**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

----------



**BÀI TẬP NHÓM MÔN HỌC**

**KIẾN TRÚC VÀ THIẾT KẾ PHẦN MỀM**

**MIOStore - Assignment 5**

**Giảng viên hướng dẫn :** Trần Đình Quế

**Nhóm môn học : 03**

**Nhóm bài tập : 08**

**Danh sách thành viên :**

Lê Thị Hoa B16DCCN151

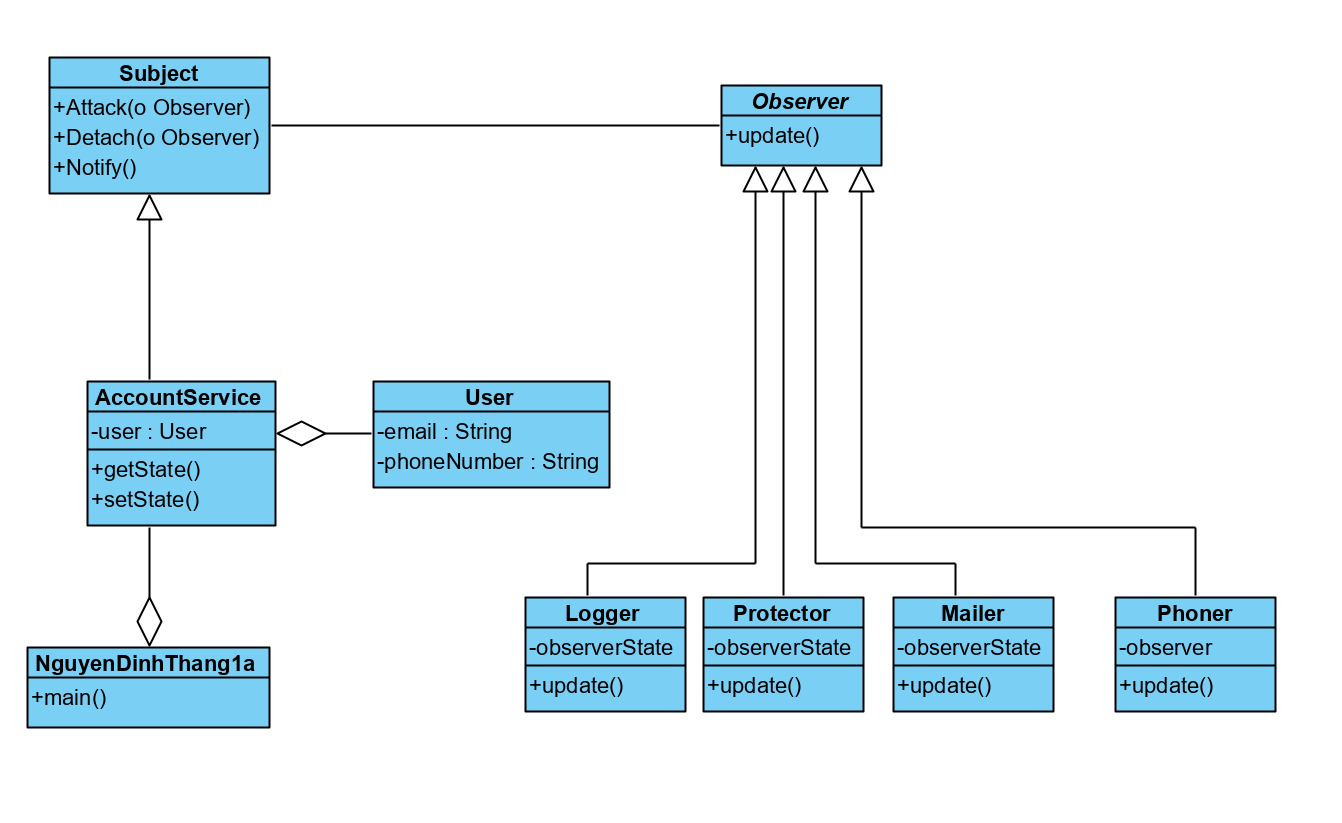
Phạm Ngọc Hoàng B16DCCN159

**Nguyễn Đình Thắng B16DCCN319**

Phan Quang Thành B16DCCN331

Trần Nhật Minh B16DCCN519

* 1. **Using Observer Pattern**
     + Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface Subject |  |
| Interface Observer |  |
| Class AccountService |  |
| Class Logger |  |
| Class Mailer |  |
| Class Phoner |  |
| Class Protector |  |
| Class User |  |
| Class NguyenDinhThang1a |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
* Lợi ích:
  + Dễ dàng mở rộng với ít sự thay đổi : mẫu này cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác. Vì vậy, nó đảm bảo nguyên tắc Open/Closed Principle (OCP).
  + Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
  + Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

Bên cạnh những lợi ích, chúng ta cần xem xét đến trường hợp cập nhật không mong muốn (Unexpected update) của Subject. Bởi vì các Observer không biết về sự hiện diện của nhau, nó có thể gây tốn nhiều chi phí của việc thay đổi Subject.

Sử dụng khi nào?

Thường được sử dụng trong mối quan hệ 1-n giữa các object với nhau. Trong đó một đối tượng thay đổi và muốn thông báo cho tất cả các object liên quan biết về sự thay đổi đó.

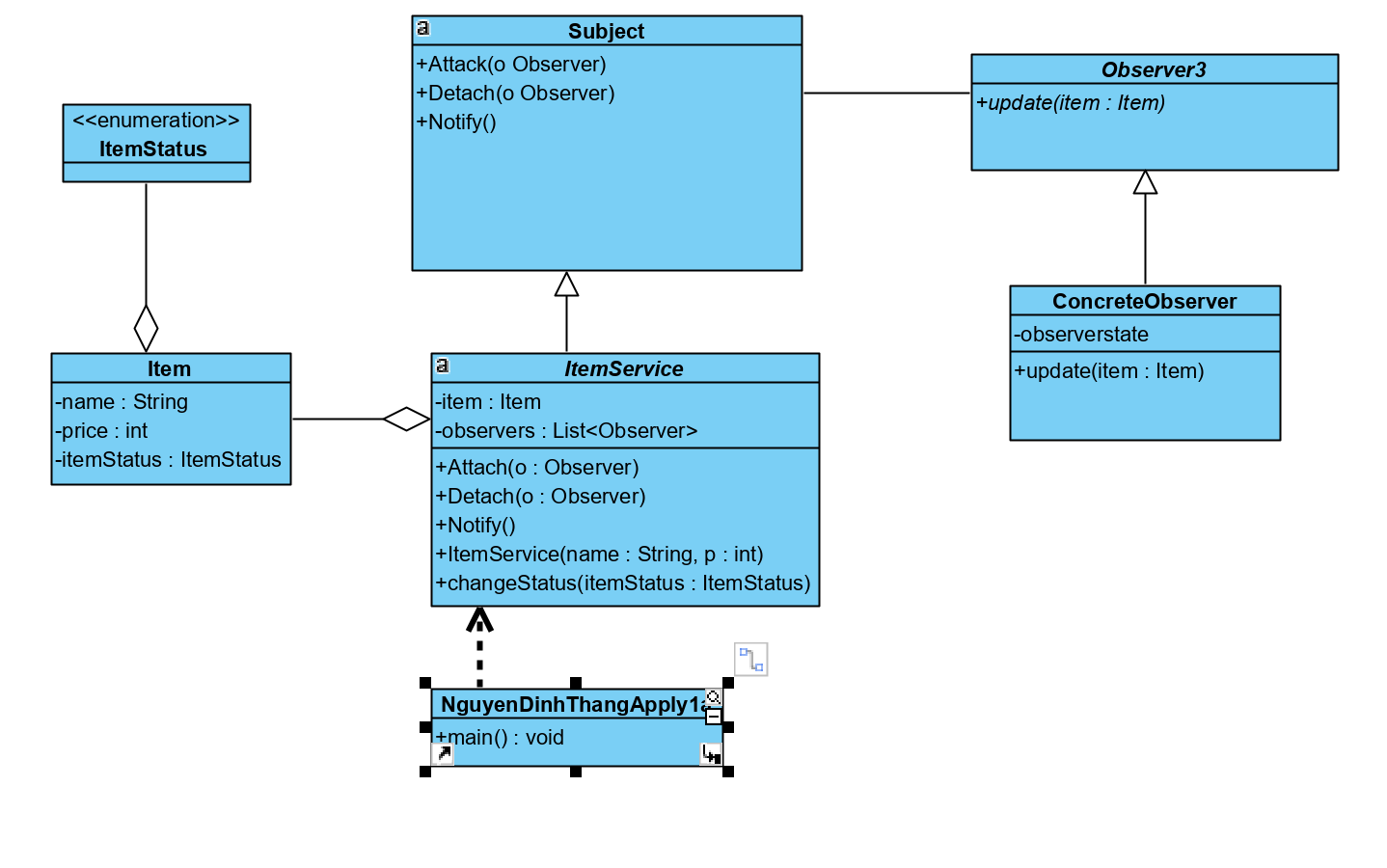
Khi thay đổi một đối tượng, yêu cầu thay đổi đối tượng khác và chúng ta không biết có bao nhiêu đối tượng cần thay đổi, những đối tượng này là ai.

Sử dụng trong ứng dụng broadcast-type communication.

Sử dụng để quản lý sự kiện (Event management).

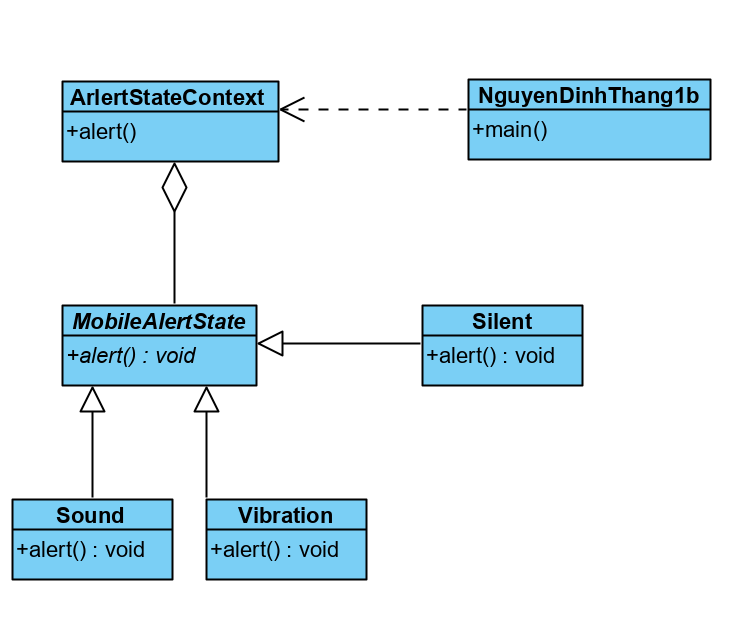
Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

* + - * Áp dụng



*Code trong file đính kèm*

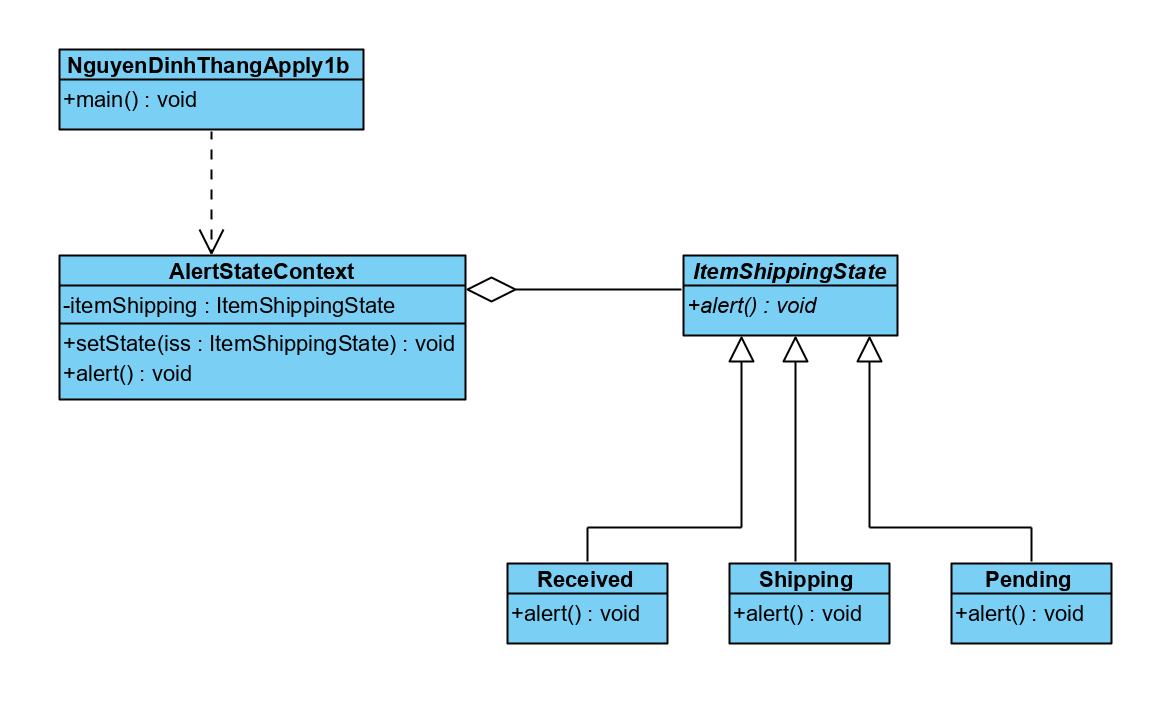
* 1. **State pattern**
* Thiết kế:



* Code:

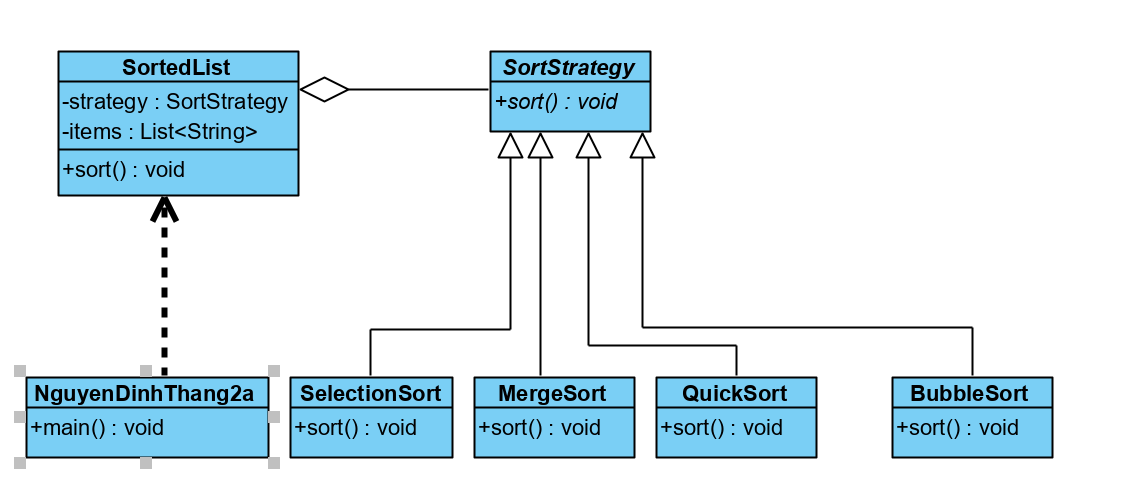
|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class AlertStateContext |  |
| Interface MobileAlertState |  |
| Class Silent |  |
| Class Sound |  |
| Class Vibration |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* + - **Áp dụng vào các trường hợp:**
      * Hành vi của 1 object phụ thuộc vào trạng thái của nó, và nó phải thay đổi hành vi theo thời gian phụ thuộc vào trạng thái của nó.
      * Có nhiều loại thao tác, trong đó, một số hàm có điều kiện phụ thuộc vào trạng thái của object. ( Các trạng thái này được chia ra làm nhiều nhánh và biểu diễn trong các lớp riêng biệt. Và một số hàm có điều kiện phụ thuộc vào trạng thái có cấu trúc tương tự nhau.)
      * Thường được áp dụng vào game: Một nhân vật trong game có nhiều trạng thái:
        1. Trạng thái bình thường.
        2. Trạng thái phẫn nộ
      * Mỗi trạng thái thì nhân vật có những thông số riêng, vd: ở trạng thái thịnh nộ thì nhân vật sẽ hóa to lên,biến hình, tăng sức tấn công, tăng năng lượng, tăng máu, đánh nhanh… mỗi cái là một chỉ số khác nhau… khi thay đổi trạng thái thì các thông số, hành vi của nhân vật cũng thay đổi.
    - Áp dụng



Code trong file đính kèm

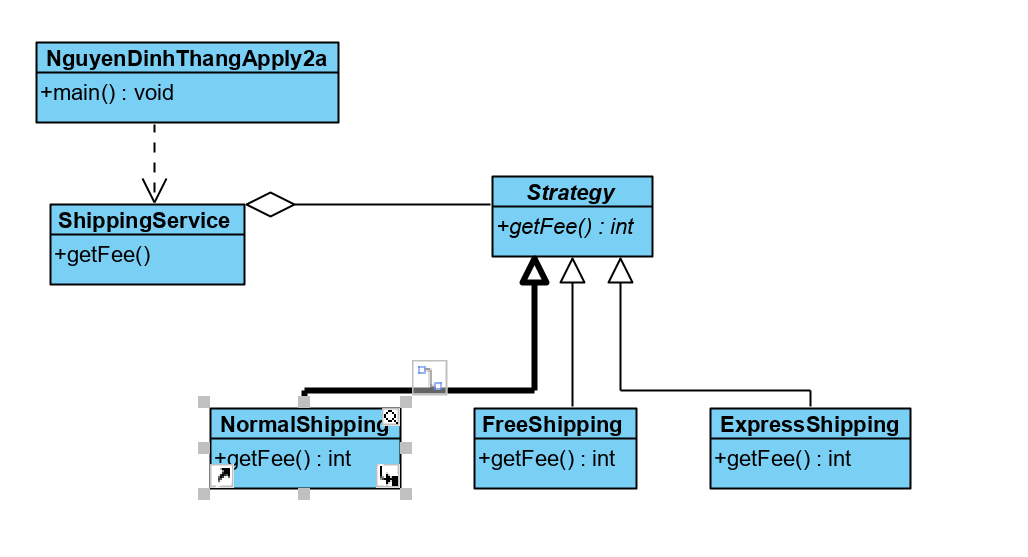
* 1. **Strategy**
* Thiết kế:



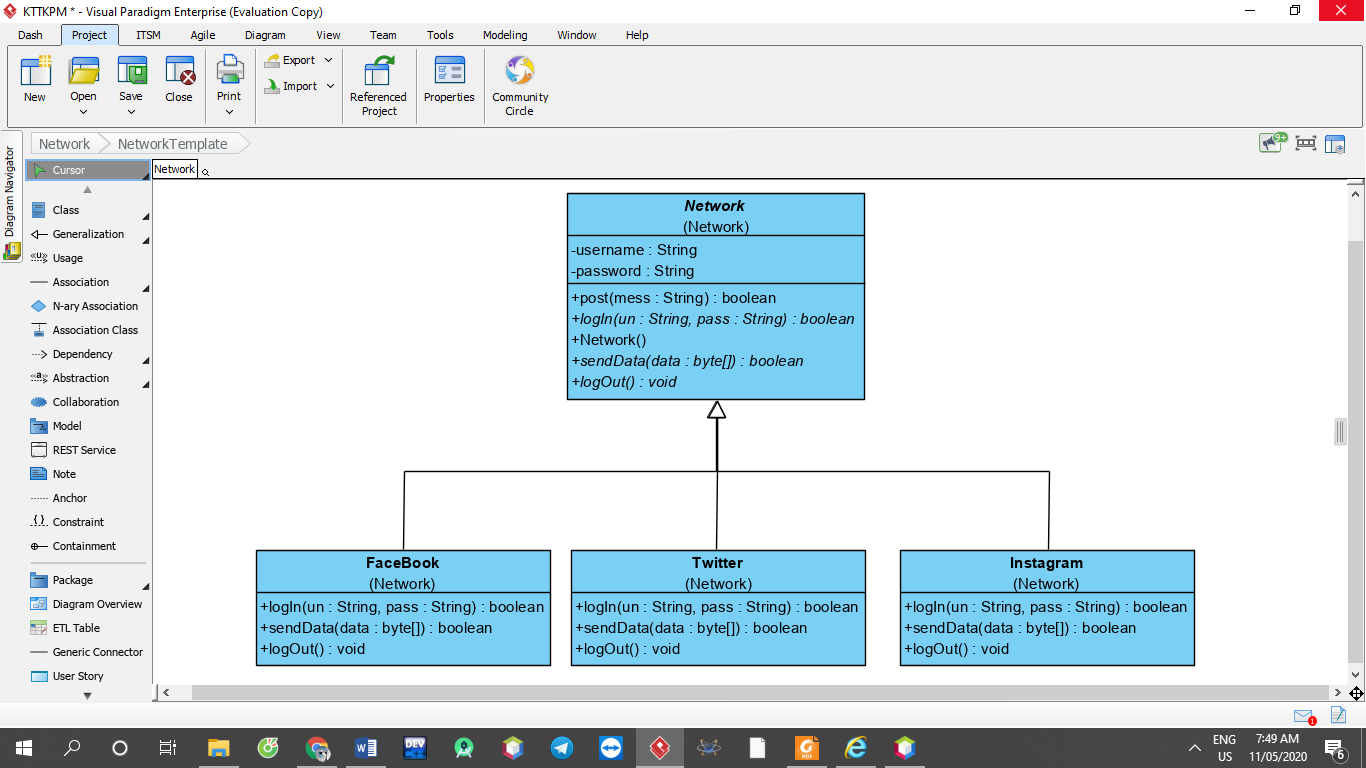
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface SortStrategy |  |
| Class SortedList |  |
| Class SelectionSort |  |
| Class QuickSort |  |
| Class MergeSort |  |
| Class BubbleSort |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
* Lợi ích:
  + Đảm bảo nguyên tắc Single responsibility principle (SRP) : một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện. Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó.
  + Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.
  + Cung cấp một sự thay thế cho kế thừa.
* Sử dụng
  + Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
  + Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
  + Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
  + Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.
* Áp dụng



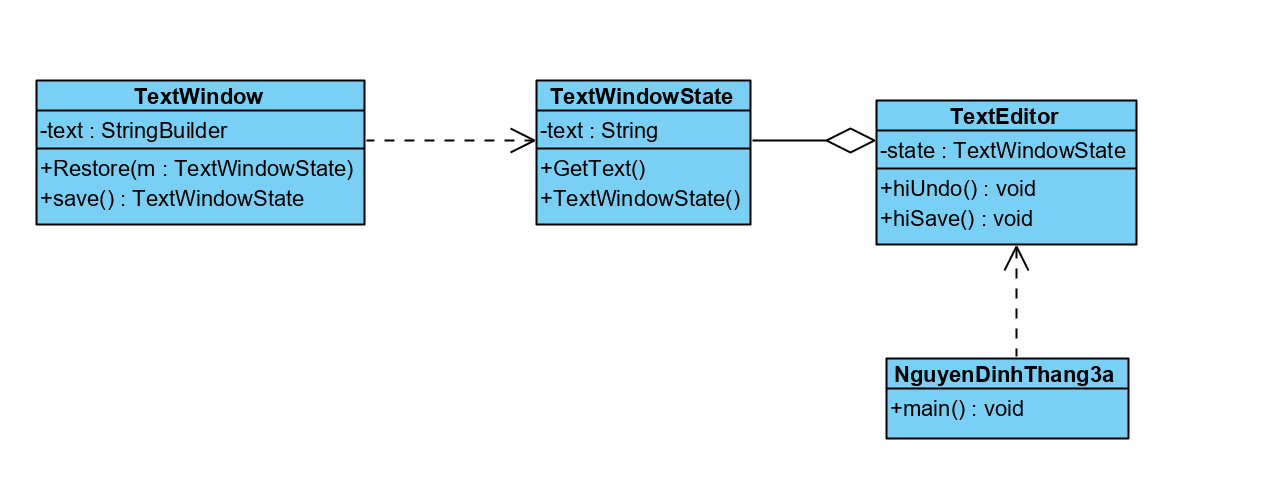
* 1. **Template Pattern**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class NetWork |  |
| Class Facebook |  |
| Class Instagram |  |
| Class Twitter |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

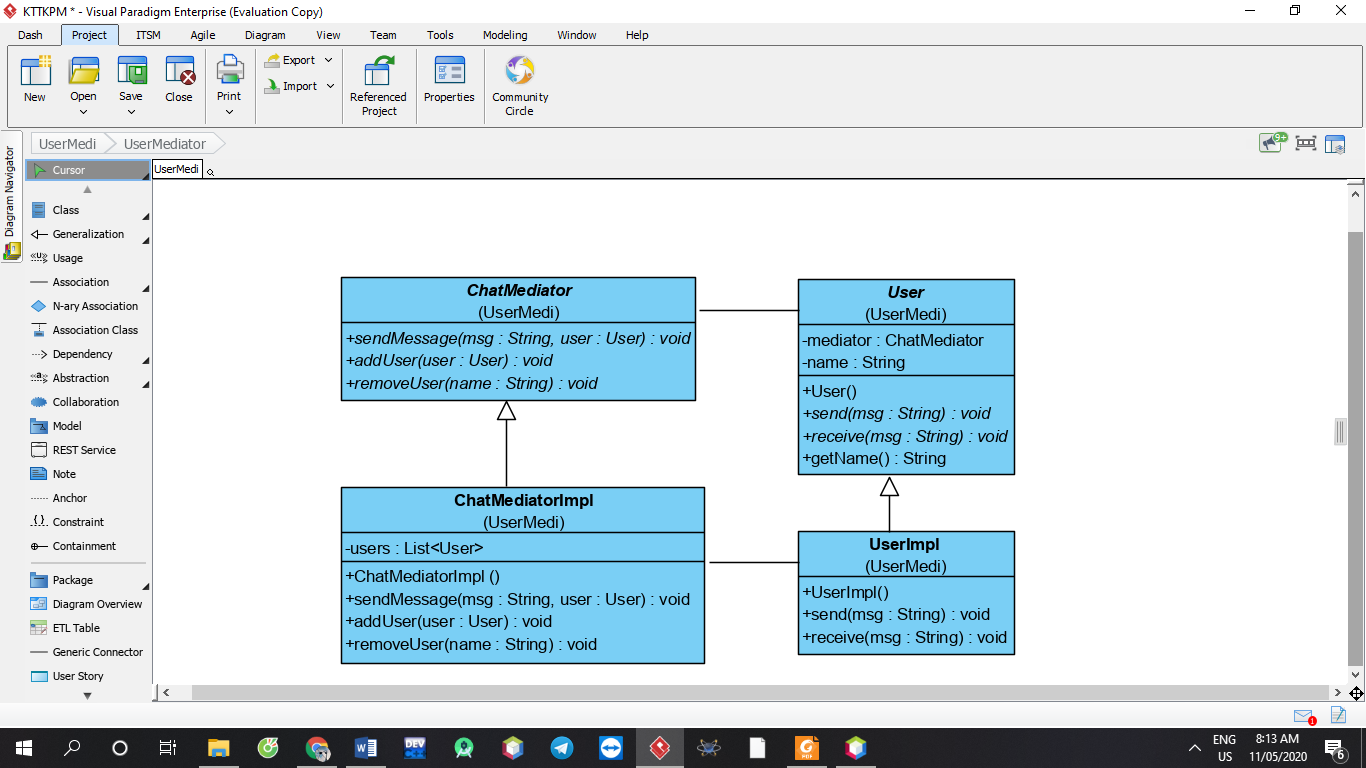
* Lợi ích:
  + Tái sử dụng code (reuse), tránh trùng lặp code (duplicate): đưa những phần trùng lặp vào lớp cha (abstract class).
  + Cho phép người dùng override chỉ một số phần nhất định của thuật toán lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.
* Sử dụng
  + Khi có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.
  + Mong muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.
  + Mong muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.
  + Các lớp cha có thể gọi các hành vi trong các lớp con của chúng một cách thống nhất (step by step).
  1. **Memento**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Class TextEditor |  |
| Class TextWindow |  |
| Class TextWindowState |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

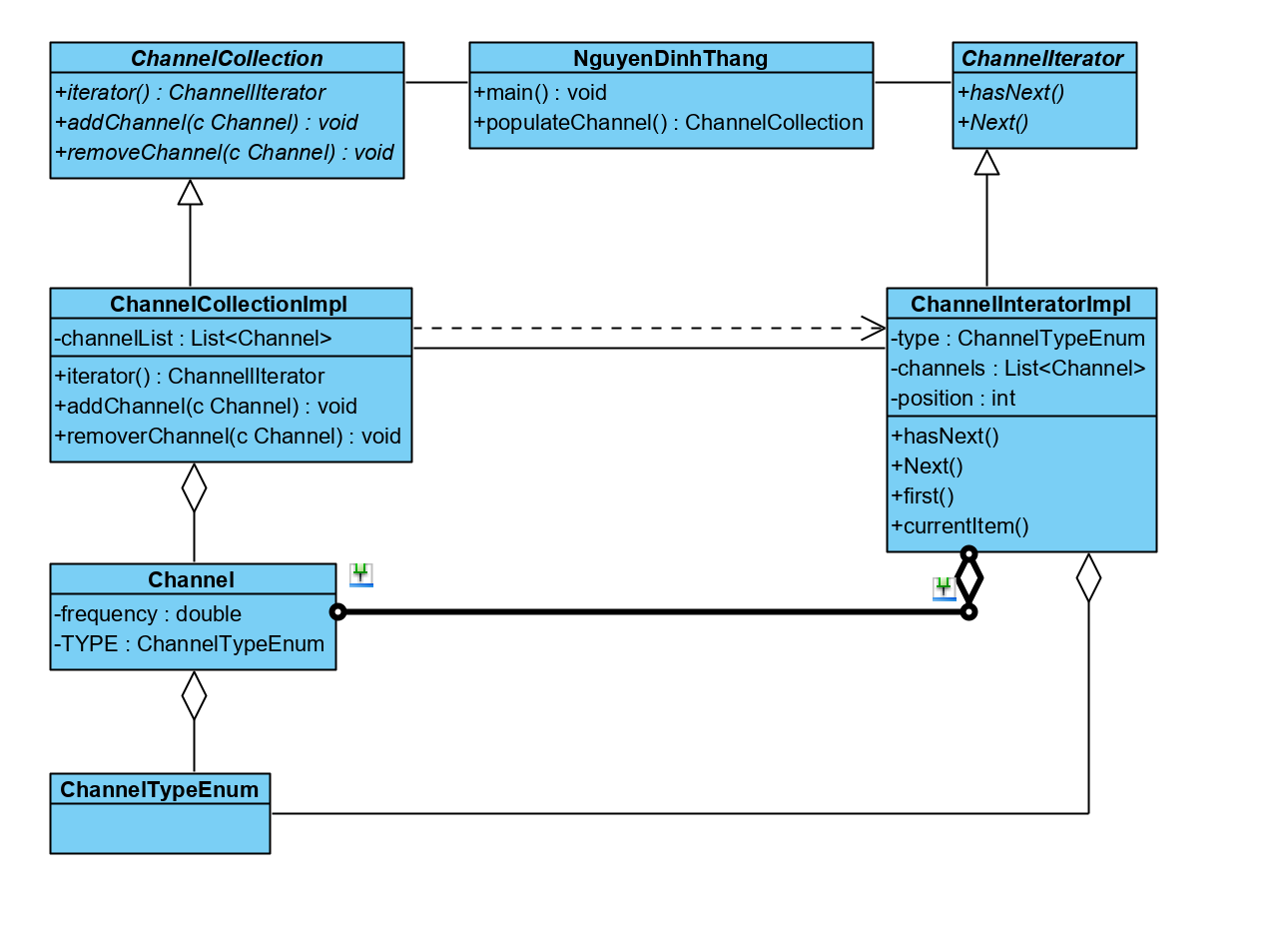
* **Kinh nghiệm**:
  + Lợi ích:
    - Bảo đảm nguyên tắc đóng gói: sử dụng trực tiếp trạng thái của đối tượng có thể làm lộ thông tin chi tiết bên trong đối tượng và vi phạm nguyên tắc đóng gói.
    - Đơn giản code của Originator bằng cách để Memento lưu giữ trạng thái của Originator và Caretaker quản lý lịch sử thay đổi của Originator.
  + Một số vấn đề cần xem xét khi sử dụng Memento Pattern:
* Khi có một số lượng lớn Memento được tạo ra có thể gặp vấn đề về bộ nhớ, performance của ứng dụng.
* Khó đảm bảo trạng thái bên trong của Memento không bị thay đổi.
* Sử dụng
  + Các ứng dụng cần chức năng cần Undo/ Redo: lưu trạng thái của một đối tượng bên ngoài và có thể restore/ rollback sau này.
  + Thích hợp với các ứng dụng cần quản lý transaction.
  1. **Mediator**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface ChatMediator |  |
| Class ChatMediatorImpl |  |
| Class User |  |
| Class UserImpl |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

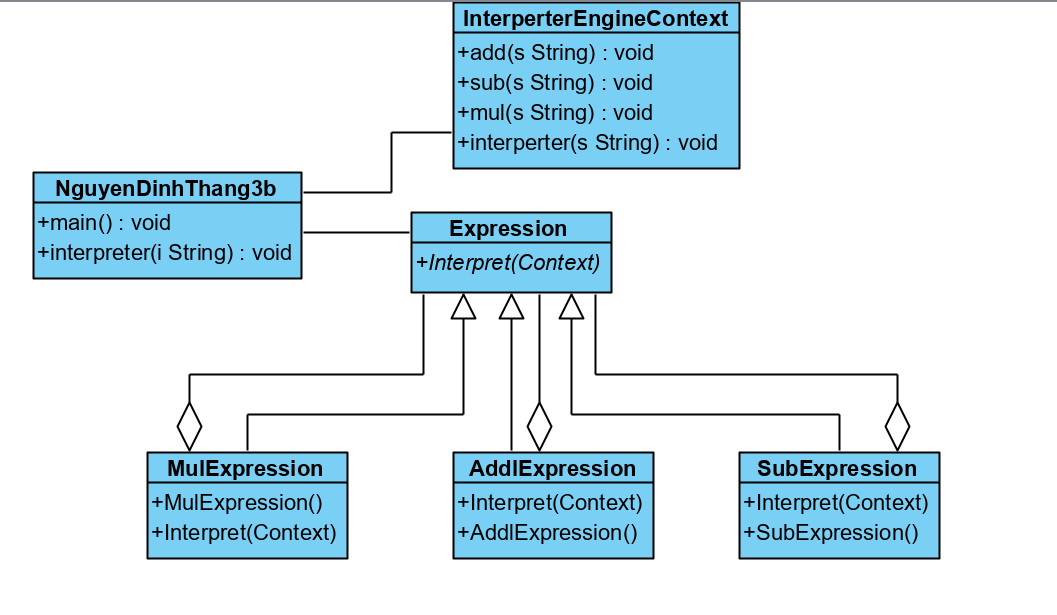
* **Kinh nghiệm**:  
  - Sử dụng:
  + Một tập các đối tượng giao tiếp theo những cách thức được xác định rõ ràng nhưng cách thức đó quá phức tạp. Sự phụ thuộc lẫn nhau giữa các đối tượng tạo ra kết quả là cách tổ chức không có cấu trúc và khó hiểu.
  + Tái sử dụng một đối tượng là khó bởi vì nó đề cập đến và giao tiếp với nhiều đối tượng khác.
  + Điều chỉnh hành vi giữa các lớp một cách dễ dàng, không cần chỉnh sửa ở nhiều lớp.
* **Điểm mạnh điểm yếu**
  + Nó giới hạn các lớp con. Một Mediator tập trung một hành vi chung mà nếu không hành vi đó sẽ được phân tán giữa các đối tượng (ví dụ về tháp điều khiển và các máy bay). Thay đổi hành vi này chỉ cần thay đổi lớp Mediator ; các lớp "đồng nghiệp"(Colleague class) có thể không cần thay đổi.
  + Nó tách riêng các lớp "đồng nghiệp".Mediator khuyến khích sự liên kết lỏng lẻo giữa các đồng nghiệp. Bạn có thể thay đổi và sử dụng lại các lớp đồng nghiệp và Mediator một cách độc lập.
  + Đơn giản hóa các giao thức đối tượng. Một mediator sẽ thay thế mối quan hệ nhiều- nhiều (many-to-many) giữa các lớp đồng nghiệp bằng quan hệ một- nhiều (one-to-many) giữa mediator và các lớp đồng nghiệp. Quan hệ một-nhiều-dễ hiểu, dễ duy trì và mở rộng.
  + Nó tóm tắt các đối tượng hợp tác, giao tiếp như thế nào. Điều đó có thể giúp làm rõ các đối tượng tương tác trong hệ thống như thế nào.
  + Nó tập trung kiểm soát.Mô hình Mediator tạo ra đối tượng hòa giải (mediator) tùy thuộc vào độ phức tạp của các tương tác. Bởi vì một mediator gói các giao thức, nó có thể phức tạp hơn bất kỳ một lớp "đồng nghiệp" nào. Điều này có thể làm cho mediator đó trở thành một khối khó bảo trì.
  1. **Interator**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface ChannelCollection |  |
| Interface ChannelIterator |  |
| Class Channel |  |
| Class ChannelCollectionImpl |  |
| Enum ChannelTypeEnum |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

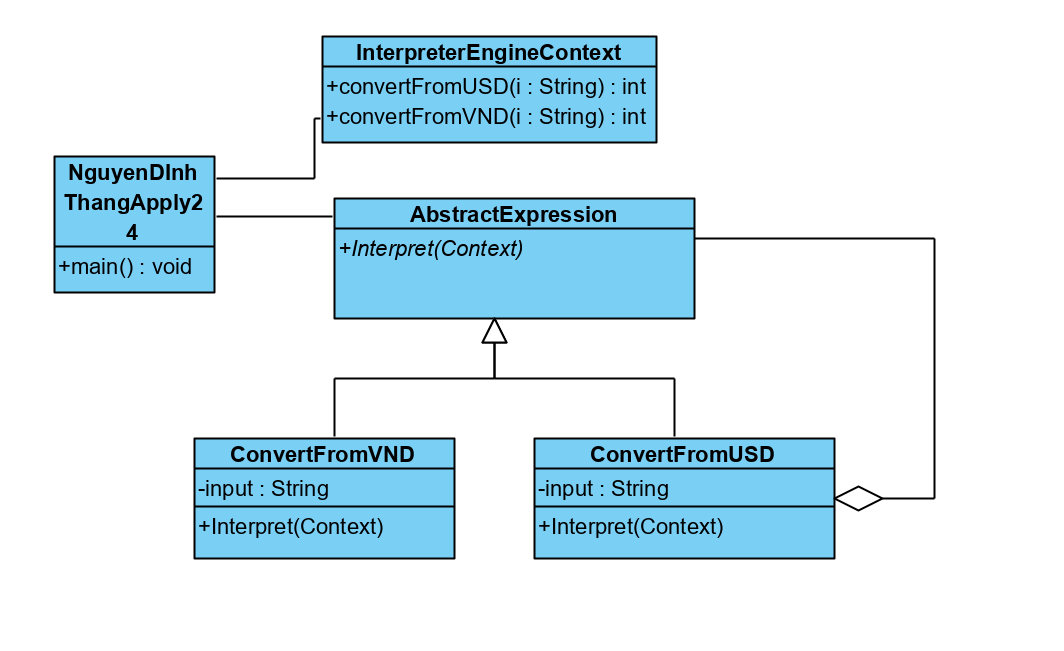
* **Kinh nghiệm**:
  + Một số lợi ích khi sử dụng Iterator Pattern:
    - Đảm bảo nguyên tắc Single responsibility principle (SRP) : chúng ta có thể tách phần cài đặt các phương thức của tập hợp và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ.
    - Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta có thể implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có mà không vi phạm bất cứ nguyên tắc gì.
    - Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.
  + Một số điểm cần xem xét khi sử dụng Iterator:
* Sử dụng iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của bộ sưu tập một cách trực tiếp.
* Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các collection đơn giản.
* Sử dụng
  + Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong nó.
  + Hỗ trợ truy xuất nhiều loại tập hợp khác nhau.
  + Cung cấp một interface duy nhất để duyệt qua các phần tử của một tập hợp.
  1. **Interpreter**
* Thiết kế:



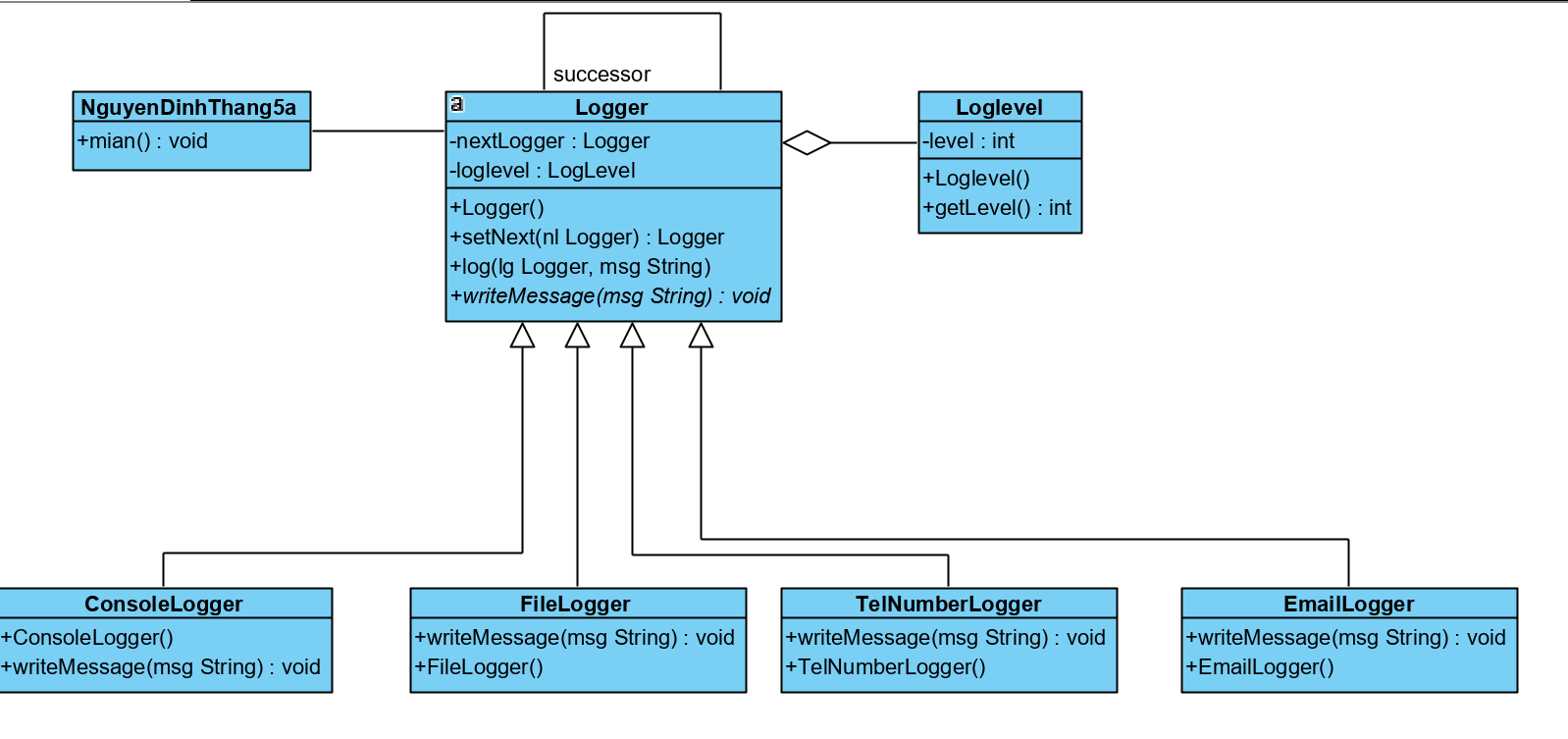
* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface Expression |  |
| Class InterpreterEngineContext |  |
| Class AddExpression |  |
| Class MultiplyExpression |  |
| Class SubtractExpression |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
  + Lợi ích:
    - Dễ dàng thay đổi và mở rộng ngữ pháp. Vì mẫu này sử dụng các lớp để biểu diễn các quy tắc ngữ pháp, chúng ta có thể sử dụng thừa kế để thay đổi hoặc mở rộng ngữ pháp. Các biểu thức hiện tại có thể được sửa đổi theo từng bước và các biểu thức mới có thể được định nghĩa lại các thay đổi trên các biểu thức cũ.
    - Cài đặt và sử dụng ngữ pháp rất đơn giản. Các lớp xác định các nút trong cây cú pháp có các implement tương tự. Các lớp này dễ viết và các phân cấp con của chúng có thể được tự động hóa bằng trình biên dịch hoặc trình tạo trình phân tích cú pháp.
  + Sử dụng:  
    Interpreter Pattern được sử dụng hiệu quả khi:
    - Bộ ngữ pháp đơn giản. Pattern này cần xác định ít nhất một lớp cho mỗi quy tắc trong ngữ pháp. Do đó ngữ pháp có chứa nhiều quy tắc có thể khó quản lý và bảo trì.
    - Không quan tâm nhiều về hiệu suất. Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.
    - Interpreter Pattern thường được sử dụng trong trình biên dịch (compiler), định nghĩa các bộ ngữ pháp, rule, trình phân tích SQL, XML, …
* Áp dụng



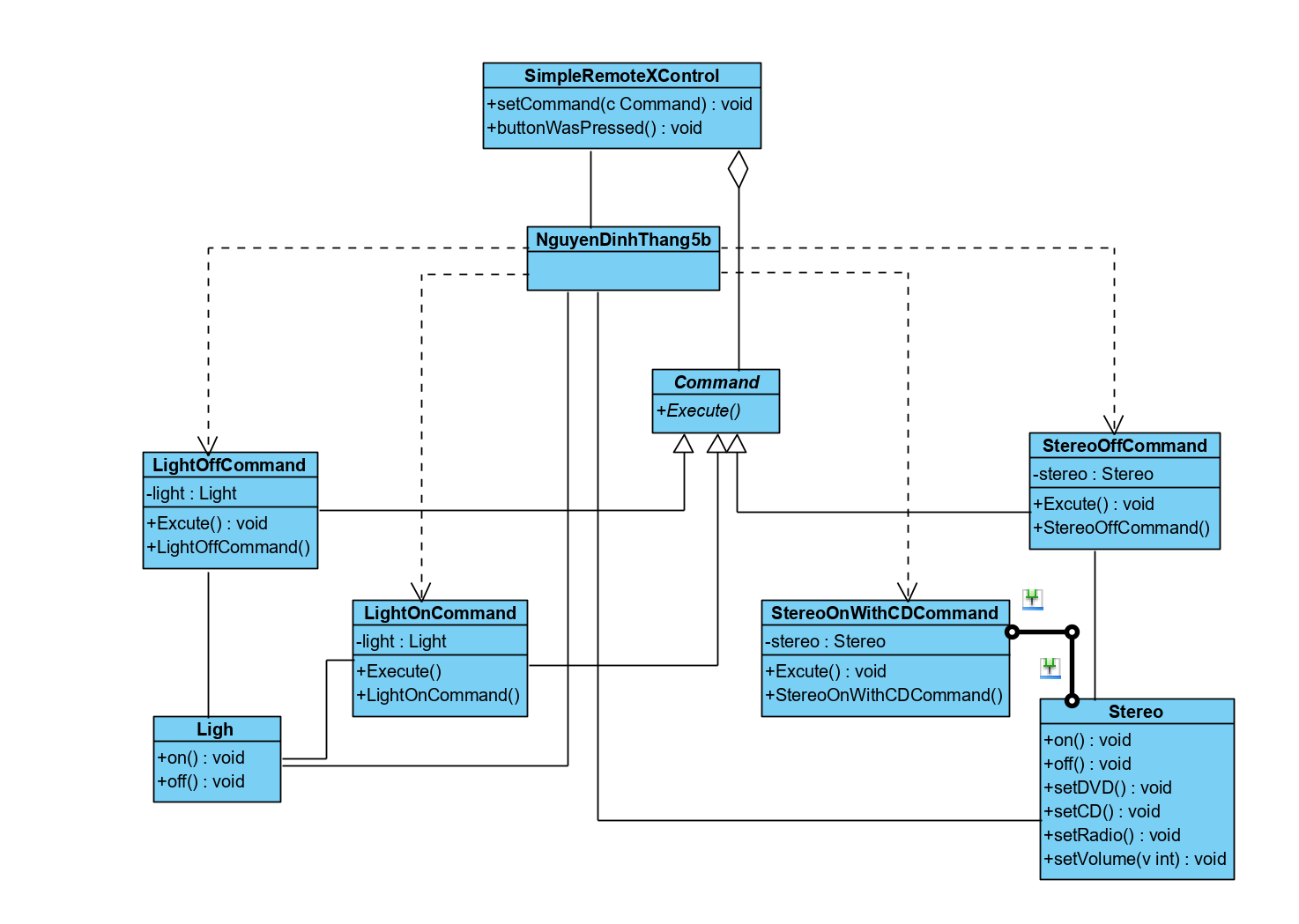
* 1. **Chain of Responsibility Pattern**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface AppLogger |  |
| Class ConsoleLogger |  |
| Class EmailLogger |  |
| Class FileLogger |  |
| Class Logger |  |
| Enum LogLevel |  |
| Class TelNumberLogger |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
  + Một số lợi ích khi sử dụng Iterator Pattern:
    - Giảm kết nối (loose coupling): Thay vì một đối tượng có khả năng xử lý yêu cầu chứa tham chiếu đến tất cả các đối tượng khác, nó chỉ cần một tham chiếu đến đối tượng tiếp theo. Tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu (sender) và các đối tượng nhận yêu cầu (receivers).
    - Tăng tính linh hoạt : đảm bảo Open/Closed Principle.
    - Phân chia trách nhiệm cho các đối tượng: đảm bảo Single Responsibility Principle.
    - Có khả năng thay đổi dây chuyền (chain) trong thời gian chạy.
    - Không đảm bảo có đối tượng xử lý yêu cầu.
* Sử dụng
  + - * Có nhiều hơn một đối tượng có khả thực xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
      * Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
      * Khi cần phải thực thi các trình xử lý theo một thứ tự nhất định..
      * Khi một tập hợp các đối tượng xử lý có thể thay đổi động: tập hợp các đối tượng có khả năng xử lý yêu cầu có thể không biết trước, có thể thêm bớt hay thay đổi thứ tự sau này.
  1. **Command**
* Thiết kế:



* Code:

|  |  |
| --- | --- |
| **Tên class** | **Code** |
| Interface Command |  |
| Class Light |  |
| Class AddExpression |  |
| Class LightOffCommand |  |
| Class LightOnCommand |  |
| Class SimpleRemoteControl |  |
| Class Stereo |  |
| Class StereoOffCommand |  |
| Class StereoOnWithCDCommand |  |
| Class NguyenDinhThang |  |
| Kết quả chạy code |  |

* **Kinh nghiệm**:
  + Lợi ích:
    - Dễ dàng thêm các Command mới trong hệ thống mà không cần thay đổi trong các lớp hiện có. Đảm bảo Open/Closed Principle.
    - Tách đối tượng gọi operation từ đối tượng thực sự thực hiện operation. Giảm kết nối giữa Invoker và Receiver.
    - Cho phép tham số hóa các yêu cầu khác nhau bằng một hành động để thực hiện.
    - Cho phép lưu các yêu cầu trong hàng đợi.
    - Đóng gói một yêu cầu trong một đối tượng. Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.
  + Sử dụng:
    - Khi cần tham số hóa các đối tượng theo một hành động thực hiện.
    - Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau.
    - Khi cần hỗ trợ tính năng undo, log, callback hoặc transaction.