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

ADVANCED PROGRAMMING (CO2039)

Semester: 232

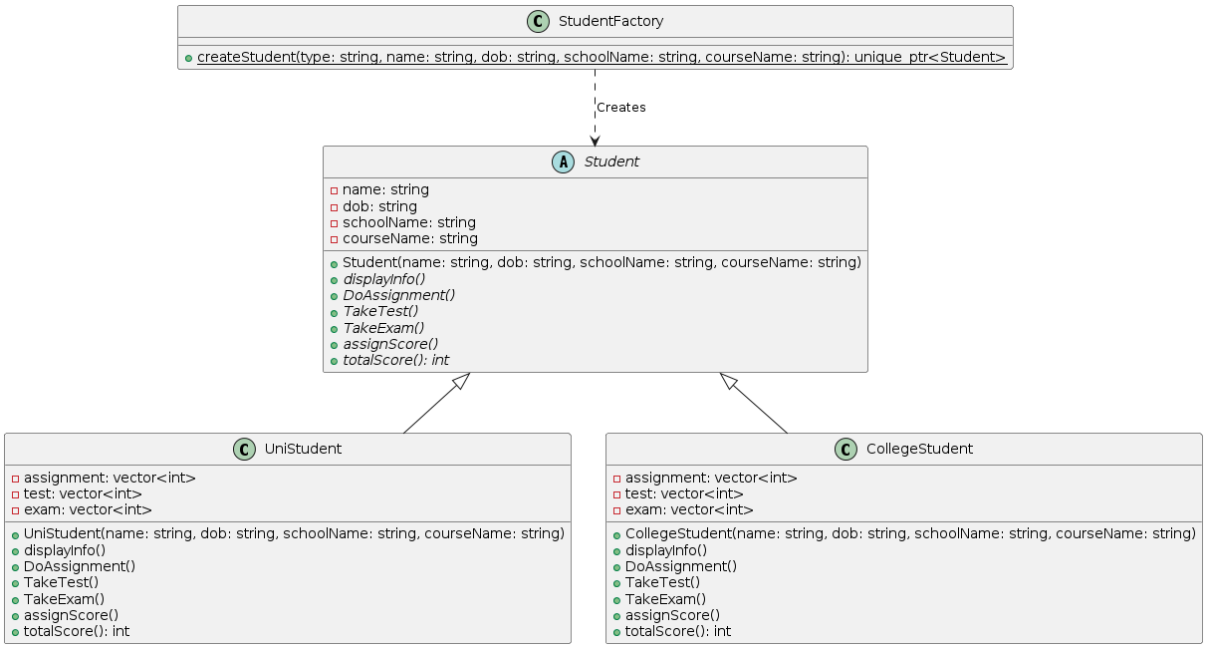
Design Pattern and Modern C++ Programming

Trần Đình Đăng Khoa - 2211649.

HO CHI MINH CITY, APRIL 2024

Contents

1 UML Diagram



StudentFactory class

- This class is responsible for creating `Student` objects.
- It has a static method `createStudent` that takes the student type, name, date of birth, school name, and course name as parameters, and returns a `unique_ptr<Student>` object.

Student abstract base class

- This is an abstract class that serves as the base for different types of students.
- It has protected members `name`, `dob`, `schoolName`, and `courseName`.
- It has a constructor that initializes these members.
- It has pure virtual functions `displayInfo`, `DoAssignment`, `TakeTest`, `TakeExam`, `assignScore`, and `totalScore` that must be implemented by derived classes.

UniStudent class

- This class is derived from the `Student` class and represents a university student.
- It has private members `assignment`, `test`, and `exam`, which are vectors storing scores.
- It has a constructor that initializes the base class members.
- It overrides and implements the pure virtual functions from the `Student` class.

CollegeStudent class

- This class is also derived from the `Student` class and represents a college student.
- It has private members `assignment`, `test`, and `exam`, which are vectors storing scores.
- It has a constructor that initializes the base class members.
- It overrides and implements the pure virtual functions from the `Student` class.

Relationships

The relationships between the classes are:

- The `StudentFactory` class has a "Creates" relationship with the `Student` class, as it creates objects of the derived classes (`UniStudent` and `CollegeStudent`) using the `createStudent` method.
- The `UniStudent` and `CollegeStudent` classes have an "Inheritance" relationship with the `Student` class, as they inherit from the abstract base class `Student`.

This UML diagram provides a visual representation of the class structure, relationships, and member functions, making it easier to understand and maintain the code.

2 Improvements with `std::vector`

- **Dynamic Memory Allocation**: Using `vector` instead of static arrays allows for dynamic memory allocation. Static arrays have fixed sizes determined at compile time, which can be limiting and inefficient, especially when dealing with unknown or variable data sizes. By using `vector`, memory is allocated dynamically as needed, providing flexibility and efficient memory usage.
- **Improved Readability and Maintainability**: The use of `vector` makes the code more readable and maintainable. The intention of storing assignment, test, and exam scores is clearer with `vector<int>` than with static arrays. Additionally, the code to initialize and populate the vectors is simpler and more concise.
- **Standard Library Functions**: The program utilizes standard library functions such as `accumulate` from the `<numeric>` header. This function simplifies the process of calculating the total score by summing up all elements of a container. By leveraging standard library functions, the code becomes more expressive and less error-prone.
- **Encapsulation and Code Organization**: The encapsulation of assignment, test, and exam scores within the `UniStudent` and `CollegeStudent` classes promotes better code organization and encapsulation. Each class manages its own data (scores) and operations (calculating total score), adhering to the principles of object-oriented programming (OOP).
- **Improved Scalability**: The use of `vector` enhances the scalability of the program. If the number of assignments, tests, or exams needs to be changed or expanded in the future, it can be easily accommodated by modifying the loop parameters without needing to change the underlying data structures.
- **Simplified Initialization and Population**: The initialization and population of assignment, test, and exam scores are simplified with the use of `vector`. There is no need to manually calculate the size of arrays or manage array indices. The `push_back` method appends elements to the end of the vector, automatically resizing it as needed.

3 Utilizing the Factory Design Pattern

Introducing the Factory Design Pattern to the C++ student management program led to several improvements:

- **Separation of Concerns**: The creation of student objects is now separated from the client code (the `main` function). This separation promotes better code organization and maintainability.
- **Abstraction**: The `StudentFactory` class abstracts away the details of object creation, making it easier to change or add new types of students in the future.
- **Extensibility**: If you need to introduce a new type of student, you only need to create a new concrete student class and modify the `StudentFactory::createStudent` method, without changing the existing code.
- **Code Reusability**: The `StudentFactory` class can be reused in other parts of the application or even in other projects that require similar functionality.
- **Loose Coupling**: The client code (`main` function) is now loosely coupled with the concrete student classes, making the system more flexible and easier to modify.

By applying the Factory Design Pattern, the code becomes more modular, scalable, and maintainable, adhering to the principles of good software design.

4 Utilizing Modern C++ Features

Incorporating modern C++ features into the student management program resulted in the following improvements:

Smart Pointers

- **Memory Safety**: By using `unique_ptr`, the program automatically manages the memory for student objects, preventing memory leaks and ensuring proper resource cleanup.
- **Exception Safety**: `unique_ptr` guarantees that resources are properly deallocated even in the presence of exceptions or early returns.
- **Reduced Boilerplate Code**: Smart pointers eliminate the need for manual memory management, reducing boilerplate code and potential errors.

Lambda Expressions

- **Code Readability**: The use of lambda expressions for generating random scores makes the code more concise and easier to understand.
- **Reusability**: Lambda expressions can be easily reused throughout the codebase, promoting code reuse and consistency.
- **Functional Programming Style**: Lambda expressions encourage a functional programming style, which can lead to more modular and testable code.

Auto Keyword

- **Reduced Verbosity**: The `auto` keyword eliminates the need to explicitly specify the type of variables, making the code more concise and easier to read.
- **Type Deduction**: The compiler deduces the type automatically, reducing the chance of type mismatches and making the code more maintainable.

Range-based for Loops (not used in this specific example)

- **Code Readability**: Range-based for loops provide a more concise and readable syntax for iterating over containers, improving code comprehension.
- **Safety**: Range-based for loops eliminate common errors associated with manual index handling, such as off-by-one errors or iterator invalidation.

By leveraging these modern C++ features, the code becomes more robust, safer, and easier to maintain, aligning with modern coding practices and promoting better code quality.

5 Conclusion

The C++ student management program presented in this document showcases the effective application of design patterns and modern C++ features to improve code quality, maintainability, and scalability. By incorporating the Factory Design Pattern, the program achieves better organization, extensibility, and decoupling of components, enabling easier addition of new student types and promoting code reuse.

Furthermore, the utilization of modern C++ features, such as smart pointers, lambda expressions, and the auto keyword, has significantly improved the program's robustness, readability, and efficiency. Smart pointers ensure automatic memory management, preventing memory leaks and enhancing exception safety. Lambda expressions promote code conciseness and encourage a functional programming style, leading to more modular and testable code. The auto keyword reduces verbosity and improves maintainability by automatically deducing variable types.

The use of the vector container instead of static arrays has also contributed to the program's flexibility, scalability, and improved code organization. Dynamic memory allocation, encapsulation of data within classes, and the utilization of standard library functions have made the code more expressive, less error-prone, and easier to maintain.

Overall, this project demonstrates the benefits of applying design patterns and embracing modern C++ features in software development. By adhering to these practices, developers can create more robust, maintainable, and scalable applications, fostering better code quality and promoting best practices in software engineering.