**Design Pattern là gì?**

### **Design Patterns là gì?**

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu đã được suy nghĩ, giải quyết trong tình huống cụ thể rồi.

Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.  
Nó không phải là ngôn ngữ cụ thể nào cả. Design patterns có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Ta thường gặp nó nhất trong lập trình OOP.

### **Tại sao phải sử dụng Design Patterns?**

* Design Pattern giúp bạn tái sử dụng mã lệnh và dẽ dàng mở rộng.
* Nó là tập hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Vậy khi bạn gặp bất kỳ khó khăn gì, design patterns là kim chỉ nam giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh.
* Design pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.
* Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau.
* Giúp cho các lập trình viên có thể hiểu code của người khác 1 cách nhanh chóng (có thể hiểu là tính communicate). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà k mất quá nhiều thời gian.

### **Khi nào nên sử dụng Design pattern?**

Giúp cho chương trình của mình thực sự đơn giản. Việc sử dụng các design pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các cách giải quyết cho những vấn đề đã có lời giải.

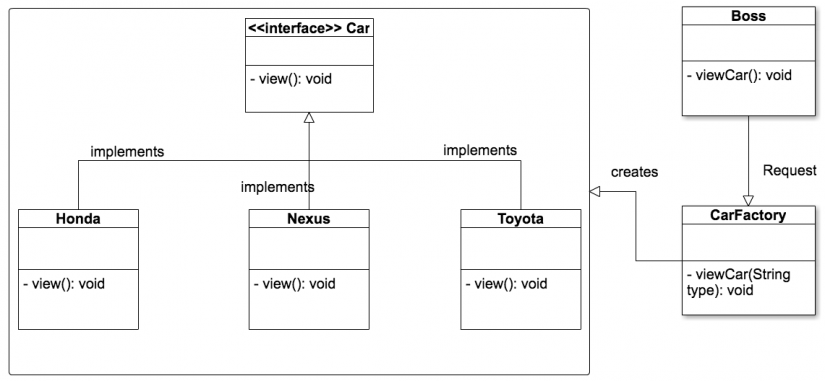
### **Phân loại Design Patterns**

Có 3 nhóm chính sau:

* **Creational Pattern** (nhóm khởi tạo) gồm: Abstract Factory, Factory Method, Singleton, Builder, Prototype. Nó sẽ giúp bạn trong việc khởi tạo đối tượng, như bạn biết để khởi tạo bạn phải sử dụng từ khóa new, nhóm **Creational Pattern** sẽ sử dụng một số thủ thuật để khởi tạo đối tượng mà bạn sẽ không nhìn thấy từ khóa này.
* **Structural Pattern** (nhóm cấu trúc) gồm: Adapter, Bridge, Composite, Decorator, Facade, Proxy và Flyweight. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.
* **Behavioral Pattern** gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng.

### **Group Creational Pattern**

1. **Factory Method**
2. **Factory method,** đầy đủ là **Factory method pattern**, là [thiết kế mẫu](https://vi.wikipedia.org/wiki/M%E1%BA%ABu_thi%E1%BA%BFt_k%E1%BA%BF_(khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh)) hướng đối tượng trong việc thiết kế [phần mềm](https://vi.wikipedia.org/wiki/Ph%E1%BA%A7n_m%E1%BB%81m) cho [máy tính](https://vi.wikipedia.org/wiki/M%C3%A1y_t%C3%ADnh), nhằm giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp nào sẽ được tạo. **Factory method** giải quyết vấn đề này bằng cách định nghĩa một phương thức cho việc tạo đối tượng, và các lớp con thừa kế có thể override để chỉ rõ đối tượng nào sẽ được tạo. Nói chung, "factory method" thường được áp dụng cho những phương thức mà nhiệm vụ chính của nó là tạo ra đối tượng.



1. **Tại sao lại phải dùng Factory pattern thay cho việc khởi tạo đối tượng thông thường? Dùng Factory pattern có lợi ích gì?**

Factory pattern đưa ra 1 ý tưởng mới cho việc khởi tạo các instance phù hợp với mỗi request từ phía Client. Sử dụng Factory pattern sẽ có những ưu điểm sau:

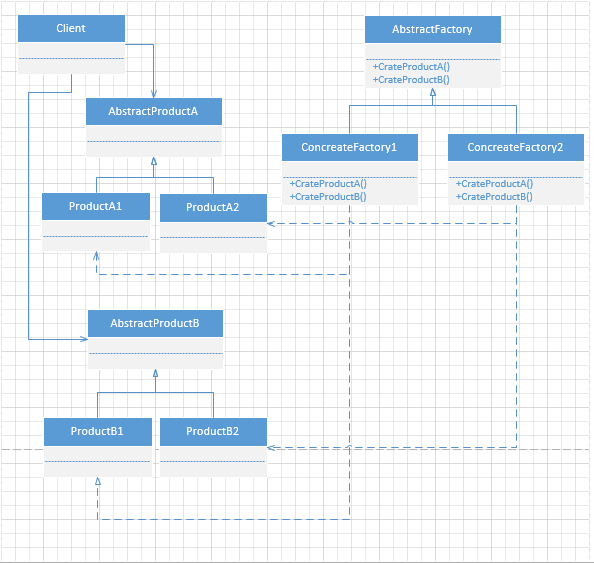
* Tạo ra 1 cách mới trong việc khởi tạo cá Object thông qua 1 interface chung.
* Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy.
* Giảm sự phụ thuộc giữa các module, các logic với các class cụ thể, mà chỉ phụ thuộc vào interface hoặc abstract class.

1. **Vậy khi nào cần sử dụng Factory pattern?**

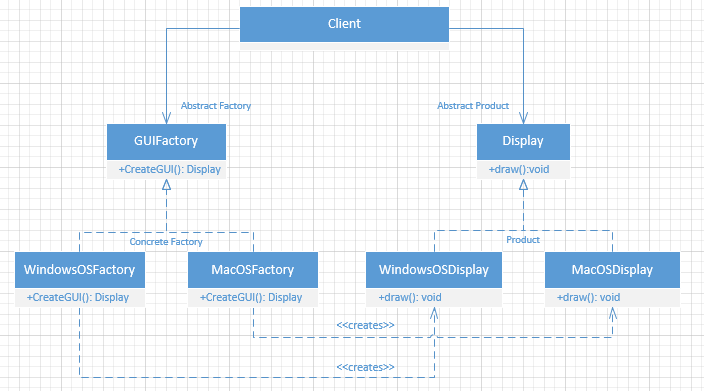
Dựa vào lợi ích của việc sử dụng Factory pattern mà ta sẽ dùng chúng với một số mục đích sau:

* Tạo ra 1 cách mới trong việc khởi tạo Object => Hiển nhiên.
* Che giấu xử lý logic của việc khởi tạo => Trong trường hợp bạn đang muốn viết 1 thư viện để người khác sử dụng.
* Giảm sự phụ thuộc => Dễ dàng cho việc mở rộng trong trường hợp bạn chưa biết chắc số lượng đối tượng (Honda, Nexus, Toyota) là đã đủ cho bài toán của mình chưa. Vì khi bạn thêm vào thì phía trên (class Boss) không cần phải sửa gì thêm, chỉ là thay đổi ở phía dưới thôi.

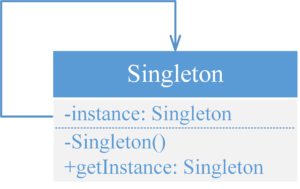
1. **Abstract Factory**
   1. **Abstract Factory Pattern**là mẫu thiết kế hướng đối tượng, nó cung cấp một lớp Interface cho tạo ra một tập hợp các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không cần chỉ ra chính xác đó là lớp cụ thể nào tại thời điểm thiết kế.



* **AbstractFactory:** định nghĩa một interface cho thao tác khởi tạo các AbstractProduct
* **ConcreteFactory**: thực thi giao tiếp AbstractFactory để tạo ra đối tượng cụ thể
* **AbstractProduct:** định nghĩa một lớp trừu tượng cho một loại đối tượng Product
* **Product:** Định nghĩa một đối tượng được tạo bởi một factory cụ thể tương ứng, Cài đặt Interface AbstractProduct
* **Client:** Sử dụng các lớp AbstractFactory và AbstractProduct trong hệ thống
  1. **Mẫu Abstract Factory được sử dụng khi nào?**
* Phía Client không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.
* Ứng dụng được cấu hình với một hoặc nhiều họ sản phẩm
* Các đối tượng cần phải được tạo ra như một tập hợp để chúng có thể tương thích với nhau
  1. **Mẫu Abstract Factory được sử dụng như thế nào?**



1. **Singleton**



* 1. **Mẫu Abstract Factory được sử dụng như thế nào?**
* **Mẫu thiết kế singleton** được thiết kế để đảm bảo chỉ có duy nhất một thực thể được tạo từ một class và nó sẽ cung cấp cho ta một phương thức để truy xuất vào thực thể đó.
* Hạn chế việc tạo ra nhiều thực thể nhưng chúng ta có thể truy xuất tới thực thể đã có sẵn.
  1. **Dùng mẫu thiết kế singleton khi nào?**

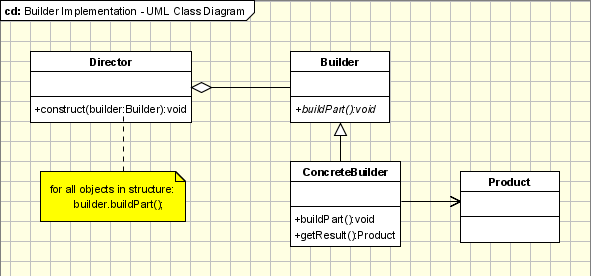
**Mẫu thiết kế singleton(singleton pattern)** là mẫu thiết kế được dùng khi chúng ta tạo ra một class mà chúng ta muốn chỉ có duy nhất một thực thể được tạo ra từ class đó và chúng ta có thể truy xuất đến thực thể đó ở bất kỳ nơi nào khi bạn muốn.

1. **Builder**
   1. **Builder pattern** là một mẫu thiết kế thuộc "Nhóm khởi tạo" (Creational Pattern). Mẫu thiết kế này cho phép lập trình viên tạo ra những đối tượng phức tạp nhưng chỉ cần thông qua các câu lệnh đơn giản để tác động nên các thuộc tính của nó.
   2. **Khi nào thì sử dụng mẫu thiết kế này?**

Một lập trình viên muốn sử dụng Builder pattern khi:

* + Anh ấy muốn thay đổi thiết kế cho việc lồng nhau của các hàm khởi tạo (Telescoping Constructor Pattern). Vấn đề này phát sinh khi lập trình viên làm việc với một lớp mà có chứa rất nhiều các thuộc tính và cần phải tạo ra nhiều hàm khởi tạo với số lượng các thuộc tính tăng dần.
  + Anh ấy cần tạo ra một đối tượng phức tạp, một đối tượng mà thuật toán để tạo tạo lập các thuộc tính là độc lập đối với các thuộc tính khác.
  1. **Cách thức triển khai thế nào?**

Mặc định, cách thức triển khai thông thường của Builder pattern gồm có 4 thành phần cơ bản:

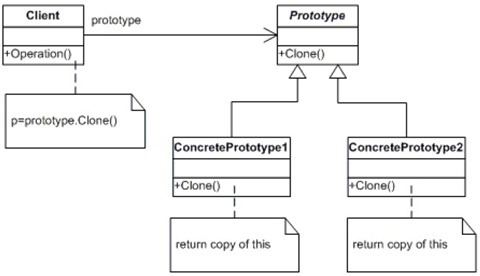


* + **Builder**: là thành phần định nghĩa một lớp trừu tượng (abstract class) để tạo ra một hoặc nhiều phần của đối tượng Product.
  + **ConcreateBuilder:** là thành phần triển khai, cụ thể hóa các lớp trừu tượng cho để tạo ra các thành phần và tập hợp các thành phần đó với nhau. thành phần này sẽ xác định và nắm giữ các thể hiện mà nó tạo ra. Đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
  + **Product:** thành phần này trong bài viết sẽ đại diện cho đối tượng phức tạp phải tạo ra.
  + **Director:** thành phần này sẽ khởi tạo đối tượng Builder.

1. **Prototype**
2. **Prototype Pattern** là một trong những pattern phổ biến trong lập trình hướng đối tượng, là một pattern thuộc nhóm Creational Patterns. Ý tưởng này là một design pattern đặc biệt có liên quan đến việc khởi tạo đối tượng (Object), thay vì tạo ra Object, Prototype pattern sử dụng việc cloning (copy nguyên mẫu của Object). Nếu việc tạo ra Object lớn và tốn nhiều tài nguyên, bạn hãy nên sử dụng prototype patern.
3. **Khi nào thì sử dụng mẫu thiết kế này?**

Giống như những mẫu thiết kế tạo lập khác (Builder, Abstract Factory và Factory Method), mẫu thiết kế Prototype ẩn việc tạo đối tượng từ client. Tuy nhiên, thay cho việc tạo ra một đối tượng không được thiết lập, nó trả về một đối tượng mới đã được thiết lập với các giá trị mà nó đã sao chép từ một đối tượng kiểu mẫu. Mẫu thiết kế Prototype không được sử dụng phổ biến trong việc xây dựng các ứng dụng nghiệp vụ (business application). Nó thường được sử dụng trong các kiểu ứng dụng xác định như đồ họa máy tính, CAD (Computer Assisted Drawing), GIS (Geographic Information Systems) và các trò chơi.

1. **Cách thức triển khai thế nào?**



Những lớp và/hoặc đối tượng trong mẫu này:

– *Prototype (ColorPrototype)*: khai báo một đối tượng cho việc sao chép của nó.

– *ConcretePrototype (Color)*: thực thi một hoạt động cho việc sao chép của nó.

– *Client (ColorManager)*: tạo ra một đối tượng mới bằng việc yêu cầu một kiểu mẫu để sao chép của nó.

Quá trình sao chép bắt đầu với việc khởi tạo class sẽ lấy làm mẫu. Client class yêu cầu một object mới của loại đó và gửi yêu cầu tới class Prototype. Một ConcretePrototype ( phụ thuộc vào loại object mà Client cần ) sẽ đảm nhận việc sao chép object client thông qua phương thức Clone(). Khi đó một object mới thuộc loại Client sẽ được tạo ra.

### **Group Structural Pattern**

* 1. **Adapter**

1. **Định nghĩa Adapter pattern:**

Adapter Pattern là pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi giao diện của một hay nhiều lớp có sẵn thành một giao diện khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các giao diện khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua giao diện trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết. Adapter Pattern còn gọi là Wrapper Pattern do cung cấp một giao diện “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có giao diện không tương thích với lớp đang viết.

1. **Các khái niệm:**

Để hiểu về sơ đồ mô tả Adapter Pattern thì trước hết bạn phải hiểu về 3 khái niệm:

* Client: Đây là lớp sẽ sử dụng đối tượng của bạn (đối tượng mà bạn muốn chuyển đổi giao diện).
* Adaptee: Đây là những lớp bạn muốn lớp Client sử dụng, nhưng hiện thời giao diện của nó không phù hợp.
* Adapter: Đây là lớp trung gian, thực hiện việc chuyển đổi giao diện cho Adaptee và kết nối Adaptee với Client.

1. **Phân loại Adapter:**

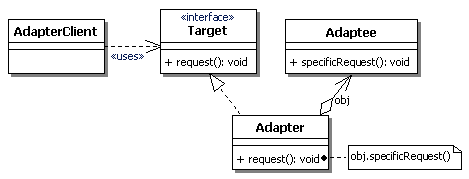
Trong hướng đối tượng có hai khái niệm quan trọng song hành cùng nhau, đó là:

* Composition: cấu thành. Nghĩa là một lớp B nào đó sẽ trở thành một thành phần của lớp A (một field trong lớp A). Tuy lớp A không kế thừa lại giao diện của lớp B nhưng nó có được mọi khả năng mà lớp B có.
* Inheritance: kế thừa. Nghĩa là một lớp Derived sẽ kế thừa từ lớp Base và thừa hưởng tất cả những gì lớp Base có. Nhờ kế thừa mà nó giúp tăng khả năng sử dụng lại code, tăng khả năng bảo trì và nâng cấp chương trình. Và do vậy kế thừa là khái niệm trọng tâm trong hướng đối tượng. Nhưng nó có một nhược điểm, đôi khi nếu chúng ta quá lạm dụng nó, nó sẽ làm cho chương trình của chúng ta phức tạp lên nhiều, điển hình là trong lập trình game. Do vậy đôi lúc trong lập trình game người ta thường có khuynh hướng thích sử dụng composition hơn.

Và ứng với hai khái niệm này sẽ cho ta hai cách để chúng ta cài đặt lớp adapter: Object Adapter và Class Adapter.

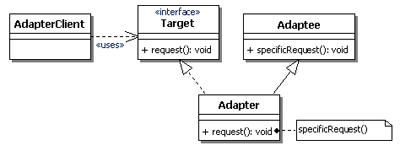
1. **Object Adapter Pattern:**

Đây là một phương pháp cài đặt Adapter Pattern dựa trên ý tưởng về composition. Một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời cài đặt giao diện mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của giao diện người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có giao diện không tương thích. Tiếp hợp đối tượng tránh được vấn đề đa thừa kế, không có trong các ngôn ngữ hiện đại (Java, C#).



1. **Class Adapter Pattern:**

Trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với giao diện không tương thích (Adaptee), đồng thời cài đặt giao diện mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của giao diện người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có giao diện không tương thích.

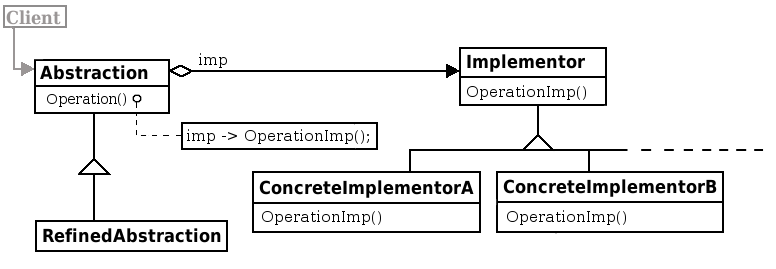


* 1. **Bridge**

1. **Định nghĩa Bridge pattern:**

**Bridge Pattern** là để tách rời phần xử lý (giải thuật…) với phần chủ thể (nơi thực thi các giải thuật đó). Phần xử lý hầu như chỉ có tác dụng bên trong phần chủ thể và không bao giờ được sử dụng ở các nơi khác. Ngoài ra các phần xử lý hầu như chỉ sử dụng cho chủ thể tương ứng (ví dụ nếu ta xác định một Bridge giữa 2 class A và A’ thì class A’ sẽ chỉ được xử dụng trong A và không sử dụng trong B (cùng abstract với A)

1. **Cấu trúc của Bridge:**



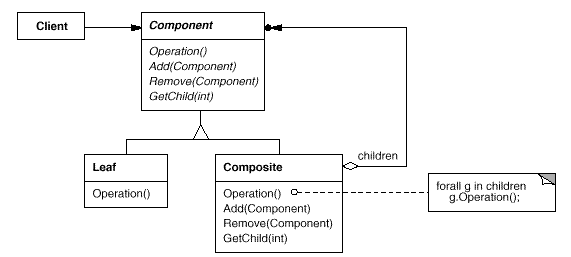
* + Client: Đối tượng cần dùng dạng thức bắc cầu
  + Abstraction: Định ra một abstract interface. Quản lí việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor)
  + Refined Abstraction: Mở rộng interface mà đã được định ra trong Abstraction
  + Implementor: Định ra các interface cho các lớp hiện thực . (Thông thường thì Abstraction interface định ra các tác vụ ở mức cao dựa trên các hiện thực của interface này.)
  + ConcreteImplementor: Hiện thực Implementor interface
  1. **Composite:**

1. **Định nghĩa Composite pattern:**

**Composite** là một mẫu thiết kế thuộc nhóm cấu trúc, cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.  
Một đối tượng Composite được tạo thành từ một hay nhiều đối tượng tương tự nhau (hoặc có một số chức năng tương tự nhau). Ý tưởng ở đây là có thể thao tác trên một nhóm đối tượng theo cách như thao tác trên một đối tượng duy nhất, các đối tượng của nhóm phải có các thao tác chung (method chung).  
Composite có thể được gọi là “Đối tượng đa hợp”.

* Composite là mẫu thiết kế dùng để tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp.
* Composite cho phép các client tác động đến từng đối tượng và các thành phần của đối tượng một cách thống nhất.

1. **Sơ đồ mẫu Composite UML:**

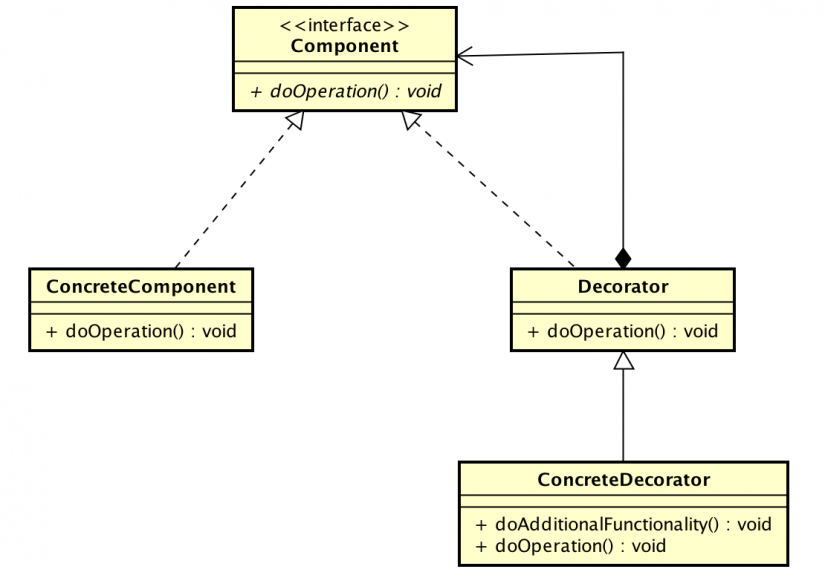


* + **Component** (Thành phần):
    - Khai báo interface hoặc abstract chung cho các thành phần đối tượng.
    - Chứa các method thao tác chung của các thành phần đối tượng.
  + **Leaf (Lá):**
    - Biểu diễn các đối tượng lá*(không có con)* trong thành phần đối tượng.
  + **Composite (Hỗn hợp):**
    - Định nghĩa một thao tác cho các thành phần có thành phần con.
    - Lưu trữ thành phần con.
    - Thực thi sự quản lý các thành phần con của giao diện Component.
  1. **Decorator:**

1. **Định nghĩa Decorator pattern:**

Là một trong nhiều những mẫu thiết kế cấu trúc thường được sử dụng. Mẫu thiết kế này sẽ 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.

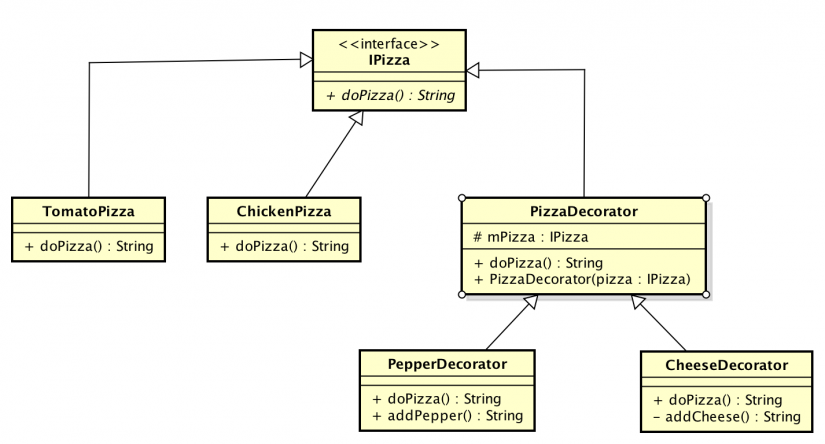
1. **Sơ đồ mẫu Composite UML:**



Những thành phần trong mẫu thiết kế Decorator:

* + - **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.

1. **Ví dụ:**

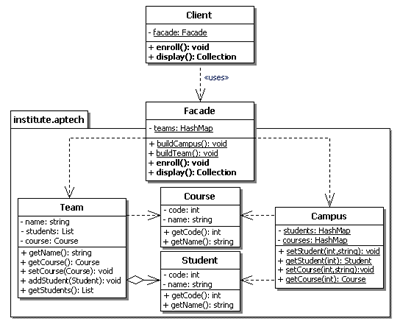


1. **Khi nào cần sử dụng:**
   * Động cơ của mẫu thiết kế Decorator là cho bạn mở rộng một hành vi của đối tượng một cách linh hoạt. Khả năng này là để gán hành vi mới một cách linh hoạt cho các đối tượng được thực hiện bới một lớp Decorator.
   * Mẫu thiết kế Decorator kết hợp tính đa hình với quyền ủy thác (delegation). Đó là tính đa hình với lớp gốc để cho các client có thể thực thi nó như lớp gốc. Trong hầu hết các trường hợp, những lời gọi phương thức được ủy thác cho lớp gốc và sau đó các kết quả được thực hiện bên trên, hoặc được *decorate*, với chức năng bổ sung. Decoration là một kỹ thuật linh hoạt vì nó thực hiện vào lúc runtime.
   1. **Façade:**
2. **Facade Pattern** là pattern cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem2). Façade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này vì chỉ cần giao tiếp với một giao diện chung duy nhất. **Façade Pattern** cho phép các đối tượng truy cập vào hệ thống con bằng cách sử dụng giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động trong hệ thống con.
3. **Vì sao phải sử dụng mẫu thiết kế này?**

Giả sử chúng ta xây dựng một lớp thư viện vậy Facade có thể:

* + - Giúp cho một thư viện của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.
    - Giúp cho các đoạn mã có sử dụng thư viện trở nên dễ đọc hơn, cũng lí do như trên.
    - Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.
    - Đóng gói tập nhiều hàm API được thiết kế không tốt bằng một hàm API đơn có thiết kế tốt hơn.

1. **Ví dụ:**



* 1. **Proxy**

1. **Proxy Pattern** là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class). Nếu như Factory Pattern giúp quản lý đối tượng tốt hơn thì **Proxy Pattern** có nhiệm vụ bảo vệ việc truy cập một đối tượng thông qua Proxy, hay còn gọi là truy cập gián tiếp. **Proxy Pattern** được sử dụng khi bạn muốn đơn giản một đối tượng phức tạp. Proxy được ủy quyền về phía ứng dụng khách cho phép tương tác với đối tượng đích theo những cách khác nhau, như gửi yêu cầu một dịch vụ nào đó, theo dõi trạng thái và vòng đời đối tượng, xây dựng lớp vỏ bảo vệ đối tượng…
2. **Phân loại:**

**Proxy Pattern** có những đặc điểm chung sau đây:

* Cung cấp mức truy cập gián tiếp vào một đối tượng.
* Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó.
* Cả Proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện. Mã máy dịch cho lớp giao diện thường “nhẹ” hơn các lớp cụ thể và do đó có thể giảm được thời gian tải dữ liệu giữa server và client.

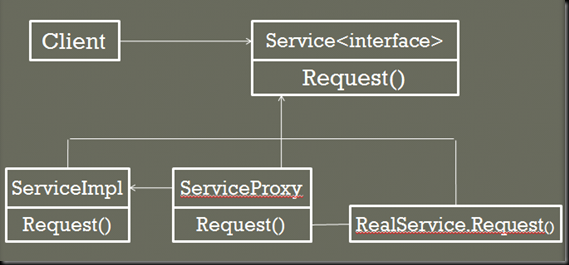
**Phân loại:**

* Remote Proxy: Client truy cập qua Remote Proxy để chiếu tới một đối tượng được bảo về nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác).
* Virtual Proxy: Virtual Proxy tạo ra một đối tượng trung gian mỗi khi có yêu cầu tại thời điểm thực thi ứng dụng, nhờ đó làm tăng hiệu suất của ứng dụng.
* Monitor Proxy: loại này sẽ thiết lập các bảo mật trên đối tượng cần bảo vệ, ngăn không cho client truy cập một số trường quan trọng của đối tượng.
* Protection Proxy: phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu.
* Firewall Proxy: bảo vệ đối tượng từ chối các yêu cầu xuất xứ từ các client không tín nhiệm.
* Cache Proxy: cung cấp không gian lưu trữ tạm thời cho các kết quả trả về từ đối tượng nào đó, kết quả này sẽ được tái sử dụng cho các client chia sẻ chung một yêu cầu gửi đến.
* Smart Reference Proxy: là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu.
* Synchronization Proxy: đảm bảo nhiều client có thể truy cập vào cùng một đối tượng mà không gây ra xung đột.

\*Khi một client nào đó chiếm dụng khóa khá lâu khiến cho số lượng các client trong danh sách hàng đợi cứ tăng lên, và do đó hoạt động của hệ thống bị ngừng trệ, có thể dẫn đến hiện tượng “tắc nghẽn”.

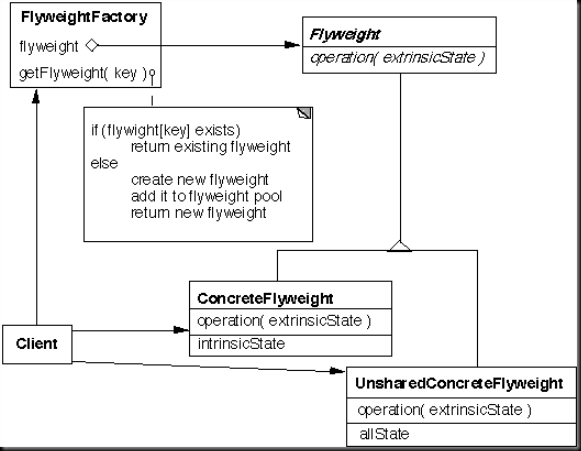
* Copy-On-Write Proxy: loại này đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn.

1. **Cấu trúc mẫu:**



* Service: là lớp đối tượng thuần ảo hay interface. Lớp này được thực thi bởi Proxy và các đối tượng khác.
* ServiceImpl: kế thừa từ Service, thực hiện đầy đủ hoặc mở rộng thêm các chức năng được định nghĩa trước tại Service.
* ServiceProxy: Là một Proxy được kế thừa từ Service. Nhiệm vụ của nó là chuyển lời gọi hàm từ Service đến ServiceImpl mỗi khi thích hợp.
  1. **Flyweight**
     1. **Flyweight** là một mẫu thiết kế phần mềm. Khi nhiều đối tượng phải được xử lý mà chương trình không thể chịu nổi một lượng dữ liệu khổng lồ, thì cần dùng **Flyweight**. Ý nghĩa: Làm phương tiện dùng chung để quản lý một số lượng lớn các đối tượng nhỏ có các đặc điểm chung, mà các đối tượng nhỏ này lại được sử dụng tùy thuộc vào hoàn cảnh, điều kiện ngoài. Đối với ngôn ngữ C++, trong mẫu **Flyweight**, dữ liệu không có các con trỏ (pointer) trỏ đến các phương thức của kiểu dữ liệu đó, vì như thế sẽ tốn rất nhiều bộ nhớ. Thay vào đó các chương trình con (subroutine) sẽ được gọi trực tiếp. Trong một vài trường hợp, **Flyweight** inheritance được thực hiện bằng cách “shift in” và “shift out” các data markers dưới dạng các chu trình tác vụ ở mức cao (higher level operation cycles) thông qua một mảng các Flyweight data.

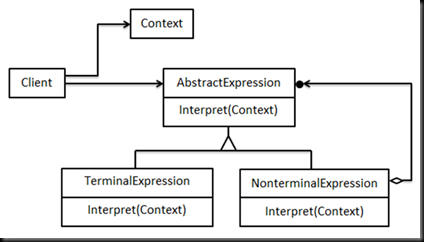
* + 1. **Cấu trúc:**



* **FlyweightFactory:** tạo ra và quản lý các đối tượng Flyweight.
* **Flyweight:** định nghĩa các phương thức chuẩn.
* **ConcreteFlyweight:** là các lớp thực thi của Flyweight, các thể hiện của cáclớp này sẽ được sử dụng tùy thuộc vào điều kiện ngoài.
* **UnsharedConreteFlyweight:** các thể hiện riêng.
* **Client:** duy trì một tham chiếu đến các Flyweight.

### **Group Behavioral Pattern**

* + - 1. **Interpreter Pattern**
  1. **Interpreter Pattern** giúp người lập trình có thể "xây dựng" những đối tượng "động" bằng cách đọc mô tả về đối tượng (file XML hoặc gì đó) rồi sau đó "xây dựng" đối tượng đúng theo mô tả đó.
  2. **Cấu trúc:**

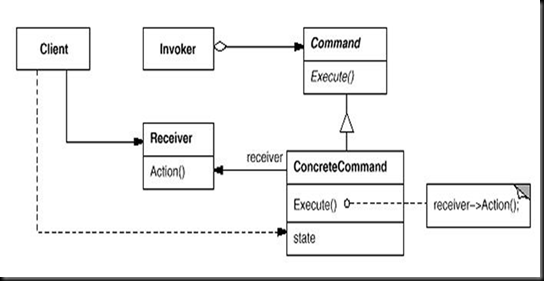


Mẫu gồm 2 thành phần chính có liên hệ với nhau là Context và Expression

* AbstractExpression: Xác định “tác vụ” mà ta có thể thực hiện được trên tất cả các nút trong cây cú pháp.
* TerminalExpression: Cài đặt tác vụ “thông dịch” cho những kí pháp nguyên tố của ngôn ngữ đặt tả.
* NonterminalExpression: Có thể chứa TerminalExpression bên trong và cũng có thể chứa một NonterminalExpression khác. Nó đóng vai trò như là “ngữ pháp” của ngôn ngữ đặc tả.
* Context: Là đối tượng thông tin để thực hiện thông dịch. Đối tượng này là toàn cục đối với quá trình thông dịch (dùng chung giữa các node).
  + - 1. **Command Pattern**
  1. **Command pattern** là một behavioral design pattern trong đó một đối tượng được sử dụng để đại diện và đóng gói tất cả các thông tin cần thiết để gọi một method. Thông tin này bao gồm tên method, các đối tượng của method đó và các giá trị cho các tham số của method.
  2. **Tình huống áp dụng:**

Dùng Command pattern khi:

* Tham chiếu đến một object.
* Xác định và thực hiện những yêu cầu tại những thời điểm khác nhau.
* Cần thực hiện thao tác Undo.
* Cần thực hiện thao tác Logging changes (trong trường hợp hệ thống bị treo).
* Cấu trúc hệ thống có dạng: điều khiển cấp cao được xây dựng trên những điều khiển nền tảng.
  1. **Cấu trúc và thành phần:**

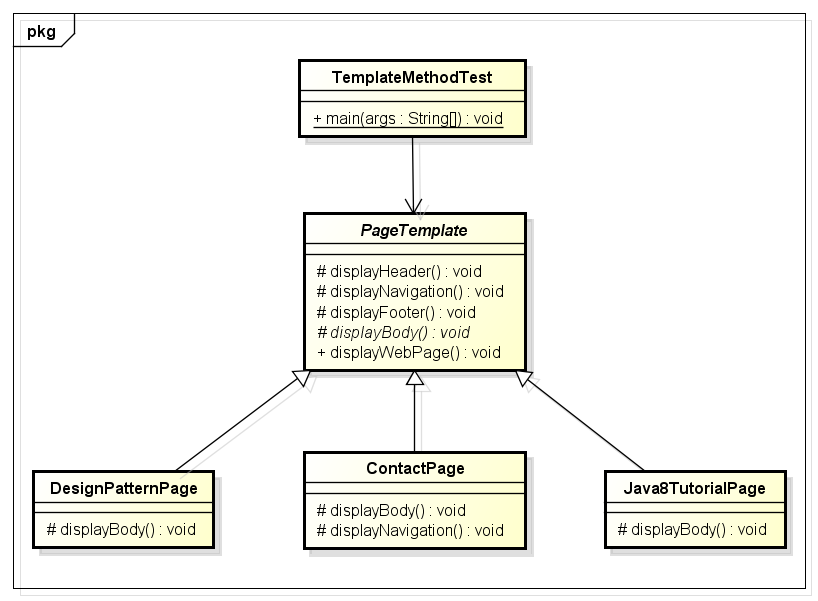


* **Command**: Trình bày giao diện để thực hiện Execute.
* **ConcreteCommand**: Thực thi Excute bằng cách gọi các thao tác tương ứng vào Receiver.
* **Client**: tạo ra một đối tượng ConcreteCommand và thiết lập receiver của nó.
* **Invoker**: gọi command thực hiện yêu cầu.
* **Receiver**: thực thi một thao tác kết hợp với thực thi yêu cầu.
  1. **Ưu điểm và hạn chế của Command Pattern:**

Mô hình Command có những ưu điểm sau:

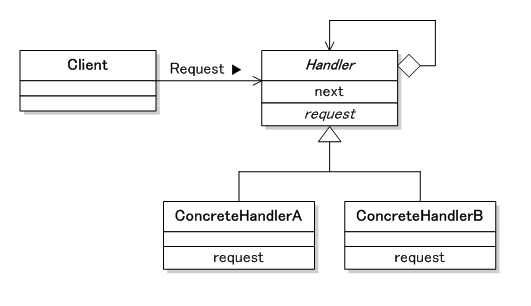
* Command Pattern tách riêng đối tượng và các điều khiển của đối tượng mà vẫn biết cách đáp ứng các request lên đối tượng đó.
* Các command là những lớp đối tượng cơ bản. Chúng có thể được vận dụng và mở rộng giống như bất kỳ đối tượng nào khác.
* Các Command có thể được tập hợp lại thành một Composite Command.
* Dễ dàng thêm vào một Command mới vì không cần phải chỉnh sửa lại các class sẵn có.
  + - 1. **Template Pattern**

1. **Mẫu Template method** định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu **Template Method** cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán, rất thích hợp cho việc tạo ra các chủng loại robot khác nhau.
2. **Ví dụ:**

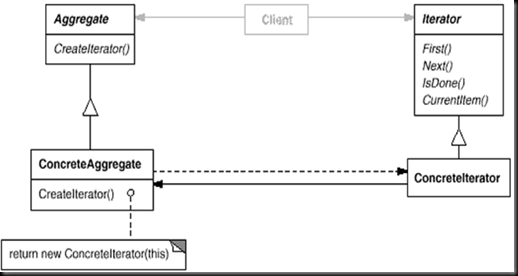


* + - 1. **Chain of Responsibility Pattern**

1. **Chain of Responsibility** là một mẫu cho phép một class nào đó thực hiện yêu cầu mà không cần biết về khả năng của các class khác. Nó tạo ra những liên kết không chặt chẽ giữa các class, và yêu cầu chỉ được chuyển qua các class thông qua các đường dẫn chung (common link). Yêu cầu sẽ được chuyển cho đến khi có 1 class thực hiện yêu cầu đó.
2. **Cấu trúc:**

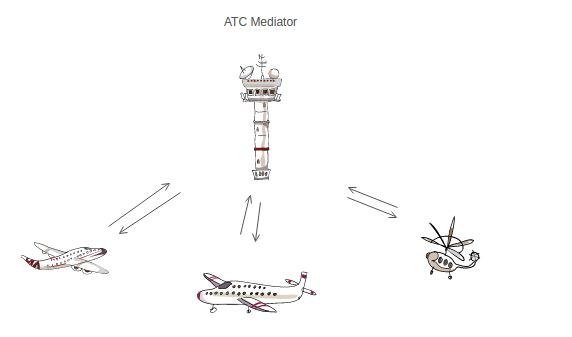


* **Handler:** Nó xác định các interface để xử lý yêu cầu. Có nghĩa là phân công nhiệm vụ xử lý cụ thể cho từng đối tượng tiếp nhận yêu cầu.
* **ConcreteHandlerA · B:** Thực hiện giao diện của "Handler". Xử lý yêu cầu hoặc nếu nó không xử lý được yêu cầu thì gửi yêu cầu đến đối tượng xử lý tiếp theo.
* **Client:** Áp dụng "Chain Of Responsibility", tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.
  + - 1. **Iterator Pattern**
  1. **Iterator** là một dạng vòng lặp. Đối với 1 danh sách các đối tượng không đồng nhất nhau thì việc đưa vào các danh sách như List, Sort, Map sẽ giúp cho việc duyệt qua danh sách dễ dàng mà không cần quan tâm đến sự hiện diện bên trong của danh sách.
  2. **Ứng dụng và cấu trúc**



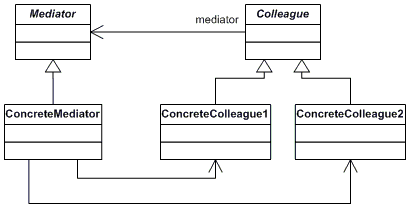
Thành phần tham gia:

* **Iterator**: định nghĩa interface cho việc truy cập và duyệt phần tử.
* **ConcreteIterator**: thi hành interface của Iterator, theo dõi vị trí hiện tại của quá trình duyệt tổ hợp.
* **Aggregate**: xây dựng interface cho việc tạo object Iterator.
* **ConcreteAggregate**: thi hành interface của Aggregate và trả về object của ConcreteIterator thích hợp. ConcreteIterator theo dõi vị trí đang duyệt của tổ hợp và tính toán được điểm tiếp theo của quá trình duyệt.
  1. **Hệ quả:**
* Hỗ trợ sự đa dạng trong việc duyệt một tổ hợp: Tổ hợp phức tạp có thể được duyệt bằng nhiều cách. Iterator giúp việc thay đổi thuật toán duyệt dễ dàng. Chỉ cần thay object Iterator này bằng object Iterator khác. Xây dựng tiếp được các class Iterator hỗ trợ các phương thức duyệt mới.
* Iterator rút gọn interface của Aggregate: interface của Iterator bỏ bớt yêu cầu phải có interface tương tự cho Aggregate, do đó làm interface của Aggregate đơn giản hơn.
* Nhiều phương pháp duyệt cùng trong quá trình đang giải quyết trên cùng một tổ hợp: Iterator tự quản lý tráng thái quá trình duyệt của nó. Do vậy, nhiều phương thức duyệt có thể xử lý cùng lúc được.
  + - 1. **Mediator Pattern**
         1. **Mediator Patern** (mô hình trung gian) được sử dụng để giảm sự phức tạp trong "giao tiếp" giữa các lớp và các đối tượng. Mô hình này cung cấp một lớp trung gian có nhiệm vụ xử lý thông tin liên lạc giữa các tầng lớp, hỗ trợ bảo trì mã code dễ dàng bằng cách khớp nối lỏng lẻo. Khớp nối lỏng lẻo ở đây được hiểu là các đối tượng tương đồng không "giao tiếp" trực tiếp với nhau mà giao tiếp thông qua người trung gian, và cho phép bạn thay thay đổi cách tương tác giữa họ một cách độc lập. (phần sau tôi sẽ lấy ví dụ để các bạn dể hiểu hơn). Thúc đẩy mối quan hệ nhiều - nhiều (many-to-many) giữa các đối tượng tượng tương đồng để đạt đến được trạng thái "full object".

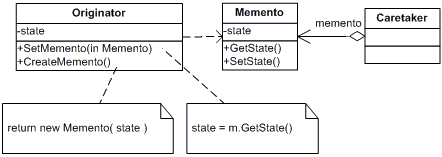


Tháp điều khiển tại sân bay có kiểm soát là một ví dụ về hoạt động của Mediator pattern. Các phi công của các máy bay đang cất cánh hoặc hạ cánh kết nối với tháp chứ không phải giao tiếp rõ ràng với nhau. Những khó khăn về việc ai có thể cất hoặc hạ cánh được thi hành bởi tháp điều khiển. Điều quan trọng cần lưu ý là tháp không kiểm soát toàn bộ chuyến bay. Nó tồn tại chỉ để thực thi các quy định an toàn trong lúc cất và hạ cánh.

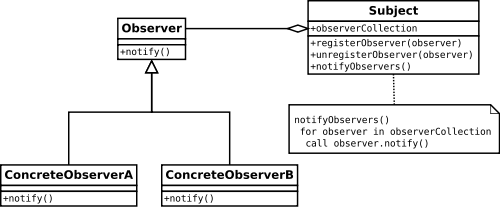
* + - * 1. **Cấu trúc**



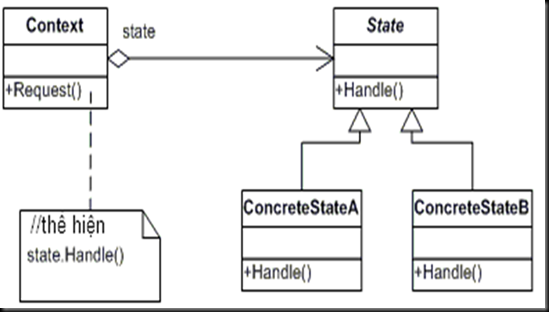
* **Mediator:** Mediator để xác định các interface giao tiếp giữa các đối tượng tương đồng.
* **ConcreteMediator:** Thực hiện interface giao tiếp từ Mediator, để điều chỉnh các thông tin liên lạc giữa các đối tượng . Thông báo cho các đối tượng tương đồng đang tồn tại, biết về mục đích giao tiếp.
* **ConcreteColleague:** Thông qua Mediator để giao tiếp với các đối tượng khác.
  + - * 1. **Nên áp dụng khi nào:**
* 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.
  + - * 1. **Ưu và nhược điểm:**
* 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. **Memento Pattern**
* **Mục đích**: Không xâm phạm tính đóng gói, lấy được và đưa ra những trạng thái trong của một đối tượng để nó có thể được khôi phục lại trạng thái đó sau này.
* **Động cơ thúc đẩy**: Đôi khi việc ghi lại trạng thái trong của một đối tượng là rất cần thiết. Việc này được yêu cầu khi thực thi việc kiểm tra và cơ chế Undo (cơ chế cho phép người dùng rút lại những thao tác không chắc chắn hoặc khôi phục lại do có lỗi). Bạn phải lưu thông tin về trạng thái ở một nơi nào đó để bạn có thể khôi phục lại đối tượng về trạng thái trước của nó. Nhưng các đối tượng thường đóng gói một vài hoặc tất cả trạng thái của chúng để làm cho các đối tượng khác không thể truy cập hoặc để cho những trạng thái của nó không thể bị lưu trữ bên ngoài. Việc phơi bày những trạng thái này sẽ vi phạm tính đóng gói, điều này có thể làm tổn hại đến tính tin cậy và tính mở rộng của ứng dụng. Chúng ta hãy xem xét ví dụ về trình soạn thảo đồ hoạ có hỗ trợ kết nối giữa các đối tượng. Một người dùng có thể kết nối hai hình chữ nhật bằng một đường thẳng và những hình chữ nhật này sẽ vẫn liên kết với nhau khi người dùng di chuyển một trong số chúng. Trình soạn thảo đảm bảo rằng đường thẳng trải ra để duy trì sự kết nối.
* **Tính ứng dụng**: Sử dụng mẫu Memento khi:
* Ảnh của một trạng thái đối tượng phải được lưu trữ để nó có thể được khôi phục sau này.
* Một giao diện trực tiếp để đạt được trạng thái sẽ để lộ chi tiết của sự thực thi và phá vỡ tính đóng gói của đối tượng.
* **Cấu trúc:**



* Originator:  khởi tạo đối tượng cần được save và restore trạng thái. Trong đó có một Memento class inner là private nên không thể truy cập từ class bên ngoài.
* Caretaker: là class helper lưu trữ và khôi phục lại *Originator* thông qua Memento object.
  + - 1. **Observer Pattern**
         1. **Có thể hiểu Observer** thuộc nhóm pattern Behavioral là một mẫu thiết kế dành cho việc một đối tượng khi thay đổi trạng thái của bản thân nó thì các đối tượng đính kèm theo cũng sẽ được thông báo. Trong trường hợp của EDP, một đối tượng phát nổ (trigger) lên một sự kiện, thì các listener được đính kèm sẽ lắng nghe và thực hiện (nếu có).
         2. **Cấu trúc:**



* **SUBJECT :**
* ObserverColection : Lưu trữ các observer đã được đăng ký
* registerObserver : Đăng ký một observer với subject.
* unregisterObserver : Hủy đăng ký một observer với subject.
* notifyObserver : Thông báo cho các observer đã đăng ký .
* **OBSERVER:** Là một interface được định nghĩa với một phương thức update() . Phương thức này sẽ nhận thay đổi mỗi khi có notify từ subject.
* **ConcreteObserver:** Kế thừa Observer Interface để implement phương thức update() để nhận thông báo từ Subject mỗi khi có notify .
* **ConcreteSubject:** Nhiệm vụ là lưu trữ status của các ConcreteObserver objects và từ các state này sẽ gửi đi thông báo mỗi khi state bị thay đổi
  + **setState optional -** set state cho Subject
  + **getState optional -** lấy state hiện tại của Subject
    - 1. **State Pattern**
         1. **State Pattern** là một trong những design pattern rất phổ biến trong phát triển game cũng như ứng dụng. Bài viết sẽ cung cấp các kiến thức cơ bản về State Pattern và cách thức hoạt động của nó.
         2. **Cấu trúc:**



* + - * 1. **Các trường hợp sử dụng State:**
* Mỗi hành vi của đối tượng phụ thuộc vào trạng thái, khi trạng thái thay đổi thì hành vi thay đổi.
* Trong môi trường ứng dụng lớn, có nhiều trạng thái và nhiều qui định cho mỗi trạng thái thì ta thường viết mỗi trạng thái rời ra thành từng lớp riêng biệt.
  + - 1. **Strategy Pattern**
         1. **Strategy pattern** (mẫu chiến lược): hiểu một cách đơn giản thì đây 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.
         2. **Cấu trúc:**

