앱 안정성 및 확장성 강화를 위한

안드로이드 아키텍처

-테스트 구현-

[Part 2. 테스트 구현 2](#_Toc116476729)

[1. 좋은 아키텍처를 위한 올바른 테스트 2](#_Toc116476730)

[2. Google은 어떻게 테스트하는가 5](#_Toc116476731)

[3. Testing Code Best Practices 6](#_Toc116476732)

[참고 링크 7](#_Toc116476733)

# Part 2. 테스트 구현

## 1. 좋은 아키텍처를 위한 올바른 테스트

1 왜 아키텍처에서 테스트가 중요한가

1.1 현실적인 필요성

-아키텍처를 변경하는 작업은 많은 코드 수정을 수반

-모든 작업을 QA에서 담당하면 비효율적

1.2 좋은 설계를 촉진

-테스트 케이스를 통해서 API를 변경했을 때의 사용성의 차이를 즉시 알 수 있음

1.3 코딩 생산성

-테스트가 FAIL 되지 않는 이상 내가 수정한 코드가 잘 움직인다는 심리적 안정성이 코드에 더 집중할 수 있게 해줌

-테스트를 안하면 개발 기간은 짧아지겠지만 언젠가 갚아야하는 부채로 남음. 이를 고려하면 실제로 테스팅 구현을 함께하는 쪽이 개발 시간이 짧음

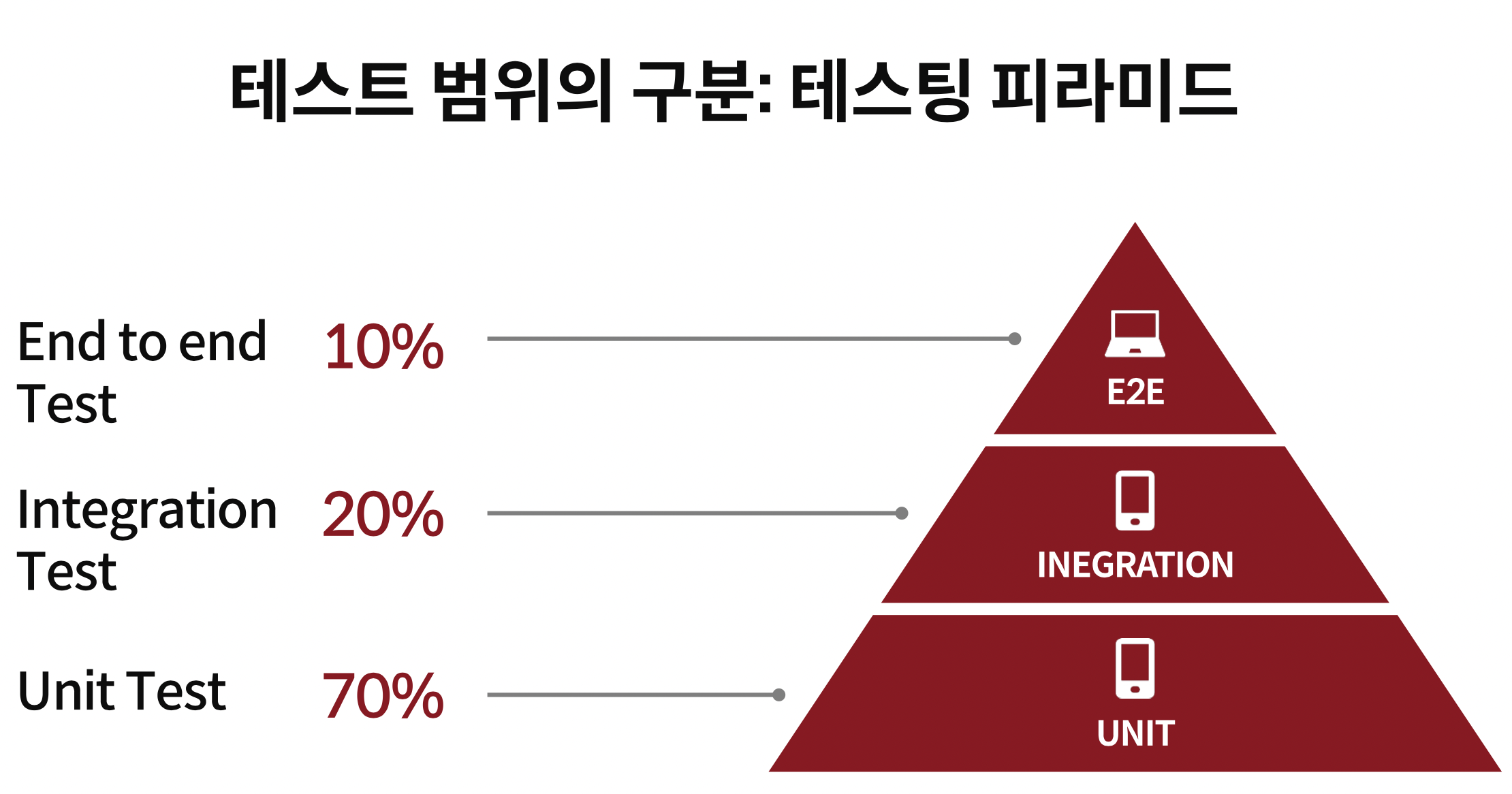
1.4 협업을 촉진

-테스트 케이스만 보면 특정 API의 기능과 의도, 올바른 사용법을 단번에 파악할 수 있기 때문에 문서로서의 기능

-코드 담장자가 아니더라도 코드를 수정할 수 있으며, 이 때 원저작자의 의도와 다른 방향으로 구현했다면 테스트도 실패

-더 효율적인 코드 리뷰 가능

2 테스트 범위의 구분: 테스팅 피라미드



[그림 2.1 테스팅 피라미드]

2.1 E2E

-크롬 브라우저를 띄운 다음, 내가 만든 검색 페이지로 들어가서 검색을 해보고 검색한 내용이 제대로 나오는지 화면상에서 확인하거나 직접 회원가입을 해보고 회원가입 후 로그인 되는지 직접 브라우저 상에 값을 입력해서 테스트 하는 방법

2.2 Intergrating test

-최소 두 개 이상의 클래스 또는 서브 시스템의 결합을 테스트하는 방법

2.3 Unit test

-코드로 코드를 테스트  
-Unit Testing이 가장 쉬우며 효과가 좋음  
-빠르고 비용이 상대적으로 낮기 때문에 개발할 때 필수적으로 작성해야 함.

ex) 프론트엔드 : jest(가장 많이 사용), enzyme / 백엔드: pytest, unittest(장고에서 기본적으로 사용)

2.4 TDD(Test Driven Development)

-테스트 주도 개발을 의미하며, 반복 테스트를 이용한 소프트웨어 방법론으로, 작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현

3 안드로이드 테스트의 종류

3.1 Local Test

-시뮬레이터나 디바이스가 아닌 JVM위에서 동작

-Junit 라이브러리 사용

cf. Local Test with androidx.test

-내부적으로 Robolectric 라이브러리(에뮬레이션 된 Context, Application, Activity)를 사용하는 JVM 테스트

-보통은 상당 수 unit test가 여기서 구현

3.2 Instrumented Test

-시뮬레이터나 디바이스에서 동작

-Espresso 라이브러리 사용

-안정성/호환성/성능 테스트는 디바이스에서 하는 걸 권장하며, 단위 테스트는 JVM에서 실행해야 함. 그리고 되지 않는 것은 시뮬레이터에서 실시하며, 디바이스에서 테스트를 하기 전에 먼저 이런 것들을 고려해볼 것

4 테스트 해야하는 것

-테스트는 실제 일어날 수 있는 에러를 예방하기 위한 것으로 의미 있는 테스트가 되어야 함

4.1 Edge case

-알고리즘이 처리하는 데이터 값이 알고리즘의 특성에 따른 일정한 범위를 넘을 경우에 발생하는 문제 케이스

ex) 분모가 0이되는 데이터를 인위적으로 넣어 테스트 진행

4.2 Corner case

-여러 가지 변수와 환경의 복합적인 상호작용으로 발생하는 문제 케이스

ex) 특정 데이터를 입력했을 때 디바이스A에서는 정상동작하지만 디바이스B에서는 오류가 발생하는 경우

같은 장치에서라도 시간이나 다른 환경에 따라 오류가 발생하기도하고 정상작동하기도 하는 경우

4.3 그 외

-모든 가능한 네트워크 에러 ex)404

-잘못된 데이터 에러ex) 포맷이 틀린 JSON 문자열

-저장소 오버플로

-중요 객체가 재생성 되는 상황 ex)설정 변경, 다크 모드 적용, 화면 회전에 따른 라이프사이클 초기화 등

5 테스트 코드 구현에 익숙하지 않은 경우

5.1 작은, 독립적인 부분부터 시작

-독립 함수(ex. 유틸리티 내부 계산 함수 등)에 대한 JVM 단위 테스트

5.2 큰 부분부터 시작

-핵심 시나리오에 대한 E2E 테스트. 단 테스트 요소들을 잘게 쪼갬

ex) 로그인 유즈케이스 -> 1.정상값 입력(->1.1저장 1.2전송 1.3UI갱신 등) 2.비정상값 입력

6. 외부 의존성은 어떻게 해결하는가?

-외부 의존성 예시 : SQLite, REST/gRPC call

-1순위: Real Code, 의존성 관계에 있는 진짜 코드를 사용

-2순위: Fake, 라이브러리에 의해 제공되는 표준 fake를 사용

-3순위 Mock/Spy/Stub, 위의 방법이 불가능 할 때

cf. 의존성 주입 라이브러리Hilt <https://developer.android.com/training/dependency-injection/hilt-testing>

## 2. Google은 어떻게 테스트하는가

1 Google에서 장애 발생 시 대처 순서

(1) Issue Tracker에 버그 보고

(2) 해당 Pull Request를 통째로 Roll back

(3) 보고된 버그를 재현할 수 있는 테스트 코드 작성(테스트 코드는 반드시 Fail되어야 정상)

(4) Pull Request 신청

-자동검사(\*test coverage 미달, 코드 스타일, 잠재적 취약성 등), 테스트 실행

-Code Review

-Merge

2 좋은 테스트의 조건

2.1 정확성(Correctness) : 검증 대상의 행위(behavior)가 실제 앱에서 동작할 것으로 기대되는 행위와 일치

2.2 명확성(Clarity) : 테스트 케이스는 코드의 사용법을 설명하듯이 구현되어야 하며 간결성(Conciseness)과 완결성(Completeness)을 보장해야 함

2.3 안정성(Resilience) : 연관되지 않은 메서드, 클래스에 변경사항이 발생해도 테스트에는 영향을 주지 않음

2.4 유용성(Helpfulness)

-실제 동작이 테스트에 반영

-mock오류, 설정 오류 등이 아닌 어떤 테스트 케이스가 실패한다면, 그것은 반드시 앱도 실패하는 것을 의미함

3 외부 의존성 처리의 원칙

-의존성은 할 수 있는 한 실제 코드를 사용해 테스트

-Network 처리 등 실행 속도가 느린 API의 경우 해당 모듈을 제공하는 개발팀에서 표준 fake구현도 제공

-위의 방법으로 불가능한 상황에서만 mock으로 테스트

->mock은 많이 쓰이지만 큰 의존성, 외부 시스템을 사용하는 경우 가급적이면 실제 코드나 fake를 사용

|  |
| --- |
| @DoNotMock(“Use SimpleQuery.create() instead of mocking”) //표준화 되어 있는 fake함수가 있으니 mock을 쓰지 마시오  abstract class Query {  abstract fun getQueryValue(): String  } |

4. 테스팅 나쁜 예시

-테스트 시간을 절약하기 위해 할 수 있는 한 많은 내용이 담긴 큰 테스트 케이스를 작성하는 경우

-View를 위한 테스트를 구현하지 않는 경우 -> 향후 성능 개선 등 다양한 이유로 리팩토링이 필요한데, 테스트가 없으면 생각하지 못한 부작용이 생김

-앱의 크래시를 막기 위해 테스트 추가보다는 그냥 catch로 삼키는 편을 택하는 경우

## 3. Testing Code Best Practices

1 테스트 코드의 대원칙 : 불변성

-테스트는 깨지기 쉽기 때문에(brittle test) 유지보수가 잘 되지 않음

-테스트 코드는 ‘순수한 리팩토링’, ‘새로운 feature 추가’, ‘버그 수정’ 등의 이유로 변경되어선 안됨

-테스트 코드 변경의 예외 : 행위의 변경(behavior changes)

2 Best Practice

2.1 Public API로 테스트, Test via Public APIs

|  |
| --- |
| fun processTransaction(transaction: Transaction) {      if (isValid(transaction)) {          saveToDatabase(transaction)      }  }  **private** fun isValid(transaction: Transaction): Boolean {      return transaction.amount < transaction.balance  }  **private** fun saveToDatabase(transaction: Transaction) {      val record = **"**${transaction.sender}**,** ${transaction.recipient}**,** ${transaction.amount}**"**  database.put(transaction.getId(), record)  } |

|  |
| --- |
| //Bad Practice  /\*  테스트를 위해 private 메서드를 public 메서드로 변경한 케이스로, 테스트를 위해 기존 디자인을 바꿈으로써 제작자의 의도와 엇나간 결과를 낼 수 있는 위험요소가 있음  \*/  @Test  fun emptyAccountShouldNotBeValid() {  val transaction = newTransaction().setSender(EMPTY\_ACCOUNT)      assertThat(**processor.isValid(transaction**)).isFalse()  }  @Test fun shouldSaveSerializedData() {  val transaction = newTransaction() .setId(123).setSender("me").setRecipient("you").setAmount(100)  **processor.saveToDatabase(transaction)**  assertThat(database.get(123)).isEqualTo("me,you,100")  } |

|  |
| --- |
| //Good Practice  @Test  fun shouldNotPerformInvalidTransactions() {      //set      processor.setAccountBalance("me", 50)      processor.setAccountBalance("you", 20)      //process      val transaction = newTransaction().setSender("me").setRecipient("you").setAmount(100)      processor.processTransaction(transaction)        //assert      assertThat(processor.getAccountBalance("me")).isEqualTo(50)      assertThat(processor.getAccountBalance("you")).isEqualTo(20)  } |

2.2 상호작용하지 않고 상태 테스트, Test State, Not Interactions

-테스트의 결과는, 또 다른 조작을 통해서 Pass/Fail 이 검증되어서는 안됨

|  |
| --- |
| //Bad  @Test  //DB에 같은 유저를 넣어보면서 테스트. Fail이면 이전 코드가 정상 동작이 검증되는 코드로, 또 다른 조작으로 또 다른 테스트를 하는 Bad case fun shouldWriteToDatabase() {    accounts.createUser("foobar")    verify(database).put("foobar")  }  //Good  @Test  //accounts 내부에서 유저가 나오는 것으로 코드를 검증 fun shouldCreateUsers() {    accounts.createUser("foobar")    assertThat(accounts.getUser("foobar")).isNotNull()  } |

2.3 완결성과 간결성에 중점을 둔 테스트, Make Your Tests Complete and Concise

2.3.1 완결성(Completeness)

-테스트 코드는 테스트하고자 하는 것을 정확히 알 수 있는 정보를 모두 갖고 있어야하며, 테스트 케이스 하나만 보면 어떤 테스트인지 그리고 기대치가 뭔지 알 수 있음

ex) assert 유저 뽑아내기 //목적과 기대치를 알 수 있음

2.3.2 간결성(Concise)

-불필요한 내용은 감춤

ex) 객체를 만들기 위해 필요한 파라미터 20개, 테스트랑 연관된 파라미터는 5개인 상황

->15개 파라미터를 수단 방법을 가리지 않고 감춤

->테스트 가독성 UP, 단 감춰진 파라미터는 테스트에 영향을 미치면 안됨

|  |
| --- |
| //BAD  @Test fun shouldPerformAddition() {    val calculator = Calculator(RoundingStrategy(),"unused", ENABLE\_COSINE\_FEATURE, 0.01, calculusEngine, false) //테스트에 불필요한 파라미터    val result = calculator.calculate(newTestCalculation())  assertThat(result).isEqualTo(5) // Where did this number come from?, 테스트 조건은?, 테스트 기대치가 5인데 왜?   }  //GOOD  @Test fun shouldPerformAddition() {    val calculator = newCalculator()   val result = calculator.calculate(newCalculation(2, Operation.PLUS, 3))    assertThat(result).isEqualTo(5)  } |

2.4 Test Behavior, Not Methods

2.5 Structure tests to emphasize behaviors

2.6 Don’t Put Logic in Tests

2.7 DAMP, Not Dry

2.8 No Shared Value

2.9 Shared Setup

# 참고 링크

Test Coverage

의존성 주입 라이브러리 Hilt

<https://developer.android.com/training/dependency-injection/hilt-testing>

테스트 코드의 완결성과 간결성

<https://testing.googleblog.com/2014/03/testing-on-toilet-what-makes-good-test.html>