DESIGN PATTERNS
OOP PROJECT

Design Patterns là gì ?

- **Design Patterns không phải là một chức năng có sẵn trong C++, cũng không phải là điều bắt buộc trong lập trìn**h. Hiểu đơn giản, design patterns là **các quy tắc và giải pháp** được đúc kết trong lập trình, giúp chúng ta viết mã nguồn rõ ràng, dễ bảo trì và có thể tái sử dụng.

  + **Vậy hãy thắc mắc các quy tắc đó từ đâu ra ?** Design patterns xuất phát từ kinh nghiệm của các lập trình viên kỳ cựu. Họ đã nhận thấy những cách giải quyết hiệu quả cho các vấn đề thường gặp trong lập trình và chia sẻ những giải pháp này dưới dạng các mẫu thiết kế để mọi người áp dụng.

  + **Bonus** : Giống như lập trình hướng đối tượng (OOP), ngoài OOP còn có nhiều mô hình lập trình khác. Tuy nhiên, OOP đã trở nên phổ biến vì các quy tắc và nguyên lý của nó rất hữu ích trong việc tổ chức và quản lý mã nguồn phức tạp.

- **Có 3 loại design patterns chính** :

  + **Creational Patterns (Mẫu khởi tạo):** Giúp quản lý quá trình khởi tạo các đối tượng trong game Mario một cách linh hoạt và tiết kiệm tài nguyên.

    *__**Ví dụ**__: *Singleton* , *Factory*, *Prototype*.

  + **Structural Patterns (Mẫu cấu trúc):** Tổ chức và xây dựng mối quan hệ giữa các đối tượng trong hệ thống của Mario:

    *__**Ví dụ**__: *Adapter , Decorator, Composite*.

  + **Behavioral Patterns (Mẫu hành vi):** Quản lý cách các đối tượng trong game Mario tương tác và phối hợp với nhau.

    *__**Ví dụ**__: *Observer, Strategy, Command*.

- Vậy game Mario mình sẽ **sử dụng design pattern nào ?**
  (Mình có thể áp dụng 8 design patterns dưới đây)

  + **CREATIONAL PATTERNS :**

    * **Singleton**:

      ~ Yêu cầu: Đảm bảo chỉ có một bản thể duy nhất của các đối tượng quản lý tài nguyên như bộ điều khiển game hoặc âm thanh.

      ~ Áp dụng: Tạo một lớp GameManager hoặc SoundController dưới dạng Singleton để quản lý trạng thái game hoặc điều khiển âm thanh, đảm bảo sự nhất quán trong toàn bộ game.

    * **Factory**:

      ~ Yêu cầu: Tạo đối tượng động cho các nhân vật (Mario, Luigi, kẻ thù) và các vật phẩm (Mushroom, Coin, FireFlower).

      ~ Áp dụng: Sử dụng Factory để tạo ra các nhân vật hoặc vật phẩm dựa trên yêu cầu của màn chơi hoặc cấp độ, giúp thêm hoặc thay đổi loại đối tượng mà không sửa mã chính.

    * **Prototype**:

      ~ Yêu cầu: Tạo bản sao nhanh của các đối tượng có cấu trúc tương tự, như nhiều kẻ thù cùng loại trong cùng màn chơi.

      ~ Áp dụng: Dùng Prototype cho các đối tượng như Enemy hoặc Item để sao chép nhanh chóng mà không cần khởi tạo lại từ đầu, giúp tối ưu hóa tài nguyên và hiệu suất.

+ **STRUCTURAL PATTERNS:**

* **Decorator**:

~ Yêu cầu: Thêm các thuộc tính hoặc năng lực đặc biệt cho nhân vật, ví dụ khi Mario thu thập được vật phẩm (như FireFlower để bắn lửa hoặc Mushroom để lớn hơn).

~ Áp dụng: Sử dụng Decorator để trang bị cho Mario các năng lực mới mà không thay đổi lớp gốc của Mario. Decorator giúp thêm hoặc gỡ bỏ năng lực của Mario một cách linh hoạt khi nhặt hoặc mất vật phẩm.

* **Composite**:

~ Yêu cầu: Tổ chức các đối tượng môi trường như cây cối, bụi cỏ, khối gạch thành các nhóm để xử lý đồng loạt.

~ Áp dụng: Dùng Composite để nhóm các đối tượng môi trường trong màn chơi, giúp dễ dàng quản lý và cập nhật toàn bộ nhóm đối tượng cùng một lúc mà không cần thao tác riêng từng đối tượng.

+ **BEHAVIORAL PATTERNS :**

* **Observer**:

~ Yêu cầu: Quản lý các sự kiện và tương tác giữa các đối tượng trong game, chẳng hạn khi Mario ăn nấm hoặc khi điểm số thay đổi.

~ Áp dụng: Dùng Observer để cập nhật các đối tượng khác (như giao diện người chơi) khi có thay đổi về trạng thái của Mario, như cập nhật điểm số hoặc số mạng còn lại.

* **Command**:

~ Yêu cầu: Đóng gói (Encapsulation mới học) các hành động của Mario (như di chuyển, nhảy, bắn).

~ Áp dụng: Dễ dàng điều khiển các method của Mario (ví dụ Mario.GoLeft(), Mario.GoRight()...)

* **State**:

~ Yêu cầu: Quản lý trạng thái của Mario (đứng yên, đi bộ, nhảy, bị thương).

~ Áp dụng: Tạo các trạng thái như StandingState, JumpingState để thay đổi hành vi của Mario dựa trên trạng thái hiện tại, giúp quản lý hành vi một cách linh hoạt và dễ mở rộng.

THE END OF THE REPORT

Khoan, vẫn chưa hiểu ?

Vậy thì tham khảo nốt cái dưới này đi 😊

- **Factory Pattern :** ta có thể sử dụng Factory Pattern để tạo ra các đối tượng như kẻ thù hoặc vật phẩm. Thay vì gọi từng hàm khởi tạo riêng cho **Goomba**, **Koopa**, hoặc **Mushroom**, ta có thể dùng một **EnemyFactory** hoặc **ItemFactory**. Mỗi khi cần tạo đối tượng mới, factory sẽ quyết định kiểu đối tượng dựa trên tham số và trả về đúng loại đó.
  + **EnemyFactory** tạo ra **Goomba** hoặc **Koopa** dựa trên điều kiện của màn chơi.
  + **ItemFactory** tạo ra **Mushroom** hoặc **Star** dựa vào loại vật phẩm mà Mario cần.

- **Singleton Pattern :** Ta có thể sử dụng Singleton Pattern để quản lý các tài nguyên dùng chung trong game, như âm thanh hoặc trạng thái của trò chơi. Singleton đảm bảo chỉ có một bản thể duy nhất của các đối tượng này để tránh lãng phí tài nguyên và đảm bảo tính nhất quán trong toàn bộ game.
  + **GameManager**: Singleton đảm bảo có duy nhất một GameManager để theo dõi và quản lý trạng thái của trò chơi (bắt đầu, tạm dừng, kết thúc).
  + **SoundController**: Đảm bảo có một SoundController duy nhất để điều khiển âm thanh, giúp giảm thiểu việc tạo các bản thể âm thanh dư thừa và tránh xung đột.

- **Prototype Pattern :** giúp sao chép nhanh các đối tượng giống nhau mà không phải khởi tạo lại toàn bộ từ đầu. Điều này đặc biệt hữu ích khi có nhiều đối tượng kẻ thù hoặc vật phẩm xuất hiện thường xuyên và có cấu trúc tương tự. (nói ngắn gọn là clone)
  + **EnemyPrototype:** Khi có nhiều kẻ thù cùng loại (như nhiều Goomba hoặc Koopa) trong một màn chơi, EnemyPrototype cho phép sao chép một kẻ thù mẫu đã có sẵn để tạo các kẻ thù mới mà không cần khởi tạo lại toàn bộ.
  + **ItemPrototype:** Khi Mario đi qua nhiều khối gạch có vật phẩm giống nhau (như đồng xu hoặc nấm), ItemPrototype cho phép sao chép các vật phẩm này để tiết kiệm tài nguyên và thời gian xử lý.

- **Decorator Pattern :** cho phép thêm tính năng mới cho Mario mà không thay đổi lớp gốc. Khi Mario thu thập vật phẩm, tính năng mới có thể được thêm vào một cách linh hoạt.
  + **FirePowerDecorator:** Thêm khả năng bắn lửa khi Mario thu thập hoa lửa.
  + **StarPowerDecorator:** Thêm khả năng bất tử tạm thời khi Mario thu thập sao.

- **Composite Pattern :** Composite Pattern được dùng để quản lý các nhóm đối tượng trong game. Thay vì xử lý từng đối tượng môi trường riêng lẻ (như cây, bụi, khối gạch), Composite giúp nhóm chúng lại và xử lý dễ dàng hơn.
  + **EnvironmentComposite:** Nhóm các đối tượng như cây cối, khối gạch, bụi cỏ để có thể xử lý đồng bộ khi vẽ hoặc cập nhật vị trí.

- **Observer Pattern :** có thể được sử dụng để cập nhật các trạng thái trong game theo sự kiện. Khi một sự kiện xảy ra (như Mario thu thập vật phẩm hoặc đánh bại kẻ thù), tất cả các đối tượng theo dõi sự kiện đó sẽ được thông báo và cập nhật trạng thái của chúng.
  + **ScoreObserver:** Cập nhật điểm số khi Mario thu thập đồng xu hoặc tiêu diệt kẻ thù.
  + **LifeObserver:** Giảm số mạng còn lại khi Mario bị trúng đòn từ kẻ thù.

- **Command Pattern :** giúp quản lý các hành động của Mario (như di chuyển, nhảy, bắn) bằng cách đóng gói mỗi hành động thành các đối tượng riêng biệt. Điều này giúp quản lý hành vi một cách linh hoạt và dễ dàng mở rộng.
  + **JumpCommand:** Đóng gói hành động nhảy của Mario để dễ dàng thực hiện khi người chơi nhấn phím tương ứng.
  + **MoveLeftCommand** và **MoveRightCommand:** Đóng gói hành động di chuyển trái và phải của Mario, giúp dễ dàng quản lý và điều chỉnh các hành vi.

- **State Pattern :** quản lý trạng thái của Mario để thay đổi hành vi dựa trên trạng thái hiện tại (như đứng, nhảy, hoặc bị thương). Khi Mario chuyển trạng thái, hành vi của anh ấy cũng thay đổi mà không cần viết lại logic chính.
  + **StandingState:** Trạng thái đứng yên khi Mario không di chuyển.
  + **JumpingState:** Trạng thái nhảy khi Mario đang ở trên không.
  + **PowerUpState:** Trạng thái tăng sức mạnh khi Mario nhận được nấm hoặc hoa lửa, giúp anh ấy mạnh hơn trong một khoảng thời gian.