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

ASSIGNMENT A3

Advanced Programming

Instructor: Dr Han Duy Phan

Student name: Nguyễn Quốc Huy

Student ID: 2053045

1. An UML diagram with all the classes and relationships and necessary details. Use the notations shown in my UML essentials cheat sheet.

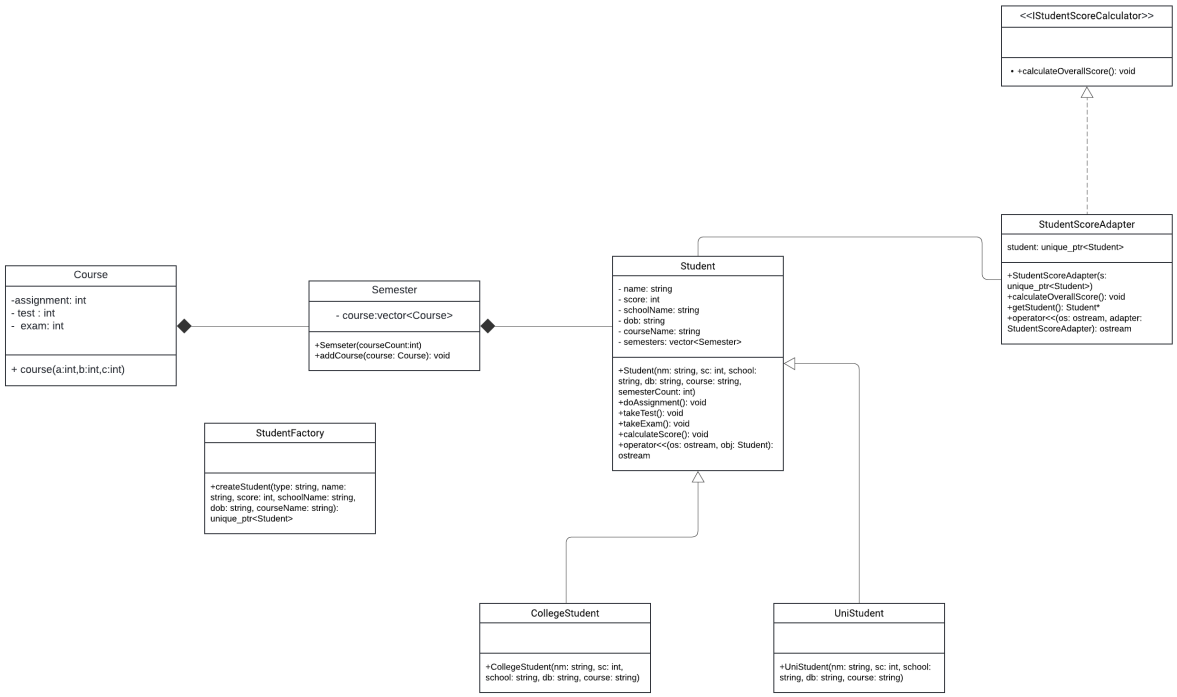

Figure 1: UML diagram for A3 Assignment

2. Detailed explanations why your changes in Part 1 lead to good improvements.

2.1 Memory Management.

Before: Uses raw pointers (Semester* semesters and Course* courses) for dynamic memory allocation. This requires explicit management of memory using new and delete, increasing the risk of memory leaks and pointer errors.

After modification: Uses std::vector to manage collections of Semesters and Courses. This automatically handles memory allocation and deallocation, significantly reducing the risk of memory leaks and making the code safer and easier to maintain.

2.2 Code duplication.

Before: Contains duplicated code across methods for different student types (UniStudent and CollegeStudent), specifically in the methods doAssignment, takeTest, and takeExam.

After modification: Inherits methods directly from the base class Student without unnecessary overrides unless needed for different behavior. This reduces code duplication and enhances maintainability.

2.3 Simple and clarify.

After modification: Offers clearer and more straightforward handling of objects and memory. There is less boilerplate for memory management, making the code easier to read and understand. It also reduces the chance of introducing bugs related to dynamic memory management.

2.4. Exception safety.

After modification: By avoiding manual new and delete, the code avoids common pitfalls that can lead to exceptions or memory leaks during resource allocation failures.

2.5 calability.

After modification: With std::vector and std::map, it's easier to scale the application to manage more students or more complex academic structures without changing the underlying memory management logic.

3. Detailed explanations why your changes in Part 2 lead to good improvements.

The after modification of the source code introduces a more advanced design by incorporating design patterns, specifically the Adapter, and Factory patterns. These patterns are implemented to structure the code better, manage object creation and operations, and ensure that the system is more scalable, maintainable, and robust. Here's a detailed explanation of why the second version is an improvement over the first:

3.1 Use of the factory pattern

Before modification: Directly creates instances of UniStudent and CollegeStudent within the main() function. This approach is straightforward but tightly couples the object creation process with the use of these objects.
After modification: Implements the StudentFactory to abstract the creation logic of Student objects. This pattern decouples the creation details from the client code (in this case, the main() function), making the system easier to modify and extend. For instance, adding a new type of student would only require changes in the factory class without touching the client code.

3.2 Use of adapter pattern

Before modification: Lacks any implementation of the Adapter pattern, leading to a more rigid design where each class must conform to the existing interfaces directly.
After modification: Introduces StudentScoreAdapter, which adapts Student objects to the new IStudentScoreCalculator interface. This allows for the Student class to be used in contexts where a different interface is expected, enhancing flexibility and enabling interoperability among components that otherwise wouldn't work together seamlessly.

3.3 Code reuse and maintenance

Before modification : More repetitive and potentially harder to maintain as the project scales. Changes in the object creation process or in the class structure require modifications in all parts of the code where these objects are created or used.
After modification: Centralizes the creation logic in the factory and the operation adaptation in the adapter. This not only reduces code duplication but also centralizes maintenance tasks. For instance, if the initialization parameters for UniStudent or CollegeStudent change, only the factory needs to be updated.

3.4 Separations of concerns

After modification: Clearly separates concerns among different parts of the application. The StudentFactory is only concerned with object creation, StudentScoreAdapter is only concerned with adapting Student objects to a specific interface, and the main logic in main()

focuses on application logic without being cluttered with creation details or specific adaptations.

4. Detailed explanations why your changes in Part 3 lead to good improvements.

4.1 Benefits of Using Smart Pointers (std::unique_ptr)

Memory Management:

Automatic Resource Management: Smart pointers take care of allocating and deallocating memory automatically. std::unique_ptr, in particular, automatically deletes the object it points to when the unique_ptr goes out of scope. This prevents memory leaks, a common problem in C++ where manual memory management is involved.

Exception Safety: With smart pointers, resources are freed automatically even if an exception is thrown and caught, ensuring no resource leak. This is part of the RAII (Resource Acquisition Is Initialization) principle in C++.

Ownership Semantics:

Unique Ownership: std::unique_ptr enforces unique ownership of the memory resource. Only one unique_ptr can own a particular resource at a time, which simplifies understanding of ownership (especially useful in large projects) and ensures that there are no dangling pointers or double deletions.

4.2 Benefit of using range-based for loop

Readability and Simplicity

Clear Intent: The range-based for loop (for (auto& item : items)) makes the code easier to read and write. It clearly indicates that the code is performing an operation on each element in the container, reducing boilerplate code and potential for errors, such as off-by-one errors or incorrect end conditions.

Less Error-Prone: Since the loop automatically handles the beginning and end of the container, there's no need to manage loop counters or iterators directly, which are common sources of errors.

Enhanced safety.

Type Safety: The use of auto in range-based for loops avoids type mismatches and makes the code robust against changes in the container's element type. If the type of elements in the container changes, the loop will continue to work without any modifications.