Table of Contents

Learn Design Patterns in Swift

Hi, there. This is Peng, iOS developer, blogger and open source contributor. This is a collection for my articles about design patterns in Swift on medium.com. Follow me on medium.com and you will get my latest articles.

I'm also active on Twitter and GitHub.

If you're developer from China, please follow my wechat public accounts on Swift and mobile app development.

Thanks

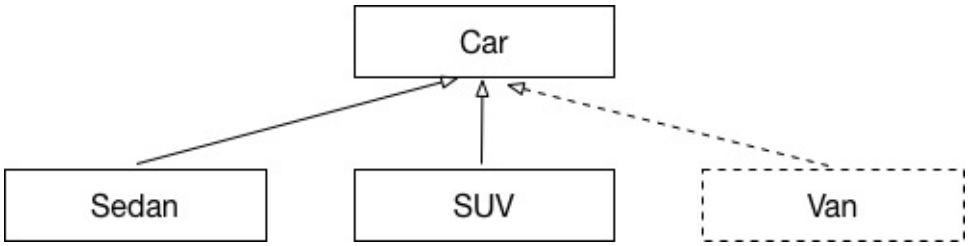
Peng

Version	Description	Date
1.0	First version.	06/18/2018

Simple Factory

If we want to learn a programming language, we need to live with it. It means to use Swift as much as possible. Today, we will start talking about design patterns in Swift. In addition to simple explanation, we also try to give figures, samples, use cases, etc. Let's get started.

The first pattern is simple factory. Simply speaking, we build a factory to produce different objects which belong to the same type. As in the following figure and code, we have different types of cars, sedan, SUV, and maybe van. They all conform to protocol Car. It means the protocol defines common interface for cars. The specific car type implement their own logic.




```
enum CarType {
    case sedan, SUV
}
```

```
protocol Car {
    func drive()
}
```

```
class Sedan: Car {
    func drive() {
        print("drive a sedan")
    }
}
```

```
let sedan = Sedan()
sedan.drive()
let suv = SUV()
suv.drive()
```

If we want to add van, we create van class.

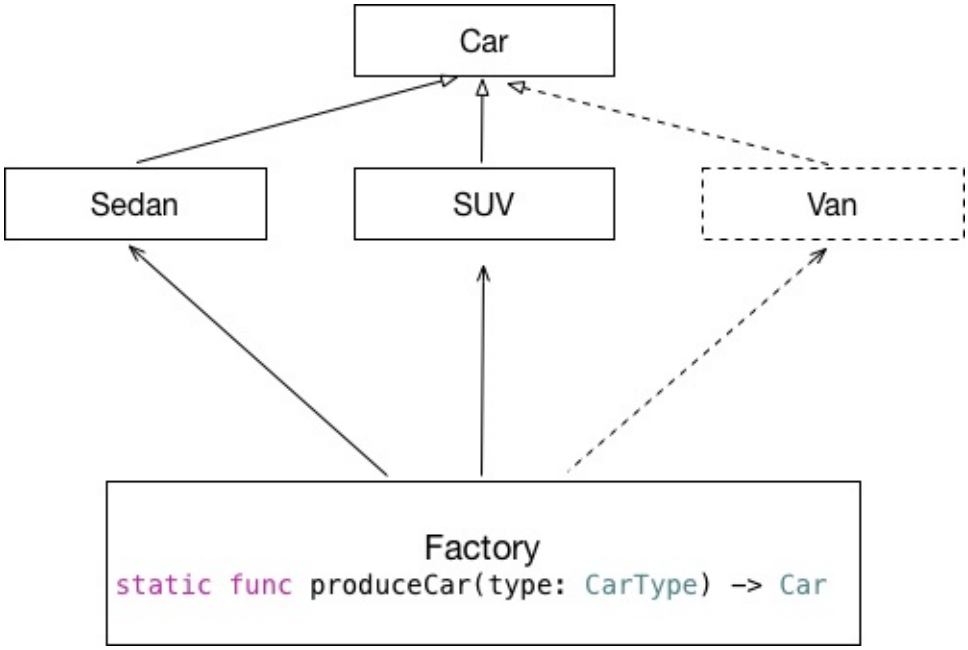
```
enum CarType {
    case sedan, SUV, van
}
```

```
class Van: Car {
    func drive() {
        print("drive a van")
    }
}
```

Then the instance

```
let van = Van()
van.drive()
```

Let’s try simple factory pattern to do the same thing. We need a factory to do the job to produce different types of cars.




```
class Factory {
    static func produceCar(type: CarType) -> Car {
        switch type {
        case .sedan:
            return Sedan()
        case .SUV:
            return SUV()
        }
    }
}
```

```
let sedan = Factory.produceCar(type: .sedan)
sedan.drive()
let suv = Factory.produceCar(type: .SUV)
suv.drive()
```

Obviously, it is the factory which creates different instances of cars. Then what if we want to add van? The van class have been defined. So here we need to change the factory to enable it to produce van.

```
class Factory {
    static func produceCar(type: CarType) -> Car {
        switch type {
        case .sedan:
            return Sedan()
        case .SUV:
            return SUV()
        case .van
            return Van()
        }
    }
}
```

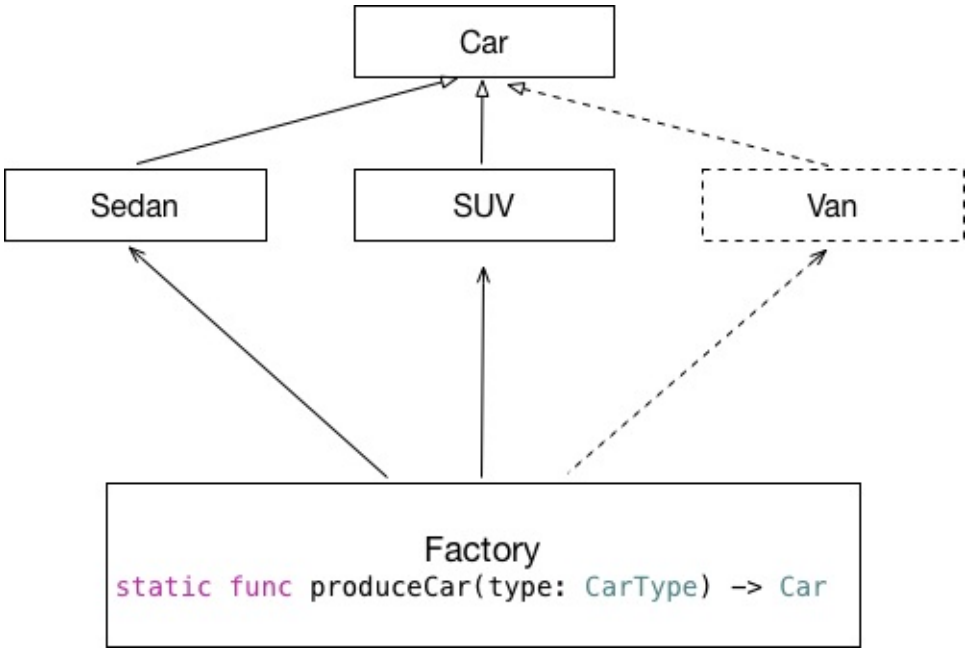
```
let sedan = Factory.produceCar(type: .sedan)
sedan.drive()
let suv = Factory.produceCar(type: .SUV)
suv.drive()
let van = Factory.produceCar(type: .van)
van.drive()
```

That’s all simple factory pattern does. But why do we need this pattern? It separates the creation from usage and move the responsibility to a specific role. Or speak in professional terms, it helps loose coupling. Let’s assume the initialization logic of sedan is changed one day. We don’t need to change every creation in the whole project. To change the logic in factory is all we need to do.

In next blog of this series, we will talk about Factory Method Pattern.

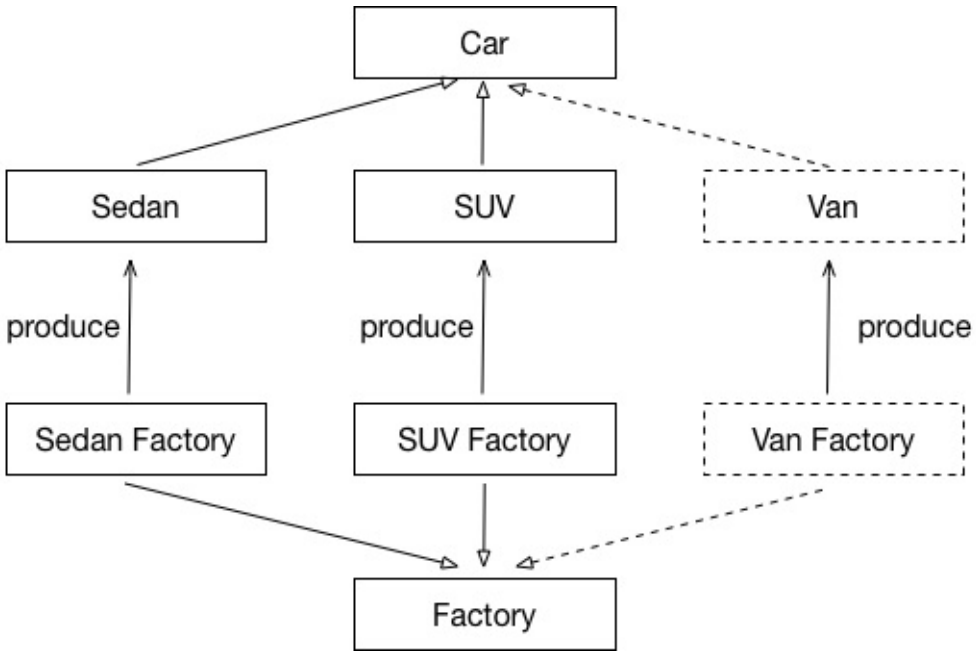
Factory Method

Do you remember the Simple Factory Pattern we talked about in last article?



As the figure tells us, we build a factory to produce different types of cars. But what if the list becomes too long. Now, we have truck, sports car, bus, etc. All the cars will be produced in just one factory. It will make the factory stressful. So we will introduce another pattern Factory Method to help it.

Since the factory feels stressful to produce so many kinds of car, why not build more factories? Good idea, that's what we will do with Factory Method pattern. We're going to build a factory for each type to make them stay focused. These factories have same interface because they all produce cars.



Then let's change our code.

```
/// abstract factory
protocol Factory {
    func produce() -> Car
}
```

```
/// concrete factory
class SedanFactory: Factory {
    func produce() -> Car{
        return Sedan()
    }
}
```

```
/// concrete factory
class SUVFactory: Factory {
    func produce() -> Car {
        return SUV()
    }
}
```

```
let sedanFactory = SedanFactory()
let sedan = sedanFactory.produce()
sedan.drive()
```

```
let suvFactory = SUVFactory()
let suv = suvFactory.produce()
suv.drive()
```

Factory Method pattern makes the codebase more flexible to add or remove new types. To add a new type, we just need a new class for the type and a new factory to produce it like the following code. We don’t need to change original codebase.

```
class Truck: Car {
    func drive() {
        print("drive a truck")
    }
}
```

```
class TruckFactory: Factory {
    func produce() -> Car {
        return Truck()
    }
}
```

```
let truckFactory = TruckFactory()
let truck = truckFactory.produce()
truck.drive()
```

That’s all for Factory Method Pattern.

Singleton

Singleton is very popular in Cocoa. We can find different use cases. The following are two examples.

```
let default = NotificationCenter.default
```

```
let standard = UserDefaults.standard
```

We will not talk about what singleton is and its advantages. You can find so many resources on the Internet. We want to focus on how to write our own.

Do you remember how to do this in the Objective-C world? Here is a template which maybe you've ever seen. The `sharedInstance = [[Car alloc] init];` will only executed once. This is guaranteed by `dispatch_once` .

```
@interface Car : NSObject
@end
```

```
@implementation car

+ (instancetype)sharedInstance {
    static Car *sharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedInstance = [[Car alloc] init];
    });
    return sharedInstance;
}

@end
```

What about Swift?

```
class Car {
    static let sharedInstance = Car()
}
```

or a little complex

```
class Car {
    static let sharedInstance: Car = {
        let instance = Car()
        return instance
    }()
}
```

Really? It’s so simple. Then where is `dispatch_once` ? How do you guarantee the codes only run once? The secret is the ‘static’ keywords. The static property will be lazily initialized once and only once.

Builder

Normally, while building a car, we build every part first and then assemble them up. As customers, we don't need to know how to produce every component. It's the producer's job to produce a functional car according to our requirements.

Do you remember our factory in Factory Method?

```
protocol Factory {
    func produce() -> Car
}
```

It's too simple to describe the real world. Let's enhance it.

```
protocol Factory {
    func produceWheel()
    func produceEngine()
    func produceChassis()
}
```

We still need concrete factory to produce the real car for example a sedan factory or a SUV factory.

```
class SedanFactory: Factory {
    func produceWheel() {
        print("produce wheel for sedan")
    }
    func produceEngine() {
        print("produce engine for sedan")
    }
    func produceChassis() {
        print("produce chassis for sedan")
    }
}
```

```
class SUVFactory: Factory {
    func produceWheel() {
        print("produce wheel for SUV")
    }
    func produceEngine() {
        print("produce engine for SUV")
    }
    func produceChassis() {
        print("produce chassis for SUV")
    }
}
```

Now, we can produce sedan and SUV. But as customers, we don’t give order to the factory directly. Let’s assume there is ‘director’ between customers and automaker. The director gets orders from customers and direct factories to produce cars.

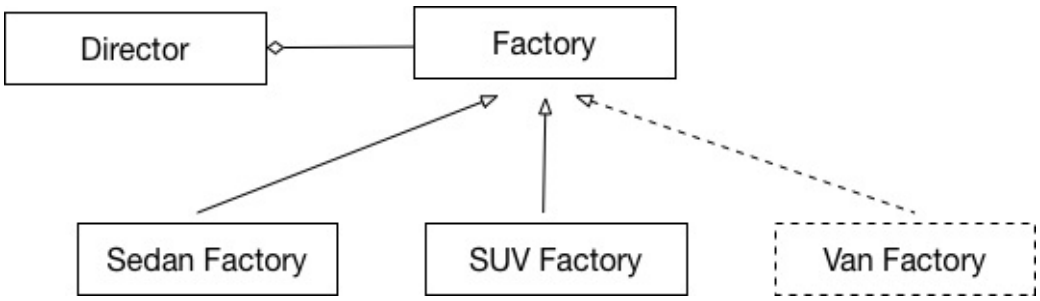
```
class Director {
    var factory: Factory

    init(factory: Factory) {
        self.factory = factory
    }

    func produce() {
        factory.produceWheel()
        factory.produceEngine()
        factory.produceChassis()
    }
}
```

Let’s start to produce cars.

```
let sedanFactory = SedanFactory()
let suvFactory = SUVFactory()
```

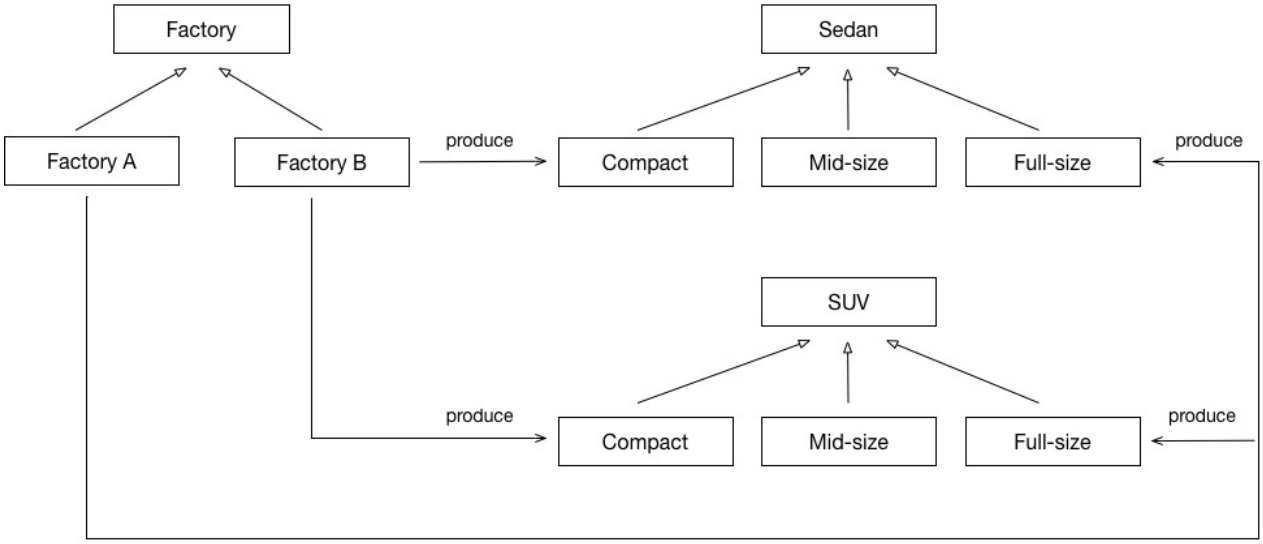
```
let sedanDirector = Director(factory: sedanFactory)
sedanDirector.produce()
let suvDirector = Director(factory: suvFactory)
suvDirector.produce()
```

So the structure becomes to the following.


Director
Factory
Sedan Factory
SUV Factory
Van Factory

Abstract Factory

Today we will talk about abstract factory pattern. It handles a little more complex use case. As we know, the sedan family has different models like compact, mid-size and full-size.SUV has same categories. Let's assume we have two factories. One is focused on compact types and the other is on full size. As the following figure tells us, factory A produces compact sedan and compact SUV. Factory B produces full-size sedan and SUV.



Let's start coding. Here is different sizes of sedans and SUVs. They all conform to respective abstract interface.

```
protocol Sedan {
    func drive()
}
```

```
class CompactSedan: Sedan {
    func drive() {
        print("drive a compact sedan")
    }
}
```

```
class MidSizeSedan: Sedan {
    func drive() {
        print("drive a mid-size sedan")
    }
}
```

```
class FullSizeSedan: Sedan {
    func drive() {
        print("drive a full-size sedan")
    }
}
```

```
protocol SUV {
    func drive()
}
```

```
class CompactSUV: SUV {
    func drive() {
        print("drive a compact SUV")
    }
}
```

```
class MidSizeSUV: SUV {
    func drive() {
        print("drive a mid-size SUV")
    }
}
```

```
class FullSizeSUV: SUV {
    func drive() {
        print("drive a full-size SUV")
    }
}
```

Then the following is our factories. They all can produce sedan and SUV. A is for compact and B is for full-size.

```
protocol Factory {
    func produceSedan() -> Sedan
    func produceSUV() -> SUV
}
```

```
class FactoryA: Factory {
    func produceSedan() -> Sedan{
        return CompactSedan()
    }

    func produceSUV() -> SUV {
        return CompactSUV()
    }
}
```

```
class FactoryB: Factory {
    func produceSedan() -> Sedan {
        return FullSizeSedan()
    }

    func produceSUV() -> SUV {
        return FullSizeSUV()
    }
}
```

Ok, let's produce sedan and SUV now.

```
let factoryA = FactoryA()
let compactSedan = factoryA.produceSedan()
let compactSUV = factoryA.produceSUV()
compactSedan.drive()
compactSUV.drive()
```

```
let factoryB = FactoryB()
let fullsizeSedan = factoryB.produceSedan()
let fullsizeSUV = factoryB.produceSUV()
fullsizeSedan.drive()
fullsizeSUV.drive()
```

Another use case is to make different themes for our UI. There are many elements involved in a theme like label, button, background, etc. But in specific theme, each has a style. Please try abstract factory to build this modal.

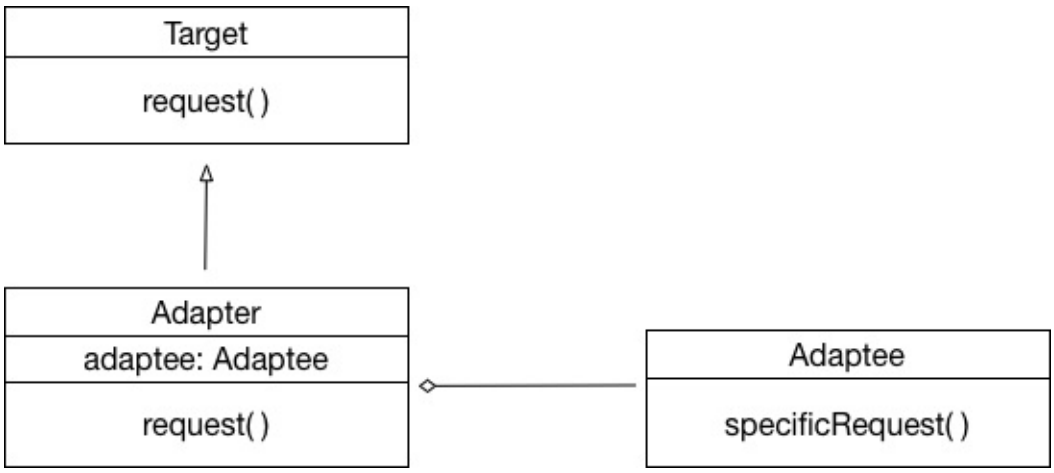
Adapter

We have finished creational patterns and will introduce structural patterns from this article. Literally, structural patterns are about structure. It means how to organize classes or instances to form larger structures. The first we will talk about is adapter pattern.

The latest adapter example in real world is lightning to 3.5 mm headphone Jack adapter. If we want to reuse old headphone with our new iPhone 7 which uses lightning connector, we need this adapter to connect them.

Obviously, the headphone jack adapter is adapter. The old 3.5 mm headphone is adaptee. In the programming world, the adaptee is old class we want to reuse. But its interface is not compatible with new interface. So we need an adapter to help them.

We can implement adapter pattern in two ways. The first one is object adapter which uses composition. There is an adaptee instance in adapter to do the job like the following figure and code tell us.



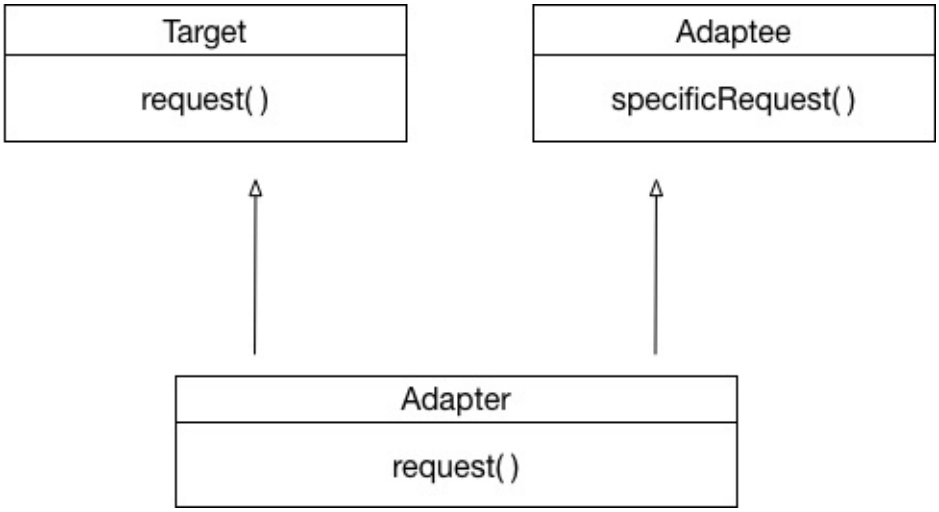

```
protocol Target {
    func request()
}
```

```
class Adapter: Target {
    var adaptee: Adaptee

    init(adaptee: Adaptee) {
        self.adaptee = adaptee
    }

    func request() {
        adaptee.specificRequest()
    }
}
```

```
class Adaptee {
    func specificRequest() {
        print("Specific request")
    }
}
```

```
// usage
let adaptee = Adaptee()
let tar = Adapter(adaptee: adaptee)
tar.request()
```

The other one is class adapter which uses multiple inheritance to connect target and adaptee.




```
protocol Target {
    func request()
}
```

```
class Adaptee {
    func specificRequest() {
        print("Specific request")
    }
}
```

```
class Adapter: Adaptee, Target {s
    func request() {
        specificRequest()
    }
}
```

```
// usage
let tar = Adapter()
tar.request()
```

This is all about the theory. In next articles, we will bring more examples in real programming world.

Bridge

Do you remember our system structure for car? We have a protocol and different implementations like the code below.

```
protocol Car {
    func drive()
}
```

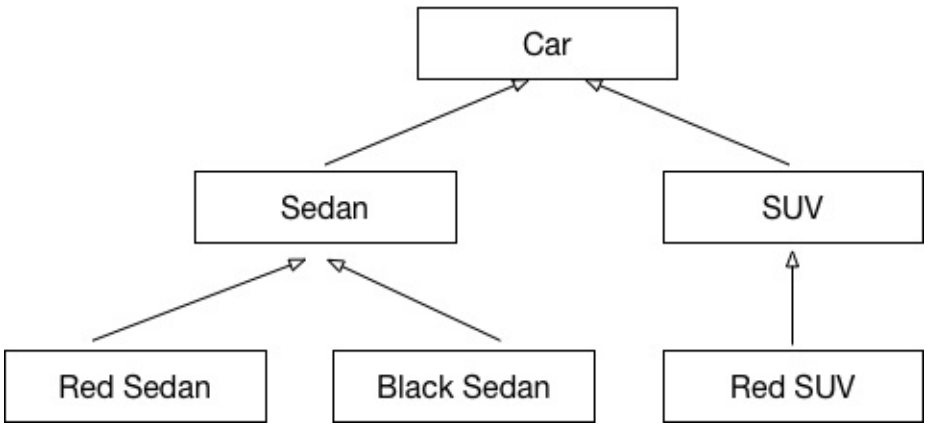
```
class Sedan: Car {
    func drive() {
        print("drive a sedan")
    }
}
```

```
class SUV: Car {
    func drive() {
        print("drive a SUV")
    }
}
```

Then what if we want to describe colored cars? This requirement add another variable in current structure. Normally, we need to add a inheritance level.

```
class RedSedan: Sedan {
    func drive() {
        print("drive a red sedan")
    }
}
```

The structure is in the below figure.



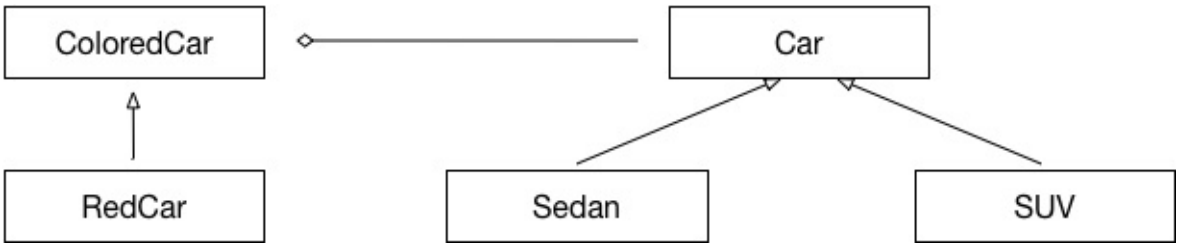
Its disadvantage is too many inheritance levels. If we want to add new colors or new car types, the whole structure will becomes more complicated.

```
protocol ColoredCar {
    var car: Car { get set }
    func drive()
}
```

```
class RedCar: ColoredCar {
    var car: Car
    init(car: Car) {
        self.car = car
    }

     func drive() {
        car.drive()
        print("It's red.")
    }
}
```

```
//usage
let sedan = Sedan()
let redSedan = RedCar(car: sedan)
redSedan.drive()
```

New structure is in the following figure.



Then there is no so many inheritances. The car and color will be scaled respectively.

Decorator

Decorator is a structural pattern to add new functions to class or instance at runtime. Compared to inheritance, it has more advantages in flexibility and scalability. We will still use our car model in this article.

```
enum CarType: String {
    case Sedan = "sedan"
    case SUV = "SUV"
}
```

```
protocol Car {
    var type: CarType{ get }
    func drive()
}
```

```
class Sedan: Car {
    var type: CarType = .Sedan
    func drive() {
        print("drive a " + type.rawValue)
    }
}
```

Now we want to make an autonomous sedan. Normally, we will create a new class whose super class is Sedan and add new feature.

```
class AutonomousSedan: Sedan {
    override func drive() {
        print("automatically drive a " + type.rawValue)
    }
}
```

It’s ok because we get what we want finally. But what if we want to add more features? What if we want to make other car autonomous? The inheritance will make the whole structure more and more complicated. Let’s see if decorator pattern can help us. We will create a new role decorator which inherits car and has an instance of car, and its subclass for specific feature.

```
class Decorator: Car {
    var car: Car
    var type: CarType {
        return car.type
    }

    init(car: Car) {
        self.car = car
    }

    func drive() {
        car.drive()
    }
}
```

```
class AutonomousCar: Decorator {
    override func drive() {
        print("automatically drive a " + type.rawValue)
    }
}
```

Let’s see how to use it.

```
let sedan = Sedan()
let autonomousSedan = AutonomousCar(car: sedan)
autonomousSedan.drive()
```

We add autonomous function to sedan without inheritance. If we want to add other new feature, create another subclass of decorator. And if we want to add autonomous function to other car, make an instance and pass it to AutonomousCar’s initializer. The whole structure is clear and scalable.

Please try to write the codes for the logic we described. If possible, please solve same questions with inheritance. You will get direct experience by comparison.

Facade

Literally, facade means

> the face of a building, especially the principal front that looks onto a street or open space.
>
> from Facade — Google Search

Similarly, as design pattern facade defines an simpler interface to an complex subsystem. For example, in our car factory, we have different departments to produce different components like engine, body, and accessories. As client, we don't care how every department does produce its own job. We just create a factory instance and get it to work.

```
class Engine {
    func produceEngine() {
        print("prodce engine")
    }
}
```

```
class Body {
    func produceBody() {
        print("prodce body")
    }
}
```

```
class Accessories {
    func produceAccessories() {
        print("prodce accessories")
    }
}
```

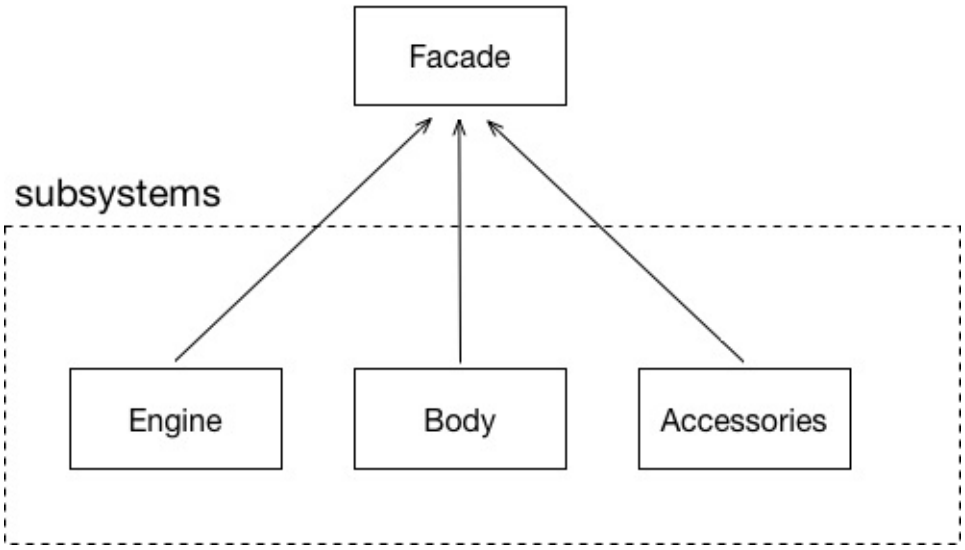
So we build a facade to provide a simple interface.

```
class FactoryFacade {
    let engine = Engine()
    let body = Body()
    let accessories = Accessories()

    func produceCar() {
        engine.produceEngine()
        body.produceBody()
        accessories.produceAccessories()
    }
}
```

Then use the factory directly.

```
let factoryFacade = FactoryFacade()
factoryFacade.produceCar()
```

The structure figure is clear.



Proxy

In this article, we will talk about proxy pattern. In this pattern, proxy is an object to help us to access another object. It simply delegates real job to that object or change its behavior.

Proxy pattern is popularly used in Cocoa which even has a specific NSProxy class in it. Another example is UIApperance protocol and other relevant types.

We will continue using our car system.

```
protocol Car {
    func drive()
}
```

```
class Sedan: Car {
    func drive() {
        print("drive a sedan")
    }
}
```

Autonomous car is so hot topic now. So let's build our own. Actually, it's not built from scratch but enhance our current car with self-driving system. So it has a internal car instance and delegate the driving to the car. But as an autonomous car, it controls the car automatically. Delegation and change are what a proxy does in this pattern.

```
class AutonomousCar: Car {
    var car: Car
    init(car: Car) {
        self.car = car
    }

    func drive() {
        car.drive()
        print("by self-driving system")
    }
}
```

Let's see how to drive it.

```
//usage
let sedan = Sedan()
let autonomousCar = AutonomousCar(car: sedan)
autonomousCar.drive()
```

Flyweight

Flyweight is about sharing. It holds a pool to store objects. The client will reuse existing objects in the pool. In previous articles, we have a general interface for car and a specific sedan class as following.

```
protocol Car {
    var color: UIColor { get }
    func drive()
}
```

```
class Sedan: Car {
    var color: UIColor
    init(color: UIColor) {
        self.color = color
    }

    func drive() {
        print("drive a sedan")
    }
}
```

we also have built our factory which produces sedan. Generally, we have sedan in different colors in stock. When there are orders, we look up in our stock first and reuse it.

```
class Factory {
    var cars: [UIColor: Car] = [UIColor: Car]()

    func getCar(color: UIColor) -> Car {
        if let car = cars[color] {
            return car
        } else {
            let car = Sedan(color: color)
            cars[color] = car
            return car
        }
    }
}
```

Let’s get cars from factory.

```
let factory = Factory()
let redSedan = factory.getCar(color: .red)
redSedan.drive()
```

Util now, we've completed structural patterns.

Command

From today, we will talk about behavioral patterns which focus on the communication between objects. It includes command, mediator, observer, state, strategy, etc.

In this article, we will introduce command pattern. As in real world, a command is a request sent from its invoker (aka caller) to its receiver. By isolating and encapsulating the request, this pattern separates the invoker and the receiver.

Then what advantage does it give us? Let's get started with our stories on the car.

```
protocol Car {
    func move()
}
```

```
class Sedan: Car {
    func move() {
        print("Sedan is moving.")
    }
}
```

```
class SUV: Car {
    func move() {
        print("SUV is moving.")
    }
}
```

Now, you, as an iOS app developer, have a sedan and drive it daily.

```
class Developer {
    let sedan = Sedan()
    func driveCar() {
        sedan.move()
    }
}
```

One day, you make so much money from App Store that you buy another SUV. If you want to drive it, you need to add an instance in the class.

```
class Developer {
    let sedan = Sedan()
    let suv = SUV()

    func driveSedan() {
        sedan.move()
    }

    func driveSUV() {
        suv.move()
    }
}
```

But what if your app becomes popular to make you a millionaire one day? You have so many nice cars that your class becomes bloated. It's command pattern's showtime.

```
protocol Command {
    func execute()
}
```

```
class MovementCommand: Command {
    var car: Car

    init(car: Car) {
        self.car = car
    }

    func execute() {
        car.move()
    }
}
```

In the command, we put in the car which is the receiver and the request to get the car to move. Then all you need to do is to replace all the cars with a command instance.

```
class Developer {
    var command: Command

    init(command: Command) {
        self.command = command
    }

    func drive() {
        command.execute()
    }
}
```

Let’s see how to use it.

```
let sedan = Sedan()
let sedanMovementCommand = MovementCommand(car: sedan)

let suv = SUV()
let suvMovementCommand = MovementCommand(car: suv)

let developer = Developer(command: sedanMovementCommand)
developer.drive()
```

The command pattern is so popular in iOS app development that there is a specific class NSInvocation to help create command. You can refer to its document for detailed explanation. Even it is not available in Swift, we can take it as example of command pattern.

Mediator

In this article we will talk about mediator pattern. Let’s start with a scenario in real world instead of explaining abstract definition. In a team, there are PM, developer and QE. When the developer completes coding for a new feature, the codes are committed to the repository. Other shareholders like QE and PM need to be notified.

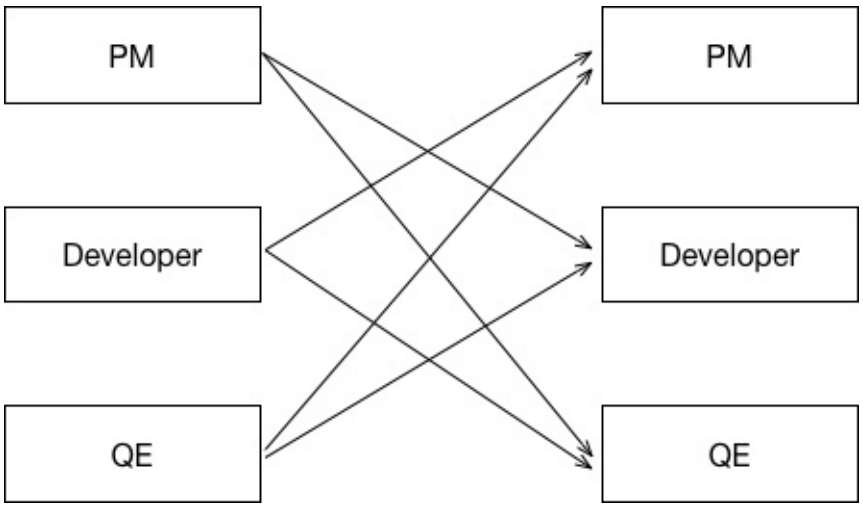
```
protocol Collogue {
    var id: String { get }
    func send(message: String)
    func receive(message: String)
}
```

```
class Developer: Collogue {
    var id: String
    var qe: QE
    var pm: PM

    init(qe: QE, pm: PM) {
        self.id = "Developer"
        self.qe = qe
        self.pm = pm
    }

    func send(message: String) {
        qe.receive(message: message)
        pm.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}
```

```
class QE: Collogue {
    var id: String
    var developer: Developer
    var pm: PM

    init(developer: Developer, pm: PM) {
        self.id = "QE"
        self.developer = developer
        self.pm = pm
    }

    func send(message: String) {
        developer.receive(message: message)
        pm.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}
```

```
class PM: Collogue {
    var id: String
    var developer: Developer
    var qe: QE

    init(developer: Developer, qe: QE) {
        self.id = "PM"
        self.developer = developer
        self.qe = qe
    }

    func send(message: String) {
        developer.receive(message: message)
        qe.receive(message: message)
    }

    func receive(message: String) {
        print(message)
    }
}
```

Every role needs to hold instances of other roles. The connection is so tight that any changes are not easy to make.



Now, we need mediator to help us simplify the system. The mediator is defined to help objects involved to communicate with each other. It means every object interacts with mediator instead of with all others. Then the object don't need to hold reference to others but the mediator. This will decouple the system.

Let's start writing the code.

```
protocol Mediator {
    func send(message: String, sender: Colleague)
}
```

```
class TeamMediator: Mediator {
    var colleagues: [Colleague] = []

    func register(colleague: Colleague) {
        colleagues.append(colleague)
    }

     func send(message: String, sender: Colleague) {
        for colleague in colleagues {
            if colleague.id != sender.id {
                colleague.receive(message: message)
            }
        }
    }
}
```

With mediator, the colleagues become as following.

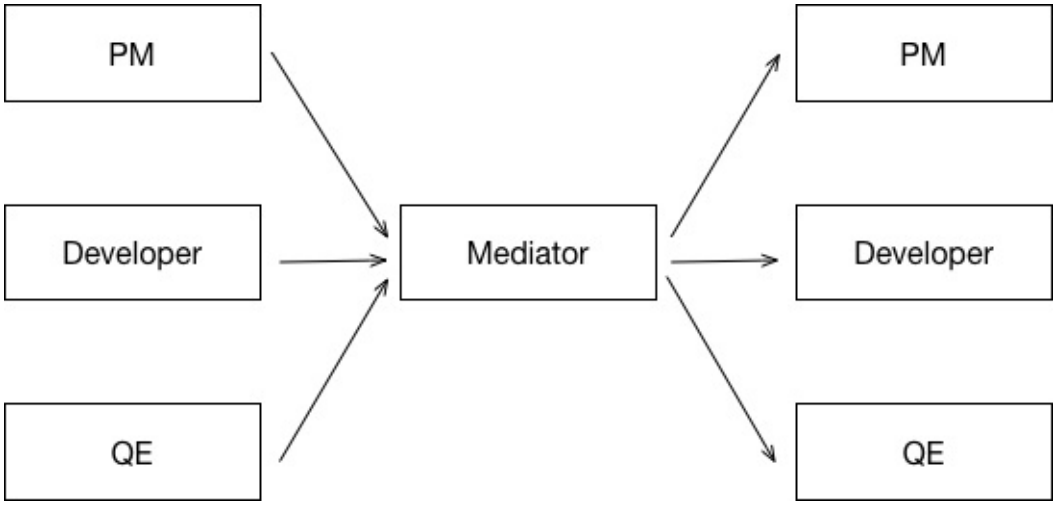
```
protocol Colleague {
    var id: String { get }
    var mediator: Mediator { get }
    func send(message: String)
    func receive(message: String)
}
```

```
class Developer: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "Developer"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("Developer received: " + message)
    }
}
```

```
class QE: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "QE"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("QE received: " + message)
    }
}
```

```
class PM: Colleague {
    var id: String
    var mediator: Mediator

    init(mediator: Mediator) {
        self.id = "PM"
        self.mediator = mediator
    }

    func send(message: String) {
        mediator.send(message: message, sender: self)
    }

    func receive(message: String) {
        print("PM received: " + message)
    }
}
```

So the structure becomes to the following.



Let’s try it.

```
//usage
let mediator = TeamMediator()
let qe = QE(mediator: mediator)
let developer = Developer(mediator: mediator)
let pm = PM(mediator: mediator)

mediator.register(colleague: developer)
mediator.register(colleague: qe)
mediator.register(colleague: pm)

mediator.send(message: "Hello world!", sender: developer)
```

Another example is the popular NotificationCenter (aka NSNotificationCenter). You can find many relevant codes on the Internet.