

VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Academic year: 2023 – 2024 – Semester 232



ASSIGNMENT 3 REPORT
CO2039 – ADVANCED PROGRAMMING
STUDENT MANAGEMENT PROGRAM

Instructors: Trương Tuấn Anh, HCMUT
Phan Duy Hân, Monash University

Student: Trần Quốc Bảo Long – 2252453
Class: CC02

Ho Chi Minh City, April 2024

CONTENTS

I.	Introduction	3
II.	Program logic	3
III.	UML diagram	9
IV.	Improvements	10
V.	Conclusion.....	12
VI.	References	12

I. INTRODUCTION

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. OOP is easier to execute and maintain, and has several advantages over procedural and functional programming.

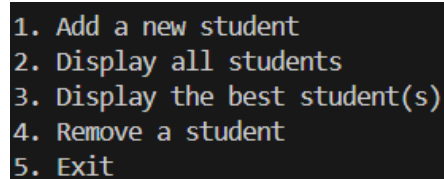
In Semester 232, for Assignment 3 in Advanced Programming, I use OOP to implement a simple student management program for tertiary students, managing college and university students at once. Users can enter the name of students and their information, and the program can generate scores randomly and assign to each student.

II. PROGRAM LOGIC

For this simple student management program, there are four functions:

- Adding students: a user will input a student's information and then choose whether he/she is a college or a university student. The scores are then randomly generated for each semester.
- Removing students: a user can remove a student from the list by entering his/her name.
- Displaying all students: the program prints out all students in the list, separated by university and college.
- Displaying best students: the program will calculate and find the best students out of university students and college students, and print out those. There can be two or more students with the highest score for either section.

All the functions are available under one while loop.



```
1. Add a new student
2. Display all students
3. Display the best student(s)
4. Remove a student
5. Exit
```

Figure 1. Main menu

Entering “5” exits the while loop, deletes the dynamic array and ends the program.

1. Adding students

```
void addStudent(vector<unique_ptr<Student>>& students) {...}
```

The program asks user to enter a student's information: name, date of birth, name of school they're enrolled in (either college or university), their course and finally asks whether they're a university student or a college student.

```
Enter name: Jane
Enter date of birth: 1/1/2000
Enter school name: HCMUT
Enter course name: Advanced_Programming
Enter course type (1 for University, 2 for College): 1
```

Figure 2. Entering information for addStudent function

Using the Factory design pattern and smart pointers, the student is assigned into either University or College section based on the courseType variable input.

```
class StudentFactory { // Upgrade using the Factory design pattern
public:
    static unique_ptr<Student> createStudent(const string& name, const string&
dob, const string& school, const string& course, const int& courseType) {
        if (courseType == 1) { // if University
            return make_unique<UniStudent>(name, dob, school, course,
courseType);
        } else if (courseType == 2) { // if College
            return make_unique<CollegeStudent>(name, dob, school, course,
courseType);
        } else {
            return nullptr;
        }
    }
};
```

The student is then pushed into the students vector. Functions doAssignment, takeTest and takeExam, overridden from Student class, will randomly assign scores for the student. Scores can be anywhere from 0 to 100.

- For university students, they have eight semesters, each semester has three assignments, two tests and one exam.
- For college students, they have four semesters, each semester has one assignment, one test and one exam.

The program prints out “Student added successfully”. If courseType differs from 1 or 2, the program prints out “Invalid choice” and resets the process.

2. Removing students

```
void removeStudent(vector<unique_ptr<Student>>& students) {...}
```

The program asks user to enter a student’s name, then find for that name on the list using the linear search algorithm.

```
Enter a student to remove: Jane
```

The program prints out “Student removed successfully.” if said name is found. If not, the program prints out “Student not found” and resets the process.

3. Displaying all students

```
void displayStudents(const vector<unique_ptr<Student>>& students) {...}
```

The program loops through the list of all students using range-based for loops, and display their information, including names and scores.

```
LIST OF STUDENTS
Name: Jack
Date of birth: 2/9/2003
School name: HCMUT
Course name: Programming
Grades:
Semester 1:
Assignments: 41 85 72
Tests: 90 60
Exam: 19

Semester 2:
Assignments: 38 80 69
Tests: 49 31
Exam: 76

Semester 3:
Assignments: 65 68 96
Tests: 23 99
Exam: 12

Semester 4:
Assignments: 22 49 67
Tests: 94 11
Exam: 33

Semester 5:
Assignments: 51 61 63
Tests: 25 24
Exam: 99

Semester 6:
Assignments: 87 66 24
Tests: 51 15
Exam: 18

Semester 7:
Assignments: 80 83 71
Tests: 13 39
Exam: 92
```

```
Semester 8:  
Assignments: 60 64 52  
Tests: 67 97  
Exam: 35
```

Figure 3. Displaying a student's information, and his/her scores

4. Displaying the best students

```
void displayBestStudents(const vector<unique_ptr<Student>>& students) {...}
```

The program loops through the list of students again, like above, and find the best students based on the total scores, using the `getTotalScore` function.

This function however, separates students into the College and University sections instead of combining both of them.

Figure 4 shows an example of the best students being displayed.

BEST STUDENT(S) AND THEIR SCORE(S)

UNIVERSITY

Name: Jack

Date of birth: 2/9/2003

School name: HCMUT

Course name: Programming

Grades:

Semester 1:

Assignments: 41 85 72

Tests: 90 60

Exam: 19

Semester 2:

Assignments: 38 80 69

Tests: 49 31

Exam: 76

Semester 3:

Assignments: 65 68 96

Tests: 23 99

Exam: 12

Semester 4:

Assignments: 22 49 67

Tests: 94 11

Exam: 33

Semester 5:

Assignments: 51 61 63

Tests: 25 24

Exam: 99

Semester 6:

Assignments: 87 66 24

Tests: 51 15

Exam: 18

```
Semester 7:  
Assignments: 80 83 71  
Tests: 13 39  
Exam: 92  
  
Semester 8:  
Assignments: 60 64 52  
Tests: 67 97  
Exam: 35  
  
COLLEGE  
Name: Jill  
Date of birth: 31/12/2004  
School name: HCMUT  
Course name: Programming  
Grades:  
Semester 1:  
Assignment: 74  
Test: 39  
Exam: 37  
  
Semester 2:  
Assignment: 0  
Test: 33  
Exam: 45  
  
Semester 3:  
Assignment: 95  
Test: 39  
Exam: 57  
  
Semester 4:  
Assignment: 71  
Test: 32  
Exam: 71
```

Figure 4. Displaying best students, both university and college

III. UML DIAGRAM

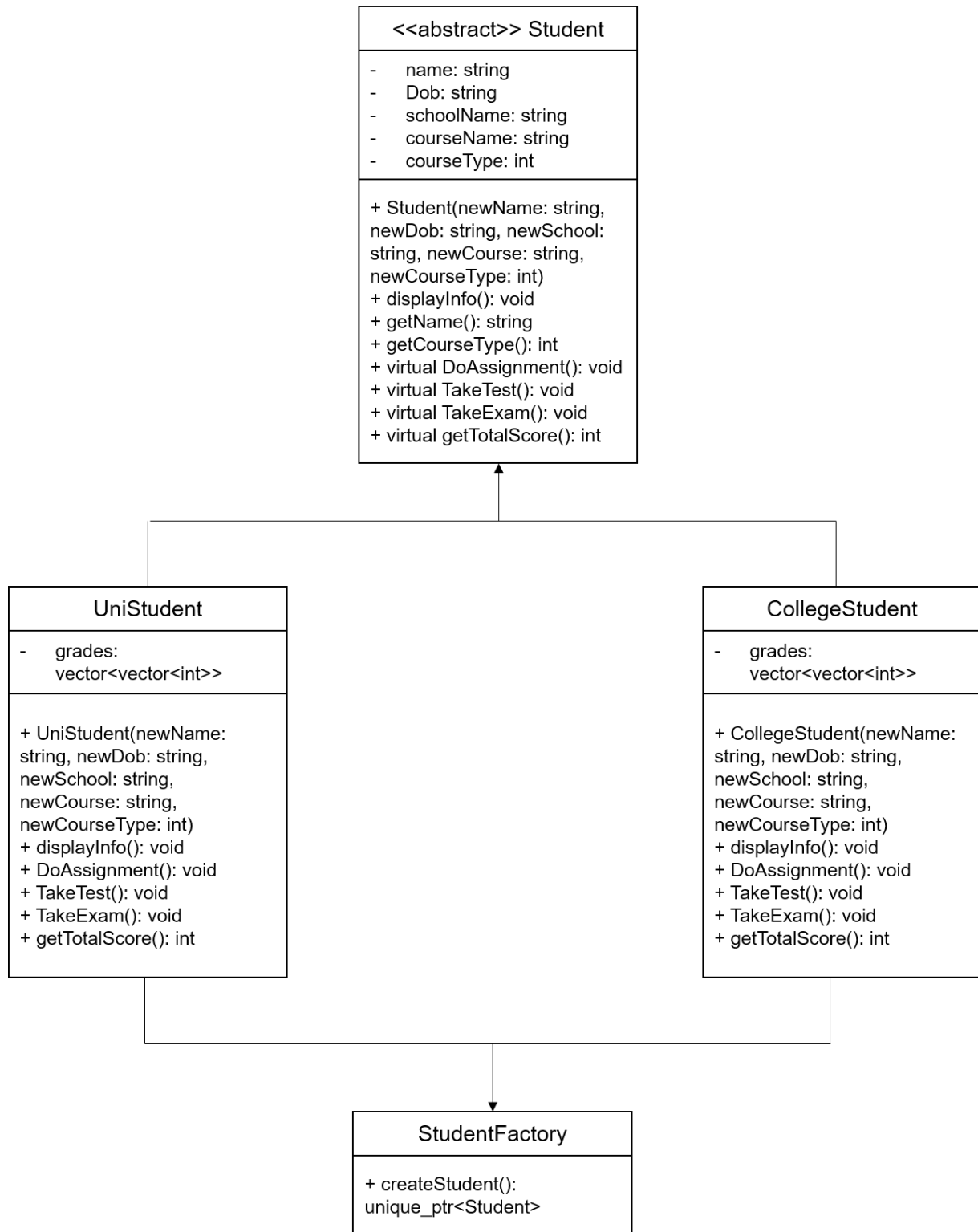


Figure 5. UML diagram for the student management program

In this diagram:

- Student is an abstract class with attributes name, dob, schoolName, courseName, and courseType. It also defines virtual methods DoAssignment(), TakeTest(), TakeExam(), and getTotalScore().
- UniStudent and CollegeStudent inherit from Student and implement their respective methods.
- Both UniStudent and CollegeStudent have the additional attribute grades, representing the grades of the students.
- StudentFactory is a class responsible for creating instances of Student subclasses based on the courseType.

IV. IMPROVEMENTS

Over three parts of this assignment, I've improved the program from Assignment 2 by using several methods, such as using data structures and modern design patterns. Here's how they've improved the efficiency of the program.

1. Part 1

As of this point, all students are still stored in the students dynamic array, requiring the variable numStudents. For Part 1, I've used vectors to store the score values for each student:

- Constructor for university students in class UniStudent. The grades vector is a two-dimensional vector, with the size of 8 x 6.

```
UniStudent(const string& newName, const string& newDob, const string&
newSchool, const string& newCourse, const int& newCourseType)
    : Student(newName, newDob, newSchool, newCourse, newCourseType),
  grades(8, vector<int>(6, 0)) {}
```

- Constructor for college students in class CollegeStudent. The grades vector is also a two-dimensional vector, with the size of 4 x 3.

```
CollegeStudent(const string& newName, const string& newDob, const string&
newSchool, const string& newCourse, const int& newCourseType)
    : Student(newName, newDob, newSchool, newCourse, newCourseType),
  grades(4, vector<int>(3, 0)) {}
```

This makes the score values for each student easier to maintain, as they handle memory allocation and deallocation internally. This also prevents segmentation faults and memory leaks like dynamic arrays.

2. Part 2

In Part 1, students are assigned directly by the `courseType` variable in the `addStudents` function. However, for Part 2, using the Factory design pattern, I've created a `StudentFactory` class for assigning students into either University or College sections:

```
class StudentFactory { // Upgrade using the Factory design pattern
public:
    static unique_ptr<Student> createStudent(const string& name, const string&
dob, const string& school, const string& course, const int& courseType) {
        if (courseType == 1) { // if University
            return make_unique<UniStudent>(name, dob, school, course,
courseType);
        } else if (courseType == 2) { // if College
            return make_unique<CollegeStudent>(name, dob, school, course,
courseType);
        } else {
            return nullptr;
        }
    }
};
```

The process of adding students in the `addStudents` function become:

```
auto newStudent = StudentFactory::createStudent(name, dob, schoolName,
courseName, courseType);
```

By using the Factory design pattern, the `StudentFactory` class provides the method `createStudent` that abstracts the creation process of Student objects. The factory's clients don't need to know which type of students to process – they simply call the `createStudent` method with the required parameters and the factory handles the object installation.

Overall, this centralizes the creation logic in one place, making it easier to manage and modify. It also adheres to the single responsibility principle by separating object creation from other concerns.

3. Part 3

For Part 3, I've used range-based for loops as well as smart pointers to improve the program from Part 2. As range-based for loops doesn't work really well with dynamic arrays, I've replaced the dynamic array with pointers with a vector of unique pointers to manage the list of students. As of this point, the `numStudents` variable is no longer needed.

Both of these methods have greatly increased the efficiency of my program. The first ones being the simplicity of syntax for iterating over elements in the Student list. The loops become more concise and easier to read, reducing the likelihood of errors and improving maintainability.

With smart pointers, I no longer have to manually allocate and deallocate memory using `new` and `delete` every time a new student is added. This eliminates the risk of forgetting to deallocate memory or releasing memory prematurely.

V. CONCLUSION

In conclusion, the implementation of a simple student management program using C++ OOP has been a challenging, yet rewarding experience. The application of OOP's principles in the implementation of the program has proven to be highly effective. The encapsulation of data and methods within the classes for College and University students, for instance, has enhanced the organization and security of the program.

The use of inheritance allowed us to create more specific types of students without having to rewrite the source code, thereby promoting code reusability and reducing redundancy. Polymorphism enabled us to interact with different object types through a unified interface, which simplified the code and made it more flexible to changes.

Overall, this project has demonstrated the power and versatility of OOP in creating robust, scalable, and maintainable software. It underscores the importance of OOP in modern software development and its relevance in building complex systems.

VI. REFERENCES

1. GeeksForGeeks. *Object-Oriented Programming in C++*.
URL: <https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/>
2. GeeksForGeeks. *The C++ Standard Template Library (STL)*.
URL: <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>
3. GeeksForGeeks. *Modern C++ Design Patterns Tutorial*.
URL: <https://www.geeksforgeeks.org/modern-c-design-patterns-tutorial/>