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

ADVANCED PROGRAMING (CO2039)

Assignment (Semester: 232)

ASSIGNMENT 3 CPP

Instructor: Han Duy Phan, *Monash University*
Truong Tuan Anh, *CSE-HCMUT*

Students: Nguyen Thien Loc - 2252460

HO CHI MINH CITY, JANUARY 2024

Contents

1 PROBLEMS

1.1 PART 1:

Improve the simple C++ student management program from Assignment 2. In this part, you need to utilize at least one of the C++ data structures below to improve your program.

1.2 PART 2:

Improve the simple C++ student management program from Part 1. In this part, you need to utilize at least one of the following design patterns to improve your program.

1.3 PART 3:

Improve the simple C++ student management program from Part 2. In this part, you need to utilize at least one of the following modern C++ features to improve your program.

1.4 PART 4:

Produce a report (PDF format) that includes all the following contents:

- An UML diagram with all the classes and relationships and necessary details. Use the notations shown in my UML essentials cheat sheet.
- Detailed explanations why your changes in Part 1 lead to good improvements.
- Detailed explanations why your changes in Part 2 lead to good improvements.
- Detailed explanations why your changes in Part 3 lead to good improvements.

2 SOLUTIONS

2.1 UML Diagram

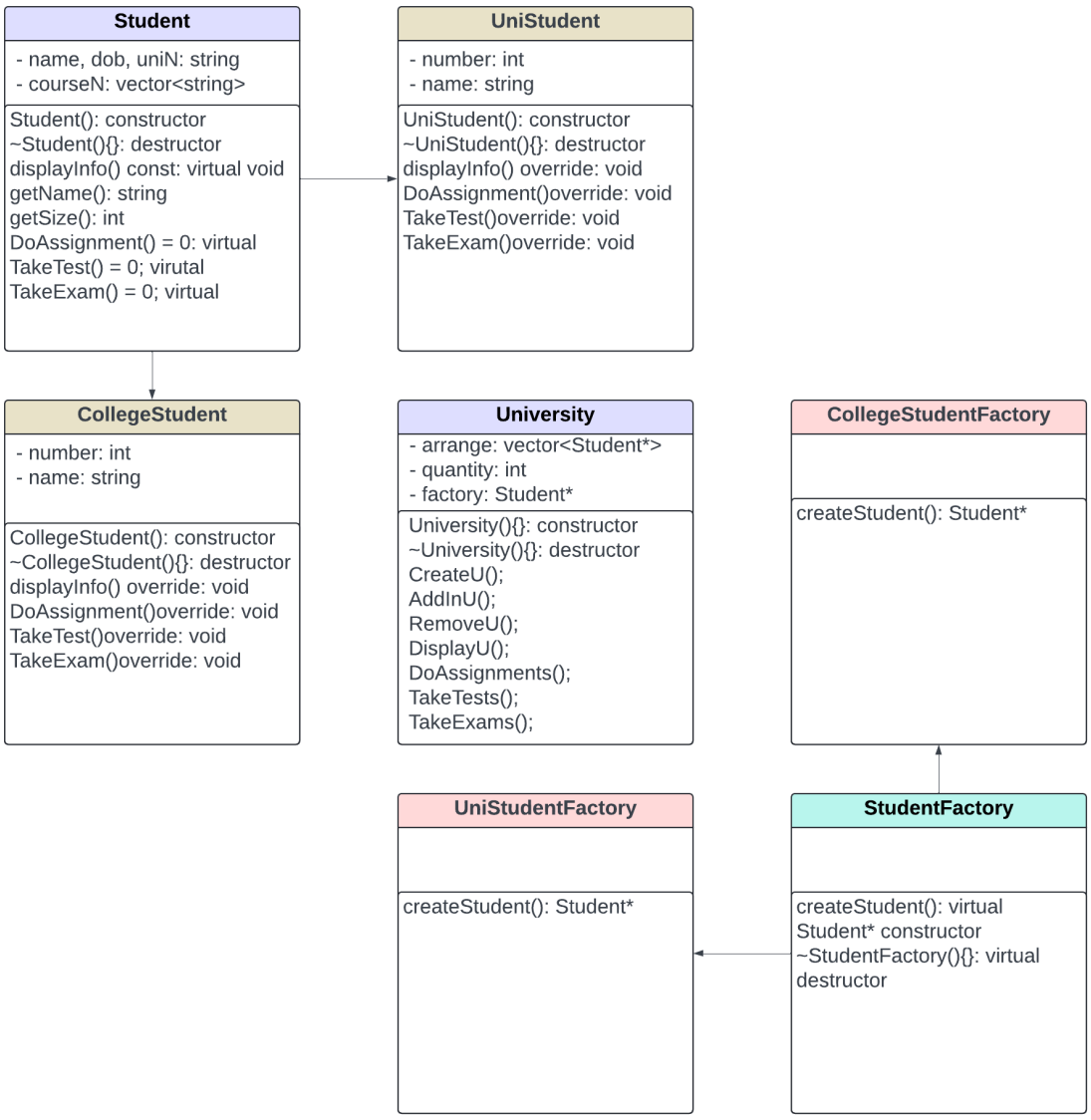

Figure 1: UML diagram

2.2 PART 1:

Initially, I considered using an *array* to store the list of students and their courses. However, after several assignments, I've come to realize that using *array* presents numerous obstacles affect its effectiveness:

- *Fixed size*: Arrays cannot be resized once they're created because they have a fixed size.
- *Wasted space*: Arrays might occupy unnecessary space when some elements in the collection remain unused.

Finally, i've decided to choose *vectors* as my final choice. Using *vectors* help me solve most of the challenges associated with using *array* and offers more advantages:

- Vectors are essentially dynamic arrays that offer a list interface, while arrays can be implemented as either static or dynamic structures with a primitive data type interface.
- Vectors can change its size automatically when an element is inserted or deleted.
- Arrays lack the ability to be directly copied or assigned, whereas Vectors can be easily copied or assigned directly.

There are some samples in codes:

```
private:
    std::string name;
    std::string dob;
    std::string uniN;
    vector<std::string> courseN;
```

Figure 2: example 1

```
private:
    vector<Student*> arrange;
    int quantity;
```

Figure 3: example 2

2.3 PART 2:

In this case, i'm using *Factory Pattern* to solve the problem.

```
class StudentFactory {
public:
    virtual ~StudentFactory() {}
    virtual Student* createStudent(const std::string& name, const std::vector<std::string>& courseN, const std::string& dob, const std::string& uniN) = 0;
};

class UniStudentFactory : public StudentFactory {
public:
    Student* createStudent(const std::string& name, const std::vector<std::string>& courseN, const std::string& dob, const std::string& uniN) override {
        return new UniStudent(name, courseN, dob, uniN);
    }
};

class CollegeStudentFactory : public StudentFactory {
public:
    Student* createStudent(const std::string& name, const std::vector<std::string>& courseN, const std::string& dob, const std::string& uniN) override {
        return new CollegeStudent(name, courseN, dob, uniN);
    }
};
```

Figure 4: *Factory Design Pattern*

Using *Factory Design Patter* gives more additional benefit:

- *Extension*: Easily introducing new type of *Students* such as *Part-time system*, can be done seamlessly without altering the main code. By simply creating a new subclass and implementing the factory method to it.
- *Code Reusability*: The factory method can be reused in different parts of the application where object creation is needed. Enabling them to share a common interface contributes to shorter, more concise code.
- *Encapsulation*:The factory method encapsulates the object creation process, simplifying the modification of this process without causing disruptions to the rest of the code.

2.4 PART 3:

I've opted for Modern Auto in this section. It simplifies my code and enhances its comprehensibility.

There are several advantages to utilizing *Modern Auto* in C++:

```cpp
void University::DisplayU(){
   for (const auto& student : arrange) {
        student->displayInfo();
        cout << endl;
    }
}

void University::TakeExams(){
    for (const auto& student : arrange) {
        student->TakeExam();
        cout << endl;
    }
}

void University::TakeTests(){
    for (const auto& student : arrange) {
        student->TakeTest();
        cout << endl;
    }
}

void University::DoAssignments(){
    for (const auto& student : arrange) {
        student->DoAssignment();
        cout << endl;
    }
}
```

Figure 5: Modern Auto

- *Concise code*: Using *Modern Auto* enables variable declaration without explicitly stating their data types, leading to more concise and readable code.
- *Flexibility*: *Modern Auto* proves especially handy when dealing with variables of complex or difficult to determine at a glance.
- *Maintenance*: By automatically adjusting to changes in variable types, *Modern Auto* can enhance code maintainability.

3 Conclusion

In conclusion, creating a student management system using Object-Oriented Programming (OOP) has taught me a great deal. It's fascinating to see how one problem can be solved in various ways, like using queues, vectors, and other methods to simplify the code.

Be side that, It's also interesting how we can refine the main code to enhance readability, reduce complexity, and make it more adaptable for future expansions in each assignment.

Finally, I would like to express my deep gratitude to Dr. Han Duy Phan and Mr. Truong Tuan Anh for their efforts in enhancing students' skills and fostering motivation and a sense of belonging to this subject, as well as with computer science.