1. Proxy

* Vấn đề: Quá trình xử lý của cơ sở dữ liệu có thể diễn ra rất chậm khi mà có rất nhiều client cùng truy cập đến và đòi hỏi xử lý một điều gì đó
* Hướng giải quyết:

+ Implement lazy initialization, tức là chỉ tạo khi cần. Khi đó client muốn truy cập đều phải chạy qua đoạn code này, tuy nhiên vấn đề phát sinh là sẽ khiến code duplicate

+ Điều hay nhất là có thể là đưa dòng code này vào chính đối tượng đó. Nhưng nếu lớp này là 3rd party thì không thể.

* Proxy đại diện cho đối tượng cơ sở dữ liệu giúp xử lý lazy initialization (quá trình khởi tạo chậm) và lưu vào bộ nhớ đệm kết quả
* Khái niệm:

+ Hay còn gọi là Surrogate là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern).

+ Điều khiển gián tiếp việc truy xuất đối tượng thông qua một đối tượng được ủy nhiệm

+ Cung cấp 1 class đại diện để quản lí sự truy xuất đến thành phần của 1 class khác

+ Giải quyết vấn đề security, perfomance, validation,…

* Sử dụng khi:

+ Cần loại bỏ một đối tượng nặng khi không có máy khách nào sử dụng nó.

+ Cần lưu trữ kết quả của các yêu cầu máy khách và quản lý vòng đời của bộ nhớ cache này, đặc biệt nếu kết quả khá lớn.

+ Muốn giữ lịch sử của các yêu cầu đối với đối tượng service.

+ Có một đối tượng dịch vụ nặng gây lãng phí tài nguyên hệ thống do luôn hoạt động, mặc dù thỉnh thoảng bạn chỉ cần nó.

+ Muốn chỉ những khách hàng cụ thể mới có thể sử dụng đối tượng dịch vụ.

+ Đối tượng service được đặt trên một máy chủ từ xa.

* Các loại Proxy:

+ Virtual Proxy: chịu trách nhiệm mã hóa một yêu cầu và các đối số của nó và gửi yêu cầu được mã hóa đến đối tượng trong một không gian địa chỉ khác.

+ Remote Proxy: Có thể cache thông tin bổ sung về đối tượng để trì hoãn việc truy cập nó.

+ Protection Proxy: Kiểm tra xem có quyền truy cập cần thiết để thực hiện yêu cầu hay không.

* Ví dụ thực tế: Thẻ tín dụng có thể được sử dụng để thanh toán giống như tiền mặt.
* Ưu/ nhược điểm:

+ Ưu điểm:

Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.

Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.

Open/Closed Principle: Bạn có thể thêm proxy mới mà không cần thay đổi service hoặc clients.

Cải thiện Performance thông qua lazy loading

+ Nhược điểm:

Mã có thể trở nên phức tạp hơn vì bạn cần phải thêm lớp mới.

Phản hồi từ service có thể bị trì hoãn.

1. FAÇADE

* Vấn đề: Khi làm việc với một số lượng lớn các đối tượng trong hệ thống hay một thư viện phức tạp, chúng ta cần phải tự khởi tạo tất cả đối tượng này, thường xuyên theo dõi các thay đổi của nó, thứ tự logic cũng phải xử lý chính xác với bên thứ 3
* Hướng giải quyết: Facade, chúng ta sẽ chỉ cần gọi Facade để thực thi các hành động dựa trên các parameters được cung cấp.
* Khái niệm:

+ Là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern).

+ Cung cấp cho chúng ta một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này

+ Cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn

* Cấu trúc:

+ Client: Đối tượng sử dụng Facade để tương tác với các subsystem thay vì gọi subsystem trực tiếp

+ Facade: Facade nắm rõ được hệ thống con nào đảm nhiệm việc đáp ứng yêu cầu của client, nó sẽ chuyển yêu cầu của client đến các đối tượng hệ thống con tương ứng.

+ Addition Facade: có thể được tạo ra để tránh việc làm phức tạp một facade. Có thể được sử dụng bởi cả client và facade.

+ Complex Subsystems: Bao gồm nhiều object khác nhau, được cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.

* Sử dụng khi:

+ Người sử dụng phụ thuộc nhiều vào các lớp cài đặt.

+ Đóng gói nhiều chức năng, che giấu thuật toán phức tạp

+ Cần một interface không rắc rối mà dễ sử dụng

+ Muốn gom nhóm chức năng lại để Client dễ sử dụng

+ Giảm sự phụ thuộc

+ Tăng khả năng độc lập và khả chuyển

* Ví dụ thực tế: Khi bạn gọi điện đến shop để đặt hàng. Khi đó tổng đài sẽ là Facade của tất cả dịch vụ và phòng ban của shop. Hệ thống sẽ cung cấp cho bạn một giao diện đơn giản qua điện thoại để đặt hàng, thanh toán, giao hàng hay nhiều dịch vụ khác nhau.
* Ưu/ nhược điểm:

+ Ưu điểm:

Có thể tách mã nguồn của mình ra khỏi sự phức tạp của hệ thống con

Hệ thống tích hợp thông qua Facade sẽ đơn giản hơn vì chỉ cần tương tác với Facade thay vì hàng loạt đối tượng khác.

Tăng khả năng độc lập và khả chuyển, giảm sự phụ thuộc

Có thể đóng gói nhiều hàm được thiết kế không tốt bằng 1 hàm có thiết kế tốt hơn.

+ Nhược điểm:

Class Facade của bạn có thể trở lên quá lớn, làm quá nhiều nhiệm vụ với nhiều hàm chức năng trong nó

Việc sử dụng Facade cho các hệ thống đơn giản, ko quá phức tạp trở nên dư thừa.

* Mối liên hệ giữa Proxy và Façade:
* Facade tương tự như Proxy ở chỗ cả hai đều đệm một thực thể phức tạp và tự khởi tạo nó.
* Không giống như Facade, Proxy có giao diện giống với đối tượng dịch vụ của nó, điều này làm cho chúng có thể hoán đổi cho nhau

1. **Factory Method**

* Khái niệm:

+ Định nghĩa **một interface** để **tạo đối tượng**, nhưng để các **đối tượng con** của nó **quyết định** class nào sẽ được tạo.

+ Hay là một design pattern cung cấp một giao diện (interface) để tạo đối tượng trong **class cha** nhưng cho phép **class con** của nó **ghi đè để tạo đối tượng** theo **những kiểu khác nhau** của bài toán , nhằm giải quyết vấn đề **tạo một đối tượng** mà **không cần thiết** **chỉ ra class nào** sẽ được tạo

* Sơ đồ:

+ Product: Là interface mà có đặc tính phổ biến nhất.

+ Creator (*abstract*) triển khai cho tất cả các hàm liên quan tới products, *ngoại trừ hàm FactoryMethod()*.

+ ConcreteCreator là class duy nhất chịu trách nhiệm tạo ra sản phẩm cụ thể ConcreteProduct. *ConcreteCreator* triển khai hàm *FactoryMethod()* là hàm thật sự tạo ra Product.

* Sử dụng khi:

+ Một class **không biết tạo** ra kiểu **class con** nào **phù hợp** với yêu cầu

+ Bạn muốn cung cấp cho người dùng thư viện hoặc framework của bạn một cách **dễ mở rộng** các **thành phần sẵn có bên trong** nó.

+ Muốn **tiết kiệm tài nguyên** hệ thống bằng việc **tái sử dụng các object đã có** thay vì xây dựng lại mỗi lần có thêm sản phẩm mới.

* Ưu/ nhược điểm:

+ Ưu điểm:

Tránh việc gắn chặt việc tạo sản phẩm với bất kỳ một loại sản phẩm nào cụ thể

Single Responsibility Principle: Dễ dàng bảo trì code với các lớp tách biệt

Open/Closed Principle: Như những mô tả ở trên, chúng ta có thể dễ dàng thêm một kiểu sản phẩm mới mà không làm ảnh hưởng đến code hiện tại.

+ Nhược điểm:

Số lớp yêu cầu cao

Yêu cầu độ phức cao trong quá trình triển khai

1. **Singleton Pattern**

* Khái niệm:

+ Là một mẫu thiết kế đảm bảo rằng **mỗi lớp** sẽ chỉ có **duy nhất một instance,** đồng thời cung cấp **một điểm truy cập toàn cục** cho instance này

+ Hay nói cách khác: lớp sẽ khởi tạo duy nhất một lần đầu tiên, những lần sau sẽ gọi lại instance đã tạo,đảm bảo việc chỉ có một instance được tạo cho toàn ứng dụng

* Sử dụng khi:

+ Muốn tại một thời điểm lớp chỉ có duy nhất một instance thực hiện công việc đó và nó có hiệu lực ở mọi nơi.

+ Singleton được ứng dụng trong nhiều trường hợp:

* + Kết nối tới database
  + Drivers
  + Ghi log vào duy nhất 1 file
  + Settings cho ứng dụng

……..

1. **ADAPTER**

* Vấn đề: Trong cuộc sống, adapter thường xuất hiện dưới dạng một thiết bị kết nối giữa các phích cắm và ổ điện khác loại. Adapter trong lập trình hướng đối tượng cũng có chức năng tương tự như thế giới thực: Nó nhận 1 interface và thích ứng để trở thành 1 interface client cần sử dụng.
* Hướng giải quyết: tạo ra một class đặc biệt, gọi là adapter, là thằng trung gian giúp thư viện trên mạng hiểu được yêu cầu của client. Tạm gọi phương thức vẽ điểm là Point(), lớp của thư viện là PolarGraph, interface của client là Graph, và PolarGraphAdapter là lớp adapter đảm nhận việc chuyển đổi.
* Khái niệm:

+ Là một structural design pattern (Không thay đổi thành phần cấu trúc)

+ Cho phép các Objects có Interface không tương thích có thể liên kết với nhau

+ Là Wrapper pattern (Giữ vai trò trung gian giữa 2 lớp)

* Ưu/nhược điểm:

+ Ưu điểm:

* Cho phép nhiều đối tượng interface khác nhau giao tiếp với nhau.
* Phân tách việc chuyển đổi interface với business logic chính của chương trình.
* Cách tiếp cận này có thêm một ưu điểm là ta có thể sử dụng adapter với các class con của adaptee. (Liskov substitution principle).
* Làm việc với adapter class thay vì sửa đổi bên trong adaptee class đã có sẵn, thuận tiện cho việc mở rộng (Open/closed principle).
* Client tiếp cận thông qua interface, thay vì implementation (Software design principle).

+ Nhược điểm:

* Tất cả yêu đầu phải được chuyển tiếp thông qua adapter, làm tăng thêm một ít chi phí
* Độ phức tạp của code nhìn chung tăng lên vì phải thêm interface và lớp.
* Vì không phải lúc nào ta cũng có thể thích nghi các method của các interface khác nhau với nhau, nên exception có thể xảy ra. Vấn đề này có thể tránh được nếu client cẩn thận hơn và adapter có tài liệu hướng dẫn rõ ràng.

* Ứng dụng:

+ Khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A.

+ Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn

+ Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau

+ Khi cần đảm bảo nguyên tắc Open/ Close trong một ứng dụng

1. **Bridge Design Pattern**

* Vấn đề: Việc thêm các loại hình dạng và màu sắc mới vào hệ thống thì sẽ phải tạo thêm nhiều lớp kế thừa.
* **Giải pháp:** Trong lớp Shape có một thuộc tính là Color, Color thì có thể thêm các màu kế thừa như Xanh Đỏ Tím Vàng tùy ý. Khi đó muốn Hình Chữ Nhật Đỏ ta chỉ cần Hình Chữ Nhật có thuộc tính Màu là đỏ thôi, tương tự với các hình khác mà không cần phải kế thừa nhiều.
* Khái niệm:

+ Là một trong những Pattern thuộc nhóm Structural Pattern.

+ Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu.

* Sử dụng khi:

+ Tách ràng buộc giữa Abstraction và Implementation.

+ Cả Abstraction và Implementation của chúng nên được mở rộng bằng subsclass.

+ Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

* Ưu/nhược điểm:

+ Ưu điểm:

* *Giảm sự phục thuộc giữa abstraction và implementation (loose coupling):* tính kế thừa trong OOP thường gắn chặt abstraction và implementation lúc build chương trình. Bridge Pattern có thể được dùng để cắt đứt sự phụ thuộc này và cho phép chúng ta chọn implementation phù hợp lúc runtime.

Giảm số lượng những lớp con không cần thiết: một số trường hợp sử dụng tính inheritance sẽ tăng số lượng subclass rất nhiều.

* *Code sẽ gọn gàn hơn và kích thước ứng dụng sẽ nhỏ hơn:* do giảm được số class không cần thiết.
* *Dễ bảo trì hơn:* các Abstraction và Implementation của nó sẽ dễ dàng thay đổi lúc runtime cũng như khi cần thay đổi thêm bớt trong tương lai.

Dễ dàng mở rộng về sau: thông thường các ứng dụng lớn thường yêu cầu chúng ta thêm module cho ứng dụng có sẵn nhưng không được sửa đổi framework/ứng dụng có sẵn vì các framework/ứng dụng đó có thể được công ty nâng cấp lên version mới.

Cho phép ẩn các chi tiết implement từ client: do abstraction và implementation hoàn toàn độc lập nên chúng ta có thể thay đổi một thành phần mà không ảnh hưởng đến phía Client.

+ Nhược điểm:

* Có thể làm tăng độ phức tạp khi áp dụng cho một lớp có tính gắn kết cao

Số lượng gọi gián tiếp tăng gấp đôi

Ảnh hưởng đến hiệu suất của chương trình

* Design Pattern liên quan:
* *Adapter* Adapter và Bridge giống nhau là đều sẽ nhờ vào một lớp khác để thực hiện một số xử lý nào đó Khác nhau: Khác về mục đích sử dụng
* Adapter được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được, giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.
* Bridge được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt. Khác nhau về thời điểm ứng dụng
* Adapter làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại)
* Bridge nên được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.
* *Abstract Factory:* có thể sử dụng cùng với Bridge. Việc ghép nối này rất hữu ích khi một số trừu tượng được xác định bởi Bridge chỉ có thể hoạt động với các triển khai cụ thể. Trong trường hợp này, Abstract Factory có thể đóng gói các quan hệ này và ẩn sự phức tạp khỏi Client.
* *Builder:* có thể kết hợp với Bridge. Director class có thể giữ vai trò là Abstraction, trong khi các Builder class khác giữ vai trò Implementation

1. **Composite Design pattern**

* **Khái niệm:**

+ Là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern).

+ Là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn.

+ Cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.

+ Được sử dụng khi chúng ta cần **xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object**.

+ Tần suất sử dụng: khá cao

* **Thành phần gồm 4 thành phần:**

**+ Component**: là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này

**+ Leaf**: là lớp hiện thực (implements) các phương thức của Component - các object không có con

**+ Composite**: lưu trữ tập hợp các Leaf và cài đặt các phương thức của Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý

**+ Client**: sử dụng Component để làm việc với các đối tượng trong Composite

* **Ưu/ nhược điểm:**

+ Ưu điểm:

* Có thể làm việc với các cấu trúc cây phức tạp thuận tiện hơn
* Nguyên tắc mở/ đóng: có thể khởi tạo các loại *phần tử mới* vào ứng dụng mà không phá vỡ code hiện có đang hoạt động với đối tượng cây

+ Nhược điểm:

* Code có thể trở nên phức tạp hơn mức bình thường, vì có rất nhiều interfaces và classes được khởi tạo cùng với mẫu
* Sử dụng khi:

+ Số lượng đối tượng ít làm giảm mức sử dụng bộ nhớ và giúp tránh các lỗi liên quan đến bộ nhớ

+ Giảm thời gian thực thi chương trình của mình bằng cách chia sẻ các đối tượng

* Không nên sử dụng:

+ Có thể khó cung cấp một interface chung cho các lớp có chức năng khác nhau quá nhiều. Trong một số trường hợp nhất định, cần tổng quát hóa quá mức interface thành phần khiến nó khó hiểu hơn

+ Lạm dụng Composite Design pattern có thể làm cho thiết kế trở nên quá *chung chung*

* **Áp dụng thực tế:**

+ Một chương trình quản lý một hệ thống tập tin

* Là một cấu trúc cây có chứa các nhánh là các thư mục (folder – composite), cũng như các nút lá là các tệp (file – leaf).

+ Gia phả trong gia đình

1. **Decorator Design pattern:** thêm chức năng mà ko cần code thêm vào class đó.

Giúp bổ sung thêm các trạng thái (nghĩa là các biến thành viên) và hoạt động (các phương thức) mới vào một object sẵn có.

* **Khái niệm:**

+ Decorator là một mẫu thiết kế thuộc nhóm **Structural Pattern**

+ Linh động thay đổi tính chất (functionality) đã có trong một đối tượng khi chương trình đang chạy (runtime) mà không ảnh hưởng đến các tình chất đã tồn tại của các đối tượng khác

+ Tần suất sử dụng: **Trung bình**

Mục đích: Mở rộng tính năng của một object tại run-time, decorator cung cấp một sự thay thế linh hoạt cho subclass để mở rộng tính năng của object

* **Thành phần:**

**+ Component**: giao diện (interface) chung để các đối tượng cần thêm chức năng trong quá trình chạy thì triển khai giao diện này

**+ ConcreteComponent** : Một cài đặt cho giao diện Component mà nó định nghĩa một đối tượng cần thêm các chức năng trong quá trình chạy

**+ Decorator** : một lớp trừu tượng dùng để duy trì một tham chiếu của đối tượng thành phần và đồng thời cài đặt các thành phần của giao diện

**+ ConcreteDecorator** : Một cài đặt của Decorator, nó cài đặt thêm các thành phần vào đầu của các đối tượng thành phần

+ *Client:* đối tượng sử dụng Component với những yêu cầu mở rộng đính kèm.

* **Ưu/ nhược điểm:**

+ Ưu điểm:

* Mở rộng hành vi của đối tượng mà không cần tạo lớp con mới
* Thêm hoặc xoá tính năng của một đối tượng trong lúc **thực thi**
* Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc

+ Nhược điểm:

* Không thể kế thừa
* Khó để xóa một wrapper cụ thể khỏi stack
* Có thể làm phức tạp quá trình khởi tạo thành phần bởi vì không chỉ phải khởi tạo thành phần mà còn phải bọc nó trong một số đối tượng khác
* **Sử dụng khi:**

+ Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này

+ Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator

+ Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức viết code

* **Áp dụng thực tế:**

+ Trong đồ họa, video, âm thanh. Ví dụ, để video streaming có thể được nén theo nhiều tỉ lệ khác nhau; Âm thanh có thể đưa ra nhiều dịch vụ chuyển đổi cùng lúc.

+ Trình duyệt và ứng dụng mobile sử dụng mẫu này để tạo ra giao diện phù hợp cho từng loại kích thước màn hình.

* Trong những tình huống sau bạn nên nghĩ tới Decorator:

+ Muốn mở rộng sealed class;

+ Mở rộng hoặc thay đổi object “động” ở runtime mà không đụng chạm đến object gốc;

+ Không muốn tạo class con (tránh sử dụng kế thừa).

* **Design Pattern liên quan:**
* *Adapter:* Decorator khác với adapter ở chỗ decorator chỉ thay đổi trách nhiệm của một đối tượng chứ không phải giao diện của nó.
* *Composite:* Decorator có thể xem là một degenerate Composite với chỉ một component. Tuy nhiên, decorator thêm các trách nhiệm bổ sung - nó không dành cho việc tập hợp object.
* *Strategy:* Decorator cho phép bạn thay đổi “da” của một đối tượng, strategy cho phép bạn thay đổi “ruột”. Đây là hai cách thay thế để thay đổi một đối tượng.

1. STRATEGY PATTERN

* Khái niệm:

+ Là mẫu thiết kế giúp bạn trừu tượng hóa những hành vi (behavior, method, function) của một đối tượng bằng cách đưa ra những cài đặt vào những lớp khác nhau.

+ Được sử dụng để tách biệt các phần của một chương trình để chúng có thể được định nghĩa và triển khai độc lập với nhau, từ đó giúp cho việc bảo trì và mở rộng mã nguồn trở nên dễ dàng hơn.

* Class Diagram:

+ **Strategy (SortStrategy): Khai báo một giao diện chung cho tất cả các thuật toán được hỗ trợ. Context sử dụng lớp này để gọi thuật toán được định nghĩa bởi một ConcreteStrategy.**

+ **ConcreteStrategy (QuickSort, ShellSort, MergeSort): Thực hiện thuật toán bằng cách sử dụng lớp Strategy**

+ **Context (SortedList): +Được cấu hình với một đối tượng ConcreteStrategy.**

**+Duy trì một tham chiếu đến một đối tượng Strategy**

**+Có thể định nghĩa một lớp cho phép Strategy truy cập vào dữ liệu của nó.**

* Sử dụng khi:

+ **Muốn sử dụng các biến thể khác nhau của một thuật toán trong một đối tượng và có thể chuyển từ một thuật toán sang thuật toán khác trong thời gian chạy.**

**+ Có nhiều lớp tương tự chỉ khác nhau trong cách thực hiện một hành vi nào đó.**

**+ Tách biệt businsess logic của một lớp (class) khỏi các chi tiết cài đặt của các thuật toán có thể không quan trọng trong ngữ cảnh của logic đó**

**+ Lớp của bạn có một câu lệnh rẽ nhánh lớn để chuyển đổi giữa các biến thể khác nhau của cùng một thuật toán.**

* **Ưu/ nhược điểm:**

**+ Ưu điểm:**

**Có thể thay thế các thuật toán linh hoạt với nhau**

**Tách biệt phần thuật toán khỏi phần sử dụng thuật toán**

**Có thể thay thế việc kế thừa bằng việc**

**thuật toán đóng gói**

**Tăng tính open-closed: Khi thay đổi thuật toán hoặc khi thêm mới thuật toán, không cần thay đổi code phần context**

+ Nhược điểm:

**Không nên áp dụng nếu chỉ có một vài xử lý và hiếm khi thay đổi.**

**Client phải nhận biết được sự khác biệt giữa các strategy.**

1. Template Method

* Khái niệm:

+ Là một pattern thuộc nhóm behavior - định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con.

+ Cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán.

**+ Xây dựng một bộ khung thuật toán trong superclass, để lại việc định nghĩa một vài bước cho các subclass mà không làm thay đổi cấu trúc chung của thuật toán**

* **Sơ đồ:**

**+ AbstractClass(Protocol)** : Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi các lớp con. Cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.

**+ ConcreteClass** : là một thuật toán cụ thể, cài đặt các phương thức của AbstractClass. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở AbstractClass (templateMethod).

* Ứng 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).

* Ưu/ nhược điểm:

+ Ưu điểm:

**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.**

+ Nhược điểm:

**Template method có càng nhiều bước để override càng khó bảo trì.**

**Bị giới hạn bởi khung thuật toán**

1. Iterator Pattern

* Khái niệm:

+ Pattern là một trong những Pattern thuộc nhóm hành vi

+ Sử dụng để “Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tập hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng của tập hợp này”

+ Nói cách khác, Iterator Pattern được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này

* Sơ đồ UML:

**+ Aggregate**: là một interface định nghĩa các phương thức để tạo đối tượng Iterator

**+ ConcreteAggregate**: cài đặt các phương thức của Aggregate

**+ Iterator**: là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử

**+ ConcreteIterator**: cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử

**+ Client** : đối tượng sử dụng Iterator Pattern

* **Lợi ích**:

**+** 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

+ Chúng ta có thể implement các loại tập hợp 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

* Sử dụng khi:

+ Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết cấu trúc bên trong của 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.

* Lưu ý khi sử dụng:

+ Có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của tập hợp một cách trực tiếp

+ Không cần thiết nếu ứng dụng chỉ hoạt động với các loại tập hợp đơn giản

1. Command Pattern
2. **Abstract Factory**
3. PROTOTYPE PATTERN