

iOS 세미나 4

2023 와플스튜디오 루키 세미나

2023.11.12, 박신홍

오늘 배울 내용

- 과제 2 리뷰
- 의존성 관리
 - 의존성 주입
 - 의존성 역전의 원칙
- 유닛 테스트
- 코드 아키텍쳐



의존성이란?

- 프로그래밍에서 의존성은 다양한 의미로 사용됨
 - 클래스(객체) 간의 의존성
 - 모듈 간의 의존성
 - 프레임워크/라이브러리 의존성
 -
- 오늘 얘기하는 건 대부분 **클래스(객체)** 간 의존성

클래스 간 의존성

간단한 예제

- 자동차가 굴러가려면 엔진이 필요하다.
- = 자동차는 엔진에 의존한다.
- = 엔진이 바뀌면 자동차는 다르게 동작한다.
- = 자동차는 엔진을 소유한다.

```
class Engine {  
    func start() {  
        // Engine starts  
    }  
  
class Car {  
    let engine = Engine()  
  
    func startCar() {  
        engine.start()  
    }  
}
```

Car

Engine

BMW 뉴 5시리즈

Sheer Driving Pleasure

👉 내전기차



- 차체 디자인은 한 번만 하고, 엔진만 다르게 해서 두 가지 모델을 날로먹고 싶다출시하고 싶다.

높은 결합도

- 오른쪽 예제에서 자동차와 엔진은 **높은 결합도**를 가짐 (결합도가 높다는 것은 웬만하면 좋은 뜻이 아님)
- 차체 디자인이 가솔린 엔진과 결합되어 있으므로, 설계대로 찍어내면 내연기관 차량이 탄생
- 전기차를 만들고 싶으면, 비슷한 설계도를 하나 더 만들어야 됨
- 코드 중복!

```
class BMWi5 {  
    let engine = ElectricEngine()  
  
    func startCar() {  
        engine.start()  
    }  
}
```

```
class GasolineEngine {  
    func start() {  
        // Engine starts  
    }  
  
    class BMW5 {  
        let engine = GasolineEngine()  
  
        func startCar() {  
            engine.start()  
        }  
    }  
  
    let myBMWCar = BMW5()
```

결합도를 느슨하게 하자

의존성 주입 (Dependency Injection)

- 생성자를 통해 Engine의 종류를 결정짓도록 하면, 더 이상 차체 디자인이 엔진에 종속되지 않는다(의존하지 않는다).
- 차체 디자인과 엔진의 결합도가 느슨해졌다!
- 이제 하나의 디자인에 엔진만 바꿔끼워서 두 가지 모델의 출시가 가능해졌다.

```
class Engine {}

class ElectricEngine: Engine {
    func startWeeing() {
        // Weeing
    }
}

class GasolineEngine: Engine {
    func startBooang() {
        // Booang
    }
}

class BMW {
    private let engine: Engine

    init(engine: Engine) {
        self.engine = engine
    }

    func start() {
        if let engine = engine as? GasolineEngine {
            engine.startBooang()
        } else if let engine = engine as? ElectricEngine {
            engine.startWeeing()
        }
    }
}

let bmw5 = BMW(engine: GasolineEngine())
let bmwi5 = BMW(engine: ElectricEngine())
```

문제가 있다..

- 각 엔진의 시동 거는 방법이 달라진다면?
- 미래에 수소차, 핵융합차 같은 것들이 생긴다면?
- 하위 객체(=엔진)에 변경사항이 생기고 종류가 추가될 때마다, 상위 객체(=차)의 설계도 같이 수정해주어야 함
- 차량 구조가 복잡해질수록 유지보수에 굉장한 걸림돌이 됨
- ➡ 차체 디자인하는 사람이 각 엔진의 특성과 종류까지 꿰고 있어야 함

```
class Engine {}

class ElectricEngine: Engine {
    func startWeeing() {
        // Weeing
    }
}

class GasolineEngine: Engine {
    func startBooang() {
        // Booang
    }
}

class BMW {
    private let engine: Engine

    init(engine: Engine) {
        self.engine = engine
    }

    func start() {
        if let engine = engine as? GasolineEngine {
            engine.startBooang()
        } else if let engine = engine as? ElectricEngine {
            engine.startWeeing()
        }
    }
}

let bmw5 = BMW(engine: GasolineEngine())
let bmwi5 = BMW(engine: ElectricEngine())
```

추상화에 의존하자

의존성 역전 (Dependency Inversion)

- 의존성 역전: 상위 모듈이 하위 모듈의 구현 내용에 의존하면 안되고, 상위 모듈과 하위 모듈 모두가 추상화된 내용에 의존해야 한다
- 번역기 ON
 - 상위 모듈 → 차량
 - 하위 모듈 → 엔진
 - 구현 내용 → 클래스
 - 추상화된 내용 → 프로토콜

```
protocol Engine {
    func start()
}

class ElectricEngine: Engine {
    func start() {
        // weeing
    }
}

class GasolineEngine: Engine {
    func start() {
        // booang
    }
}

class BMW {
    private let engine: Engine
    init(engine: Engine) {
        self.engine = engine
    }
}

let bmw5 = BMW(engine: GasolineEngine())
let bmwi5 = BMW(engine: ElectricEngine())
```

추상화에 의존하자

의존성 역전 (Dependency Inversion)

- 차량 설계도 입장에서는, 엔진이 전기 엔진이든 가솔린 엔진이든 관심 없다.
- Engine이라는 추상화된 프로토콜을 준수하기만 한다면, 차량 설계도는 정상 작동할 것이다.
- 👉 차체 디자이너는 차체에만, 엔진 만드는 사람은 엔진을 잘 구현하는데에만 집중하면 됨
- cf. IoC (Inversion of Control, 제어의 역전)
 - 일반적인 소프트웨어의 흐름: 차량 설계도가 내부 엔진의 작동 방식까지 결정
 - 역전된 흐름: 외부에서 차량이 어떻게 작동할지 결정

```
protocol Engine {
    func start()
}

class ElectricEngine: Engine {
    func start() {
        // weeing
    }
}

class GasolineEngine: Engine {
    func start() {
        // booang
    }
}

class BMW {
    private let engine: Engine

    init(engine: Engine) {
        self.engine = engine
    }

    func start() {
        engine.start()
    }
}

let bmw5 = BMW(engine: GasolineEngine())
let bmwi5 = BMW(engine: ElectricEngine())
```

유닛 테스트

- "차체 디자이너는 차체에만, 엔진 만드는 사람은 엔진을 잘 구현하는데에만 집중하면 됨"
- 차체 디자이너는 차체만 테스트하고, 엔지니어는 엔진만 테스트하면 된다.
 - 말 그대로 한 번에 한 가지 단위만 집중적으로 테스트하면 된다는 뜻.
- 의존성이 느슨하게 결합되었을 때만 가능한 일!

유닛 테스트

차체 디자이너가 할 일:

```
import XCTest

class MockEngine: Engine {
    var startWasCalled = false

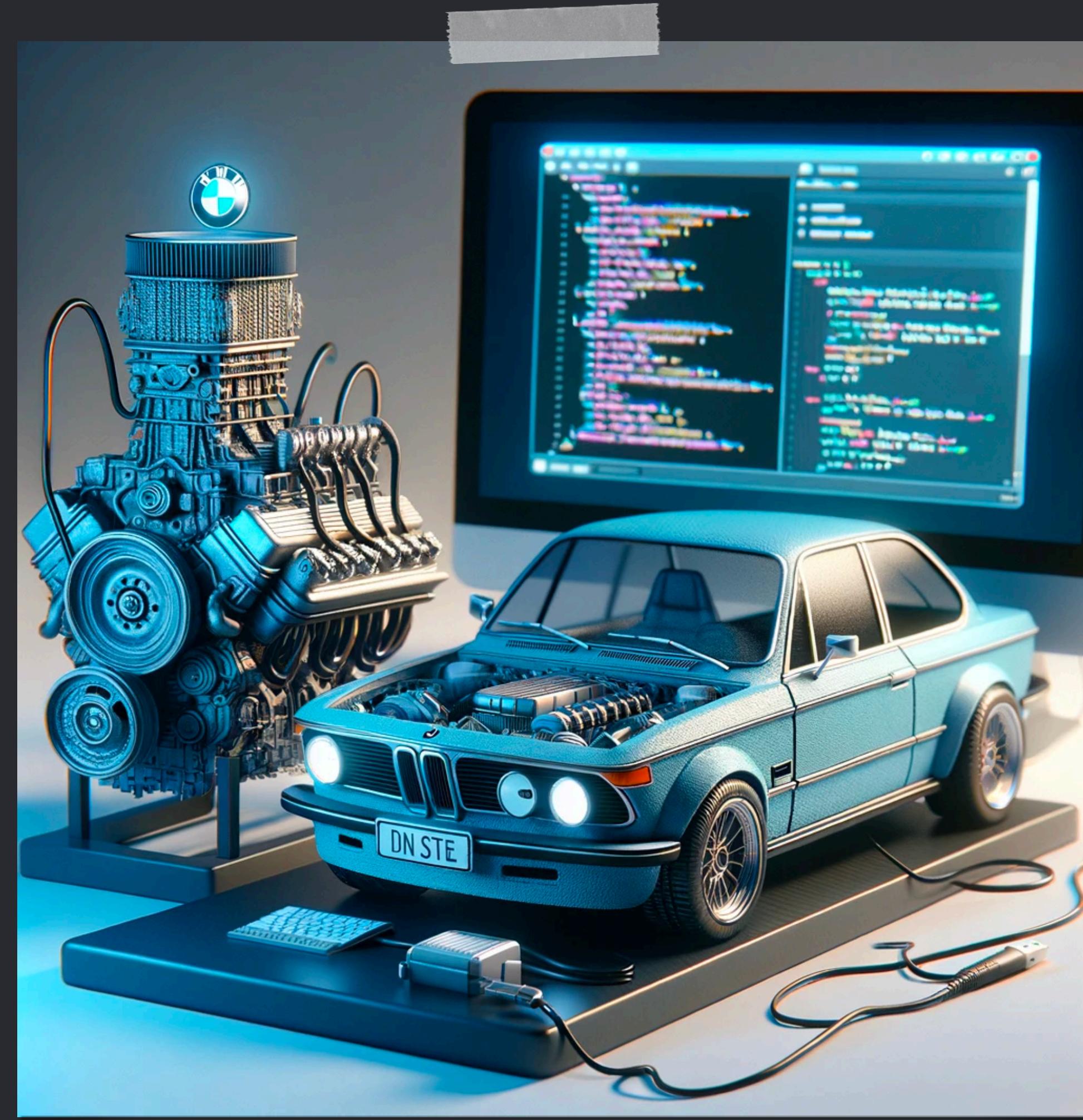
    func start() {
        startWasCalled = true
    }
}

class BMWTests: XCTestCase {

    func testBMWStartCallsEngineStart() {
        // Arrange
        let mockEngine = MockEngine()
        let bmw = BMW(engine: mockEngine)

        // Act
        bmw.start()

        // Assert
        XCTAssertTrue(mockEngine.startWasCalled, "start() method should be called on the engine")
    }
}
```



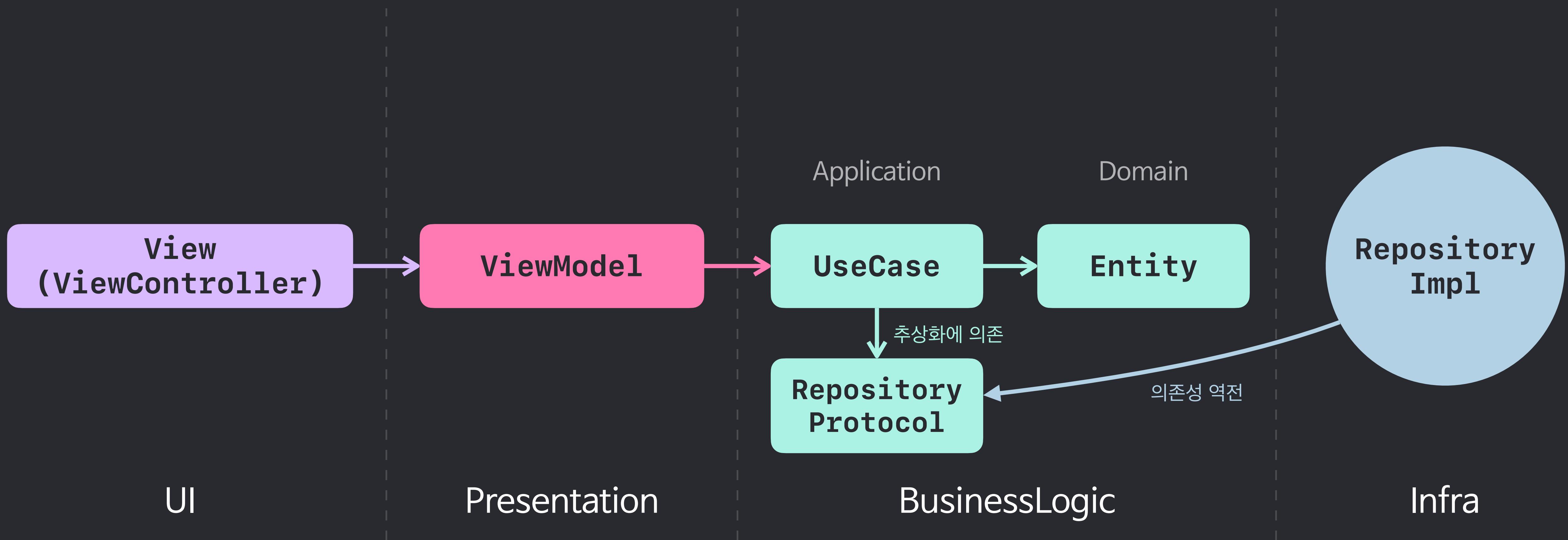
코드 아키텍쳐

더러운 코드부터 구경

- 데이터와 UI 관련 코드가 짬뽕되어 있음.
 - 👉 재사용 불가
 - 👉 테스트 불가
 - 👉 유지보수 어려움
- 응답이 String 타입이라는 것을 가정하고 디코드하고 있음
 - 👉 향후 응답 형식이 바뀌면 UI 관련 코드도 수정해주어야 함

```
// In UIViewController
func fetchDataAndUpdateUI() {
    self.activityIndicator.startAnimating()
    let url = URL(string: "https://example.com/data")!
    let task = URLSession.shared.dataTask(with: url) { [weak self] data, response, error in
        guard let self = self else { return }
        DispatchQueue.main.async {
            self.activityIndicator.stopAnimating()
        }
        if let error = error {
            DispatchQueue.main.async {
                self.showAlert(message: "Error occurred")
            }
            return
        }
        guard let data = data,
              let dataString = String(data: data, encoding: .utf8) else {
            DispatchQueue.main.async {
                self.showAlert(message: "Invalid data received")
            }
            return
        }
        // Updating UI on the main thread
        DispatchQueue.main.async {
            self.dataLabel.text = "Fetched Data: \(dataString)"
            self.saveDataLocally(dataString)
        }
    }
    task.resume()
}
```

클린 아키텍쳐



클린 코드

UI → Presentation

- Combine을 통해 ViewModel 데이터를 UI에 반영
 - (물론 Delegate 패턴으로 할 수도 있음)
- 데이터는 Presentation 레이어 밑에서부터 알아서 불러왔다고 가정
- UI 레이어는 UI 관련 작업에만 집중

```
class ExampleViewController: UIViewController {  
    @IBOutlet var dataLabel: UILabel!  
    @IBOutlet var activityIndicator: UIActivityIndicatorView!  
    private var viewModel = DataViewModel()  
    private var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        bindViewModel()  
        viewModel.fetchData()  
    }  
  
    private func bindViewModel() {  
        viewModel.$dataString  
            .receive(on: DispatchQueue.main)  
            .sink { [weak self] dataString in  
                self?.dataLabel.text = dataString  
                self?.activityIndicator.stopAnimating()  
            }  
            .store(in: &cancellables)  
  
        viewModel.$showAlert  
            .receive(on: DispatchQueue.main)  
            .sink { [weak self] showAlert in  
                if showAlert {  
                    self?.showAlert(message: self?.viewModel.alertMessage ?? "")  
                }  
            }  
            .store(in: &cancellables)  
    }  
  
    func showAlert(message: String) {  
        let alert = UIAlertController(title: "Alert", message: message)  
        alert.addAction(UIAlertAction(title: "OK", style: .default))  
        present(alert, animated: true)  
    }  
}
```

클린 코드

Presentation → BusinessLogic

- ViewModel(Presentation Layer)는 UseCase(BusinessLogic)에 의존
- ViewModel은 비즈니스 로직에는 관심이 없어야 한다 (느슨한 결합).
- UseCase가 반환한 데이터를 가지고 어떻게 UI에 잘 보이도록 가공할지만 고민하면 됨.

```
class DataViewModel: ObservableObject {  
    @Published var dataString: String = ""  
    @Published var showAlert: Bool = false  
    @Published var alertMessage: String = ""  
    private var dataUseCase: DataUseCase  
    private var cancellables = Set<AnyCancellable>()  
  
    init(dataUseCase: DataUseCase = DataUseCase()) {  
        self.dataUseCase = dataUseCase  
    }  
  
    func fetchData() {  
        dataUseCase.execute { [weak self] result in  
            DispatchQueue.main.async {  
                switch result {  
                    case .success(let dataModel):  
                        self?.dataString = dataModel.dataString  
                        self?.saveDataLocally(dataModel.dataString)  
                    case .failure:  
                        self?.showAlert = true  
                        self?.alertMessage = "Error occurred"  
                }  
            }  
        }  
    }  
}
```

클린 코드

BusinessLogic / Domain / Entity

- 내가 만드는 앱의 도메인 비즈니스 규칙이 포함됨
- 서버에서 받은 데이터를 규칙에 맞게 변환하고, 유효성을 검사하고, 등등.

```
struct DataModel {  
    let rawData: String  
  
    // Computed property that applies business rules to rawData  
    var processedData: String {  
        // Example business rule: Convert rawData to a specific format  
        // or apply some validation or transformation.  
        return transformData(rawData)  
    }  
  
    // A method to encapsulate some business logic  
    private func transformData(_ data: String) -> String {  
        // Implement transformation logic here.  
        // For example, you might parse JSON, sanitize strings,  
        // perform calculations, etc., depending on your domain's needs.  
  
        // This is a placeholder for illustrative purposes.  
        return "Processed: \(data)"  
    }  
  
    // Additional business rules can be added as methods or computed properties  
    // Example: Validation of data integrity  
    var isValidData: Bool {  
        // Implement validation logic here.  
        // Return true if the data meets certain criteria; otherwise, false.  
  
        // This is a placeholder for illustrative purposes.  
        return !rawData.isEmpty  
    }  
}
```

클린 코드

BusinessLogic / Application / Repository

- 데이터를 불러오는 Repository의 프로토콜을 선언만 해준다.
- [주의] 실제 Repository의 구현은 Infra 레이어에서 해준다.
 - 왜? 저수준에서 데이터를 어떻게 취득하는지는 비즈니스 로직에서 알 바가 아님.
 - REST API를 사용하든, RPC, 웹소켓, 로컬 DB, 등등 무엇을 사용하든 그것은 비즈니스 로직과는 관계가 없음

```
protocol DataRepositoryProtocol {  
    func fetchData(completion: @escaping (Result<DataModel, Error>) -> Void)  
}
```

클린 코드

BusinessLogic / Application / UseCase

- Repository를 통해 불러온 데이터를, 특정 비즈니스 로직에 따라 가공한 뒤 반환한다.
- 의존성 역전 원칙이 적용된 것 확인!
- 마찬가지로 데이터를 어떻게 불러오는지는 관심 없고, 그걸 어떻게 처리하고 가공하는지에만 관심 있다.
- FAQ: ViewModel과 UseCase 모두 데이터를 가공한다고 했는데, 둘의 차이는?

```
class DataUseCase {  
    private var dataRepository: DataRepositoryProtocol  
  
    init(dataRepository: DataRepositoryProtocol) {  
        self.dataRepository = dataRepository  
    }  
  
    func execute(completion: @escaping (Result<DataModel, Error>) -> Void) {  
        dataRepository.fetchData(completion: completion)  
    }  
}
```

클린 코드

Infra

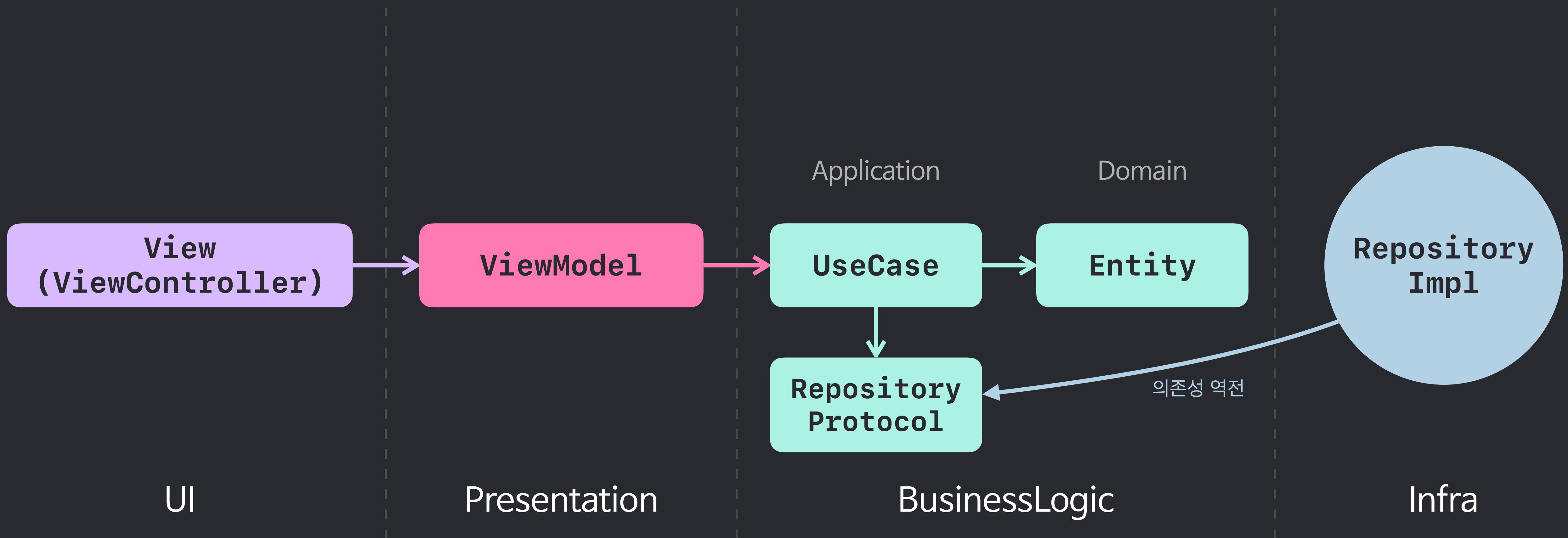
- Repository의 실제 구현체. 저수준에서 데이터를 불러오는 역할을 담당한다.
- 서버에서 데이터를 불러오는 것은 Infra 레이어의 많은 역할 중 하나.
- iOS 내부 다양한 프레임워크 (ML, Audio, Metal 등등)를 저수준에서 다룰 때에도 이곳에서 Wrapper를 구현한다.

```
class DataRepository: DataRepositoryProtocol {
    func fetchData(completion: @escaping (Result<DataModel, Error>) -> Void) {
        let url = URL(string: "https://example.com/data")!
        URLSession.shared.dataTask(with: url) { data, response, error in
            if let error = error {
                completion(.failure(error))
                return
            }
            guard let data = data,
                  let dataString = String(data: data, encoding: .utf8) else {
                completion(.failure(URLError(.badServerResponse)))
                return
            }
            completion(.success(DataModel(dataString: dataString)))
        }.resume()
    }
}
```

클린 아키텍쳐

각 레이어 간에 의존 관계가 있지만, 의존성 주입과 의존성 역전의 원칙 덕분에 전부
느슨하게 결합되어 있으므로 싹 다 유닛 테스트가 가능하다.

다른 말로 재사용과 유지보수가 쉬워진다.



과제

상세 스펙은 곧 알려드릴게요 😭

- 과제 2, 3에서 구현한 내용을 MVVM 기반으로 리팩토링하기
- ViewController/View → ViewModel → UseCase → Repository 의 형태로 개발
- 의존성 주입과 의존성 역전의 원칙 적용할 것
- 영화 별점 매기기
- 영화 별점은 UserDefaults에 저장
 - (UserDefaults도 로컬 데이터이므로 Repository 레이어에서 구현)

