**Design Pattern phổ biến trong Swift**

**Nhóm Behavior :**

**Chain of Responsibility**

**Ví dụ :**

final class MoneyPile {

let value: Int

var quantity: Int

var nextPile: MoneyPile?

init(value: Int, quantity: Int, nextPile: MoneyPile?) {

self.value = value

self.quantity = quantity

self.nextPile = nextPile

}

func canWithdraw(amount: Int) -> Bool {

var amount = amount

func canTakeSomeBill(want: Int) -> Bool {

return (want / self.value) > 0

}

var quantity = self.quantity

while canTakeSomeBill(want: amount) {

if quantity == 0 {

break

}

amount -= self.value

quantity -= 1

}

guard amount > 0 else {

return true

}

if let next = self.nextPile {

return next.canWithdraw(amount: amount)

}

return false

}

}

final class ATM {

private var hundred: MoneyPile

private var fifty: MoneyPile

private var twenty: MoneyPile

private var ten: MoneyPile

private var startPile: MoneyPile {

return self.hundred

}

init(hundred: MoneyPile,

fifty: MoneyPile,

twenty: MoneyPile,

ten: MoneyPile) {

self.hundred = hundred

self.fifty = fifty

self.twenty = twenty

self.ten = ten

}

func canWithdraw(amount: Int) -> String {

return "Can withdraw: \(self.startPile.canWithdraw(amount: amount))"

}

}

Sử dụng :

let ten = MoneyPile(value: 10, quantity: 6, nextPile: nil)

let twenty = MoneyPile(value: 20, quantity: 2, nextPile: ten)

let fifty = MoneyPile(value: 50, quantity: 2, nextPile: twenty)

let hundred = MoneyPile(value: 100, quantity: 1, nextPile: fifty)

// Build ATM.

var atm = ATM(hundred: hundred, fifty: fifty, twenty: twenty, ten: ten)

atm.canWithdraw(amount: 310) // Cannot because ATM = 300

atm.canWithdraw(amount: 100) // Can withdraw - 1x100

atm.canWithdraw(amount: 165) // Cannot withdraw because ATM doesn't has bill with value of 5

atm.canWithdraw(amount: 30) // Can withdraw - 1x20, 2x10

## **Command**

Ví dụ

protocol DoorCommand {

func execute() -> String

}

class OpenCommand : DoorCommand {

let doors:String

required init(doors: String) {

self.doors = doors

}

func execute() -> String {

return "Opened \(doors)"

}

}

class CloseCommand : DoorCommand {

let doors:String

required init(doors: String) {

self.doors = doors

}

func execute() -> String {

return "Closed \(doors)"

}

}

class HAL9000DoorsOperations {

let openCommand: DoorCommand

let closeCommand: DoorCommand

init(doors: String) {

self.openCommand = OpenCommand(doors:doors)

self.closeCommand = CloseCommand(doors:doors)

}

func close() -> String {

return closeCommand.execute()

}

func open() -> String {

return openCommand.execute()

}

}

Sử dụng :

let podBayDoors = "Pod Bay Doors"

let doorModule = HAL9000DoorsOperations(doors:podBayDoors)

doorModule.open()

doorModule.close()

## **Iterator**

struct Novella {

let name: String

}

struct Novellas {

let novellas: [Novella]

}

struct NovellasIterator: IteratorProtocol {

private var current = 0

private let novellas: [Novella]

init(novellas: [Novella]) {

self.novellas = novellas

}

mutating func next() -> Novella? {

defer { current += 1 }

return novellas.count > current ? novellas[current] : nil

}

}

extension Novellas: Sequence {

func makeIterator() -> NovellasIterator {

return NovellasIterator(novellas: novellas)

}

}

Sử dụng :

let greatNovellas = Novellas(novellas: [Novella(name: "The Mist")] )

for novella in greatNovellas {

print("I've read: \(novella)")

}

## **Mediator**

struct Programmer {

let name: String

init(name: String) {

self.name = name

}

func receive(message: String) {

print("\(name) received: \(message)")

}

}

protocol MessageSending {

func send(message: String)

}

final class MessageMediator: MessageSending {

private var recipients: [Programmer] = []

func add(recipient: Programmer) {

recipients.append(recipient)

}

func send(message: String) {

for recipient in recipients {

recipient.receive(message: message)

}

}

}

Sử dụng :

func spamMonster(message: String, worker: MessageSending) {

worker.send(message: message)

}

let messagesMediator = MessageMediator()

let user0 = Programmer(name: "Linus Torvalds")

let user1 = Programmer(name: "Avadis 'Avie' Tevanian")

messagesMediator.add(recipient: user0)

messagesMediator.add(recipient: user1)

spamMonster(message: "I'd Like to Add you to My Professional Network", worker: messagesMediator)

**Observer**

Ví dụ :

protocol PropertyObserver : class {

func willChange(propertyName: String, newPropertyValue: Any?)

func didChange(propertyName: String, oldPropertyValue: Any?)

}

final class TestChambers {

weak var observer:PropertyObserver?

private let testChamberNumberName = "testChamberNumber"

var testChamberNumber: Int = 0 {

willSet(newValue) {

observer?.willChange(propertyName: testChamberNumberName, newPropertyValue: newValue)

}

didSet {

observer?.didChange(propertyName: testChamberNumberName, oldPropertyValue: oldValue)

}

}

}

final class Observer : PropertyObserver {

func willChange(propertyName: String, newPropertyValue: Any?) {

if newPropertyValue as? Int == 1 {

print("Okay. Look. We both said a lot of things that you're going to regret.")

}

}

func didChange(propertyName: String, oldPropertyValue: Any?) {

if oldPropertyValue as? Int == 0 {

print("Sorry about the mess. I've really let the place go since you killed me.")

}

}

}

Sử dụng :

var observerInstance = Observer()

var testChambers = TestChambers()

testChambers.observer = observerInstance

testChambers.testChamberNumber += 1

## **Strategy**

Ví dụ :

protocol PrintStrategy {

func print(\_ string: String) -> String

}

final class Printer {

private let strategy: PrintStrategy

func print(\_ string: String) -> String {

return self.strategy.print(string)

}

init(strategy: PrintStrategy) {

self.strategy = strategy

}

}

final class UpperCaseStrategy: PrintStrategy {

func print(\_ string: String) -> String {

return string.uppercased()

}

}

final class LowerCaseStrategy: PrintStrategy {

func print(\_ string:String) -> String {

return string.lowercased()

}

}

Sử dụng :

var lower = Printer(strategy: LowerCaseStrategy())

lower.print("O tempora, o mores!")

var upper = Printer(strategy: UpperCaseStrategy())

upper.print("O tempora, o mores!")

# Nhóm Creational

## **Abstract Factory**

Ví dụ :

Protocols

protocol Decimal {

func stringValue() -> String

// factory

static func make(string : String) -> Decimal

}

typealias NumberFactory = (String) -> Decimal

// Number implementations with factory methods

struct NextStepNumber: Decimal {

private var nextStepNumber: NSNumber

func stringValue() -> String { return nextStepNumber.stringValue }

// factory

static func make(string: String) -> Decimal {

return NextStepNumber(nextStepNumber: NSNumber(value: (string as NSString).longLongValue))

}

}

struct SwiftNumber : Decimal {

private var swiftInt: Int

func stringValue() -> String { return "\(swiftInt)" }

// factory

static func make(string: String) -> Decimal {

return SwiftNumber(swiftInt:(string as NSString).integerValue)

}

}

Abstract factory

enum NumberType {

case nextStep, swift

}

enum NumberHelper {

static func factory(for type: NumberType) -> NumberFactory {

switch type {

case .nextStep:

return NextStepNumber.make

case .swift:

return SwiftNumber.make

}

}

}

Sử dụng :

let factoryOne = NumberHelper.factory(for: .nextStep)

let numberOne = factoryOne("1")

numberOne.stringValue()

let factoryTwo = NumberHelper.factory(for: .swift)

let numberTwo = factoryTwo("2")

numberTwo.stringValue()

## **Factory Method**

Ví dụ :

protocol Currency {

func symbol() -> String

func code() -> String

}

class Euro : Currency {

func symbol() -> String {

return "€"

}

func code() -> String {

return "EUR"

}

}

class UnitedStatesDolar : Currency {

func symbol() -> String {

return "$"

}

func code() -> String {

return "USD"

}

}

enum Country {

case unitedStates, spain, uk, greece

}

enum CurrencyFactory {

static func currency(for country:Country) -> Currency? {

switch country {

case .spain, .greece :

return Euro()

case .unitedStates :

return UnitedStatesDolar()

default:

return nil

}

}

}

Sử dụng :

let noCurrencyCode = "No Currency Code Available"

CurrencyFactory.currency(for: .greece)?.code() ?? noCurrencyCode

CurrencyFactory.currency(for: .spain)?.code() ?? noCurrencyCode

CurrencyFactory.currency(for: .unitedStates)?.code() ?? noCurrencyCode

CurrencyFactory.currency(for: .uk)?.code() ?? noCurrencyCode

## **Prototype**

Ví dụ :

class ChungasRevengeDisplay {

var name: String?

let font: String

init(font: String) {

self.font = font

}

func clone() -> ChungasRevengeDisplay {

return ChungasRevengeDisplay(font:self.font)

}

}

Sử dụng :

let Prototype = ChungasRevengeDisplay(font:"GotanProject")

let Philippe = Prototype.clone()

Philippe.name = "Philippe"

let Christoph = Prototype.clone()

Christoph.name = "Christoph"

let Eduardo = Prototype.clone()

Eduardo.name = "Eduardo"

## **Singleton**

Sử dụng :

class DeathStarSuperlaser {

static let sharedInstance = DeathStarSuperlaser()

private init() {

// Private initialization to ensure just one instance is created.

}

}

Ví dụ :

let laser = DeathStarSuperlaser.sharedInstance

[Behavioral](Behavioral) |

[Creational](Creational) |

Structural

Nhóm Stuctural

## **Adapter**

Ví dụ :

protocol OlderDeathStarSuperLaserAiming {

var angleV: NSNumber {get}

var angleH: NSNumber {get}

}

Ap1 :

struct DeathStarSuperlaserTarget {

let angleHorizontal: Double

let angleVertical: Double

init(angleHorizontal:Double, angleVertical:Double) {

self.angleHorizontal = angleHorizontal

self.angleVertical = angleVertical

}

}

Adapter :

struct OldDeathStarSuperlaserTarget : OlderDeathStarSuperLaserAiming {

private let target : DeathStarSuperlaserTarget

var angleV:NSNumber {

return NSNumber(value: target.angleVertical)

}

var angleH:NSNumber {

return NSNumber(value: target.angleHorizontal)

}

init(\_ target:DeathStarSuperlaserTarget) {

self.target = target

}

}

## **Composite**

Ví dụ :

Component :

protocol Shape {

func draw(fillColor: String)

}

Con ( lá ) :

final class Square : Shape {

func draw(fillColor: String) {

print("Drawing a Square with color \(fillColor)")

}

}

final class Circle : Shape {

func draw(fillColor: String) {

print("Drawing a circle with color \(fillColor)")

}

}

Composite

final class Whiteboard : Shape {

lazy var shapes = [Shape]()

init(\_ shapes:Shape...) {

self.shapes = shapes

}

func draw(fillColor: String) {

for shape in self.shapes {

shape.draw(fillColor: fillColor)

}

}

}

Sử dụng :

var whiteboard = Whiteboard(Circle(), Square())

whiteboard.draw("Red")

Facade

Ví dụ :

enum Eternal {

static func set(\_ object: Any, forKey defaultName: String) {

let defaults: UserDefaults = UserDefaults.standard

defaults.set(object, forKey:defaultName)

defaults.synchronize()

}

static func object(forKey key: String) -> AnyObject! {

let defaults: UserDefaults = UserDefaults.standard

return defaults.object(forKey: key) as AnyObject!

}

}

Sử dụng :

Eternal.set("Disconnect me. I’d rather be nothing", forKey:"Bishop")

Eternal.object(forKey: "Bishop")

Proxy :

## **Protection Proxy**

Ví dụ :

protocol DoorOperator {

func open(doors: String) -> String

}

class HAL9000 : DoorOperator {

func open(doors: String) -> String {

return ("HAL9000: Affirmative, Dave. I read you. Opened \(doors).")

}

}

class CurrentComputer : DoorOperator {

private var computer: HAL9000!

func authenticate(password: String) -> Bool {

guard password == "pass" else {

return false;

}

computer = HAL9000()

return true

}

func open(doors: String) -> String {

guard computer != nil else {

return "Access Denied. I'm afraid I can't do that."

}

return computer.open(doors: doors)

}

}

Sử dụng :

let computer = CurrentComputer()

let podBay = "Pod Bay Doors"

computer.open(doors: podBay)

computer.authenticate(password: "pass")

computer.open(doors: podBay)

## **Virtual Proxy**

Ví dụ :

protocol HEVSuitMedicalAid {

func administerMorphine() -> String

}

class HEVSuit : HEVSuitMedicalAid {

func administerMorphine() -> String {

return "Morphine aministered."

}

}

class HEVSuitHumanInterface : HEVSuitMedicalAid {

lazy private var physicalSuit: HEVSuit = HEVSuit()

func administerMorphine() -> String {

return physicalSuit.administerMorphine()

}

}

Sử dụng :

let humanInterface = HEVSuitHumanInterface()

humanInterface.administerMorphine()