

i🍏S 환경에서 SOLID 연습하기

Mason

Agenda

✓ Why?

✓ SOLID?

✓ S, O, L, I, D


✓ Example

✓ Summary

Why?



결합도는 낮게 

응집도는 높게 

유지보수의 용이성
재사용성
생산성

SOLID란?

SOLID (객체 지향 설계)

위키백과, 우리 모두의 백과사전.

컴퓨터 프로그래밍에서 SOLID란 로버트 마틴^{[1][2]}이 2000년대 초반^[3]에 명명한 객체 지향 프로그래밍 및 설계의 다섯 가지 기본 원칙을 마이클 페더스가 두문자어 기억술로 소개한 것이다. 프로그래머가 시간이 지나도 유지 보수와 확장이 쉬운 시스템을 만들고자 할 때 이 원칙들을 함께 적용할 수 있다.^[3] SOLID 원칙들은 소프트웨어 작업에서 프로그래머가 소스 코드가 읽기 쉽고 확장하기 쉽게 될 때까지 소프트웨어 소스 코드를 리팩터링하여 코드 냄새를 제거하기 위해 적용할 수 있는 지침이다. 이 원칙들은 애자일 소프트웨어 개발과 적응적 소프트웨어 개발의 전반적 전략의 일부다.^[3]

✓ 객체지향 프로그래밍

✓ 설계의 5가지 원칙

✓ 유지 보수가 쉬운

✓ 읽기 쉬운

✓ 확장이 쉬운

+

@

SOLID

SRP : 단일 책임 원칙 (Single responsibility principle)

- ✓ 한 클래스(함수)는 하나의 책임만 가져야 한다.
- ✓ 클래스 코드를 수정할 이유는 하나여야 한다.



Check Point

- ✓ 클래스의 인스턴스 변수가 너무 많다.
- ✓ 속성과 상관없는 메서드가 많다.
- ✓ 클래스나 메서드를 설명하기 위해서 'and, if, or'를 많이 사용한다.

SOLID

Example

before

```
class Handler {  
    func handle() {  
        let data = requestDataToAPI()  
        let array = parse(data: data)  
        saveToDB(array: array)  
    }  
  
    private func requestDataToAPI() -> Data {  
        // send API request and wait the response  
    }  
  
    private func parse(data: Data) -> [String] {  
        // parse the data and create the array  
    }  
  
    private func saveToDB(array: [String]) {  
        // save the array in a database (CoreData/Realm/...)  
    }  
}
```


after

```
class Handler {  
  
    let apiHandler: APIHandler  
    let parseHandler: ParseHandler  
    let dbHandler: DBHandler  
  
    init(apiHandler: APIHandler,  
        parseHandler: ParseHandler,  
        dbHandler: DBHandler) {  
        self.apiHandler = apiHandler  
        self.parseHandler = parseHandler  
        self.dbHandler = dbHandler  
    }  
  
    func handle() {  
        let data = apiHandler.requestDataToAPI()  
        let array = parseHandler.parse(data: data)  
        dbHandler.saveToDB(array: array)  
    }  
}
```

```
class APIHandler {  
    func requestDataToAPI() -> Data { ... }  
}  
  
class ParseHandler {  
    func parse(data: Data) -> [String] { ... }  
}  
  
class DBHandler {  
  
    func saveToDB(array: [String]) { ... }  
}
```

SOLID

OCP : 개방 폐쇄 원칙 (Open-closed principle)

✓ 확장(extension)에는 열려있으나 변경(modification)에는 닫혀 있어야 한다.

= 기능 및 요구사항이 추가될 때 기존의 요소의 변경을 최소화

✓ 객체지향 프로그래밍의 가장 큰 유연성, 재사용성, 유지보수성을 얻을 수 있다.

Check Point

✓ 새로운 기능이나 케이스가 추가될 때 마다 기존의 코드를 변경해야 한다.

✓ 자신의 속성보다는 외부의 속성을 의존하고 있지는 않은가? (결합도)

✓ 인터페이스보다는 구현한 타입에 의존하고 있지는 않은가?

SOLID

Example

before

```
enum Country {
    case korea
    case japan
    case china
}

class Flag {
    let country: Country

    init(country: Country) {
        self.country = country
    }
}

func printNameOfCountry(flag: Flag) {
    switch flag.country {
    case .china:
        print("중국")
    case .korea:
        print("한국")
    case .japan:
        print("일본")
    }
}
```

```
enum Country {
    case korea
    case japan
    case china

    var name: String {
        switch self {
        case .china:
            return "중국"
        case .korea:
            return "한국"
        case .japan:
            return "일본"
        }
    }
}

class Flag {
    let country: Country

    init(country: Country) {
        self.country = country
    }
}

func printNameOfCountry(flag: Flag) {
    print(flag.country.name)
}
```

after

```
protocol Country {  
    var name: String { get }  
}  
  
struct Korea: Country {  
    let name: String = "한국"  
}  
  
struct Japan: Country {  
    let name: String = "일본"  
}  
  
struct China: Country {  
    let name: String = "중국"  
}
```

```
class Flag {  
    let country: Country  
  
    init(country: Country) {  
        self.country = country  
    }  
}  
  
func printNameOfCountry(flag: Flag) {  
    print(flag.country.name)  
}
```

SOLID

LSP : 리스코프 치환 원칙 (Liskov substitution principle)

- ✓ 하위 타입은 기반 타입을 대체할 수 있어야 한다.
- ✓ 상속을 사용했을 때 서브클래스는 자신의 슈퍼클래스 대신 사용되도 같은 동작을 해야한다.

Check Point

- ✓ 자식클래스에 너무 많은 override가 구현되어 있다.
- ✓ 수직적 확장과 수평적 확장 중 어느것이 필요한 상황인지 생각해본다.
- ! 상속을 사용하면 강한 결합도가 생기기 때문에 주의 필요.

SOLID

Example

before

```
class 직사각형 {  
    var 너비: Float = 0  
    var 높이: Float = 0  
  
    var 넓이: Float {  
        return 너비 * 높이  
    }  
}  
  
class 정사각형: 직사각형 {  
    override var 너비: Float {  
        didSet {  
            높이 = 너비  
        }  
    }  
}
```

```
func printArea(of 직사각형: 직사각형) {  
    직사각형.높이 = 5  
    직사각형.너비 = 2  
    print(직사각형.넓이)  
}  
  
let rectangle = 직사각형()  
printArea(of: rectangle) // 10  
  
let square = 정사각형()  
printArea(of: square) // 4
```


after

```
protocol 사각형 {  
    var 넓이: Float { get }  
}
```

```
class 직사각형: 사각형 {  
    private let 너비: Float  
    private let 높이: Float  
  
    init(너비: Float, 높이: Float) {  
        self.너비 = 너비  
        self.높이 = 높이  
    }  
  
    var 넓이: Float {  
        return 너비 * 높이  
    }  
}
```

```
class 정사각형: 사각형 {  
    private let 변의길이: Float  
  
    init(변의길이: Float) {  
        self.변의길이 = 변의길이  
    }  
  
    var 넓이: Float {  
        return 변의길이 * 변의길이  
    }  
}
```

SOLID

DIP : 의존관계 역전 원칙 (Dependency inversion principle)

- ✓ 상위레벨 모듈은 하위레벨 모듈에 의존하면 안된다.
- ✓ 두 모듈은 추상화된 인터페이스(프로토콜)에 의존해야 한다.
- ✓ 추상화 된 것은 구체적인 것에 의존하면 안되고,
구체적인 것이 추상화된 것에 의존해야 한다.

Check Point

- ✓ 내부적으로 생성하는 하위 모듈이 존재하는가? (주입X)
- ✓ 상위레벨 모듈이 재사용 가능한가?
- ✓ 하위레벨 모듈의 구체적인 타입이 존재하는가?

SOLID

Example

before

```
class 구형맥북 {  
    func 전원켜기() { }  
}  
  
class 개발자 {  
    let 장비: 구형맥북 = 구형맥북()  
  
    func 개발시작() {  
        장비.전원켜기()  
    }  
}
```

의존성 주입 DI

```
class 구형맥북 {  
    func 전원켜기() { }  
}  
  
class 개발자 {  
    let 장비: 구형맥북  
  
    init(장비: 구형맥북) {  
        self.장비 = 장비  
    }  
  
    func 개발시작() {  
        장비.전원켜기()  
    }  
}
```

구체화보단 추상화된 인터페이스에 의존하도록

```
protocol 장비 {  
    func 전원켜기()  
}  
  
class 개발자 {  
    let 장비: 장비  
  
    init(맥북: 장비) {  
        self.장비 = 맥북  
    }  
  
    func 개발시작() {  
        장비.전원켜기()  
    }  
}
```

```
class 구형맥북: 장비 {  
    func 전원켜기() { }  
}  
  
class 신형맥북: 장비 {  
    func 전원켜기() { }  
}  
  
class 테스트용장비: 장비 {  
    func 전원켜기() { }  
}
```

SOLID

ISP : 인터페이스 분리 원칙 (Interface segregation principle)

- ✓ 클래스 내에서 사용하지 않는 인터페이스는 구현하지 말아야 한다.
- ✓ ISP는 인터페이스에 대한 SRP와도 같다.
 - = 인터페이스도 필요한(관련있는) 기능에 따라 분리하자.

Check Point

- ✓ 프로토콜을 채택하고 어쩔 수 없이 의미없는 구현을 하고 있진 않은가?
- ✓ 해당 프로토콜이 하나의 역할을 하는가?

SOLID

Example

before

```
protocol GestureProtocol {  
    func didTap()  
    func didLongTap()  
    func didDoubleTap()  
}  
  
class GestureButton: GestureProtocol {  
    func didTap() { }  
    func didLongTap() { }  
    func didDoubleTap() { }  
}  
  
class DoubleTapButton: GestureProtocol {  
    func didDoubleTap() { }  
  
    // Uesless..  
    func didTap() { }  
    func didLongTap() { }  
}
```

after

```
protocol GestureProtocol {  
    func didTap()  
    func didLongTap()  
    func didDoubleTap()  
}
```



```
protocol TapGestureProtocol {  
    func didTap()  
}  
  
protocol LongTapGestureProtocol {  
    func didLongTap()  
}  
  
protocol DoubleTapGestureProtocol {  
    func didDoubleTap()  
}
```

```
class GestureButton: TapGestureProtocol,  
                    LongTapGestureProtocol,  
                    DoubleTapGestureProtocol {  
    func didTap() { }  
    func didLongTap() { }  
    func didDoubleTap() { }  
}  
  
class DoubleTapButton: DoubleTapGestureProtocol {  
    func didDoubleTap() { }  
}  
  
class LongAndTapButton: LongTapGestureProtocol,  
                       TapGestureProtocol {  
    func didLongTap() { }  
    func didTap() { }  
}
```

```
class GestureButton: TapGestureProtocol,
                    LongTapGestureProtocol,
                    DoubleTapGestureProtocol {

    func didTap() { }
    func didLongTap() { }
    func didDoubleTap() { }
}
```

```
class LongAndTapButton: DoubleTapGestureProtocol & LongTapGestureProtocol {
```

```
    func doSomething(button: DoubleTapGestureProtocol) {
        button.didDoubleTap()
        button.didLongTap()
    }
```

```
class LongAndTapButton: LongTapGestureProtocol,
                        TapGestureProtocol {

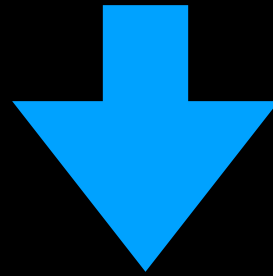
    func didLongTap() { }
    func didTap() { }
}
```

iS Example

Massive View Controller의 역할을 나눠보자

- 컨트롤러가 데이터를 직접적으로 관리하지 않고 데이터를 관리하는 객체로 역할 분담

```
class HamburgerViewController: UIViewController, UITableViewDelegate {  
    @IBOutlet weak var tableView: UITableView!  
    private let hamburgerCellIdentifier = "cell"  
    private var hamburgers: [Hamburger] = [ShrimpBurger(), BulgogiBurger(), CheeseBurger()]  
}
```



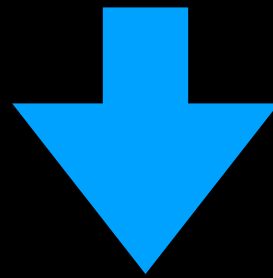
```
class HamburgersManager {  
    private var hamburgers: [Hamburger] = [ShrimpBurger(), BulgogiBurger(), CheeseBurger()]  
}
```

```
class HamburgerViewController: UIViewController, UITableViewDelegate {  
    @IBOutlet weak var tableView: UITableView!  
    private let hamburgerCellIdentifier = "cell"  
    private let hamburgersManager: HamburgersManager = HamburgersManager()  
}
```

Massive View Controller의 역할을 나눠보자

- 사용하는 곳에선 어떻게 변할까?

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {  
    return hamburgers.count  
}
```



```
class HamburgersManager {  
    private var hamburgers: [Hamburger] = [ShrimpBurger(), BulgogiBurger(), CheeseBurger()]  
  
    var numberOfHamburgers: Int {  
        return hamburgers.count  
    }  
}
```

```
func tableView(_ tableView: UITableView, numberOfRowsInSectionSection section: Int) -> Int {  
    return hamburgersManager.numberOfHamburgers  
}
```

Massive View Controller의 역할을 나눠보자

- 원하는 데이터를 모델관리 객체에서 가져오자

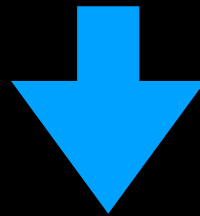
```
class HamburgersManager {  
    private var hamburgers: [Hamburger] = [ShrimpB  
    var numberOfHamburgers: Int { ... }  
  
    func hamburger(at index: Int) -> Hamburger {  
        return hamburgers[index]  
    }  
}
```

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {  
    let cell = tableView.dequeueReusableCell(withIdentifier: hamburgerCellIdentifier, for: indexPath)  
    if let hamburgerTableViewCell = cell as? HamburgerTableViewCell {  
        hamburgerTableViewCell.burgerImageView.image = hamburgersManager.hamburger(at: indexPath.row).image  
        hamburgerTableViewCell.nameLabel.text = hamburgersManager.hamburger(at: indexPath.row).name  
        hamburgerTableViewCell.sauceLabel.text = hamburgersManager.hamburger(at: indexPath.row).sauce  
        hamburgerTableViewCell.pattyLabel.text = hamburgersManager.hamburger(at: indexPath.row).patty  
        // 전혀 짧아지지 않은거 같은데?  
    }  
    return cell  
}
```


Massive View Controller의 역할을 나눠보자

- View(Cell)의 속성들의 설정하는 역할을 속성을 가지고 있는 View(Cell)에게 나눠보자.

```
class HamburgerTableViewCell: UITableViewCell {
    @IBOutlet weak var burgerImageView: UIImageView!
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var sauceLabel: UILabel!
    @IBOutlet weak var pattyLabel: UILabel!
}
```



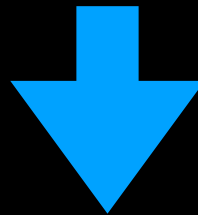
```
class HamburgerTableViewCell: UITableViewCell {
    @IBOutlet weak var burgerImageView: UIImageView!
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var sauceLabel: UILabel!
    @IBOutlet weak var pattyLabel: UILabel!

    func set(by burger: Hamburger) {
        nameLabel.text = burger.name
        sauceLabel.text = burger.sauce
        pattyLabel.text = burger.patty
        burgerImageView.image = burger.image
    }
}
```

Massive View Controller의 역할을 나눠보자

- 사용하는 곳에선 어떻게 변할까?

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: hamburgerCellIdentifier, for: indexPath)
    if let hamburgerTableViewCell = cell as? HamburgerTableViewCell {
        hamburgerTableViewCell.burgerImageView.image = hamburgersManager.hamburger(at: indexPath.row).image
        hamburgerTableViewCell.nameLabel.text = hamburgersManager.hamburger(at: indexPath.row).name
        hamburgerTableViewCell.sauceLabel.text = hamburgersManager.hamburger(at: indexPath.row).sauce
        hamburgerTableViewCell.pattyLabel.text = hamburgersManager.hamburger(at: indexPath.row).patty
        // 전혀 짧아지지 않은거 같은데?
    }
    return cell
}
```



```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: hamburgerCellIdentifier, for: indexPath)
    if let hamburgerTableViewCell = cell as? HamburgerTableViewCell {
        let hamburger = hamburgerStorage.hamburger(at: indexPath.row)
        hamburgerTableViewCell.set(by: hamburger)
        // 줄어들었다! 설정해줘야하는 사항이 늘어나도 뷰컨트롤러의 코드가 늘어나지 않는다.
    }
    return cell
}
```

구체적인 타입이 아닌 프로토콜(추상화)에 의존해보자

```
class ShrimpBurger: HamburgerSettable {  
    let image: UIImage  
    let name: String  
    let sauce: String  
    let patty: String  
  
    init(image: UIImage, name: String, sauce: String, patty: String) {  
        self.image = image  
        self.name = name  
        self.sauce = sauce  
        self.patty = patty  
    }  
  
    convenience init() {  
        self.init(image: UIImage(named: "새우")!, name: "새우버거", sauce: "마요네즈", patty: "새우")  
    }  
}
```

구체적인 타입이 아닌 프로토콜(추상화)에 의존해보자

```
protocol HamburgerSettable {
    var name: String { get }
    var sauce: String { get }
    var patty: String { get }
    var image: UIImage { get }
}

protocol HamburgerStoragable {
    var numberOfHamburgers: Int { get }
    func hamburger(at index: Int) -> HamburgerSettable
}
```

```
class HamburgerTableViewCell: UITableViewCell {
    @IBOutlet weak var burgerImageView: UIImageView!
    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var sauceLabel: UILabel!
    @IBOutlet weak var pattyLabel: UILabel!

    func set(by burger: HamburgerSettable) {
        nameLabel.text = burger.name
        sauceLabel.text = burger.sauce
        pattyLabel.text = burger.patty
        burgerImageView.image = burger.image
    }
}
```

```
class HamburgerStorage: HamburgerStoragable {
    private let hamburgers: [HamburgerSettable] = [ShrimpBurger(), BulgogiBurger(), CheeseBurger()]

    var numberOfHamburgers: Int {
        return hamburgers.count
    }

    func hamburger(at index: Int) -> HamburgerSettable {
        return hamburgers[index]
    }
}
```

의존성 주입을 해보자

```
class HamburgerViewController: UIViewController, UITableViewDelegate {
    @IBOutlet weak var tableView: UITableView!
    private let hamburgerCellIdentifier = "cell"
    let hamburgerStorage: HamburgerStoragable

    // 이 외에도 의존성을 주입하는 방법은 다양합니다.
    init(hamburgerStorage: HamburgerStoragable) {
        self.hamburgerStorage = hamburgerStorage
        super.init(nibName: nil, bundle: nil)
    }

    // ... 생략
```


```
class HamburgerStorage: HamburgerStoragable {
    private var hamburgers: [HamburgerSettable]


    init(hamburgers: [HamburgerSettable]) {
        self.hamburgers = hamburgers
    }

    var numberOfHamburgers: Int {
        return hamburgers.count
    }

    func hamburger(at index: Int) -> HamburgerSettable {
        return hamburgers[index]
    }
}
```

Summary

결합도는 낮게 

응집도는 높게 

만병통치약은 없다 

참고

위키피디아 : [https://ko.wikipedia.org/wiki/SOLID \(객체 지향 설계\)#cite_note-ub-solid-2](https://ko.wikipedia.org/wiki/SOLID_(객체_지향_설계)#cite_note-ub-solid-2)
<https://en.wikipedia.org/wiki/SOLID>

SOLID Principles every Developer Should Know:

<https://blog.bitsrc.io/solid-principles-every-developer-should-know-b3bfa96bb688>

<https://github.com/ochococo/OOD-Principles-In-Swift>

<http://www.nextree.co.kr/p6960/>

<https://wnstkdyu.github.io/2018/08/08/solidWithSwift/>

<https://soojin.ro/blog/solid-principles-in-swift>

<https://namu.wiki/w/객체%20지향%20프로그래밍>

<https://medium.com/ios-expert-series-or-interview-series/solid-design-principle-using-swift-34bb1731cfb3>