***University Management System – OOP Case Study***

### **Submitted by: *ANISH SEHRAWAT***

### **Roll Number: *24CSU017***

### **Course: FOCP-II**

## **1. Introduction**

The University Management System is a C++ project designed to simulate core university operations such as managing students, professors, courses, departments, enrollments, and grading. The system was built using key Object-Oriented Programming (OOP) concepts, including:

* Class design
* Encapsulation and data hiding
* Inheritance and polymorphism
* Aggregation and composition
* Exception handling

This project fulfills the requirements of Assignments 2, 3, and 4 by modeling real-world entities and integrating them into a single, menu-driven console application.

## **2. Class Design Overview**

### **2.1 Core Classes**

|  |  |
| --- | --- |
| **Class** | **Description** |
| Person | Base class with attributes like name, ID, age, contact. |
| Student | Derived from Person, with GPA, enrollment date, and program. |
| Graduate Student | Inherits from Student, adds research topic, advisor, and thesis title. |
| Professor | Derived from Person, with department and specialization. |
| Full Professor | Extends Professor includes years of service and research grants. |
| Course | Represents a course with code, title, credits, and assigned instructor. |
| Department | Holds professors; uses composition. |
| University | Main system manager; aggregates students, courses, and departments. |
| EnrollmentManager | Manages which students are enrolled in which courses. |
| GradeBook | Tracks and analyzes student grades. |

## **3. OOP Concepts Demonstrated**

### **🔹 Encapsulation**

* All class attributes are private or protected.
* Public setters validate data (e.g., GPA must be 0.0–4.0, age must be reasonable).
* Classes like EnrollmentManager and GradeBook encapsulate logic internally.

### **🔹 Inheritance**

* Student and Professor inherit from Person.
* GraduateStudent inherits from Student.
* FullProfessor inherits from Professor.

### **🔹 Polymorphism**

* displayDetails() and calculatePayment() are virtual in Person and overridden in subclasses to provide specific behavior.

### **🔹 Aggregation & Composition**

* Department *contains* professors (composition).
* Course *has* a professor as an instructor (aggregation).
* Student *enrolls in* multiple Courses (aggregation).
* University *manages* everything (composition + aggregation).

## **4. Exception Handling**

### **Custom Exception Hierarchy:**

* UniversitySystemException (base)
* EnrollmentException
* GradeException
* PaymentException

### **Usage:**

* Trying to add invalid grades or over-enroll a course throws meaningful errors.
* Payment calculation errors or invalid attributes (like empty names or negative credits) are handled gracefully.
* All exceptions are caught and displayed using try-catch blocks in main().

## **5. System Features Demonstrated**

### **Functionalities:**

* Creating professors, students, and courses
* Assigning professors to courses
* Enrolling students
* Adding and calculating grades
* Displaying student and professor details
* Calculating and printing payments
* Logging and handling invalid operations

## **6. Sample Output**

--- University Report ---  
Name: John, Age: 20, ID: 201, Contact: [john@uni.edu](mailto:john@uni.edu)Program: CS, GPA: 3.5, Enrolled: 2023-08  
Payment Due: $10000  
---  
Name: Jane, Age: 24, ID: 202, Contact: [jane@uni.edu](mailto:jane@uni.edu)Program: CS, GPA: 3.8, Enrolled: 2022-08  
Research: AI, Advisor: Dr. Alice, Thesis: Ethics in AI  
Payment Due: $12000  
---  
Avg Grade: 88.5

## **7. Code**

#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <algorithm>

#include <stdexcept>

using namespace std;

// -------------------- Exception Hierarchy --------------------

class UniversitySystemException : public exception {

string msg;

public:

UniversitySystemException(const string& m) : msg(m) {}

const char\* what() const noexcept override { return msg.c\_str(); }

};

class EnrollmentException : public UniversitySystemException {

public:

EnrollmentException(const string& m) : UniversitySystemException("Enrollment Error: " + m) {}

};

class GradeException : public UniversitySystemException {

public:

GradeException(const string& m) : UniversitySystemException("Grade Error: " + m) {}

};

class PaymentException : public UniversitySystemException {

public:

PaymentException(const string& m) : UniversitySystemException("Payment Error: " + m) {}

};

// -------------------- Core Classes --------------------

class Person {

protected:

string name, contactInfo;

int age, ID;

public:

Person(string n, int a, int id, string contact) {

if (n.empty()) throw UniversitySystemException("Name cannot be empty.");

if (a <= 0 || a > 120) throw UniversitySystemException("Invalid age.");

name = n; age = a; ID = id; contactInfo = contact;

}

virtual void displayDetails() {

cout << "Name: " << name << ", Age: " << age << ", ID: " << ID << ", Contact: " << contactInfo << endl;

}

virtual double calculatePayment() { return 0.0; }

virtual ~Person() {}

};

class Student : public Person {

protected:

string enrollmentDate, program;

double GPA;

public:

Student(string n, int a, int id, string contact, string enroll, string prog, double gpa)

: Person(n, a, id, contact), enrollmentDate(enroll), program(prog) {

setGPA(gpa);

}

void setGPA(double g) {

if (g < 0.0 || g > 4.0) throw UniversitySystemException("GPA must be between 0.0 and 4.0");

GPA = g;

}

double getGPA() const { return GPA; }

void displayDetails() override {

Person::displayDetails();

cout << "Program: " << program << ", GPA: " << GPA << ", Enrolled: " << enrollmentDate << endl;

}

double calculatePayment() override { return 10000.0; }

};

class GraduateStudent : public Student {

string researchTopic, advisor, thesisTitle;

public:

GraduateStudent(string n, int a, int id, string contact, string enroll, string prog, double gpa,

string topic, string adv, string thesis)

: Student(n, a, id, contact, enroll, prog, gpa),

researchTopic(topic), advisor(adv), thesisTitle(thesis) {}

void displayDetails() override {

Student::displayDetails();

cout << "Research: " << researchTopic << ", Advisor: " << advisor << ", Thesis: " << thesisTitle << endl;

}

double calculatePayment() override { return 12000.0; }

};

class Professor : public Person {

protected:

string department, specialization, hireDate;

public:

Professor(string n, int a, int id, string contact, string dept, string spec, string hDate)

: Person(n, a, id, contact), department(dept), specialization(spec), hireDate(hDate) {}

void displayDetails() override {

Person::displayDetails();

cout << "Department: " << department << ", Specialization: " << specialization << ", Hire Date: " << hireDate << endl;

}

double calculatePayment() override { return 50000.0; }

};

class FullProfessor : public Professor {

int yearsOfService;

double researchGrants;

public:

FullProfessor(string n, int a, int id, string contact, string dept, string spec, string hDate, int years, double grants)

: Professor(n, a, id, contact, dept, spec, hDate), yearsOfService(years), researchGrants(grants) {}

double calculatePayment() override {

return 80000.0 + 1500 \* yearsOfService + researchGrants;

}

};

// -------------------- Course, Department, University --------------------

class Course {

string code, title, description;

int credits;

Professor\* instructor;

public:

Course(string c, string t, int cr, string desc, Professor\* inst)

: code(c), title(t), credits(cr), description(desc), instructor(inst) {

if (cr <= 0) throw UniversitySystemException("Course credits must be positive.");

}

string getCode() const { return code; }

};

class Department {

string name, location;

double budget;

vector<Professor\*> professors;

public:

Department(string n, string loc, double b) : name(n), location(loc), budget(b) {}

void addProfessor(Professor\* p) {

professors.push\_back(p);

}

vector<Professor\*> getProfessors() {

return professors;

}

};

class University {

vector<Department\*> departments;

map<int, Student\*> students;

map<string, Course\*> courses;

public:

void addDepartment(Department\* d) {

departments.push\_back(d);

}

void addStudent(Student\* s) {

students[s->calculatePayment()] = s;

}

void addCourse(Course\* c) {

courses[c->getCode()] = c;

}

void generateReport() {

cout << "\n--- University Report ---" << endl;

for (map<int, Student\*>::iterator it = students.begin(); it != students.end(); ++it) {

Student\* s = it->second;

s->displayDetails();

cout << "Payment Due: $" << s->calculatePayment() << "\n---\n";

}

}

~University() {

for (map<int, Student\*>::iterator it = students.begin(); it != students.end(); ++it) delete it->second;

for (map<string, Course\*>::iterator it = courses.begin(); it != courses.end(); ++it) delete it->second;

for (size\_t i = 0; i < departments.size(); ++i) delete departments[i];

}

};

// -------------------- GradeBook and Enrollment --------------------

class GradeBook {

map<int, double> grades;

public:

void addGrade(int studentID, double grade) {

if (grade < 0 || grade > 100) throw GradeException("Invalid grade.");

grades[studentID] = grade;

}

double calculateAverageGrade() {

if (grades.empty()) return 0.0;

double total = 0;

for (map<int, double>::iterator it = grades.begin(); it != grades.end(); ++it)

total += it->second;

return total / grades.size();

}

};

class EnrollmentManager {

map<string, vector<int>> courseEnrollments;

public:

void enrollStudent(string courseCode, int studentID) {

if (courseEnrollments[courseCode].size() >= 50)

throw EnrollmentException("Course is full.");

courseEnrollments[courseCode].push\_back(studentID);

}

int getEnrollmentCount(string courseCode) {

return courseEnrollments[courseCode].size();

}

};

// -------------------- Main Test Program --------------------

int main() {

try {

University uni;

// Create Professors

FullProfessor\* prof1 = new FullProfessor("Dr. Alice", 55, 101, "[alice@uni.edu](mailto:alice@uni.edu)", "CS", "AI", "2001", 20, 20000);

Department\* csDept = new Department("Computer Science", "B1", 500000);

csDept->addProfessor(prof1);

uni.addDepartment(csDept);

// Create Course

Course\* course1 = new Course("CS101", "Intro to CS", 3, "Basics of computing", prof1);

uni.addCourse(course1);

// Add Students

Student\* s1 = new Student("John", 20, 201, "[john@uni.edu](mailto:john@uni.edu)", "2023-08", "CS", 3.5);

GraduateStudent\* s2 = new GraduateStudent("Jane", 24, 202, "[jane@uni.edu](mailto:jane@uni.edu)", "2022-08", "CS", 3.8, "AI", "Dr. Alice", "Ethics in AI");

uni.addStudent(s1);

uni.addStudent(s2);

// Enroll Students

EnrollmentManager em;

em.enrollStudent("CS101", 201);

em.enrollStudent("CS101", 202);

// Add Grades

GradeBook gb;

gb.addGrade(201, 85);

gb.addGrade(202, 92);

// Report

uni.generateReport();

cout << "Avg Grade: " << gb.calculateAverageGrade() << endl;

} catch (UniversitySystemException& ex) {

cerr << "System Error: " << ex.what() << endl;

} catch (exception& ex) {

cerr << "General Error: " << ex.what() << endl;

}

return 0;

}