**Creational patterns**

-**Factory Method** “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.”

-Singleton “Singleton Pattern đảm bảo một lớp chỉ có một thực thể (instance) duy nhất được tạo ra và đồng thời cung cấp một truy cập toàn cục đến đối tượng được tạo ra”

+ EagerInitialization

+ LazyInitialization

+ StaticBlockInitialization

+ TheEnumWay

+ TheSolutionOfBillPugh

- Abstract Factory “Abstract Factory cung cấp một giao diện lớp có chức năng tạo ra một tập hợp các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không chỉ ra đó là những lớp cụ thể nào tại thời điểm thiết kế. Về bản chất, Abstract Factory Pattern chỉ khác Factory Pattern ở chỗ bản thân đối tượng không được chỉ ra cụ thể tại thời điểm thiết kế, tức nó là một giao diện hoặc lớp trừu tượng (interface, abstract). Nếu như Factory Patttern phân loại đối tượng dựa trên tham số đầu vào thì đối với Abstract Factory Pattern, thủ tục createObject() còn phụ thuộc thêm vào các yếu tố phụ khác như môi trường hệ điều hành chẳng hạn. Ứng với mỗi yếu tố phụ thứ hai ta có một lớp Factory cụ thể. “

-Prototype “[Prototype Pattern](http://en.wikipedia.org/wiki/Prototype_pattern) giúp cho việc khởi tạo object bằng một object nguyên mẫu (prototype), là copy của object “mẫu” đó. Ý tưởng của mẫu là chúng ta được cung cấp một object và sẽ dùng chính object này để như là một hình mẫu (template) khi cần tạo lập một đối tượng mới. Việc tạo lập object mới sẽ dựa trên object mẫu mà không sử dụng toán tử new hoặc constructor … được cụng cấp bởi ngôn ngữ lập trình. Lý do là chúng ta không biết được thông tin nội tại chi tiết bên trong object và object có thể che dấu và chỉ cung cấp ra bên ngoài một lượng thông tin giới hạn. Do vậy ta không thể dùng toán tử new và sao chép những dữ liệu được object cung cấp (vốn không đầy đủ) cho một object mới. Cách tốt nhất là để cho chính object “mẫu” tự xác định thông tin và dữ liệu sao chép.”

+ Shallow copy

+ Deep copy

- Builder “Builder Pattern là một mẫu thiết kế xây dựng một đối tượng phức tạp từng bước một từ một đối tượng đơn giản.  
  
**Trường hợp sử dụng và lợi ích:**

* Tạo đối tượng đơn phức tạp bằng cách chỉ rõ kiểu và nội dung của nó.Các đối tượng (được) xây dựng được bảo vệ từ trong chi tiết của quá trình khởi tạo.
* Muốn tách riêng tiến trình xây dựng đối tượng phức tạp ra khỏi phần tạo nên đối tượng.
* Tách riêng phần code khởi tạo và miêu tả
* Mang lại sự điều khiển uyển chuyển hơn trong tiến trình khởi dựng đối tượng.
* Liên quan tới mẫu (pattern):Abstract Factory, Composite

**Collectional patterns**

- Composite “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, hay còn gọi là [mẫu số chung nhỏ nhất](http://vi.wikipedia.org/w/index.php?title=M%E1%BA%ABu_s%E1%BB%91_chung_nh%E1%BB%8F_nh%E1%BA%A5t&action=edit&redlink=1).”

+ Individual Components (terminal nodes)

+ Composite Components (non-terminal nodes)

+ **Leaf** , **Composite** , **Component**

- Iterator “Cung cấp một cách để truy xuất các thành phần của một đối tượng hợp nhất một cách tuần tự mà không cho thấy representation bên dưới của nó” một Iterator được thiết kế cho phép bạn 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

+ Filtered iterators

+ Internal iterators

+ External iterators

**Flyweight** là một [mẫu thiết kế](http://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)) phần mềm. Khi nhiều đối tượng (objects) phải được xử lí mà chương trình không thể chịu nổi một lượng dữ liệu khổng lồ, thì cần dùng flyweight.

Mẫu thiết kế này gọi là “hạng ruồi” flyweight nguyên do thay vì phải làm việc với nhiều đối tượng độc lập, to lớn, bạn giảm bớt kích thước chúng bằng việc tạo một tập hợp các đối tượng dùng chung nhỏ hơn, gọi là flyweights mà bạn có thể cài đặt vào lúc thực thi chương trình để chúng trông giống như những đối tượng lớn hơn. Mỗi đối tượng to lớn có thể tiêu tốn nhiều tài nguyên hệ thống, bằng cách tách những điểm giống nhau của các đối tượng này, và dựa trên việc cấu hình thời gian thực để mô phỏng lại các đối tượng lớn, bạn đã làm giảm bớt gánh nặng lên tài nguyên hệ thống.

+ Intrinsic Information

+ Extrinsic Information

-**Visitor** là mẩu thiết kế([Design Patterns](http://vi.wikipedia.org/wiki/Design_Patterns)) cho phép định nghĩa các thao tác(operations) trên một tập hợp các đối tượng (objects) không đồng nhất (về kiểu) mà không làm thay đổi định nghĩa về lớp(classes) của các đối tượng đó. Để đạt được điều này, trong mẩu thiết kế visitor ta định nghĩa các thao tác trên các lớp tách biệt gọi các các lớp visitors, các lớp này cho phép tách rời các thao tác với các đối tượng mà nó tác động đến. Với mỗi thao tác được thêm vào, một lớp visitor tương ứng được tạo ra.

+ combine with Composite

-Decorator Decorator pattern chính xác là một design pattern thuộc nhóm Structural Pattern, Pattern này cho phép thêm một phương thức (method) hoặc thuộc tính (field, property) vào một đối tượng có sẵn một cách linh động. Hành động add này sảy ra lút runtime nên.  
Về bản chất, design pattern này sẽ tạo ra một lớp bao trùm lên đối tượng đã tồn tại, và khi sử dụng lớp mới này, ta có thể truy xuất các thuộc tính của đối tượng cũ cũng như có thêm những phương thức và thuộc tính mới ở lớp mới vừa được tạo ra.

-**Adapter** còn gọi là **đạng thức thiết kế điều hợp** điều chỉnh một [interface](http://vi.wikipedia.org/w/index.php?title=Interface_(khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh)&action=edit&redlink=1) của một [lớp](http://vi.wikipedia.org/wiki/L%E1%BB%9Bp_(khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh)) vào một interface khác mà người dùng mong đợi. Một *adapter* cho phép các lớp làm việc với nhau, mà bình thường là không thể do sự không tương thích về interfaces, bằng cách bao đóng interface của riêng nó cho phù hợp với một lớp có sẵn.

Có 2 loại **mẫu điều hợp** (*adapter pattern*): <http://vi.wikipedia.org/wiki/Adapter_pattern>

+Loại [*Object Adapter pattern*](http://vi.wikipedia.org/w/index.php?title=Object_Adapter_pattern&action=edit&redlink=1) *thường dùng interface*

+Loại [*Class Adapter pattern*](http://vi.wikipedia.org/w/index.php?title=Class_Adapter_pattern&action=edit&redlink=1) *thường dùng abstract*

*+adaptee*

*-Chain of responsibility:* <http://sinhvienluoi.blogspot.com/2012/09/chain-of-responsibility-pattern.html>

Giống như tên gọi của nó ... Chain of Responsibility là một mẫu thiết kế giải quyết cho việc **thực hiện 1 chuỗi các tác vụ có trình tự** mà mỗi 1 tác vụ trong chuỗi đó được đảm nhiệm bởi 1 class.