**DESIGN PATTERN CREATIONAL**

**Loại:**

[Creational Pattern](https://gpcoder.com/category/design-pattern/creational-pattern/) (nhóm khởi tạo – 4 mẫu) gồm: Factory Method, Abstract Factory, Prototype, Singleton.

**Định nghĩa:**

Là mẫu thiết kế trong lập trình hướng đối tượng thuộc nhóm Creational (nhóm khởi tạo), tập trung vào việc tạo đối tượng một cách linh hoạt và hiệu quả. Nó cung cấp các phương pháp tiêu chuẩn để tạo đối tượng, che giấu quá trình tạo đối tượng khỏi client và giảm sự ràng buộc giữa các lớp.

(Tên pattern) giải quyết các vấn đề liên quan đến việc tạo đối tượng + (chỗ này ghép từng định nghĩa pattern)

+ Việc đảm bảo rằng chỉ có một thể hiện duy nhất của một lớp (Singleton)

+ Việc tạo ra đối tượng dựa trên các điều kiện khác nhau (Factory Method)

+ Việc tạo ra các đối tượng có liên quan với nhau mà không cần chỉ định rõ lớp cụ thể (Abstract Factory)

+ Việc tạo ra các đối tượng sử dụng một bản sao (Prototype)

**Lợi ích**

Tách biệt quy trình khởi tạo: Các design pattern creational giúp tách biệt quy trình khởi tạo đối tượng khỏi mã nguồn chính. Điều này cho phép bạn tạo ra đối tượng một cách linh hoạt, mà không phụ thuộc vào cách khởi tạo cụ thể.

Tăng tính linh hoạt: Các design pattern creational giúp tăng tính linh hoạt trong việc tạo đối tượng. Bạn có thể thay đổi cách tạo đối tượng mà không ảnh hưởng đến mã nguồn sử dụng đối tượng đó.

Giảm sự phụ thuộc: Các design pattern creational giảm sự phụ thuộc giữa lớp tạo đối tượng và lớp sử dụng đối tượng. Thay vì gọi trực tiếp hàm tạo đối tượng, bạn tương tác thông qua một giao diện hoặc một lớp trung gian. Điều này giúp mã nguồn trở nên dễ bảo trì, mở rộng và thay đổi.

Tăng khả năng kiểm thử: Các design pattern creational tạo điều kiện thuận lợi cho việc kiểm thử và gỡ lỗi. Bạn có thể dễ dàng thay thế các đối tượng tạo trong quá trình kiểm thử bằng các đối tượng giả lập hoặc đối tượng mock để kiểm tra các tình huống và kịch bản khác nhau.

Tăng khả năng tái sử dụng: Các design pattern creational thúc đẩy việc tái sử dụng mã nguồn. Thay vì tạo đối tượng trực tiếp, bạn sử dụng một lớp tạo đối tượng để tạo ra các phiên bản mới. Điều này giúp tăng khả năng tái sử dụng và giảm sự trùng lặp trong mã nguồn.

Tạo ra các đối tượng phức tạp: Các design pattern creational cho phép tạo ra các đối tượng phức tạp một cách có tổ chức. Bạn có thể tạo ra các đối tượng với cấu trúc phức tạp, các tham số khác nhau và các bước khởi tạo phức tạp mà không cần làm cho mã nguồn trở nên rối rắm.

**Ví dụ:**

**Factory Method:**

Ví dụ, trong một ứng dụng quản lý tài khoản ngân hàng. Trong ứng dụng này, chúng ta có nhiều loại tài khoản ngân hàng như tài khoản tiết kiệm, tài khoản vãng lai và tài khoản vay.

Thay vì tạo mới các đối tượng tài khoản trực tiếp bằng từ khóa "new" từ một lớp cụ thể, chúng ta có thể sử dụng Factory Method để tạo ra các đối tượng tài khoản dựa trên yêu cầu cụ thể của người dùng.

Ví dụ, chúng ta có một lớp trừu tượng là AccountCreator định nghĩa một phương thức tạo đối tượng là createAccount(). Các lớp con của AccountCreator sẽ triển khai phương thức này để tạo ra các đối tượng tài khoản cụ thể.

Trong trường hợp người dùng muốn tạo tài khoản tiết kiệm, chúng ta có một lớp AccountSavingsCreator triển khai AccountCreator và triển khai phương thức createAccount() để tạo ra một đối tượng Tài khoản tiết kiệm.

Trong trường hợp người dùng muốn tạo tài khoản vãng lai, chúng ta có một lớp AccountCheckingCreator triển khai AccountCreator và triển khai phương thức createAccount() để tạo ra một đối tượng Tài khoản vãng lai.

Trong trường hợp người dùng muốn tạo tài khoản vay, chúng ta có một lớp AccountLoanCreator triển khai AccountCreator và triển khai phương thức createAccount() để tạo ra một đối tượng Tài khoản vay.

Khi người dùng yêu cầu tạo một tài khoản, chúng ta chỉ cần sử dụng AccountCreator tương ứng với loại tài khoản mà người dùng muốn tạo. AccountCreator sẽ chịu trách nhiệm tạo ra đối tượng tài khoản cụ thể tương ứng với yêu cầu của người dùng.

**Abstract Factory:**

Ví dụ, một nhà sản xuất ô tô có thể có các Factory con như "LuxuryCarFactory" (nhà máy sản xuất ô tô sang trọng) và "SportsCarFactory" (nhà máy sản xuất ô tô thể thao). Mỗi Factory con sẽ cung cấp các cách tạo riêng biệt cho các bộ phận ô tô, bao gồm động cơ, lốp xe, hệ thống treo và nội thất.

Khi nhận được yêu cầu sản xuất một chiếc ô tô cụ thể, Abstract Factory sẽ được sử dụng để tạo ra các bộ phận phù hợp với dòng xe đó. Ví dụ, nếu yêu cầu là một chiếc ô tô sang trọng, nhà máy sẽ sử dụng LuxuryCarFactory để tạo ra động cơ mạnh mẽ, lốp xe cao cấp và nội thất sang trọng. Trong trường hợp một chiếc ô tô thể thao, SportsCarFactory sẽ được sử dụng để tạo ra động cơ mạnh mẽ hơn, lốp xe thích nghi với đường đua và nội thất thể thao.

Abstract Factory cho phép nhà sản xuất ô tô tạo ra các dòng sản phẩm khác nhau mà không cần thay đổi mã nguồn trong quá trình sản xuất. Nó giúp tăng tính linh hoạt và dễ dàng mở rộng trong việc sản xuất các mẫu xe mới và thay đổi thiết kế của xe.

**Prototype:**

Ví dụ trong một ứng dụng xử lý hình ảnh. Giả sử chúng ta có một ứng dụng cho phép người dùng vẽ và chỉnh sửa các hình ảnh. Trong ứng dụng này, mỗi hình ảnh có thể có nhiều thuộc tính như màu sắc, kích thước, độ trong suốt và hiệu ứng đặc biệt.

Khi người dùng muốn thêm một hình ảnh mới, chúng ta có thể sử dụng Prototype design pattern. Đầu tiên, chúng ta tạo một hình ảnh mẫu (prototype) ban đầu với các thuộc tính mặc định. Khi người dùng muốn thêm một hình ảnh mới, chúng ta sẽ sao chép (clone) hình ảnh mẫu này để tạo ra một bản sao mới.

Sau khi tạo bản sao, người dùng có thể tùy chỉnh các thuộc tính của hình ảnh mới theo ý muốn, chẳng hạn như thay đổi màu sắc, điều chỉnh kích thước, áp dụng hiệu ứng đặc biệt, và nhiều hơn nữa. Nhờ Prototype design pattern, chúng ta không cần tạo mới một đối tượng từ đầu với mỗi hình ảnh mới, mà chỉ cần sao chép hình ảnh mẫu và tùy chỉnh các thuộc tính theo nhu cầu.

Việc sử dụng Prototype design pattern trong trường hợp này giúp tiết kiệm tài nguyên vì không cần tạo mới các đối tượng lớn và phức tạp cho mỗi hình ảnh mới. Nó cũng cho phép người dùng dễ dàng tùy chỉnh các thuộc tính của hình ảnh mà không ảnh hưởng đến hình ảnh mẫu ban đầu.

**Singleton:**

Một đối tượng ghi log (logging) trong một ứng dụng. Trong ứng dụng này, chúng ta cần một cơ chế để ghi lại các thông tin quan trọng, lỗi hoặc hoạt động của hệ thống vào một tệp log.

Để đảm bảo rằng chỉ có duy nhất một đối tượng ghi log trong suốt vòng đời của ứng dụng và đảm bảo việc ghi log được thực hiện từ mọi phần của ứng dụng, chúng ta có thể sử dụng Singleton design pattern.

Lớp ghi log sẽ triển khai Singleton design pattern bằng cách đặt hàm khởi tạo là private để ngăn chặn việc tạo đối tượng từ bên ngoài. Nó sẽ cung cấp một phương thức tiếp cận toàn cục để truy xuất đối tượng duy nhất của ghi log.

Khi các phần khác của ứng dụng cần ghi log, chúng chỉ cần gọi phương thức tiếp cận của lớp ghi log để thực hiện việc ghi log. Nếu đối tượng ghi log chưa được tạo ra, nó sẽ được tạo ra lần đầu tiên và lưu trữ. Các lần gọi tiếp theo sẽ trả về đối tượng ghi log đã tồn tại.

Việc sử dụng Singleton design pattern trong trường hợp này đảm bảo rằng chúng ta không tạo ra nhiều đối tượng ghi log và đảm bảo rằng việc ghi log được thực hiện từ mọi phần của ứng dụng. Nó cũng giúp tiết kiệm tài nguyên và tăng tính nhất quán trong việc ghi log.

**DESIGN PATTERN STRUCTUAL**

**Loại:**

[Structural Pattern](https://gpcoder.com/category/design-pattern/structuaral-pattern/) (nhóm cấu trúc – 6 mẫu) gồm: Adapter, Bridge, Composite, Decorator, Facade, Proxy.

**Định nghĩa:**

Là mẫu thiết kế thuộc nhóm Structual (Cấu trúc), tập trung vào cách tổ chức các đối tượng và lớp thành các cấu trúc lớn hơn để giải quyết các vấn đề về sự kết hợp và tương tác giữa chúng.

(Tên pattern) giúp xây dựng các cấu trúc phức tạp bằng cách kết hợp các đối tượng và lớp theo cách linh hoạt và hiệu quả. Giúp giải quyết các vấn đề về + (chỗ này ghép từng định nghĩa pattern)

+ Mở rộng chức năng của một đối tượng mà không làm thay đổi cấu trúc hiện có (Decorator)

+ Phân chia một đối tượng lớn thành các phần nhỏ hơn và làm việc với các phần đó (Composite)

+ Che giấu các chi tiết triển khai phức tạp của một đối tượng (Proxy)

+ Cung cấp một giao diện thống nhất cho các đối tượng không tương thích (Adapter)

+ Tổ chức các đối tượng thành các cấu trúc phức tạp (Facade)

**Lợi ích**

Tăng tính linh hoạt và mở rộng: Các design pattern cấu trúc giúp tách biệt các thành phần khác nhau trong hệ thống, từ đó tăng tính linh hoạt và khả năng mở rộng của mã nguồn. Chúng giúp giảm sự phụ thuộc giữa các thành phần, cho phép thay đổi một thành phần mà không ảnh hưởng đến các thành phần khác.

Tăng khả năng tái sử dụng: Bằng cách áp dụng các design pattern cấu trúc, ta có thể tách biệt các phần của hệ thống thành các thành phần độc lập, có khả năng tái sử dụng cao. Điều này giúp giảm lặp lại mã nguồn và tăng khả năng sử dụng lại các thành phần trong các dự án khác.

Tăng tính bảo mật: Các design pattern cấu trúc cung cấp cách thức để tăng tính bảo mật của hệ thống. Chúng giúp giới hạn truy cập vào các thành phần quan trọng và đảm bảo rằng chỉ có những thành phần được ủy quyền mới có thể tương tác với chúng.

Cải thiện độ bảo dưỡng: Các design pattern cấu trúc giúp tạo ra mã nguồn có cấu trúc rõ ràng và dễ hiểu. Điều này làm cho việc bảo dưỡng và sửa lỗi trở nên dễ dàng hơn, vì các thành phần được phân chia rõ ràng và dễ dàng tìm hiểu.

Tăng hiệu suất và tối ưu hóa: Một số design pattern cấu trúc có thể được sử dụng để tối ưu hóa hiệu suất của hệ thống. Chúng giúp giảm độ phức tạp của mã nguồn, loại bỏ sự trùng lặp và tăng hiệu suất thực thi.

**Ví dụ:**

**Adapter:**

Việc kết nối các thiết bị điện tử có giao diện khác nhau vào một ổ cắm chung.

Giả sử bạn có một điện thoại di động mới với cổng USB-C, nhưng bạn muốn kết nối điện thoại này với ổ cắm điện có giao diện USB-A truyền thống. Trong trường hợp này, bạn có thể sử dụng một cái chuyển đổi (adapter) để chuyển đổi giao diện từ USB-C sang USB-A.

Adapter ở đây chịu trách nhiệm chuyển đổi giao diện USB-C của điện thoại thành giao diện USB-A mà ổ cắm điện sẽ nhận dạng và sử dụng. Adapter cung cấp một giao diện tương thích cho ổ cắm và cho phép điện thoại di động kết nối và sử dụng nguồn điện từ ổ cắm.

Trong trường hợp này, Adapter design pattern giúp tương thích giữa hai giao diện không tương thích và cho phép họ làm việc cùng nhau. Điện thoại di động và ổ cắm điện có thể được xem như các thành phần riêng lẻ và không cần biết về nhau. Adapter đóng vai trò là một lớp trung gian giữa chúng và cho phép họ tương tác một cách hợp lý.

Adapter design pattern trong ví dụ này giúp tái sử dụng ổ cắm điện USB-A hiện có, trong khi vẫn hỗ trợ các thiết bị mới có giao diện USB-C. Nó cũng giúp giảm sự phụ thuộc vào các giao diện cụ thể và tăng tính linh hoạt trong việc kết nối các thiết bị điện tử khác nhau.

**Bridge:**

Hệ thống quản lý hiển thị đồ họa trên nhiều nền tảng khác nhau, chẳng hạn như trên máy tính và điện thoại di động.

Trong hệ thống này, Bridge design pattern được sử dụng để tách rời phần hiển thị đồ họa (graphics) và nền tảng (platform) trong việc vẽ đồ họa trên các thiết bị khác nhau.

Cụ thể, có hai thành phần chính trong hệ thống này: Graphics (đồ họa) và Platform (nền tảng). Graphics chịu trách nhiệm về việc vẽ các hình ảnh, đường thẳng, văn bản và các yếu tố đồ họa khác. Platform đại diện cho các nền tảng cụ thể như Windows, macOS, Android hoặc iOS.

Sử dụng Bridge design pattern, chúng ta có thể tạo ra một cấu trúc linh hoạt để hiển thị đồ họa trên các nền tảng khác nhau. Lớp Graphics sẽ có một tham chiếu đến một đối tượng Platform, và các phương thức của Graphics sẽ gọi các phương thức của Platform để vẽ đồ họa trên nền tảng tương ứng.

Ví dụ, khi vẽ một hình tròn, lớp Graphics sẽ gọi phương thức vẽ hình tròn của Platform để thực hiện việc vẽ hình tròn trên nền tảng cụ thể như Windows, macOS hoặc Android.

Bridge design pattern cho phép chúng ta mở rộng hệ thống để hỗ trợ thêm nền tảng mới mà không cần sửa đổi sâu trong cấu trúc hiện có. Nó cũng giúp tách rời các yếu tố đồ họa khỏi nền tảng, tăng tính linh hoạt và khả năng mở rộng trong việc phát triển ứng dụng đa nền tảng.

**Composite:**

Cấu trúc cây của một tổ chức.

Giả sử bạn đang xây dựng một hệ thống quản lý tổ chức với các thành phần như phòng ban, nhóm làm việc và nhân viên. Mỗi phòng ban có thể chứa nhiều nhóm làm việc và mỗi nhóm làm việc có thể chứa nhiều nhân viên.

Trong trường hợp này, Composite design pattern sẽ được sử dụng để xây dựng cấu trúc cây phân cấp của tổ chức. Lớp gốc là tổ chức, và các lớp con bao gồm phòng ban, nhóm làm việc và nhân viên. Các lớp con này đều thực hiện một giao diện chung, cho phép chúng được xử lý như một thành phần đơn lẻ hoặc như một thành phần có cấu trúc hơn (composite).

Một lớp composite như phòng ban có thể chứa nhiều lớp composite con như các nhóm làm việc, trong khi các nhóm làm việc có thể chứa nhiều lớp lá như nhân viên. Cấu trúc cây này cho phép tổ chức được biểu diễn một cách phân cấp và có tính chất tự giống nhau ở mọi cấp độ.

Với Composite design pattern, bạn có thể thực hiện các hoạt động như thêm, xóa hoặc truy cập vào các thành phần trong cây tổ chức một cách thống nhất và thuận tiện. Bạn cũng có thể thực hiện các hoạt động trên toàn bộ cây tổ chức mà không cần xác định rõ loại thành phần cụ thể.

Composite design pattern giúp tạo ra một cấu trúc cây đơn giản và linh hoạt để quản lý tổ chức và thành phần của nó. Nó cũng tăng tính tái sử dụng và khả năng mở rộng của mã nguồn, vì bạn có thể thêm các loại thành phần mới vào cây tổ chức mà không cần thay đổi cấu trúc tổng quát của nó.

**Decorator:**

Quy trình tùy chỉnh và thêm topping cho bánh kem theo yêu cầu của khách hàng.

Giả sử bạn làm việc trong một cửa hàng bánh kem và khách hàng có thể tùy chỉnh và thêm các topping vào bánh kem của họ. Các topping có thể bao gồm kem tươi, trái cây tươi, kẹo, sô cô la và nhiều loại nguyên liệu khác.

Trong trường hợp này, Decorator design pattern được sử dụng để tạo ra các phiên bản tùy chỉnh của bánh kem với các topping khác nhau mà không cần tạo ra các lớp con riêng biệt cho mỗi sự kết hợp.

Ví dụ, bạn có thể có một bánh kem cơ bản với kem và lớp bánh. Nếu khách hàng muốn thêm topping như trái cây tươi, bạn có thể sử dụng Decorator design pattern để tạo ra một lớp Decorator cho topping trái cây và gắn nó vào bánh kem cơ bản. Tương tự, bạn có thể tạo ra các lớp Decorator khác cho các loại topping khác nhau như kẹo, sô cô la, kem tươi.

Khách hàng có thể tùy chọn và thêm bất kỳ số lượng topping và sắp xếp chúng theo ý muốn. Decorator design pattern giúp tái sử dụng các topping có sẵn và cho phép tạo ra những phiên bản tùy chỉnh của bánh kem mà không cần thay đổi cấu trúc ban đầu của nó.

Decorator design pattern giúp tách rời việc thêm topping từ bánh kem gốc, tạo ra một cấu trúc linh hoạt và dễ mở rộng cho việc tùy chỉnh bánh kem. Nó cũng cho phép thêm các topping mới mà không làm thay đổi các lớp hiện có, tăng tính linh hoạt và sự tái sử dụng trong thiết kế.

**Façade:**

Hệ thống thanh toán trực tuyến trong một ứng dụng mua sắm trực tuyến.

Giả sử bạn đang xây dựng một ứng dụng mua sắm trực tuyến cho phép người dùng thực hiện thanh toán trực tuyến khi mua hàng. Hệ thống thanh toán có thể bao gồm nhiều bước như xác nhận thông tin thanh toán, xử lý thanh toán qua cổng thanh toán, cập nhật trạng thái thanh toán và thông báo kết quả cho người dùng.

Trong trường hợp này, Facade design pattern được sử dụng để tạo ra một giao diện đơn giản và thuận tiện để người dùng có thể thực hiện thanh toán trực tuyến mà không cần quan tâm đến các bước phức tạp bên trong hệ thống.

Bạn có thể tạo ra một lớp Facade (giao diện) có phương thức thanh toán, và lớp này sẽ tương tác với các lớp phức tạp bên trong hệ thống như lớp xác nhận thông tin thanh toán, lớp xử lý thanh toán và lớp cập nhật trạng thái. Giao diện đơn giản này sẽ cung cấp các phương thức cần thiết cho người dùng để thực hiện thanh toán.

Khi người dùng gọi phương thức thanh toán trên giao diện Facade, nó sẽ xử lý các bước phức tạp bên trong hệ thống và trả về kết quả cho người dùng. Người dùng không cần quan tâm đến việc làm việc với từng lớp phức tạp riêng lẻ trong hệ thống thanh toán.

Facade design pattern giúp che giấu sự phức tạp của hệ thống thanh toán và cung cấp một giao diện đơn giản cho người dùng. Nó giảm sự phụ thuộc và sự rườm rà của việc tương tác với các lớp phức tạp, cung cấp một cách tiếp cận thuận tiện và tăng tính rõ ràng trong thiết kế của hệ thống.

**Proxy:**

Hệ thống tải ảnh từ internet.

Giả sử bạn đang xây dựng một ứng dụng hiển thị ảnh từ internet. Trong quá trình hiển thị, ứng dụng phải tải ảnh từ các nguồn khác nhau như máy chủ ảnh chính và các dịch vụ lưu trữ ảnh. Tuy nhiên, việc tải ảnh từ các nguồn này có thể mất thời gian và tài nguyên.

Trong trường hợp này, Proxy design pattern được sử dụng để tạo ra một lớp Proxy (đại diện) để quản lý việc tải ảnh. Lớp Proxy sẽ kiểm tra xem ảnh đã được tải về chưa. Nếu ảnh đã tồn tại, lớp Proxy trả về ảnh từ bộ nhớ cache mà không cần thực hiện tải lại từ nguồn gốc. Ngược lại, nếu ảnh chưa tồn tại, lớp Proxy sẽ thực hiện việc tải ảnh từ nguồn gốc và lưu vào bộ nhớ cache để sử dụng lần sau.

Lớp Proxy đóng vai trò là một giao diện giống như lớp thực tế để người dùng có thể truy cập ảnh một cách thuận tiện. Proxy có thể cung cấp các chức năng bổ sung như đăng nhập, kiểm soát quyền truy cập, quản lý bộ nhớ cache và ghi log.

Proxy design pattern giúp tối ưu hóa việc tải ảnh từ internet bằng cách kiểm tra và sử dụng cache khi có thể. Nó giảm thời gian và tài nguyên cần thiết để tải ảnh và cung cấp trải nghiệm nhanh chóng cho người dùng. Ngoài ra, Proxy cũng cung cấp khả năng mở rộng để thêm các chức năng bổ sung mà không ảnh hưởng đến người dùng cuối.

**DESIGN PATTERN BEHIVIOR**

**Loại:**

[Behavioral Pattern](https://gpcoder.com/category/design-pattern/behavior-pattern/) (nhóm tương tác/ hành vi – 8 mẫu) gồm: Template Method, Chain of Responsibility, Command, Iterator, Mediator, Observer, State, Strategy.

**Định nghĩa:**

Là mẫu thiết kế thuộc nhóm Behivior (nhóm tương tác/ hành vi), nó tập trung vào cách đối tượng và lớp tương tác và hoạt động với nhau để thực hiện các hành vi và tương tác trong ứng dụng.

(Tên pattern) + Giúp giải quyết vấn đề về + (chỗ này ghép từng định nghĩa pattern)

+ Quản lý trạng thái và luồng điều khiển (State, Strategy)

+ Xử lý sự kiện và thông báo (Observer, Mediator)

+ Phân tách trách nhiệm và chức năng giữa các đối tượng (Command, Chain of Responsibility)

+ Quản lý tương tác giữa các đối tượng (Iterator)

+ Cung cấp một khung mẫu để xác định cấu trúc và quy trình chung cho một thuật toán (Template Method)

**Lợi ích**

Tách biệt trách nhiệm: Các design pattern behavior giúp tách biệt trách nhiệm và chức năng giữa các đối tượng trong hệ thống. Điều này giúp giảm sự phụ thuộc và làm cho mã nguồn trở nên dễ bảo trì, mở rộng và thay đổi.

Tăng tính linh hoạt: Các design pattern behavior tăng tính linh hoạt của mã nguồn. Chúng cho phép thay đổi và thay thế các hành vi một cách linh hoạt, mà không ảnh hưởng đến các đối tượng sử dụng hành vi đó.

Tái sử dụng mã nguồn: Các design pattern behavior khuyến khích tái sử dụng mã nguồn. Thay vì lặp lại cùng một logic xử lý trong nhiều đối tượng, chúng cho phép bạn tách rời logic xử lý đó thành một đối tượng riêng biệt và sử dụng lại nó trong các đối tượng khác.

Giảm sự phụ thuộc: Các design pattern behavior giảm sự phụ thuộc giữa các thành phần trong hệ thống. Thay vì giao tiếp trực tiếp, các thành phần tương tác thông qua giao diện chung hoặc trung gian. Điều này giúp mã nguồn trở nên độc lập và dễ dàng bảo trì.

Tăng khả năng mở rộng: Các design pattern behavior cung cấp cơ chế để mở rộng chức năng của hệ thống. Bạn có thể thêm mới các hành vi, phương thức hoặc cách thức xử lý mà không ảnh hưởng đến các thành phần hiện có. Điều này giúp mở rộng hệ thống một cách linh hoạt và dễ dàng.

Cải thiện khả năng kiểm thử: Các design pattern behavior tạo điều kiện thuận lợi cho việc kiểm thử và gỡ lỗi. Chúng cho phép bạn kiểm tra và xử lý các hành vi một cách độc lập và tạo ra các bộ kiểm thử dễ dàng.

**Ví dụ:**

**Template Method:**

Quy trình xây dựng nhà.

Khi xây dựng một ngôi nhà, có một quy trình cố định mà các công việc phải tuân theo. Tuy nhiên, mỗi công việc trong quy trình có thể có các bước khác nhau, tùy thuộc vào loại công việc và vật liệu được sử dụng.

Trong trường hợp này, Template Method design pattern được sử dụng để tạo ra một lớp cơ sở (base class) chứa quy trình xây dựng nhà, và các lớp con (subclasses) sẽ triển khai các bước cụ thể cho từng công việc.

Ví dụ, lớp cơ sở có thể chứa các phương thức như "xây dựng móng", "xây dựng khung cột", "lắp đặt hệ thống điện" và "hoàn thiện nội thất". Các lớp con sẽ triển khai các bước cụ thể cho từng công việc. Ví dụ, lớp con A có thể triển khai bước xây dựng móng bằng bê tông, trong khi lớp con B có thể triển khai bước xây dựng móng bằng thép.

Khi xây dựng một ngôi nhà, các lớp con sẽ được sử dụng để thực hiện các bước cụ thể cho từng công việc, nhưng quy trình tổng thể vẫn được điều khiển bởi lớp cơ sở. Lớp cơ sở xác định quy trình chung và sử dụng các phương thức mẫu (template methods) để gọi các bước cụ thể trong lớp con.

Template Method design pattern giúp tách rời quy trình chung từ các bước cụ thể trong một quy trình và cho phép tái sử dụng quy trình chung trong các loại công việc khác nhau. Nó cũng tạo ra một sự cấu trúc linh hoạt và dễ mở rộng cho quy trình xây dựng nhà và các công việc tương tự.

**Chain of Responsibility:**

Quá trình xử lý yêu cầu trong một công ty.

Giả sử bạn làm việc trong một công ty và có một quá trình xử lý yêu cầu từ khách hàng. Công ty của bạn có các bộ phận khác nhau như bộ phận bán hàng, bộ phận kỹ thuật, và bộ phận tài chính. Mỗi bộ phận có trách nhiệm xử lý một phần của yêu cầu từ khách hàng.

Trong trường hợp này, Chain of Responsibility design pattern được sử dụng để tạo ra một chuỗi liên kết các bộ phận để xử lý yêu cầu. Mỗi bộ phận được liên kết với bộ phận tiếp theo trong chuỗi và xử lý yêu cầu theo trình tự.

Ví dụ, khi một yêu cầu từ khách hàng được nhận, bộ phận bán hàng sẽ kiểm tra và xử lý các yêu cầu liên quan đến đặt hàng và giá cả. Nếu bộ phận bán hàng không thể xử lý yêu cầu, nó sẽ chuyển yêu cầu cho bộ phận kỹ thuật để xem xét và giải quyết các vấn đề kỹ thuật. Nếu bộ phận kỹ thuật cũng không thể xử lý yêu cầu, nó sẽ chuyển yêu cầu cho bộ phận tài chính để xử lý các vấn đề liên quan đến thanh toán và tài chính.

Quá trình chuyển giao yêu cầu giữa các bộ phận trong chuỗi liên kết được thực hiện dựa trên một tiêu chí xác định xem bộ phận nào có thể xử lý yêu cầu. Mỗi bộ phận trong chuỗi chỉ đảm nhận trách nhiệm xử lý yêu cầu của mình và chuyển tiếp yêu cầu cho bộ phận tiếp theo nếu cần.

Chain of Responsibility design pattern giúp tách rời quá trình xử lý yêu cầu từ khách hàng và tạo ra một cấu trúc linh hoạt và dễ mở rộng. Nó cho phép thêm hoặc thay đổi các bộ phận trong chuỗi mà không ảnh hưởng đến các bộ phận khác và quá trình xử lý. Đồng thời, nó cũng giúp giảm sự phụ thuộc giữa các bộ phận và tăng tính linh hoạt trong thiết kế.

**Command:**

Hệ thống điều khiển từ xa trong một căn hộ thông minh.

Giả sử bạn có một căn hộ thông minh với nhiều thiết bị điện tử như đèn, máy điều hòa, TV, và cửa điện tử. Bạn muốn tạo ra một hệ thống điều khiển từ xa để điều khiển các thiết bị này một cách dễ dàng.

Trong trường hợp này, Command design pattern được sử dụng để tạo ra các đối tượng Command đại diện cho các hành động cụ thể mà người dùng muốn thực hiện trên các thiết bị. Mỗi đối tượng Command chứa các phương thức như "bật đèn", "tắt đèn", "điều chỉnh nhiệt độ", "thay đổi kênh TV", và "mở cửa".

Hệ thống điều khiển từ xa có một bộ điều khiển (Remote Control) chứa các nút bấm tương ứng với các đối tượng Command. Khi người dùng nhấn một nút bấm, bộ điều khiển sẽ tạo ra một đối tượng Command tương ứng và gọi phương thức thực hiện (execute) trên đối tượng đó.

Mỗi đối tượng Command biết cách thực hiện hành động cụ thể trên thiết bị tương ứng. Ví dụ, đối tượng Command cho hành động "bật đèn" sẽ gửi tín hiệu bật đèn cho hệ thống đèn trong căn hộ thông minh.

Command design pattern cho phép người dùng tương tác với hệ thống điều khiển từ xa một cách dễ dàng và trừu tượng. Nó tách rời người dùng và hành động cụ thể được thực hiện trên các thiết bị, cho phép thêm hoặc thay đổi các hành động mà không ảnh hưởng đến người dùng và các thiết bị khác. Ngoài ra, nó cung cấp tính linh hoạt và dễ mở rộng trong việc xây dựng hệ thống điều khiển từ xa.

**Iterator:**

Quá trình duyệt danh sách khách hàng trong một hệ thống quản lý khách hàng.

Giả sử bạn đang làm việc trong một hệ thống quản lý khách hàng cho một công ty. Công ty của bạn lưu trữ thông tin khách hàng trong một danh sách và bạn cần duyệt qua từng khách hàng để thực hiện các hoạt động như hiển thị thông tin khách hàng, tìm kiếm khách hàng, hoặc thống kê dữ liệu.

Trong trường hợp này, Iterator design pattern được sử dụng để tạo ra một đối tượng Iterator để duyệt qua danh sách khách hàng. Đối tượng Iterator cung cấp các phương thức như "tiếp theo" để di chuyển đến khách hàng tiếp theo trong danh sách, "có tiếp theo" để kiểm tra xem còn khách hàng tiếp theo hay không, và "lấy hiện tại" để trả về khách hàng hiện tại.

Khi cần duyệt qua danh sách khách hàng, bạn sẽ sử dụng đối tượng Iterator để lặp qua từng khách hàng một cách tuần tự. Bạn có thể truy cập thông tin của từng khách hàng và thực hiện các hoạt động tương ứng.

Iterator design pattern giúp tách rời quá trình duyệt qua danh sách khách hàng và tăng tính linh hoạt trong việc thực hiện các hoạt động trên danh sách. Nó cho phép thay đổi cách duyệt qua danh sách mà không ảnh hưởng đến các hoạt động khác và ngược lại. Ngoài ra, nó cũng cung cấp khả năng mở rộng để thực hiện các loại duyệt danh sách khác nhau trong tương lai.

**Mediator:**

hệ thống chat nhóm trong một ứng dụng di động.

Giả sử bạn đang làm việc trên một ứng dụng di động có chức năng chat nhóm, trong đó người dùng có thể tham gia vào nhiều nhóm chat khác nhau. Mỗi nhóm chat có nhiều thành viên và khi một thành viên gửi tin nhắn trong nhóm, các thành viên khác cũng nhận được tin nhắn đó.

Trong trường hợp này, Mediator design pattern được sử dụng để tạo ra một đối tượng Mediator đại diện cho kênh giao tiếp giữa các thành viên trong nhóm chat. Đối tượng Mediator sẽ giữ trạng thái của từng thành viên và chịu trách nhiệm phân phối tin nhắn từ một thành viên đến các thành viên khác.

Khi một thành viên gửi tin nhắn, nó sẽ gửi tin nhắn đến đối tượng Mediator. Đối tượng Mediator sẽ nhận tin nhắn và chịu trách nhiệm phân phối tin nhắn đến tất cả các thành viên khác trong nhóm chat. Mỗi thành viên nhận tin nhắn từ Mediator và hiển thị nội dung tin nhắn trên giao diện người dùng của họ.

Mediator design pattern giúp giảm sự phụ thuộc giữa các thành viên trong nhóm chat và tạo ra một kênh giao tiếp trung gian. Nó cho phép các thành viên gửi và nhận tin nhắn mà không cần phải trực tiếp liên lạc với nhau. Đồng thời, nó cung cấp tính linh hoạt và mở rộng trong việc thêm các tính năng khác như xử lý thông báo, quản lý thành viên, hoặc thiết lập quyền truy cập trong nhóm chat.

**Observer:**

Hệ thống thông báo tin tức trực tuyến.

Giả sử bạn đang làm việc trên một hệ thống thông báo tin tức trực tuyến, trong đó người dùng có thể đăng ký và theo dõi các danh mục tin tức khác nhau. Khi có tin tức mới được đăng tải trong một danh mục, người dùng đã đăng ký sẽ nhận được thông báo về tin tức đó.

Trong trường hợp này, Observer design pattern được sử dụng để tạo ra một đối tượng Observer đại diện cho người dùng đăng ký thông báo. Mỗi người dùng sẽ là một Observer và đăng ký để nhận thông báo từ các danh mục tin tức mà họ quan tâm.

Khi một tin tức mới được đăng tải trong một danh mục, đối tượng danh mục (Subject) sẽ thông báo cho tất cả các Observer đã đăng ký với nó. Mỗi Observer nhận thông báo và cập nhật giao diện người dùng của mình để hiển thị tin tức mới.

Observer design pattern cho phép tách rời quá trình thông báo và xử lý thông báo từ hệ thống tin tức. Nó cho phép người dùng đăng ký và nhận thông báo mà không phụ thuộc vào quá trình đăng tải tin tức. Đồng thời, nó cũng cung cấp tính linh hoạt và mở rộng trong việc thêm các danh mục tin tức mới hoặc quản lý người dùng và đăng ký thông báo.

**State:**

Trạng thái đăng nhập của một ứng dụng.

Giả sử bạn đang phát triển một ứng dụng di động và có tính năng đăng nhập. Ứng dụng của bạn có ba trạng thái chính: chưa đăng nhập, đã đăng nhập và đăng xuất. Mỗi trạng thái sẽ có những hành vi và giao diện người dùng khác nhau.

Trong trường hợp này, State design pattern được sử dụng để tạo ra các đối tượng State đại diện cho từng trạng thái của ứng dụng. Mỗi đối tượng State sẽ chứa các phương thức như hiển thị giao diện đăng nhập, xử lý đăng nhập, hiển thị giao diện đã đăng nhập và xử lý đăng xuất.

Khi người dùng mở ứng dụng, ứng dụng sẽ ở trạng thái chưa đăng nhập. Người dùng sẽ thấy giao diện đăng nhập và có thể nhập thông tin đăng nhập để đăng nhập vào ứng dụng. Khi đăng nhập thành công, ứng dụng chuyển sang trạng thái đã đăng nhập và hiển thị giao diện đã đăng nhập cho người dùng. Người dùng có thể thực hiện các hoạt động trong trạng thái này. Khi người dùng chọn đăng xuất, ứng dụng chuyển sang trạng thái đăng xuất và hiển thị lại giao diện đăng nhập.

State design pattern cho phép ứng dụng chuyển đổi linh hoạt giữa các trạng thái và xử lý hành vi tương ứng với từng trạng thái. Nó giúp tách rời logic xử lý trạng thái khỏi các hành động cụ thể và giao diện người dùng. Đồng thời, nó cung cấp tính mở rộng và dễ bảo trì trong việc thêm các trạng thái mới hoặc thay đổi hành vi của các trạng thái hiện có.

**Strategy:**

Một ứng dụng chương trình đặt hàng trực tuyến với các phương thức thanh toán khác nhau.

Giả sử bạn đang phát triển một ứng dụng cho phép người dùng đặt hàng trực tuyến và có các phương thức thanh toán khác nhau như thanh toán bằng thẻ tín dụng, chuyển khoản ngân hàng hoặc thanh toán khi nhận hàng. Mỗi phương thức thanh toán sẽ có quy trình xử lý và luồng làm việc khác nhau.

Trong trường hợp này, Strategy design pattern được sử dụng để tạo ra các đối tượng Strategy đại diện cho từng phương thức thanh toán. Mỗi đối tượng Strategy sẽ chứa các phương thức như thanh toán, kiểm tra tính hợp lệ của thanh toán và xử lý các yêu cầu liên quan đến thanh toán.

Khi người dùng đặt hàng và chọn phương thức thanh toán, ứng dụng sẽ sử dụng đối tượng Strategy tương ứng với phương thức thanh toán được chọn để thực hiện quy trình thanh toán. Đối tượng Strategy sẽ xử lý các bước cần thiết để hoàn thành thanh toán theo quy trình riêng của từng phương thức thanh toán.

Strategy design pattern cho phép ứng dụng linh hoạt trong việc chọn và thay đổi phương thức thanh toán mà không ảnh hưởng đến quy trình chung của ứng dụng. Nó tách rời logic xử lý thanh toán và quy trình thanh toán chung, đồng thời cung cấp khả năng mở rộng để thêm các phương thức thanh toán mới hoặc thay đổi hành vi của các phương thức thanh toán hiện có.