**Creational patterns**

1. **Factory Method**

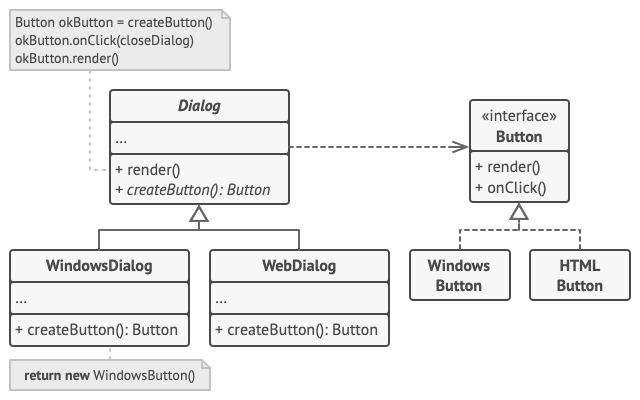
**Factory Method Design Pattern**hay gọi ngắn gọn là **Factory Pattern** là một trong những Pattern thuộc nhóm **Creational Design Pattern**. Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

**Factory Pattern** đúng nghĩa là một **nhà máy**, và nhà máy này sẽ “**sản xuất**” các đối tượng theo yêu cầu của chúng ta.

Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

**Demo**



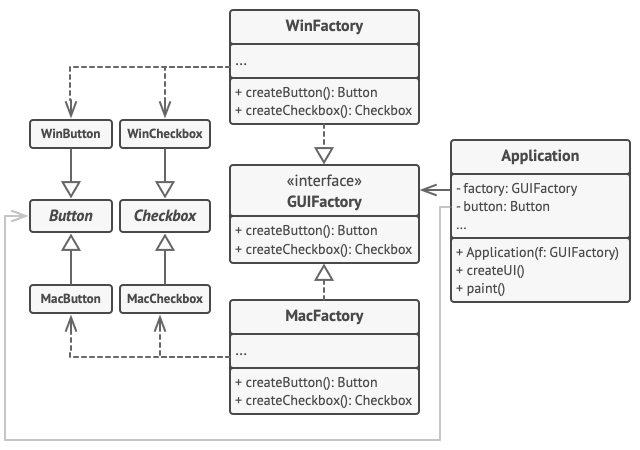
1. **Abstract Factory**

**Abstract Factory pattern** là một trong những **Creational pattern**. Nó là phương pháp tạo ra một Super-factory dùng để tạo ra các Factory khác. Hay còn được gọi là Factory của các Factory. Abstract Factory Pattern là một Pattern cấp cao hơn so với Factory Method Pattern.

Trong Abstract Factory pattern, một interface có nhiệm vụ tạo ra một Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Mỗi Factory được tạo ra có thể tạo ra các object bằng phương pháp giống như Factory pattern.

Hãy tưởng tượng, Abstract factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

**Demo**



1. **Builder**

**Builder pattern** là một trong những **Creational pattern**. Builder pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đôi tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng đôc lập với các đối tượng khác.

Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.

Có ba vấn đề chính với Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:

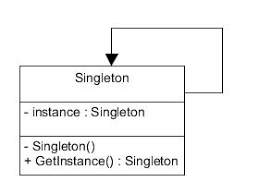
* Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.
* Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
* Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

Chúng ta có thể xử lý những vấn đề này với một số lượng lớn các tham số bằng việc cung cấp một hàm khởi tạo với những tham số bắt buộc và các method getter/ setter để cài đặt các tham số tùy chọn. Vấn đề với hướng tiếp cận này là trạng thái của Object sẽ không nhất quán cho tới khi tất cả các thuộc tính được cài đặt một cách rõ ràng. Nếu cần xây dựng một đối tượng Immutable thì cách này cũng không thể thực hiện được.

1. **Singleton**

**Singleton** là 1 trong 5 design pattern của nhóm **Creational Design Pattern**.

**Singleton** đảm bảo chỉ duy nhất **một thể hiện (instance)** được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

[](https://gpcoder.com/wp-content/uploads/2018/09/singleton-pattern.png)

Sử dụng Singleton khi chúng ta muốn:

1. Đảm bảo rằng chỉ có một instance của lớp.
2. Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
3. Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.
4. **Prototype**

**Prototype pattern** là một trong những **Creational pattern**. Nó có nhiệm vụ khởi tạo một đối tượng bằng cách **clone** một đối tượng đã tồn tại thay vì khởi tạo với từ khoá **new**. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, chúng ta có thể thay đổi dữ liệu của nó mà không ảnh hưởng đến đối tượng gốc.

Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.

Trong Java cung cấp mẫu prototype pattern này bằng việc implement interface [**Cloneable**](https://gpcoder.com/2361-object-cloning-trong-java/) và sử dụng method **clone()** để tạo object có đầy đủ thuộc tính của đối tượng ban đầu.

**Structural patterns**

1. **Adaptor**

Adapter Pattern (Người chuyển đổi) là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface 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 interface “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có interface không tương thích với lớp đang viết.

1. **Bridge**

**Bridge Pattern** là một trong những Pattern thuộc nhóm cấu trúc (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.

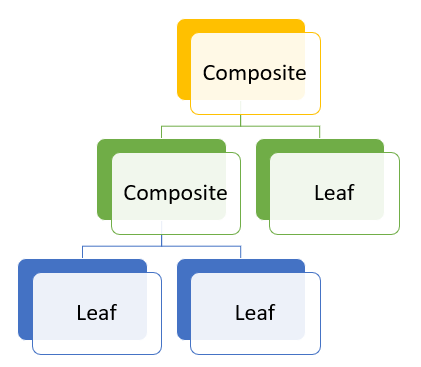
Điều đó có nghĩa là, ban đầu chúng ta thiết kế một class với rất nhiều xử lý, bây giờ chúng ta không muốn để những xử lý đó trong class đó nữa. Vì thế, chúng ta sẽ tạo ra một class khác và move các xử lý đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

**Bridge Pattern** khá giống với mẫu [**Adapter Pattern**](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/) ở chỗ là sẽ nhờ vào một lớp khác để thực hiện một số xử lý nào đó. Tuy nhiên, ý nghĩa và mục đích sử dụng của hai mẫu thiết kế này hoàn toàn khác nhau:

* **Adapter Pattern** hay còn gọi là **Wrapper pattern** được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được. Adapter Pattern giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.
* **Bridge Pattern** được sử dụng được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
* **Adapter Pattern**làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại). **Bridge Pattern** nên được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.

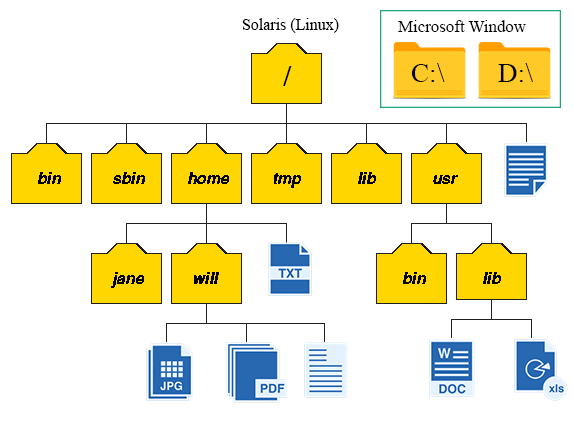
1. **Composite**

**Composite** là một mẫu thiết kế thuộc nhóm cấu trúc (**Structural Pattern**). Composite Pattern là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.

[](https://gpcoder.com/wp-content/uploads/2018/11/Composite.png)

Composite Pattern được sử dụng khi chúng ta cần **xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object**. Composite pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng 1 object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

**Ví dụ:** Một chương trình quản lý một hệ thống tập tin với cấu trúc như hình sau:

[](https://gpcoder.com/wp-content/uploads/2017/12/folder-stucture.png)

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). Một folder có thể chứa một hoặc nhiều file hoặc folder. Do đó, folder là một đối tượng phức tạp và file là một đối tượng đơn giản. File và Folder có nhiều thao tác và thuộc tính chung, chẳng hạn như: di chuyển (cut) , sao chép (copy), liệt kê (view) hoặc các thuộc tính thư mục như tên tệp và kích thước.

Với cấu trúc như vậy sẽ dễ dàng và thuận tiện hơn để quản lý file và folder thống nhất bằng cách xây dựng một Interface có đầy đủ các phương thức và thuộc tính chung cho cả file và folder.

1. **Decorator**

**Decorator pattern** là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

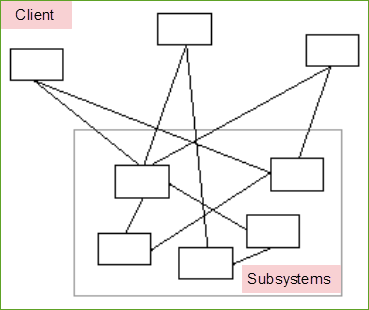
Decorator pattern sử dụng [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/#Cai_dat_Adapter_Pattern_nhu_the_nao) thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là **Wrapper**hay **Smart Proxy**.

1. **Façade**

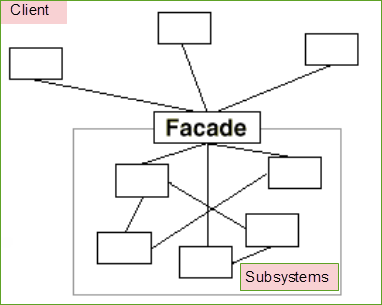
**Facade Pattern** là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Pattern này cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

**Facade Pattern** 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.

Ví dụ khi không có Facade, hệ thống của chúng ta trông giống như sau:

[](https://gpcoder.com/wp-content/uploads/2018/11/facade-1.png)

Khi có Facade, nó trông giống như sau:

[](https://gpcoder.com/wp-content/uploads/2018/11/facade-2.png)

**Façade Pattern** tương tự với [**Adapter Pattern**](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/). Hai Pattern này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. **Adapter Pattern** chuyển đổi mã nguồn để làm việc được với mã nguồn khác. Nhưng **Façade Pattern** cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

1. **Flyweight**

Trong một số tình huống trong phát triển phần mềm, chúng ta có thể cải thiện hiệu suất của ứng dụng với việc sử dụng **Cache**. Hãy tưởng tượng rất nhiều đối tượng được tạo ra và lãng phí bộ nhớ. Mô hình **Flyweight** được tạo ra để tránh vấn đề này và tối ưu hóa hiệu suất.

Flyweight Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ chúng hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp.

Flyweight Pattern được sử dụng khi chúng ta cần tạo một số lượng lớn các đối tượng của 1 lớp nào đó. Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp. Flyweight Pattern có thể được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy performance của hệ thống được tối ưu.

Flyweight object là **immutable**, nghĩa là không thể thay đổi khi nó đã được khởi tạo.

1. **Proxy**

**Proxy Pattern** là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern).

Proxy có nghĩa là “ủy quyền” hay “đại diện”. Mục đích xây dựng Proxy pattern cũng chính vì muốn tạo ra một đối tượng sẽ ủy quyền, thay thế cho một đối tượng khác.

Proxy Pattern là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class). Mẫu Proxy (người đại diện) đại diện cho một đối tượng khác thực thi các phương thức, phương thức đó có thể được định nghĩa lại cho phù hợp với múc đích sử dụng.

Để đơn giản hơn bạn có thể nghĩ đến khái niệm HTTP proxy trong mạng máy tính, nó là một gateway giữa trình duyệt (client) và máy chủ (server). HTTP proxy giúp nâng cao trải nghiệm người dùng, tăng tốc với lưu đệm các dữ liệu, loại bỏ các trang quảng cáo, giới hạn các vùng thông tin được xem… Proxy Pattern cũng có chung một mục đích như với HTTP proxy.

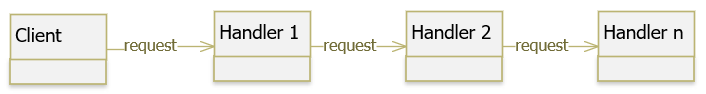
Proxy Pattern còn được gọi là **Surrogate** (thay thế) hoặc **Placeholder**(trình giữ chỗ)

**Behavioral patterns**

1. **Chain of Responsibility**

**Chain of Responsibility (COR)** là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Chain of Responsiblity cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.



Chain of Responsibility Pattern hoạt động như một danh sách liên kết ([Linked list](https://gpcoder.com/2555-linkedlist-trong-java/)) với việc [đệ quy](https://gpcoder.com/2327-giai-thuat-de-quy-trong-java/) duyệt qua các phần tử (recursive traversal).

1. **Command**

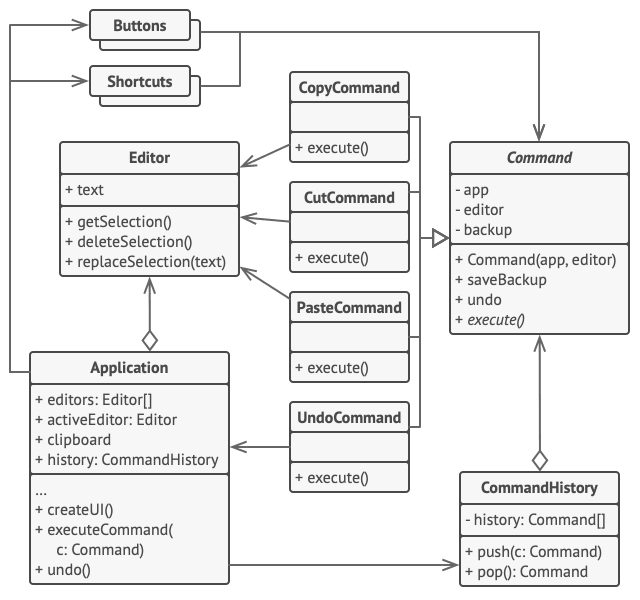
**Command Pattern** là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo/redo), transtraction.

Nói cho dễ hiểu, Command Pattern cho phép tất cả những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

Command dịch ra nghĩa là ra lệnh. Commander nghĩa là chỉ huy, người này không làm mà chỉ ra lệnh cho người khác làm. Như vậy, phải có người nhận lệnh và thi hành lệnh. Người ra lệnh cần cung cấp một class đóng gói những mệnh lệnh. Người nhận mệnh lệnh cần phân biệt những interface nào để thực hiện đúng mệnh lệnh.

Command Pattern còn được biết đến như là **Action** hoặc **Transaction**.

**Demo**

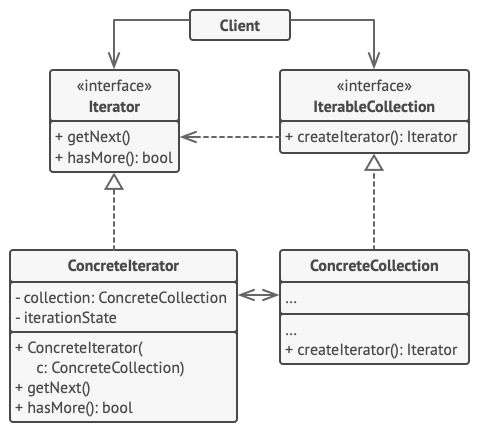


1. **Iterator**

Iterator là một Behavioral Pattern cho phép duyệt qua các phần tử của một tập hợp mà không để lộ biểu diễn cơ bản của nó (Ví dụ: danh sách, ngăn xếp, cây, v.v.). Nó được sử dụng để “Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này”.

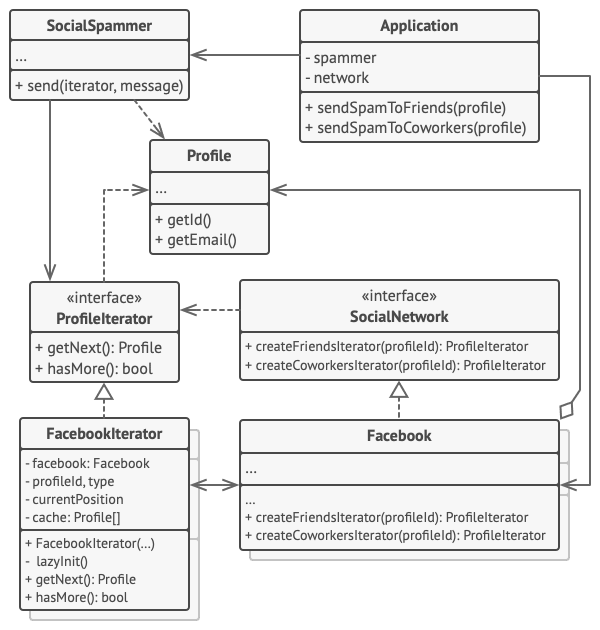
Nói cách khác, một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.

## **Structure**



1. **Iterator interface**: khai báo các hàm cần thiết để duyệt qua một tập hợp: tìm lấy phần tử tiếp theo, truy xuất vị trí hiện tại, v.v
2. **Concrete Iterator: triển khai các thuật toán cụ thể để duyệt qua một collection**
3. **Collection** **interface**: khai báo một hoặc nhiều phương thức tương thích giữa iterator với collection.
4. **Concrete Collection: trả về các instance mới của một lớp iterator cụ thể mỗi khi client yêu cầu**
5. **Client**: Làm việc với cả collections và iterators thông qua interfaces của chúng

Demo



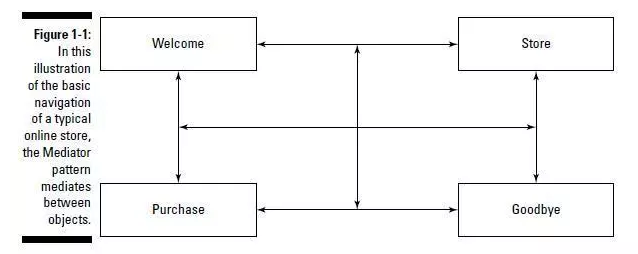
1. **Mediator**

**Mediator Pattern** là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Mediator có nghĩa là người trung gian. Pattern này nói rằng “Định nghĩa một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Mediator thúc đẩy sự khớp nối lỏng lẻo (loose coupling) bằng cách ngăn không cho các đối tượng đề cập đến nhau một cách rõ ràng và nó cho phép bạn thay đổi sự tương tác của họ một cách độc lập”.

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.

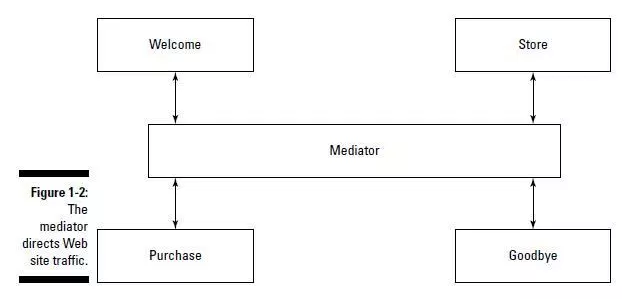
Khớp nối lỏng lẻo ở đây được hiểu là các đối tượng tương đồng không “giao tiếp” trực tiếp với nhau mà giao tiếp thông qua người trung gian, và nó cho phép thay thay đổi cách tương tác giữa chúng một cách độc lập.

Mediator Patern thúc đẩy mối quan hệ nhiều – nhiều (many-to-many) giữa các đối tượng tượng với nhau để đạt đến được kết quả mong muốn.



Hình bên trên là một ví dụ về mẫu thiết kế Mediator. Theo hình ta đang có một website với 4 trang. Website cho phép khách hàng duyệt qua kho hàng và đặt mua. Khách hàng có thể đi  từ trang này qua trang khác theo đường vẽ trên hình. Ở đây có một vấn đề phát sinh. Tại từng trang, bạn phải viết mã để nhận biết khi nào khách hàng muốn nhảy qua trang khác và kích hoạt trang đó. Tại một trang bạn có quá nhiều đường để đi tới trang khác, và vì vậy sẽ phát sinh nhiều đoạn code trùng lặp trên nhiều trang khác nhau.

Chúng ta có thể sử dụng mẫu Mediator để đóng gói tất cả các đường dẫn tới trang vào một module duy nhất, và đặt nó vào trong một đối tượng Mediator. Từ bây giờ, từng trang chỉ cần phải thông báo bất cứ sự thay đổi nào cho Mediator, và Mediator tự biết điều hướng trang cần thiết cho khách hàng, như trong hình bên dưới.



Chúng ta có thể tạo ra một Mediator với chức năng điều hướng trang. Tại đây chúng ta có thể chỉnh sửa và thay đổi dễ dàng. Đó chính là chức năng của Mediator (Người trung gian).

1. **Observer**

**Observer Pattern** là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc **một** – **nhiều** giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.

Observer Pattern còn gọi là **Dependents**, **Publish**/**Subcribe** hoặc **Source**/**Listener**.

1. **State**

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

1. **Strategy**
2. **Template Method**
3. **Visitor**