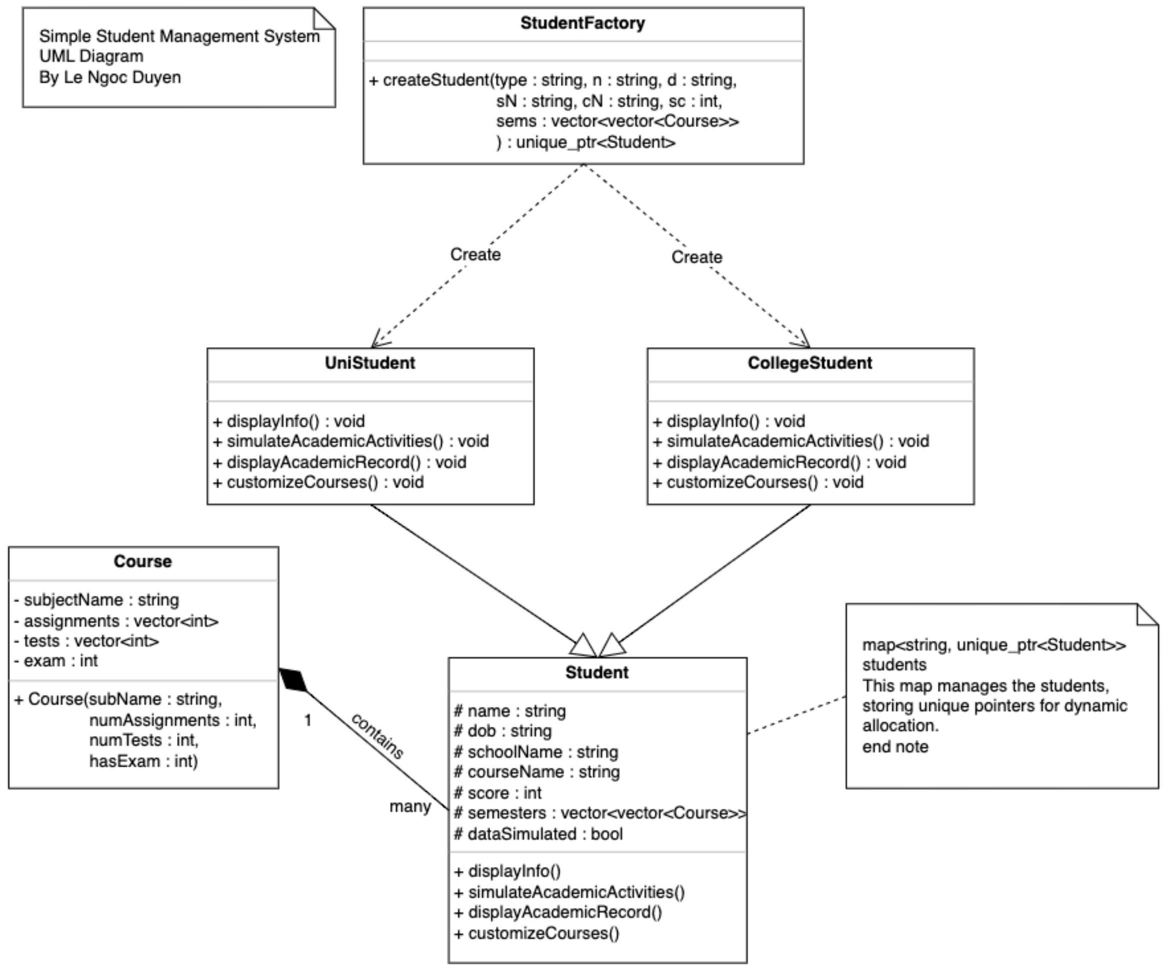
ADVANCED PROGRAMMING ASSIGNMENT 3 REPORT

1. UML DIAGRAM


Simple Student Management System
UML Diagram
By Le Ngoc Duyen
StudentFactory
+ createStudent(type : string, n : string, d : string, sN : string, cN : string, sc : int, sems : vector<vector<Course>>) : unique_ptr<Student>
Create
Create
UniStudent
+ displayInfo() : void
+ simulateAcademicActivities() : void
+ displayAcademicRecord() : void
+ customizeCourses() : void
CollegeStudent
+ displayInfo() : void
+ simulateAcademicActivities() : void
+ displayAcademicRecord() : void
+ customizeCourses() : void
Course
- subjectName : string
- assignments : vector<int>
- tests : vector<int>
- exam : int
+ Course(subName : string, numAssignments : int, numTests : int, hasExam : int)
1
contains
many
Student
# name : string
# dob : string
# schoolName : string
# courseName : string
# score : int
# semesters : vector<vector<Course>>
# dataSimulated : bool
+ displayInfo()
+ simulateAcademicActivities()
+ displayAcademicRecord()
+ customizeCourses()
map<string, unique_ptr<Student>> students
This map manages the students, storing unique pointers for dynamic allocation.
end note

2. EXPLANATIONS FOR IMPROVEMENTS

2.1. Part 1

In the first part of Assignment 3, the main objective was to improve the student management program from Assignment 2 by utilizing specific C++ data structures. Here's a detailed changes and why they enhance the program:

C++ Data Structures Used

1. **Vector**: Vectors are used extensively to handle dynamic arrays that can change in size.
    - **vector<vector<Course>> semesters;** within each student class allows each student to have multiple semesters, each containing a dynamic list of courses.

This reflects the real-world scenario where the number of courses per semester can vary.

- **vector<int> assignments;** and **vector<int> tests;** in the **Course** struct represent lists of assignment scores and test scores, respectively. Using vectors here provides the flexibility to handle a varying number of assignments and tests, especially since university and college students have different academic loads.

2. **Map**: A map is used to store student objects with a unique ID as the key:
    - **map<string, Student*> students;** allows for quick lookups, insertions, and deletions of students based on their student ID. This is a significant improvement over a simple vector, as it avoids the need to iterate over the entire collection to find a student, thus enhancing performance for operations involving large numbers of students. Maps also automatically keep the keys in sorted order, which can be beneficial for tasks that require ordered data.

2.2. Part 2

Incorporating the Factory design pattern in Part 2 of the student management system provides improvements over the initial implementation provided in Part 1.

The main alterations required to adapt the Factory design pattern:

1. **Creation of a Factory Class:** a StudentFactory class responsible for generating Student objects.

   This class will contain a method to decide whether to return a UniStudent or CollegeStudent based on input parameters.

2. **Modifying the addStudent Function**

   The addStudent function will need to be modified to use this factory for creating student objects. This centralizes object creation and simplifies the addStudent function.

These changes decentralize the responsibility for creating Student objects from the main part and move it into a factory class. This not only adheres to the Factory design pattern but also makes the code cleaner and more maintainable by separating concerns and centralizing the creation logic in one place.

2.2. Part 3

The enhancements made in the provided code for Part 3 involve modern C++ features that improve the robustness, safety, and maintainability of the student management system. Let's detail the specific changes and their benefits:

1. Smart Pointers

   Changes Made:

   Replaced raw pointers with std::unique_ptr in the students map declaration and creation process.

   From:

   map<string, Student*> students;

   To:

   map<string, unique_ptr<Student>> students;

   From:

   students[id] = new UniStudent(name, dob, schoolName, courseName, score, semesters);

   To:

   students[id] = StudentFactory::createStudent(type, name, dob, schoolName, courseName, score, semesters);

   Where createStudent now returns a unique_ptr<Student>.

   Improvement:

- **Memory Management**: Smart pointers automatically manage the lifetime of objects they point to. Using **unique_ptr** ensures that student objects are properly deallocated when they are removed from the map or when the map is destroyed. This avoids memory leaks which are a common problem with raw pointers.

- **Safety**: **unique_ptr** enforces unique ownership semantics, preventing accidental copying of the pointer, which could lead to double deletion issues.

2. Range For Loop

Changes Made:

Used range-based for loops for iterating over containers.

From:

```
for (map<string, Student*>::iterator it = students.begin(); it != students.end(); ++it) {
   it->second->displayInfo();
   it->second->displayAcademicRecord();
}
```

To:

```
for (const auto& pair : students) {
   pair.second->displayInfo();
   pair.second->displayAcademicRecord();
}
```

Improvement:

- **Readability and Safety:** Range-based for loops make the code more readable and less prone to errors such as off-by-one or incorrect iterator usage.
- **Less Code:** Eliminates the need for explicit iterator handling, making the code cleaner and easier to understand.

3. Auto

Changes Made:

Used the auto keyword to simplify type declarations, especially when dealing with iterators or complex types.

From:

```
map<string, Student*>::iterator it = students.begin();
```

To:

```
auto it = students.begin();
```

Improvement:

- **Type Inference**: auto automatically deduces the type of a variable from its initializer. This is particularly handy with complex types like iterators, reducing the verbosity of the code and minimizing the risk of type mismatches.