**Seminar team iOS**

Nội dung: Một số design pattern thường dùng trong iOS

Tác giả: Nguyễn Đức Hà

Ngày thực hiện: 23/06/2016

# Giới thiệu về design pattern

Trong khi thiết kế các framework của Cocoa và Cocoa Touch, Apple đã áp dụng rất nhiều các design pattern. Một số design pattern tiêu biểu như MVC, Singleton, Delegation,... Tuy nhiên không phải ai cũng nhận ra những dòng code mình sử dụng đang áp dụng design pattern gì. Bài viết này sẽ trình bày các kiến thức cơ bản về design pattern nói chung và các design pattern thường được áp dụng trong lập trình iOS nói riêng, qua đó giúp lập trình viên hiểu và áp dụng các design pattern vào dự án của mình.

## Khái niệm design pattern

Design pattern được nhắc đến lần đầu tiên trong cuốn sách *Design Patterns: Elements of Reusable Object-Oriented Software* của bốn tác giả: Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides (thường được biết đến với tên “Gang of Four”). Định nghĩa vắn tắt của design pattern được đề cập trong cuốn sách là “Giải pháp cho một vấn đề trong một ngữ cảnh”. Ngữ cảnh ở đây được hiểu là tình huống thường xảy ra, có thể áp dụng design pattern vào đó. Vấn đề là mục tiêu cần đạt được trong ngữ cảnh đó. Như vậy có thể định nghĩa design pattern như sau: design pattern là giải pháp đã được chứng minh tính hiệu quả, có thể áp dụng cho các vấn đề chung, lặp lại trong một ngữ cảnh nhất định.

## Các lợi ích của design pattern

* Sử dụng design pattern giúp cho việc trao đổi về dự án ngắn gọn và dễ hiểu hơn. PM chỉ cần nói sử dụng singleton là dev sẽ hiểu mà không cần giải thích quá nhiều.
* Tiết kiệm thời gian nghiên cứu với những vấn đề chung thường gặp phải. Design pattern đã cung cấp giải pháp cho những vấn đề đó, giải pháp đó đã được chứng minh là hiệu quả.
* Design pattern là giải pháp ở tầng thiết kế, nó có thể được áp dụng vào bất kì ngôn ngữ hướng đối tượng nào.
* Giúp cho quá trình cài đặt và bảo trì thuận tiện và dễ dàng hơn.

# Phân loại design pattern

Design pattern được chia làm 3 loại, tương ứng với chức năng của nó: creational patterns, structural patterns, behavior patterns.

## Creational patterns

Các pattern thuộc loại này được dùng để khởi tạo một đối tượng nhưng che giấu phần logic khởi tạo và không sử dụng hàm khởi tạo trực tiếp (alloc init hoặc new). Sử dụng design pattern này giúp cho việc tạo đối tượng được linh hoạt hơn.

Một số creational pattern phổ biến: singleton, factory, abstract factory, builder,…

## Structural patterns

Đây là các pattern liên quan đến quan hệ giữa các đối tượng. Nó giúp cho việc xác định quan hệ giữa các đối tượng trở nên đơn giản hơn.

Một số structural pattern phổ biến: adapter, facade, decorator, bridge,..

## Behavior patterns

Các pattern này liên quan đến việc giao tiếp giữa các đối tượng, làm tăng tính linh hoạt khi thực hiện giao tiếp.

Một số behavior pattern phổ biến: observer, command, memento,…

# Một số design pattern thường dùng trong iOS

Phần này của bài viết sẽ giới thiệu một số design pattern thường được sử dụng trong iOS, bao gồm định nghĩa, cách cài đặt và ý nghĩa của pattern đó.

## Singleton pattern

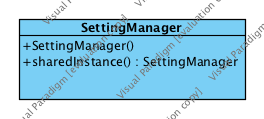
Singleton pattern đảm bảo rằng một class chỉ có duy nhất một instance và có thể được truy cập một cách toàn cục (các class khác đều sử dụng chung một instance đó). Class luôn theo dõi instance duy nhất mà nó tạo ra và đảm bảo rằng không có thêm instance nào được tạo ra nữa.

Một số thư viện Cocoa Touch sử dụng singleton pattern: NSFileManager, NSURLSession, UIApplication, NSUserDefaults,…

Trường hợp có thể áp dụng singleton pattern: khi cần truy cập đến các tài nguyên toàn cục như setting, session.

Hạn chế khi sử dụng singleton: instance được tạo khi sử dụng singleton không thể được copy, retain và release.

Sơ đồ UML của singleton pattern:



Tạo singleton sử dụng Objective-C:

+ (instancetype)sharedInstance {

static SettingManager \*instance = nil;

static dispatch\_once\_t once\_token;

dispatch\_once(&once\_token, ^{

instance = [[SettingManager alloc] init];

});

return instance;

}

Tạo singleton sử dụng Swift 2:

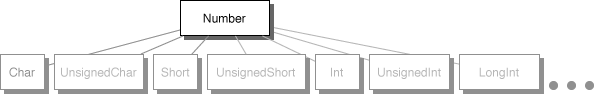
static let sharedInstance = SettingManager()

## Abstract Factory pattern

Abstract Factory còn được biết đến với tên gọi khác là Class cluster. Class cluster pattern nhóm các class con riêng biệt vào một class cha (class con là private còn class cha là public). Class cha sẽ cung cấp interface để tạo ra các class con mà không cần phải xác định class con riêng biệt. Logic phức tạp để tạo ra class con sẽ được che giấu khỏi client.

Class cluster là design pattern khá phổ biến trong Cocoa và Cocoa Touch. Một số class áp dụng Class Cluster: NSNumber, NSString, NSMutableString, NSData, NSMutableData, NSArray, NSMutableArray, NSDictionary, NSMutableDictionary.

Kiến trúc Class cluster sử dụng trong class NSNumber

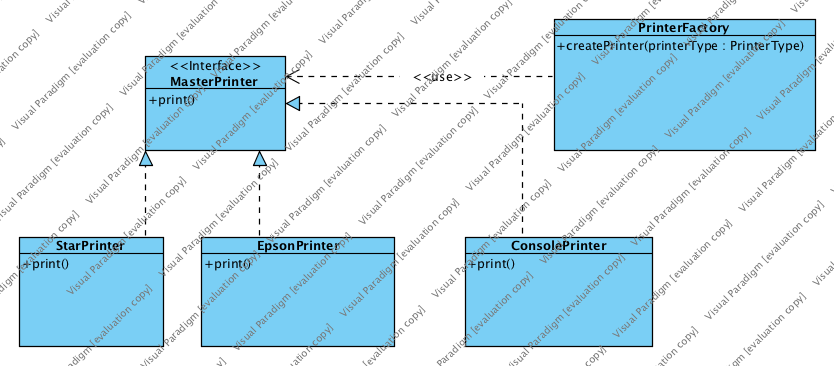


Trong kiến trúc này, NSNumber là class cha được public, các class con như Char, Short, Int,… được giấu. Client sẽ sử dụng các hàm khởi tạo để tạo ra instance của class con như:

[Code]

Trường hợp áp dụng Class cluster: khi có một nhóm các class liên quan đến nhau hoặc phụ thuộc vào nhau, có thể sử dụng class cluster để khởi tạo và giảm độ phức tạp khi sử dụng đối với client.

Sơ đồ UML:



Cài đặt Abstract Factory:

* Tạo protocol MasterPrinter và các class StarPrinter, EpsonPrinter, ConsolePrinter implement protocol này:

MasterPrinter:

@protocol MasterPrinter <NSObject>

@property (nonatomic, weak) id<PrinterDelegate> delegate;

- (void)print;

@end

StarPrinter:

- (void)print {

NSLog(@"Start printing to Star printer");

[self printWithCompletion:^(NSString \*message) {

}];

}

- (void)printWithCompletion:(void (^)(NSString \*message))completion {

dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, 5 \* NSEC\_PER\_SEC), dispatch\_get\_main\_queue(), ^{

completion(@"Print to Star printer successfully");

});

}

EpsonPrinter:

- (void)print {

NSLog(@"Start printing to Epson printer");

[self printWithCompletion:^(NSString \*message) {

}];

}

- (void)printWithCompletion:(void (^)(NSString \*message))completion {

dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, 5 \* NSEC\_PER\_SEC), dispatch\_get\_main\_queue(), ^{

completion(@"Print to Epson printer successfully");

});

}

ConsolePrinter:

- (void)print {

NSLog(@"Start printing to console");

[self printWithCompletion:^(NSString \*message) {

}];

}

- (void)printWithCompletion:(void (^)(NSString \*message))completion {

dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, 5 \* NSEC\_PER\_SEC), dispatch\_get\_main\_queue(), ^{

completion(@"Print to console successfully");

});

}

* Tạo PrinterFactory:

+ (id<MasterPrinter>)createPrinter:(PrinterType)printerType {

switch (printerType) {

case PrinterTypeEpson:

return [[EpsonPrinter alloc] init];

case PrinterTypeStar:

return [[StarPrinter alloc] init];

default:

return [[ConsolePrinter alloc] init];

}

}

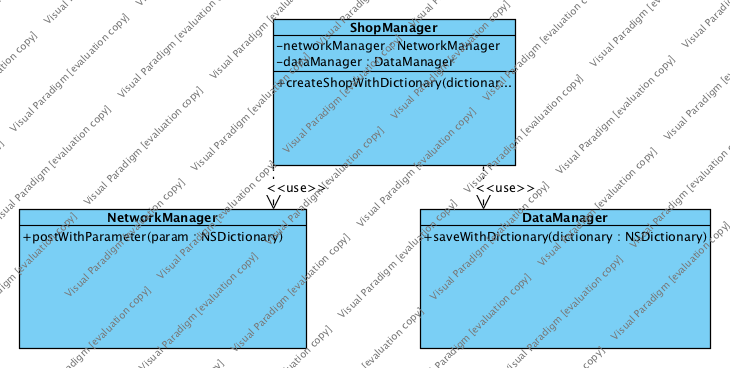
## Facade

Facade cung cấp một interface chung cho một tập các interface trong hệ thống. Façade định nghĩa interface ở mức cao hơn, giúp cho việc sử dụng trở nên dễ dàng, đồng thời che giấu sự phức tạp của hệ thống đối với client.

Trong Cocoa Touch, UIImage là class sử dụng Façade pattern. Ví dụ hàm + (id)imageNamed: (NSString \*)name khởi tạo một UIImage với tên ảnh trong bundle. Hàm này làm nhiều chức năng như load ảnh từ bundle, vẽ ảnh lên image context, đồng thời che giấu sự phức tạp trong việc khởi tạo các định dạng ảnh khác nhau như PNG, JPEG, PDF,… Người dùng chỉ cần truyền tên ảnh và class UIImage sẽ tự nhận biết được định dạng ảnh và khởi tạo tương ứng.

Trường hợp sử dụng Façade pattern: khi hệ thống có các hàm phức tạp, không để người dùng gọi trực tiếp thì có thể sử dụng Façade để tạo ra các API đơn giản hơn cho người dùng sử dụng.

Sơ đồ UML:



Cách sử dụng Façade:

* Class NetworkManager và DataManager:

- (void)postWithParameter:(NSDictionary \*)param;

- (void)saveWithDictionary:(NSDictionary \*)dictionary;

* Class ShopManager:

Class ShopManager chứa instance của 2 class trên và tạo một API đóng gói API của 2 class trên. API này sẽ được cung cấp cho client.

- (void)createShopWithDictionary:(NSDictionary \*)dictionary {

[self.dataManager saveWithDictionary:dictionary];

if (![SettingManager sharedInstance].offlineEnable) {

[self.networkManager postWithParameter:dictionary];

}

}

## Decorator

Decorator pattern là pattern cho phép thêm các hành vi, chức năng vào một đối tượng mà không làm thay đổi code của đối tượng đó. Đây là phương thức thay thế cho việc tạo ra class con.

Trong iOS, có hai cách cài đặt phổ biến Decorator pattern là: Category (Extension trong Swift) và Delegation.

* Category (Extension)

Category (Extension) là cơ chế của Objective-C (Swift), cho phép thêm các hàm vào một object mà không phải tạo ra class con. Người dùng có thể dùng category (extension) cho bất kì class nào bao gồm cả các class không được phép truy cập code như UIView, NSString, UIImage,…

Các hàm trong category sẽ được thêm vào ở compile-time. Khác với decorator pattern thông thường, category không đóng gói instance của class được mở rộng.

Một số chú ý khi sử dụng Category:

* Không thể thêm biến vào một class sử dụng category.
* Không viết đè một hàm của class vì nó có thể gây ra ảnh hưởng không mong muốn.
* Delegation

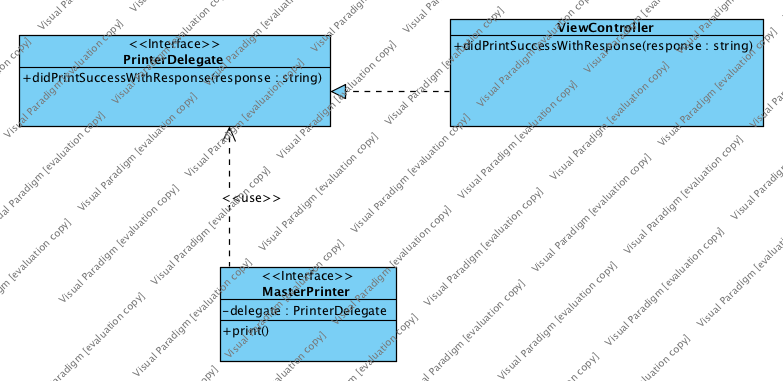
Delegation là một cách cài đặt khác của Decorator pattern. Đây là cơ chế trong đó một đối tượng đóng vai trò thay mặt cho hoặc phối hợp với một đối tượng khác. Đối tượng chủ (delegating object) giữ một liên kết yếu (weak reference) đến một đối tượng khác (delegate object) và gửi thông điệp đến ở thời điểm thích hợp.

DataSource cũng là cách sử dụng Delegation pattern giống như Delegate. Sự khác nhau giữa DataSource và Delegate là ở mối quan hệ với delegating object. Trong khi Delegate thay mặt delegating object điều khiển giao diện người dùng thì DataSource thay mặt để điều khiển dữ liệu.

Ví dụ với UITableView:

* UITableView là delegating object. UITableView chứa liên kết yếu là delegate đến đối tượng khác.
* ViewController là delegate object, ở đây implement các hàm của UITableViewDelegate và UITableViewDataSource. ViewController sẽ đóng vai trò thay thế UITableView trong việc xác định các giá trị như numberOfRowsInSection, numberOfSection, cellForRowAtIndexPath,…

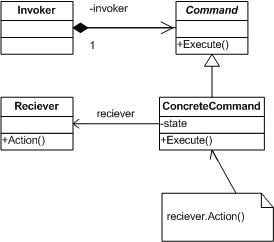
Sơ đồ UML:



## Command

Command design pattern đóng gói một request thành một đối tượng. Request được đóng gói này sẽ trở nên linh hoạt hơn so với request ban đầu, nó có thể được truyền giữa các đối tượng, lưu trữ, thay đổi và đưa vào hàng đợi. Để cài đặt pattern này trong iOS, có hai cách là sử dụng cơ chế Target – Action và Invocation.

Sơ đồ UML của Command pattern:



* Target – Action

Đây là cơ chế cho phép một đối tượng điều khiển (kế thừa từ UIControl như UIButton, UISlider,…) truyền thông điệp tới một đối tượng có thể hiểu và xử lý được thông điệp đó. Cơ chế Target – Action thường được áp dụng để xử lý các sự kiện của UIControl như Touch up inside, Value changed,…

Target còn gọi là receiver là đối tượng nhận thông điệp, có thể là custom class của UIView hoặc UIViewController. Target object sẽ cài đặt phương thức tương ứng được mô tả trong action.

Action là thông điệp được UIControl object truyền đến target, đó cũng chính là phương thức mà target cài đặt, tương ứng với thông điệp của action. Action được xác định bởi selector (định danh duy nhất ở runtime của một phương thức).

Ví dụ sử dụng Target – Action trong Objective-C và Swift: bắt sự kiện tap vào button và xử lý trong ViewController:

[button addTarget:self action:@selector(buttonDidTap:) forControlEvents:UIControlEventTouchUpInside];

button.addTarget(self, action: #selector(buttonDidTap:), forControlEvents: .TouchUpInside)

* Invocation

Invocation trong iOS sử dụng lớp NSInvocation. Một instance của NSInvocation chứa target object, method selector và các tham số của method. NSInvocation có thể thay đổi được và thực thi khi cần thiết. Invocation cho phép hoàn tác các thao tác nếu muốn.

Invocation được sử dụng trong việc quản lý hoàn tác, chuyển tiếp thông điệp và timer, nó cũng được sử dụng khi muốn tách đối tượng gửi thông điệp và đối tượng nhận thông điệp.

Ví dụ cài đặt Invocation trong Objective-C: áp dụng vào việc quản lý hoàn tác khi xóa object. Khi xóa object, tạo một instance của NSInvocation để đóng gói command, truyền target (receiver), action (message) và argument. Vì đây là command xử lý hoàn tác nên action sẽ là thêm object (ngược với sự kiện xóa object ở trên).

- (void)deleteObjectAtIndex:(NSInteger)index {

NSString \*deletedObject = self.objects[index];

NSMethodSignature \*signature = [self methodSignatureForSelector:@selector(addObject:atIndex:)];

NSInvocation \*undoInvocation = [NSInvocation invocationWithMethodSignature:signature];

[undoInvocation setSelector:@selector(addObject:atIndex:)];

[undoInvocation setTarget:self];

//Index will start from 2

[undoInvocation setArgument:&deletedObject atIndex:2];

[undoInvocation setArgument:&index atIndex:3];

[undoInvocation retainArguments];

[self.undoStack addObject:undoInvocation];

[self.objects removeObjectAtIndex:index];

}

## Observer

Observer pattern định nghĩa một quan hệ một – nhiều giữa các đối tượng, qua đó khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng liên quan cũng được thông báo và cập nhật tự động. Observer pattern hoạt động tương tự mô hình publish – subcribe, trong đó đối tượng chính và các observer của nó có quan hệ không chặt chẽ. Quan hệ không chặt chẽ này thể hiện ở việc observing và observed object không biết nhiều thông tin về nhau nhưng vẫn có thể giao tiếp được.

Hai cách phổ biến để cài đặt Observer pattern là NSNotification và Key-Value Observing (KVO).

* NSNotification

NSNotification dựa trên mô hình publish – subcribe như đã nói ở trên, trong đó publisher là đối tượng gửi thông điệp đến các đối tượng khác (subcriber). Publisher không cần biết thông tin gì về subcriber. NSNotification được sử dụng rất phổ biến trong iOS.

Mục đích chính của notification là thông báo cho các đối tượng khác về một sự kiện để đối tượng đó có thể phản hồi thích hợp.

Đối tượng nhận được notification chỉ có thể phản ứng sau khi sự kiện đã diễn ra, nghĩa là nó không thể ảnh hưởng trực tiếp đến sự kiện.

Một số notification được định nghĩa sẵn như UIKeyboardWillShowNotification, UIKeyboardWillHideNotification, UIApplicationDidEnterBackgroundNotification. Có thể tự tạo notification bằng cách sử dụng NSNotificationCenter như sau:

* Khi object có thay đổi và muốn thông báo sự thay đổi này:

[[NSNotificationCenter defaultCenter] postNotificationName:name object:object userInfo:info];

* Observer đăng ký để nhận thông báo:

[[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(selector:) name:name object:object];

* KVO

Trong KVO, một đối tượng có thể yêu cầu được thông báo sự thay đổi của một property nhất định (có thể là property của chính nó hoặc của đối tượng khác). KVO khác với notification ở một điểm quan trọng là nó không sử dụng một đối tượng trung tâm để thông báo đến các observer mà thay vào đó thông báo được chuyển trực tiếp đến observing object.

Cách sử dụng KVO với Objective-C:

* Thêm observer cho thuộc tính cần được theo dõi sử dụng hàm addObserver:forKeyPath:options:context:
* Observer implement hàm [observeValueForKeyPath:ofObject:change:context:](file:///Applications/Xcode.app/Contents/Developer/Documentation/DocSets/com.apple.adc.documentation.iOS.docset/Contents/Resources/Documents/documentation/Cocoa/Reference/Foundation/Protocols/NSKeyValueObserving_Protocol/index.html#//apple_ref/occ/instm/NSObject/observeValueForKeyPath:ofObject:change:context:) để nhận notify khi thuộc tính thay đổi và xử lý.

# Reference

<https://developer.apple.com/legacy/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW6>

<https://www.raywenderlich.com/46988/ios-design-patterns>

<https://www.raywenderlich.com/86477/introducing-ios-design-patterns-in-swift-part-1>

<https://sourcemaking.com/design_patterns>

<http://www.tutorialspoint.com/design_pattern/>

<http://www.appcoda.com/understanding-key-value-observing-coding/>