안드로이드 테스트 수행 전 알아야 할 것

TDD 순서

1. 실패테스트 작성
2. 합격테스트 작성 : 기능을 정의하는 통과 테스트의 작성
3. 코드 작성 : 테스트를 통과시킬 수 있는 코드의 작성
4. 테스트 실행 : 모든 테스트가 통과
5. 리팩토링 코드 : 중복성 제거

(출처 https://medium.com/mobility/why-developers-scared-to-refactor-code-47efd1b854e7)

TDD정의

간단 정의테스트를 먼저 작성하고, 테스트를 통과시키는 코드를 구현한 후 리팩토링하는 절차를 거친다.
(출처 https://gist.github.com/benelog/903e21cad1624263fd7825131f67ff82)

테스트 코드란?

- 단순히 말하면 '검증'을 위한 코드.
- Mock : 테스트에 쓰이는 가짜 객체 (Mockito, JMock, PowerMock 등이 있다.)
    - Mock Object를 사용하면 일반적인 방법으로 Instance로 만들 수 없는 Abstract class 나 Interface를 Test 할 수 있다.
- Unit 으로 하는 테스트가 모두 유닛 테스트는 아니다. 폭넓게 테스트 코드라고 볼 수 있다.
- 테스트 코드 작성 자체를 'TDD를 한다' 라고 볼 수는 없다. TDD는 테스트를 작성하는 하나의 방식일 뿐이다.

테스트 코드를 만드는 이유

- 디버깅의 편의를 위해 : 테스트 코드 작성이 능숙해지면 실제 애플리케이션을 실행하고 수동으로 반복 테스트하는 것보다 **훨씬 더 빠르고 정교하게 코드의 동작을 확인하고** 오류를 수정할 수 있다.
- **설계를 개선**할 수 있다 : 테스트 하기 쉬운 코드는 역할과 책임이 잘 나눠져 있는 코드이다. 이러한 코드는 재활용과 기능 추가, 버그 개선에 편하다. 테스트를 의식하며 코드를 작성할 때 이러한 구조의 코드 작성에 도움이 될 수 있다.
- 테스트 자체가 동작하는 예제이자 명세 : 다른 사람의 애플리케이션이나 라이브러리 전체를 실행시키지 않아도 코드가 실행된 결과를 이해할 수 있다.
- **반복적으로 수행할 회귀테스트의 자동화** : 기능을 추가하거나 개선할 때 시간을 아껴준다.
- 개발 작업에 더 집중 가능 : 테스트를 통과한다는 명확한 목표가 있고 이를 빠른 시점에 명확한 신호로 알려줄 수 있으며 이를 통해서 작업의 난이도와 간격을 스스로 적당하게 조절할 수 있다.

(출처 https://gist.github.com/benelog/903e21cad1624263fd7825131f67ff82)

안드로이드 테스트의 종류

1. Unit 테스트

- 일반적으로 코드의 유닛 단위 (메소드, 클래스, 컴포넌트) 의 기능을 실행하는 방식
- Ex. JUnit, Mockito, PowerMock, Robolectric

* * mock (또는 mocking) 을 해야하는 이유

(출처 : http://www.baeldung.com/mockito-vs-easymock-vs-jmockit

http://hyunalee.tistory.com/33)

- TDD (또는 ATDD, BDD) 등의 테스트를 중심으로 하는 개발 주도적 방법론에서는 **이미 코드를 작성했다고 가정**한다. 또는 단순히 종속성에 의존하여 기존 기능을 구현하는 클래스에 대한 테스트를 하고자 한다.
- 이를 위해서는 **현재 테스트 중인 객체의 의존성을 제어할 수 있는 대체 객체가 필요**하다.
- 이러한 대체 객체를 만드는 경우 극단적인 값의 변환, 예외 처리 등에 의해 시간이 많이 걸리게 된다.
- 이 시간을 단축하고, **테스트 코딩을 단순화할 수 있는 방법이 mock** 이다.
- dummy objects: 전달되지만 실제로 사용되지는 않는다. 일반적으로 매개 변수 목록을 채우기 위해 사용된다. 객체만 만들기 위해서 사용된다.
- fake objects: 작동하는 구현체는 가지고 있으나 일반적으로 프로덕션에 적합하지 않은 객체로, 제한된 기능을 가진 객체이다. 보통 테스트 목적으로 사용된다. 복잡한 로직이나 객체 내부에서 필요로 하는 다른 외부 객체의 동작을 비교적 단순화하여 구현한 객체이다. 다른객체와의 의존성을 제거하기 위해 사용된다.
- stub : 테스트 중에 작성된 호출에 대한 미리 준비된 답변을 제공한다. 일반적으로 테스트를 위해 프로그래밍 된 내용 외에는 응답하지 않는다. 더미 객체가 마치 실제로 동작하는 것처럼 보이게 만들어 놓은 객체로 특정한 값을 리턴해주거나 특정 메시지를 출력하는 작업을 한다.
- mock : 객체가 수신할 것으로 예상되는 호출들을 예측하여 미리 프로그래밍한 객체이다.
- test spy : 테스트에 사용된 객체에 대해서 특정 객체가 사용되었는지, 그 객체의 예상된 메소드가 정상적으로 호출됐는지 확인하여 호출 여부를 몰래 감시하여 기록한다. 이후 요청이 들어오면 해당 정보를 전달한다.

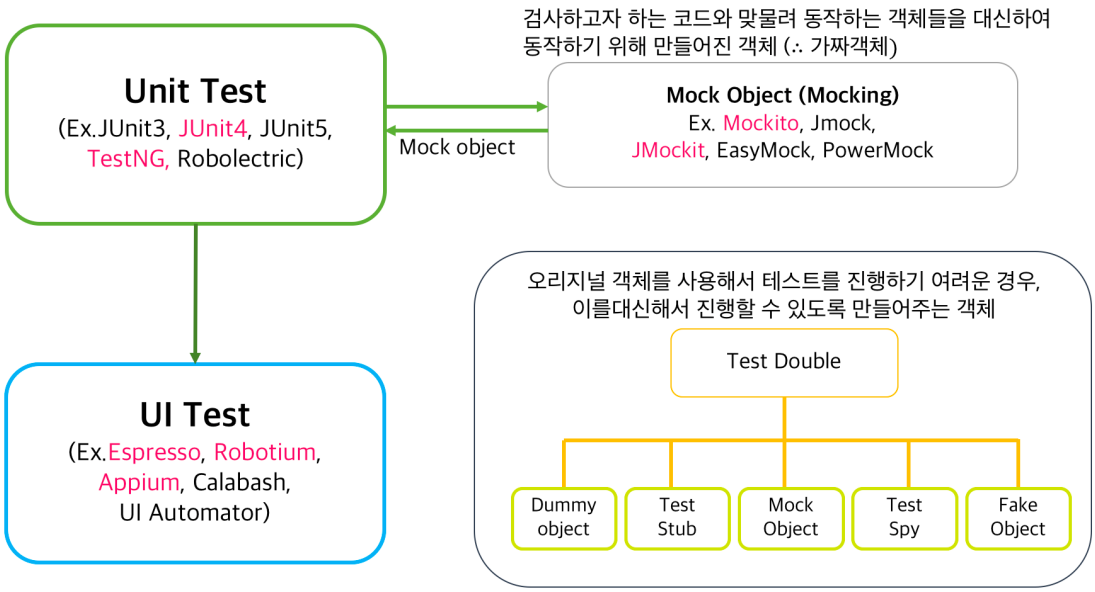
2. UI 테스트

- 사용자 인터랙션 (버튼클릭, 텍스트 입력 등)을 평가
- Espresso, UIAutomator, Robotium, Appium, Calabash

테스트가 가능한 방식으로 앱 구조를 갖추어야 한다.

- 뷰 : 액티비티, 프래그먼트에 해당하는 부분으로 UI 변화가 생기는 부분. 뷰만이 프레젠터에 말을 걸 수 있다.

- 프레젠터 : 무엇을 보여줄 것인지에 대한 비지니스 로직을 담는 부분. 레파지토리에서 정보를 요청하거나 뷰에 정보를 전달한다. 유닛테스트가 까다로워지지 않게 하려면 가능한 안드로이드 상세 코드를 프리젠터에 넣지 않아야 한다.
- 레파지토리 : 데이터가 로컬에서 가져와야 하는지 또는 네트워크에서 가져와야 하는지를 결정하는 과정에서 프레젠터와 소통한다.
- 모델 : 프레젠터로부터 뷰에 정보를 전달하기 위해 사용되는 모델로 전형적인 POJO 이다.
  (POJO : Plain Old Java Object. 오래된 방식의 간단한 자바 오브젝트.)



안드로이드 테스트의 장벽

- Mock을 쓰기 어려운 기본 프레임워크 구조
- 빈약한 기본 Mock 클래스들 (필요한 동작은 override 하여 직접 구현해야하는 문제점)
- 기본적으로 제공되는 테스트를 쓰고 배우기 쉽지 않다. 여러 클래스 중에서 어떠한 것을 써야하는가에 대해 알기 위해서는 모든 것을 정확하게 알고 있어야만 한다.
- UI 테스트 본연의 어려움이 있다. 안드로이드는 UI생성과 이벤트를 다루는 비중이 높은 편이다. UI 객체의 속성은 자주 바뀌고, 익명 클래스를 통해 처리되는 이벤트는 Mock 객체로 바꾸고 추적하는데 어렵다.
- 느린 테스트 실행 속도. 한 줄을 고쳐도 패키징 -> 설치 -> 실행 사이클을 돌기 때문에 속도가 오래 걸린다.

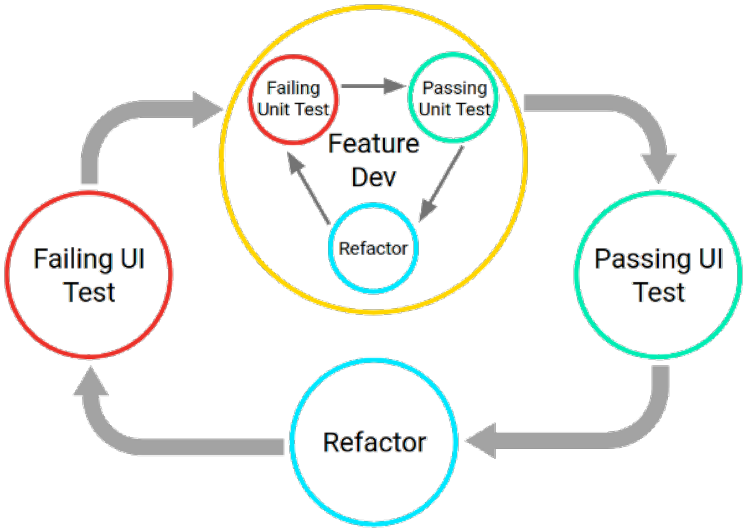
(출처 https://academy.realm.io/kr/posts/aw212-android-unit-ui-test-recorder-data-binding-include/?)

안드로이드 테스트

(전체 출처 https://developer.android.com/training/testing/fundamentals.html#dev-workflow)

- 반복 개발 워크플로우의 사용 (Use an iterative development workflow)

- 반복적인 기능을 개발 할 때는 새로운 테스트를 작성하거나 기존 유닛 테스트에 케이스를 추가하는 것으로 시작된다.
- 아래의 사이클은 반복적인 테스트 주도 개발과 관련한 사이클로서, 해당 사이클 세트는 앱이 모든 use case를 만족할 때 까지 계속되어야 한다.



- 테스트 피라미드

(참고 - 피라미드 테스트 / 애자일 https://books.google.co.kr/books?id=e9ELCwAAQBAJ&pg=PA3&lpg=PA3&dq=android+pyramid+test&source=bl&ots=s-8Llj9LGJ&sig=6jhS0DrBWaOlc4qTlhdQjkqp57A&hl=ko&sa=X&ved=0ahUKEwjb26WKqsTWAhXFzbwKHSNmBMwQ6AEIbDAM#v=onepage&q=android%20pyramid%20test&f=false)

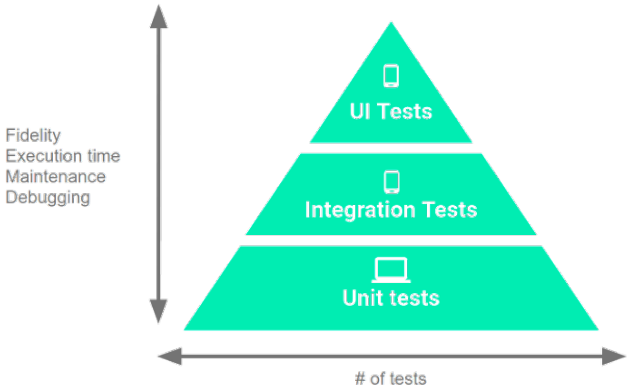
(참고 2 - 출처 : http://kimjihyok.info/2017/04/06/android-testing-%EC%A0%95%EB%A6%AC/)

1. Unit Test

- JVM 상에서 구동되는 로컬 테스트.
- 실행 시간이 매우 짧으며 안드로이드 혹은 다른 라이브러리 코드에 대한 의존성이 없어야 한다.
- 코드의 가장 작은 단위의 코드를 테스트 한다. 일반적으로 메소드 혹은 클래스가 된다.
- 특정 클래스의 모든 Public 메소드가 구현한대로 동작하는가, 특정 클래스의 state가 올바른 state에 존재하는지와 같은 것을 테스트한다.

2. Integration Test

- 단위 테스트가 완료된 모듈들을 합쳐서 하는 다음 과정의 테스트.
- 비지니스로직을 테스트 하는 경우가 많은데 여기서 비지니스 조직은 앱에서 데이터가 가공되는 로직을 말한다.



테스트피라미드는 앱의 세가지 테스트 유형 (소형, 중형, 대형)을 포함해야하는 방법을 보여준다.

- 소규모테스트(Unit Test) : 프로덕션 시스템과 별도로 실행할 수 있는 단위 테스트. 일반적으로 모든 주요 구성 요소를 모의 테스트하며 컴퓨터에서 빠르게 실행해야 한다.
- 중간테스트(Integration Test) : 작은 테스트와 큰 테스트사이에 있는 통합 테스트. 여러 구성 요소를 통합하며 에뮬레이터 또는 실제 장치에서 실행된다.
- 대규모 테스트(UI Test) : UI 워크플로우를 완료하여 실행되는 통합 및 UI 테스트. 핵심 최종 사용자 작업이 에뮬레이터 또는 실제 장치에서 예상대로 작동하는지를 확인한다.

작은 테스트가 빠르고 집중적으로 이루어지며, 신속하게 실패를 처리할 수는 있으나 통과 테스트를 통해 앱을 완벽하게 작동시킬 수 있다는 확신을 가지기는 어렵다. 반면 큰 테스트의 경우 반대의 문제를 가지게 된다. 따라서 각 카테고리의 테스트 비율은 기본적으로 소70% 중20% 대10%로 나누는 것이 바람직하다.

- 소규모 테스트(Unit Test)의 작성

앱의 기능을 추가하고 변경하면 해당 기능에 대하여 단위 테스트를 만들고 실행하여 원하는 기능이 잘 작동하는 지에 대해서 확인해야 한다.

1. **Robolectric**

- 앱의 테스트 환경에서 단위테스트가 **Android 프레임워크와 더 광범위하게 상호작용** 해야할 때 사용한다.
- 본 툴은 안드로이드 프레임워크를 에뮬레이트하는 테스트에 용이(친화적)하며, java 기반의 논리 스텁을 실행한다.
- Robolectric 테스트는 Android 디바이스에서 실행되는 테스트의 정확도와 거의 비슷하나 **디바이스 테스트보다 더 빨리 실행된다**.

- 다음의 Android 플랫폼 측면을 지원한다.
    - Android 4.1 (API 16) 이상
    - Android Gradle Plugin 버전 2.4 이상
    - Component Lifecycles
    - Event loops
    - All Resources
- [Robolectric (1) | 출처 : http://robolectric.org/]
    - Android 에뮬레이터 또는 기기에서 테스트를 실행하는 속도는 느린 편이다. TDD에서 이 방법은 결코 좋은 방법이 아니다.
    - Robolectric 은 **안드로이드 SDK jar파일을 제거하여 안드로이드 앱의 개발을 테스트할 수 있는 단위 테스트 프레임워크이다**.
    - 테스트는 워크스테이션의 JVM 내부에서 빠르게 실행된다.
    - 코드에 '@RunWith(RobolectricTestRunner.class)' 를 사용한다.
    - Android SDK 가 제공하는 클래스에 가짜 동작을 심어 JVM에서 Android 코드를 실행한다.
    - 모의 프레임워크(Mockito, mock out)를 사용할 수도 있으나 **기본적으로 블랙 박스 테스트 스타일과 가까우므로 리팩토링에 대한 테스트가 더욱 효과적**으로 이루어질 수 있고 앱의 동작에 집중 할 수 있다.
- [Robolectric (2) | 출처 : https://gist.github.com/benelog/903e21cad1624263fd7825131f67ff82]
    - JVM 에서 테스트해도 동일한 결과를 보장하는 문자열, 날짜처리, 프로토콜 파싱 영역에 대해서 이득이 많다. 따라서 주로 **java.lang, java.util, java.io 패키지가** 다루는 영역에 우선적으로 집중하는 것이 좋다.

- Android SDK에 일부 종속성이 있는 경우
- Activity, Fragment 등과 같이 Android 구성요소에 대한 제한된 작업을 제공하며, 예측(계측)테스트 보다 속도가 빠르다.
- Unit Test가 안드로이드 프레임워크와 더 광범위하게 상호작용하는 경우에 사용하기 좋다.

2. Mock objects

- 수정 된 버전에 대해 단위 테스트를 실행하여 앱이 상호 작용하는 Android 프레임워크의 요소를 모니터링 할 수 있다.
- **Android 시스템과 상호 작용하는 코드를 테스트하고자** 하면 Mockito와 같은 프레임워크를 사용하여 mock 객체를 구성해야 한다.
- 코드에 리소스에 대한 참조 또는 Android 프레임워크와의 복잡한 상호작용이 포함되어 있는 경우 Robolectric과 같은 다른 방식의 단위 테스트를 사용해야 한다.

(즉 시스템과의 상호작용에는 Mockito / 더 복잡한 경우는 Robolectric 수준의 툴이 필요)

- **Mockito** (출처 : https://github.com/mockito/mockito/wiki/Mockito-features-in-Korean)
    - 자바에서 단위테스트를 하기 위해 Mock 을 만들어 주는 프레임워크
    - 간단하게 stub, use, verify를 사용한다.

- mock() 와 spy() 를 이용하여 Mock을 생성할 수 있다.
- @mock annotation을 통해서 쉽게 mock을 생성할 수 있다.

- Mockito 홈페이지 http://site.mockito.org/
- 참고 : https://code.google.com/archive/p/mockito/wikis/MockitoFeaturesInKorean.wiki

3. Instrumented Unit tests
- 물리적 장치 또는 에뮬레이터에서도 계측 단위 테스트를 실행할 수 있다.
- 이 테스트는 프레임워크의 Mock 이나 stubbing 이 필요하지 않다.
- 이 테스트 방법은 로컬 단위 테스트보다 실행 시간이 현저하게 느리므로 실제 장치 하드웨어에 대한 응용 프로그램의 동작을 평가하는 것이 필수적일 때만 사용하는 편이 좋다.

● 매체 테스트 작성 (Medium Test)

개발 환경에서 앱의 각 유닛 테스트를 마친 후에는 에뮬레이터나 장치에서 실행할 때 구성 요소가 올바르게 작동하는지 확인해야 한다. 본 테스트 과정은 개발 프로세스 중에서 이 부분을 확인할 수 있다.
본 테스트는 앱의 일부 구성 요소가 실제 하드웨어에 의존하고 있는 경우 특히 중요하다.
ex. 서비스 테스트, 통합 테스트, 외부 종속성의 동작을 시뮬레이트하는 밀접한 UI 테스트
일반적으로 실제 장치보다는 Firebase Test Lab과 같이 **에뮬레이터 된 장치 또는 클라우드 기반 서비스에서 응용 프로그램을 테스트 하는 편이 좋다.** 여러 화면의 크기와 하드웨어 구성 조합을 보다 쉽고 빠르게 테스트 할 수 있기 때문.

● 대규모 테스트 작성 (Large Test)

앱의 크기가 작은 경우 전체 앱의 기능을 평가하는 대규모 테스트 세트가 하나만 필요할 수 있으나 그렇지 않은 경우 팀 소유권, 기능별 업종, 사용자 목표 별로 테스트를 나누어야 한다.
작성한 대규모 워크플로우 기반 테스트 각각에 대해서는 해당 워크플로우에 포함 된 각 UI 구성 요소의 기능을 확인하는 medium test도 작성해야 한다.
(참고 : https://developer.android.com/topic/libraries/testing-support-library/index.html?hl=ko)

- JUnit4 Rules (출처 : https://developer.android.com/training/testing/junit-rules.html)

Android 테스팅 지원 라이브러리에는 'AndroidJUnitRunner' 라는 JUnit 규칙이 포함되어 있다
JUnit 규칙은 유연성을 높이고 테스트에 필요한 상용구 코드를 줄인다.

- ActivityTestRule

  단일 활동에 대한 기능 테스트를 제공한다.

  @test, @before 가 annotate 되기 전에 테스트가 시작되며 테스트가 종료되면 @after를 annotate 한다.
- ServiceTestRule

  테스트 기간의 전후에 서비스를 시작하고 종료하는 간단한 메커니즘을 제공한다.

  또한 서비스를 시작하거나 바인딩할 때 서비스가 성공적으로 연결되도록 한다.

- AndroidJUnitRunner (출처 : https://developer.android.com/training/testing/junit-runner.html)

AndroidJUnitRunner 클래스는 JUnit 테스트 runner로 Expresso, UI Automator 테스팅 프레임워크를 사용하는 클래스를 포함하여 JUnit3, 4의 테스트 클래스에서 사용할 수 있다.
@RunWith(AndroidJUnit4.class) 가 코드 접두에 와야 한다.

- Espresso

Espresso는 다음과 같은 인앱 상호작용을 자동화하면서 비동기 작업을 동기화 한다.

- View객체에 대한 작업 수행
- 앱의 프로세스 경계를 넘은 워크플로우 완성. (Android 8.0 (API 26) 이상에서만 사용 가능)
- 접근성 요구 사항이 있는 사용자가 앱의 사용방식을 평가한다.
- RecyclerView, AdapterView 개체 찾기 및 활성화
- 보내는 intent의 상태확인
- WebView 객체 내의 DOM 구조 확인
- 앱 내에서 장기 실행 백그라운드의 작업 추적

- [Espresso (1) | 출처 : https://developer.android.com/training/testing/espresso/index.html]
  - UI 테스트 프레임워크의 일종
  - 사용자의 조작을 코드로 구현하여 재생하고 그에 대한 변화를 검사하는데에 사용한다.
  - (참고 : https://youngjaekim.wordpress.com/2015/03/07/espresso%EB%A1%9C-%EC%95%88%EB%93%9C%EB%A1%9C%EC%9D%B4%EB%93%9C-ui-%ED%85%8C%EC%8A%A4%ED%8A%B8%ED%95%98%EA%B8%B0/)
- [Espresso (2) | 출처 : http://www.vogella.com/tutorials/AndroidTestingEspresso/article.html]
  - Espresso는 안정적인 사용자 인터페이스 테스트를 쉽게 작성할 수 있도록 만든 Android용 테스트 프레임워크이다.
  - 테스트 액션을 애플리케이션의 사용자 인터페이스와 자동으로 동기화한다.
  - 프레임워크는 테스트가 실행되기 전 Activity가 실행되도록 한다.
  - 에스프레소의 기본적인 세가지 구성요소는 아래와 같다.
    - ViewMatchers : 현재 뷰 계층에서 뷰를 찾을 수 있다.
    - ViewActions : 뷰에서 작업을 수행할 수 있다.
    - ViewAssertions : 뷰에서 상태를 확인할 수 있다.
    - 에스프레소 테스트 케이스의 구성은 다음과 같다

Base Espresso Test

```
onView(ViewMatcher)                    ❶
 .perform(ViewAction)                  ❷
   .check(ViewAssertion);              ❸
```

- UI Automator (UI 자동화)

UI Automator 프레임워크는 현재 표시된 UI의 계층구조 검사, 스크린 샷 캡처, 장치의 현재 상태 분석과 같이 앱을 대신하여 시스템 앱 내의 상호작용을 수행한다.

- [UI Automator | 출처 : https://developer.android.com/training/testing/ui-automator.html]

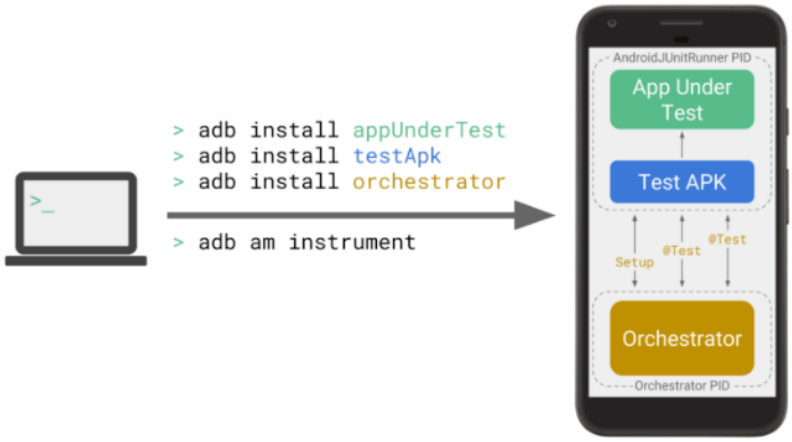
- 시스템 및 설치된 응용 프로그램에서 교차 응용 프로그램 기능 UI 테스트에 적합한 UI 테스트 프레임워크
- UI Automator API를 사용하면 테스트 장치에서 설정 메뉴 또는 앱 실행기를 여는 것과 같은 작업을 수행할 수 있다.
- 해당 프레임워크는 테스트 코드가 대상 응용 프로그램의 내부 구현 세부 정보에 의존하지 않는 **블랙박스 스타일의 자동 테스트를** 하는데에 적합하다.
- 주요 기능은 다음과 같다.
  - Viewer 레이아웃 계층을 검사한다. (UI 자동화 뷰어 참고)
  - 성태 정보를 검색하고 대상 장치에서 작업을 수행한다. (장치 상태 액세스 참고)
  - 앱 간의 UI 테스트를 지원한다. (UI 자동화 API 참고)

  - UI 자동화 뷰어
    - Android 장치에 현재 표시된 UI 구성요소를 검색하고 분석할 수 있는 편리한 GUI를 제공한다.
    - 레이아웃 계층을 검사하고 장치를 전면에서 볼 수 있는 UI 구성 요소를 확인 할 수 있다.
    - <android-sdk>/tools/ 디렉토리에서 확인할 수 있다.
  - 장치 상태 액세스
    - UIDevice 대상 응용프로그램이 실행되는 장치에 액세스하고 작업을 수행하는 클래스를 제공한다.
    - 이 메소드를 통해 현재 방향이나 표시 크기와 같은 장치 속성에 액세스할 수 있으며, 아래와 같은 작업들을 수행할 수 있다.
      - 장치 회전
      - 키 또는 D 패드의 버튼 누르기
      - 뒤로, 홈, 메뉴 버튼 누르기
      - 알림창 열기
      - 스크린샷 UiDevice.pressHome() 메서드를 통해 사용할 수 있다.
  - UI 자동화 API
    - 타겟팅하는 앱의 구현 세부 정보를 알 필요없이 강력한 테스트를 작성할 수 있다.

(그 외 참고 : https://developer.android.com/training/testing/ui-automator.html#ui-automator-viewer)

- Android Test Orchestrator

- Instrumentation (애플리케이션 계층 코드를 구현하기위한 기본 클래스 / 참고 : https://developer.android.com/reference/android/app/Instrumentation.html)
- 자체 호출 내에서 각 응용 프로그램의 테스트를 실행할 수 있다.
- 다음과 같은 이점이 있다.
  - 공유 상태가 없다 : 각 테스트는 자체 Instrumentation 인스턴스에서 실행되므로 테스트 칸의 공유 상태를 장치의 CPU 또는 메모리에 누적되지 않는다.
  - 충돌이 격리된다 : 한 테스트에서 충돌이 발생하더라도 자체 인스턴스만 가져 오므로 Instrumentation 제품 군의 다른 테스트를 계속 실행된다.



- 위와 같이 기기에서 완전히 테스트 분리를 실현할 수 있다.

참고 : https://developer.android.com/training/testing/junit-runner.html#using-android-test-orchestrator
출처 : https://developers-kr.googleblog.com/2017/08/android-testing-support-library-10-is.html

---

그 외의 테스트 Tool 및 비교

Unit 테스트를 위한 Tool

- **JMock**
  - Mock 객체로 자바코드의 테스트 중심 개발을 지원하는 라이브러리이다.
  - Mock Object는 프로그램 객체 간 상호 작용을 설계하고 테스트하는데 도움을 준다.
    - (출처 : http://www.jmock.org/)
  - 다른 클래스에 의존하는 클래스를 독립적으로 테스트 할 수 있다.
  - 하나의 클래스가 다른 클래스에 종속된다고 가정하면 종속 클래스의 mock object를 만들고 초기 속성을 설정하여 종속 클래스를 테스트할 수 있다.
    - (출처 : http://www.askeygeek.com/jmock-for-beginners/)

(참고 : http://blog.naver.com/PostView.nhn?blogId=wipos&logNo=80162292128&parentCategoryNo=&categoryNo=46&viewDate=&isShowPopularPosts=false&from=postView)

- **JMcockit**
  - http://jmockit.org/index.html
  - Java용 자동 테스팅 toolkit
  - 바이트 코드 계측에 크게 의존한다.
  - 모의 객체를 런타임 시 응용 프로그램 바이트 코드로 직접 연결한다. 따라서 **인터페이스를 구현하지 하는 객체도 mock이 가능하다**. 이는 프록시 기반의 mock 라이브러리보다 좋은 점으로 볼 수 있다.
    - (출처 : http://winterbe.com/posts/2009/08/18/introducing-jmockit/)

- **EasyMock**
  - http://easymock.org
  - 인터페이스를 기반으로 Mock 객체를 만들 수 있다.
  - CreateMock, Record, Replay, Verify 의 단계로 동작한다.
    - CreateMock : 인터페이스에 해당하는 Mock 객체를 만든다 .
    - Record : Mock 객체 메소드의 예상되는 동작을 녹화한다. 생성한 Mock 객체의 어떠한 메소드가 호출되어야 하며 몇 번 호출되어야 하는지 등의 정보를 기록한다.
    - Replay : 예정된 상태로 재생한다. Replay메소드가 호출되고 난 후 실제 동작이 수행된다.
    - Verify : 예상했던 행위가 발생했는지 검증한다.
    - (출처 : http://blog.naver.com/PostView.nhn?blogId=wipos&logNo=80162292128&parentCategoryNo=&categoryNo=46&viewDate=&isShowPopularPosts=false&from=postView)

- **PowerMock**
  - 다른 mock 라이브러리를 강력한 기능으로 확장하는 프레임워크
  - 사용자 정의 클래스 로더 및 바이트 코드 조작을 사용하여 정적메소드, 생성자, 최종 클래스 및 메소드, 개인메소드, 정적 initializer 제거 등을 가능하게 한다.
  - 현재 EasyMock과 Mockito 를 지원하고 있다.
    - (출처 : https://github.com/powermock/powermock)

- **TestNG**
  - JUnit과 NUnit에서 영감을 얻을 프레임워크.
  - 다음과 같은 특징을 가지고 있다.
    - Annotation
    - 멀티스레드에 안전하게 코드를 테스트한다.

- Xml을 이용하여 유연한 테스트 설정을 한다.
- Data-driven testing 을 지원한다.
- 강력한 실행 모델
- application 서버 테스팅을 위해 의존하는 method
- ignore, IDEA, Ant, Maven.. 등을 지원한다.
- 불필요한 코드를 만들 필요가 없고 클래스를 확장할 필요도 없다.
- JUnit으로의 변경이 쉽다.
- (출처 : http://testng.org/doc/)

Tool 비교

- EasyMock vs Mockito
  (출처: https://github.com/mockito/mockito/wiki/Mockito-vs-EasyMock)

	EasyMock	Mockito
		EasyMock에서 시작했으나 현재는 많이 달라져 EasyMock과 코드를 공유하지 않는다.
code example	`import static org.easymock.classextension.EasyMock.*;` `List mock = createNiceMock(List.class);` `expect(mock.get(0)).andStubReturn("one");` `expect(mock.get(1)).andStubReturn("two");` `mock.clear();` `replay(mock);` `someCodeThatInteractsWithMock();` `verify(mock);`	`import static org.mockito.Mockito.*;` `List mock = mock(List.class);` `when(mock.get(0)).thenReturn("one");` `when(mock.get(1)).thenReturn("two");` `someCodeThatInteractsWithMock();` `verify(mock).clear();`
differences		no record, replay modes – verify, stub 의 사용가능
	verify(), when()	expect(mock.foo()), mock.foo()
Verification in order	`Control control = createStrictControl();` `List one = control.createMock(List.class);` `List two = control.createMock(List.class);` `expect(one.add("one")).andReturn(true);` `expect(two.add("two")).andReturn(true);` `control.replay();` `someCodeThatInteractsWithMocks();` `control.verify();`	`List one = mock(List.class);` `List two = mock(List.class);` `someCodeThatInteractsWithMocks();` `InOrder inOrder = inOrder(one, two);` `inOrder.verify(one).add("one");` `inOrder.verify(two).add("two");`

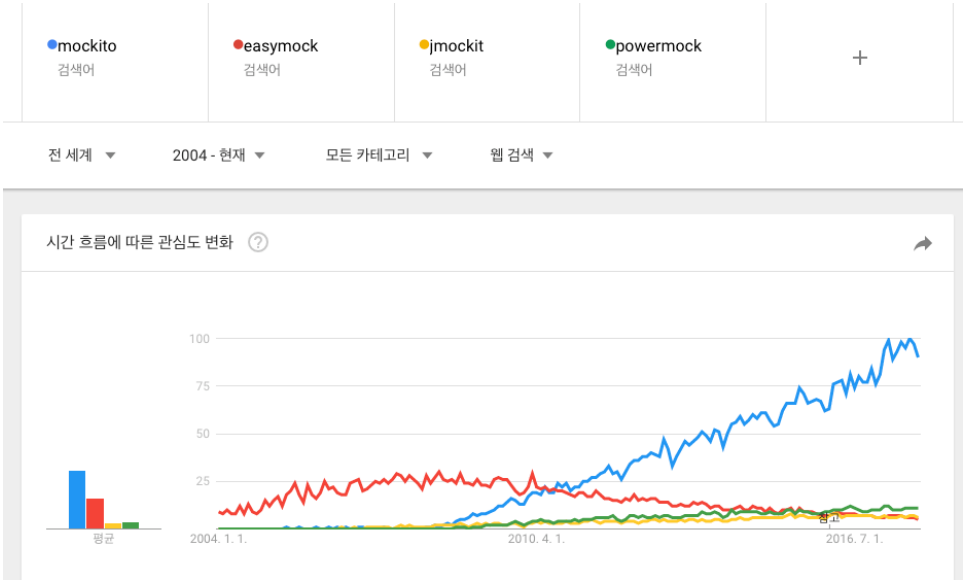
Subbing void method	`List mock = createNiceMock(List.class);` `mock.clear();` `expectLastCall().andThrow(new RuntimeException());` `replay(mock);`	`List mock = mock(List.class);` `doThrow(new RuntimeException()).when(mock).clear();`
Exact number of times verification and argument matchers	`List mock = createNiceMock(List.class);` `mock.clear();` `expectLastCall().times(3);` `expect(mock.add(anyObject())).andReturn(true).atLeastOnce();` `replay(mock);` `someCodeThatInteractsWithMock();` `verify(mock);`	`List mock = mock(List.class);` `someCodeThatInteractsWithMock();` `verify(mock, times(3)).clear();` `verify(mock, atLeastOnce()).add(anyObject());`

- EasyMock vs Mockito vs JMockit

(출처 : http://www.baeldung.com/mockito-vs-easymock-vs-jmockit)

(Google trend:

: https://trends.google.com/trends/explore?date=all&q=mockito,easymock,jmockit,powermock&hl=ko)



	EasyMock	Mockito	JMockit
		EasyMock의 후속모델. PowerMock과 사용시 좋다.	독자적인 라이브러리
Test Setup	@Mock, @TestSubject 사용	@Mock, @InjectMocks 사용	부분 모의 객체에 대한 특정한 annotation이 없다. @Injectable, @Mocked, @Tested 사용

Test Setup

- Example code

- EasyMock

```
@RunWith(EasyMockRunner.class)
public class LoginControllerTest {

    @Mock
    private LoginDao loginDao;

    @Mock
    private LoginService loginService;

    @TestSubject
    private LoginController loginController = new LoginController();
}
```

mock을 하기 위해서는 모든 테스트 메소드에서 'EasyMock.replay(mock)' 을 호출해야 한다.

- Mockito

```
public class LoginControllerTest {

    @Mock
    private LoginDao loginDao;

    @Spy
    @InjectMocks
    private LoginService spiedLoginService;

    @Mock
    private LoginService loginService;

    @InjectMocks
    private LoginController loginController;

    @Before
    public void setUp() {
        loginController = new LoginController();
        MockitoAnnotations.initMocks(this);
    }
}
```

@Mock : 필드를 정의하는데 사용되는 클래스의 mock object를 만든다.

@InjectMocks : 생성된 mock object를 annotation 'mock'에 삽입시킨다.

- JMockit

<table>
<tr><td></td><td colspan="3">

```
@RunWith(JMockit.class)
public class LoginControllerTest {

    @Injectable
    private LoginDao loginDao;

    @Injectable
    private LoginService loginService;

    @Tested
    private LoginController loginController;
}
```

@Tested : 테스트 된 인스턴스가 만들어진다.</td></tr>
<tr><td>Verifying No Calls to Mock</td><td>replay mock and lastly, verify.</td><td>accepts a mock : verifyZeroInteractions() method</td><td>don't specify expectations for that mock.
do a FullVerifications(mock)</td></tr>
<tr><td>Verifying No Calls to Mock
- Example code</td><td colspan="3">- EasyMock

```
@Test
public void assertThatNoMethodHasBeenCalled() {
    EasyMock.replay(loginService);
    loginController.login(null);
    EasyMock.verify(loginService);
}
```

- Mockito

```
@Test
public void assertThatNoMethodHasBeenCalled() {
    loginController.login(null);
    Mockito.verifyZeroInteractions(loginService);
}
```

- JMockit

```
@Test
public void assertThatNoMethodHasBeenCalled() {
    loginController.login(null);
    new FullVerifications(loginService) {};
}
```
</td></tr>
<tr><td>Defining Mocked Method Calls and Verifying Calls to Mocks</td><td>mocking method call : EasyMock.expect(mock.method(args)).andReturn(value)

verifying call : EasyMock.verify(mock)
(*must call it always after call :</td><td>mocking method call : Mockito.when(mock.method(args)).thenReturn(value)

verifying call : Mockito.verify(mock).method(args)</td><td>mocking method call : mock.method(args);
result = value;
(inside any Exceptions block)

verifying call :
new Verifications(){{mock.call</td></tr>
</table>

<table>
<tr>
<td></td>
<td>EasyMock.replay(mock))

verifying args :
isA(Class.class), anyString(), anyInt()…</td>
<td>
verifying args :
특정 값의 전달,
any(), anyString(), anyInt() …</td>
<td>(value)}} or
new Verifications(mock){{}}
→ to verify every expected call previously defined

verifying args :
any, anyString, anyLong and a lot more of that kind of special values</td>
</tr>
<tr>
<td>Defining Mocked Method Calls and Verifying Calls to Mocks
- example code</td>
<td colspan="3">- EasyMock

```
@Test
public void assertTwoMethodsHaveBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    EasyMock.expect(loginService.login(userForm)).andReturn(true);
    loginService.setCurrentUser("foo");
    EasyMock.replay(loginService);

    String login = loginController.login(userForm);

    Assert.assertEquals("OK", login);
    EasyMock.verify(loginService);
}

@Test
public void assertOnlyOneMethodHasBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    EasyMock.expect(loginService.login(userForm)).andReturn(false);
    EasyMock.replay(loginService);

    String login = loginController.login(userForm);

    Assert.assertEquals("KO", login);
    EasyMock.verify(loginService);
}
```


- Mockito</td>
</tr>
</table>

```java
@Test
public void assertTwoMethodsHaveBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    Mockito.when(loginService.login(userForm)).thenReturn(true);

    String login = loginController.login(userForm);

    Assert.assertEquals("OK", login);
    Mockito.verify(loginService).login(userForm);
    Mockito.verify(loginService).setCurrentUser("foo");
}

@Test
public void assertOnlyOneMethodHasBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    Mockito.when(loginService.login(userForm)).thenReturn(false);

    String login = loginController.login(userForm);

    Assert.assertEquals("KO", login);
    Mockito.verify(loginService).login(userForm);
    Mockito.verifyNoMoreInteractions(loginService);
}
```

- JMockit

```java
@Test
public void assertTwoMethodsHaveBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    new Expectations() {{
        loginService.login(userForm); result = true;
        loginService.setCurrentUser("foo");
    }};

    String login = loginController.login(userForm);

    Assert.assertEquals("OK", login);
    new FullVerifications(loginService) {};
}

@Test
public void assertOnlyOneMethodHasBeenCalled() {
    UserForm userForm = new UserForm();
    userForm.username = "foo";
    new Expectations() {{
        loginService.login(userForm); result = false;
        // no expectation for setCurrentUser
    }};

    String login = loginController.login(userForm);

    Assert.assertEquals("KO", login);
    new FullVerifications(loginService) {};
}
```

<table>
<tr><td></td><td colspan="3">defined steps for testing : record, replay, verify
1. record is done in a new Exceptions(){{}} block
2. replay is done simply by invoking a method of the tested class
3. verification is done inside a new Verifications(){{}} block</td></tr>
<tr><td>Mocking Exception Throwing</td><td>.andThrow(new ExcptionClass())
after an EasyMock.expect(…)</td><td>.thenThrow(ExceptionClass.class) after a Mockito.when(mock.method(args))</td><td>just return an Exception as the result of a mocked method call instead of the ‘normal’ return</td></tr>
<tr><td></td><td colspan="3">- EasyMock

```
@Test
public void mockExceptionThrowing() {
    UserForm userForm = new UserForm();
    EasyMock.expect(loginService.login(userForm)).andThrow(new IllegalAr …
    EasyMock.replay(loginService);

    String login = loginController.login(userForm);

    Assert.assertEquals("ERROR", login);
    EasyMock.verify(loginService);
}
```

- Mockito

```
@Test
public void mockExceptionThrowin() {
    UserForm userForm = new UserForm();
    Mockito.when(loginService.login(userForm))
                                    .thenThrow(IllegalArgumentException.class);
    String login = loginController.login(userForm);

    Assert.assertEquals("ERROR", login);
    Mockito.verify(loginService).login(userForm);
    Mockito.verifyZeroInteractions(loginService);
}
```

- JMockit

```
@Test
public void mockExceptionThrowing() {
    UserForm userForm = new UserForm();
    new Expectations() {{
        loginService.login(userForm); result = new IllegalArgumentExce; …
        // no expectation for setCurrentUser
    }};

    String login = loginController.login(userForm);

    Assert.assertEquals("ERROR", login);
    new FullVerifications(loginService) {};
}
```
</td></tr>
</table>

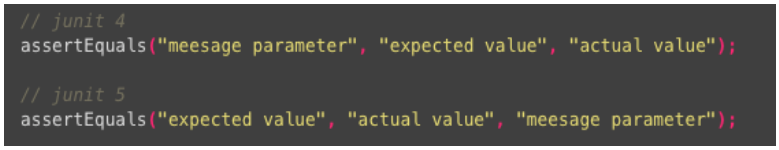
	(그 외는 출처 참조..)
	• JMockit as it forces you to use those in blocks, so tests get more structured. • Easiness of use is important, JMockit will be the chosen option for its fixed-always-the-same structure. • Mockito is the most known so that the community will be bigger.

- JUnit 4 :
  - 자바에서 가장 널리 사용되는 unit test framework.
  - 3과 4 간의 차이는 큰 편이다.
  - 불필요한 양식을 따를 필요가 없으며 @Test annotation을 이용하여 테스트 케이스를 작성한다.
  - (출처 | http://kimjihyok.info/2017/04/06/android-testing-%EC%A0%95%EB%A6%AC/)

- JUnit3 vs JUnit4 vs JUnit5 vs TestNG

	JUnit3	**JUnit4**	**JUnit5**	**TestNG**
JDK Required	JDK 1.2+ and higher version	Java5 and higher	Java8 and higher (lamda)	JDK 7 and higher
Annotation				
Test annotation	testXXX pattern	@Test	@Test	@Test
Run before the first method in the current class is invoked	None	@BeforeClass	@BeforeAll	@BeforeClass
Run after all the test methods in the current class have been run	None	@AfterClass	@AfterAll	@AfterClass
Run before each test method	override setup()	@Before	@BeforeEach	@BeforeMethod
Run after each test method	override teardown()	@After	@AfterEach	@AfterMethod
Ignore test	Commend out or remove code	@ignore	@disabled	@Test(enable=false)
Expected exception	catch exception assert success	@Test(expected = ArithmeticException.class)		@Test(expected = ArithmeticException.class)

Timeout	None	@Test(timeout = 1000)		@Test(timeout = 1000)
Test factory for dynamic test	None	None	@TestFactory	@Factory
Nested test	None	None	@Nested	None
Tagging and filtering	None	@Category	@Tag	
Register custom extensions	None	None	@ExtendWith	
Run before all tests in this suite have run		None		@BeforeSuite
Run after all tests in this suite have run		None		@AfterSuite
Run before the test		None		@BeforeTest
Run after the test		None		@AfterTest
Run before the first test method that belongs to any of these groups in invoked		None		@Before
Other difference				
Architecture		everything bundled into single jar file.	3sub-projects 1. JUnit Platform 2. JUnit Jupiter 3. JUnit Vintage	
	1. JUnit Platform : It defines the 'TestEngine' API for developing new testing frameworks that runs on the platform. 2. JUnit Jupiter : It has all new junit annotacion and 'TestEngine' implementation to run tests written with these annotations. 3. JUnit Vintage : To Support running JUnit3 and JUnit4 written tests on the JUnit5 platform			
Assertions		first parameter	second parameter	

(행 중간 중간에 어떤 값이나 조건이 맞거나 혹은 틀렸다고 주장하거나 단언한다는 의미)	

(출처 : http://www.asjava.com/junit/junit-3-vs-junit-4-comparison/
https://howtodoinjava.com/junit-5/junit-5-vs-junit-4/
https://gloriajun.github.io/testing/2017/03/21/testing-Junit4vsJunit5/
https://www.mkyong.com/unittest/junit-4-vs-testng-comparison/
)

- JUnit4 vs TestNG (자세히 비교)

Annotation	
JUnit4	`@BeforeClass` `public static void oneTimeSetUp() {` `  // one-time initialization code` `  System.out.println("@BeforeClass - oneTimeSetUp");` `}` ● @BeforeClass, @AfterClass 의 사용을 위해서는 정적 메소드를 선언해주어야 한다.
TestNG	`@BeforeClass` `public void oneTimeSetUp() {` `  // one-time initialization code` `  System.out.println("@BeforeClass - oneTimeSetUp");` `}` ● annotation에 대해서 별다른 제약조건이 존재하지 않는다. ● 더 자세한 annotation 을 제공 (ex. before → beforeMethod …)
Exception Test : what exception throws from the unit test	
JUnit4	`@Test(expected = ArithmeticException.class)` `public void divisionWithException() {` `  int i = 1/0;` `}`
TestNG	`@Test(expectedExceptions = ArithmeticException.class)` `public void divisionWithException() {` `  int i = 1/0;` `}`
Ignore Test : whether it should ignore the unit test	

JUnit4

```
@Ignore("Not Ready to Run")
@Test
public void divisionWithException() {
  System.out.println("Method is not ready yet");
}
```

TestNG

```
@Test(enabled=false)
public void divisionWithException() {
  System.out.println("Method is not ready yet");
}
```

Time Test : an unit test takes longer than the specified number of milliseconds to run, the test will terminated and marks as fail

JUnit4

```
@Test(timeout = 1000)
public void infinity() {
    while (true);
}
```

TestNG

```
@Test(timeOut = 1000)
public void infinity() {
    while (true);
}
```

Suite Test : bundle a few unit test and run it together.

JUnit4

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        JunitTest1.class,
        JunitTest2.class
})
public class JunitTest5 {
}
```

- @Runwuth and @Suite are use to run the suite test.
- JUnitTest1 and JUnitTest2 run together after JUnitTest5 executed.

TestNG

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">
    <classes>
       <class name="com.fsecure.demo.testng.TestNGTest1" />
       <class name="com.fsecure.demo.testng.TestNGTest2" />
    </classes>
  </test>
</suite>
```

- XML file is use to run the suite test.
- TestNGTest1 and TestNGTest2 will run it together.

```
@Test(groups="method1")
public void testingMethod1() {
  System.out.println("Method - testingMethod1()");
}

@Test(groups="method2")
public void testingMethod2() {
    System.out.println("Method - testingMethod2()");
}

@Test(groups="method1")
public void testingMethod1_1() {
    System.out.println("Method - testingMethod1_1()");
}

@Test(groups="method4")
public void testingMethod4() {
    System.out.println("Method - testingMethod4()");
}
```

- TestNG can go more than bundle class testing, it can bundle method testing as well.
- "Grouping" concept, every method is tie to a group, it can categorize tests according to features.

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">
    <groups>
      <run>
        <include name="method1"/>
      </run>
    </groups>
    <classes>
       <class name="com.fsecure.demo.testng.TestNGTest5_2_0" />
    </classes>
  </test>
</suite>
```

- we can execute the unit test with group "method1" only.
- With "Grouping" test concept, the integration test possibility is unlimited.

Parameterized Test : vary parameter value for unit test	
JUnit4	• @RunWith and @Parameters is use to provide parameter value for unit test. • @Parameters have to return List[], and the parameter will pass into class constructor as argument.

	`` @RunWith(value = Parameterized.class) public class JunitTest6 { private int number; public JunitTest6(int number) { this.number = number; } @Parameters public static Collection<Object[]> data() { Object[][] data = new Object[][] { { 1 }, { 2 }, { 3 }, { 4 } }; return Arrays.asList(data); } @Test public void pushTest() { System.out.println("Parameterized Number is : " + number); } } ``` • we have to follow the “JUnit” way to declare the parameter, and the parameter has to pass into constructor in order to initialize the class member as parameter value for the testing. • The return type of parameters class is “List[]”, data has been limited to String or a primitive value for testing.
TestNG	• XML file or @DataProvider is use to provide vary parameter for testing. 1. XML file for parameterized test. • Only @Parameters declares in method which needs parameter for testing, the parametric data will provide in TESTNG’s XML configuration files. • We can reuse a single test case with different data sets even get different results. <Unit Test> ``` public class TestNGTest6_1_0 { @Test @Parameters(value="number") public void parameterIntTest(int number) { System.out.println("Parameterized Number is : " + number); } } ``` <XML file>

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="testing">

    <parameter name="number" value="2"/>

    <classes>
       <class name="com.fsecure.demo.testng.TestNGTest6_0" />
    </classes>
  </test>
</suite>
```

2. @DataProvider foe parameterized test

- While pulling data values into an XML file can be quite handy, tests occasionally require complex types, which can't be represented as String or a primitive value.
- TestNG handles this scenario with its @DataProvider annotation, which facilitates the mapping of complex types to a test method.

<@DataProvider for Vector, String or Integer as parameter>

```
@Test(dataProvider = "Data-Provider-Function")
    public void parameterIntTest(Class clzz, String[] number) {
       System.out.println("Parameterized Number is : " + number[0]);
       System.out.println("Parameterized Number is : " + number[1]);
    }

    //This function will provide the patameter data
    @DataProvider(name = "Data-Provider-Function")
    public Object[][] parameterIntTestProvider() {
        return new Object[][]{
                    {Vector.class, new String[] {"java.util.AbstractList",
"java.util.AbstractCollection"}},
                    {String.class, new String[] {"1", "2"}},
                    {Integer.class, new String[] {"1", "2"}}
                   };
    }
```

<@DataProvider for objects as parameter>

<table>
<tr><td></td><td>

```
@Test(dataProvider = "Data-Provider-Function")
public void parameterIntTest(TestNGTest6_3_0 clzz) {
   System.out.println("Parameterized Number is : " + clzz.getMsg());
   System.out.println("Parameterized Number is : " + clzz.getNumber());
}

//This function will provide the patameter data
@DataProvider(name = "Data-Provider-Function")
public Object[][] parameterIntTestProvider() {

    TestNGTest6_3_0 obj = new TestNGTest6_3_0();
    obj.setMsg("Hello");
    obj.setNumber(123);

    return new Object[][]{
                {obj}
    };
}
```


TestNG's parameterized test is very user friendl and flexible.
It can support many complex data type as parameter value and the possibility is unlimited.

</td></tr>
<tr><td colspan="2">Dependency Test : method are test base on dependency, which will execute before a desired method.
(If the dependent method fails, than all subsequent tests will be skipped, not marked as failed)</td></tr>
<tr><td>JUnit4</td><td>JUnit framework is focus on test isolation. it did not support this feature at the moment.</td></tr>
<tr><td>TestNG</td><td>It use "dependOnMethods" to implement the dependency testing.

```
@Test
public void method1() {
   System.out.println("This is method 1");
}

@Test(dependsOnMethods={"method1"})
public void method2() {
    System.out.println("This is method 2");
}
```


The method2() will execute only if method1() is run successfully,
Else method2() will skip the test.

</td></tr>
</table>

(출처 :

https://www.mkyong.com/unittest/junit-4-vs-testng-comparison/

http://coronasdk.tistory.com/747

http://slides.com/sergeypirogov/testng-vs-junit-battle#/7

http://testng.org/doc/documentation-main.html#annotations

)

UI 테스트를 위한 Tool

- **Robotium**
  - 네이티브 및 하이브리드 응용 프로그램을 완벽하게 지원하는 Android 테스트 자동화 프레임워크
  - 자동 블랙박스 UI 테스트를 쉽게 작성할 수 있다.
  - 이점은 다음과 같다.
    - 네이티브 및 하이브리드 모두 테스트할 수 있다
    - 테스트 중인 애플리케이션에 대해 최소한의 지식만 필요하다
    - 프레임워크는 Android Activity를 자동으로 처리한다
    - 빠른 test case 실행 가능
    - Maven, Gradle, Ant 와 원활하게 통합되어 지속적인 통합으로 테스트 실행 (등)
      - (출처 : https://github.com/RobotiumTech/robotium)
  - https://robotium.com/
  - 안드로이드 기반 단위 테스트 환경에서 요구사항이 변경되더라도 테스트 코드 작성에 드는 노력을 줄여준다
  - 테스트 코드를 간결하게 유지시켜 테스트 코드의 유지보수를 편리하게 해준다.
  - 가급적 세부적인 안드로이드 기술을 감추고 동작이나 기능에 집중해서 테스트를 작성할 수 있도록 도와준다.
  - 짧은 시간 내에 테스트 케이스를 작성할 수 있다.
  - CI 시스템의 경우 Maven, Ant 의 통합을 쉽게 할 수 있도록 지원한다.
    - (출처: https://subak.io/?p=58)

- **Appium**
  - 네이티브, 하이브리드 및 모바일 웹 응용 프로그램용 오픈 소스 테스트 자동화 프레임 워크
  - 기본 응용 프로그램을 테스트 할 때 SDK를 포함하거나 응용 프로그램을 다시 컴파일하지 않아도 된다.
  - 원하는 테스트 사례, 프레임 워크 및 도구를 사용할 수 있다.
  - 다음과 같은 모바일 자동화 요구 사항을 충족시키도록 설계되었다.
    - 자동화하기 위해 앱을 다시 컴파일하거나 어떤 식으로든 수정할 필요가 없다.
    - 테스트를 작성하고 실행하기 위해 특정언어나 프레임워크에 얽매이면 안된다.
    - 모바일 자동화 프레임 워크는 자동화 API에 관련하여 새로운 방향을 제시하면 안된다.
    - 모바일 자동화 프레임 워크는 오픈 소스이어야 한다.
      - (출처 : http://appium.io)
  - 하나의 테스트 코드로 서로 다른 플랫폼(Android, iOS 등) 의 앱을 테스트할 수 있도록 해주어 제품 유지보수의 비용을 줄여준다.

(출처, 참고 :
https://domich.wordpress.com/2016/01/11/appium-%EC%95%A0%ED%94%BC%EC%9B%80-%ED%94

%84%EB%A1%9C%ED%8C%8C%EC%9D%BC%EB%A7%81-%EA%B8%B0%EB%B0%98-ui-%ED%85%8C%EC%8A%A4%ED%8A%B8-%EC%9E%90%EB%8F%99%ED%99%94-%EB%8F%84%EA%B5%AC/)

- **Calabash**
  - 안드로이드, iOS 기본 및 하이브리드 응용 프로그램을 위한 자동화 된 테스트 툴
  - 무료 오픈 소스 프로젝트

(출처 : http://calaba.sh/)

Tool 비교

- Espresso vs UI Automator

Espresso	UI Automator
-test automation framework -test many things inside your application in simple way	- framework for automation testing outside of application - ex. notification, access to any application
- 앱 내부에서 기능적 UI 테스트를 실행하기 적합한 UI 프레임워크	- 시스템과 설치되어있는 앱 전반에 걸쳐 앱 간의 기능적 UI테스트를 실행하기에 적합한 UI 테스트 프레임워크

(출처 : http://alexzh.com/tutorials/android-testing-espresso-uiautomator-together/)

- Espresso vs Appium vs Robotium

Espresso	Appium	Robotium
- provides a set of APIs to build UI tests to test user flows within an app. - well-suited for writing **white box-style automated tests,** where the test code utilizes implementation code details from the app under test.	- an open-source tool for **automating native, mobile web, and hybrid applications on iOS and Android platforms.** - Native apps are those written using the iOS, Android or Windows SDKs.	- automatic **black-box test** case **for Android applications** so test developers implemented classes. - test case developers can write function, system and acceptance test scenarios, spanning multiple Android activities.

- Google for the sole purpose of functional testing of an Android UI. - **white box testing** - **inside** the application	- designed to be a cross-platform test platform. - trades-offs focuses on **black box testing** - externally to the application	- full support for Activity, Dialog, Toast, Menu and Context…
- If you're selecting **just one framework**, then for developers building a native Android application.	- If the tests need to **support multiple platforms** and you need to validate how the app reacts to outside factors.	
	- supported : **iOS, Android, FirefoxOS**	- minimal time needed to write solid test cases. - test cases are more robust due to the run-time binding to **GUI components.** - Integrates smoothly with **Maven or Ant** to run tests as part of continuous integration.

(출처 : https://saucelabs.com/blog/appium-vs-espresso
https://www.linkedin.com/pulse/two-majorly-used-android-test-tools-robotium-vs-jaybrata-chakraborty)

- Robotium vs Appium (자세히)

	Robotium	Appium
Support for Mobile Platforms	can be used for testing a **wide variety of Android Apps.**	supports iOS and Firefox OS along with Android. It **allows testers to use of standard automation APIs on different platforms.**
Re-complication of actual APK	requires to **re-compile or modify** the actual application under test to use of standard automation APIs on all platforms.	does **not re-compile or modify** the actual app in order to run the automation.
Type of Application	can be used **evaluating both native and hybrid mobile applications for Android.**	can be used to **test mobile web** apps along with the **native and hybrid mobile applications**. can also be used to evaluate mobile websites across these web browsers.

Security limitations	it lets click throughout only on the application that is under testing. (ex. if application opens the camera and tries to take a photo, the test ends with a fail.)	
Option to Choose Programming Language	is designed specially to test software for a specific mobile platform. Android application testing -> test case only in Java	Java, PHP, C#, Ruby, Python, Perl, Object-C, Clojure, JavaScript, Node.js
Performance	requires testing professionals to work with the source code while automating tests. so testers have to invest additional time and efforts to write test cases for Robotium.	impacted due to reduced XPath support on mobile devices.

(출처 : https://www.linkedin.com/pulse/two-majorly-used-android-test-tools-robotium-vs-jaybrata-chakraborty)