1.Abstract Factory

Sử dụng Abstract Factory khi code của bạn cần làm việc với các biến thể của các product liên quan, nhưng không muốn phụ thuộc vào concrete class của những product đó (Chúng có thể không được biết trước hoặc đơn giản là bạn muốn mở rộng trong tương lai).

Abstract Factory cung cấp cho bạn một interface cho việc tạo các objects từ mỗi class của dòng product. Miễn là code của bạn tạo objects thông qua interface này, bạn không phải lo lắng về việc tạo sai biến thể của product (sai ở đây có nghĩa là nó không khớp với bất kì product nào đã được tạo trong ứng dụng của bạn).

* Xem xét dùng Abtract Factory khi bạn có một class với một tập các Factory Method.
* Trong một chương trình được thiết kế tốt thì mỗi một class có trách nhiệm chỉ cho một việc. Khi một class giải quyết nhiều kiểu products, nó nên tách các factory method thành một factory class hay nói cách khách là sử dụng Abstract Factory

2.Factory Method

* Sử dụng Factory Method khi bạn không biết trước kiểu và các phụ thuộc của object mà code sẽ làm việc với nó.
* Sử dụng Factory Method khi bạn muốn cung cấp cho người dùng thư viện hoặc framework của bạn một cách để mở rộng các thành phần sẵn có bên trong nó.
* Sử dụng Factory Method khi bạn 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.

3.Singleton

Singleton được sử dụng khi:

* Có thể chắc chắn rằng một lớp chỉ có một instance
* Có khả năng truy cập đến instance từ mọi nơi (global access)
* Đối tượng singleton chỉ được khởi tạo duy nhất một lần khi nó được yêu cầu lần đầu.
* Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …
* Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,…

4.Prototype

* 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.
* Mẫu thiết kế Prototype tạo ra các bản sao của các đối tượng mẫu tồn tại trước đó. Cách tốt nhất để thực hiện việc này trong .NET là sử dụng iCloneable interface có sẵn trên các đối tượng được dùng như các kiểu mẫu. ICloneable interface có một phương thức gọi là Clone trả về một đối tượng là một bản sao của đối tượng gốc.
* Khi thực hiện chức năng Clone, bạn cần chú ý đến 2 kiểu khác nhau sau: deep copy và shallow copy. Shallow copy thì dễ dàng hơn nhưng chỉ sao chép các trường dữ liệu trong bản thân đối tượng – không phải các đối tượng mà kiểu mẫu đưa ra. Deep copy sao chép đối tượng kiểu mẫu và tất cả đối tượng nó đưa ra. Shallow copy thì dễ thực thi vì lớp Object có một phương thức MemberwiseClone trả về một shallow copy của đối tượng. Chiến lược sao chép cho deep copy có thể phức tạp hơn – một số đối tượng không được sao chép dễ dàng (chẳng hạn như Threads, các kết nối cơ sở dữ liệu, …).
* Tính linh hoạt nhờ vào sự đa xạ của phương thức clone(), ta có thể tạo lập 1 đối tượng mới mà ko hề cần viết mã gọi hàm tạo lập một cách cụ thể (việc cần tạo cụ thể 1 kiểu nào đó thì ta giao cho các quá trình tiền xử lý khác, và các quá trình tiền xử lý này hẳn cũng ko phụ thuộc vào hàm tạo lập cụ thể, một lần nữa ta lại được giải phóng…)
* Tránh việc tạo nhiều lớp con cho mỗi tượng như của Abstract Factory Pattern
* Giảm chi phí đáng kể so với tạo lập 1 đối tượng theo phương thức chuẩn, gọi toán tử new (gọi hàm tạo lập cụ thể)
* Ta có thể tùy chỉnh các thuộc tính đối tượng mẫu để tạo ra những đối tượng mẫu mới. Hay có thể hiểu là ta tạo ra một “chuẩn” class mới từ class có sẵn mà không cần viết code để định nghĩa.
* Ngoài ra còn rất nhiều lợi ích khác từ Design pattern: Prototype mà ta có thể ứng dụng vào trong việc lập trình

5.Adapter

Adapter được ứng dụng rộng rãi trong nhiều trường hợp, ta thường thấy Adapter xuất hiện trong những tình huống cần nâng cấp hệ thống cũ và có nhiều class cũ nhưng vẫn chứa method quan trọng, làm cho hệ thống hiệu quả hơn thông qua việc làm các component giao tiếp với nhau dù không liên quan đến nhau.

6.Bridge

Bridge được sử dụng khi:

* Khi muốn tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.
* Khi cả Abstraction và Implementation của chúng nên được mở rộng bằng subclass.
* Thay đổi trong thành phần được bổ sung thêm của một Abstraction mà không ảnh hưởng đối với các Client

7.Composite Pattern

* Khi bạn muốn tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp.
* 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, bạn cần tổng quát hóa quá mức interface thành phần khiến nó khó hiểu hơn.

8.Decorator

9.Facade

Dưới đây chúng ta có thể liệt kê một số trường hợp mà khi gặp sẽ phải cân nhắc sử dụng Facade pattern:

* *Muốn gom nhóm chức năng lại để Client dễ sử dụng.* Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
* *Giảm sự phụ thuộc.* Khi bạn muốn phân lớp các hệ thống con. Dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, do đó giúp giảm sự phụ thuộc của các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.
* *Tăng khả năng độc lập và khả chuyển*
* *Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt.* Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.
* *Đóng gói nhiều chức năng, che giấu thuật toán phức tạp.*
* *Cần một interface không rắc rối mà dễ sử dụng.*

Ví dụ: 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.

10.Proxy

Dưới đây chúng ta có thể liệt kê một số trường hợp mà khi gặp sẽ phải cân nhắc sử dụng Proxy pattern:

* *Lazy initialization (virtual proxy):* Khi bạn 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.
* *Access control (protection proxy):* Khi bạn muốn chỉ những khách hàng cụ thể mới có thể sử dụng đối tượng dịch vụ.
* *Local execution of a remote service (remote proxy):* Đây là khi đối tượng service được đặt trên một máy chủ từ xa.
* *Logging requests (logging proxy):* Khi bạn muốn giữ lịch sử của các yêu cầu đối với đối tượng service.
* *Caching request results (caching proxy):* Khi bạn 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.
* *Smart reference:* Khi bạn cần loại bỏ một đối tượng nặng khi không có máy khách nào sử dụng nó.

11.Command

Dưới đây chúng ta có thể liệt kê một số trường hợp mà khi gặp sẽ phải cân nhắc sử dụng Command pattern:

* Khi cần tham số hóa các đối tượng theo một hành động thực hiện (biến action thành parameter)
* Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau (delay action)
* Khi cần hỗ trợ tính năng undo, log, callback hoặc transaction
* Phối hợp nhiều Command với nhau theo thứ tự

12.State

Chúng ta sẽ áp dụng State Pattern trong các trường hợp như sau:

* Hành vi của một đội tượng phụ thuộc vào trạng thái của nó. Tại thời điểm runtime, khi đối tượng thực hiện hành vi, trạng thái của nó sẽ thay đổi theo.
* Đối tượng có nhiều trường hợp sử dụng với các hành vi của nó, nhiều hành vi phụ thuộc vào trạng thái của đối tượng. Hay nói cách khác, đối tượng có nhiều trạng thái, mỗi trạng thái có những hành vi khác nhau.

13.ObServer

Observer Pattern được áp dụng khi:

* Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ
* Cần mở rộng dự án với ít sự thay đổi nhất.

14.Strategy

* Muốn sử dụng các biến thể khác nhau của một xử lý trong một đối tượng và có thể chuyển đổi giữa các xử lý trong runtime.
* Khi có nhiều lớp tương đương chỉ khác cách chúng thực thi một vài hành vi.
* Khi muốn tách biệt business logic của một lớp khỏi implementation details của các xử lý.
* Khi lớp có toán tử điều kiện lớn chuyển đổi giữa các biến thể của cùng một xử lý.

15.Template Method

* 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).

16.MVC

Mô hình MVC được ứng dụng trong nhiều ngôn ngữ lập trình khác nhau, nhưng phổ biến nhất là ứng dụng ASP.NET MVC hay PHP MVC.

MVC đang là mô hình được ứng dụng rất nhiều trong lập trình.

Hệ thống MVC phát triển tốt sẽ cho phép phát triển front – end, back – end cùng trên hệ thống mà không có sự can thiệp, chia sẻ, chỉnh sửa các tập tin trong khi một hoặc hai bên vẫn đang làm việc.