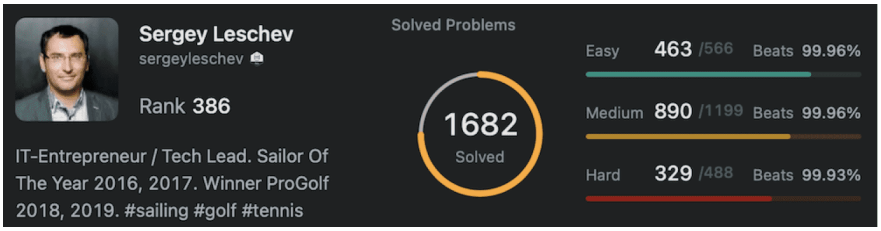
S.Leschev Design Patterns (Swift 5+)

🏆 Awards

Ranking #Dev: Global TOP 300 (Certificate)



Golden Award Muad'Dib's Challenge

CODILITY AWARD

Codility confirms that a person who identified to us on-line as:

Sergey Leschev

was awarded
Codility Golden Award for the Muad'Dib's Challenge

This is the **31st** Codility Golden Award granted this month.

Review detailed assessment

AWARD TYPE:	Codility Golden Award (Muad'Dib's)
EXPIRATION DATE:	2025-04-02 17:00:00 GMT
CHALLENGE POSTED:	2022-04-02 17:00:00 GMT
CHALLENGE SOLVED:	2022-04-03 07:31:31 GMT
TIME SINCE POSTED:	15 hours ago
ASSESSMENT:	correct functionality and scalability

Codility_

Languages: Swift.

Algorithmic skills: Dynamic programming, Greedy algorithms, Binary search, Stack and Queues, Sorting, Time Complexity.

Contest: Algorithms, Data Structures.

Google Engineering Level: L6+

Design large-scale systems / 2022

Table of Contents

Behavioral	Creational	Structural
🐝 Chain Of Responsibility	🏉 Abstract Factory	🔌 Adapter
👯 Command	👷 Builder	🌉 Bridge
🎶 Interpreter	🏭 Factory Method	🌿 Composite
🍫 Iterator	🔂 Monostate	🍧 Decorator
💐 Mediator	🃏 Prototype	🎁 Facade
💾 Memento	💍 Singleton	🪽 Flyweight
👓 Observer		☔ Protection Proxy
🐊 State		🍬 Virtual Proxy
💡 Strategy		
🏃 Visitor		
📝 Template Method		

Behavioral

> In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.
>
> **Source:** wikipedia.org

🐝 Chain Of Responsibility

The chain of responsibility pattern is used to process varied requests, each of which may be dealt with by a different handler.

Example:

```
protocol Withdrawing {
    func withdraw(amount: Int) -> Bool
}

final class MoneyPile: Withdrawing {
    let value: Int
    var quantity: Int
    var next: Withdrawing?

    init(value: Int, quantity: Int, next: Withdrawing?) {
        self.value = value
        self.quantity = quantity
        self.next = next
    }
```

```
    func withdraw(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.next {
            return next.withdraw(amount: amount)
        }

        return false
    }
}

final class ATM: Withdrawing {

    private var hundred: Withdrawing
    private var fifty: Withdrawing
    private var twenty: Withdrawing
    private var ten: Withdrawing
```

```
    private var startPile: Withdrawing {
        return self.hundred
    }

    init(hundred: Withdrawing,
           fifty: Withdrawing,
         twenty: Withdrawing,
            ten: Withdrawing) {

        self.hundred = hundred
        self.fifty = fifty
        self.twenty = twenty
        self.ten = ten
    }

    func withdraw(amount: Int) -> Bool {
        return startPile.withdraw(amount: amount)
    }
}
```

Usage

```
// Create piles of money and link them together 10 < 20 < 50 < 100.**
let ten = MoneyPile(value: 10, quantity: 6, next: nil)
let twenty = MoneyPile(value: 20, quantity: 2, next: ten)
let fifty = MoneyPile(value: 50, quantity: 2, next: twenty)
let hundred = MoneyPile(value: 100, quantity: 1, next: fifty)

// Build ATM.
var atm = ATM(hundred: hundred, fifty: fifty, twenty: twenty, ten: ten
atm.withdraw(amount: 310) // Cannot because ATM has only 300
atm.withdraw(amount: 100) // Can withdraw - 1x100
```

👫 Command

The command pattern is used to express a request, including the call to be made and all of its required parameters, in a command object. The command may then be executed immediately or held for later use.

Example:

```
protocol DoorCommand {
    func execute() -> String
}

final class OpenCommand: DoorCommand {
    let doors:String
    required init(doors: String) {
        self.doors = doors
    }
    func execute() -> String {
        return "Opened \(doors)"
    }
}

final class CloseCommand: DoorCommand {
    let doors:String

    required init(doors: String) {
        self.doors = doors
    }
    func execute() -> String {
        return "Closed \(doors)"
    }
}
```

```
final 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()
    }
}
```

Usage:

```
let podBayDoors = "Pod Bay Doors"
let doorModule = HAL9000DoorsOperations(doors:podBayDoors)

doorModule.open()
doorModule.close()
```

🎶 Interpreter

The interpreter pattern is used to evaluate sentences in a language.

Example

```
protocol IntegerExpression {
    func evaluate(_ context: IntegerContext) -> Int
    func replace(character: Character, integerExpression: IntegerExpre
    func copied() -> IntegerExpression
}

final class IntegerContext {
    private var data: [Character:Int] = [:]

    func lookup(name: Character) -> Int {
        return self.data[name]!
    }

    func assign(expression: IntegerVariableExpression, value: Int) {
        self.data[expression.name] = value
    }
}

final class IntegerVariableExpression: IntegerExpression {
    let name: Character

    init(name: Character) {
        self.name = name
    }

    func evaluate(_ context: IntegerContext) -> Int {
        return context.lookup(name: self.name)
    }
```

```
    func replace(character name: Character, integerExpression: Intege
        if name == self.name {
            return integerExpression.copied()
        } else {
            return IntegerVariableExpression(name: self.name)
        }
    }

    func copied() -> IntegerExpression {
        return IntegerVariableExpression(name: self.name)
    }
}

final class AddExpression: IntegerExpression {
    private var operand1: IntegerExpression
    private var operand2: IntegerExpression

    init(op1: IntegerExpression, op2: IntegerExpression) {
        self.operand1 = op1
        self.operand2 = op2
    }

    func evaluate(_ context: IntegerContext) -> Int {
        return self.operand1.evaluate(context) + self.operand2.evalua
    }

    func replace(character: Character, integerExpression: IntegerExpre
        return AddExpression(op1: operand1.replace(character: characte
                             op2: operand2.replace(character: characte
    }

    func copied() -> IntegerExpression {
        return AddExpression(op1: self.operand1, op2: self.operand2)
    }
}
```

Usage

```
var context = IntegerContext()

var a = IntegerVariableExpression(name: "A")
var b = IntegerVariableExpression(name: "B")
var c = IntegerVariableExpression(name: "C")

var expression = AddExpression(op1: a, op2: AddExpression(op1: b, op2

context.assign(expression: a, value: 2)
context.assign(expression: b, value: 1)
context.assign(expression: c, value: 3)

var result = expression.evaluate(context)
```

🍫 Iterator

The iterator pattern is used to provide a standard interface for traversing a collection of items in an aggregate object without the need to understand its underlying structure.

Example:

```
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)
    }
}
```

Usage

```
let greatNovellas = Novellas(novellas: [Novella(name: "The Mist")] )
for novella in greatNovellas {
    print("I've read: \(novella)")
}
```

💐 Mediator

The mediator pattern is used to reduce coupling between classes that communicate with each other. Instead of classes communicating directly, and thus requiring knowledge of their implementation, the classes send messages via a mediator object.

Example

```
protocol Receiver {
    associatedtype MessageType
    func receive(message: MessageType)
}

protocol Sender {
    associatedtype MessageType
    associatedtype ReceiverType: Receiver

    var recipients: [ReceiverType] { get }

    func send(message: MessageType)
}

struct Programmer: Receiver {
    let name: String

    init(name: String) {
        self.name = name
    }

    func receive(message: String) {
        print("\(name) received: \(message)")
    }
}
```

```
final class MessageMediator: Sender {
    internal var recipients: [Programmer] = []

    func add(recipient: Programmer) {
        recipients.append(recipient)
    }

    func send(message: String) {
        for recipient in recipients {
            recipient.receive(message: message)
        }
    }
}
```

Usage

```
func spamMonster(message: String, worker: MessageMediator) {
    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",
```

💾 Memento

The memento pattern is used to capture the current state of an object and store it in such a manner that it can be restored at a later time without breaking the rules of encapsulation.

Example

```
typealias Memento = [String: String]
```

Originator

```
protocol MementoConvertible {
    var memento: Memento { get }
    init?(memento: Memento)
}

struct GameState: MementoConvertible {

    private enum Keys {
        static let chapter = "com.valve.halflife.chapter"
        static let weapon = "com.valve.halflife.weapon"
    }

    var chapter: String
    var weapon: String

    init(chapter: String, weapon: String) {
        self.chapter = chapter
        self.weapon = weapon
    }
```

```
    init?(memento: Memento) {
        guard let mementoChapter = memento[Keys.chapter],
              let mementoWeapon = memento[Keys.weapon] else {
            return nil
        }

        chapter = mementoChapter
        weapon = mementoWeapon
    }


    var memento: Memento {
        return [ Keys.chapter: chapter, Keys.weapon: weapon ]
    }
}
```

Caretaker

```
enum CheckPoint {

    private static let defaults = UserDefaults.standard

    static func save(_ state: MementoConvertible, saveName: String) {
        defaults.set(state.memento, forKey: saveName)
        defaults.synchronize()
    }

    static func restore(saveName: String) -> Any? {
        return defaults.object(forKey: saveName)
    }
}
```

Usage

```
var gameState = GameState(chapter: "Black Mesa Inbound", weapon: "Crow

gameState.chapter = "Anomalous Materials"
gameState.weapon = "Glock 17"
CheckPoint.save(gameState, saveName: "gameState1")

gameState.chapter = "Unforeseen Consequences"
gameState.weapon = "MP5"
CheckPoint.save(gameState, saveName: "gameState2")

gameState.chapter = "Office Complex"
gameState.weapon = "Crossbow"
CheckPoint.save(gameState, saveName: "gameState3")

if let memento = CheckPoint.restore(saveName: "gameState1") as? Memen
    let finalState = GameState(memento: memento)
    dump(finalState)
}
```

👓 Observer

The observer pattern is used to allow an object to publish changes to its state. Other objects subscribe to be immediately notified of any changes.

Example

```
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,
        }
        didSet {
            observer?.didChange(propertyName: testChamberNumberName, o
        }
    }
}

final class Observer : PropertyObserver {
    func willChange(propertyName: String, newPropertyValue: Any?) {
        if newPropertyValue as? Int == 1 {
            print("Okay. Look.")
        }
    }
```

```
    func didChange(propertyName: String, oldPropertyValue: Any?) {
        if oldPropertyValue as? Int == 0 {
            print("Sorry about the mess.")
        }
    }
}
```

Usage

```
var observerInstance = Observer()
var testChambers = TestChambers()
testChambers.observer = observerInstance
testChambers.testChamberNumber += 1
```

 State

The state pattern is used to alter the behaviour of an object as its internal state changes. The pattern allows the class for an object to apparently change at run-time.

Example

```
final class Context {
    private var state: State = UnauthorizedState()

    var isAuthorized: Bool {
        get { return state.isAuthorized(context: self) }
    }

    var userId: String? {
        get { return state.userId(context: self) }
    }
```

```
    func changeStateToAuthorized(userId: String) {
        state = AuthorizedState(userId: userId)
    }

    func changeStateToUnauthorized() {
        state = UnauthorizedState()
    }
}

protocol State {
    func isAuthorized(context: Context) -> Bool
    func userId(context: Context) -> String?
}

class UnauthorizedState: State {
    func isAuthorized(context: Context) -> Bool { return false }
    func userId(context: Context) -> String? { return nil }
}

class AuthorizedState: State {
    let userId: String
    init(userId: String) { self.userId = userId }
    func isAuthorized(context: Context) -> Bool { return true }
    func userId(context: Context) -> String? { return userId }
}
```

Usage

```
let userContext = Context()
(userContext.isAuthorized, userContext.userId)
userContext.changeStateToAuthorized(userId: "admin")
(userContext.isAuthorized, userContext.userId) // now logged in as "ad
userContext.changeStateToUnauthorized()
(userContext.isAuthorized, userContext.userId)
```

💡 Strategy

The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time.

Example

```
struct TestSubject {
    let pupilDiameter: Double
    let blushResponse: Double
    let isOrganic: Bool
}

protocol RealnessTesting: AnyObject {
    func testRealness(_ testSubject: TestSubject) -> Bool
}

final class VoightKampffTest: RealnessTesting {
    func testRealness(_ testSubject: TestSubject) -> Bool {
        return testSubject.pupilDiameter < 30.0 || testSubject.blushRe
    }
}
final class GeneticTest: RealnessTesting {
    func testRealness(_ testSubject: TestSubject) -> Bool {
        return testSubject.isOrganic
    }
}
final class BladeRunner {
    private let strategy: RealnessTesting
    init(test: RealnessTesting) { self.strategy = test }
    func testIfAndroid(_ testSubject: TestSubject) -> Bool {
        return !strategy.testRealness(testSubject)
    }
}
```

Usage

```
let rachel = TestSubject(pupilDiameter: 30.2,
                         blushResponse: 0.3,
                         isOrganic: false)

// Deckard is using a traditional test
let deckard = BladeRunner(test: VoightKampffTest())
let isRachelAndroid = deckard.testIfAndroid(rachel)

// Gaff is using a very precise method
let gaff = BladeRunner(test: GeneticTest())
let isDeckardAndroid = gaff.testIfAndroid(rachel)
```

 Template Method

The template method pattern defines the steps of an algorithm and allows the redefinition of one or more of these steps. In this way, the template method protects the algorithm, the order of execution and provides abstract methods that can be implemented by concrete types.

Example

```
protocol Garden {
    func prepareSoil()
    func plantSeeds()
    func waterPlants()
    func prepareGarden()
}
```

```
extension Garden {
    func prepareGarden() {
        prepareSoil()
        plantSeeds()
        waterPlants()
    }
}

final class RoseGarden: Garden {

    func prepare() {
        prepareGarden()
    }

    func prepareSoil() {
        print ("prepare soil for rose garden")
    }

    func plantSeeds() {
        print ("plant seeds for rose garden")
    }

    func waterPlants() {
       print ("water the rose garden")
    }
}
```

Usage

```
let roseGarden = RoseGarden()
roseGarden.prepare()
```

🏃 Visitor

The visitor pattern is used to separate a relatively complex set of structured data classes from the functionality that may be performed upon the data that they hold.

Example

```
protocol PlanetVisitor {
    func visit(planet: PlanetAlderaan)
    func visit(planet: PlanetCoruscant)
    func visit(planet: PlanetTatooine)
    func visit(planet: MoonJedha)
}

protocol Planet {
    func accept(visitor: PlanetVisitor)
}

final class MoonJedha: Planet {
    func accept(visitor: PlanetVisitor) { visitor.visit(planet: self)
}

final class PlanetAlderaan: Planet {
    func accept(visitor: PlanetVisitor) { visitor.visit(planet: self)
}

final class PlanetCoruscant: Planet {
    func accept(visitor: PlanetVisitor) { visitor.visit(planet: self)
}

final class PlanetTatooine: Planet {
    func accept(visitor: PlanetVisitor) { visitor.visit(planet: self)
}
```

```
final class NameVisitor: PlanetVisitor {
    var name = ""

    func visit(planet: PlanetAlderaan)  { name = "Alderaan" }
    func visit(planet: PlanetCoruscant) { name = "Coruscant" }
    func visit(planet: PlanetTatooine)  { name = "Tatooine" }
    func visit(planet: MoonJedha)       { name = "Jedha" }
}
```

Usage

```
let planets: [Planet] = [PlanetAlderaan(), PlanetCoruscant(), PlanetTa

let names = planets.map { (planet: Planet) -> String in
    let visitor = NameVisitor()
    planet.accept(visitor: visitor)

    return visitor.name
}

names
```

Creational

> In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.
>
> **Source:** wikipedia.org

🍔 Abstract Factory

The abstract factory pattern is used to provide a client with a set of related or dependant objects. The "family" of objects created by the factory are determined at run-time.

Example

Protocols

```
protocol BurgerDescribing {
    var ingredients: [String] { get }
}

struct CheeseBurger: BurgerDescribing {
    let ingredients: [String]
}

protocol BurgerMaking {
    func make() -> BurgerDescribing
}

// Number implementations with factory methods
final class BigKahunaBurger: BurgerMaking {
    func make() -> BurgerDescribing {
        return CheeseBurger(ingredients: ["Cheese", "Burger", "Lettuce
    }
}

final class JackInTheBox: BurgerMaking {
    func make() -> BurgerDescribing {
        return CheeseBurger(ingredients: ["Cheese", "Burger", "Tomato"
    }
}
```

Abstract factory

```
enum BurgerFactoryType: BurgerMaking {

    case bigKahuna
    case jackInTheBox

    func make() -> BurgerDescribing {
        switch self {
        case .bigKahuna:
            return BigKahunaBurger().make()
        case .jackInTheBox:
            return JackInTheBox().make()
        }
    }
}
```

Usage

```
let bigKahuna = BurgerFactoryType.bigKahuna.make()
let jackInTheBox = BurgerFactoryType.jackInTheBox.make()
```

👷 Builder

The builder pattern is used to create complex objects with constituent parts that must be created in the same order or using a specific algorithm. An external class controls the construction algorithm.

Example

```
final class DeathStarBuilder {
    var x: Double?
    var y: Double?
    var z: Double?

    typealias BuilderClosure = (DeathStarBuilder) -> ()

    init(buildClosure: BuilderClosure) {
        buildClosure(self)
    }
}

struct DeathStar : CustomStringConvertible {
    let x: Double
    let y: Double
    let z: Double

    init?(builder: DeathStarBuilder) {
        if let x = builder.x, let y = builder.y, let z = builder.z {
            self.x = x
            self.y = y
            self.z = z
        } else {
            return nil
        }
    }

    var description:String {
        return "Death Star at (x:\(x) y:\(y) z:\(z))"
    }
}
```

Usage

```
let empire = DeathStarBuilder { builder in
    builder.x = 0.1
    builder.y = 0.2
    builder.z = 0.3
}

let deathStar = DeathStar(builder:empire)
```

🏭 Factory Method

The factory pattern is used to replace class constructors, abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

Example

```
protocol CurrencyDescribing {
    var symbol: String { get }
    var code: String { get }
}

final class Euro: CurrencyDescribing {
    var symbol: String { return "€" }
    var code: String { return "EUR" }
}

final class UnitedStatesDolar: CurrencyDescribing {
    var symbol: String { return "$" }
    var code: String { return "USD" }
}
```

```
enum Country {
    case unitedStates
    case spain
    case uk
    case greece
}

enum CurrencyFactory {
    static func currency(for country: Country) -> CurrencyDescribing?
        switch country {
            case .spain, .greece:
                return Euro()
            case .unitedStates:
                return UnitedStatesDolar()
            default:
                return nil
        }
    }
}
```

Usage

```
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
```

🔂 Monostate

The monostate pattern is another way to achieve singularity. It works through a completely different mechanism, it enforces the behavior of singularity without imposing structural constraints. So in that case, monostate saves the state as

static instead of the entire instance as a singleton. SINGLETON and MONOSTATE - Robert C. Martin

Example:

```
class Settings {

    enum Theme {
        case `default`
        case old
        case new
    }

    private static var theme: Theme?

    var currentTheme: Theme {
        get { Settings.theme ?? .default }
        set(newTheme) { Settings.theme = newTheme }
    }
}
```

Usage:

```
import SwiftUI
// When change the theme
let settings = Settings() // Starts using theme .old
settings.currentTheme = .new // Change theme to .new
// On screen 1
let screenColor: Color = Settings().currentTheme == .old ? .gray : .wh
// On screen 2
let screenTitle: String = Settings().currentTheme == .old ? "Itunes Co
```

Prototype

The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone. This practise is particularly useful when the construction of a new object is inefficient.

Example

```
class MoonWorker {

    let name: String
    var health: Int = 100

    init(name: String) {
        self.name = name
    }

    func clone() -> MoonWorker {
        return MoonWorker(name: name)
    }
}
```

Usage

```
let prototype = MoonWorker(name: "Sam Bell")
var bell1 = prototype.clone()
bell1.health = 12
var bell2 = prototype.clone()
bell2.health = 23
var bell3 = prototype.clone()
bell3.health = 0
```

💍 Singleton

The singleton pattern ensures that only one object of a particular class is ever created. All further references to objects of the singleton class refer to the same underlying instance. There are very few applications, do not overuse this pattern!

Example:

```
final class ElonMusk {

    static let shared = ElonMusk()

    private init() {
        // Private initialization to ensure just one instance is creat
    }
}
```

Usage:

```
let elon = ElonMusk.shared // There is only one Elon Musk folks.
```

Structural

> In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. **Source:** wikipedia.org

🔌 Adapter

The adapter pattern is used to provide a link between two otherwise incompatible types by wrapping the "adaptee" with a class that supports the interface required by the client.

Example

```
protocol NewDeathStarSuperLaserAiming {
    var angleV: Double { get }
    var angleH: Double { get }
}
```

Adaptee

```
struct OldDeathStarSuperlaserTarget {
    let angleHorizontal: Float
    let angleVertical: Float
    init(angleHorizontal: Float, angleVertical: Float) {
        self.angleHorizontal = angleHorizontal
        self.angleVertical = angleVertical
    }
}
```

Adapter

```
struct NewDeathStarSuperlaserTarget: NewDeathStarSuperLaserAiming {

    private let target: OldDeathStarSuperlaserTarget

    var angleV: Double {
        return Double(target.angleVertical)
    }

    var angleH: Double {
        return Double(target.angleHorizontal)
    }

    init(_ target: OldDeathStarSuperlaserTarget) {
        self.target = target
    }
}
```

Usage

```
let target = OldDeathStarSuperlaserTarget(angleHorizontal: 14.0, angle
let newFormat = NewDeathStarSuperlaserTarget(target)

newFormat.angleH
newFormat.angleV
```

 Bridge

The bridge pattern is used to separate the abstract elements of a class from the implementation details, providing the means to replace the implementation details without modifying the abstraction.

Example

```
protocol Switch {
    var appliance: Appliance { get set }
    func turnOn()
}

protocol Appliance {
    func run()
}

final class RemoteControl: Switch {
    var appliance: Appliance
    func turnOn() {
        self.appliance.run()
    }
    init(appliance: Appliance) {
        self.appliance = appliance
    }
}

final class TV: Appliance {
    func run() {
        print("tv turned on");
    }
}

final class VacuumCleaner: Appliance {
    func run() {
        print("vacuum cleaner turned on")
    }
}
```

Usage

```
let tvRemoteControl = RemoteControl(appliance: TV())
tvRemoteControl.turnOn()

let fancyVacuumCleanerRemoteControl = RemoteControl(appliance: VacuumC
fancyVacuumCleanerRemoteControl.turnOn()
```

🌿 Composite

The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.

Example

Component

```
protocol Shape {
    func draw(fillColor: String)
}
```

Leafs

```
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 {
    private lazy var shapes = [Shape]()

    init(_ shapes: Shape...) {
        self.shapes = shapes
    }

    func draw(fillColor: String) {
        for shape in self.shapes {
            shape.draw(fillColor: fillColor)
        }
    }
}
```

Usage:

```
var whiteboard = Whiteboard(Circle(), Square())
whiteboard.draw(fillColor: "Red")
```

🍧 Decorator

The decorator pattern is used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This provides a flexible alternative to using inheritance to modify behaviour.

Example

```
protocol CostHaving {
    var cost: Double { get }
}
protocol IngredientsHaving {
    var ingredients: [String] { get }
}

typealias BeverageDataHaving = CostHaving & IngredientsHaving
struct SimpleCoffee: BeverageDataHaving {
    let cost: Double = 1.0
    let ingredients = ["Water", "Coffee"]
}

protocol BeverageHaving: BeverageDataHaving {
    var beverage: BeverageDataHaving { get }
}

struct Milk: BeverageHaving {
    let beverage: BeverageDataHaving
    var cost: Double {
        return beverage.cost + 0.5
    }
    var ingredients: [String] {
        return beverage.ingredients + ["Milk"]
    }
}
```

```
struct WhipCoffee: BeverageHaving {

    let beverage: BeverageDataHaving

    var cost: Double {
        return beverage.cost + 0.5
    }

    var ingredients: [String] {
        return beverage.ingredients + ["Whip"]
    }
}
```

Usage:

```
var someCoffee: BeverageDataHaving = SimpleCoffee()
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients
someCoffee = Milk(beverage: someCoffee)
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients
someCoffee = WhipCoffee(beverage: someCoffee)
print("Cost: \(someCoffee.cost); Ingredients: \(someCoffee.ingredients
```

🎁 Facade

The facade pattern is used to define a simplified interface to a more complex subsystem.

Example

```
final class Defaults {
    private let defaults: UserDefaults

    init(defaults: UserDefaults = .standard) {
        self.defaults = defaults
    }

    subscript(key: String) -> String? {
        get {
            return defaults.string(forKey: key)
        }
        set {
            defaults.set(newValue, forKey: key)
        }
    }
}
```

Usage

```
let storage = Defaults()
// Store
storage["Bishop"] = "Disconnect me. I'd rather be nothing"
// Read
storage["Bishop"]
```

🌿 Flyweight

The flyweight pattern is used to minimize memory usage or computational expenses by sharing as much as possible with other similar objects.

Example

```
// Instances of SpecialityCoffee will be the Flyweights
struct SpecialityCoffee {
    let origin: String
}

protocol CoffeeSearching {
    func search(origin: String) -> SpecialityCoffee?
}

// Menu acts as a factory and cache for SpecialityCoffee flyweight obj
final class Menu: CoffeeSearching {

    private var coffeeAvailable: [String: SpecialityCoffee] = [:]

    func search(origin: String) -> SpecialityCoffee? {
        if coffeeAvailable.index(forKey: origin) == nil {
            coffeeAvailable[origin] = SpecialityCoffee(origin: origin
        }

        return coffeeAvailable[origin]
    }
}

final class CoffeeShop {

    private var orders: [Int: SpecialityCoffee] = [:]
    private let menu: CoffeeSearching
```

```
    init(menu: CoffeeSearching) {
        self.menu = menu
    }

    func takeOrder(origin: String, table: Int) {
        orders[table] = menu.search(origin: origin)
    }

    func serve() {
        for (table, origin) in orders {
            print("Serving \(origin) to table \(table)")
        }
    }
}
```

Usage

```
let coffeeShop = CoffeeShop(menu: Menu())

coffeeShop.takeOrder(origin: "Yirgacheffe, Ethiopia", table: 1)
coffeeShop.takeOrder(origin: "Buziraguhindwa, Burundi", table: 3)

coffeeShop.serve()
```

☂ Protection Proxy

The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. Protection proxy is restricting access.

Example

```
protocol DoorOpening {
    func open(doors: String) -> String
}

final class HAL9000: DoorOpening {
    func open(doors: String) -> String {
        return ("HAL9000: Affirmative, Dave. I read you. Opened \(doo
    }
}
final class CurrentComputer: DoorOpening {
    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)
    }
}
```

Usage

```
let computer = CurrentComputer()
let podBay = "Pod Bay Doors"

computer.open(doors: podBay)

computer.authenticate(password: "pass")
computer.open(doors: podBay)
```

🍬 Virtual Proxy

The proxy pattern is used to provide a surrogate or placeholder object, which references an underlying object. Virtual proxy is used for loading object on demand.

Example

```
protocol HEVSuitMedicalAid {
    func administerMorphine() -> String
}
final class HEVSuit: HEVSuitMedicalAid {
    func administerMorphine() -> String {
        return "Morphine administered."
    }
}
final class HEVSuitHumanInterface: HEVSuitMedicalAid {
    lazy private var physicalSuit: HEVSuit = HEVSuit()
    func administerMorphine() -> String {
        return physicalSuit.administerMorphine()
    }
}
```

Usage

```
let humanInterface = HEVSuitHumanInterface()
humanInterface.administerMorphine()
```

Project Guidelines (L6+)

A set of best practices in my projects.

- [x] Git
- [x] Documentation
- [x] Environments
- [x] Code Style
- [x] Logging
- [x] API

Licenses & certifications

- 🏆 LeetCode Global TOP 300 (Swift: Certificate, Sources: Swift).
- 🏆 Golden Award Muad'Dib's Challenge (Swift: Certificate, Sources: Swift).
- 2022 June LeetCode Challenge (2022-06-30).
- 2022 May LeetCode Challenge (2022-05-31).
- 2022 Apr LeetCode Challenge (2022-04-30).
- LeetCode Dynamic Programming (2022-05-07).
- Graph Theory (2022-04-30).
- SQL (2022-04-26).
- Algorithm I (2022-04-30), Algorithm II (2022-05-21).
- Data Structure I (2022-04-30), Data Structure II (2022-05-21).
- Binary Search I (2022-04-28), Binary Search II (2022-05-18).
- Programming Skills I (2022-04-28), Programming Skills II (2022-05-18).
- LinkedIn Skill Asessment (Mobile): Swift (Programming Language), Object-Oriented Programming (OOP), Objective-C, C++, Ionic, JSON, XML, Android, Kotlin, Maven, Java, REST APIs.
- Health & Fitness iOS App / Fitness Motivation / AppStore (Sources: SwiftUI) @ S. Leschev.
- Utility MacOS App / Calc-It / Core (Sources: Swift) @ S. Leschev.

Latest Projects

[iOS] Live Stream & Video Chat is the best streaming and video chatting tool. (L6+)

Role: Senior iOS Developer, Tech Lead. Development architecture and new features.

Tech Stack:

- Swift 5+.
- VIPER (Dependency Injection, Assembly, Services, Interactor, Presenter, State, Adapter) + MVVM (Combine, PromiseKit).
- Alomofire, Decodable, Combine.
- GCD/Operations.
- Agora Video SDK, Chat SDK, Beautification SDK. WebRTC. GRPC.
- Modular architecture (Frameworks, Development Pods).
- SwiftGen (Localization, Image, Colors).
- SwiftLint.
- Auth: Facebook, Google, Apple ID.
- Firebase, Crashlytics, Amplitude, AppsFlyer.
- Push-notifications (Firebase).
- UIKit, Autolayout, Core Animations, Skeleton, Lottie.
- Git (Flow, CodeReview), Figma.

[iOS] Health & Fitness iOS App (L6+)

Role: Senior iOS Developer, Tech Lead. Development architecture and new features.

Tech Stack:

- Swift 5+.
- Clean Swift Architecture.
- Alamofire, ObjectMapper.
- GCD/Operations.
- AVFoundation, Streaming: HLS (Cloudflare/nginx).
- AirPlay [Composition (video+audio), Secondary Display].
- Realm.
- Modular architecture (Frameworks, Development Pods).
- SwiftGen (Localization, Image, Colors).
- SwiftLint.
- Auth: Facebook, Google, Apple ID, Fitbit.
- Amplitude, Crashlytics, AppsFlyer (+OneLink).
- Analytics: Facebook (SKAd + Conversions API).
- Push-notifications (OneSignal).
- UIKit, Autolayout, Core Animations, Lottie.
- Git (Flow, CodeReview), Zeplin, Figma, Sketch.

[iOS] Health & Fitness iOS App (Motivations Coach, Pet Project) (L6+)

Role: iOS Developer.

Tech Stack:

- SwiftUI.
- Watch Extension (WatchOS).
- AppClip Extension.
- Widget (iOS 14).
- ObjectMapper.
- URLSession.
- Keychain.
- Lottie.
- Push Notifications.
- GCD/Operations.
- Git, Figma, Sketch.

Website: motivations.coach

Sources: SwiftUI

Contacts

I have a clear focus on time-to-market and don't prioritize technical debt.

🚀 #startups #management #cto #swift #typescript #database

📧 Email: sergey.leschev@gmail.com

👋 LinkedIn: https://linkedin.com/in/sergeyleschev

👋 Twitter: https://twitter.com/sergeyleschev

👋 Github: https://github.com/sergeyleschev

🌎 Website: https://sergeyleschev.github.io

🖨️ PDF: Download

ALT: SIARHEI LIASHCHOU

leader, knowledge, qualifications, education, tips, skills, multitasking, references, success, work, job, tie, challenges, abilities, impress, responsibility, future, weeknesses, benefits, results, team player, strengths, interview, degress, examples, strengths, experienced, problem solver, candidate, agency, objective, initiative, team, dreams, conflict, can-do, training, questions, job, work, career, created, swift, typescript, javascript, sql, nosql, postgresql, oracle, sql server, react, redux, swiftui, objective-c, devops, aws, mongodb, pl/sql, angular, project management, nodejs, nextjs, nestjs, api, agile, amplitude, analytics, appclip, appstore, bash, css, jira, confluence, git, graphql, html, html5, mvp, mvvm, nginx, ssh, prime react, rest, teamcity, typeorm, uikit, uml, viper, widgets, xcode, json, linux, docker, mobx, tvOS, watchOS