TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

———————— * ————————

Chuyên đề Công Nghệ Phần Mềm

Áp dụng Design Pattern vào dự án

Người hướng dẫn :

ThS. Vũ Đình Hồng

Người thực hiện :

Nguyễn Ngọc Cường - 51303020
Nguyễn Nghĩa Dinh - 51303254

TP. HỒ CHÍ MINH
Ngày 4 tháng 12 năm 2016

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

———————— * ————————

Chuyên đề Công Nghệ Phần Mềm

Áp dụng Design Pattern vào dự án

Người hướng dẫn :

ThS. Vũ Đình Hồng

Người thực hiện :

Nguyễn Ngọc Cường - 51303020
Nguyễn Nghĩa Dinh - 51303254

TP. HỒ CHÍ MINH
Ngày 4 tháng 12 năm 2016

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm đồ án của riêng tôi / chúng tôi và được sự hướng dẫn của ThS. Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo. Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc. Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 4 tháng 12 năm 2016

Nguyễn Ngọc Cường

Nguyễn Nghĩa Dinh

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

_____
_____
_____
_____
_____
_____
_____
_____
_____

Hồ Chí Minh , Ngày ___ tháng ___ năm ___

(Kí và ghi họ tên)

Phần hướng dẫ của GV chấm bài

_____
_____
_____
_____
_____
_____
_____
_____
_____

Hồ Chí Minh , Ngày ___ tháng ___ năm ___

(Kí và ghi họ tên)

Mục lục

Chương 1

Sơ lược Design Pattern

1.1 Design Pattern là gì ?

Pattern mô tả một giải pháp chung đối với một vấn đề nào đó trong thiết kế thường được "lặp lại" trong nhiều dự án. Nói một cách khác, một pattern có thể được xem như một "khuôn mẫu" có sẵn áp dụng được cho nhiều tình huống khác nhau để giải quyết một vấn đề cụ thể. Trong bất kỳ hệ thống phần mềm hướng đối tượng nào chúng ta cũng có thể bắt gặp các vấn đề lặp lại.

Design patterns có thể xem là một kĩ thuật lập trình hướng đối tượng (OOP) nâng cao, áp dụng một cách triệt để các tính chất của OOP như bao đóng (encapsulation), trừu tượng (abstract), kế thừa (inheritance), đa hình (polymorphism) cũng như các khái niệm như object, class, interfaces, generic...để đưa ra các mẫu thiết kế hoàn thiện.

1.2 Đặc điểm chung của Design Pattern

- **Pattern được hiểu theo nghĩa tái sử dụng ý tưởng hơn là mã lệnh**
  Pattern cho phép các nhà thiết kế có thể cùng ngồi lại với nhau và cùng giải quyết một vấn đề nào đó mà không phải mất nhiều thời gian tranh cãi. Trong rất nhiều trường hợp, dự án phần mềm thất bại là do các nhà phát triển không có được sự hiểu biết chung trong các vấn đề về kiến trúc phần mềm. Ngoài ra, pattern cũng cung cấp những thuật ngữ và khái niệm chung trong thiết kế. Nói một cách đơn giản, khi đề cập đến một pattern nào đấy, bất kỳ ai biết pattern đó đều có thể nhanh chóng hình dung ra "bức tranh" của giải pháp. Và cuối cùng, nếu áp

dụng pattern hiệu quả thì việc bảo trì phần mềm cũng được tiến hành thuận lợi hơn, nắm bắt kiến trúc hệ thống nhanh hơn.

- **Pattern hỗ trợ tái sử dụng kiến trúc và mô hình thiết kế phần mềm theo quy mô lớn**

  Cần phân biệt design pattern với framework. Framework hỗ trợ tái sử dụng mô hình thiết kế và mã nguồn ở mức chi tiết hơn. Trong khi đó, design pattern được vận dụng ở mức tổng quát hơn, giúp các nhà phát triển hình dung và ghi nhận các cấu trúc tĩnh và động cũng như quan hệ tương tác giữa các giải pháp trong quá trình thiết kế ứng dụng đối với một chuyên khu riêng biệt.

- **Pattern đa tương thích:**

  Pattern không phụ thuộc vào ngôn ngữ lập trình, công nghệ hoặc các nền tảng lớn như J2EE của Sun hay Microsoft .NET Framework.

  Tiềm năng ứng dụng của pattern là rất lớn. Các thiết kế dựa trên pattern được sử dụng khá nhiều ở các phần mềm mã nguồn mở, trong nền tảng J2EE hoặc .NET... Trong các dạng ứng dụng này, có thể dễ dàng nhận ra một số tên lớp chứa các tiền tố hoặc hậu tố như Factory, Proxy, Adapter...

1.3 Phân loại Pattern

Pattern được phân loại ra làm 3 nhóm chính sau đây:

- **Nhóm cấu thành (Creational Pattern):**

  Gồm Factory, Abstract Factory, Singleton, Prototype, Builder... Liên quan đến quá trình khởi tạo đối tượng cụ thể từ một định nghĩa trừu tượng (abstract class, interface).

- **Nhóm cấu trúc tĩnh (Structural Pattern):**

  Gồm Proxy, Adapter, Wrapper, Bridge, Facade, Flyweight, Visitor... Liên quan đến vấn đề làm thế nào để các lớp và đối tượng kết hợp với nhau tạo thành các cấu trúc lớn hơn.

- **Nhóm tương tác động (Behavioral Pattern):**

  Gồm Observer, State, Command, Iterator... Mô tả cách thức để các lớp hoặc đối tượng có thể giao tiếp với nhau.

Chương 2

Các Design Pattern

2.1 Adapter Pattern

2.1.1 Định Nghĩa

Adapter Pattern biến đổi giao diện của một lớp thành một giao diện khác mà các đối tượng client có thể hiểu được. Lớp với giao diện được tạo ra đó gọi là Adapter. Nguyên tắc cơ bản của Adapter Pattern nằm ở chỗ làm thế nào để các lớp với các giao diện không tương thích có thể làm việc được với nhau.

Nguyên lý xây dựng Adapter Pattern khá đơn giản: chúng ta xây dựng một lớp với một giao diện mong muốn sao cho lớp đó giao tiếp được với một lớp cho trước ứng với một giao diện khác.

Adapter Pattern không quản lý tập trung các đối tượng gần giống nhau như Factory Pattern, mà kết nối với nhiều lớp không có liên quan gì với nhau. Ví dụ lớp A sau khi thực thi giao diện của nó và vẫn muốn bổ sung các phương thức từ một lớp B nào đó, chúng ta có thể kết nối A với B thông qua hình thức kế thừa hoặc liên kết đối tượng như một thành phần. Adapter Pattern có sự giống nhau một chút với Proxy Pattern ở chỗ nó tận dụng tối đa tính chất "uỷ quyền" (delegation); lớp Adapter sẽ kết nối với một đối tượng nào đó gọi là Adaptee và Adapter sẽ được uỷ quyền truy cập vào Adaptee, lớp Adapter đóng vai trò như một kênh trung gian để client truy cập vào một số các thành phần quan trọng của lớp Adaptee.

2.1.2 Đặt điểm

Adapter Pattern hướng tập trung vào giải quyết sự tương thích giữa hai giao diện đang tồn tại, giảm công sức viết lại mã lệnh xuống mức tối thiểu có thể được.

Tái sử dụng giao diện cũ và Adapter Pattern chỉ thực sự cần thiết khi mọi thứ đã được thiết kế từ trước.

2.1.3 Phạm vi ứng dụng

Adapter Pattern được ứng dụng trong các trường hợp:

Cần tích hợp một vài module vào chương trình. Không thể sát nhập trực tiếp module vào chương trình (ví dụ như module thư viện đã được dịch ra .DLL, .CLASS...).

Module đang tồn tại không có giao diện mong muốn như: Cần nhiều hơn phương thức cho module đó. Một số phương thức có thể được nạp chồng.

2.2 Wrapper Pattern

2.2.1 Định Nghĩa

Wrapper Pattern là một trường hợp đặc biệt của Adapter Pattern. Nếu một Adapter chỉ đơn thuần là "nhúng" (wrap) các lớp với các giao diện không tương thích với nhau để chúng có thể hoạt động cùng nhau thì có thể được gọi bằng tên riêng Wrapper Pattern. Khi đó lớp Adapter còn được gọi là lớp Wrapper. Đây là quan hệ "có một", tức là một giao diện không tương thích có thể được nhúng vào thành một phần của một giao diện khác.

2.2.2 Đặc điểm

Đối tượng Wrapper mô phỏng tất cả các hành vi (hàm, thủ tục) của giao diện được nhúng bởi các hành vi với tên y hệt. Thí dụ nếu lớp được nhúng A có thủ tục SpecificRequest() thì lớp Wrapper cũng phải có thủ tục SpecificRequest() tham chiếu đến thủ tục cùng tên của A. (Ngoài ra đối tượng Wraper có thể được bổ sung các phương thức khác nếu cần thiết). Đặc điểm

này được đưa ra dựa trên nguyên tắc thiết kế "Law of Demeter" nói rằng không nên tham chiếu một đối tượng sâu hơn một lớp.

Các phương thức trong Adaptee được "nhúng" trong Wrapper bằng cách truyền lời gọi cùng với các tham số tới phương thức tương ứng trong Adaptee, và trả về kết quả giống như vậy. Các thành viên (thuộc tính, trường, sự kiện) được nhúng trong Wrapper có tính chất giống hệt như trong các lớp được nhúng (tên, kiểu dữ liệu, phạm vi truy cập...).

Từ các đặc điểm ở trên, có thể thấy rằng Wrapper Pattern cho phép một module chương trình tương tác được trong một môi trường khác biệt với môi trường phát triển của module đó (ví dụ C++ và Java).

2.3 Strategy Pattern

2.3.1 Định Nghĩa

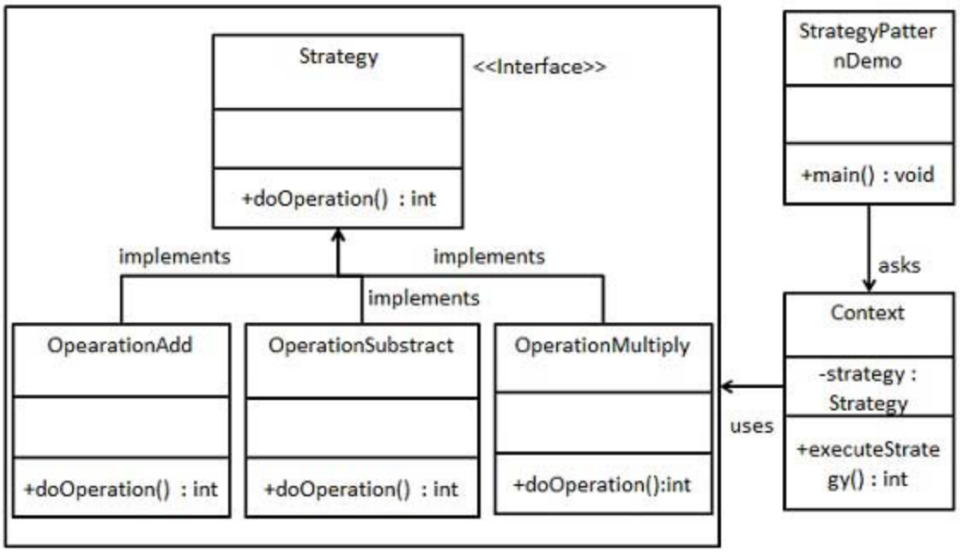
Strategy là mẫu thiết kế dùng để định nghĩa một họ các thuật toán, đóng gói mỗi thuật toán đó và làm cho chúng có khả năng thay đổi dễ dàng.

2.3.2 Ứng dụng của Strategy Pattern

Trong phát triển phần mềm, đôi lúc ta có gặp một số tình huống mà ở đó một số class nó chỉ khác nhau ở thuật toán để thực hiện công việc. Thay vì phải tạo ra nhiều classes gần như là giống nhau thì ta sẽ chỉ tách những phần thuật toán khác nhau ra thành những class khác.

2.3.3 Ví dụ Strategy Pattern

Ví dụ:



Step 1

Create an interface.

Strategy.java

```
public interface Strategy {
   public int doOperation(int num1, int num2);
}
```

Step 2

Create concrete classes implementing the same interface.

OperationAdd.java

```
public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 + num2;
   }
}
```

OperationSubstract.java

```
public class OperationSubstract implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 - num2;
   }
}
```

OperationMultiply.java

```
public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Step 3

Create *Context* Class.

Context.java

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Step 4

Use the *Context* to see change in behaviour when it changes its *Strategy*.

StrategyPatternDemo.java

```
public class StrategyPatternDemo {
   public static void main(String[] args) {
      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}
```

Step 5

Verify the output.

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

2.4 Command Pattern

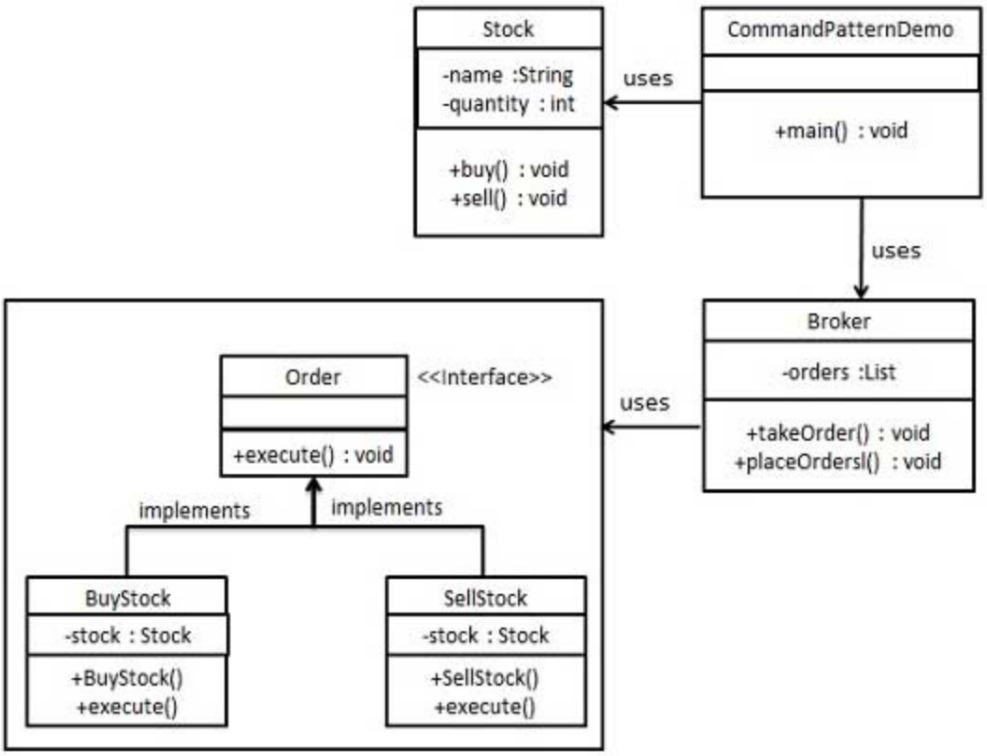
2.4.1 Định Nghĩa

Command Pattern là một Design Pattern trong đó nó được dùng để đóng gói các yêu cầu khác nhau thành từng đối tượng riêng biệt, qua đó cho phép ta tham chiếu đến client với những yêu cầu khác nhau.

Mẫu Command đóng gói yêu cầu như một đối tượng, làm cho nó có thể được truyền như 1 tham số, và được lưu trữ lại theo những cách thức khác nhau.

2.4.2 Ví dụ Command Pattern

Ví dụ:



Step 1

Create a command interface.

Order.java

```
public interface Order {
   void execute();
}
```

Step 2

Create a request class.

Stock.java

```
public class Stock {

   private String name = "ABC";
   private int quantity = 10;

   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```

Step 3

Create concrete classes implementing the *Order* interface.

BuyStock.java

```
public class BuyStock implements Order {
   private Stock abcStock;

   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}
```

SellStock.java

```
public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

Step 4

Create command invoker class.

Broker.java

```
import java.util.ArrayList;
import java.util.List;

   public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){

      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

Step 5

Use the Broker class to take and execute commands.

CommandPatternDemo.java

```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();

      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);

      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);

      broker.placeOrders();
   }
}
```

Step 6

Verify the output.

```
Stock [ Name: ABC, Quantity: 10 ] bought
Stock [ Name: ABC, Quantity: 10 ] sold
```

2.5 Khác biệt giữa Wrapper Pattern và Adapter Pattern

Sự khác biệt giữa Wrapper và Adapter nằm ở mục đích sử dụng: Adapter Pattern định hướng cho một đối tượng đang tồn tại có thể làm việc được với các đối tượng khác và biến đổi logic theo một cách thức nào đó, trong khi Wrapper Pattern chỉ đơn thuần cung cấp một giao diện kết hợp các đối tượng được xây dựng từ cùng một ngôn ngữ hoặc khác ngôn ngữ, trên cùng một hệ điều hành hoặc trên những hệ điều hành khác nhau.

Chương 3

Một số design pattern trong iOS

3.1 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.

Tạo singleton sử dụng Objective-C:

```
+ (instancetype)sharedInstance {
    static SettingManager *instance = nil;
    static dispatch_once_t once_token;
    dispatch_once(&amp;amp;amp;amp;amp;amp;once_token, ^{
        instance = [[SettingManager alloc] init];
    });

    return instance;

}
```

Tạo singleton sử dụng Swift 2:

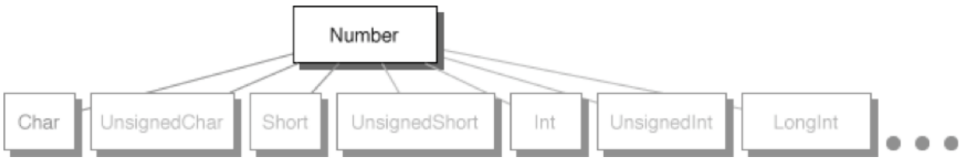
```
static let sharedInstance = SettingManager()
```

3.2 Abstract Factory pattern

bstract 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



Kiến trúc class cluster sử dụng trong 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:

```
[ NSNumber   numberWithInt:1]
[ NSNumber   numberWithBool:  YES ]
```

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.

Cài đặt Abstract Factory:

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

```
MasterPrinter:

@protocol MasterPrinter &amp;amp;amp;amp;lt;NSObject&amp;amp;amp;amp;gt;;
@property (nonatomic, weak) id&amp;amp;amp;amp;lt;PrinterDelegate&amp;amp;amp;amp;gt; 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&amp;amp;amp;amp;amp;lt;MasterPrinter&amp;amp;amp;amp;amp;gt;)createPrinter:(PrinterType)
printerType {
    switch (printerType) {
    case PrinterTypeEpson:
        return [[EpsonPrinter alloc] init];
    case PrinterTypeStar:
        return [[StarPrinter alloc] init];
    default:
        return [[ConsolePrinter alloc] init];
    }
}
```

3.3 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.

3.4 Decorator

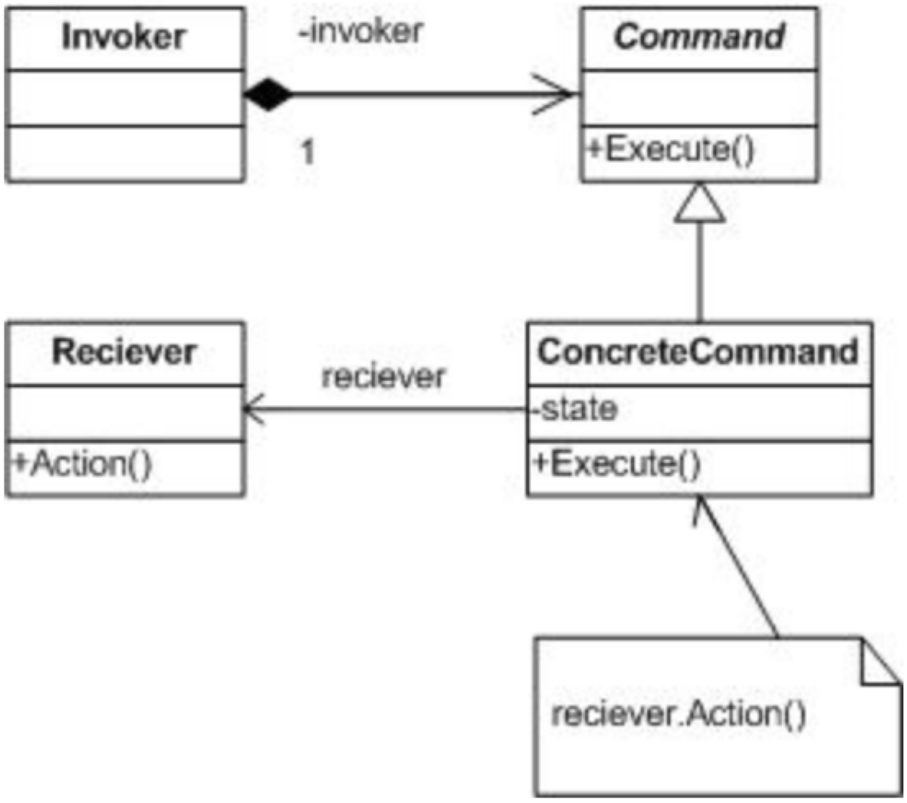
Decorator pattern là pattern cho phép thêm các hành vi, chức năng vào một đối tượng mà không làm thay đổi code của đối tượng đó. Đây là phương thức thay thế cho việc tạo ra class con.

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

3.5 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:



Sơ đồ UML của Command pattern

3.6 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).