

iOS 프로그래밍

App Deployment & Code Review



오늘의 학습 목표

- * 배포 과정
- * 사용성 분석과 리포트
- * 인스트루먼트와 메모리 관리
- * 객체지향 설계와 구현
- * 코드리뷰

앱 배포하기



Ad-Hoc



App Store

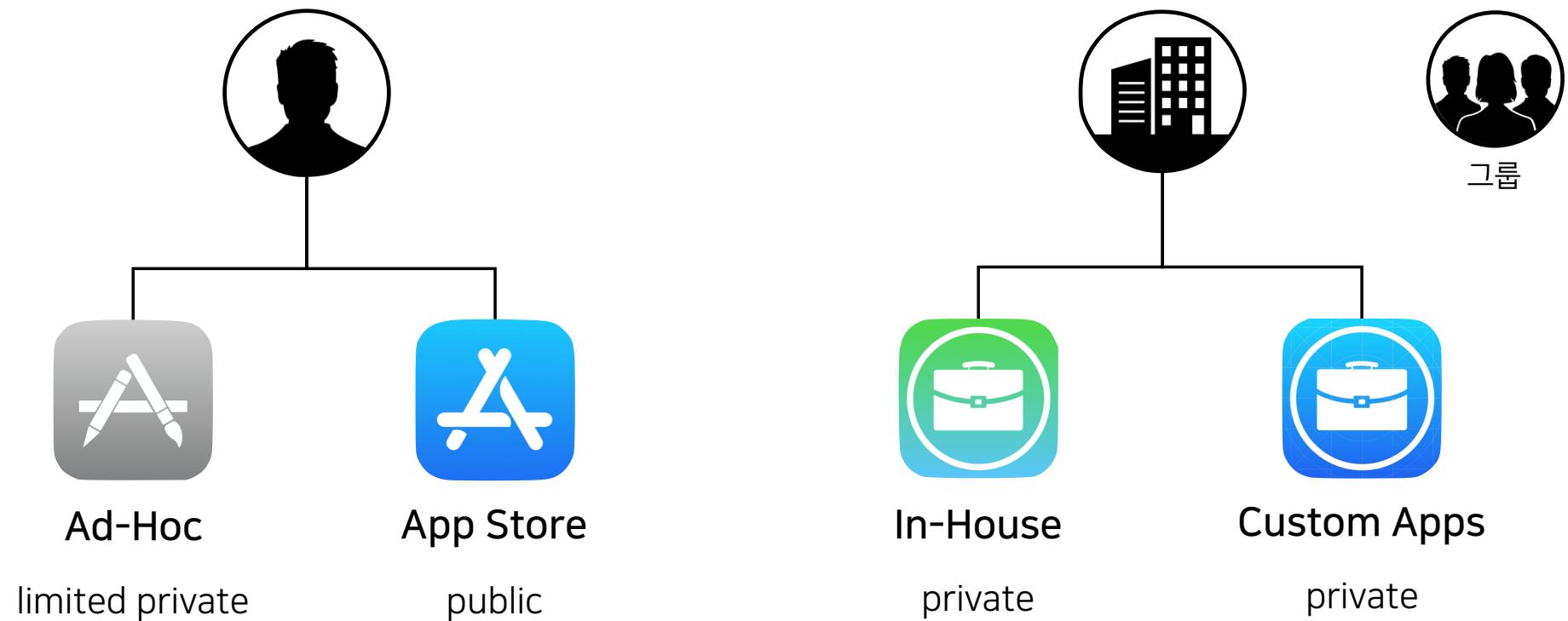


In-House



Custom Apps

앱 배포하기

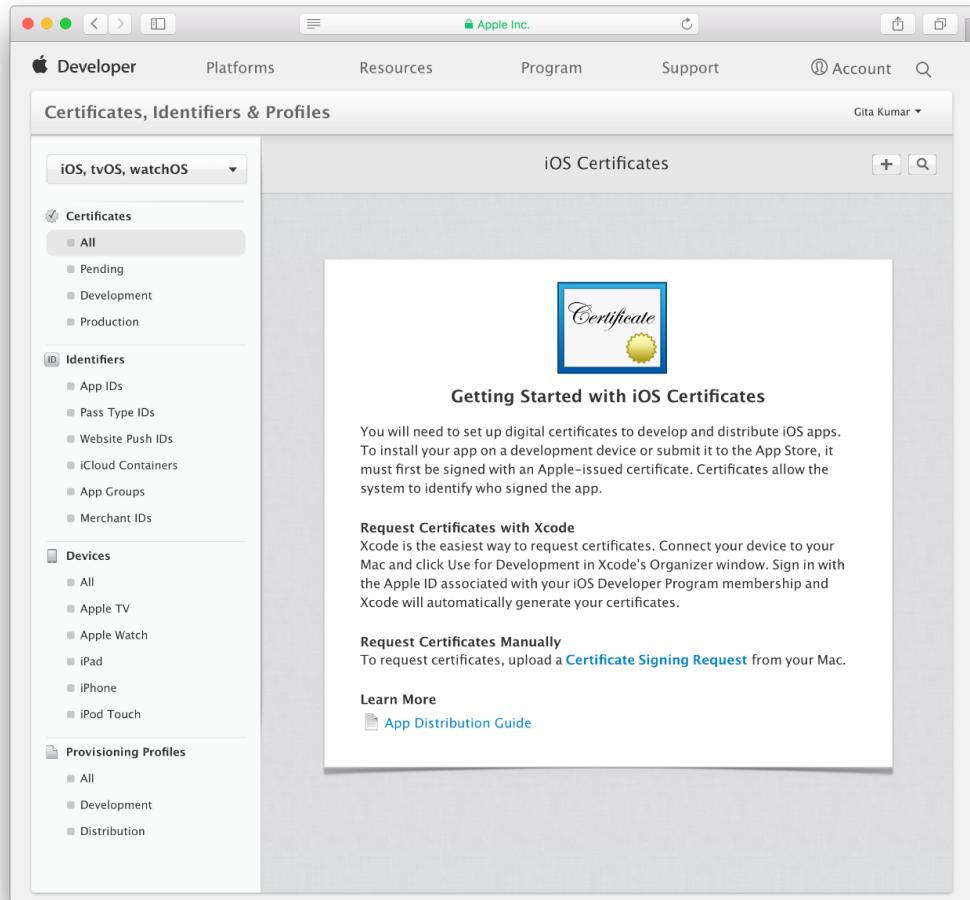


개발자 프로그램

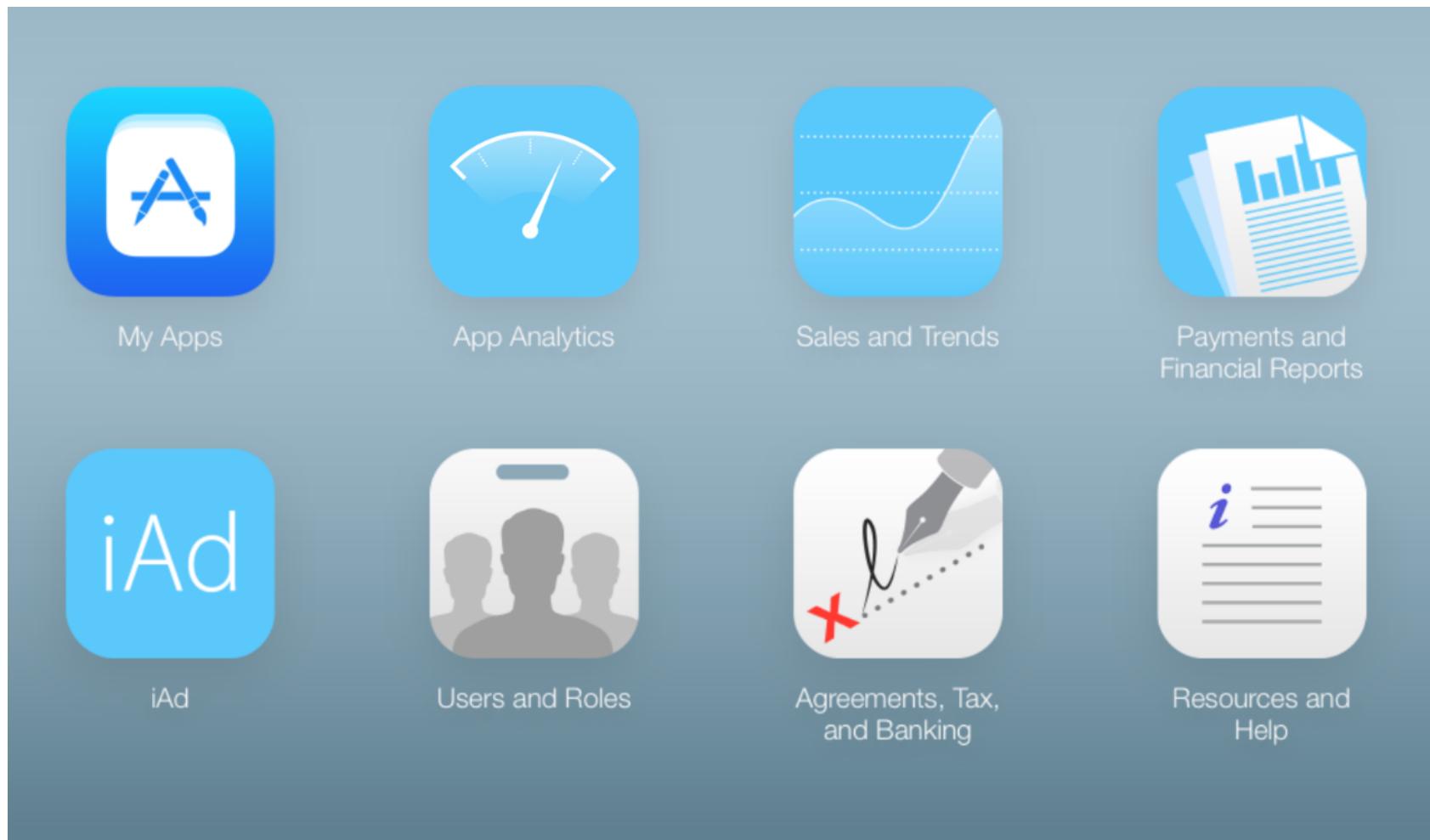
- * 일반
 - * [연회비 \\$99](#)
 - * [앱 스토어 출시 가능](#)
- * 기업 Enterprise
 - * [연회비 \\$299](#)
 - * [앱 스토어 출시 불가](#)
 - * [사내 배포 가능](#)



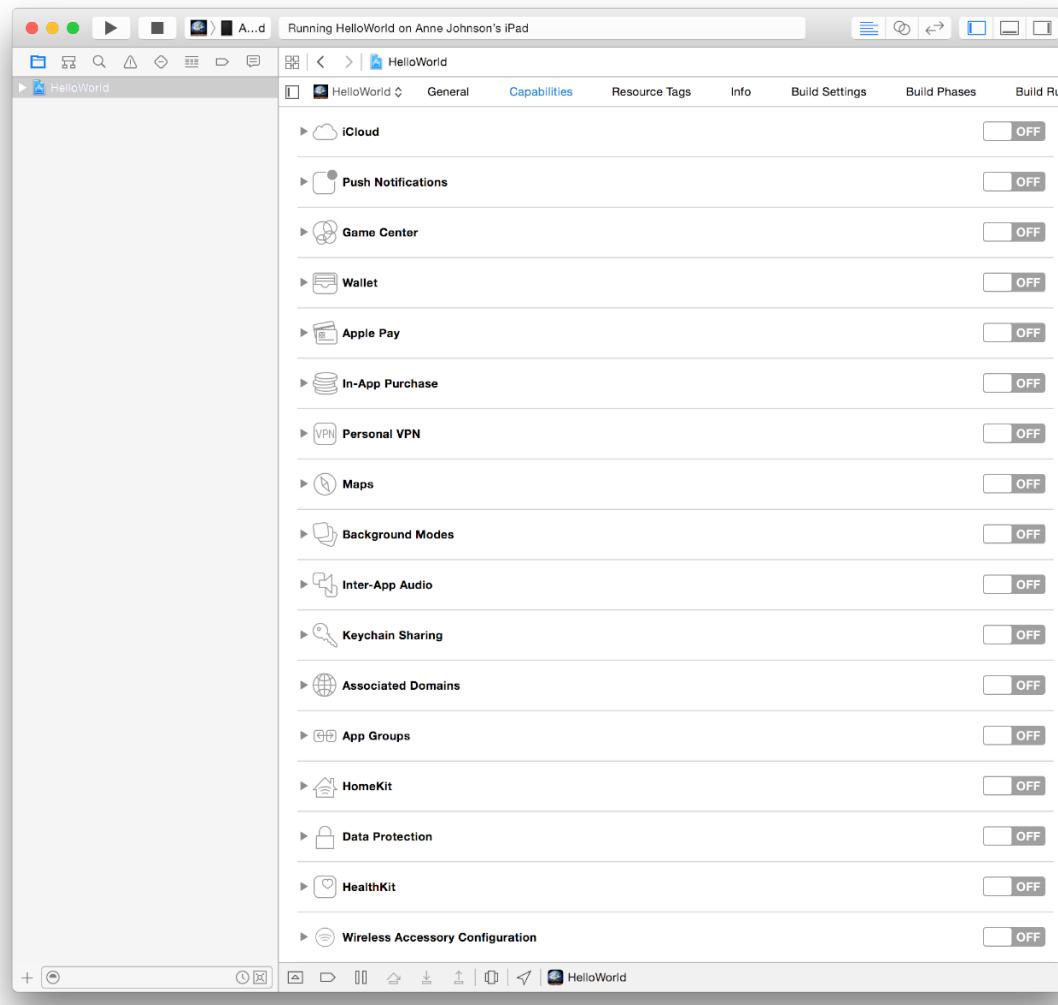
인증서와 프로파일



iTunes Connect



Entitlements



앱 배포 Distribution

- * Run on Device
- * Ad-hoc
- * Enterprise
- * OTA (or Email)
- * MDM

<https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html>

Archive

Product > Archive

The image shows two screenshots of Xcode. The top screenshot displays the 'Archive' interface with the project 'BMFreshApp' selected and a 'Generic iOS Device' target. The bottom screenshot shows the 'BMFreshAppUITests' test file open, displaying Swift code for UI testing setup.

```
override func setUp() {
    super.setUp()

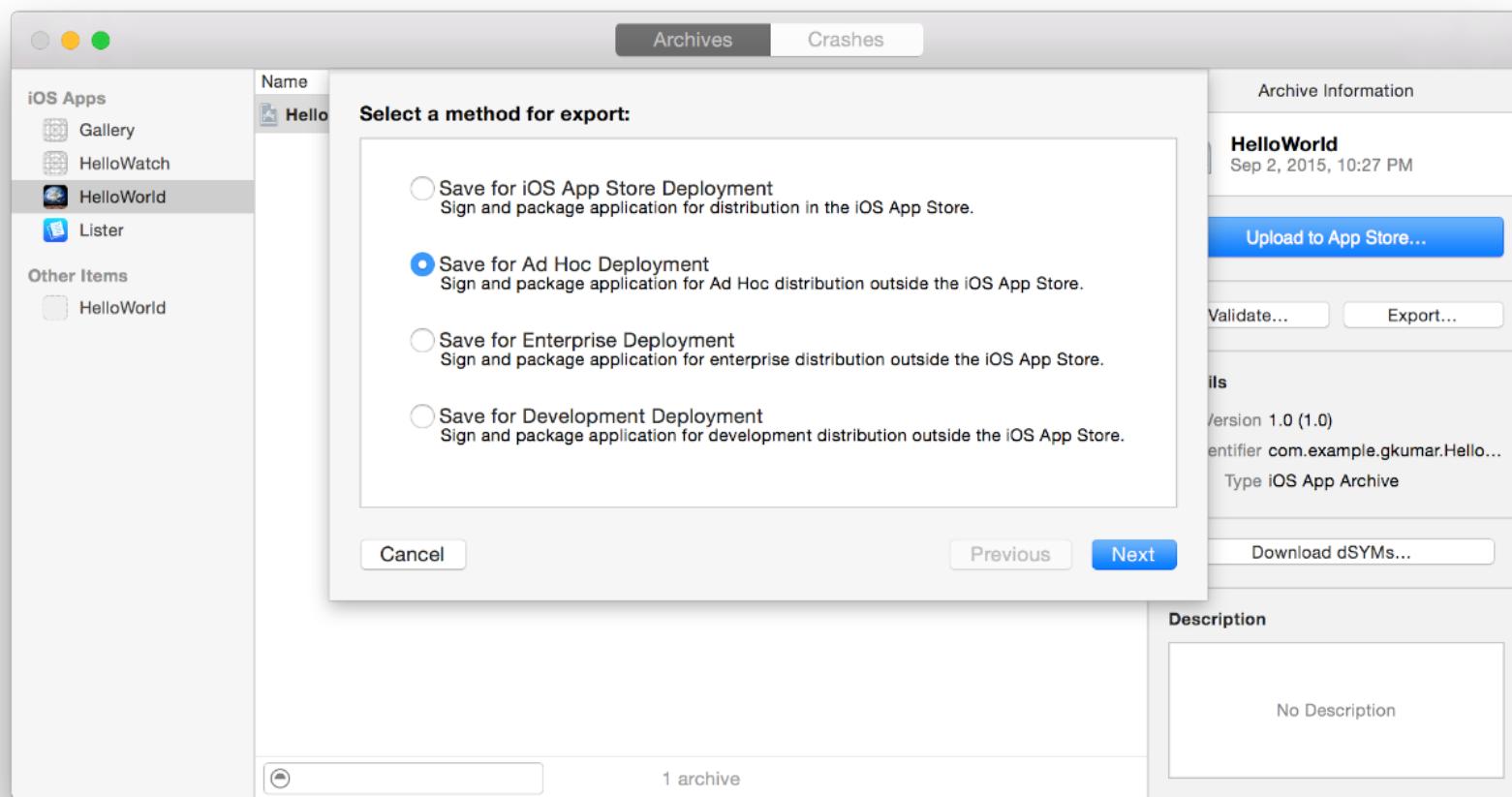
    // Put setup code here. This method is called before each test method.

    // In UI tests it is usually best to stop immediately after failure so as
    // not to hit the timeout which can otherwise occur.
    continueAfterFailure = false

    // UI tests must launch the application that contains the test
    // scene. Once a scene has been presented, the application's
    // implementation of didEnterBackground: will be called.
    XCUIApplication().launch()

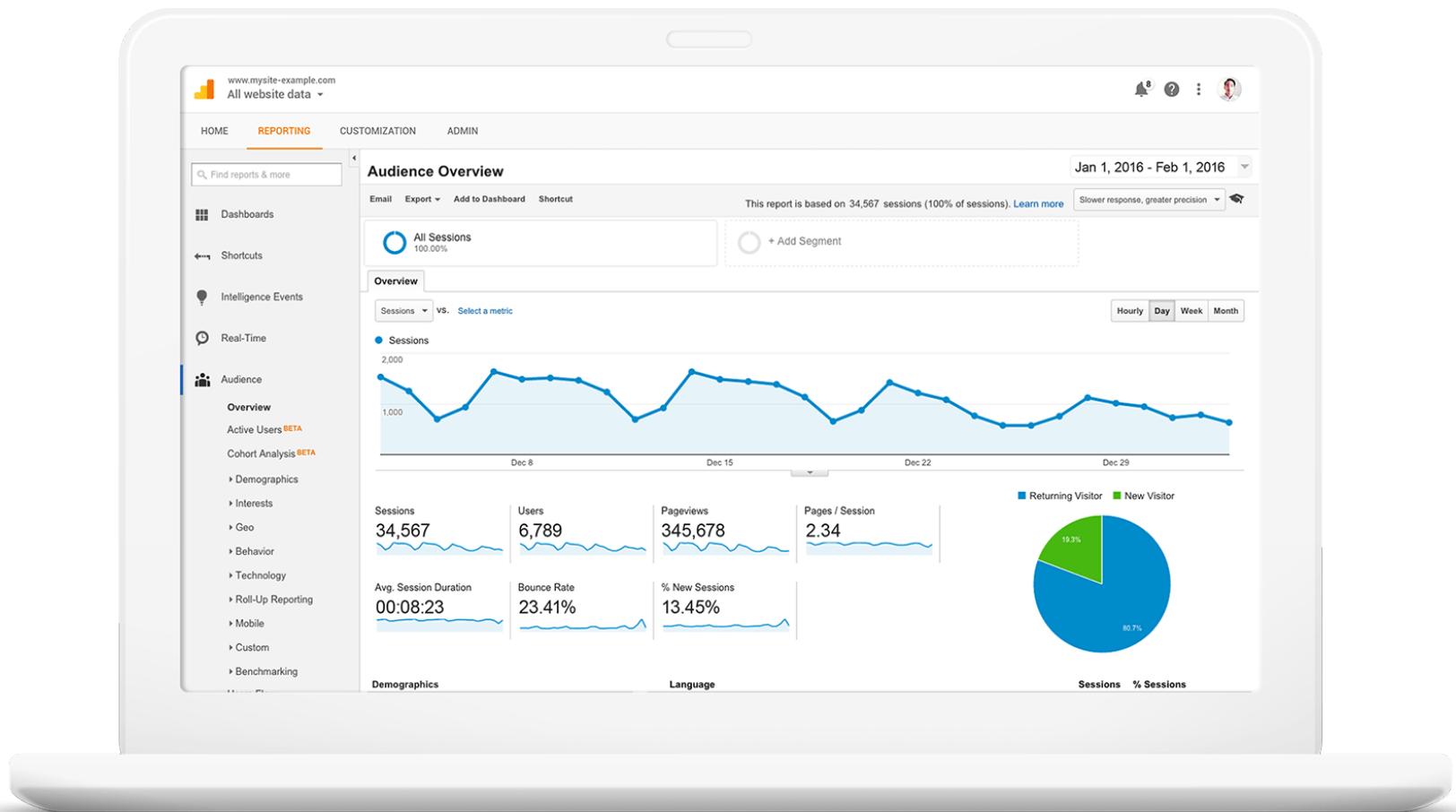
    // In UI tests it's important to set the initial orientation,
    // so that your test is presented correctly.
}
```

Archive Export...



앱 사용성 분석

Google Analytics



Firebase

The screenshot shows the Firebase homepage with a focus on the 'Products' section. The background features a large orange-to-white gradient overlay. The main heading 'Improve app quality' is centered above three product cards: Crashlytics, Performance Monitoring, and Test Lab.

Products Use Cases Pricing Docs Support Search Language Go to console

Improve app quality

Firebase gives you insights into app performance and stability, so you can channel your resources effectively.

 Crashlytics

iOS  

Reduce your troubleshooting time by turning an avalanche of crashes into a manageable list of issues. Get clear, actionable insight into which issues to tackle first by seeing the user impact right in the Crashlytics dashboard. Realtime alerts will help you stay on top of stability even on the go. Crashlytics is the primary crash reporter for Firebase.

[Learn more](#) [Go to docs](#) 

 Performance Monitoring

iOS  

Diagnose app performance issues occurring on your users' devices. Use traces to monitor the performance of specific parts of your app and see a summarized view in the Firebase console. Stay on top of your app's start-up time and monitor HTTP requests without writing any code.

[Learn more](#) [Go to docs](#) 

 Test Lab

 iOS  

Run automatic and customized tests for your app on virtual and physical devices hosted by Google. Use Firebase Test Lab throughout your development lifecycle to discover bugs and inconsistencies so that you can offer up a great experience on a wide variety of devices.

[Learn more](#) [Go to docs](#) 

crashlytics

The screenshot shows the fabric.io dashboard for the Cannonball 2.3.1 app. The top navigation bar includes the fabric.io logo, a search bar, and user profile information for Jeff Seibert from Twitter, Inc.

The main summary card displays the following data:

- 5572 ISSUES
- 191k NON-FATALS
- 324k CRASHES
- 84k USERS AFFECTED
- 159k USERS AFFECTED

A bar chart shows crash counts over time, with a significant spike around September 25th.

The detailed issues list includes the following entries:

ID	File / Line	Description	Version	Crashes	Users
#1004	CannonballService.java line 111	io.fabric.example.cannonball.library.app.CannonballService.onLocationChanged	2.3.1	36k	26k
#2241	GcmPushProvider.java line 36	io.fabric.example.cannonball.client.core.vendor.google.gcm.GcmPushProvider...	2.3.1	20k	20k
#1006	ConnectionFragment.java line 536	io.fabric.example.cannonball.client.feature.connection.ConnectionFragment.o...	2.3.1	15k	7815
#2847	SignInFragment.java line 273	io.fabric.example.cannonball.client.feature.signin.SignInFragment.getGoogleT...	2.3.1	14k	875
#3301	GoogleMapAdapter.java line 97	io.fabric.example.cannonball.library.vendor.google.map.GoogleMapAdapter.set...	2.3.1	13k	2147

At the bottom of the list, there is a link to "ServerClient.java line 1111".

Userhabit

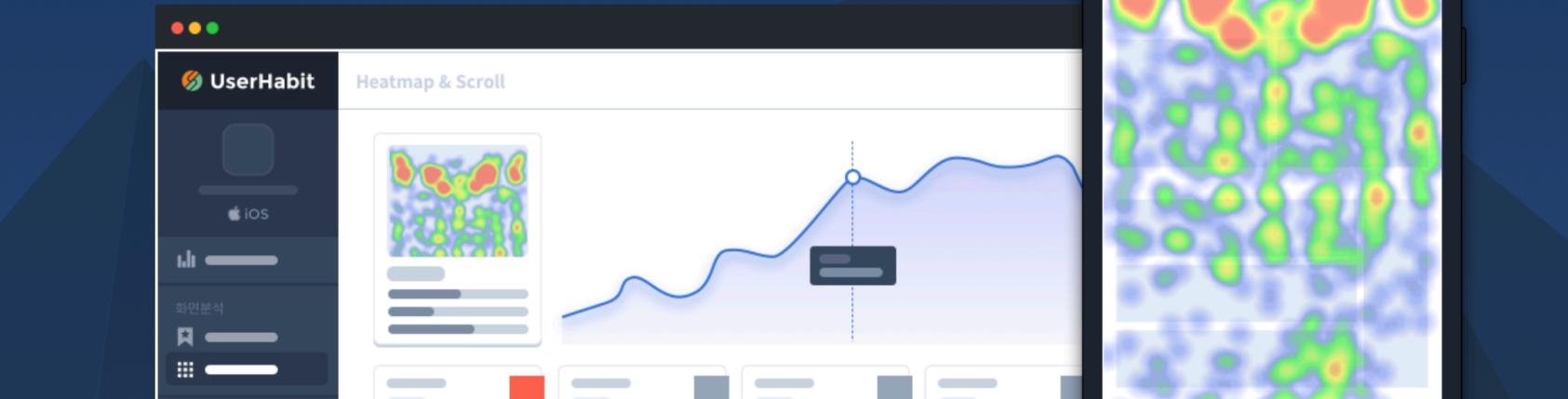
코드 한줄로 사용자의 행동을 보고
앱 성장을 가속화 하세요.

앱 최적화를 위한 UX 기반 모바일 애널리틱스, 유저해빗

사용자 행동을 이해하고 싶다면 지표만으로는 부족합니다.

사용자의 행동패턴을 이해하면 분석의 가치가 커집니다.

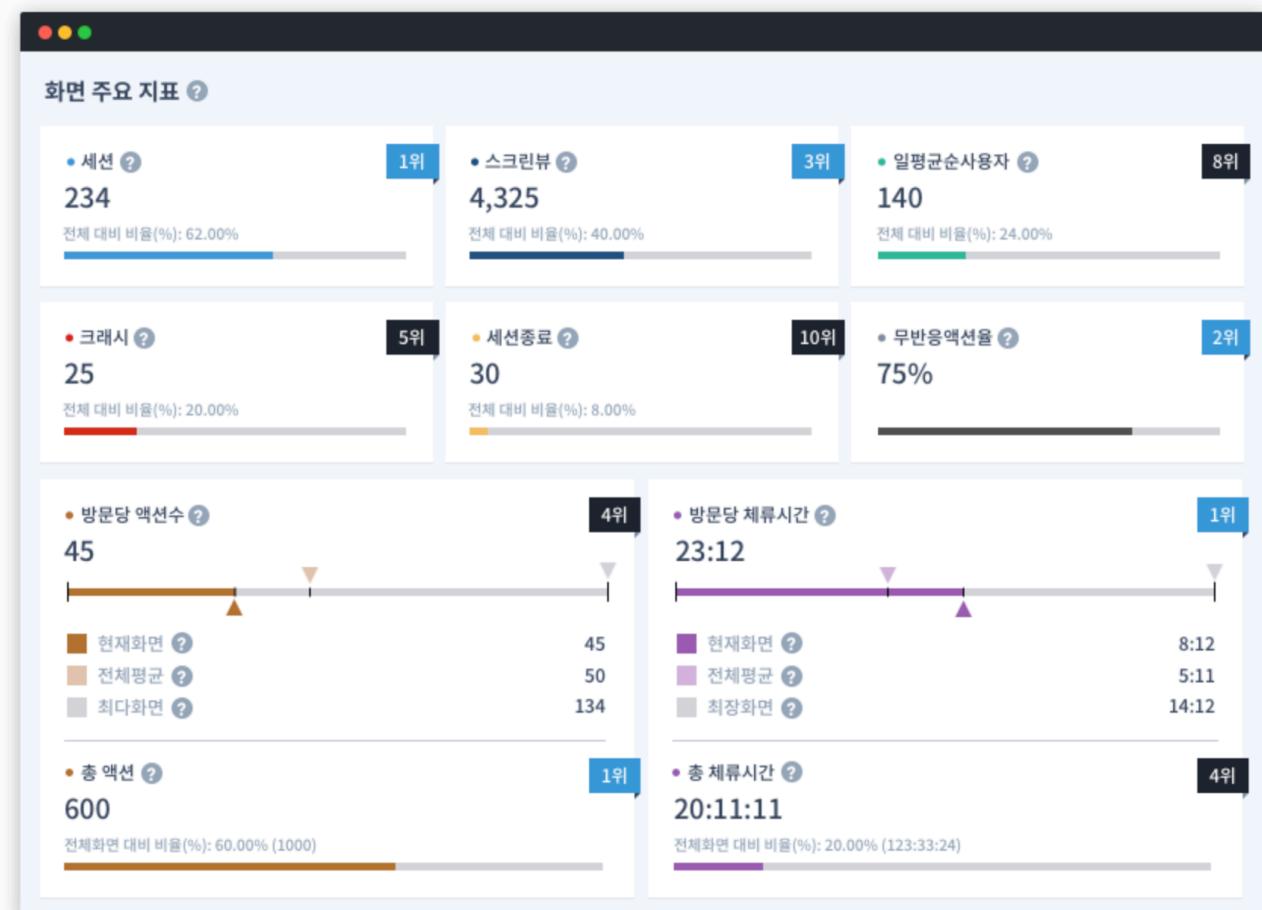
유저해빗 시작하기



UX분석을 위한 주요지표

앱 서비스가 사용자에게 얼마나 매력적인지, 얼마나 자주 사용하는지, 충분히 잘 사용하고 있는지를 알기 위해 사용자의 활동성을 파악할 수 있는 여러가지 종합적인 지표들을 제공합니다.

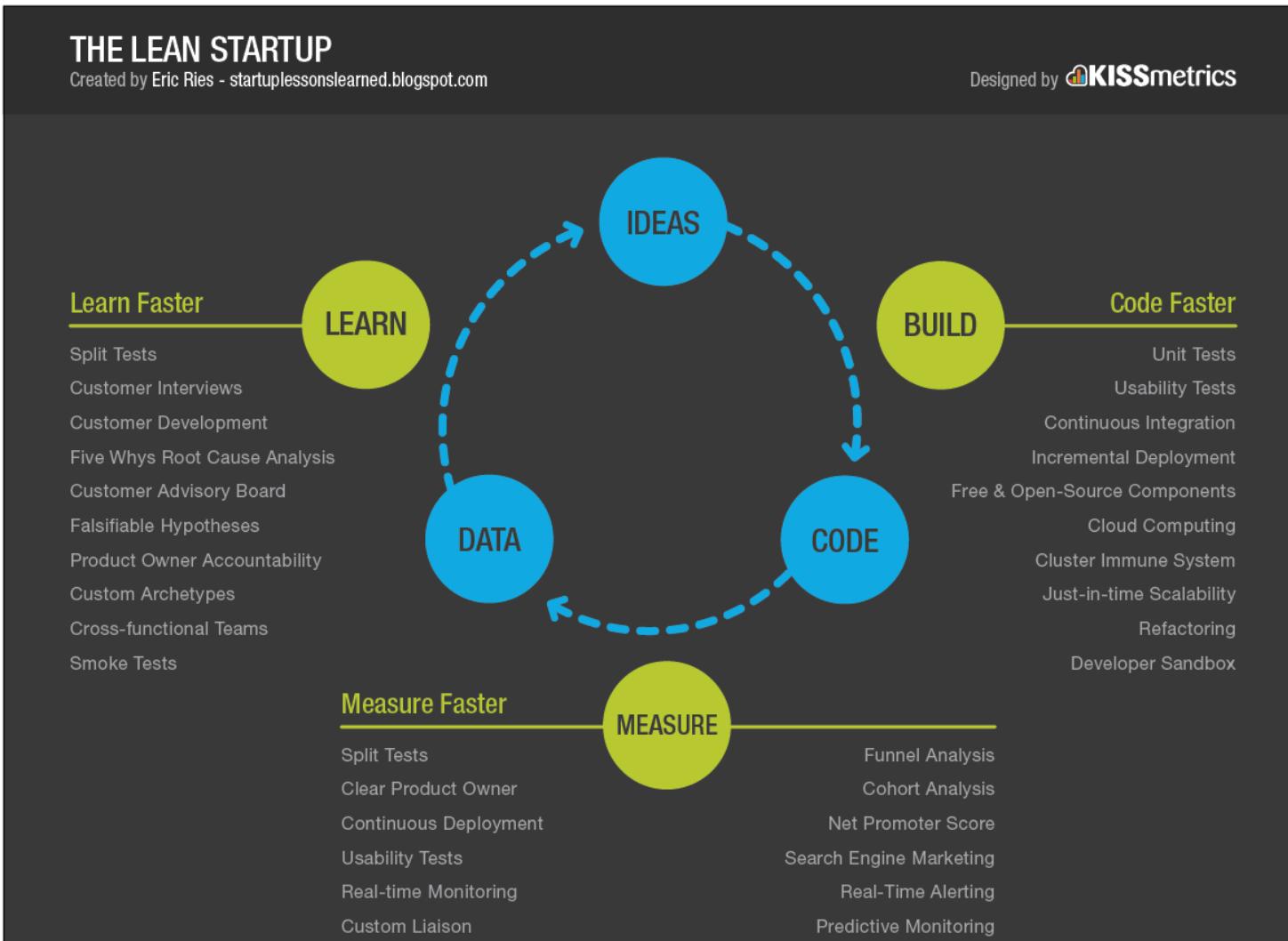
서비스의 규모를 파악하는 일반적인 지표를 포함해 화면분석을 위한 8가지 지표 등을 제공하며 사용자의 활동이 긍정적인지 부정적인지 파악할 수 있습니다.



HCI 사용성 원칙

- * 접근성 Accessibility
- * 일관성 Consistency
- * 직접성 Directness
- * 효율성 Efficiency
- * 친밀성 Familiarity
- * 피드백 Feedback
- * 유연성 Flexibility
- * 오류수용성 Forgiveness
- * 정보제공성 Informativeness
- * 예측성 Predictability
- * 단순성 Simplicity
- * 사용자 조작 User Control
- * 가시성 Visibility

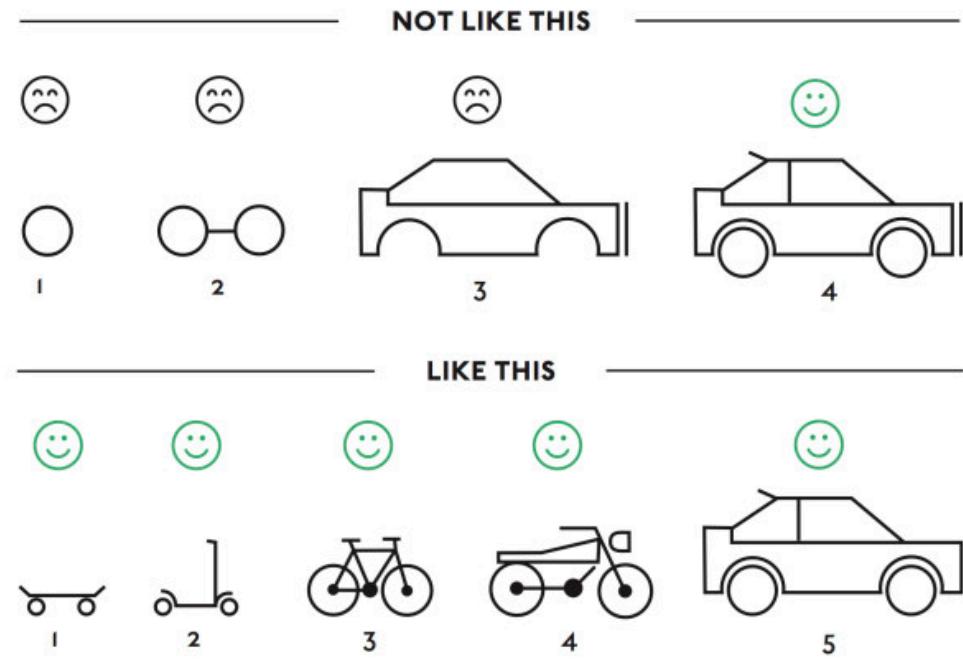
Lean Startup



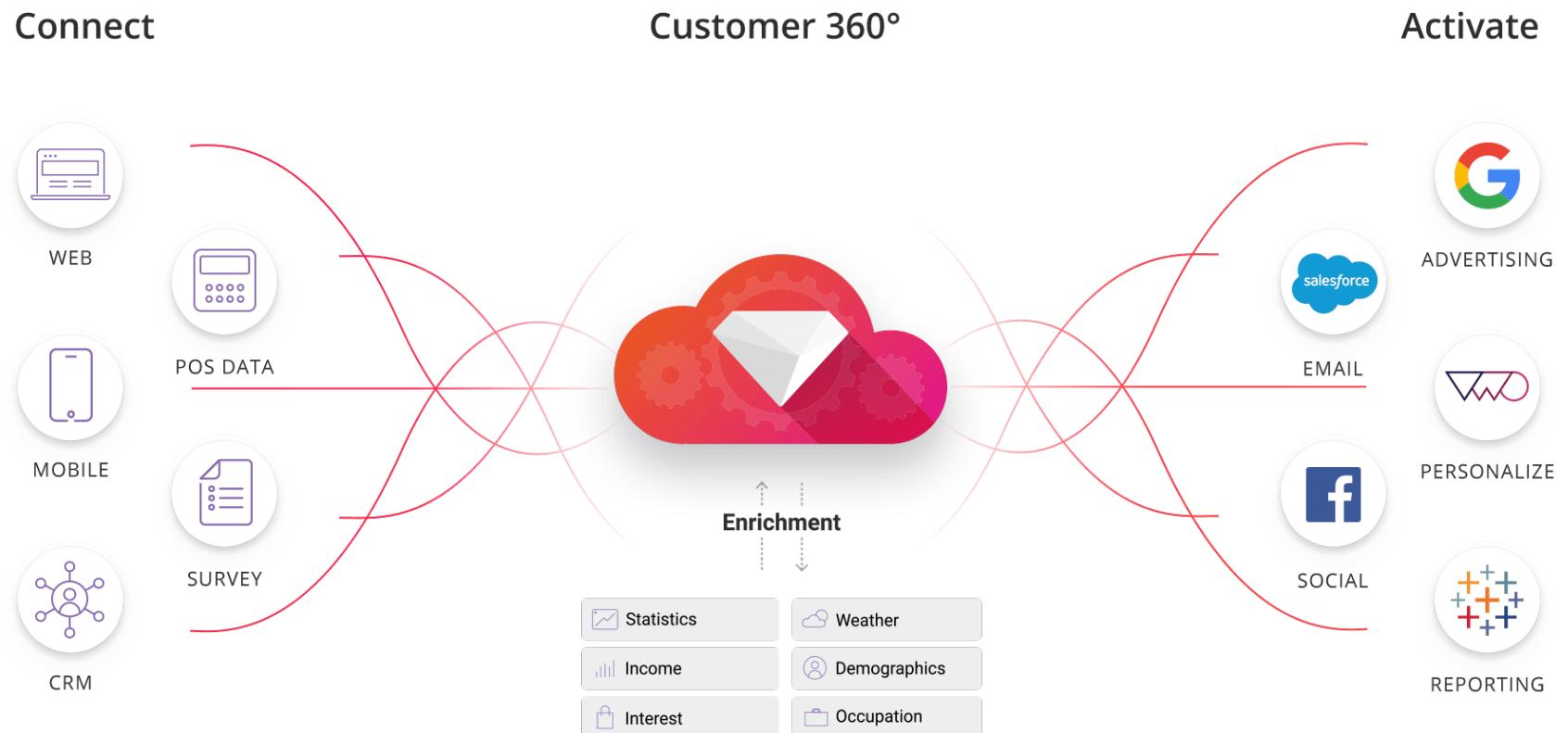
MVP

HOW TO BUILD A

MINIMUM VIABLE PRODUCT (MVP)



App Marketing



고객 개발

사용자 유치 (Acquisition)

사용자가 어떻게 제품을 접하는가?

사용자 활성화 (Activation)

사용자가 처음으로 제품을 사용했을 때
경험이 좋았는가?

사용자 유지 (Retention)

사용자가 제품을 계속 사용하는가?

매출 (Revenue)

어떻게 돈을 버는가?

추천 (Referral)

다른 사람에게 제품을 소개하는가?

Manual Reference Count



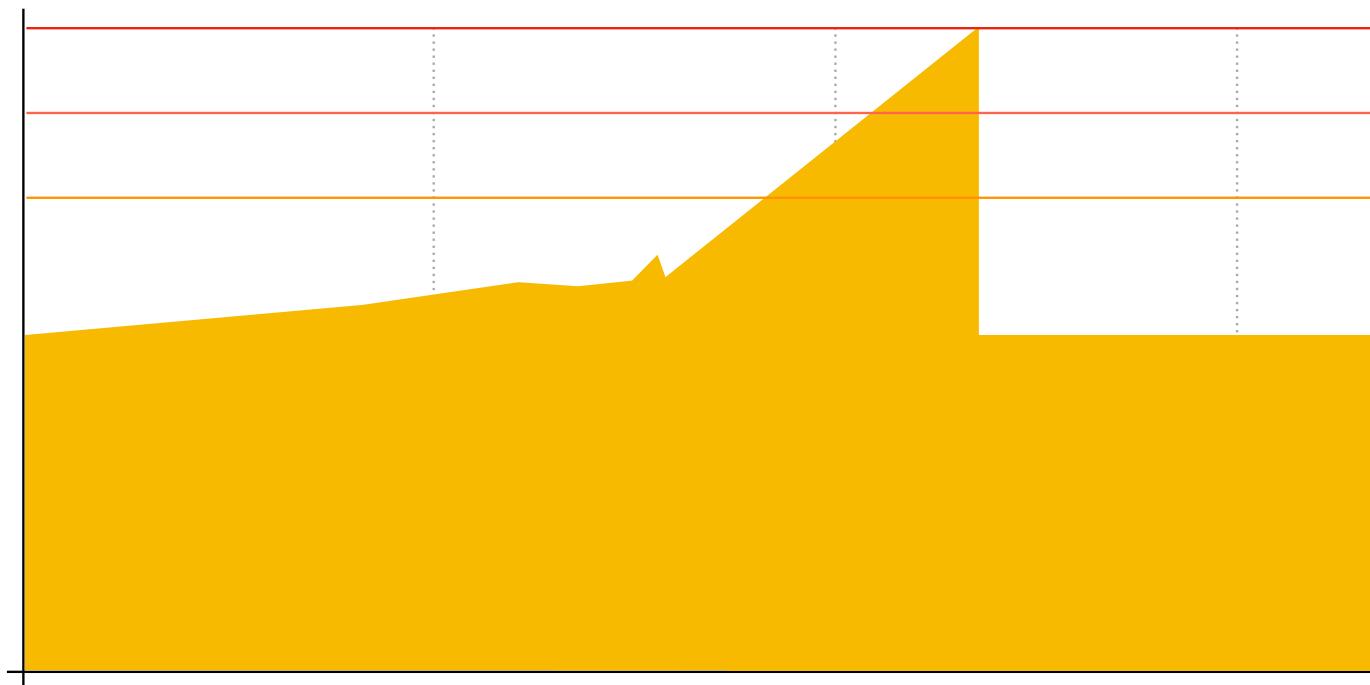
iOS Memory



iOS Memory



iOS Memory



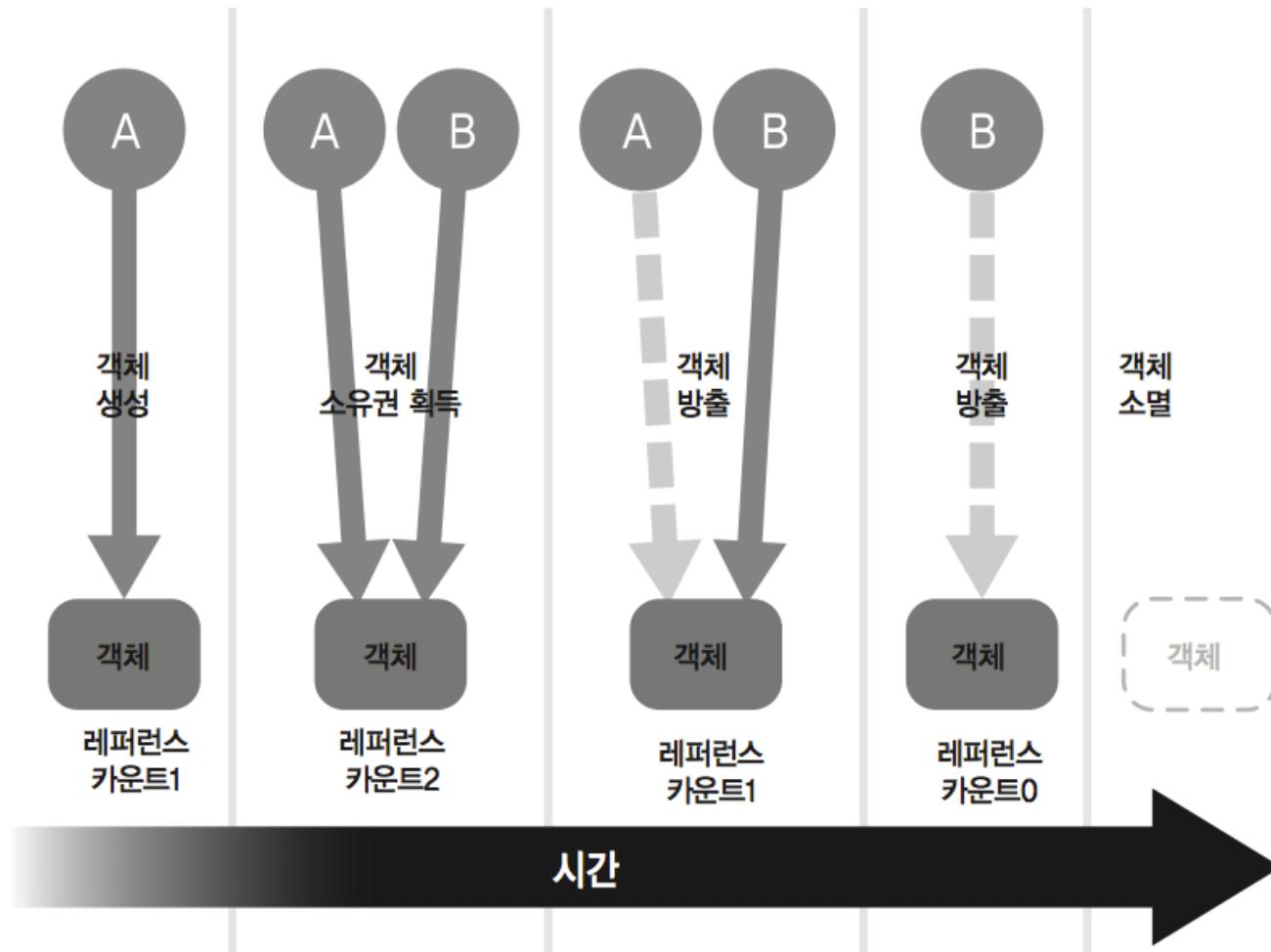


그림 1-4. 레퍼런스 카운팅을 이용한 메모리 관리

수동 메모리 관리

만약 `+alloc`, `+new`, `+copy` / `-retain` 을 했으면 `-release`

`-retain` 메서드를 사용했다는 것은

다 사용한 이후에 `-release`를 호출하겠다는 것

`release` 메시지를 보내더라도 모두 메모리 해제되는 것은 아님. retain count가 0이 될 때만 소멸한다. (`-dealloc`호출)

소유권을 자동으로 관리할 때 `autorelease` pool

`autorelease` 횟수만큼 `release` 메시지를 날린다.

하지만 iOS에서는 `-release`가 더 효과적임

앱이 종료되면 모든 객체는 릴리스된다.

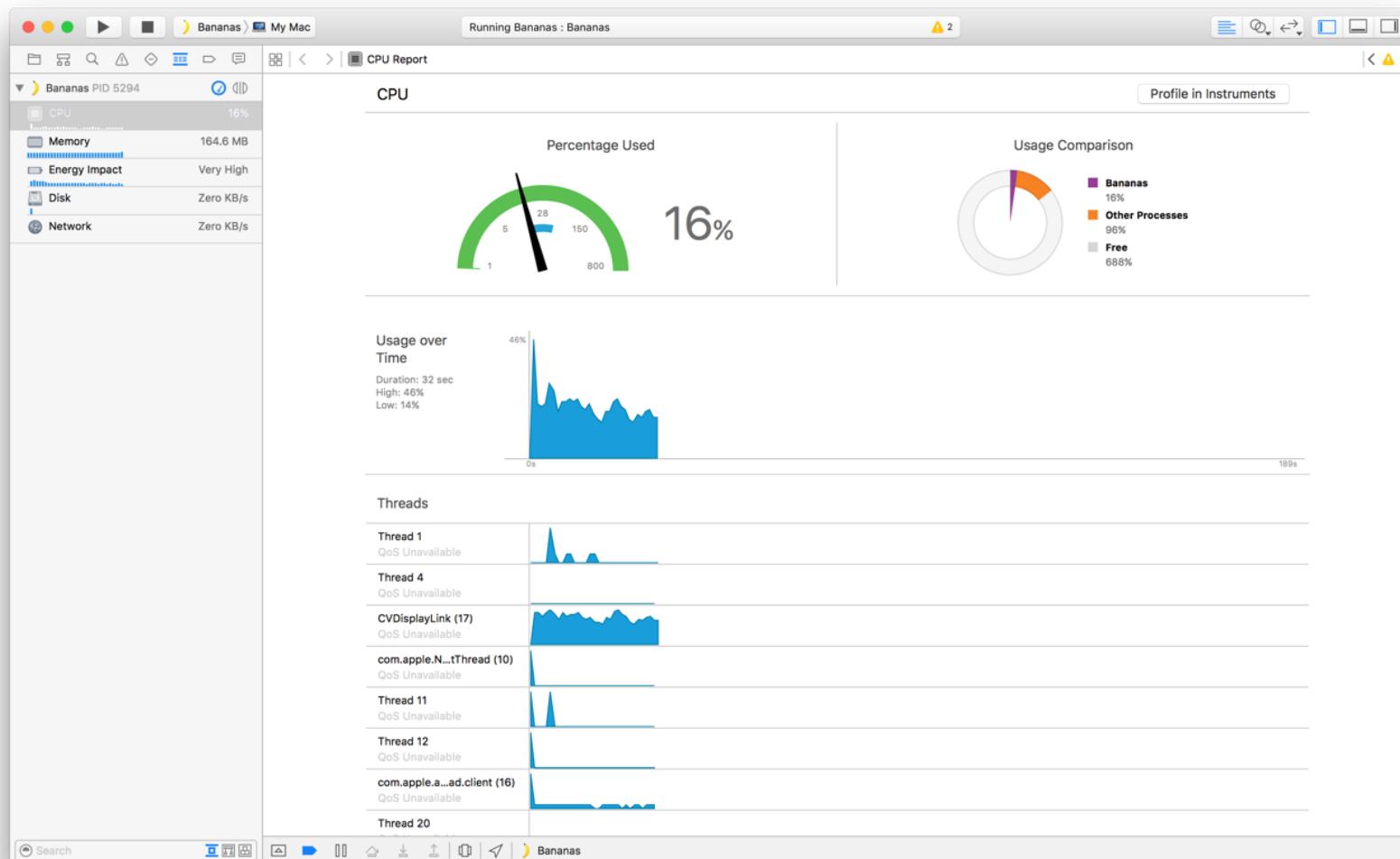
Instruments

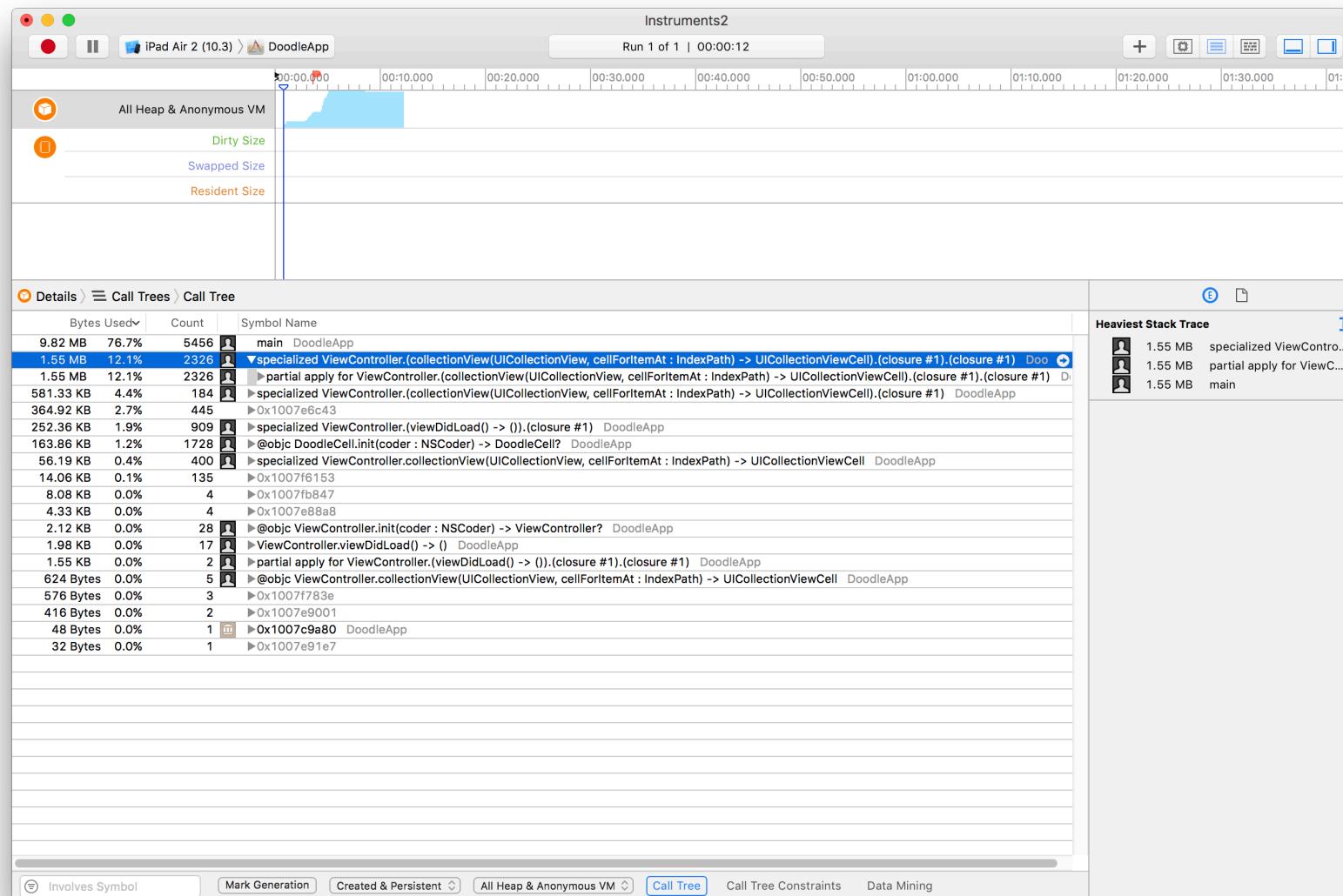


InstXrayents



Debug Gauge

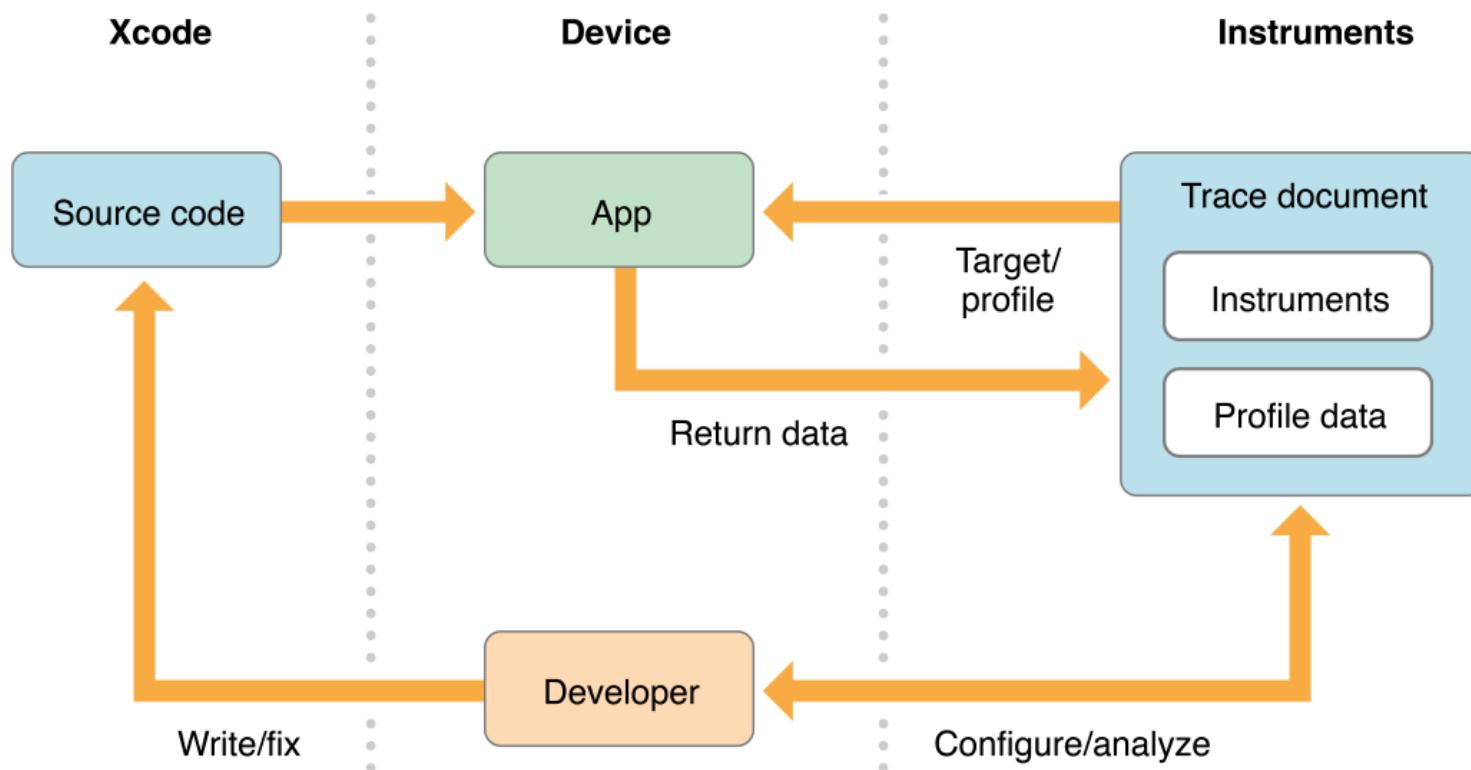




인스트루먼트

- * Examine the behavior of one or more apps or processes
- * Examine device-specific features, such as Wi-Fi and Bluetooth
- * Perform profiling in a simulator or on a physical device
- * Create custom DTrace instruments to analyze aspects of system and app behavior
- * Track down problems in your source code
- * Conduct performance analysis on your app
- * Find memory problems in your app, such as leaks, abandoned memory, and zombies
- * Identify ways to optimize your app for greater power efficiency
- * Perform general system-level troubleshooting
- * Save instrument configurations as templates

Workflow



Automatic Reference Count



strong 변수

모든 객체 포인터 변수는 strong 변수임 (기본값)

자동으로 retain 처리됨

이전 객체는 대입할 때 릴리스된다.

strong 변수는 초기값을 0으로 설정한다.

MRC : f2 = f1;

ARC : [f1 retain]; [f2 release]; f2 = f1;

property 속성 strong / __strong

강한 참조 기본값

```
class Robot : NSObject {
    var name : String
    var nemesis : Robot?
    var model : Int

    override init() {
        name = ""
        nemesis = nil
        model = 0
    }

    deinit {
        print("robot was deinit")
    }
}
```

```
var robot1 : Robot?
robot1 = Robot()
var robot2 : Robot?
var robot3 : Robot?
robot2 = robot1
robot3 = robot1

//...

robot2 = nil
robot3 = nil
var workArray = [robot1]
robot1 = nil
workArray.removeAll()
```

강한 참조 순환 문제

```
var robot1 : Robot? = Robot()
var robot2 : Robot? = Robot()

robot1?.nemesis = robot2
robot2?.nemesis = robot1

robot1 = nil
robot2 = nil
```

둘 다 참조 카운트는 1이라서 메모리 해제되지 않음!

약한 참조 weak

```
class Robot : NSObject {
    var name : String
    weak var nemesis : Robot?
    var model : Int

    override init() {
        name = ""
        nemesis = nil
        model = 0
    }

    deinit {
        print("robot-\(name) was deinit")
    }
}
```

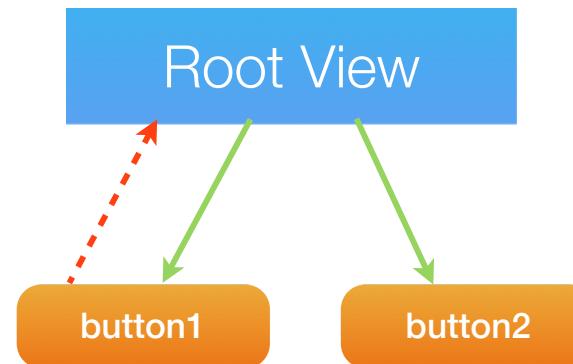
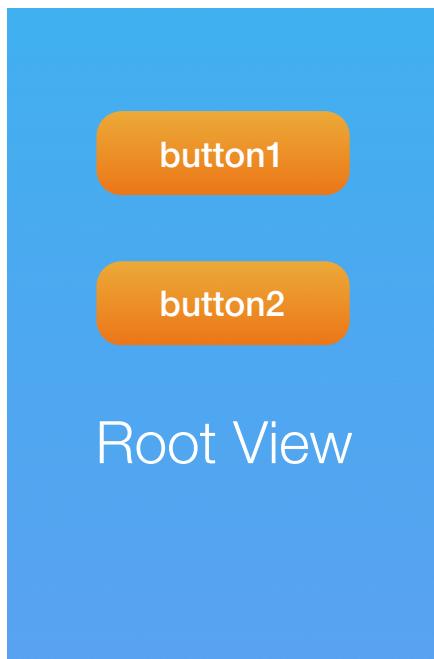
```
var robot1 : Robot? = Robot()
var robot2 : Robot? = Robot()

robot1?.name = "thomas"
robot1?.nemesis = robot2
robot2?.name = "9j"
robot2?.nemesis = robot1

robot1 = nil
robot2 = nil
//robot-thomas was deinit
//robot-9j was deinit
```

weak 키워드

- View 소유 관계 (Parent View own Child views)
- 객체가 사라지면 약한참조는 nil로 바뀜



unowned 미소유참조

unowned 키워드를 사용하면 참조 카운터를 반영하지 않음

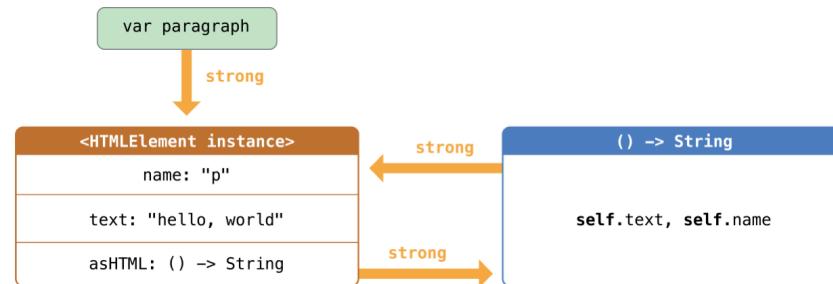
객체가 사라져도 nil로 바뀌지 않고, 항상 값이 있다고 가정

옵셔널 타입은 안되고 class 또는 class-protocol만 가능

클로저 강한 참조 순환

https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html#/apple_ref/doc/uid/TP40014097-CH20-ID5

```
class HTMLElement {  
  
    let name: String  
    let text: String?  
  
    lazy var asHTML: () -> String = {  
        if let text = self.text {  
            return "<\(self.name)>\\(text)</\\(self.name)>"  
        } else {  
            return "<\(self.name) />"  
        }  
    }  
  
    init(name: String, text: String? = nil) {  
        self.name = name  
        self.text = text  
    }  
  
    deinit {  
        print("\(name) is being deinitialized")  
    }  
  
}  
  
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")  
print(paragraph!.asHTML())  
// Prints "<p>hello, world</p>"
```



클로저 강한 참조 순환

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!]
        (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

```
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```

unowned 미소유참조

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized") }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being deinitialized") }
}

var john : Customer? = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
john = nil
```

@autoreleasepool 블록

<Objective-C>

```
@autoreleasepool {  
    ...  
}
```

<Swift>

```
autoreleasepool { () -> () in  
}
```

객체지향 설계와 구현

객체 설계 원칙 - SOLID

1. SRP (Single-Responsibility Principle)

소프트웨어 요소(클래스, 모듈, 함수 등)는 응집도 있는 하나의 책임을 갖는다. 클래스를 변경해야 하는 이유는 단지 - 응집도여야 한다.

2. OCP (Open-Close Principle)

소프트웨어 요소는 확장 가능하도록 열려있고, 변경에는 닫혀있어야 한다. 새 기능을 추가할 때 변경하지 말고 새 클래스나 함수를 만들어서 변하는 부분을 최소화한다.

3. LSP (Liskov Substitution Principle)

서브타입은 (상속받은) 기본 타입으로 대체가능해야 한다. 자식 클래스는 부모 클래스 동작(의미)를 바꾸지 않는다.

4. DIP (Dependency-Inversion Principle)

상위레벨 모듈은 하위레벨 모듈에 의존하면 안된다. (둘 다 추상화된 인터페이스에 의존해야 한다)

추상화는 구체화에 의존하면 안되고, 구체화는 추상화에 의존해야 한다.

5. ISP (Interface-Segregation Principle)

클라이언트 객체는 사용하지 않는 메소드에 의존하면 안된다.

단일 책임 원칙 - 함수

```
struct InputView {  
    func readInput() {  
        print("실행 좌표를 입력하세요.")  
        let userInput = readLine()  
        guard let input = userInput else { return }  
        print(seperateCoordinates(userInput: input))  
    }  
}
```

//... 생략



```
struct InputView {  
    func readLine(prompt: String) -> String {  
        print(prompt)  
        let line = readLine()  
        guard let input = line else { return "" }  
        return input  
    }  
}
```

//... 생략

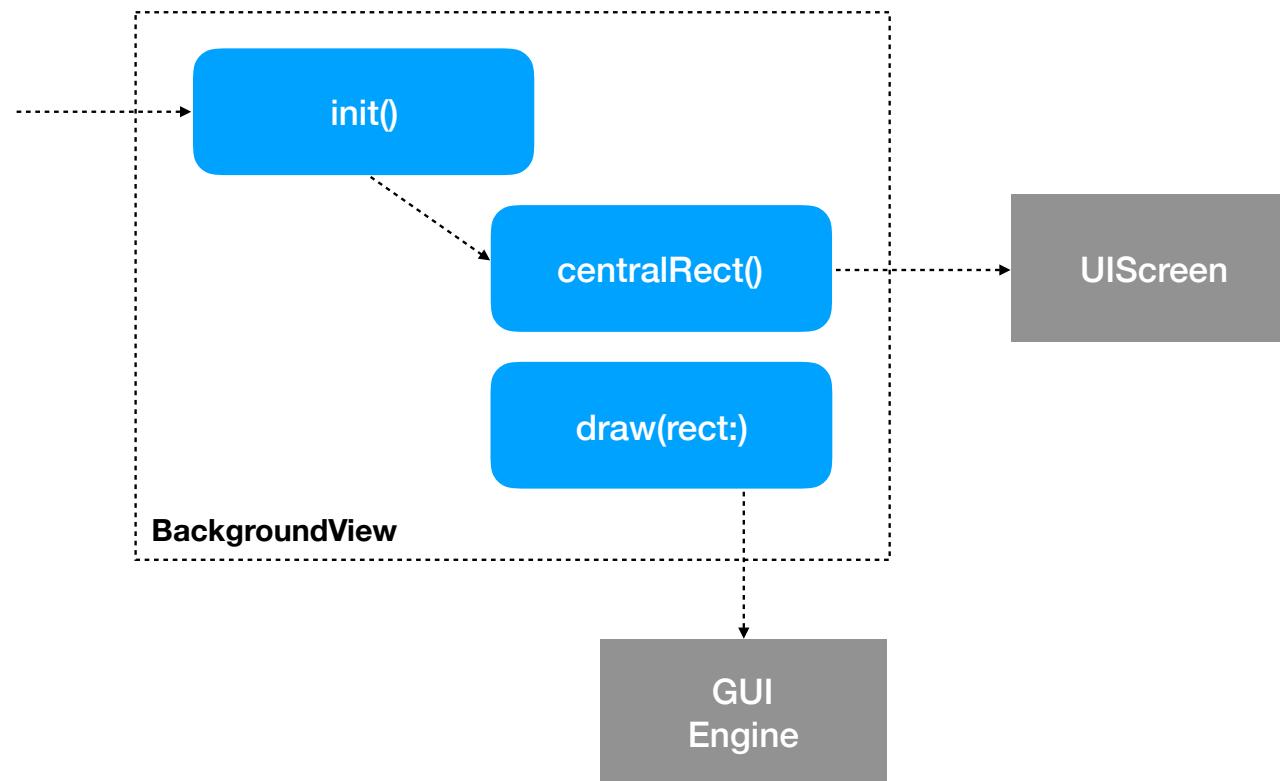
단일 책임 원칙 - 객체

```
class BackgroundView: UIView {
    private let Steps = 8
    //생략
    convenience init() { self.init(frame: ChessBackgroundView.centralRect()) }

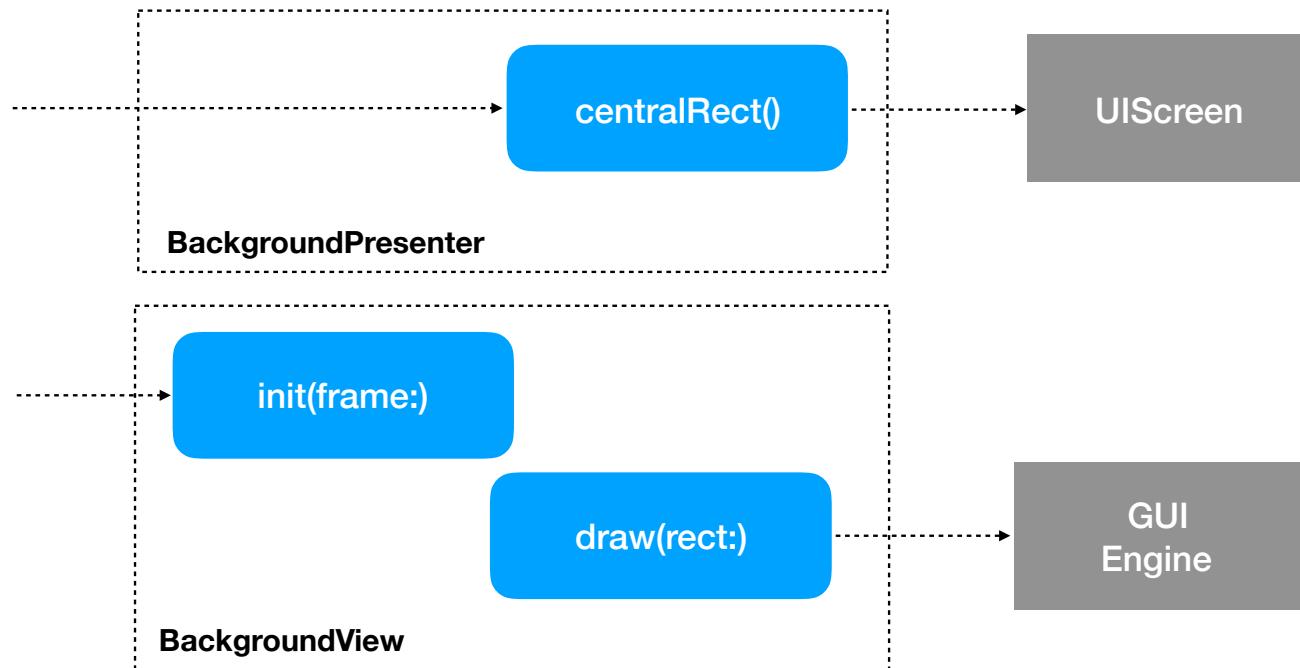
    private static func centralRect() -> CGRect {
        let screenRect = UIScreen.main.bounds
        let verticalMargin = (screenRect.height - screenRect.width) / 2
        return CGRect(x: 0, y: verticalMargin,
                      width: screenRect.width, height: screenRect.width)
    }

    override func draw(_ rect: CGRect) {
        let width = frame.width / CGFloat(Steps)
        let height = frame.height / CGFloat(Steps)
        var binaryFlag = false
        for y in 0..
```

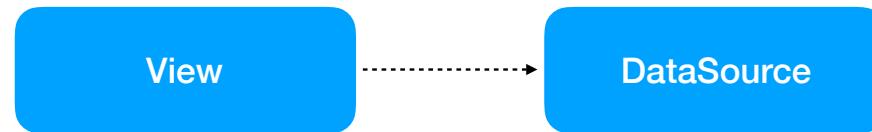
Code 설계



Code 설계와 응집도

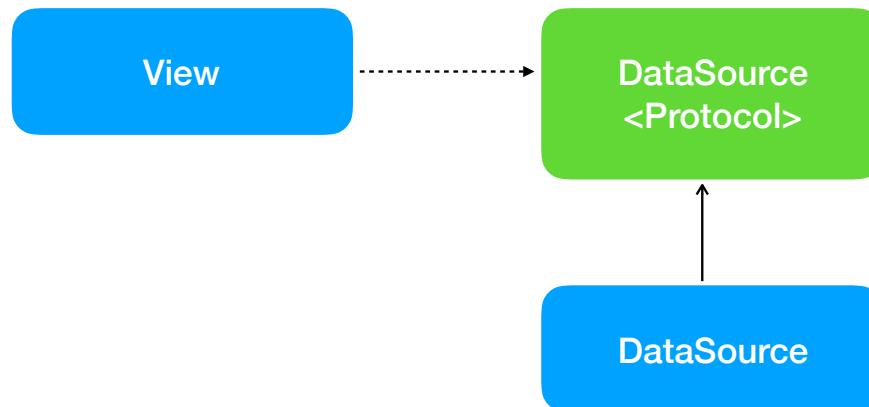


열림-닫힘 원칙 - OCP



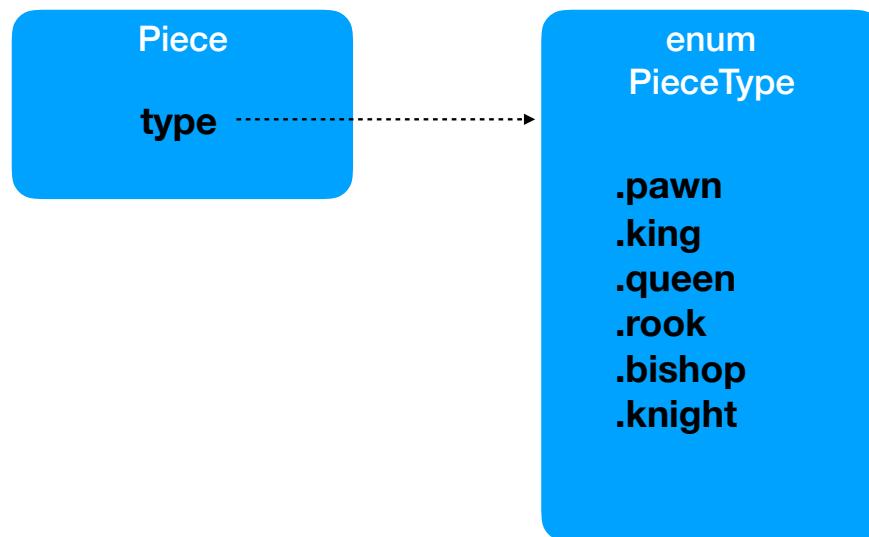
핵심 접근 방식은 추상화

열림-닫힘 원칙 - OCP

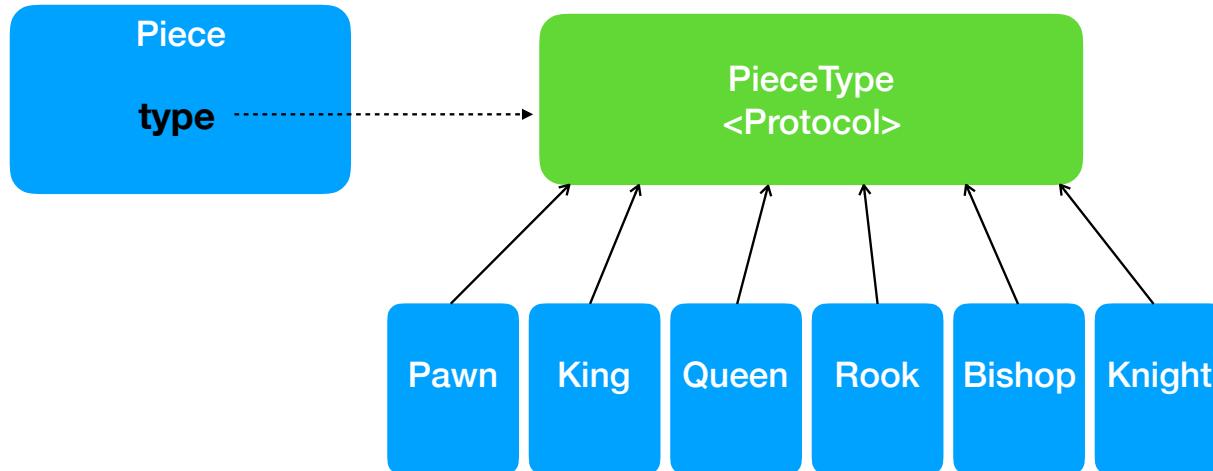


Protocol-Oriented Programming

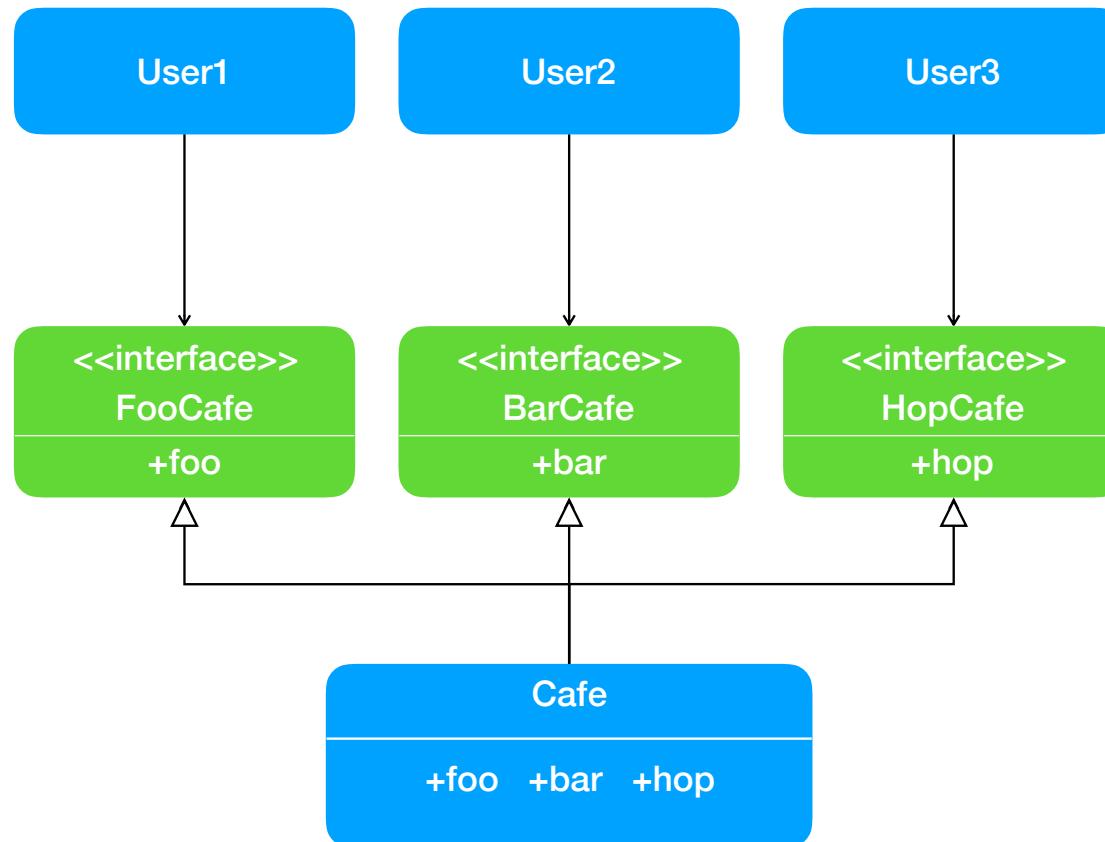
Code 설계



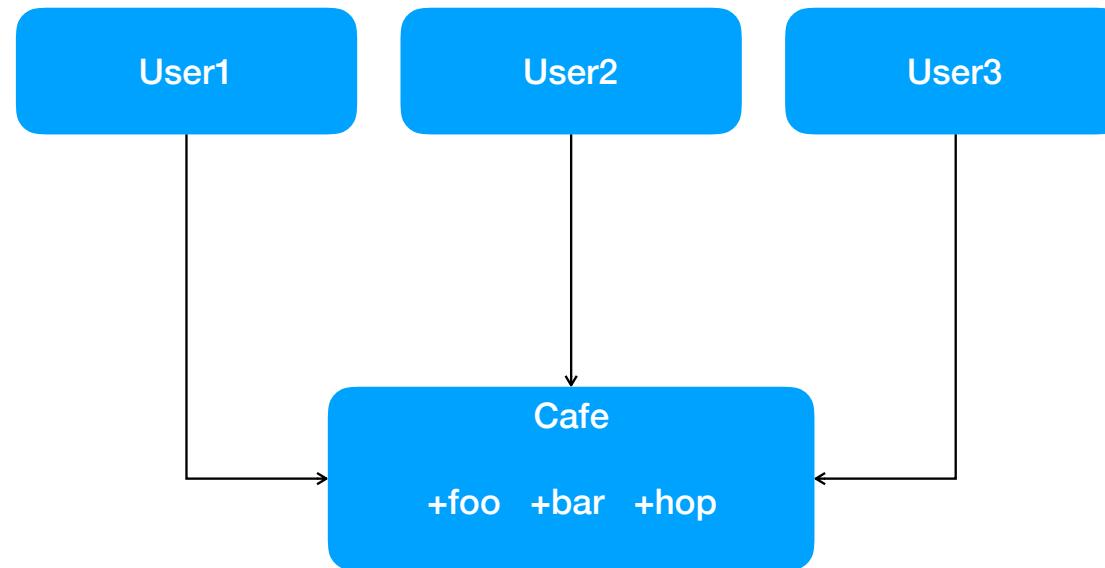
주상 탑입 - OCP, LSP



인터페이스분리 원칙 - ISP



인터페이스분리 원칙 - ISP



인터페이스분리 원칙 - ISP

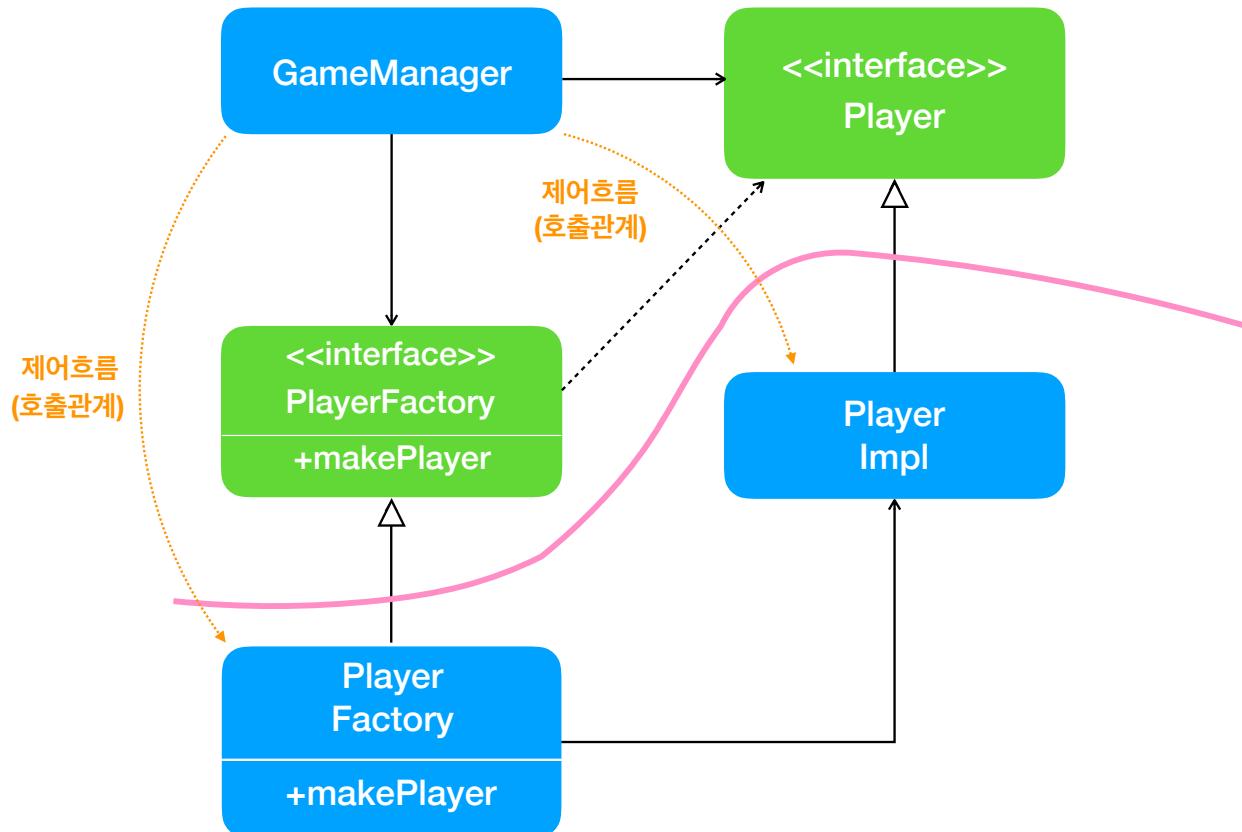
```
protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
    func defaultFrame(path:(x : Int, y: Int)) -> CGRect
}

protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
}
protocol GameManagerGeometric {
    func defaultFrame(for: (x : Int, y: Int)) -> CGRect
}

class ViewController: UIViewController {
    private var positionManager :
        (GameManagerAction & GameManagerGeometric)!

    func setManager(_ manager :
        GameManagerAction & GameManagerGeometric) {
        self.positionManager = manager
    }
}
```

의존성역전 원칙 - DIP



의존성역전 원칙 - DIP

Constructor Injection

```
class BackgroundView: UIView {
    private var presenter : BackgroundPresenter!
    convenience init(presenter: BackgroundPresenter) {
        self.init(frame: presenter.centralRect())
        self.presenter = presenter
    }
}
```

Setter Injection

```
protocol GameManagerAction {
    func positionMap() -> Array<Array<Piece?>>
    func defaultFrame(path:(x : Int, y: Int)) -> CGRect
}
class ViewController: UIViewController {
    private var manager : GameManagerAction!
    func setManager(_ manager : GameManagerAction) {
        self.positionManager = manager
    }
}
```

Interface Injection

```
protocol PieceImageSetter {
    func setImageBy(piece: Piece)
}
class PieceImageView: UIImageView, PieceImageSetter {

    func setImageBy(piece: Piece) {
        self.image = piece.type.image(color: piece.color.rawValue)
    }
}
```

객체지향의 사실과 오해

역할, 책임, 협력 관점에서 본 객체지향
The Essence of Object-Orientation
Roles, Responsibilities, and Collaborations .023 .024



(,,)

// Leaders .023

//

오브젝트

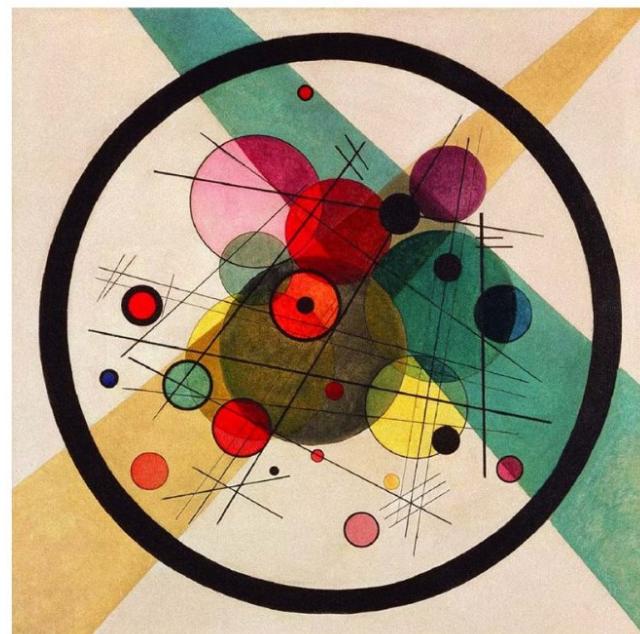
OBJECTS

코드로 이해하는
객체지향 설계

조영호 지음

IT Leaders .023

//
우리의 스



(,,)

책임 할당하기

- 데이터보다 행동을 먼저 결정하라

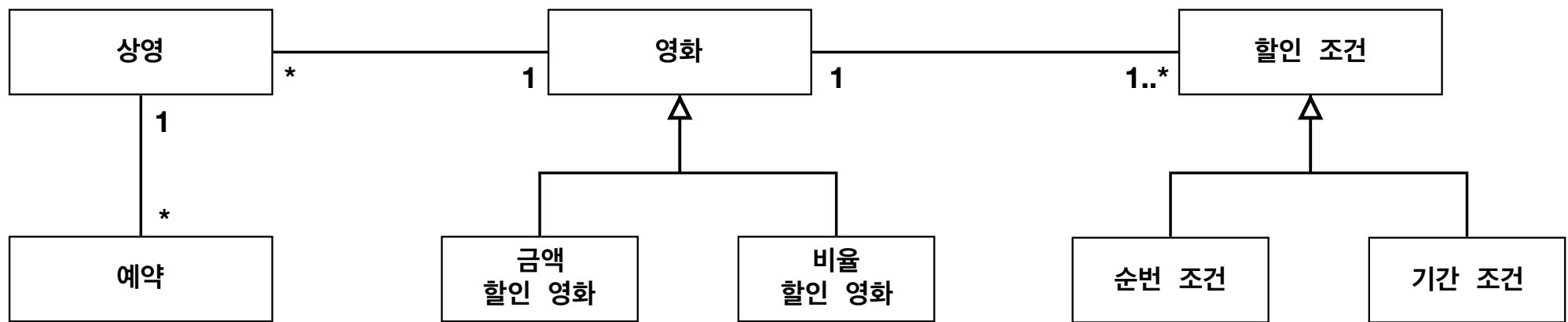
외부에 제공하는 행동

- 협력이라는 문맥 안에서 책임을 결정하라

클래스를 결정하고 그 클래스 책임을 찾기보다는
메시지를 결정하고 이 메시지를 누구에게 전송할지 찾아라.

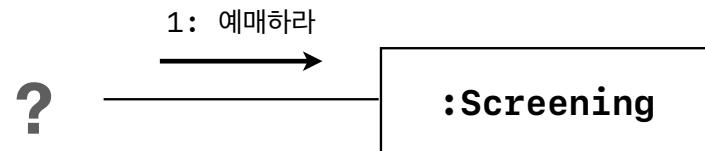
- 책임 주도 설계와 GRASP 패턴

도메인 개념 : 극장



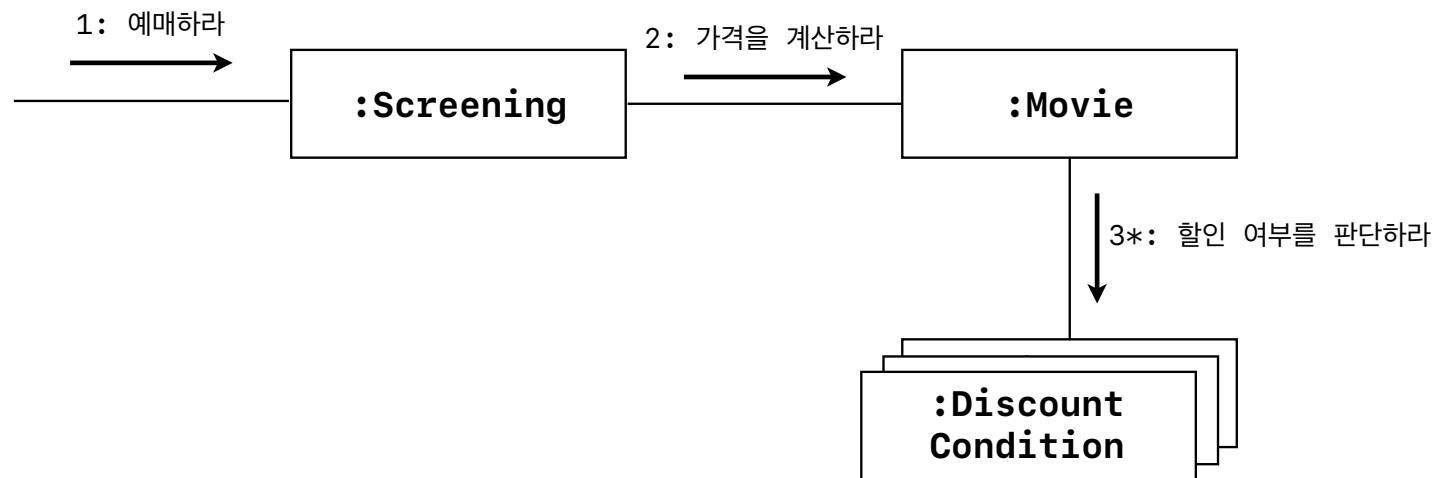
정보 담당자 Information Expert

메시지를 전송할 객체는 무엇을 원하는가?

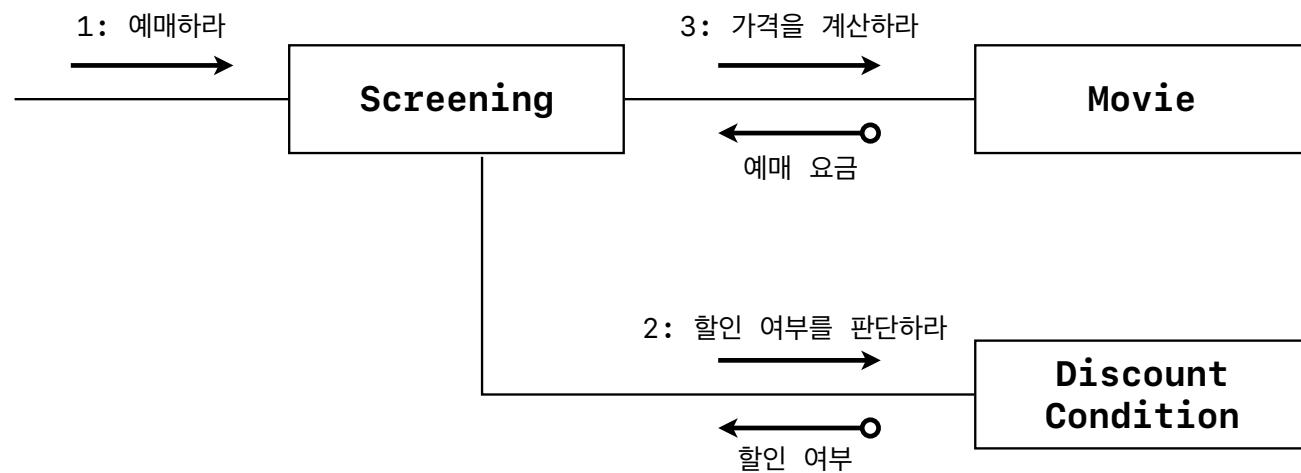


메시지를 수신할 적합한 객체는 누구인가?

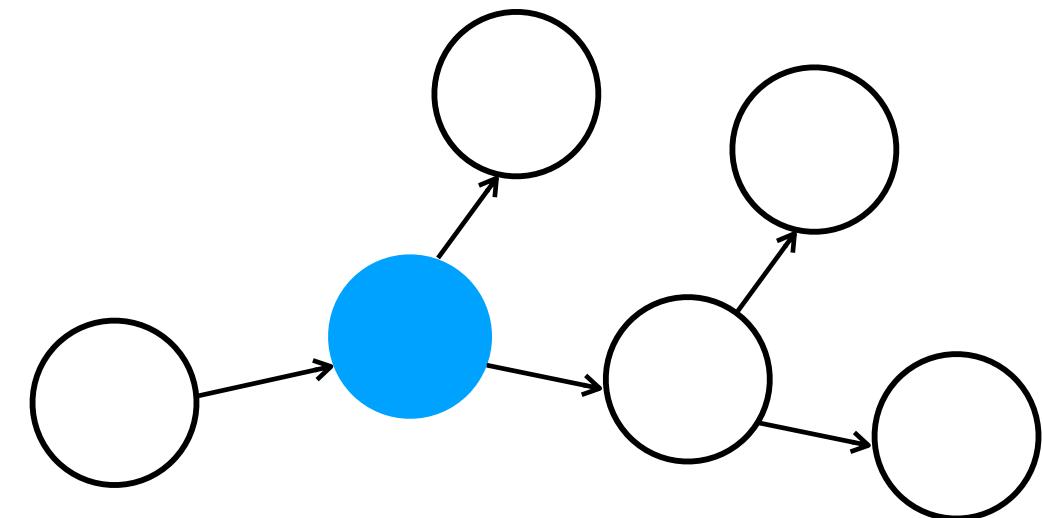
정보 담당자 Information Export



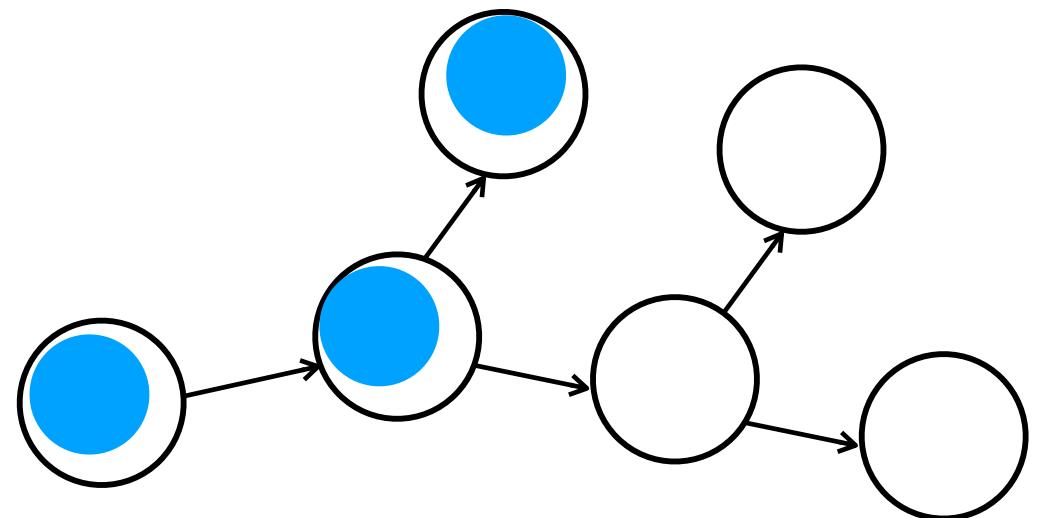
높은 응집도와 낮은 결합도



변경과 응집도

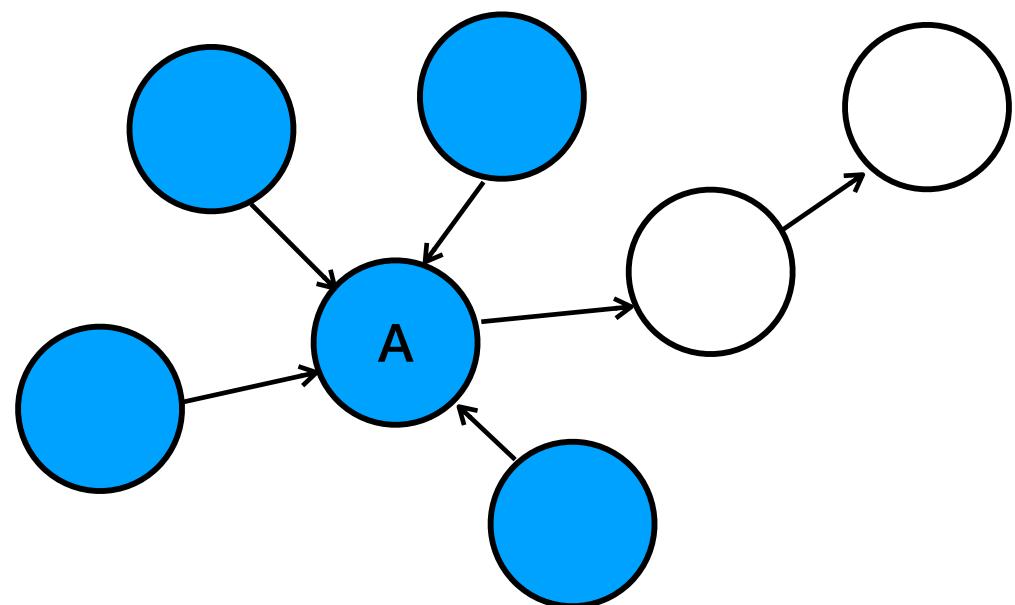


높은 응집도(High Cohesion)

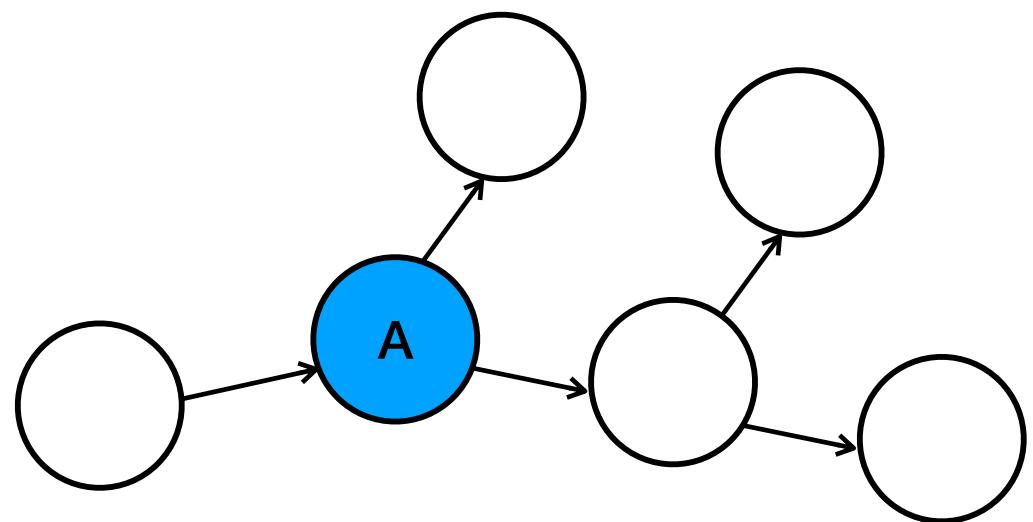


낮은 응집도(Low Cohesion)

변경과 결합도



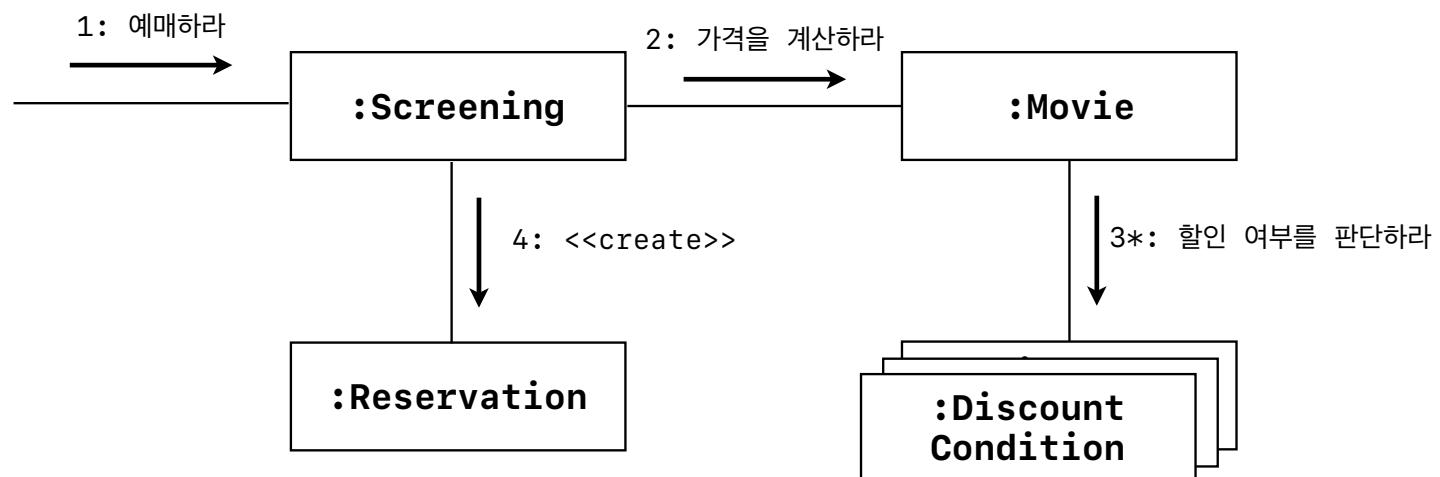
높은 결합도(High Coupling)



낮은 결합도(Low Coupling)

창조자 Creator

- B가 A를 포함하거나 협력을 위해 조합하는 경우
- B가 A를 기록하는 경우
- B가 A를 긴밀하게(복잡하게) 사용하는 경우
- B가 A의 초기값을 갖고 있는 경우



구현을 통한 검증 Screening

메시지 응답을 위한
메소드 추가 →

```
class Screening {  
    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {  
    }  
}
```

책임에 필요한
인스턴스 변수 추가 →

```
class Screening {  
    private var movie : Movie  
    private var sequence : Int  
    private var whenScreened : DateTime  
  
    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {  
    }  
}
```

책임에 필요한
내용을 구현 →

```
class Screening {  
    private var movie : Movie  
    private var sequence : Int  
    private var whenScreened : DateTime  
  
    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {  
        return Reservation(with: customer, screening: self, fee:calculateFee(audienceCount), audienceCount)  
    }  
  
    func calculateFee(int audienceCount) -> Money {  
        return movie.calculateMovieFee(self).times(audienceCount)  
    }  
}
```

구현을 통한 검증

Movie

메시지 응답을 위한
메소드 추가 →

```
class Movie {  
    func calculateMovieFee(screening: Screening) -> Money {  
    }  
}
```

책임에 필요한
인스턴스 변수 추가 →

```
class Movie {  
    private let title : String  
    private let runningTime : TimeInterval  
    private let fee : Money  
    private var discountConditions = Array<DiscountCondition>()  
  
    private let movieType : MovieType  
    private let discountAmount : Money  
    private let discountPercent : Double  
  
    func calculateMovieFee(screening: Screening) -> Money {  
    }  
}
```

구현을 통한 검증

Movie

책임에 필요한
내용을 구현 →

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }

    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private var discountConditions = Array<DiscountCondition>()

    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

    private func isDiscountable(for: Screening) -> Bool {
        return discountConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }
}
```

구현을 통한 검증

DiscountCondition

메시지 응답을 위한
메소드 추가 →

```
class DiscountCondition {
    func isSatisfied(by screening: Screening) -> Bool {
    }
}
```

책임에 필요한
내용을 구현 →

```
class DiscountCondition {
    private let type : DiscountConditionType
    private let sequence : Int
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    func isSatisfied(by screening: Screening) -> Bool {
        if type == .period {
            return isSatisfiedByPeriod(screening)
        }
        return isSatisfiedBySequence(screening)
    }

    private func isSatisfiedByPeriod(_ screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreened.toLocalTime()) <= 0) &&
            endTime.isAfter(to: screening.whenScreened.toLocalTime() >= 0)
    }

    private func isSatisfiedBySequence(_ screening: Screening) -> Bool {
        return sequence == screening.sequence
    }
}
```

구현을 통한 검증

Screening & DiscountCondition

정보 제공을 위한
접근성 변경 →

```
class Screening {  
    private var movie : Movie  
    private(set) var sequence : Int  
    private(set) var whenScreened : DateTime  
  
    func reserve(with customer: Customer, audienceCount: Int) -> Reservation {  
        return Reservation(with: customer, screening: self, fee:calculateFee(audienceCount), audienceCount)  
    }  
  
    func calculateFee(int audienceCount) -> Money {  
        return movie.calculateMovieFee(self).times(audienceCount)  
    }  
}
```

책임에 필요한
타입 추가 →

```
class DiscountCondition {  
    enum DiscountConditionType {  
        case Sequence, Period  
    }  
    private let type : DiscountConditionType  
    private let sequence : Int  
    private let dayOfWeek : DayOfWeek  
    private let startTime : DateTime  
    private let endTime : DateTime  
  
    //... 생략
```

SRP (단일 책임 원칙)

- 새로운 할인 조건 추가
- 순번 조건을 판단하는 로직 변경
- 기간 조건을 판단하는 로직이 변경되는 경우

응집도 판단하기

클래스가 하나 이상의 이유로 변경해야 한다면 응집도 낮은 것이다.
변경의 이유를 기준으로 클래스를 분리하라.

클래스의 인스턴스를 초기화하는 시점이 아니라, 나중에 서로 다른 속성을 초기화하고 있다면 응집도가 낮은 것이다.
초기화되는 속성의 그룹을 기준으로 클래스를 분리하라.

메소드 그룹이 속성 그룹을 사용하는지 여부로 나뉜다면 응집도가 늦은 것이다.
속성 그룹을 기준으로 클래스를 분리하라.

타입 분리하기

PeriodCondition - SequenceCondition

```
class PeriodCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    init(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) {
        self.dayOfWeek = dayOfWeek
        self.startTime = startTime
        self.endTime = endTime
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreended.toLocalTime() <= 0) &&
            endTime.isAfter(to: screening.whenScreended.toLocalTime() >= 0)
    }
}
```

```
class SequenceCondition {
    private let sequence : Int

    init(with sequence: Int) {
        self.sequence = sequence
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return sequence == screening.sequence
    }
}
```

두 종류 객체와 협력하는 Movie



타입 분리하기

Movie

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }
    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

    private var periodConditions = Array<PeriodCondition>()
    private var sequenceConditions = Array<SequenceCondition>()

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

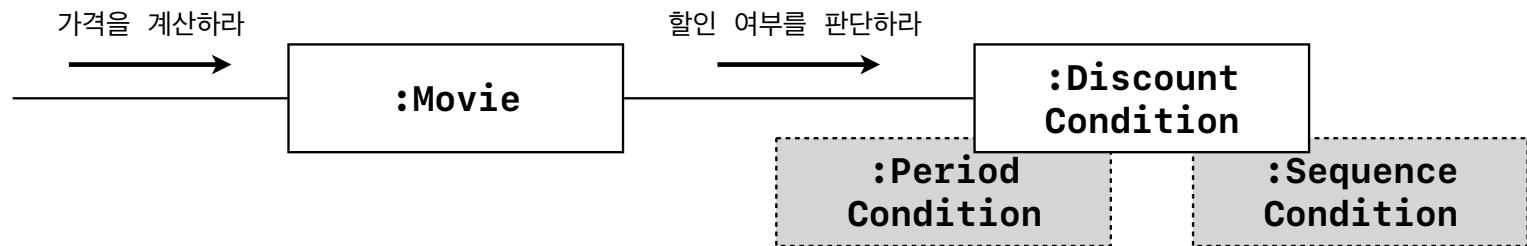
    private func mayPeriodConditions(with screening: Screening) -> Bool {
        return periodConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }

    private func maySequenceConditions(with screening: Screening) -> Bool {
        return sequenceConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }

    private func isDiscountable(for: Screening) -> Bool {
        return mayPeriodConditions(with: screening) || maySequenceConditions(with: screening)
    }
}
```

다형성 적용하기

DiscountCondition



```
protocol DiscountCondition {
    func isSatisfied(by screening: Screening) -> Bool
}
```

```
class PeriodCondition : DiscountCondition {
}
```

```
class SequenceCondition : DiscountCondition {
}
```

다형성 적용하기 | Movie

```
class Movie {
    enum MovieType {
        case AmountDiscount
        case PercentDiscount
        case NoneDiscount
    }
    private let title : String
    private let runningTime : TimeInterval
    private let fee : Money
    private let movieType : MovieType
    private let discountAmount : Money
    private let discountPercent : Double

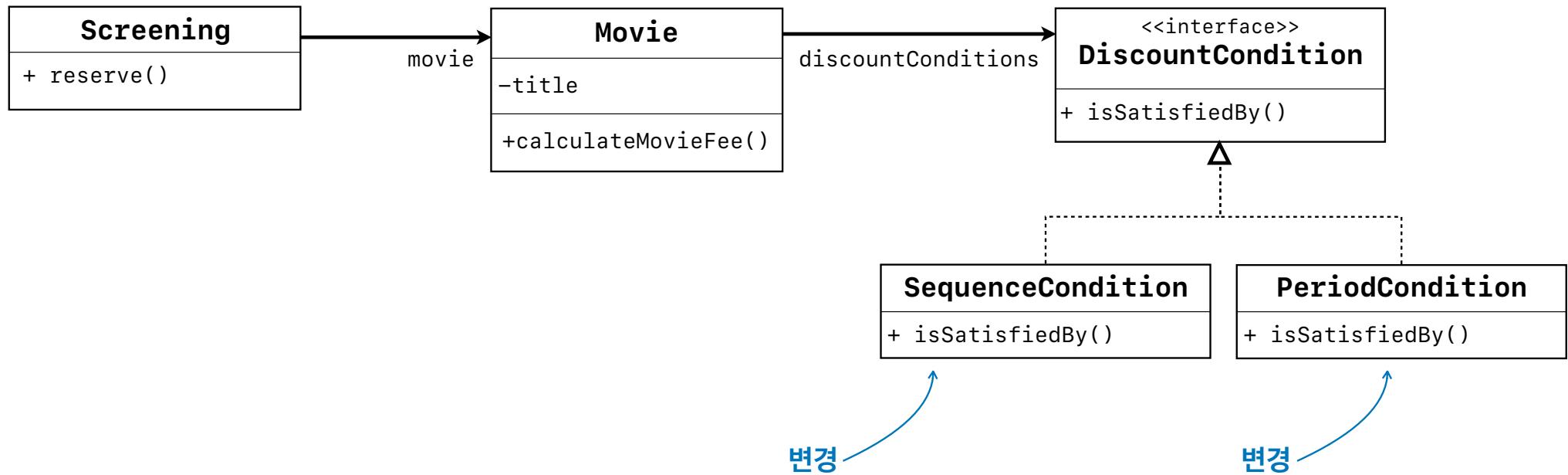
    private var discountConditions = Array<DiscountCondition>()

    func calculateMovieFee(screening: Screening) -> Money {
        if isDiscountable(for: screening) {
            return fee.minus(calculateDiscountAmount())
        }
        return fee
    }

    private func isDiscountable(for: Screening) -> Bool {
        return discountConditions.filter{ $0.isSatisfied(by: screening) }.count > 0
    }
}
```

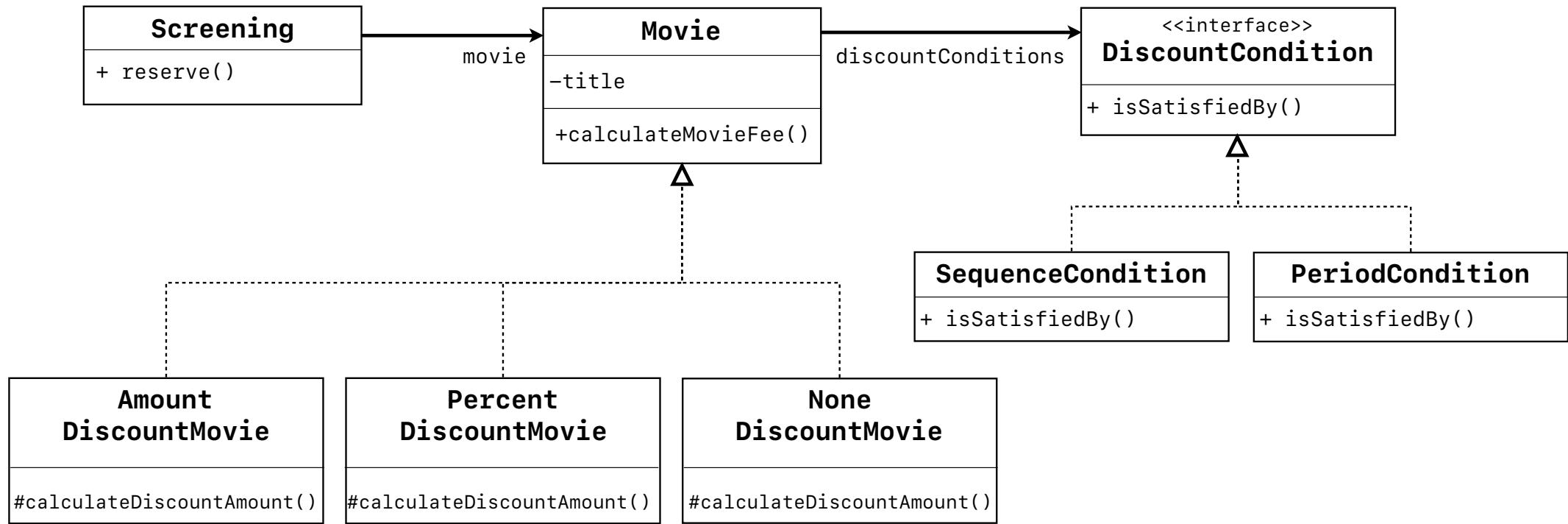
변경으로부터 보호

Protected Variations



다형성 적용하기

Movie



몬스터 메소드

ReservationAgency

```
class ReservationAgency {
    func reserve(screening : Screening, customer: Customer, audienceCount: Int) -> Reservation {
        let movie = screening.getMovie()
        var discountable = false
        for condition in movie.getDiscountConditions() {
            if condition.getType() == .Period {
                discountable = screening.getWhenScreened().getDayOfWeek().equals(codition.getDayOfWeek()) &&
                    condition.getStartTime().compareTo(screening.getWhenScreend().toLocalTime()) <= 0 &&
                    condition.getEndTime().compareTo(screening.getWhenScreend().toLocalTime()) <= 0 &&
            }
            else {
                discountable = condition.getSequence() == screening.getSequence()
            }

            if discountable { break }
        }

        var fee : Money
        if discountable {
            var discountAmount = Money.ZERO
            switch movie.getMovieType() {
            case AMOUNT_DISCOUNT:
                discountAmount = movie.getDiscountAmount()
            case PERCENT_DISCOUNT:
                discountAmount = movie.getFee().times(movie.getDiscountPercent())
            case NONE_DISCOUNT:
                break
            }
            fee = movie.getFee().minus(discountAmount).times(audienceCount)
        }
        else {
            free = movie.getFee()
        }

        return Reservation(custom, screening, fee, audienceCount)
    }
}
```

몬스터 메소드

ReservationAgency

- 어떤 일을 수행하는지 파악하기 어렵기 때문에 코드 전체를 이해하는 데 시간이 오래 걸린다.
- 하나의 메소드 안에서 너무 많은 작업을 처리하기 때문에 변경이 필요할 때 찾기 어렵다.
- 메소드 내부의 일부 로직만 수정하더라도 메소드 나머지 부분에 버그가 생길 수 있다.
- 로직의 일부만 재사용하는 것이 불가능하다.
- 코드를 재사용하는 유일한 방법은 원하는 코드를 복사해서 붙여넣어야 해서 중복이 생긴다.

```
class ReservationAgency {  
    func reserve(screening : Screening, customer: Customer, audienceCount: Int) -> Reservation {  
        let discountable = mayDiscountable(screening)  
        let fee = calculateFee(with: screening, discountable: discountable, audienceCount: audienceCount)  
        return Reservation(screening, customer, audienceCount, fee)  
    }  
}
```

메시지와 메소드

[anObject foobar];

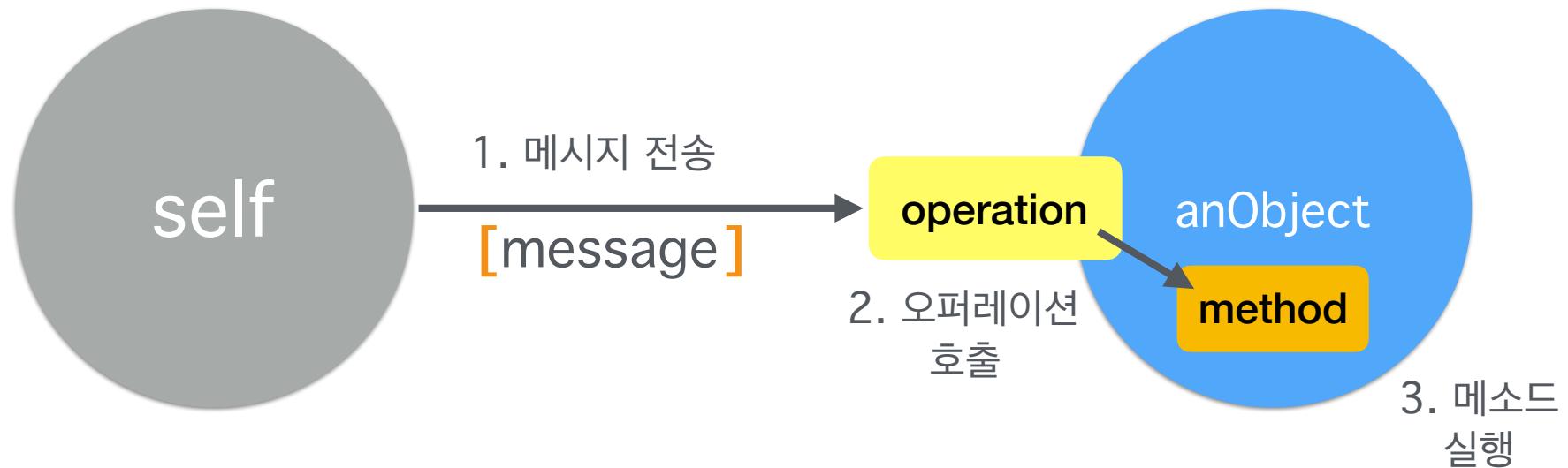
anObject.foobar()

instance

method



메시지, 오퍼레이션, 메소드



디미터 법칙

* **객체들** 협력 경로를 제한하면 결합도를 낮출 수 있다

→ 낯선 자에게 말하지 말라. 오직 인접한 이웃하고만 말하라

* 아래 조건을 만족하는 인스턴스에만 메시지를 전송하자

- self 객체
- 메소드의 매개변수
- self 내부 속성
- self 내부 속성으로 선언한 콜랙션 요소
- 메소드 내에서 생성한 지역 객체

디미터 법칙 예시

before

```
class ReservationAgency {
    func reserve(screening: Screening, customer: Customer, audienceCount: Int) -> Reservation {
        let movie = screening.getMovie()
        var discountable = false
        for condition in movie.getDiscountConditions() {
            if condition.getType() == .Period {
                discountable = screening.whenScreened().getDayOfWeek().equals(condition.getDayOfWeek()) &&
                    condition.getStartTime().compareTo(screening.whenScreened().toLocalTime()) <= 0 &&
                    condition.getEndTime().compareTo(screening.whenScreened().toLocalTime()) <= 0 &&
            }
            else {
                discountable = condition.getSequence() == screening.getSequence()
            }

            if discountable { break }
        }
    }

    var fee : Money
    if discountable {
        var discountAmount = Money.ZERO
        switch movie.getMovieType() {
        case AMOUNT_DISCOUNT:
            discountAmount = movie.getDiscountAmount()
        case PERCENT_DISCOUNT:
            discountAmount = movie.getFee().times(movie.getDiscountPercent())
        case NONE_DISCOUNT:
            break
        }
        fee = movie.getFee().minus(discountAmount).times(audienceCount)
    }
}
```

디미터 법칙 예시

```
class ReservationAgency {  
    func reserve(screening : Screening, customer: Customer, audienceCount: Int) -> Reservation {  
        let fee = screening.calculateFee(audienceCount)  
        return Reservation(custom, screening, fee, audienceCount)  
    }  
}
```

디미터 법칙 함정 exception

```
class PeriodCondition : DiscountCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date

    init(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) {
        self.dayOfWeek = dayOfWeek
        self.startTime = startTime
        self.endTime = endTime
    }

    func isSatisfied(by screening: Screening) -> Bool {
        return dayOfWeek.equals(screening.whenScreened.dayOfWeek) &&
            startTime.compare(to: screening.whenScreened.toLocalTime() <= 0) &&
            endTime.isAfter(to: screening.whenScreened.toLocalTime() >= 0)
    }
}

class PeriodCondition : DiscountCondition {
    private let dayOfWeek : DayOfWeek
    private let startTime : Date
    private let endTime : Date
    // 중간 생략
    func isSatisfied(by screening: Screening) -> Bool {
        return screening.isDiscountable(dayOfWeek: dayOfWeek, startTime: startTime, endTime: endTime)
    }
}

extension Screening {
    func isDiscountable(dayOfWeek: DayOfWeek, startTime: DateTime, endTime: DateTime) -> Bool {
        return dayOfWeek.equals(whenScreened.dayOfWeek) &&
            startTime.compare(to: whenScreened.toLocalTime() <= 0) &&
            endTime.isAfter(to: whenScreened.toLocalTime() >= 0)
    }
}
```

명령-쿼리 분리 원칙

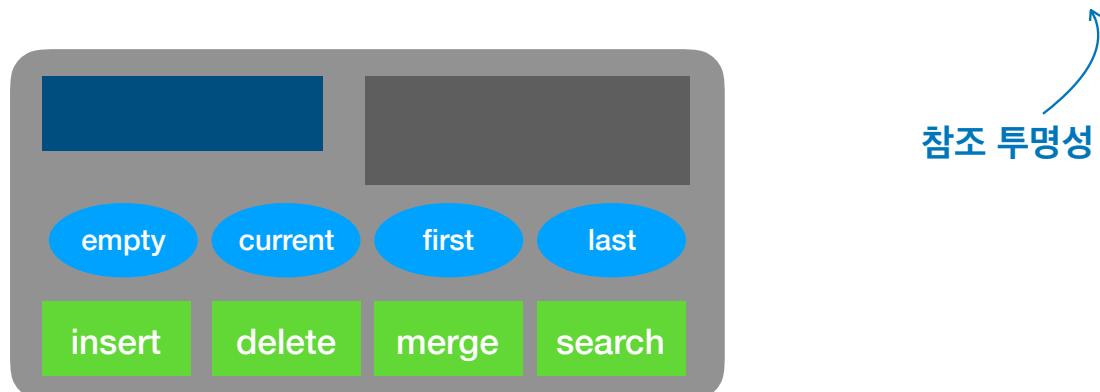
- * 프로시저와 함수

→ 프로시저 : 부수효과를 발생할 수 있지만 값을 반환할 수 없다

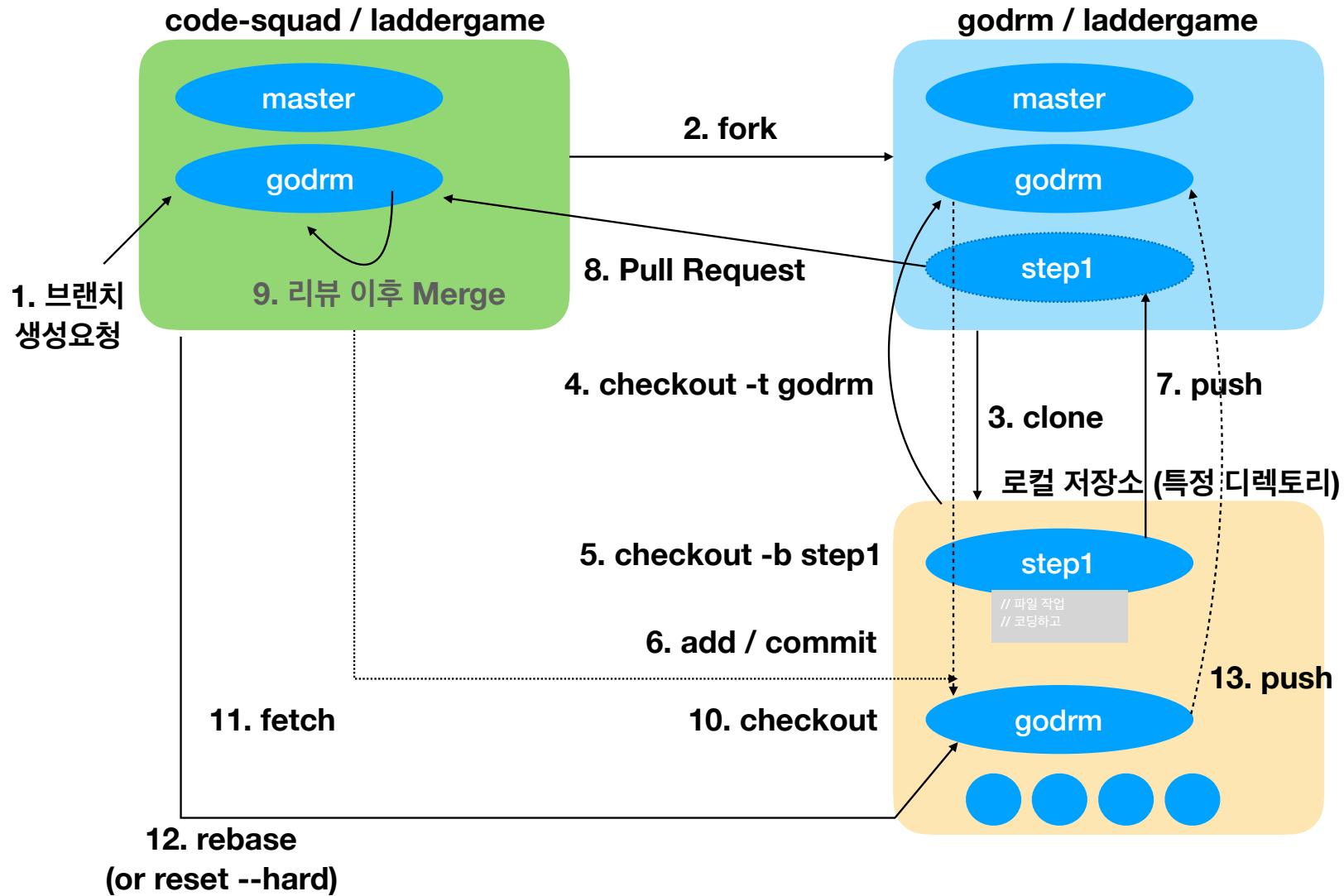
* 객체 상태를 변경하는 명령은 반환값을 가질 수 없다 (mutable)

→ 함수 : 값을 반환할 수 있지만 부수효과를 발생시킬 수 없다

* 객체 정보를 반환하는 쿼리는 상태를 변경할 수 없다 (immutable)



코드리뷰 방식



소프트웨어 모듈 존재 이유

1. 기능 구현

- 실행되어야 하고 제대로 동작해야 한다.

2. 라이프 사이클

- 간단한 작업으로 모듈의 기능을 변경 가능해야 함
- 라이프 사이클이 계속 되기 때문에 변경은 피할 수 없음
- 변경하기 어려운 모듈은 동작하더라도 개선

3. 의사소통

- 코드를 읽는 사람과 의사소통 하는 것
- 읽는 사람과 의사소통하기 어려운 모듈은 개선

소프트웨어 유지보수란
지금처럼만 잘 동작하도록 만드는 것이 아니고
변화하는 세상에서 계속 유용하도록 만드는 것이다

Software maintenance is
not 'keep it working like before'.
It is 'keep it being useful in a changing world'

– Jessica Kerr

Best Practices & Styles

- * Tab vs. Space
- * Braces Styles
- * Swift API Design Guideline
- * Variable, Function, Class - Naming Rules
- * Struct/Class, Sync/Async, Functional/OOP
- * Design Pattern vs. Implement Pattern

구글 리뷰 가이드라인

<https://google.github.io/eng-practices/review/>

<https://soojin.ro/review/>

- 코드가 잘 설계되었는지
- 사용자에게 유용한 기능을 제공하는지
- UI 변경이 합리적이고 보기 좋은지
- 병렬 작업이 안전하게 실행되는지
- 불필요하게 복잡하진 않은지
- 지금 당장 필요없는 기능을 개발하진 않았는지
- 유닛 테스트가 적절한지
- 테스트는 잘 설계되었는지
- 모든 이름을 잘 지었는지
- 주석은 명료하고 유용한지, 무엇보다는 왜를 나타내는지
- 문서에 반영이 되었는지
- 스타일 가이드를 따르는지

리뷰 요청 문구 예시

rpc: remove size limit on RPC server message freelist.

Servers like FizzBuzz have very large messages and would benefit from reuse. Make the freelist larger, and add a goroutine that frees the freelist entries slowly over time, so that idle servers eventually release all freelist entries.

Construct a Task with a TimeKeeper to use its TimeStr and Now methods.

Add a Now method to Task, so the borglet() getter method can be removed (which was only used by OOMCandidate to call borglet's Now method). This replaces the methods on Borglet that delegate to a TimeKeeper.

Allowing Tasks to supply Now **is** a step toward eliminating the dependency on Borglet. Eventually, collaborators that depend on getting Now from the Task should be changed to use a TimeKeeper directly, but this has been an accommodation to refactoring **in** small steps.

Continuing the long-range goal of refactoring the Borglet Hierarchy.

애자일 설계

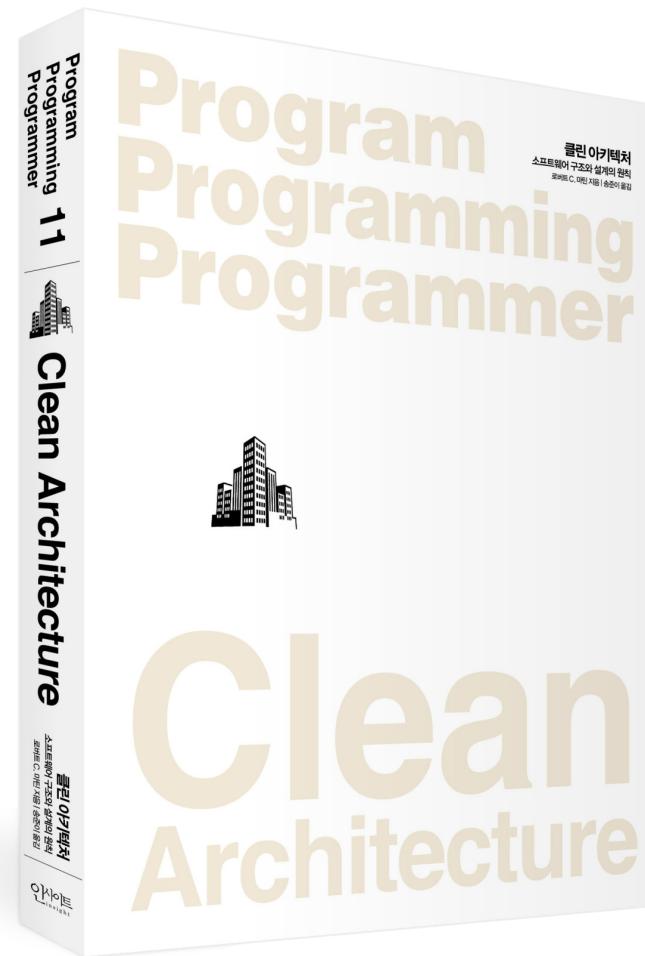
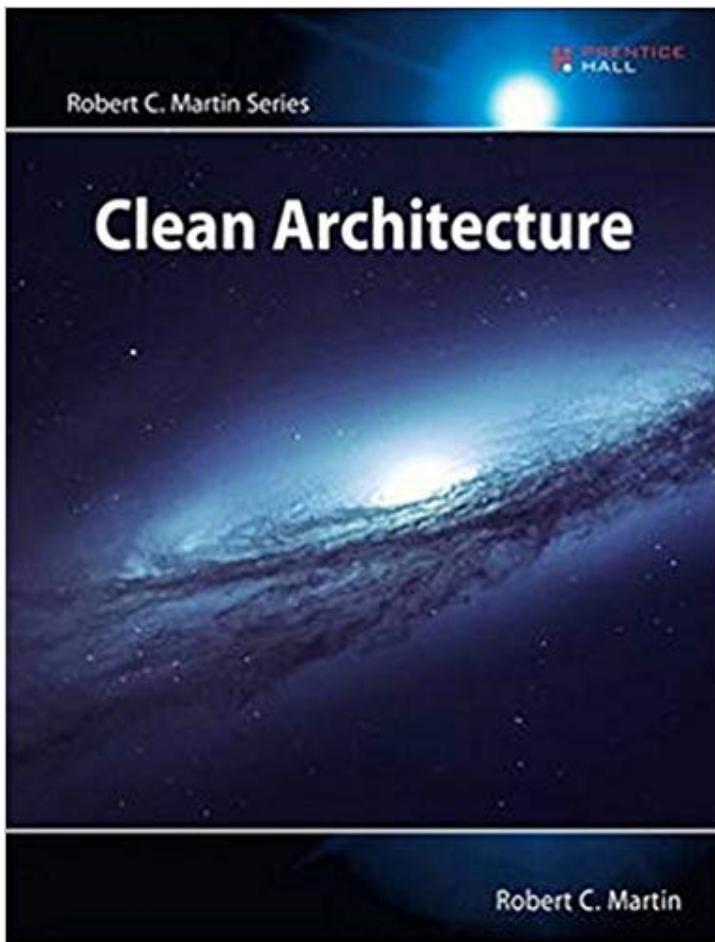
부패한 설계와 부패한 코드를 청소하기

클린 아키텍처와 클린 코드를 유지하기

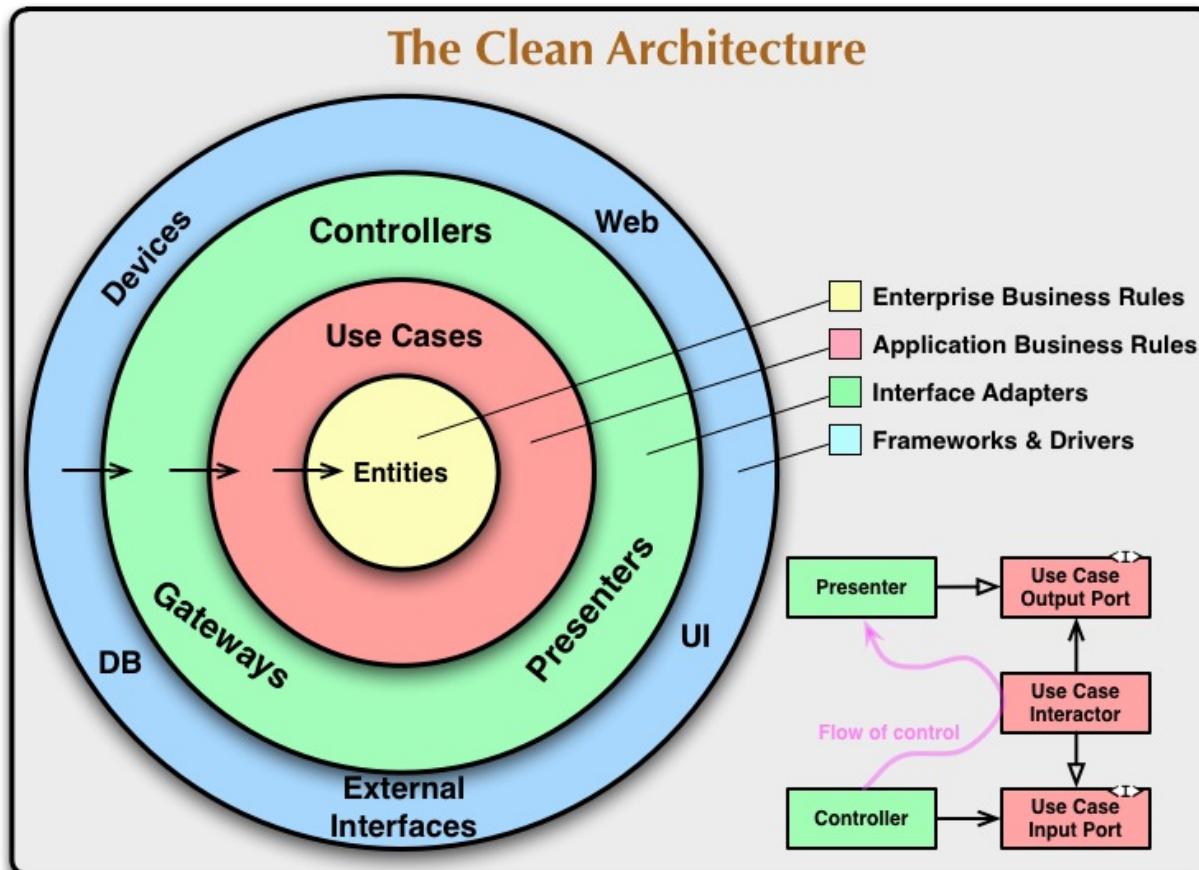
애자일 설계는 결과가 아니라 과정이다

매순간 시스템의 설계를 가능한 간단하고, 명료하고, 명확한 표
현으로 유지하려는 노력이다

클린 아키텍처

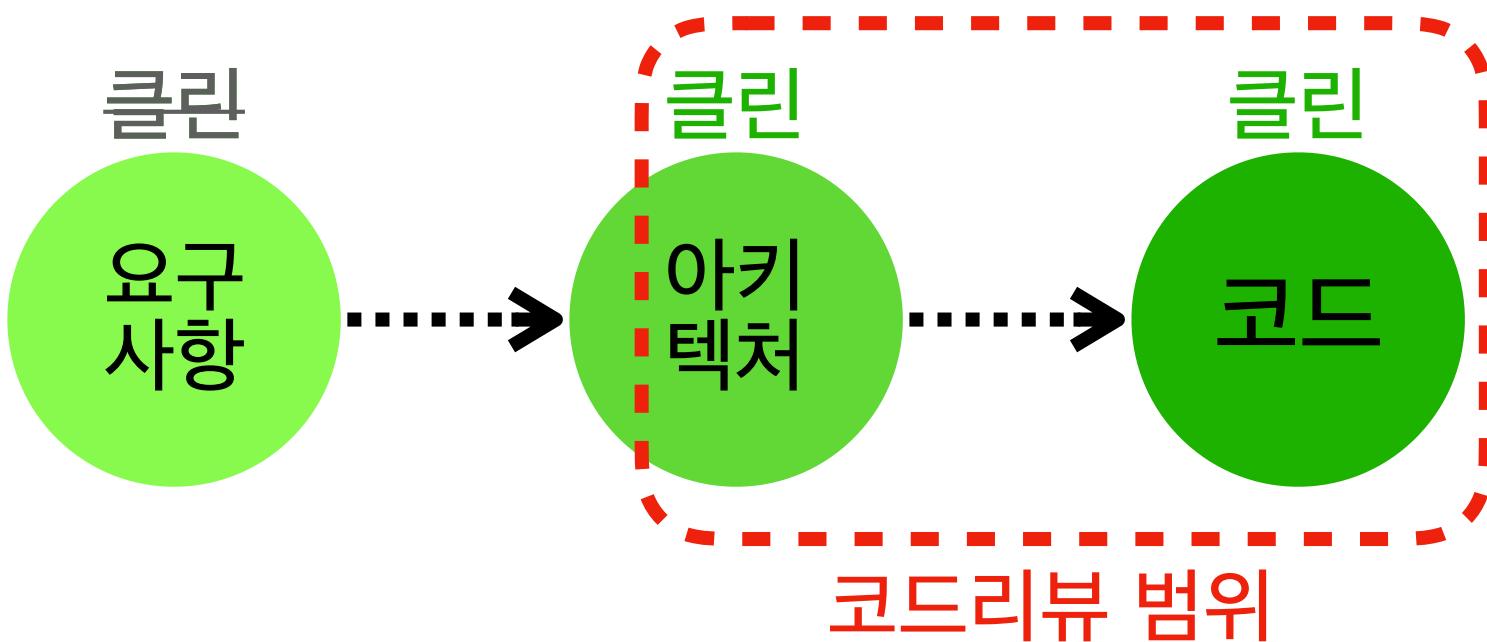


The Clean Architecture



<http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

클린 코드와 클린 아키텍처



설계의 악취

부패하고 있는 소프트웨어의 냄새

1. 경직성

시스템을 변경하기 어렵다. 변경하려면 시스템의 다른 부분들까지 많이 변경해야 한다.

2. 취약성

변경을 하면 부분과 개념적/직접적으로 아무런 관련이 없는 부분까지 망가진다.

3. 부동성

시스템을 다른 시스템에서 재사용할 수 있는 콤포넌트로 구분하기 어렵다.

4. 점착성

옳은 동작을 하는 것이 잘못된 동작을 하는 것보다 더 어렵다.

5. 불필요한 복잡성

직접적인 효용/쓸모가 전혀 없는 기반구조가 설계에 포함되어 있다.

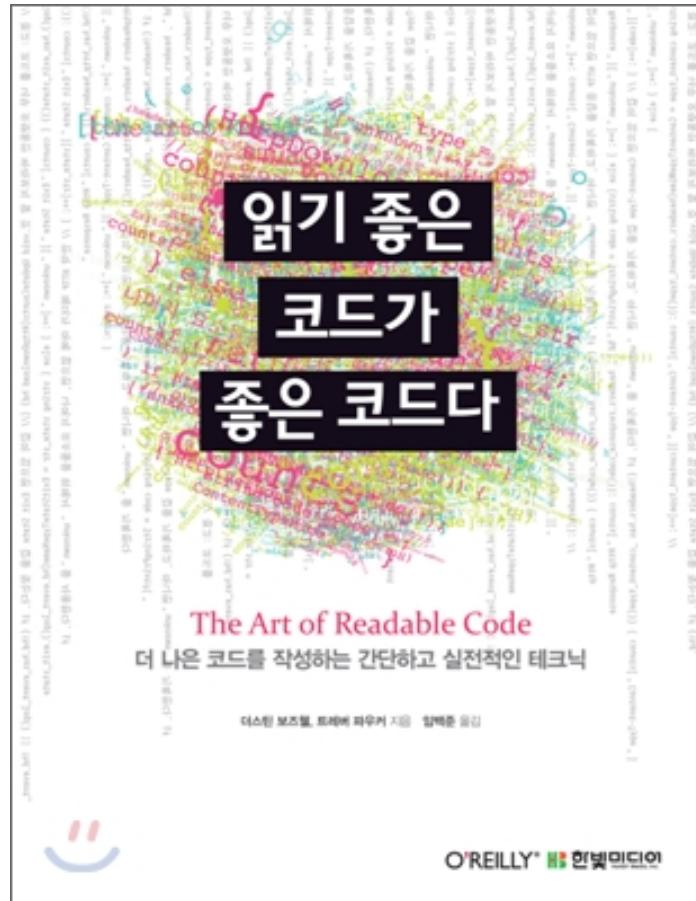
6. 불필요한 반복

단일 추상 개념으로 통합할 수 있는 반복적인 구조가 설계에 포함되어 있다.

7. 불투명성

읽고 이해하기 어렵다. 그 의도를 잘 표현하지 못한다.

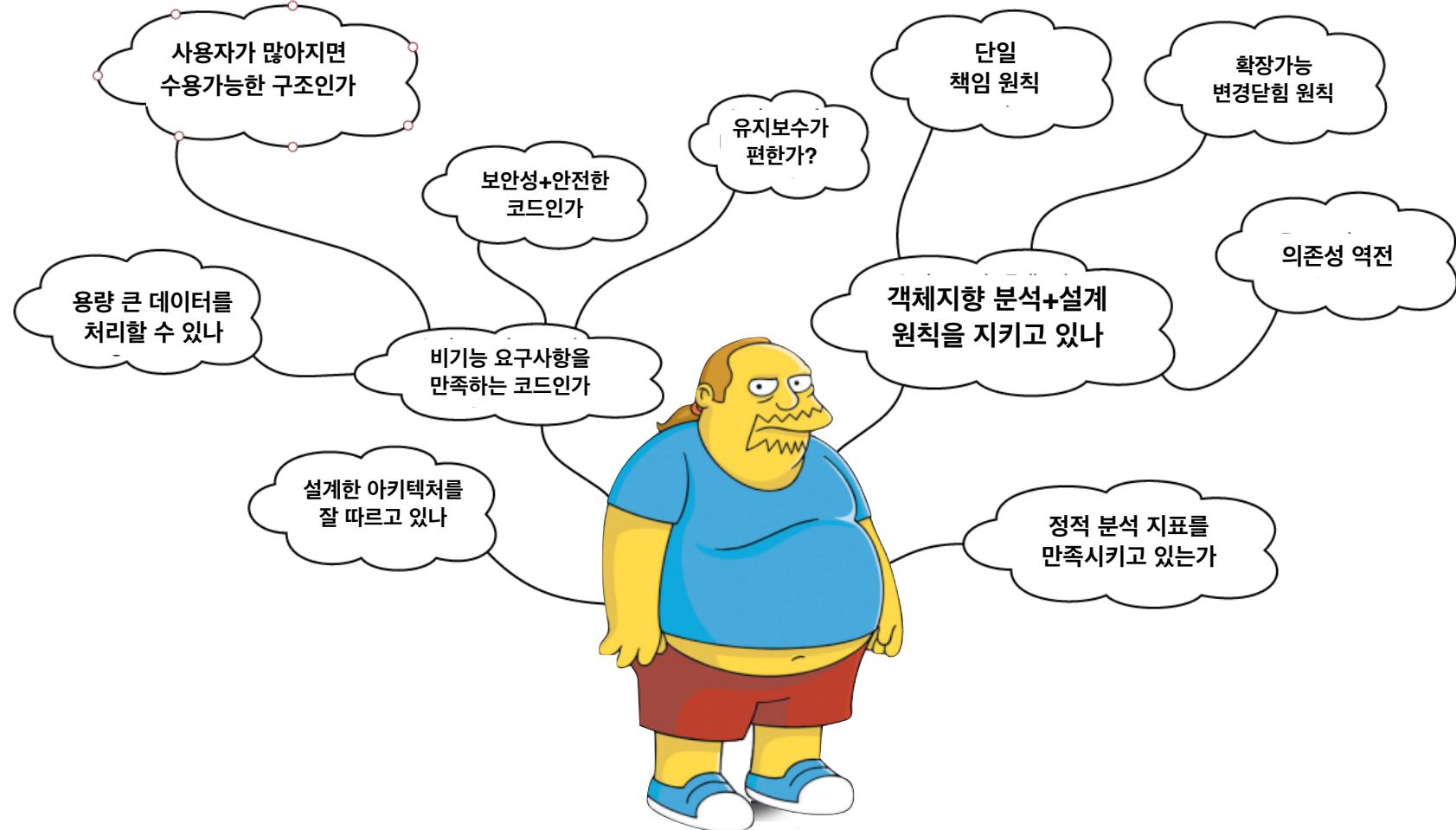
코드 읽기





코드리뷰를 한다는 것은...





깊이있는 코드 리뷰

비기능 요구사항

유지보수 편리성 Main - 가독성, 테스트 편리성, 디버그 편의성, 설정 편리성

재사용성 Reusability - DRY, components, generics

안정성 Reliability - Exception handling and cleanup

개선 편리성 Extensibility

안전성 Security - 데이터 검증, 인증, 권한, 암호화

성능 Performance - 문자열 반복 처리, Lazy Loading, 비동기, 병렬처리

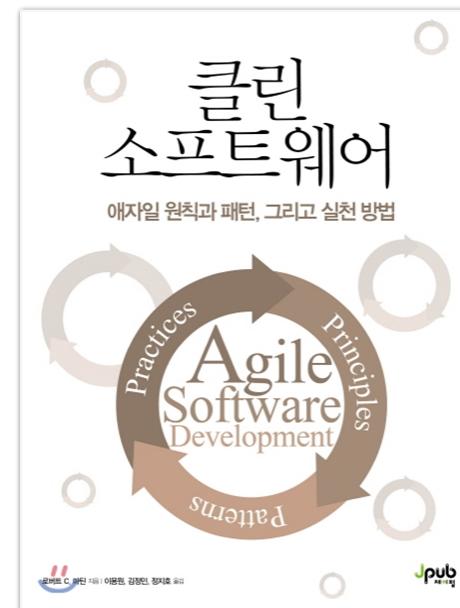
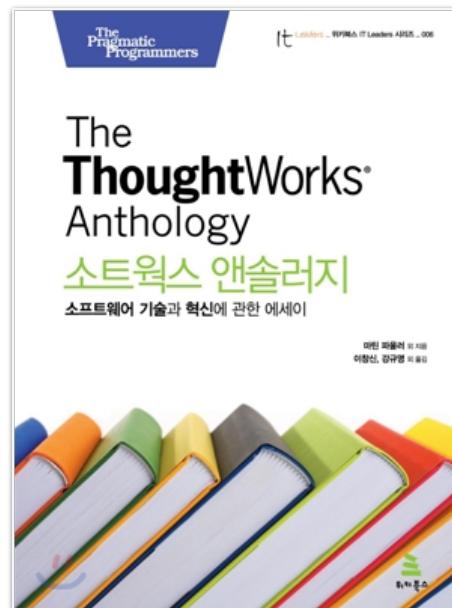
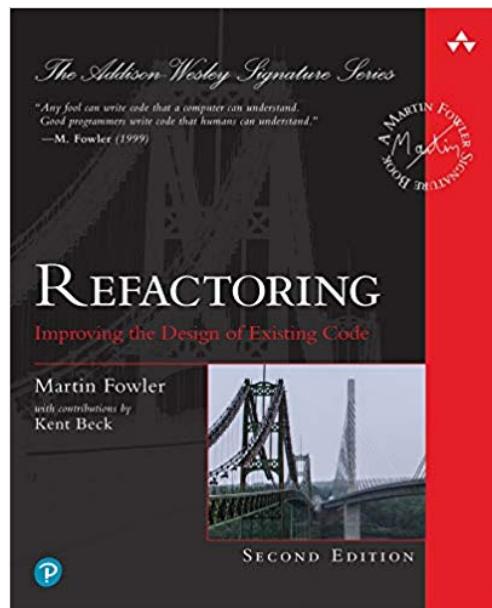
규모 확장성 Scalability - 대용량 데이터나 대용량 사용자 처리

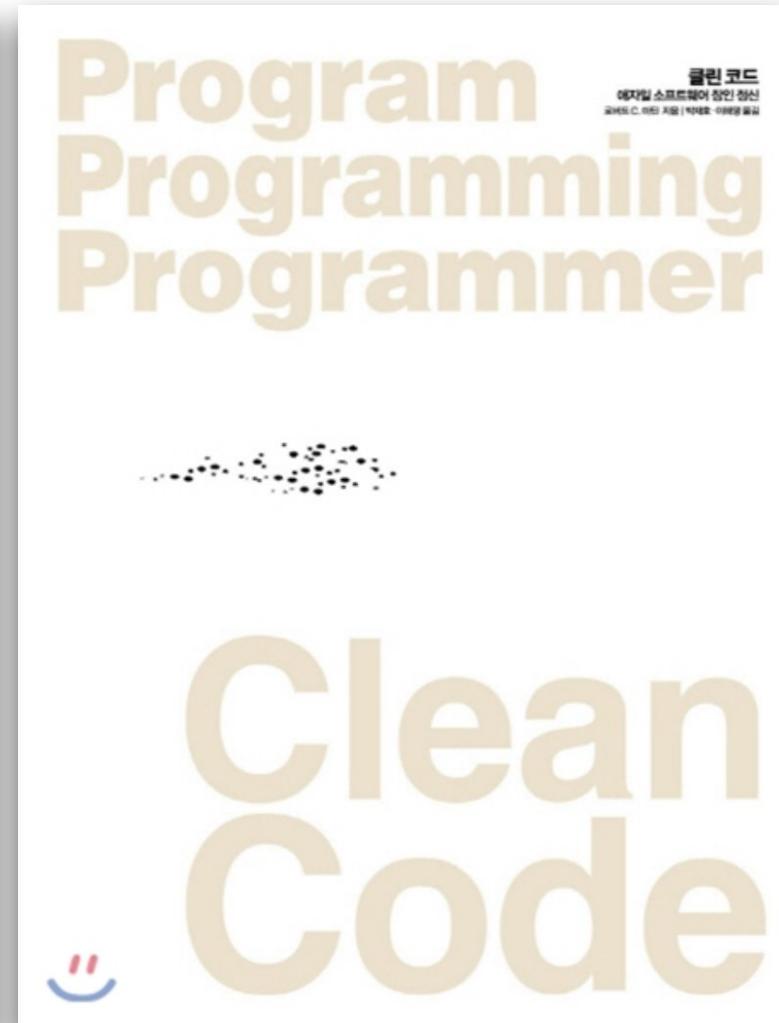
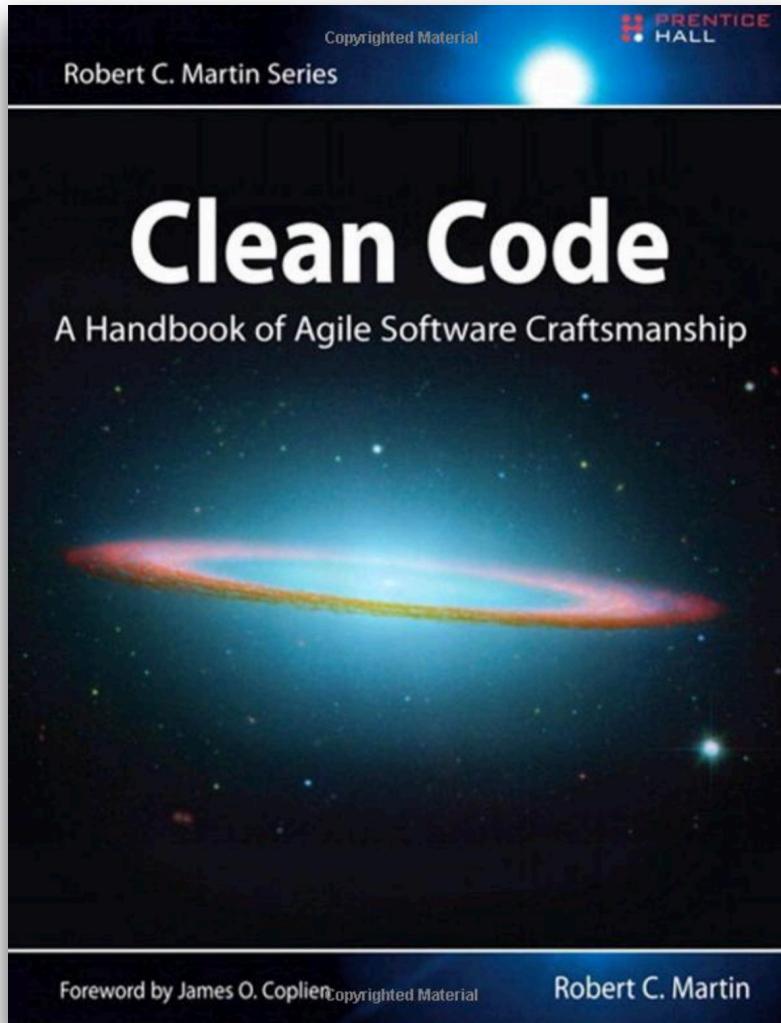
사용성 Usability - 인터페이스 설계와 사용 편리성

셀프 코드리뷰



참고 도서





나쁜 냄새

연속적으로 따라가는 탐색을 피하라

한 소스 파일에 여러 언어를 사용한다

일관성을 유지하라

경계 조건을 캡슐화하라

당연한 동작을 구현하지 않는다

서술적 변수

너무 많은 인수

중복

경계를 올바로 처리하지 않는다

출력 인수

안전 절차 무시

과도한 정보

성의없는 주석

부적절한 정보

기초 클래스가 파생 클래스에 의존한다

죽은 함수

플래그 인수

쓸모없는 주석

주석 처리된 코드

추상화 수준이 올바르지 못하다

함수는 한 가지만 해야 한다

중복된 주석

수직 분리

부정 조건은 피하라

숨겨진 시간순서를 나열하라

일관성 부족

설정 정보는 최상위 단계에 둬라

조건을 캡슐화하라

선택자 인수

여러 단계로 빌드해야 한다

잡동사니

기능 욕심

이름과 기능이 일치하는 함수

인위적 결합

알고리즘을 이해하라

죽은 코드

여러 단계로 테스트해야 한다

잘못 지운 책임

모호한 의도

논리적 의존성은 물리적으로 드러내라

비교문 보다는 다형성을 사용하라

매직 숫자는 상수로 교체하라

간단한 설계 규칙 (우선순위)

1. 테스트를 통과할 것 (상)

- 테스트 코드는 자신의 코드가 어떤 동작을 해야 하는지 명확하도록 도와줌
- 테스트 코드를 통과하는 모듈을 작성하는 것은 TDD가 아니더라도 필요
- 테스트가 가능하고 테스트가 쉬운 구조가 (못하는것보다는) 좋은 구조

2. 의도를 드러낼 것 (중)

- 변수이름, 함수이름, 클래스이름에서도 의도를 표현
- private 인지 public 인지 선언한 속성에 대해서도 명확한 의도가 있어야 함

3. 중복을 제거할 것 (중)

- 중복된 코드가 보이면 메소드를 분리하거나 타입을 분리하거나 객체를 분리해서 중복을 제거
- 메소드 인터페이스가 동일하면 프로토콜을 활용하고, 구현 내용은 같고 타입만 다르면 제네릭을 활용

4. 구성 요소를 최소화할 것 (하)

- 함수의 매개변수, 객체의 속성이나 메소드도 꼭 필요한 경우에 추가
- 명확한 역할이 없는 요소는 존재 의미도 없는 경우가 많다. 의도를 다시 한 번 고민해보세요