Orange Digital Center

TECHIE KIT

UNIVERSITY SYSTEM

University Management System

- Prepared By:
  - Mohammed Khalaf Ismail
- Under supervision:
  - Eng. Abdel Sattar Ahmed (ITI Instructor)

AUGUST 10, 2023

CREATIVA SOHAG

Contents

1 Problem Definition:

Our problem is about to make a university organization system. We are tasked to design and implement a University Management System using Object-Oriented Programming (OOP) principles. The system should allow for the efficient management of courses, students, and instructors within the university. The system should be capable of performing various operations related to course registration, student enrollment, and instructor management.

The University Management System should consist of the following main entities and functionalities:

1.1 Courses:

Each course has a unique course code, title, description, and maximum capacity of students. Courses can be created, updated, and deleted. The system should keep track of the enrolled students and the assigned instructor for each course.

1.2 Students:

Each student has a unique student ID, name, date of birth, and contact information. Students can enroll in courses, drop courses, and view their enrolled courses. The system should prevent enrolling in courses that have reached maximum capacity.

1.3 Instructors:

Each instructor has a unique instructor ID, name, specialization, and contact information. Instructors can be assigned to courses and can be changed if needed. The system should prevent assigning an instructor to multiple courses at the same time.

1.1 Functionalities:

Enroll a student in a course (if there's available space). Drop a course for a student.

Assign an instructor to a course. View the list of all courses, their details, enrolled students, and assigned instructors. View the list of all students, their details, and enrolled courses. View the list of all instructors and their assigned courses.

Our program should demonstrate proper encapsulation, inheritance, and polymorphism through the use of classes and objects.

When developing your solution, prioritize modularity, code reusability, and clarity. Implement robust error handling and user input validation to guarantee the system's reliability.

In the future, provide an intuitive user interface that allows users to interact with the system effectively. While a graphical user interface (GUI) is not mandatory, the system should be user-friendly and intuitive.

2 Solution:

Creating a comprehensive University Management System using Object-Oriented Programming (OOP) involves designing and implementing classes, methods, and interactions that reflect the problem description. Below is a high-level solution description for the University Management System:

1. Class Design:

   1.1 Person class:
   - Attributes: name, age, id
   - Methods: pure abstract method =➔ role ()

   1.2 Student Class:
   - Attributes: name, age, id, GPA.
   - Methods: role ().

   1.3 Instructors class:
   - Attributes: name, age, id.
   - Methods: role ().

   1.4 Employee class:
   - Attributes: name, age, salary.
   - Methods: role ()

   1.5 Manager class (singleton class):
   - Attributes: name, age, salary.
   - Methods:
     1) role ()
     2) setSalary (int Salary)
     3) getSalary ()

   1.6 Course Class:
   - Attributes: title, course ID.
   - Methods:
     1) Details ().

   1.7 Department class:
   - Attributes: Vector of course list.
   - Methods:
     1) Details ()
     2) addCourse ().

   1.8 Faculty Class:
   - Attributes: title
   - Methods:
     1) Details ()

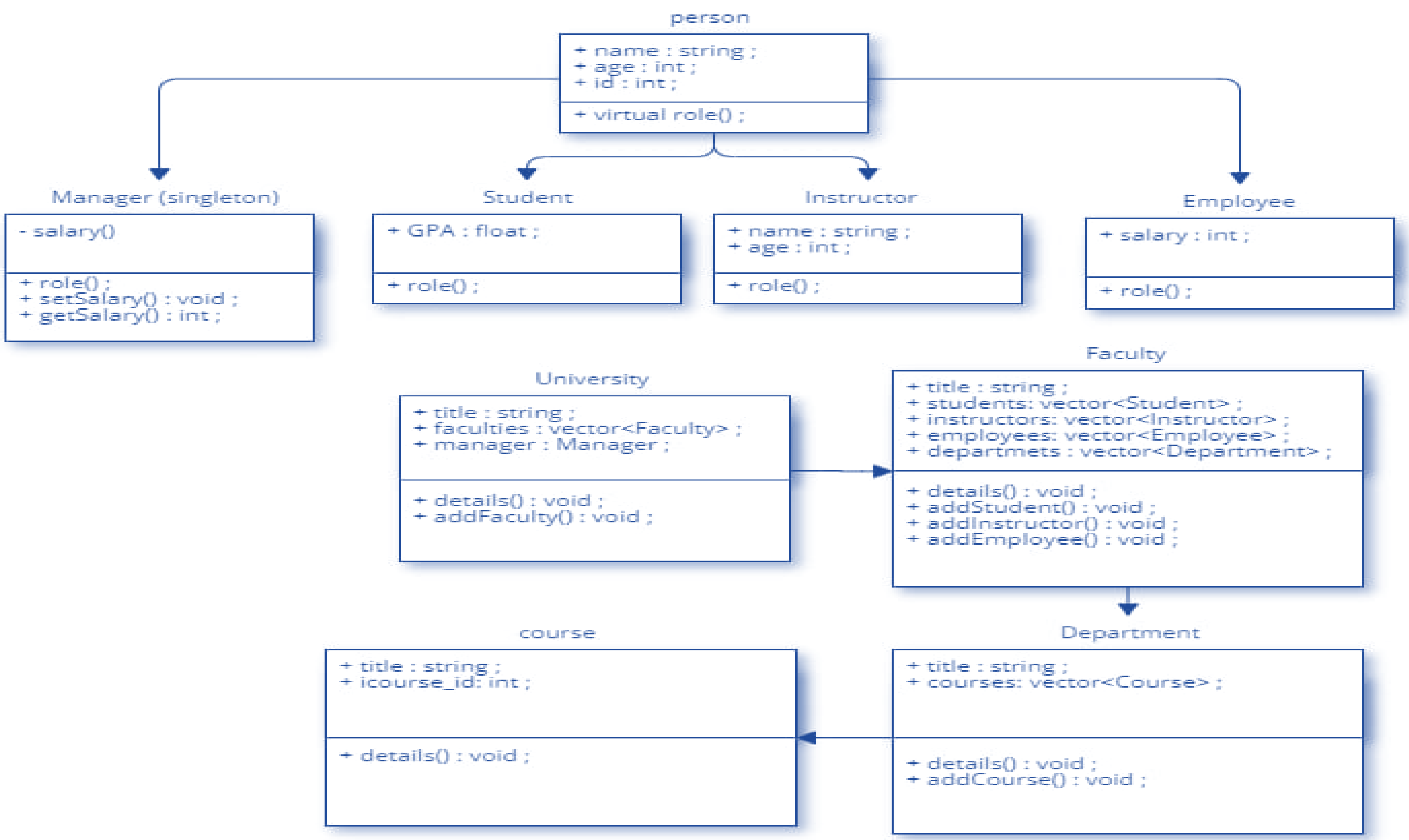
2) addStudents ().
3) addInstructors ().
4) addEmployee ().

1.9 University Class:

- Attributes: title
- Methods:

1) Details ()
2) addFaculty ().
3)

3 UML:



4 Steps:

Step 1: Define Person Class

Create a base class Person with attributes: name, age, and id.

Implement a constructor to initialize these attributes.

Declare a pure virtual method role () to be overridden by subclasses.

Implement a destructor to clean up resources.

```cpp
class Person
{
private:
public:
    string name;
    int age;
    int id;
    Person();
    Person(string n, int a, int id);
    virtual void role() = 0;
    ~Person();
};

Person::Person()
{
}
Person::Person(string n, int a, int ID) : name{n}, age{a}, id{ID}
{
}
Person::~Person()
{
    cout<<"Bye !\n";
}
```

Step 2: Implement Derived Classes

Create derived classes Student, Instructor, Employee, and Manager from the Person base class.

Add specific attributes for each derived class (GPA for Student, salary for Employee, etc.).

Override the role() method in each derived class.

```cpp
class Student : public Person
{
private:
    /* data */
public:
    float GPA;
    Student();
    Student(string n, int a, int ID, float gpa);
    void role()
    {
    }
    ~Student();
};

Student::Student() ···
Student::Student(string n, int a, int ID, float gpa) ···

Student::~Student() ···

class Instructor : public Person ···

Instructor::Instructor() ···
Instructor::Instructor(string n, int a, int ID) ···
Instructor::~Instructor() ···
class Employee : public Person
{
private:
public:
    int salary;
    Employee();
    Employee(string n, int a, int ID, int s);
    ~Employee();
    void role()
    {
    }
};

Employee::Employee()
{
}
Employee::Employee(string n, int a, int ID, int s)
{
    name = n;
    age = a;
    id = ID;
    salary = s;
}

Employee::~Employee()
{
}
```

Step 3: Implement Singleton Pattern for Manager

Implement the Singleton pattern for the Manager class to ensure only one instance is created.

Add private constructor and a static method getInstance () to retrieve the instance.

Implement methods to set and get the salary for the manager.

```cpp
class Manager : public Person
{
private:
    int salary;
    static Manager *instance; // The single instance

    // Private constructor to prevent instantiation
    Manager() {}

public:
    void role()
    {
    }
    void setSalary(int Salary);

    int getSalary();

    // Method to get the instance
    static Manager *getInstance()
    {
        if (!instance)
        {
            instance = new Manager();
        }
        return instance;
    }
```

```cpp
    void showMessage()
    {
        std::cout << "Hello from Singleton!" << std::endl;
    }
};
// Initialize the static instance to nullptr
Manager *Manager::instance = nullptr;

void Manager::setSalary(int Salary)
{
    salary = Salary;
}
int Manager::getSalary()
{
    return salary;
}
```

Step 4: Implement Course Class

Create a Course class with attributes title and courseID.

Implement a constructor to initialize these attributes.

Add a details () method to display course details.

```cpp
class Course
{
private:
    /* data */
public:
    string title;
    int courseID;
    void details();
    Course();
    Course(string t, int courseid);
    ~Course();
};

> Course::Course() ···
> Course::Course(string t, int courseid) ···

> Course::~Course() ···
> void Course::details() ···
```

Step 5: Implement Faculty Class

Create a Faculty class to manage students, instructors, and employees. Include vectors for each entity and methods to add details for each.

```cpp
class Faculty
{
private:
    /* data */
public:
    string title;
    vector<Student> students{};
    vector<Instructor> instructors{};
    vector<Employee> employees{};
    vector<Department> depts{};
    void details();
    void addStudents();
    void addInstructor();
    void addEmployee();

    Faculty();
    Faculty(string t);
    ~Faculty();
};

Faculty::Faculty() ···
Faculty::Faculty(string t) ···

Faculty::~Faculty() ···
void Faculty::details() ···
void Faculty::addStudents() ···
void Faculty::addInstructor() ···
void Faculty::addEmployee() ···
```

Step 5: Implement Department Class

Implement a Department class to manage courses within a department.

Include a vector of Course objects and methods to add and display courses.

```
class Department
{
private:
    /* data */
public:
    void details();
    void addCourse();
    vector<Course> courses{};
    Department();
    ~Department();
};

> Department::~Department() ···
> void Department::details() ···
void Department::addCourse()
{
    string title;
    int courseid;
    cout << "Enter Course name: ";
    cin >> title;
    cout << "Enter Course ID: ";
    cin >> courseid;

    // Create a new Course object
    Course newCourse(title, courseid);

    // Add the new Course to the vector
    courses.push_back(newCourse);

    // Write Course data to a text file
    ofstream outFile("courses.txt", ios::app);
    if (outFile.is_open())
    {
        outFile << "Course Name: " << title << "            "
                << "Coures ID: " << courseid << endl;
        outFile.close();
        cout << "Course Name added and data written to file." << endl;
    }
    else
    {
        cout << "Error opening file." << endl;
    }
}
```

Step 7: Implement University Class:

Implement a University class to manage faculties within a university.

Include a vector of Faculty objects and methods to add and display faculties.

```cpp
class University
{
private:
public:
    string title;
    vector<Faculty> faculties{};
    void details();
    void addFaculty();
    University()
    {
    }
    ~University();
};

University::~University() ...
void University::details() ...
void University::addFaculty()
{
    string title;
    cout << "Enter Faculty name: ";
    getline(cin, title);

    // Create a new faculty object
    Faculty newFaculty(title);

    // Add the new faculty to the vector
    faculties.push_back(newFaculty);

    // Write instructor data to a text file
    ofstream outFile("faculties.txt", ios::app);
    if (outFile.is_open())
    {
        outFile << title << endl;
```

Step 8: Data Persistence

Use file I/O operations to store data for students, instructors, employees, courses, faculties, etc.

Implement methods to read and write data to text files for each class.

Step 9: Display Details

Implement methods in each class to display relevant details, such as course information, faculty details, etc.

Step 10: Testing and Refining

Test the program by creating instances of various classes, adding data, and interacting with the system. Debug and refine the program as needed to ensure correct functionality and error handling.

5 Int main:

In the main function (User Interaction in Main), create instances of various classes to interact with the system.

Allow users to input data for students, instructors, employees, courses, and faculties.

Use methods from different classes to manage the university system.

```
int main()
{
    // Create instances of various classes to interact with the system.
    Faculty F1;
    F1.addStudents();
    F1.addInstructor();

    University U1;
    U1.addFaculty();

    Department D;
    D.addCourse();

     // Get the Singleton instance
    Manager *man = Manager::getInstance();
    man->setSalary(7000);
    cout << man->getSalary();

    return 0;
}
```

6 Output:

In the main function, you have created instances of various classes and utilized their methods to interact with the University Management System. Let's break down the output of the main function step by step:

1. Faculty F1; - You create an instance of the Faculty class named `F1`.
2. F1.addStudents(); - You call the addStudents() method on the F1 instance of the Faculty class. This prompts you to input details for a student, such as name, age, ID, and GPA. Once you provide this information, it adds the student to the “students” vector within the “Faculty” object and writes the student data to a file named "students.txt".

3. F1.addInstructor(); - You call the “addInstructor()” method on the “F1” instance of the “Faculty” class. Similar to the previous step, you input details for an instructor, including name, age, and ID. The instructor information is added to the `instructors` vector and written to a file named "instructors.txt".
4. University U1; - You create an instance of the “University” class named “U1”.
5. U1.addFaculty(); - You call the “addFaculty()” method on the “U1” instance of the “University” class. This prompts you to input a faculty name and then adds this faculty to the `faculties` vector within the `University` object. The faculty information is written to a file named "faculties.txt".
6. “Department D;” - You create an instance of the “Department class named D.
7. “D.addCourse();” - You call the “addCourse()” method on the D instance of the ”Department” class. This prompts you to input course details, including the course name and ID. The new course is added to the “courses” vector within the “Department”object, and its information is written to a file named "courses.txt".
8. “Manager *man = Manager::getInstance*();”* - You create a pointer to the “Manager” class instance using the Singleton pattern. This ensures that there is only one instance of the “Manager” class throughout the program.
9. `man->setSalary(7000);` - You set the salary of the manager to 7000 using the setSalary() method.
10. cout << "The new value of salary= ";- You output a message indicating that you're about to print the new salary value.
11. cout << man->getSalary(); - You retrieve and print the salary of the manager using the “getSalary()” method.
12. return 0; - The program ends, and control returns to the operating system. The output from the previous steps will be displayed in the console, showing the interactions with the different classes and their methods.

The overall output will include prompts for user input, as well as messages indicating the addition of students, instructors, faculties, courses, and the updated salary of the manager. The program demonstrates how the classes and their methods work together to manage various aspects of the university system.

7 Conclusion:

The University Management System project is a comprehensive application that utilizes Object-Oriented Programming (OOP) principles to simulate and manage various aspects of a university environment. The project focuses on designing and implementing classes for different entities such as students, instructors, employees, courses, departments, faculties, and a manager. These classes are organized hierarchically and interact in a structured manner to create a cohesive system.

Key features of the project include:

1. Class Hierarchy: The project demonstrates a well-defined class hierarchy with a base class `Person` and several derived classes such as "Student`, "Instructor", "Employee", and "Manager". Each class encapsulates specific attributes and behaviors related to its role within the university.
2. Abstraction and Polymorphism: The use of pure virtual methods and inheritance allows for abstraction and polymorphism. The " role() " method, declared as a pure virtual method in the "Person" class and overridden in the derived classes, showcases the concept of polymorphism.
3. Singleton Design Pattern: The implementation of the Singleton design pattern ensures that there is only one instance of the `Manager` class, which controls and manages a specific aspect of the system (in this case, the manager's salary).
4. User Interaction and Data Persistence: The project enables user interaction through console input and output, allowing users to input data for students, instructors, faculties, courses, and departments. The system also demonstrates data persistence by writing relevant information to text files for future reference.
5. Modularity and Reusability: The modular design of the classes allows for easy expansion and modification. New features, attributes, and behaviors can be added to the system without significantly impacting the existing codebase.
6. File I/O Operations: The project employs file input/output operations to store and retrieve data, enhancing the realism of the simulation by maintaining records over different sessions.
7. Demonstration of OOP Concepts: The project effectively illustrates OOP concepts such as encapsulation, inheritance, abstraction, and polymorphism. It showcases how these concepts can be used to model and manage complex real-world scenarios.

In conclusion, the University Management System project provides an insightful and practical example of applying OOP principles to create a software system that simulates various aspects of a university. It highlights the versatility and flexibility of OOP for modeling and managing diverse entities and interactions within a cohesive structure. This project serves as a valuable learning experience for understanding software design, class relationships, and practical implementation of OOP concepts.