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

Advanced Programming (CO2039)

Assignment 3

Applying Modern C++ Features & Design Patterns in Student Management Program

---

Instructors: Phan Duy Hãn
Trương Tuấn Anh
Student: Hồ Khánh Nam – 2252500

HO CHI MINH CITY, April 2024

Contents

1 Introduction

In the realm of programming, Design Pattern and Object-oriented programming, or OOP, are the fundamental concepts that stand as pillars of modern development methodologies. These concepts not only shape the way we structure and organize our code but also significantly influence the scalability, maintainability, and efficiency of software systems.

The Object-oriented programming is a computer programming model that organizes software design around the data, or objects. An object in OOP can be defined as a data field that has unique attributes and behaviours. Moreover, these objects can interact with each other through well-defined interfaces, enabling modular, reusable, and scalable software development. In terms of Design Pattern, it is repeatable solution to a commonly occuring problem in software design. It represents a blueprint that encapsulates best practices and design principles for solving specific design challenge. The Design Patterns provide a structured approach to solving problems, promoting code reuseability, maintainability, and scalability.

In this assignment, I will improve my Student Management program code from the previous assignments by using the application of OOP and Design patterns. Besides, I also utilize some features that are available in C++ standard library to make the code more efficient and readable.

2 Class Diagram for the Student Management program

2.1 Class Diagram Overview

The overview of all classes used to making the Student Management Program is shown below:

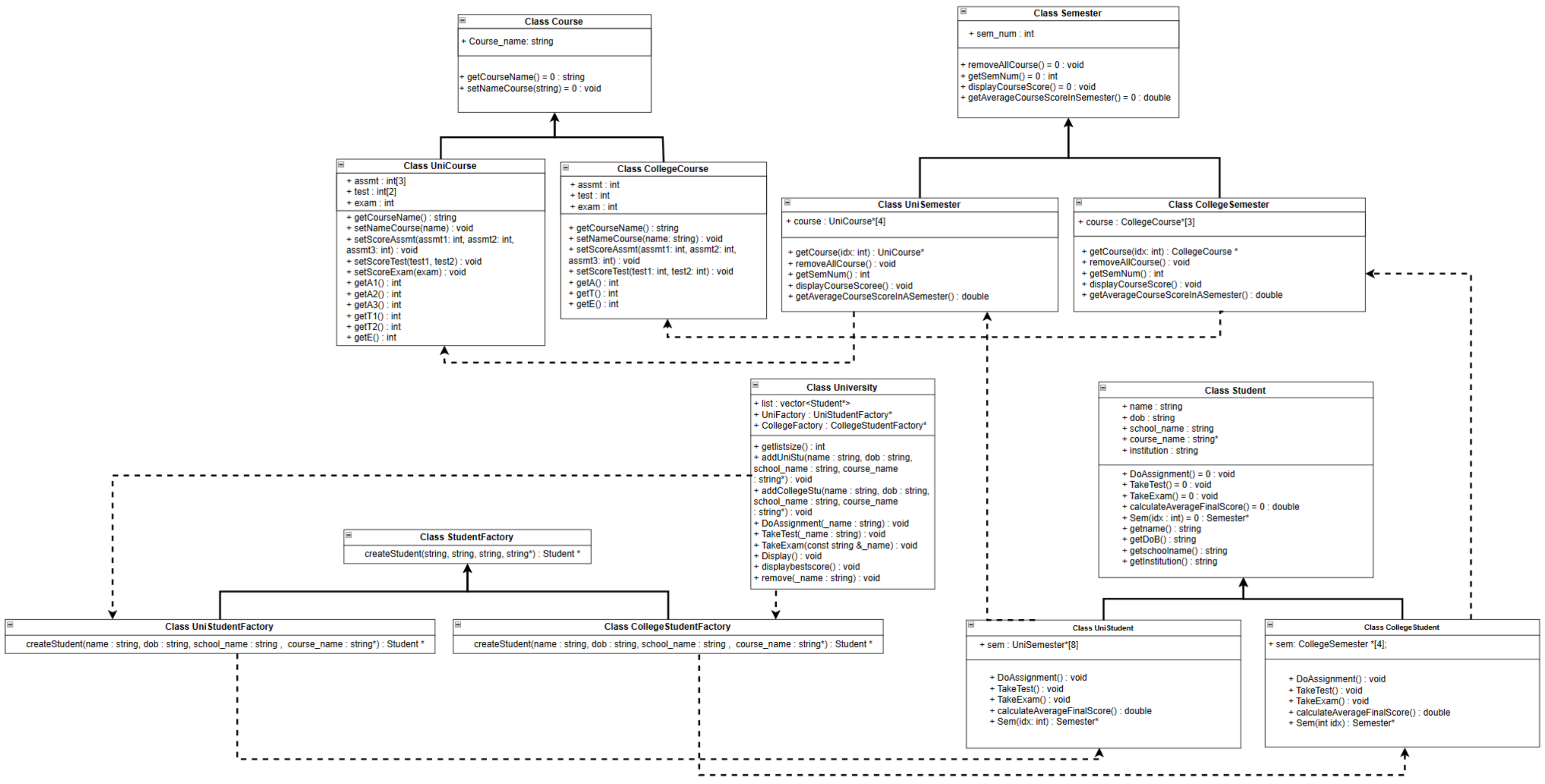
Class Course
+ Course_name: string
+ getCourseName() = 0 : string
+ setNameCourse(string) = 0 : void
Class Semester
+ sem_num : int
+ removeAllCourse() = 0 : void
+ getSemNum() = 0 : int
+ displayCourseScore() = 0 : void
+ getAverageCourseScoreInSemester() = 0 : double
Class UniCourse
+ assmt : int[3]
+ test : int[2]
+ exam : int
+ getCourseName() : string
+ setNameCourse(name) : void
+ setScoreAssmt(assmt1: int, assmt2: int, assmt3: int) : void
+ setScoreTest(test1, test2) : void
+ setScoreExam(exam) : void
+ getA1() : int
+ getA2() : int
+ getA3() : int
+ getT1() : int
+ getT2() : int
+ getE() : int
Class CollegeCourse
+ assmt : int
+ test : int
+ exam : int
+ getCourseName() : string
+ setNameCourse(name: string) : void
+ setScoreAssmt(assmt1: int, assmt2: int, assmt3: int) : void
+ setScoreTest(test1: int, test2: int) : void
+ getA() : int
+ getT() : int
+ getE() : int
Class UniSemester
+ course : UniCourse*[4]
+ getCourse(idx: int) : UniCourse*
+ removeAllCourse() : void
+ getSemNum() : int
+ displayCourseScoree() : void
+ getAverageCourseScoreInASemester() : double
Class CollegeSemester
+ course : CollegeCourse*[3]
+ getCourse(idx: int) : CollegeCourse *
+ removeAllCourse() : void
+ getSemNum() : int
+ displayCourseScore() : void
+ getAverageCourseScoreInASemester() : double
Class University
+ list : vector<Student*>
+ UniFactory : UniStudentFactory*
+ CollegeFactory : CollegeStudentFactory*
+ getlistsize() : int
+ addUniStu(name : string, dob : string, school_name : string, course_name : string*) : void
+ addCollegeStu(name : string, dob : string, school_name : string, course_name : string*) : void
+ DoAssignment(_name : string) : void
+ TakeTest(_name : string) : void
+ TakeExam(const string &_name) : void
+ Display() : void
+ displaybestscore() : void
+ remove(_name : string) : void
Class Student
+ name : string
+ dob : string
+ school_name : string
+ course_name : string*
+ institution : string
+ DoAssignment() = 0 : void
+ TakeTest() = 0 : void
+ TakeExam() = 0 : void
+ calculateAverageFinalScore() = 0 : double
+ Sem(idx : int) = 0 : Semester*
+ getname() : string
+ getDoB() : string
+ getschoolname() : string
+ getInstitution() : string
Class StudentFactory
createStudent(string, string, string, string*) : Student *
Class UniStudentFactory
createStudent(name : string, dob : string, school_name : string , course_name : string*) : Student *
Class CollegeStudentFactory
createStudent(name : string, dob : string, school_name : string , course_name : string*) : Student *
Class UniStudent
+ sem : UniSemester*[8]
+ DoAssignment() : void
+ TakeTest() : void
+ TakeExam() : void
+ calculateAverageFinalScore() : double
+ Sem(idx: int) : Semester*
Class CollegeStudent
+ sem: CollegeSemester *[4];
+ DoAssignment() : void
+ TakeTest() : void
+ TakeExam() : void
+ calculateAverageFinalScore() : double
+ Sem(int idx) : Semester*

Figure 2.1.1: *UML classes diagram for the Student Management Program*

To make the program based on the prompt of the assignment, I specified the Student information in 2 sub classes: CollegeStudent and UniStudent. A Student Class object must have some attributes that include the score of several courses in some Semester. Moreover, the number of Courses and the Semesters of a student is dependent on which type of student is. Therefore, I created the based class Course and Semester, then creating derived classes for each of them based on 2 types of Student, which are UniCourse and CollegeCourse classes for Course Class, UniSemester and CollegeSemester classes for Semester class.

2.2 Details of components in the class diagram

a. Course Class and derived classes: UniCourse & CollegeCourse

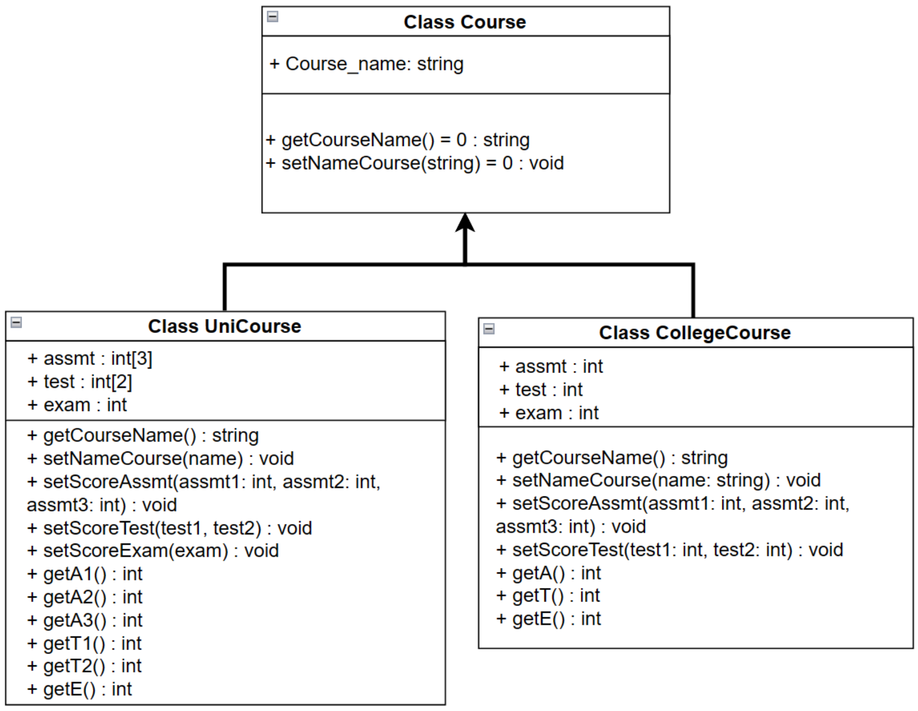
Class Course
+ Course_name: string
+ getCourseName() = 0 : string
+ setNameCourse(string) = 0 : void
Class UniCourse
+ assmt : int[3]
+ test : int[2]
+ exam : int
+ getCourseName() : string
+ setNameCourse(name) : void
+ setScoreAssmt(assmt1: int, assmt2: int, assmt3: int) : void
+ setScoreTest(test1, test2) : void
+ setScoreExam(exam) : void
+ getA1() : int
+ getA2() : int
+ getA3() : int
+ getT1() : int
+ getT2() : int
+ getE() : int
Class CollegeCourse
+ assmt : int
+ test : int
+ exam : int
+ getCourseName() : string
+ setNameCourse(name: string) : void
+ setScoreAssmt(assmt1: int, assmt2: int, assmt3: int) : void
+ setScoreTest(test1: int, test2: int) : void
+ getA() : int
+ getT() : int
+ getE() : int

Figure 2.2.1: *UML diagram of Course class and 2 derived classes*

The Course class contains the information of the course's name and course's scores of the student. Since there are 2 types of students (University student and College Student), the method of setting and getting courses' name and score for these 2 types student would be different. Therefore, I divide into 2 subclasses (UniCourse Class and CollegeCourse Class) that derived from the Course Class with different methods. In UniCourse Class, it contains the integer scores of 3 assignments, 2 tests and 1 exam as the attributes, with the methods from the bases class to get the course's name, and the setting - getting methods for assigning and returning values of the student's scores. Meanwhile, the CollegeCourse Class contains the scores of 1 assignment, 1 test and 1 exam as attributes, and the methods is different from the UniCourse Class, with 3 getting and 3 setting methods (the UniCourse Class has 6 getting and

3 setting methods)

b. Semester Class and derived classes: UniSemester & CollegeSemester

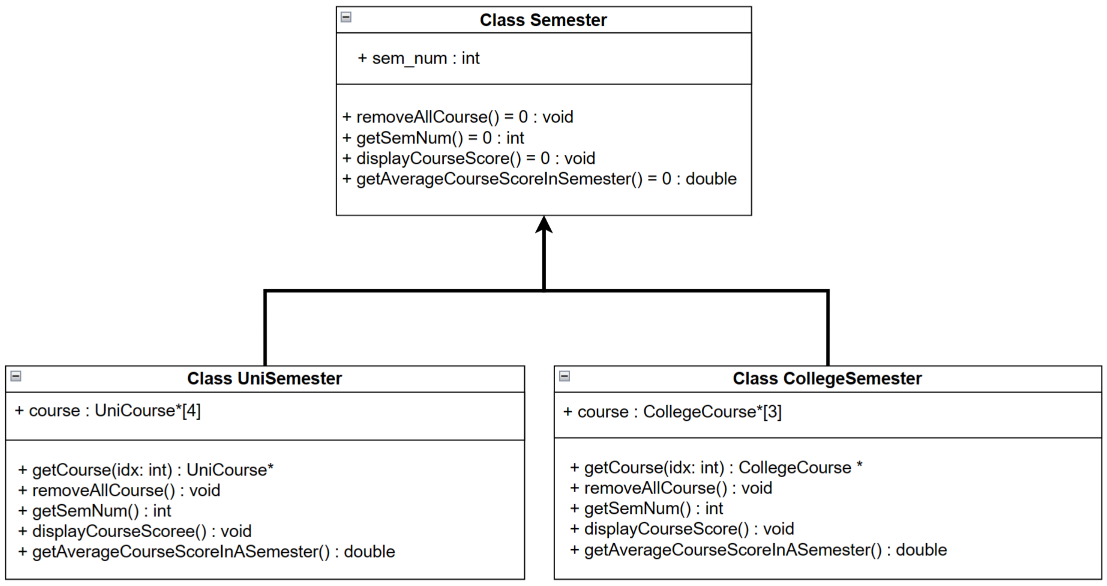
Class Semester
+ sem_num : int
+ removeAllCourse() = 0 : void
+ getSemNum() = 0 : int
+ displayCourseScore() = 0 : void
+ getAverageCourseScoreInSemester() = 0 : double
Class UniSemester
+ course : UniCourse*[4]
+ getCourse(idx: int) : UniCourse*
+ removeAllCourse() : void
+ getSemNum() : int
+ displayCourseScoree() : void
+ getAverageCourseScoreInASemester() : double
Class CollegeSemester
+ course : CollegeCourse*[3]
+ getCourse(idx: int) : CollegeCourse *
+ removeAllCourse() : void
+ getSemNum() : int
+ displayCourseScore() : void
+ getAverageCourseScoreInASemester() : double

Figure 2.2.2: *UML diagram of Semester class and 2 derived classes*

The Semester Class would comprise the information of the Students' Courses in a semester. Since the University Students have 4 courses and College Students have 3 courses in a semester, I divide into 2 subclasses that derive from the Semester class based types of students (UniSemester class and CollegeSemester class). There is not much difference in methods of 2 subclasses, except for the `getCourse()` method. Particularly, the `getCourse()` method in UniSemester Class would return the instance of UniCourse Class, while the method in CollegeCourse Class return the instance of CollegeCourse Class.

c. Class Student and derived classes: UniStudent & CollegeStudent

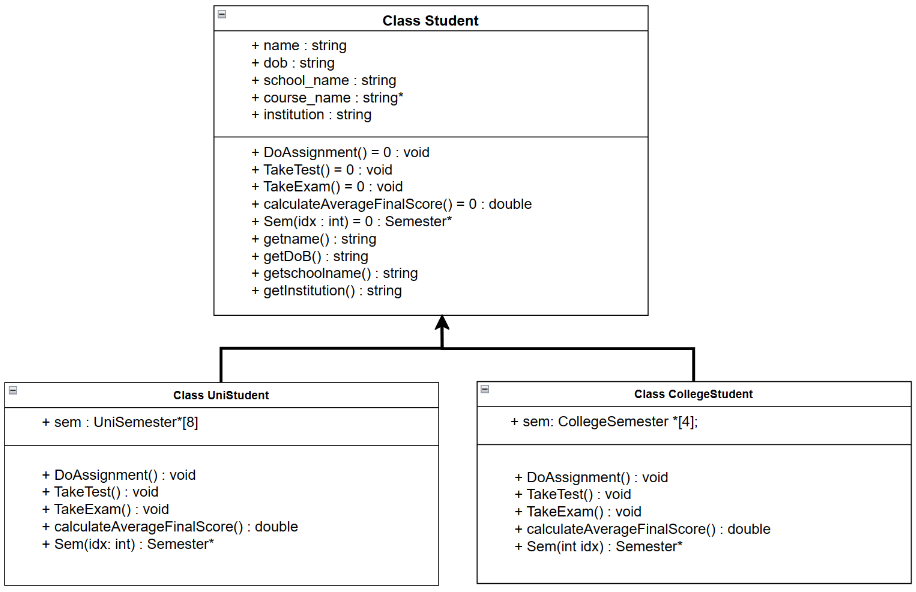

Figure 2.2.3: *UML diagram of Student class and 2 derived classes*

In the Student Class, it contains all of the student's information, which include the student's name, birthday, school's name, institution and average score. There are also 2 subclass for 2 types of student, which are UniStudent Class and CollegeStudent Class. Each subclass has different attribute in terms of Semester information (The UniStudent has 8 semesters, while the CollegeStudent has only 4 semesters). Moreover, the subclasses have the same methods, which are `DoAssignment()`, `TakeTest()`, `TakeExam()`, `calculateAverageFinalScore()` and `Sem()` methods.

d. StudentFactory Class and derived classes: UniStudentFactory & CollegeStudentFactory

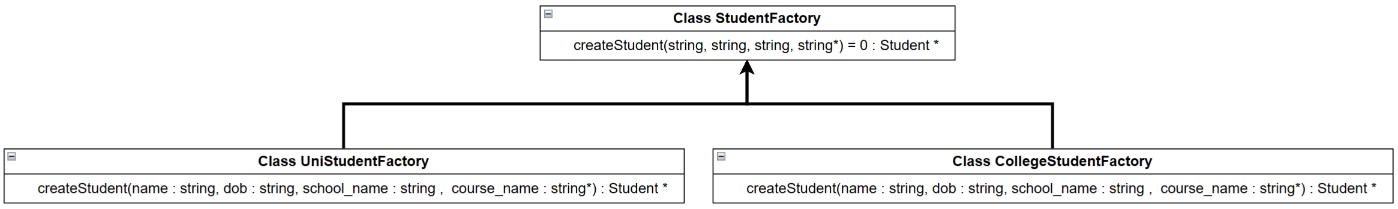
Class StudentFactory
createStudent(string, string, string, string*) = 0 : Student *
Class UniStudentFactory
createStudent(name : string, dob : string, school_name : string , course_name : string*) : Student *
Class CollegeStudentFactory
createStudent(name : string, dob : string, school_name : string , course_name : string*) : Student *

Figure 2.2.4: *UML diagram of StudentFactory class and 2 derived classes*

The StudentFactory Class with the derived classes (UniStudentFactory and CollegeStudentFactory) are used for making the method of creating object for the UniStudent Class and CollegeStudent Class. These classes are used based on the Factory Design Pattern, which would be explained in Section 4 of the report.

e. University Class

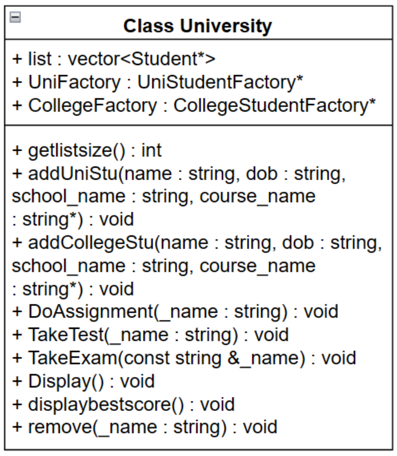
Class University
+ list : vector<Student*>
+ UniFactory : UniStudentFactory*
+ CollegeFactory : CollegeStudentFactory*
+ getlistsize() : int
+ addUniStu(name : string, dob : string, school_name : string, course_name : string*) : void
+ addCollegeStu(name : string, dob : string, school_name : string, course_name : string*) : void
+ DoAssignment(_name : string) : void
+ TakeTest(_name : string) : void
+ TakeExam(const string &_name) : void
+ Display() : void
+ displaybestscore() : void
+ remove(_name : string) : void

Figure 2.2.5: *UML diagram of University class*

The University Class holds the information of the Students list and some methods for adding, removing, displaying students' scores and randomly assigning students' score. The attributes of the class include the array of the instance of Student Class, the pointer of UniStudentFactory Class and CollegeStudentFactory Class (which are used for utilizing the `createStudent()` method).

3 Utilizing Vector for improving the program

std::vector in C++ STL is a class template that contains the vector container and its member functions. It is defined inside the <vector> header file. The member functions of the std::vector class provide various functionalities to vector containers.

The syntax to declare vector in C++:

```
std::vector<dataType> vectorName;
```

where the data type is the type of data of each element of the vector. We can remove the **std::** if we have already used the std namespace.

In my source code of assignment 2, I used the dynamic array for the pointer **list** of the Student Class in order to manage students' information in the University Class:

```
class University {
private:
    Student **list;
    int listsize;
public:
    University() {
        list = new Student*[capacity];
        listsize = 0;
    }
    void add(const int& option, const string& name, const string& dob, const
    string& school_name, string *course_name) {
        if (listsize == capacity) {
            capacity *= 2;
            Student **tmp = new Student*[capacity];
            for (int i = 0; i < listsize; i++) {
                tmp[i] = list[i];
                list[i] = nullptr;
            }
```

```cpp
            delete [] list;
            list = tmp;
        }
        if (option == 1)
            list[listsize++] = new UniStudent(name, dob, school_name, course_name);
        else if (option == 2)
            list[listsize++] = new CollegeStudent(name, dob, school_name, course_name);
        cout << "\n[" << name << "] " << "has been added on the list!\n";
    }
    void remove(const string& _name) {
        int cnt = 0;
        Student **tmp;
        for (int i = 0; i < listsize; i++) {
            if (list[i]->getname() == _name) {
                cnt++;
            }
        }
        if (cnt == 0) {
            cout << "[" << _name << "] is not available on the list!\n";
        }
        else {
            int new_size = listsize - cnt;
            tmp = new Student*[capacity];
            int cnt_tmp = 0;
            for (int i = 0; i < listsize; i++) {
                if (list[i]->getname() != _name) {
                    tmp[cnt_tmp++] = list[i];
                    list[i] = nullptr;
                }
                else {
                    delete list[i];
                    list[i] = nullptr;
                }
            }
            delete [] list;
            list = tmp;
            listsize = new_size;
            cout << "[" << _name << "] has been removed on the list!\n";
        }
    }
    ~University() {
        for (int i = 0; i < listsize; i++) {
            delete list[i];
        }
        delete [] list;
        listsize = 0;
    }
};
```

Figure 3.0.1: *University Class implementation in Assignment 2*

To create a new dynamic array for list of student, I initialize the pointer **list** in the constructor `University()` of University class and when the instance of University class is destroyed, the pointer **list** will be deleted in the destructor ∼`University()`.

However, there are some drawbacks when using dynamic array for managing the list of student. When the user wants to add more students in the list, which makes the list size exceeds the capacity (max size) of the array that has been set before, then we have to resize the array by store the data in temporary list array, reinitialize the old array with new size and get back the data from temporary. This method can make the code more complex and lengthy. Therefore, I improve my code by using the vector instead of using dynamic array:

```cpp
class University {
private:
    vector<Student*> list;
public:
    University() {}
    void add(const int& option, const string& name, const string& dob, const
    string& school_name, string *course_name) {
        if (option == 1) list.push_back(new UniStudent(name, dob, school_name,
    course_name));
        else if (option == 2) list.push_back(new CollegeStudent(name, dob,
    school_name, course_name));
        cout << "\n[" << name << "] " << "has been added on the list!\n";
    }
    void remove(const string& _name) {
        int cnt = 0;
        vector<Student*> tmp;
        int listsize = list.size();
        for (int i = 0; i < listsize; i++) {
            if (list[i]->getname() == _name) {
                cnt++;
            }
        }
        if (cnt == 0) {
            cout << "[" << _name << "] is not available on the list!\n";
        }
        else {
            for (int i = 0; i < listsize; i++) {
                if (list[i]->getname() != _name) {
                    tmp.push_back(list[i]);
                }
                else {
                    delete list[i];
                    list[i] = nullptr;
                }
            }
            for (int i = 0; i < listsize; i++) list.pop_back();
            int newsize = tmp.size();
            for (int i = 0; i < newsize; i++) {
                list.push_back(tmp[i]);
            }
            for (int i = 0; i < newsize; i++) tmp.pop_back();
            cout << "[" << _name << "] has been removed on the list!\n";
        }
    }
    ~University() {
        int listsize = list.size();
        for (int i = listsize - 1; i >= 0; i--) {
            delete list[i];
            list.pop_back();
        }
    }
};
```

Figure 3.0.2: *University Class implementation in Assignment 3*

By using the Vector, it makes the code more brief and efficient since the Vector is more convenience and easy to use rather than the dynamic array, as the vector can take only O(1) time complexity to add or remove the elements and dynamically resize themselves. Besides, since the vector is already in the standard library STL, the user can use additional functions that are in the library such as `push_backK()`, `begin()`, `end()`, `pop_back()`, ...

4 Utilizing Factory Design pattern for improving the program

In OOP, the Factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. By creating objects by calling a factory method - either specified in a interface and implemented by child classes, or implemented in a base class and optionally overidden by derived class - rather than calling a constructor.

In my assignment 3, I applied Factory design pattern to make factory methods that create UniStudent and CollegeStudent objects, which is based on the UML diagram of StudentFactory class (**Figure 2.2.4**). Particularly, I created the StudentFactory for the interface of the concrete classes, which are the UniStudentFactory class and CollegeStudentFactory class. The concrete classes contain the createStudent method

```
class StudentFactory {
public:
    virtual Student* createStudent(string, string, string, string *) = 0;
    virtual ~StudentFactory() {};
};

class UniStudentFactory : public StudentFactory {
public:
    Student *createStudent(string name, string dob, string school_name, string *
    course_name) override {
        return new UniStudent(name, dob, school_name, course_name);
    }
};

class CollegeStudentFactory : public StudentFactory {
public:
    Student *createStudent(string name, string dob, string school_name, string *
    course_name) override {
        return new CollegeStudent(name, dob, school_name, course_name);
    }
};
```

Figure 4.0.1: *the Classes for Factory Design Pattern implementation*

While in the University class (which manages all of the Student information), the add function will utilize the creating method of the UniStudentFactory and the CollegeStudentFactory to return the instance of UniStudent class or CollegeStudent class:

```
private:
    vector<Student *> list;
    UniStudentFactory *UniFactory;
    CollegeStudentFactory *CollegeFactory;
public:
University() {
    UniFactory = new UniStudentFactory();
    CollegeFactory = new CollegeStudentFactory();
}
void addUniStu(const string& name, const string& dob, const string& school_name,
    string *course_name) {
```

```
    list.push_back(UniFactory->createStudent(name, dob, school_name, course_name))
    ;
    cout << "\n[" << name << "] " << "has been added on the list!\n";
}
void addCollegeStu(const string& name, const string& dob, const string&
    school_name, string *course_name) {
    list.push_back(CollegeFactory->createStudent(name, dob, school_name,
    course_name));
    cout << "\n[" << name << "] " << "has been added on the list!\n";
}
```

Figure 4.0.2: *Adding Student methods implementation in University class*

By using the Factory Design Pattern, I can apply the polymorphism in OOP and manage to return the instance of 1 in 2 classes without any errors. Besides, if possible, I can easily make other classes of new type of Student that based on the interface class (Student) by adding a new concrete class derived from Student class and the new student factory class derived from the StudentFactory class, in order to make a create methods for the new Student class.

5 Utilizing Modern Smart Pointers, Modern Range For Loop and Auto keyword for improving the program

5.1 Modern Smart Pointers

A smart pointer is an abstract data type that simulates a pointer while providing some added features such as automatic memory management or bounds checking. The purpose of using smart pointer is to reduce the risk of memory leaks, dangling pointers and other memory-related errors in some programming languages that use manual memory management.

In C++, there are several types of smart pointer that are included in the <memory> standard library, such as: Unique pointer (**std::unique_ptr**), Shared pointer (**std::shared_ptr**), Weak pointer (**std::weak_ptr**)

In the assignment 3, I improve the createStudent method in the UniStudentFactory class and CollegeStudentFactory class (**Figure 4.0.1**) by utilizing the unique pointer:

```
class StudentFactory {
public:
    virtual unique_ptr<Student> createStudent(string, string, string, string *) =
    0;
    virtual ~StudentFactory() {};
};

class UniStudentFactory : public StudentFactory {
public:
    unique_ptr<Student> createStudent(string name, string dob, string school_name,
     string *course_name) override {
        return make_unique<UniStudent>(name, dob, school_name, course_name);
    }
};

class CollegeStudentFactory : public StudentFactory {
public:
    unique_ptr<Student> createStudent(string name, string dob, string school_name,
     string *course_name) override {
        return make_unique<CollegeStudent>(name, dob, school_name, course_name);
    }
};
```

Figure 5.1.1: *Adding Students method implementation using Unique Pointer*

In the University class, instead of using vector of pointer of Student class in **Figure 4.0.2**, I use unique pointer for the vector of Student class pointer and 2 pointers of UniStudentFactory Class and CollegeStudentFactory Class.

```
private:
    vector<unique_ptr<Student>> list;
    unique_ptr<UniStudentFactory> UniFactory;
    unique_ptr<CollegeStudentFactory> CollegeFactory;
public:
University() {
    UniFactory = make_unique<UniStudentFactory>();
    CollegeFactory = make_unique<CollegeStudentFactory>();
}
void addUniStu(const string& name, const string& dob, const string& school_name,
    string *course_name) {
    list.push_back(UniFactory->createStudent(name, dob, school_name, course_name))
    ;
    cout << "\n[" << name << "] " << "has been added on the list!\n";
}
void addCollegeStu(const string& name, const string& dob, const string&
    school_name, string *course_name) {
    list.push_back(CollegeFactory->createStudent(name, dob, school_name,
    course_name));
    cout << "\n[" << name << "] " << "has been added on the list!\n";
}
```

Figure 5.1.2

By using the unique pointer, I can reduce my code length as there is no need to use the **delete** keyword in order to deallocate the pointer. It also helps the memory to be more well-managed since the unique pointer can automatically deallocate the pointer itself after it goes out of scope or when the program terminates

5.2 Modern Range for loop & Auto keyword

Modern range for loop (or range-based for loop) in C++ is a loop function that executes a for loop over a range. It is used as a more readable equivalent to traditional for loop operating over a range of values, such as all elements in a container.

A syntax for range for loop function is:

```
    for ( range_declaration : range_expression )
        loop_statement
```

where:

- **range_declaration**: a declaration of a named variable, whose type is the type of the element of the sequence represented by range_expression, or a reference to that type. Often uses the **auto** keyword for automatic type deduction.
- **range_expression**: any expression that represents a suitable sequence or a braced-init-list.
- **loop_statement**: any statement, typically a compound statement, which is the body of the loop

The auto keyword in C++ specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In the case of functions, if their return type is auto then that will be evaluated by return type expression at runtime. Good use of auto is to avoid long initialization when creating iterators for containers.

In my program, I use for-range loop for the `DoAssignment()`, `TakeTest()`, `TakeExam()`, `display()`, `displaybestscore()` functions to iterate list of students for assigning or displaying each student's score.

```cpp
void DoAssignment(const string& _name) {
    bool found = false;
    for (auto &it: list) {
        if (it->getname() == _name) {
            it->DoAssignment();
            found = true;
        }
    }
    cout << endl;
    if (!found) cout << "[" << _name << "] is not on the list\n";
    else cout << "[" << _name << "] has done Assignment\n";
}
```

Figure 5.2.1: *DoAssignment function in University Class*

```cpp
void TakeTest(const string& _name) {
    bool found = false;
    for (auto &it: list) {
        if (it->getname() == _name) {
            it->TakeTest();
            found = true;
        }
    }
    cout << endl;
    if (!found) cout << "[" << _name << "] is not on the list\n";
    else cout << "[" << _name << "] has taken Test\n";
}
```

Figure 5.2.2: *TakeTest function in University Class*

```cpp
void TakeExam(const string& _name) {
    bool found = false;
    for (auto &it: list) {
        if (it->getname() == _name) {
            it->TakeExam();
            found = true;
        }
    }
    cout << endl;
    if (!found) cout << "[" << _name << "] is not on the list\n";
    else cout << "[" << _name << "] has taken Exam\n";
}
```

Figure 5.2.3: *TakeExam function in University Class*

```cpp
void display() {
    for (auto it = list.begin(); it != list.end(); it++) {
        cout << it - list.begin() + 1 << ". Stu name: " << (*it)->getname() << "; School's name: " << (*it)->getschoolname() << "; Institution: " << (*it)->getInstitution() << endl;
        cout << "Student's score:" << endl;
        int sem_size = ((*it)->getInstitution() == "University") ? 8 : 4;
        for (int j = 0; j < sem_size; j++) {
            cout << "Semester " << j + 1 << ":" << endl;
            (*it)->Sem(j)->displayCourseScore();
            cout << "Average score in Semester " << j + 1 << ": " << (*it)->Sem(j)->getAverageCourseScoreInASemester() << endl;
            cout << endl;
        }
        if (it != list.end() - 1) cout << "====================================================================\n";
    }
}
```

Figure 5.2.4: *display function in University Class*

When using the for-range loop and Auto keyword, my code is more readable, simple and easy to understand. Moreover, the for-range loop can make my code become more efficient as it optimizes iteration under the hood, resulting in faster execution compared to traditional loops and reduce the chance of off-by-one errors and other common mistakes compared to the traditional for loops. The Auto keyword can help avoid unnecessary type conversions and makes code easier to maintain, especially when dealing with complex types of templates

6 Reference

References

[1] Source Making. *Design Patterns*. Link: https://sourcemaking.com/design_patterns

[2] Alexander S. Gillis and Sarah Lewis. July 2021. *objected-oriented programming (OOP)*. Link: https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

[3] Wikipedia. December 4, 2023. *Smart Pointer*. Link: https://en.wikipedia.org/wiki/Smart_pointer

[4] Geeksforgeeks. March 27, 2024. *Vector in C++ STL*. Link: https://www.geeksforgeeks.org/vector-in-cpp-stl/

[5] Wikipedia. March 29, 2024. *Factory method pattern*. Link: https://en.wikipedia.org/wiki/Factory_method_pattern

[6] Geeksforgeeks. March 22, 2024. *Range-based for loop in C++*. Link: https://www.geeksforgeeks.org/range-based-loop-c/

[7] Geeksforgeeks. November 02, 2023. *Type Inference in C++*. Link: https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/