value} added for student {student_id} in course {course_id}.")

def list_grades(self, student_id=None, course_id=None):
    grades = self.data["grades"]
    if student_id:
        grades = [g for g in grades if g["student_id"] == student_id]
    if course_id:
        grades = [g for g in grades if g["course_id"] == course_id]

    if grades:
        print("Grades:")
        for grade in grades:
            print(
                f"- Student ID: {grade['student_id']}, Course ID: {grade['course_id']}, "
                f"Assignment ID: {grade['assignment_id']}, Grade: {grade['grade_value']}"
            )
    else:
        print("No grades found.")
```

main_menu()

The Student Menu provides an interactive interface for students to manage their activities within the system. It offers several options, including viewing enrolled courses, checking grades for courses, and viewing assignment grades. Students can also enroll in new courses by selecting from the list of available courses and update their schedules. The menu ensures seamless interaction by using class methods such as view_courses, view_grades, and enroll_course. Additionally, students can view their course schedules in detail and logout when done, exiting to the main menu. This design allows students to manage their academic activities conveniently.

The Instructor Menu is tailored for instructors to manage their teaching responsibilities. Instructors can view the courses they are teaching, assign grades to students for specific assignments, and add new courses to their teaching list. They can also create assignments for their courses, enhancing the structure and evaluation process of the courses. The menu utilizes methods such as view_courses, assign_grade, assign_course, and create_assignment to provide a streamlined experience. Once finished, instructors can logout to return to the main menu.

The Admin Menu enables administrators to manage all entities within the system. Admins can add new students, instructors, and courses by collecting necessary details and registering them in the

platform. They can also view lists of all students, instructors, and courses, ensuring they have full visibility and control over the system's data. Using methods like add_course, add_student, and list_courses, the admin menu simplifies the management process. Admins can logout at any time to return to the main menu.

The Main Menu serves as the central entry point of the system, offering users options to sign up, log in, or exit the application. New users can register by selecting the "Sign Up" option, providing their details, and being assigned roles such as student, instructor, or admin. Registered users can log in with their credentials, and based on their roles, they are directed to their respective menus for further interactions. The application also allows users to exit the system gracefully. This hierarchical menu structure ensures a seamless and role-specific user experience while maintaining clarity and functionality.

```python
def student_menu(admin, student_id):
    while True:
        print("\n--- Student Menu ---")
        print("1. View Courses")
        print("2. View Grades for Courses")
        print("3. View Assignment Grades")
        print("4. Enroll in a Course")
        print("5. View Schedule")
        print("6. Logout")

        choice = input("Select an option: ")
```

```python
def admin_menu(admin):
    while True:
        print("\n--- Admin Menu ---")
        print("1. Add Student")
        print("2. Add Instructor")
        print("3. Add Course")
        print("4. List Students")
        print("5. List Instructors")
        print("6. List Courses")
        print("7. Logout")
        choice = input("Select an option: ")
```

```python
def instructor_menu(admin, instructor_id):
    # Retrieve instructor data at the beginning of the menu
    instructor_data = next((i for i in admin.data["instructors"] if i["instructor_id"] == instructor_id), None)

    # Check if instructor data exists
    if not instructor_data:
        print(f"Error: No instructor found with ID {instructor_id}.")
        return  # Exit the function if no instructor is found

    # Create the instructor instance once
    instructor = Instructor(
        instructor_data["name"],
        instructor_data["phone"],
        instructor_data["address"],
        instructor_data["date_of_birth"],
        instructor_data["instructor_id"]
    )

    # Instructor menu options
    while True:
        print("\n--- Instructor Menu ---")
        print("1. View Courses")
        print("2. Assign Grade")
        print("3. Add Course to Teach")
        print("4. Create Assignment")
        print("5. Logout")
        choice = input("Select an option: ")
```

```python
def main():
    admin = PlatformAdmin()

    while True:
        print("\n--- Main Menu ---")
        print("1. Sign Up")
        print("2. Login")
        print("3. Exit")
        choice = input("Select an option: ")

        if choice == "1":
            name = input("Enter your name: ")
            email = input("Enter your email: ")
            phone = input("Enter phone number: ")
            address = input("Enter address: ")
            date_of_birth = input("Enter date of birth (YYYY-MM-DD): ")
            password = input("Enter password: ")
            role = input("Enter role (student/instructor/admin): ")
            if role not in ["student", "instructor", "admin"]:
                print("Invalid role. Please choose 'student', 'instructor', or 'admin'.")
                continue
            admin.sign_up(name, email, phone, address, date_of_birth, password, role)

        elif choice == "2":
            email = input("Enter email: ")
            password = input("Enter password: ")
            role = admin.login(email, password)
            if role == "admin":
                admin_menu(admin)
            elif role == "instructor":
                instructor_id = input("Enter your instructor ID: ")
                instructor_menu(admin, instructor_id)
            elif role == "student":
                student_id = input("Enter your student ID: ")
                student_menu(admin, student_id)
            else:
                print("Login failed.")

        elif choice == "3":
            print("Goodbye!")
            break


if __name__ == "__main__":
    main()
```