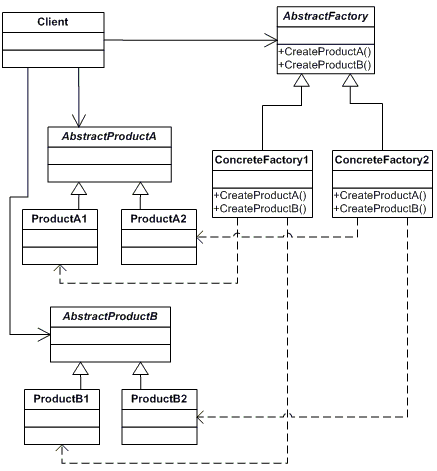
**MẪU THIẾT KẾ - CTK44**

1. **Abstract Factory**



1. **Mục đích**

- Abstract Factory Pattern giúp đảm bảo rằng các product mà bạn nhận được từ một factory đều tương thích với nhau.

- Abstract Factory Pattern giúp hạn chế sự phụ thuộc giữa creator và concrete products.

- Abstract Factory Pattern giúp gom các giai đoạn code tạo ra product vào một nơi trong chương trình, nhờ đó giúp dễ theo dõi và thao tác.

- Với Abstract Factory Pattern, chúng ta có thể thoải mái thêm nhiều loại product mới vào chương trình mà không làm thay đổi các đoạn code nền tảng đã có trước đó.

- Ví dụ, lấy trường hợp của một người quản lý điện thoại, người quản lý các số điện thoại. Số điện thoại tuân theo một quy tắc tạo mã cụ thể theo khu vực cho từng quốc gia. Nếu tại một thời điểm nào đó, ứng dụng phải được thay đổi để hỗ trợ thêm số tạo thành một quốc gia mới, mã của ứng dụng sẽ phải được thay đổi và nó sẽ ngày càng trở nên phức tạp hơn. Để ngăn chặn nó, Abstract Factory được sử dụng.

1. **Ý nghĩa**

- 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

1. **Áp dụng**

- Phía client sẽ không phụ thuộc vào việc những sản phẩm được tạo ra như thế nào.

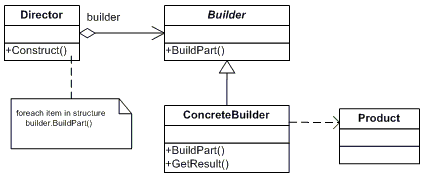
- Ứng dụng sẽ được cấu hình với một hoặc nhiều họ sản phẩm.

- Các đối tượng cần phải được tạo ra như một tập hợp để có thể tương thích với nhau

- Chúng ta muốn cung cấp một tập các lớp và chúng ta muốn thể hiện các ràng buộc, các mối quan hệ giữa chúng mà không phải là các thực thi của chúng(interface). Trong trường hợp không định nghĩa được các lớp trừu tượng Product thì việc tạo ra các ConcreteProduct theo mẫu này là rất khó hoặc không thể

- Mỗi khi có thêm một "sản phẩm" ta lại phải định nghĩa thêm một lớp "sản xuất"

1. **Builder**



1. **Mục đích**

Builder Pattern là một mẫu Design Pattern được tạo ra để cung cấp những giải pháp linh hoạt để giải quyết các vấn đề khác nhau xung quanh việc khởi tạo một đối tượng trong Lập trình hướng đối tượng. Mục đích chính của Builder Pattern chính là tách riêng việc khởi tạo một đối tượng ra khỏi biểu diễn của nó.

1. **Ý nghĩa**

Nhóm Pattern này giúp chúng ta:

- Giải quyết các công việc tạo và khởi tạo các đối tượng.

- Các mẫu sẽ tạo ra một cơ chế đơn giản, thống nhất khi tạo ra các thể hiện của đối tượng.

- Cho phép đóng gói các chi tiết về các lớp nào được khởi tạo và cách các thể hiện này được tạo ra. Vậy chúng ta sẽ bắt đầu cùng tìm hiểu và áp dụng Builder Pattern vào Swift.

1. **Áp dụng**

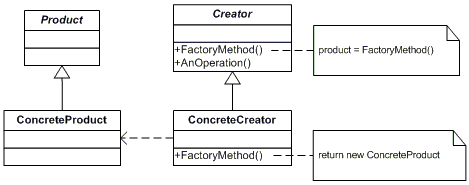
Builder được sử dụng khi:

- Sử dụng mẫu Builder để tránh sử dụng “telescopic constructor” (Gọi là telescopic constructor là vì khi một class có nhiều constructor với nhiều parameter trong constructor sẽ gây khó khăn cho người lập trình để nhớ và sử dụng cái nào cho đúng). Builder Pattern cho phép bạn xây dựng các object từng bước, chỉ sử dụng những bước bạn thực sự cần. Sau khi triển khai pattern, bạn không phải nhồi nhét hàng tá tham số vào các constructor của mình nữa.

- Sử dụng Builder Pattern khi bạn muốn code của mình có thể tạo các cách thể hiện khác nhau của một số sản phẩm (ví dụ: nhà bằng đá và bằng gỗ). Builder Pattern có thể được áp dụng khi việc xây dựng các bản trình bày khác nhau của sản phẩm bao gồm các bước tương tự chỉ khác nhau về chi tiết.

- Sử dụng Builder để tạo cây Composite hoặc các đối tượng phức tạp khác. Builder Pattern cho phép bạn tạo sản phẩm theo từng bước. Bạn có thể trì hoãn việc thực hiện một số bước mà không làm hỏng sản phẩm cuối cùng. Bạn thậm chí có thể gọi đệ quy các bước, điều này rất hữu ích khi bạn cần xây dựng một cây đối tượng. Một Builder không để lộ sản phẩm chưa hoàn thành khi đang chạy các bước xây dựng. Điều này ngăn không cho client code tìm nạp kết quả không đầy đủ.

1. **Factory Method**



1. **Mục đích**

Pattern này được sinh ra nhằm mục đích khởi tạo một đối tượng mới mà không cần thiết phải chỉ ra một cách chính xác class nào sẽ được khởi tạo.

1. **Ý nghĩa**

- Giúp việc khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đó. Người dùng không biết logic thực sự được khởi tạo bên dưới phương thức factory.

- Mẫu thiết kế này cho phép các lớp con chọn kiểu đối tượng cần tạo.

- Nó thúc đẩy sự liên kết lỏng lẻo bằng cách loại bỏ sự cần thiết phải ràng buộc các lớp cụ thể vào code. Nghĩa là code chỉ tương tác với interface hoặc lớp abstract, để nó sẽ làm việc với bất kỳ lớp nào implements interface đó hoặc extends lớp abstract.

- Factory Pattern giúp giảm sự phụ thuộc giữa các module: cung cấp 1 hướng tiếp cận với Interface thay vì các implement. Giúp chuơng trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client sẽ không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.

- Việc mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra những sub class và implement thêm vào factory method.

- Dễ dạng quản lý life cycle của các Object được tạo bởi Factory Method Pattern.

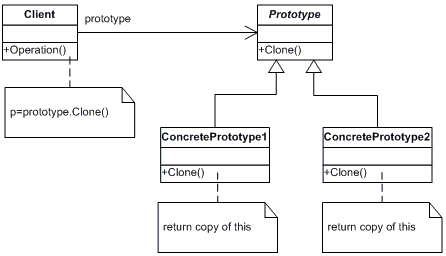
- Thống nhất về mặt naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

1. **Áp dụng**

“Factory method” thường được sử dụng trong việc thiết kế toolkit hay framework, đoạn mã của framework cần thiết phải tạo ra một đối tượng là những lớp con tương ứng tăng tính mềm dẻo trong sử dụng framework đó.

Cách sử dụng, khi nào bạn nên sử dụng mẫu thiết kế Factory Method.

1. **Prototype**



1. **Mục đích**

- Prototype là một design pattern thuộc nhóm Creational Pattern - những mẫu thiết kế cho việc khởi tạo object của lớp.

- Prototype quy định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này mà không làm cho code phụ thuộc vào các lớp của chúng.

- 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.

- Prototype Pattern cung cấp cơ chế để copy từ object ban đầu sang object mới và thay đổi giá trị một số thuộc tính nếu cần.

- Tần suất sử dụng: medium

1. **Ý nghĩa**

Tránh việc tạo nhiều lớp con cho mỗi đối tượng tạo như của Abstract Factory Pattern.

Giảm chi phí để tạo ra một đối tượng mới theo “chuẩn”, tức là việc này tăng Performance so với việc sử dụng từ khóa new để tạo đối tượng mới.

1. **Áp dụng**

- 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…)

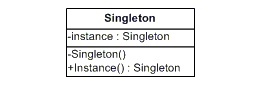
- Tránh việc tạo nhiều lớp con cho mỗi tượng như của Abstract Factory Pattern

- 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.

1. **Singleton**



1. **Mục đích**

Trong design pattern, mẫu thiết kế Singleton Pattern được dùng để đảm bảo chỉ có duy nhất một instance trong một class, và class đó sẽ cung cấp phương thức toàn cục để truy cập đến thực thể đó. Như vậy, Singleton Pattern được dùng khi bạn tạo ra một class mà bạn chỉ muốn chỉ có duy nhất một thực thể là instance của class đó và bạn có thể truy cập đến nó ở bất kỳ nơi đâu khi bạn muốn.

1. **Ý nghĩa**

"Tạo ra một đối tượng duy nhất và tồn tại trong suốt vòng đời của ứng dụng, nói cách khác, nó chỉ cho phép khởi tạo 1 lần duy nhất"

"Lý do phổ biến nhất cho việc này là để kiểm soát một số tài nguyên được chia sẻ, ví dụ, cơ sở dữ liệu."

"Hãy tưởng tượng rằng bạn đã tạo ra một đối tượng, và sau một thời gian, cố gắng tạo một đối tượng mới. Trong trường hợp này, bạn sẽ muốn nhận đối tượng cũ, thay vì tạo một thể hiện mới."

Lý do nó không thể được thực hiện với một hàm tạo: "Nó không thể được thực hiện với một hàm tạo thông thường vì mọi lời gọi hàm tạo luôn trả về một đối tượng mới theo thiết kế."

1. **Áp dụng**

"Hầu hết các đối tượng trong một ứng dụng đều chịu trách nhiệm cho công việc của chúng và truy xuất dữ liệu tự lưu trữ (self-contained data) và các tham chiếu trong phạm vi được đưa ra của chúng. Tuy nhiên, có nhiều đối tượng có thêm những nhiệm vụ và có ảnh hưởng rộng hơn, chẳng hạn như quản lý các nguồn tài nguyên bị giới hạn hoặc theo dõi toàn bộ trạng thái của hệ thống. Những nhiệm vụ của các đối tượng này thường yêu cầu chỉ có một đối tượng của một lớp. Các ví dụ bao gồm đối tượng request call API, đối tượng play âm thanh trong ứng dụng game, ... Những phần khác trong ứng dụng phụ thuộc vào những đối tượng đặc biệt này và cần có cách để tìm ra chúng. Đây là nơi mà mẫu thiết kế Singleton được sử dụng.

Bạn gặp một sự cố về hiệu năng hệ thống. Cùng một thời điểm, các bạn đang sử dụng một lúc nhiều đối tượng và chúng làm tiêu tốn quá nhiều tài nguyên của hệ thống. Đây là vấn đề mà bạn cần phải khắc phục, và Singleton pattern có thể giúp bạn thực hiện được điều đó."

"Đối với một vài class thì việc chỉ có duy nhất một instance (object) là rất quan trọng. Bạn có thấy là các chương trình thường xuyên có một tập tin cấu hình duy nhất không. Nó không phải là bất thường đối với một chương trình cho phép bạn biết làm thế nào nó được thực hiện thông qua một file log duy nhất. Hay Giao diện ứng dụng thường xuyên có một cửa sổ chính, và họ thường lấy đầu vào từ bàn phím một cách chính xác. Hay nhiều ứng dụng khác thì cần phải nói chuyện với một cơ sở dữ liệu chính xác. Nếu bạn chỉ có một instance của một class và rất nhiều mã cần truy cập đến instance đó, thì thật là ngớ ngẩn khi chuyển object từ method này sang method khác. Làm thế nào để chúng ta đảm bảo một class chỉ có một thể hiện duy nhất và dễ dàng truy nhập được? Một object toàn cục có thể làm cho nó dễ dàng truy nhập được nhưng không ngăn cấm bạn tạo thêm nhiều object khác. Một giải pháp tốt hơn là làm cho class đó tự bản thân nó có thể điều khiển được thực thể chính của nó. Class đó có thể đảm bảo rằng không có thực thể khác được tạo ra (bằng cách chặn đứng các yêu cầu tạo object mới) và cung cấp cách thức để truy nhập được thực thể chính của nó. Đó là singleton pattern."