andrena OBJECTS

Experts in agile software engineering

Agile Methoden in der nativen App-Entwicklung

ObjektForum Mannheim

25. November 2019

Jan Bauer

jan.bauer@andrena.de

Steffen Heberle

steffen.heberle@andrena.de

Was verstehen wir unter nativer App-Entwicklung?

- Android mit Java oder Kotlin
- iOS mit Swift
- Smartphone
- Tablet

Was verstehen wir unter „Agilen Methoden“?

Scrum

- Rollen
- Events
- Artefakte

XP-Praktiken (Agile Methoden)

- Pair Programming
- Continuous Integration
- Testing
- Clean Code
- Refactoring
- …

Software-Qualität

- Verständlichkeit
- Wartbarkeit
- Erweiterbarkeit
- Wiederverwendbarkeit
- Testbarkeit

Überblick

- Clean Code und Refactoring
- Testing
  - UI-Tests
  - Unit-Tests
  - Integration-Tests
- Architektur
- Continuous Integration

Clean Code durch Refactoring

- Refactoring: Verbesserung der Code-Struktur ohne das beobachtbare Verhalten zu verändern

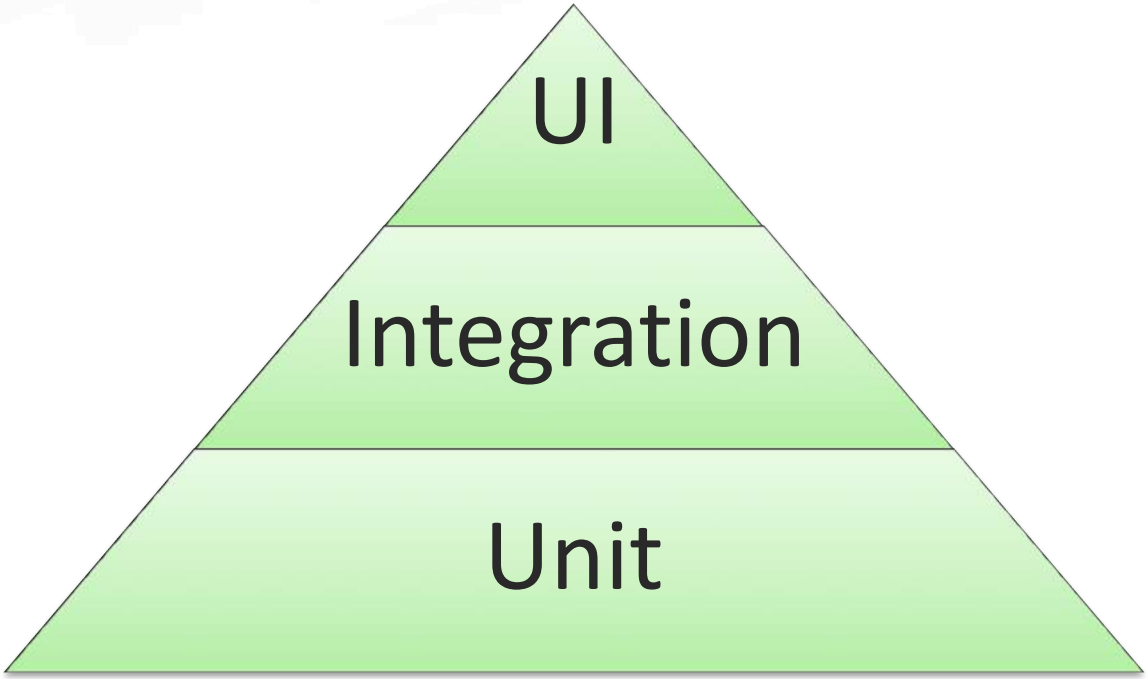
- Sicherheit und Vertrauen durch
  - Automatisierte Tests
  - Automatisierte Refactorings

Refactoring in der Praxis

- Android
  - IntelliJ / Android Studio
- iOS
  - Xcode

  ⇒ Compiler-Driven Refactoring

Testpyramide



UI-Tests

manuell

- Ad hoc oder Exploratory Testing
- Vergleich Android und iOS
- Vielfalt der Displaygrößen

UI-Tests

automatisiert

- Kosten-Nutzen-Verhältnis
- Vermeidung von Regression
- Tools
  - Android: Espresso
  - iOS: XCTest mit XCUI-Klassen

Unit-Tests

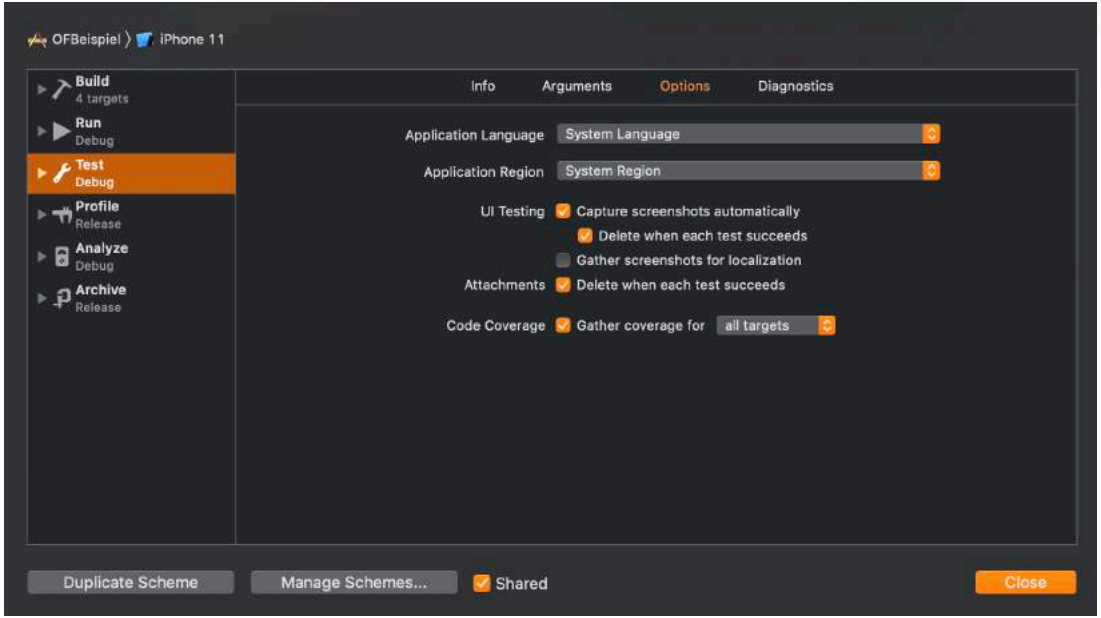
Android

- JUnit
- Code Coverage mit Jacoco
- JUnit- oder AssertJ-Matcher
- Test-Isolation mit z.B. Mockito, MockK

Unit-Tests

iOS

- XCTest
- Code Coverage mit Xcode
- Matcher: Nimble
- Test-Isolation: SwiftyMocky



iOS Test-Matcher

XCTest

```
class MyTest: XCTestCase {

    func testTwoPlusTwoIsFour() {
        let result = 2 + 2
        XCTAssertEqual(result, 4)
    }

}
```

Nimble

```
class MyTest: XCTestCase {

    func testTwoPlusTwoIsFour() {
        let result = 2 + 2
        expect(result).to(equal(4))
    }

}
```

Unit-Tests

iOS Test-Isolation mit SwiftyMocky

```
//sourcery: AutoMockable
protocol Service {

    func receiveNextMessage()
    func send(message: String)

}
```


Unit-Tests

iOS Test-Isolation mit SwiftyMocky

```
class MyIsolatedTest: XCTestCase {

    func testDemonstrateGiven() {
        let service = MyServiceMock()
        Given(service, .receiveNextMessage(willReturn: "Message in a bottle."))


        let message = service.receiveNextMessage()


        expect(message).to(contain("bottle"))
    }

}
```


Unit-Tests

iOS Test-Isolation mit SwiftyMocky

```
class MyIsolatedTest: XCTestCase {

    func testDemonstrateVerify() {
        let service = MyServiceMock()


        service.sendMessage(content: "De Do Do Do De Da Da Da")


        Verify(service, .atLeastOnce, .sendMessage(content: .value("De Do Do Do De Da Da Da")))
    }

}
```

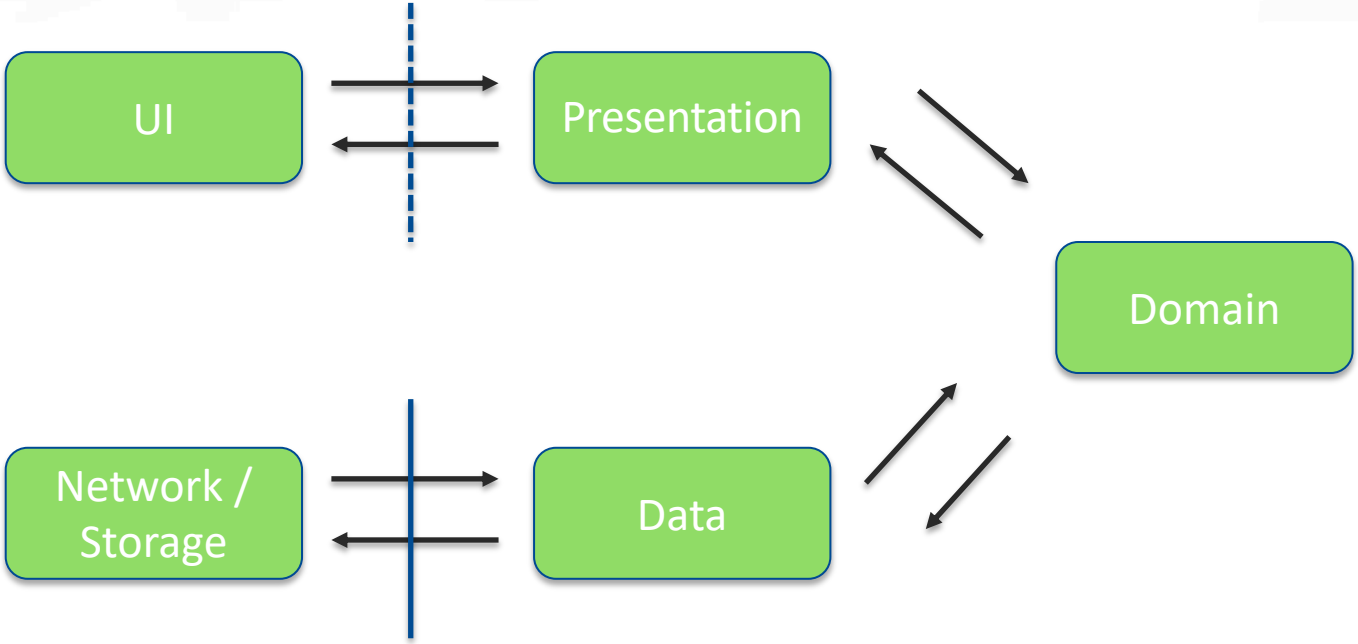

Unit-Tests

iOS App-Delegate

- wird aufgerufen, wenn der App-Start systemseitig abgeschlossen ist
- erledigt evtl. aufwändige Arbeit
- verzögert Test-Start

⇒ lade minimalen App-Delegate für Tests aus main.swift

Integrationstests



Architektur

Warum ist Struktur notwendig?

- Testbarkeit
- Erweiterbarkeit
- Leichtere Zusammenarbeit mit weniger Merge-Konflikten
- Entkopplung von Technologien: z.B. Abstraktion der Persistenz
- Wiederverwendbarkeit von Code
  - In verschiedenen Apps und Modulen (Library, Framework)
  - Auf beiden Plattformen (z.B. Kotlin Native)
- Wiederverwendbarkeit von ganzen Modulen in verschiedenen Apps

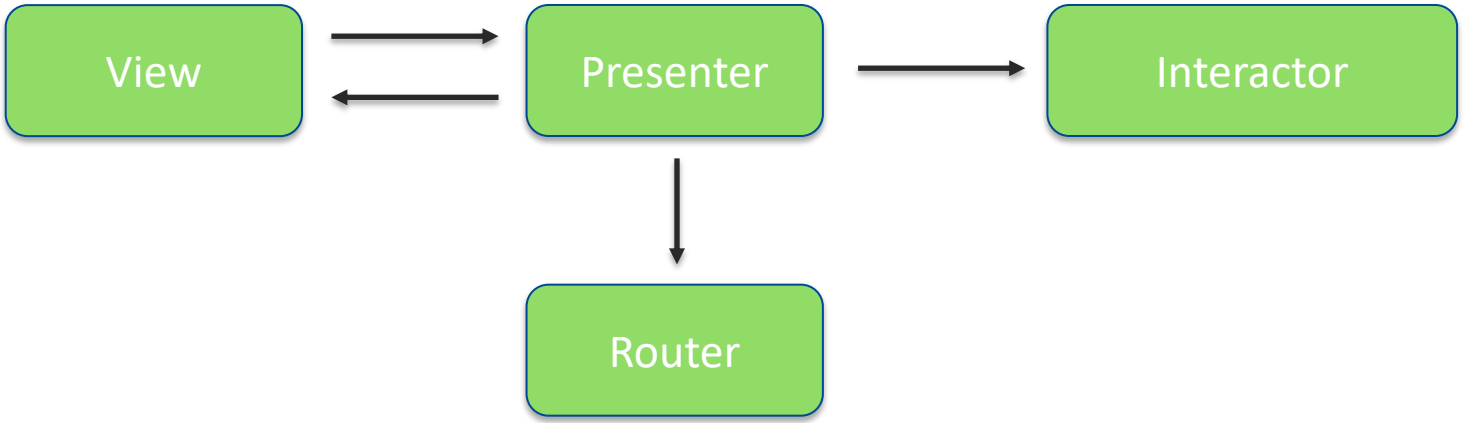
Architektur

UI-Patterns

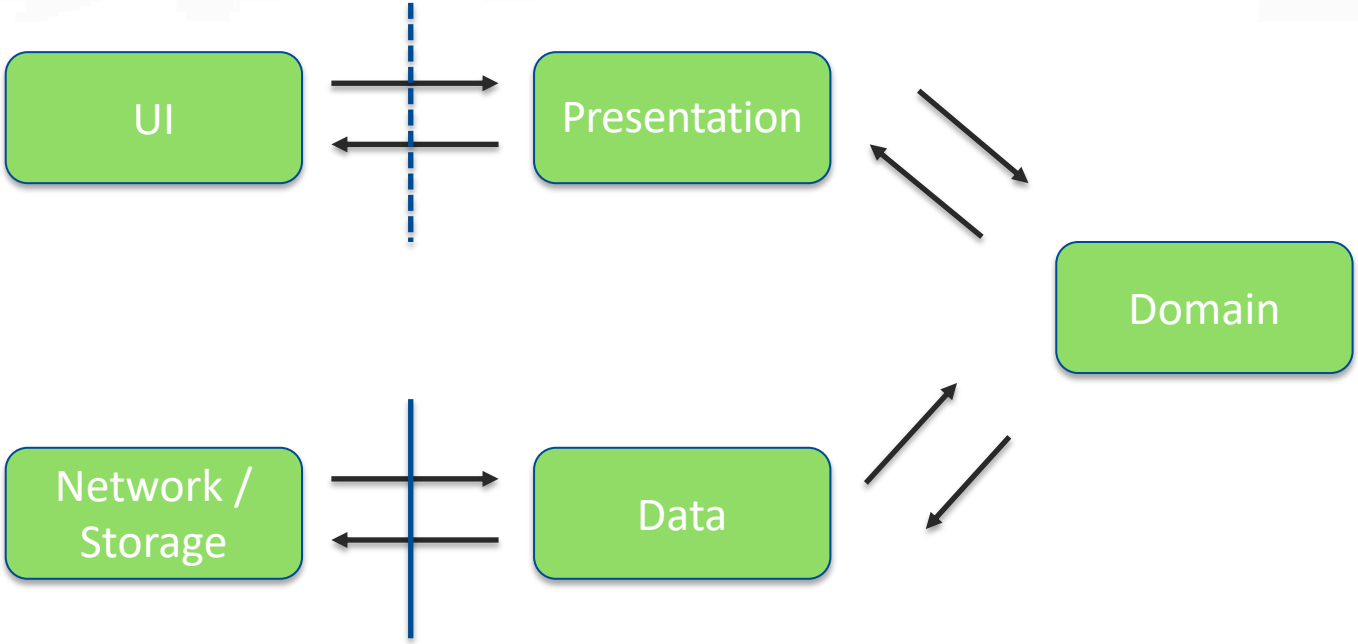
- UI-Module bestehend aus:
  - View - z.B. Fragment, UIViewController
  - View-Logik - z.B. ViewModel, Presenter
  - Datenquelle - z.B. Service, Interactor
  - Router
- Einzelne UI-Module sind voneinander unabhängig
- Gemeinsamkeiten in Code und Styling werden extrahiert

Architektur

Elemente eines UI-Moduls



Integrationstests



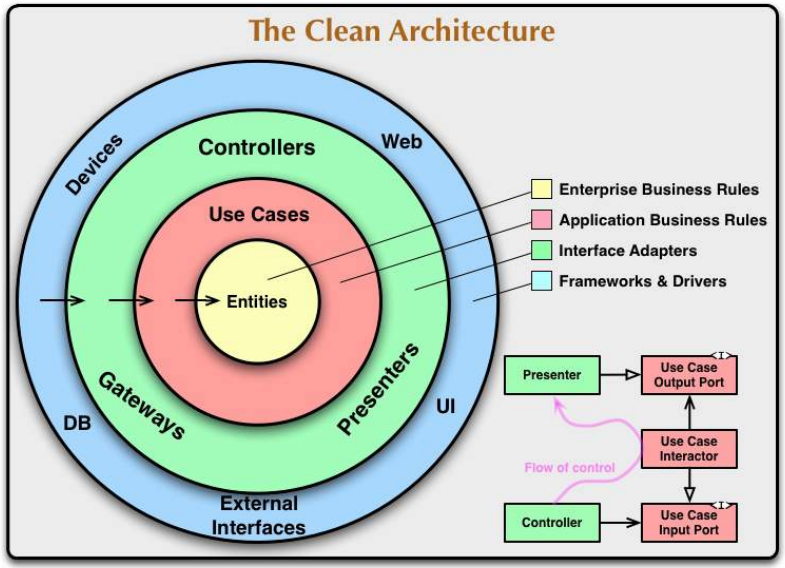
Architektur

Durch UI-Patterns erreichte Ziele

- Testbarkeit
- Erweiterbarkeit
- Wiederverwendbarkeit
- Abstraktion

Architektur

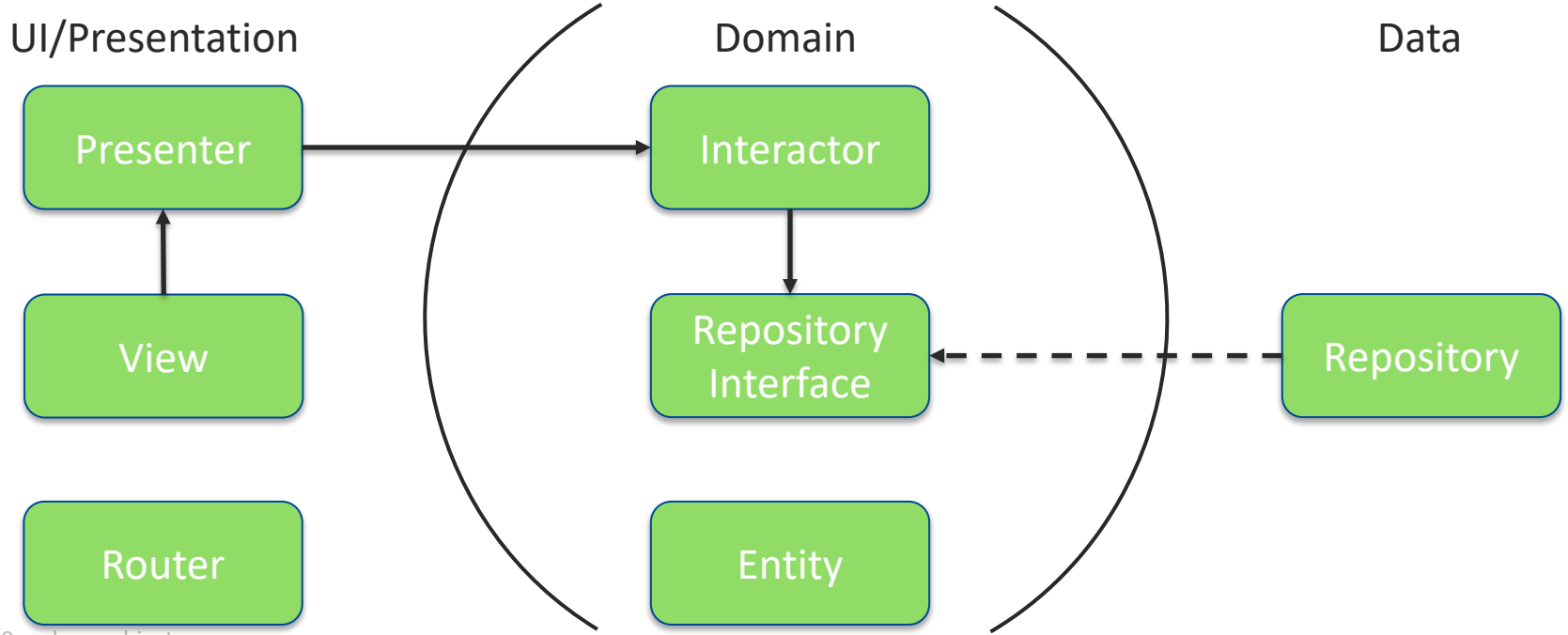
Clean Architekture



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

Architektur

Clean Architecture mit VIPER



Architektur

Durch Clean Architecture erreichte Ziele

- Testbarkeit
- Abstraktion der UI
- Leichtere Zusammenarbeit
- Wiederverwendbarkeit

Continuous Integration / Deployment

- Releasefähiges Artefakt ist jederzeit verfügbar
- Schnelles Feedback reduziert Risiken
  - Code kompiliert
  - Tests sind grün
  - Stakeholder können den Entwicklungsstand auf ihrem Smartphone installieren
  - Kompatibilität mit neuen OS-Versionen, verschiedenen Gerätetypen
- Merge-Konflikte fallen kleiner aus und sind leichter zu lösen

Continuous Integration / Deployment

Bestandteile einer Pipeline

- Repository
- Unit-Tests
- Integrationstests auf Simulator/Emulator
- Artefakt bauen
- App Signing
- Deployment
  - Google Play Store, Google Beta
  - Apple App Store, Testflight
  - Eigene Lösung

Continuous Integration / Deployment

verschiedene Werkzeuge

- Azure DevOps
- Fastlane
- Jenkins
- Firebase Test Lab

Anhang

main.swift

Damit main.swift verwendet wird, muss “@UIApplicationMain” von AppDelegate entfernt werden.

```
import Foundation
import UIKit

private func getIntegrationTestDelegate() -> AnyClass? {
  return NSClassFromString("MyIntegrationTestTarget.MyIntegrationTestAppDelegate")
}

private func isUnitTest() -> Bool {
  return NSClassFromString("XCTest") != nil
}

private func getDelegateClassName() -> String? {
  if let integrationTestDelegate = getIntegrationTestDelegate() {
    return NSStringFromClass(integrationTestDelegate)
  }

  if isUnitTest() {
    return nil
  }

  return NSStringFromClass(AppDelegate.self)
}

UIApplicationMain(
  CommandLine.argc,
  CommandLine.unsafeArgv,
  nil,
  getDelegateClassName()
)
```

Anhang

Häufig verwendete Bibliotheken

- Kotlin
  - Koin
  - mockK
- Kotlin/Java
  - Timber
  - Retrofit
  - RxJava
  - Threeten (JSR 310)
- Swift
  - Swinject
  - Nimble
  - SwiftyMocky
  - OHHTTPStubs
  - RxSwift
  - RxDataSources
  - TinyConstraints