II.

보다 나은 아키텍처를 위한 테스트 구현

강 사 룡

첫 소개에서 언급했듯 새로운 아키텍처 적용 혹은 기존 아키텍처의 변경 과정은 상당한 양의 회귀 오류를 수반할 수 있습니다. 이를 일일이 사람이 검증하는 것은 매우 비효율적이므로 설계 변경은 반드시 잘 작성된 자동화 테스트 구조 안에서 이뤄질 필요가 있습니다. 또한 잘 설계된 코드는 테스트하기 쉬운 코드이기도 합니다. 본격적으로 아키텍처 구현에 들어가기 전에 반드시 알아야 할 자동화 테스트 구현을 배워봅시다.

TABLE OF CONTENT.

좋은 아키텍처를 위한 올바른 테스트.

왜 아키텍처에서 테스트가 중요한가

현실적인 필요성	• "QA doesn't scale" • QA가 보다 더 생산적으로 일할 수 있다
좋은 설계를 촉진	• single responsibility • test case를 통해서 API를 변경했을 때의 사용성의 차이를 즉시 알 수 있다
코딩 생산성	• 확신을 갖고 작업할 수 있음: "테스트가 fail 되지 않은 한 내가 수정한 코드가 잘 움직이고 있다" 심리적 안정성이 코드에 더 집중할 수 있게 해줌 • 실제로 테스팅 구현을 함께 하는 쪽이 개발 시간이 짧음

왜 아키텍처에서 테스트가 중요한가

협업을 촉진

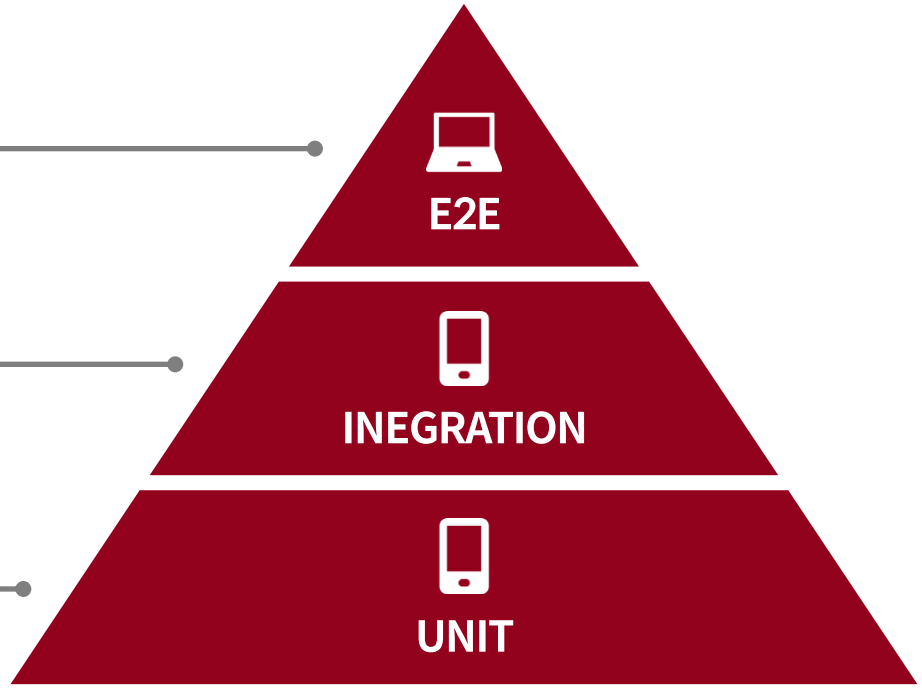
- 문서로서의 테스트 코드: 테스트 케이스만 보면 특정 API의 기능과 의도, 올바른 사용버을 단번에 파악 가능
- 코드 담당자(owner)가 아니더라도 코드를 수정할 수 있음 (원저작자의 의도와 다른 방향으로 구현했다면 테스트도 실패)
- 더 효율적인 code review

테스트 범위의 구분: 테스팅 피라미드

End to end Test 10%

Integration Test 20%

Unit Test 70%



출처: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

Android 테스트의 종류: 로컬 테스트(Local Test)

로컬 테스트 (test 폴더)

- JVM 위에서 동작
- JUnit 라이브러리 사용

로컬 테스트 with androidx.test

- 내부적으로 Robolectric 라이브러리를 사용하는 JVM 테스트
  - 에뮬레이션 된 Context, Application, Activity 등을 제공
- 보통은 상당수 unit test가 여기서 구현됨
- Big local test: integration test도 여기서 구현될 수 있음

Android 테스트의 종류: Instrumented Test

Instrumented test (androidTest 폴더)

- 시뮬레이터 혹은 실제 장치에서 동작
- Espresso 라이브러리를 함께 사용
- Small instrumented test: 단위가 여기에 구현될 수도 있음

When to use real device for testing

- DO - 안정성 / 호환성 테스트, 성능 테스트
- DON'T - 단위 테스트: 할 수 있는 한 JVM에서 실행해야 함. 그리고 되지 않는 것은 시뮬레이터에서. 실 기기 테스트를 하기 전에 먼저 이런 것들을 고려해볼 것

무엇을 테스트 해야 하나?

대원칙: 테스트는 실제 일어날 수 있는 에러를 예방하기 위한 것

→ 의미 있는 테스트가 되어야 함

좋은 예: Edge cases

- Boundary condition
- 모든 가능한 네트워크 에러
- 잘못된 데이터 (예: 포맷이 틀린 json 문자열)
- 저장소 오버플로
- 중요 객체가 재생성 되는 상황 (흔한 예: configuration change)

Baby steps
: 테스트 코드 구현에 익숙하지 않다면? (1단계)

작은, 독립적인 부분부터 시작

- 독립 함수에 대한 JVM 단위 테스트
- 예: 계산/변환 로직을 가진 클래스
- 참고 서적: 테스트 주도 개발(TDD by Example)

큰 부분부터 시작

- 핵심 시나리오(Critical User Journey)에 대한 E2E test
- 주의: 변경이 많지 않거나, 변경이 된다는 사실 자체가 서비스에 중요한 곳들에 한정 → 그러지 않으면 깨지기 쉬운(brittle) 테스트를 양산하게 됨

Baby steps
: 테스트 코드 구현에 익숙하지 않다면? (2단계)

실제/의사 디버깅 과정에서 테스트 코드를 적용

외부 의존성은 어떻게 해결하나?

예: SQLite, REST/gRPC call

Real code > Fake >> Mock/Spy/Stub

- 1순위: 의존성 관계에 있는 진짜 코드를 사용 - 예: in-memory DB
- 2순위: 라이브러리에 의해 제공되는 표준 fake를 사용
- 3순위: 위의 방법이 불가할 때, mock 사용

“Prefer realism over isolation”

Hilt! - d.android.com/training/dependency-injection/hilt-testing

그외 테스트 작성 팁

- ✓ type-safe한 matcher를 적극 활용할 것 : hamcrest, truth 등 + built-in matcher
- ✓ interaction보다 state를 체크할 것 (다음 강의 참조)
- ✓ 필요시 shared code를 적절히 사용할 것 (다음 강의 참조)
- ✓ Android API를 mocking하지 말 것: 특히 Context
- ✓ Robolectric! via androidx.test

  Fragment 독립 생성, Life Cycle 제어 등 많은 개선이 있었음

Google은
어떻게 테스트하는가.

Google Scale

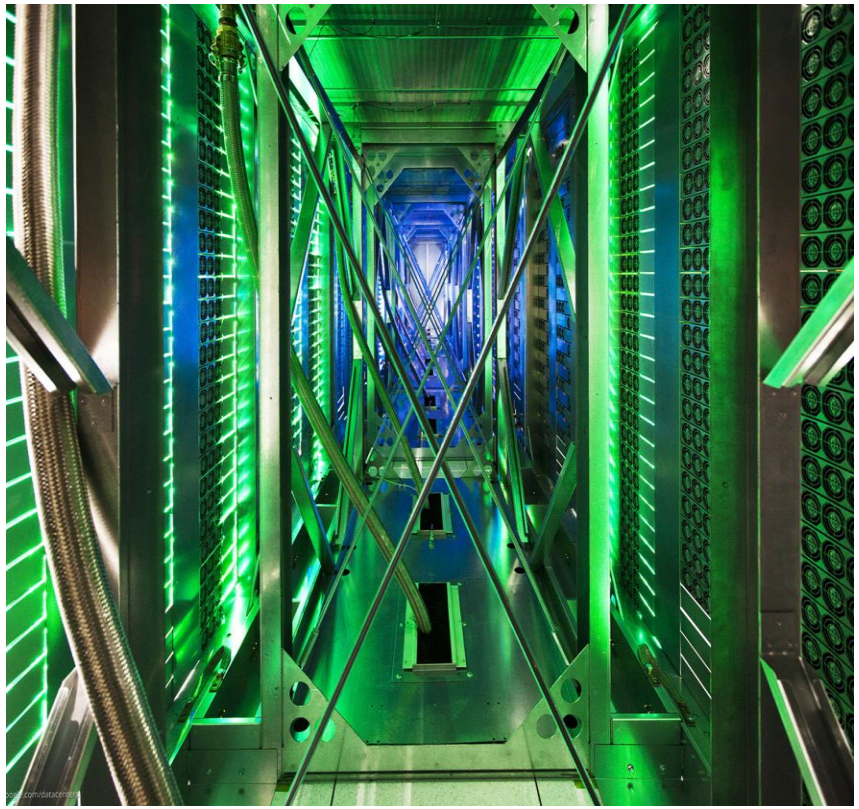
30,000+ engineers

60,000 changes per day

2 billion lines of code

9 million source files

500 million test cases per day



실패 사례: Google Web Server in 2005

거의 테스트 코드가 없었음

- 특정 시점에서는 무려 80% 이상의 PR이 버그로 인해 롤백되었음
- 개발자들이 기능을 수정할 때 잘 동작하리라는 확신이 없어짐

이 문제를 분석한 결과, 자동화 테스트만이 해결책이라는 결론

- 코드 변경시에는 반드시 테스트를 함께 구현하도록 정책 변경
- 1년 뒤 → 긴급 패치 숫자가 절반으로 감소
- 현재 GWS 단일 시스템에서만 수만 개의 테스트 케이스

Google에서 장애가 발생하면?

1. Issue Tracker에 버그 보고
2. 해당 Pull Request를 통째로 roll back
3. **보고된 버그를 재현할 수 있는 테스트 코드 작성**
   (물론 테스트 케이스는 fail)
4. 테스트 코드를 pass 할 수 있는 코드 작성
5. Pull Request 신청 (이하는 통상 PR 처리 수순)
   a. 자동 검사 (test coverage 미달, 코드 스타일, 잠재적 취약성 등), 테스트 실행
   b. Code Review
   c. Merge

Google의 테스트 정책

높은 Test Coverage*

→ *여기서 테스트 커버리지는 오직 small-sized test로만 측정함

Beyoncé Rule

→ “If you liked it, then you should put a test on it.”

좋은 테스트의 조건

정확성 (Correctness)

검증 대상의 행위(behavior)가 실제 앱에서 동작할 것으로 기대되는 행위와 일치

명확성 (Clarity)

테스트 케이스는 코드의 사용법을 설명하듯이 구현되어야 한다
→ 간결성(Conciseness)과 완결성Completeness 보장

연관되지 않은 변경사항들에 대한
안정성 (Resilience)

목적 혹은 행위가 변경되지 않는 한, 테스트도 변경하지 않아야 함.

좋은 테스트의 조건

유용성 (Helpfulness)

- 실제의 동작이 테스트에 반영
- 어떤 테스트 케이스가 실패한다면, 그것은 반드시 앱도 실패한다는 것을 의미해야 함
  (mock의 오류, 설정 오류, 등등이 아니라)

외부 의존성 처리의 원칙

1. 의존성은 할 수 있는 한 실제 코드를 사용해서 테스트
2. network등 실행 속도가 느린 API의 경우, 해당 모듈을 제공하는 개발팀에서 표준 fake 구현도 제공
3. 위 방법으로 불가능한 상황에서만 mock으로 테스트

```
@DoNotMock("Use SimpleQuery.create() instead of mocking.")

abstract class Query {

  abstract fun getQueryValue(): String

}
```

More complicated test

Functional test with..	• UI action • Network faking - hermetic test, less flaky • Network request validation - 사용자에게 보이지 않는 결과들을 검증 • UI 변경에도 테스트가 수정될 필요 없도록 구성. Network, multimedia playback 등에 대한 fake sync도 구현
E2E test tool based on script	• Eg. Firebase Test Lab • 핵심 사용자 시나리오 (Critical User Journey)를 high-level 차원에서 테스트함 - 주로 PM이 작성해서 관리하는 경우가 많음
Snapshot Test	깨지기 쉬운 테스트가 될 가능성이 농후하므로, 목적 설정이 아주 중요

테스팅 안티패턴

- ✓ 느린 테스트 시간을 절약하기 위해 할 수 있는 한 많은 내용이 담긴 큰 테스트 케이스를 작성한다
- ✓ View를 위한 테스트를 구현하지 않는다
  - 향후 성능 개선 등 다양한 이유로 리팩토링이 필요한데, 테스트가 없으면 생각하지 못한 부작용이 생김
- ✓ 앱의 크래시를 막기 위해 테스트 추가보다는 그냥 catch로 삼키는 편을 택한다

Testing Code Best Practices.

대원칙 #1: 테스트 불변성(Unchanging Test)

왜 테스트 코드는 유지보수가 잘 안 되는가?

→ 깨지기 쉬운 테스트(brittle test) 때문

테스트 코드는 아래와 같은 이유로 변경되어선 안 된다:

- 순수한 리팩토링
- 새로운 feature 추가
- 버그 수정

유일한 예외: 행위의 변경 (behavior changes)

Best Practice #1 Test via Public APIs

테스트 코드는 대상 API의 사용자와 동일한 조건을 가진다

→ 즉, 반드시 public 메소드만 호출해야 한다

테스트 코드에서 불려지기 위한 목적만으로
private/protected 메소드를 public으로 바꿔서는 안 된다

→ 원 소스의 의도를 깨뜨리게 되므로

Best Practice #1 Test via Public APIs

```
fun processTransaction(transaction: Transaction) {
  if (isValid(transaction)) {
    saveToDatabase(transaction)
  }
}

private fun isValid(t: Transaction): Boolean =
  return t.amount < t.sender.balance

private fun saveToDatabase(t: Transaction) {
  val s = "${t.sender}, ${t.recipient()}, ${t.amount()}"
  database.put(t.getId(), s)
}
```

Best Practice #1 Test via Public APIs

```
fun processTransaction(transaction: Transaction) {
  if (isValid(transaction)) {
    saveToDatabase(transaction)
  }
}

private fun isValid(t: Transaction): Boolean =
  return t.amount < t.sender.balance

private fun saveToDatabase(t: Transaction) {
  val s = "${t.sender}, ${t.recipient()}, ${t.amount()}"
  database.put(t.getId(), s)
}
```

BAD

```
@Test
fun emptyAccountShouldNotBeValid() {
  assertThat(processor.isValid(newTransaction().setSender(EMPTY_ACCOUNT)))
      .isFalse()
}
@Test
fun shouldSaveSerializedData() {
  processor.saveToDatabase(newTransaction()
      .setId(123)
      .setSender("me")
      .setRecipient("you")
      .setAmount(100))
  assertThat(database.get(123)).isEqualTo("me,you,100")
}
```

GOOD

```
@Test
fun shouldTransferFunds() {
  processor.setAccountBalance("me", 150)
  processor.setAccountBalance("you", 20)

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100))

  assertThat(processor.getAccountBalance("me")).isEqualTo(50)
  assertThat(processor.getAccountBalance("you")).isEqualTo(120)
}
```

GOOD

```
@Test
fun shouldNotPerformInvalidTransactions() {
  processor.setAccountBalance("me", 50)
  processor.setAccountBalance("you", 20)

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100))

  assertThat(processor.getAccountBalance("me")).isEqualTo(50)
  assertThat(processor.getAccountBalance("you")).isEqualTo(20)
}
```

Best Practice #2
Test State, Not Interactions

테스트의 결과는, 또 다른 조작을 통해서
통과(pass)/실패(fail)가 검증되어서는 안 된다

```
// Bad

@Test
fun shouldWriteToDatabase() {
  accounts.createUser("foobar")
  verify(database).put("foobar")
}
```

GOOD

```
@Test
fun shouldCreateUsers() {
  accounts.createUser("foobar")
  assertThat(accounts.getUser("foobar")).isNotNull()
}
```

Best Practice #3
Make Your Tests Complete and Concise

테스트 코드는,

- ✓ 테스트하고자 하는 것을 정확히 알 수 있는 정보를 모두 갖고 있어야 한다
  → 완결성: completeness
- ✓ 반대로 불필요한 내용은 감춰야 한다 → 간결성: conciseness
- ✓ 더 자세한 설명: https://testing.googleblog.com/2014/03/testing-on-toilet-what-makes-good-test.html

BAD

```
@Test
fun shouldPerformAddition() {
  val calculator = Calculator(RoundingStrategy(),
      "unused", ENABLE_COSINE_FEATURE, 0.01, calculusEngine, false)

  val result = calculator.calculate(newTestCalculation())
  assertThat(result).isEqualTo(5) // Where did this number come from?
}
```

GOOD

```
@Test
fun shouldPerformAddition() {
  val calculator = newCalculator()
  val result = calculator.calculate(newCalculation(2, Operation.PLUS, 3))
  assertThat(result).isEqualTo(5)
}
```

Best Practice #4
Test Behaviors, Not Methods

보통 메소드와 테스트 케이스는 1:N

커버리지에만 신경을 쓸 경우에도 이런 문제가 생길 수 있음

흔히 보이는 안티 패턴

- 1:1 - 하나의 테스트 케이스에, 한 개의 메소드의 여러 가지의 검증 조건을 한 번에 넣는다
- M:1 - 특정 기능 검증을 위해 여러 메소드를 한 번에 검증한다 (항상 나쁜 것은 아님)

Code to be tested

```
fun displayTransactionResults(user: User, transaction: Transaction) {
  ui.showMessage("You bought a " + transaction.itemName)
  if (user.balance < LOW_BALANCE_THRESHOLD) {
    ui.showMessage("Warning: your balance is low!")
  }
}
```

Bad: test methods

```
@Test
fun testDisplayTransactionResults() {
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      Transaction("Some Item", dollars(3)))

  assertThat(ui.getText()).contains("You bought a Some Item")
  assertThat(ui.getText()).contains("your balance is low")
}
```

Good: test behaviors

```
@Test
fun displayTransactionResults_showsItemName() {
  transactionProcessor.displayTransactionResults(
      User(), Transaction("Some Item"))
  assertThat(ui.getText()).contains("You bought a Some Item")
}

@Test
fun displayTransactionResults_showsLowBalanceWarning() {
  transactionProcessor.displayTransactionResults(
      newUserWithBalance(
          LOW_BALANCE_THRESHOLD.plus(dollars(2))),
      Transaction("Some Item", dollars(3)))
  assertThat(ui.getText()).contains("your balance is low")
}
```

Best Practice #5
Structure tests to emphasize behaviors

테스트 케이스의 3요소가 명확히 분리되는 것이 좋음

GIVEN

테스트 조건 / 환경 / 기본 상태

WHEN

테스트 대상 행위

THEN

의도한 결과

Good

```
@Test
fun transferFundsShouldMoveMoneyBetweenAccounts() {
  // Given two accounts with initial balances of $150 and $20
  val account1 = newAccountWithBalance(usd(150))
  val account2 = newAccountWithBalance(usd(20))

  // When transferring $100 from the first to the second account
  bank.transferFunds(account1, account2, usd(100))

  // Then the new account balances should reflect the transfer
  assertThat(account1.getBalance()).isEqualTo(usd(50))
  assertThat(account2.getBalance()).isEqualTo(usd(120))
}
```

Another good example

```
@Test
fun shouldTimeOutConnections() {
  // Given two users
  val user1 = newUser()
  val user2 = newUser()
  // And an empty connection pool with a 10-minute timeout
  val pool = newPool(Duration.minutes(10))
  // When connecting both users to the pool
  pool.connect(user1)
  pool.connect(user2)
  // Then the pool should have two connections
  assertThat(pool.getConnections()).hasSize(2)
  // When waiting for 20 minutes
  clock.advance(Duration.minutes(20))
  // Then the pool should have no connections
  assertThat(pool.getConnections()).isEmpty()
  // And each user should be disconnected
  assertThat(user1.isConnected()).isFalse()
  assertThat(user2.isConnected()).isFalse()
}
```

Best Practice #6
Don’t Put Logic in Tests

테스트 코드가 대상이 되는 코드와
동일한 로직을 공유하지 않도록 유의해야 한다

대상 코드와 다른 로직을 쓰는 게 아니라
할 수 있는 한 로직을 제거해야 한다

Bad

```
@Test
fun shouldNavigateToAlbumsPage() {
  val baseUrl = "http://photos.google.com/"
  val nav = Navigator(baseUrl)
  nav.goToAlbumPage()
  assertThat(nav.getCurrentUrl()).isEqualTo(baseUrl + "/albums")
}
```

Good

```
@Test
fun shouldNavigateToPhotosPage() {
  val nav = Navigator("http://photos.google.com/");
  nav.goToPhotosPage()
  assertThat(nav.getCurrentUrl()))
      .isEqualTo("http://photos.google.com//albums") // Oops!
}
```

Best Practice #7 DAMP, Not DRY

DRY (Don't Repeat Yourself)

→ 같은 코드를 반복하지 말라 → 가장 일반적인 조언

DAMP (Descriptive And Meaningful Phrases)

→ 반복 코드를 함수로 만들어서 재사용한 결과가,
오히려 지금까지 언급한 원칙을 해쳐서는 안 된다

Bad

```
@Test
fun shouldAllowMultipleUsers() {
  val users = createUsers(false, false)
  val forum = createForumAndRegisterUsers(users)
  validateForumAndUsers(forum, users)
}
@Test
public void shouldNotAllowBannedUsers() {
  val users = createUsers(true)
  val forum = createForumAndRegisterUsers(users)
  validateForumAndUsers(forum, users)
}

// Lots more tests...

private fun createUsers(boolean... banned): List<User> {
  // ...
}
// ...
```

Good

```
@Test
fun shouldAllowMultipleUsers() {
  val user1 = newUser().setState(State.NORMAL).build()
  val user2 = newUser().setState(State.NORMAL).build()

  val forum = Forum()
  forum.register(user1)
  forum.register(user2)

  assertThat(forum.hasRegisteredUser(user1)).isTrue()
  assertThat(forum.hasRegisteredUser(user2)).isTrue()
}
```

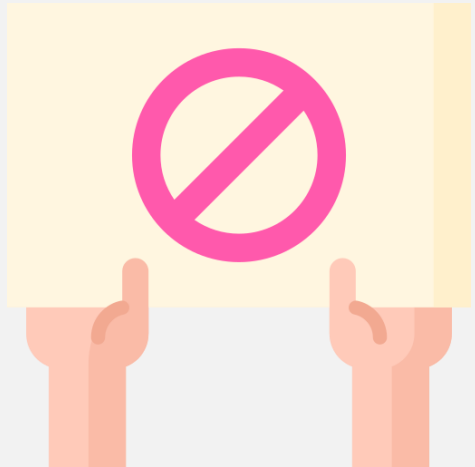
Good

```
@Test
fun shouldNotRegisterBannedUsers() {
  val user = newUser().setState(State.BANNED).build()

  val forum = Forum()
  try {
    forum.register(user)
  } catch(ignored: BannedUserException) {}

  assertThat(forum.hasRegisteredUser(user)).isFalse()
}
```

Best Practice #8 No Shared Value

테스트 케이스 간의 상태를 공유하면 안 됨

객체내 필드 수준에서의 불변성(immutability)을 제공하지 않은
Kotlin / Java에서 특히 위험



Bad

```
private val ACCOUNT_1 = Account.newBuilder()
    .setState(AccountState.OPEN).setBalance(50).build()
private val ACCOUNT_2 = Account.newBuilder()
    .setState(AccountState.CLOSED).setBalance(0).build()
private val ITEM = Item.newBuilder()
    .setName("Cheeseburger").setPrice(100).build()
```

Hundreds of lines of other tests...

```
@Test
fun canBuyItem_returnsFalseForClosedAccounts() {
  assertThat(store.canBuyItem(ITEM, ACCOUNT_1)).isFalse()
}
```

Good

```
private fun newContact(): Contact.Builder =
  Contact.newBuilder()
    .setFirstName("Grace")
    .setLastName("Hopper")
    .setPhoneNumber("555-123-4567")

@Test
fun fullNameShouldCombineFirstAndLastNames() {
  val contact = newContact()
      .setFirstName("Ada")
      .setLastName("Lovelace")
      .build()
  assertThat(contact.getFullName()).isEqualTo("Ada Lovelace")
}
```

Best Practice #9 Shared Setup

값이 아닌, 설정을 공유하라

예를 들어 초기 상태 설정을 위한 헬퍼들을 만드는 것은 적극 권장됨

Bad

```
private lateinit var nameService: NameService
private lateinit var userStore: UserStore

@Before
fun setUp() {
  nameService = NameService()
  nameService.set("user1", "Donald Knuth")
  userStore = UserStore(nameService)
}
```

Hundreds of lines of other tests...

```
@Test
fun shouldReturnNameFromService() {
  val user = userStore.get("user1")
  assertThat(user.getName()).isEqualTo("Donald Knuth")
}
```

Good

```
private lateinit nameService: NameService
private lateinit userStore: UserStore

@Before
fun setUp() {
  nameService = NameService()
  nameService.set("user1", "Donald Knuth")
  userStore = UserStore(nameService)
}

@Test
fun shouldReturnNameFromService() {
  nameService.set("user1", "Margaret Hamilton")
  val user = userStore.get("user1")

  assertThat(user.getName()).isEqualTo("Margaret Hamilton")
}
```

Good example of helper method

```
fun assertUserHasAccessToAccount(user: User, account: Account) {
  for (long userId : account.getUsersWithAccess()) {
    if (user.id == userId) {
      return
    }
  }
  fail("${user.name} cannot access ${account.name}")
}
```

References

Android에서 앱 테스트	d.android.com/training/testing
Google 코드랩: Testing Basics	d.android.com/codelabs/advanced-android-kotlin-training-testing-basics
Google Testing Blog	https://testing.googleblog.com/
TotT (Testing on the Toilet)	https://testing.googleblog.com/search/label/TotT

수고하셨습니다!