**Báo cáo: Design Pattern trong lập trình**

**I. Lý thuyết:**

**1. Giới thiệu Design Pattern:**

Design pattern (mẫu thiết kế phần mềm) là những giải pháp đã được kiểm chứng cho các vấn đề phổ biến trong thiết kế phần mềm. Chúng cung cấp một phương pháp tiếp cận chuẩn hoá để giải quyết những thách thức trong phát triển phần mềm, giúp code dễ bảo trì, mở rộng và tái sử dụng.

**2. Lợi ích của Design Pattern:**

* Tăng khả năng tái sử dụng mã nguồn.
* Giảm thiểu độ phức tạp cho hệ thống.
* Cải thiện khả năng giao tiếp trong nhóm bằng cách dùng chung một từ vựng thiết kế.
* Dễ bảo trì, mở rộng hệ thống phần mềm.

**3. Một số Design Pattern tiêu biểu:**

**3.1. Singleton Pattern:**

* Mục đích: Đảm bảo chỉ có một instance duy nhất của một lớp và cung cấp quyền truy cập toàn cục đến instance đó.
* Ứng dụng: Quản lý kết nối Database, hệ thống Logger, quản lý Config.
* Ví dụ Java:

public class Database {

private static Database instance = new Database();

private Database() {} // private constructor

public static Database getInstance() {

return instance;

}

}

* Ví dụ JavaScript:

const Database = (function () {

let instance;

function createInstance() {

return { name: "DB Connection" };

}

return {

getInstance: function () {

if (!instance) {

instance = createInstance();

}

return instance;

}

};

})();

**3.2. Factory Method Pattern:**

* **Mục đích**: Cho phép tạo đối tượng mà không chỉ rõ lớp cụ thể của đối tượng đó.
* **Ứng dụng**: Hệ thống giao diện người dùng, hệ thống khởi tạo đa dạng các loại tài khoản, sản phẩm.
* **Ví dụ Java**:

abstract class Dialog {

public void renderWindow() {

Button okButton = createButton();

okButton.render();

}

public abstract Button createButton();

}

class WindowsDialog extends Dialog {

public Button createButton() {

return new WindowsButton();

}

}

**3.3. Observer Pattern:**

* **Mục đích**: Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng, để khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc được thông báo.
* **Ứng dụng**: Event system, các ứng dụng UI.
* **Ví dụ JavaScript**:

class Subject {

constructor() {

this.observers = [];

}

subscribe(observer) {

this.observers.push(observer);

}

notify(data) {

this.observers.forEach(observer => observer.update(data));

}

}

class Observer {

update(data) {

console.log('Observer received:', data);

}

}

const subject = new Subject();

const observer1 = new Observer();

subject.subscribe(observer1);

subject.notify('Hello Observers!');

**3.4. Strategy Pattern:**

* **Mục đích**: Cho phép lựa chọn thuật toán khi chạy chương trình.
* **Ứng dụng**: Các hệ thống tính toán, hệ thống thanh toán linh hoạt.
* **Ví dụ Java**:

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class PaypalPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

**3.5. Decorator Pattern:**

* **Mục đích**: Thêm chức năng cho đối tượng một cách linh hoạt mà không thay đổi cấu trúc bên trong.
* **Ứng dụng**: Trang trí UI components, thêm tính năng runtime cho đối tượng.
* **Ví dụ JavaScript**:

function coffee() {

return "Coffee";

}

function withMilk(coffeeFunc) {

return function() {

return coffeeFunc() + " + Milk";

}

}

const milkCoffee = withMilk(coffee);

console.log(milkCoffee()); // Coffee + Milk

**4.So sánh các pattern:**

|  |  |  |
| --- | --- | --- |
| Pattern | Mục đích chính | ứng dụng phổ biến |
| Singleton | Một instance duy nhất | Kết nối DB, Logger |
| Factory Method | Tạo đối tượng không chỉ rõ lớp cụ thể | UI, sản phẩm |
| Observer | Thông báo thay đổi trạng thái | Event-driven systems |
| Strategy | Thay đổi thuật toán runtime | Hệ thống thanh toán, tính toán |
| Decorator | Thêm tính năng cho đối tượng động | UI Component, middleware |

**II**. **Singleton Pattern và kịch bản:**

**1. Singleton Pattern**

### 1.1 Định nghĩa

Singleton là một mẫu thiết kế thuộc nhóm Creational, đảm bảo rằng một lớp chỉ có duy nhất một instance (đối tượng) tồn tại trong toàn bộ chương trình, đồng thời cung cấp một phương thức truy cập toàn cục đến nó.

**2. Kịch bản thực tế nâng cao: Logger (Hệ thống ghi log duy nhất)**

### 2.1 Mô tả kịch bản

Trong một hệ thống ứng dụng lớn, tất cả các hành động, lỗi, hoặc thông tin cần ghi log phải được chuyển về một nơi duy nhất để đảm bảo kiểm soát và thống nhất. Do đó, ta chỉ nên có **một Logger duy nhất** cho toàn bộ hệ thống. Mỗi khi có log mới, Logger sẽ:

* In thông tin có thời gian cụ thể
* Đánh số thứ tự log
* Đảm bảo rằng chỉ một instance Logger tồn tại

**3. Giải thích đoạn mã**

* **Singleton đảm bảo** rằng chỉ một đối tượng Logger được khởi tạo trong toàn bộ ứng dụng.
* Phương thức getInstance() luôn trả về cùng một đối tượng.
* Mỗi khi gọi log(), hệ thống sẽ:
  + Ghi thời gian hiện tại
  + Tăng số lượng log (logCount)
  + In ra nội dung log với thông tin chi tiết

**4. Kết luận**

Singleton Pattern rất hữu ích khi chỉ nên tồn tại một đối tượng duy nhất trong toàn bộ hệ thống. Với kịch bản Logger, mẫu thiết kế này giúp đảm bảo tất cả ghi log đều được thống nhất, kiểm soát và dễ mở rộng (thêm ghi ra file, thêm cấp độ log: INFO, WARNING, ERROR...).

**III. Factory Method Pattern và kịch bản:**

**1. Giới thiệu**

Factory Method là một mẫu thiết kế thuộc nhóm Creational. Mục đích của nó là **ủy quyền việc tạo đối tượng** cho các lớp con, thay vì khởi tạo trực tiếp trong lớp cha. Điều này giúp tăng tính mở rộng mà không cần sửa mã nguồn cũ.

**2. Kịch bản: Giao diện người dùng đa nền tảng**

**2.1 Mô tả**

Ứng dụng cần tạo các thành phần giao diện (Button, Checkbox) phù hợp theo từng hệ điều hành. Factory Method sẽ giúp tạo ra các đối tượng cụ thể mà không cần thay đổi logic ứng dụng chính.

**3. Giải thích**

Tùy vào hệ điều hành mà sử dụng Factory tương ứng (WindowsDialog, MacDialog).

Factory tạo ra đối tượng phù hợp mà không cần sửa đổi mã chính.

**4. Kết luận**

* Cho phép một đối tượng (Subject) thông báo tự động đến nhiều đối tượng khác (Observers) khi có thay đổi.
* Giảm sự phụ thuộc giữa các thành phần hệ thống.
* Phù hợp với các ứng dụng như: hệ thống thông báo, giám sát dữ liệu, UI tự cập nhật.

**IV. Strategy Pattern và kịch bản:**

**1. Định nghĩa:**

Là một mẫu thiết kế hành vi (behavioral Pattern), cho phép định nghĩa một hoặc các thuật toán, đóng gói riêng biệt từng thuật toán và làm cho chúng có thể thay đổi lẫn nhau mà không ảnh hưởng đến các đối tượng sử dụng thuật toán đó. Vì vậy Strategy giúp tách rời hành vi ra khỏi object chính, làm cho việc thay đổi hành vi linh hoạt hơn mà không cần sửa đổi object.

**2. Ưu điểm:**

* Tách biến thuật toán: Các thuật toán được tách riêng thành các class độc lập.
* Dễ mở rộng: Thêm thuật toán mới mà không ảnh hưởng đến code cũ.
* **Linh hoạt runtime**: Có thể thay đổi hành vi của đối tượng trong khi chương trình đang chạy.

### Nhược điểm:

* **Tăng số lượng lớp**: Mỗi hành vi là một class riêng biệt, nếu quá nhiều sẽ làm chương trình phình ra.
* **Không cần thiết cho các hành vi quá đơn giản**: Nếu hành vi chỉ có 1-2 dòng không thay đổi, Strategy sẽ dư thừa.

**3. UML tổng quát:**

+----------------+ +--------------------+

| Context |<>----------------->| Strategy |

|----------------| |---------------------|

| - strategy | | + executeBehavior() |

| + setStrategy()| +---------------------+

| + perform() |

+----------------+

^

|

+------------------+------------------+

| |

+--------------------+ +--------------------+

| ConcreteStrategyA | | ConcreteStrategyB |

| + executeBehavior()| | + executeBehavior() |

+------------------------+ +------------------------+

**4. kịch bản thực tế : Al Game Behavior**

### 4.1 Mô tả kịch bản

Trong game AL, người chơi đóng vai nhân vật có thể:

* Di chuyển với nhiều kiểu: đi bộ, chạy, bay, bơi
* Tấn công kẻ địch: có xác suất thắng/thua
* Thay đổi cách di chuyển bất kỳ lúc nào
* Nhân vật có tối đa 3 mạng, thua trận sẽ bị trừ một mạng, hết mạng thì dừng trò chơi

### 4.2 Phân tích theo Strategy

* **Di chuyển**: là hành vi thay đổi theo runtime, dùng interface Behavior
* **Tấn công**: là hành vi chiến đấu, dùng interface AttackStrategy
* **Context**: class Character, chứa Behavior và AttackStrategy

**. Tổng kết**

Strategy Pattern là một mẫu thiết kế cực kỳ hữu ích trong việc xây dựng hệ thống có thể thay đổi hành vi linh động khi runtime.  
Trong lĩnh vực phát triển game, Strategy Pattern đặc biệt quan trọng để quản lý các hành vi AI phức tạp như di chuyển, tấn công, phòng thủ,... giúp cho game trở nên sinh động và dễ bảo trì.

**V. Observer pattern và kịch bản:**

**1. Giới thiệu**

Observer là một mẫu thiết kế thuộc nhóm Behavioral, cho phép một đối tượng Subject thông báo thay đổi đến tất cả các Observer đang đăng ký theo dõi nó.

**2. Kịch bản nâng cao: Ứng dụng tin tức**

**2.1 Mô tả**

Hệ thống tin tức có nhiều người dùng theo dõi. Khi có bản tin mới, toàn bộ người theo dõi sẽ được thông báo tự động.

**3. Giải thích**

* NewsAgency đóng vai trò Subject.
* User là Observer, đăng ký nhận thông báo.
* Khi gọi notifyAll(), tất cả các user đều nhận được tin tức mới.

**4. Tổng kết**

* Cho phép một đối tượng (Subject) thông báo tự động đến nhiều đối tượng khác (Observers) khi có thay đổi.
* Giảm sự phụ thuộc giữa các thành phần hệ thống.
* Phù hợp với các ứng dụng như: hệ thống thông báo, giám sát dữ liệu, UI tự cập nhật.

**VI. Decorator pattern và kịch bản:**

**1. Giới thiệu**

Decorator là mẫu thiết kế thuộc nhóm Structural. Nó cho phép thêm hành vi mới cho đối tượng mà không thay đổi cấu trúc lớp gốc.

**2. Kịch bản nâng cao: Gói cà phê tuỳ chọn**

**2.1 Mô tả**

Một cửa hàng cà phê cho phép khách hàng chọn cà phê cơ bản và thêm topping như sữa, đường, caramel,... Các topping có thể kết hợp theo ý muốn.

**3. Giải thích**

BasicCoffee là đối tượng gốc.

MilkDecorator, SugarDecorator bọc thêm hành vi cho coffee.

Dễ dàng thêm topping mới mà không cần sửa class gốc.

**4. Kết luận**

* Cho phép thêm chức năng mới động cho đối tượng mà không sửa đổi lớp gốc.
* Linh hoạt trong việc kết hợp nhiều hành vi mở rộng.
* Thường được dùng trong giao diện người dùng, xử lý văn bản, hoặc các hệ thống có nhiều tùy chọn kết hợp.