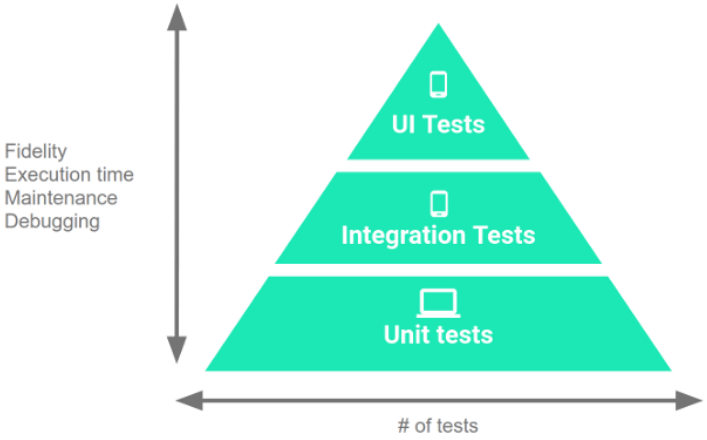
Unit test

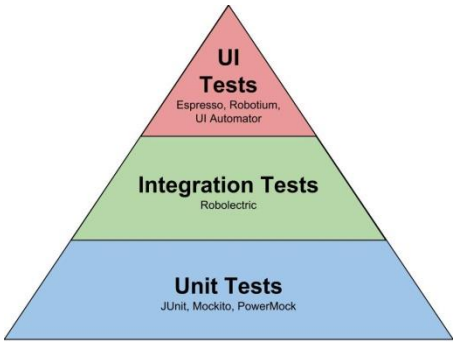
Introduction

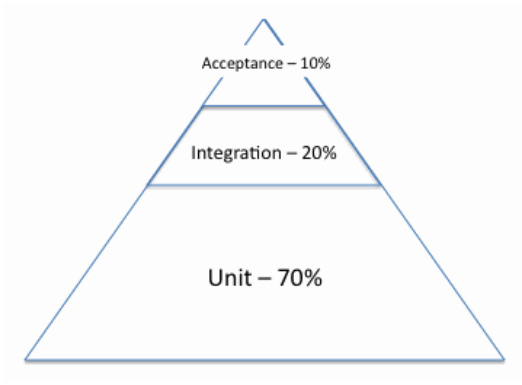
The goal of unit test is to isolate each part of the program and show that individual parts are correct.

Test Pyramid

Unit test cover 70% of the test. No need any android device.







Layer of an App

User Interface Layer

- Consists primarily of Android Views which aren't Unit testable.
- UI logics are not good fit for automated testing any way.

Application layer

- User input handling and flow control
- Contains Android Components Activities, Fragments & Services which are not unit testable.

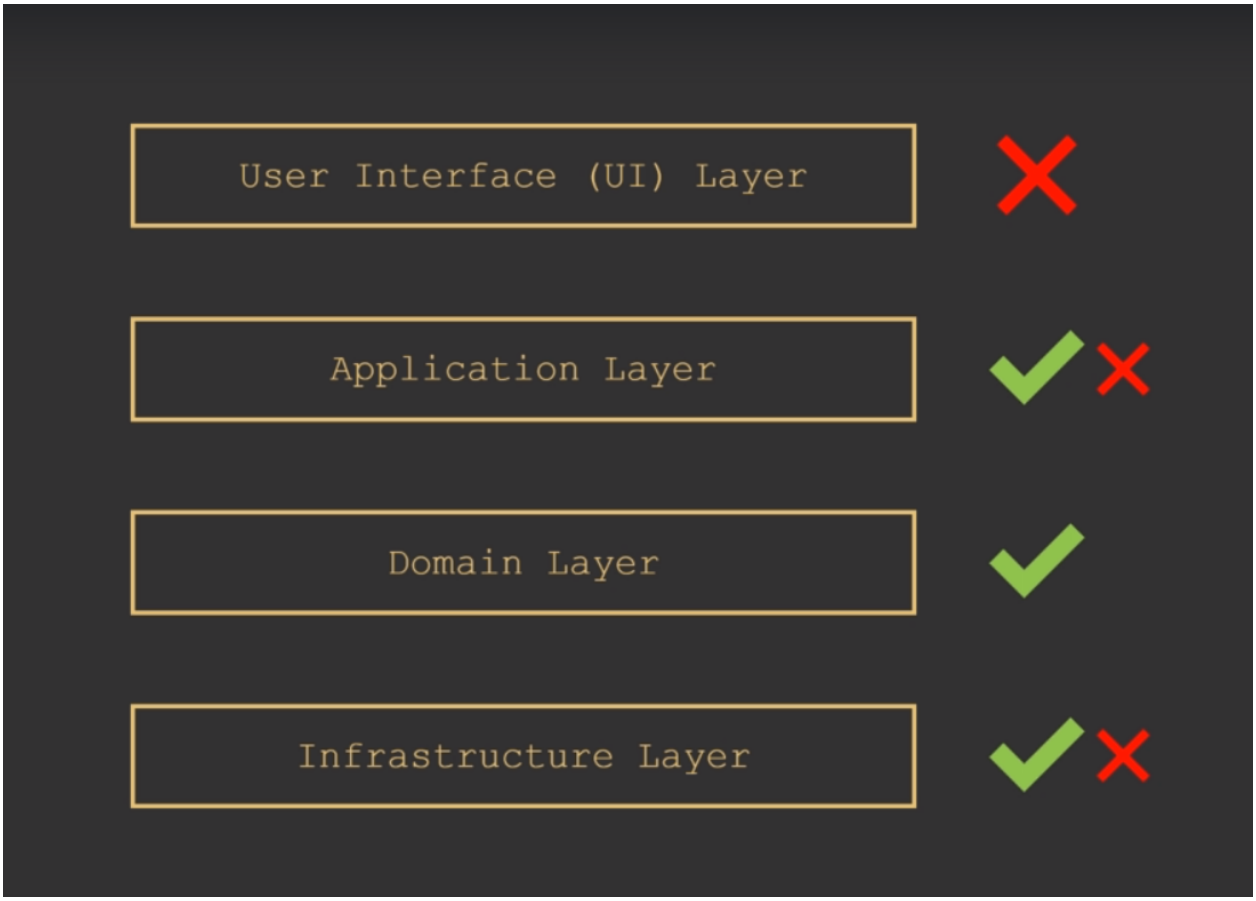
Domain Layer

- Use cases are very good abstraction for domain / business logic.
- Don’t put business logics inside Activity or Fragment.
- Don’t issue static calls from this layer

Infrastructure Layer

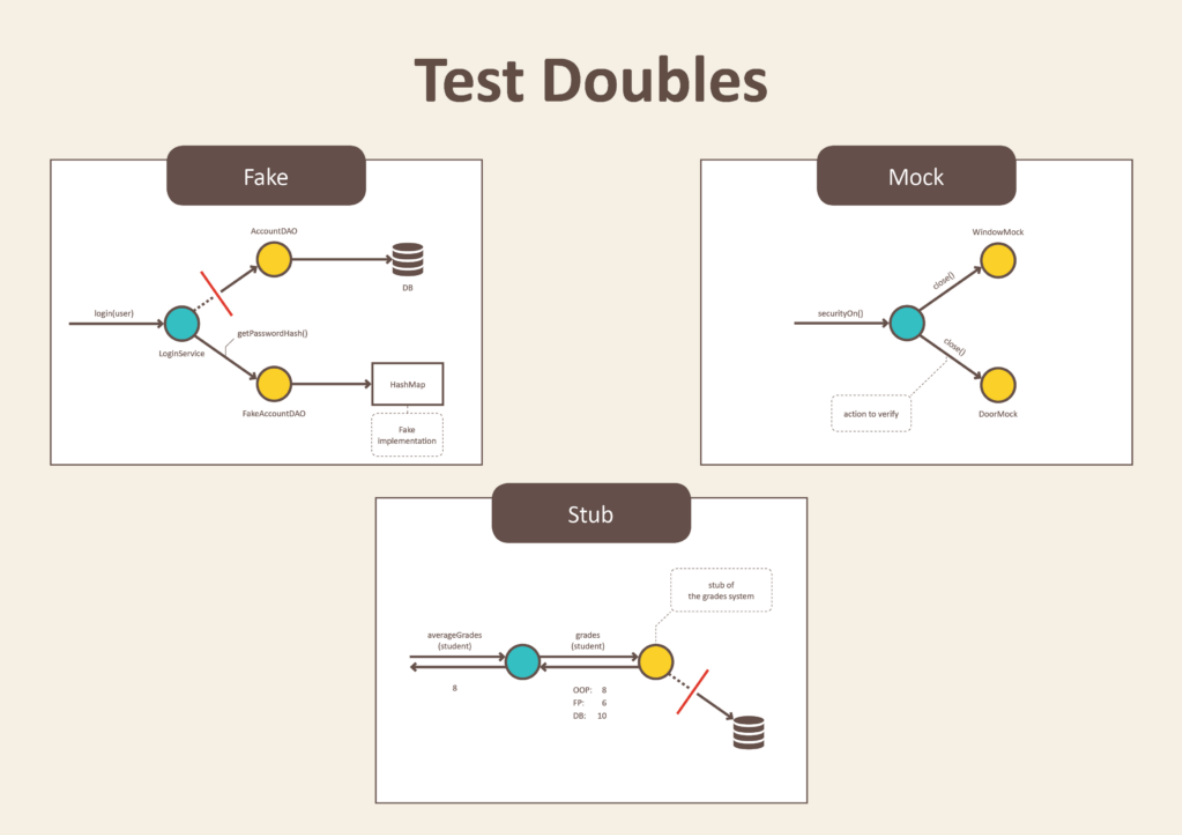
- Often Used 3rd party libraries.

Unit test on which Layer



Test Doubles

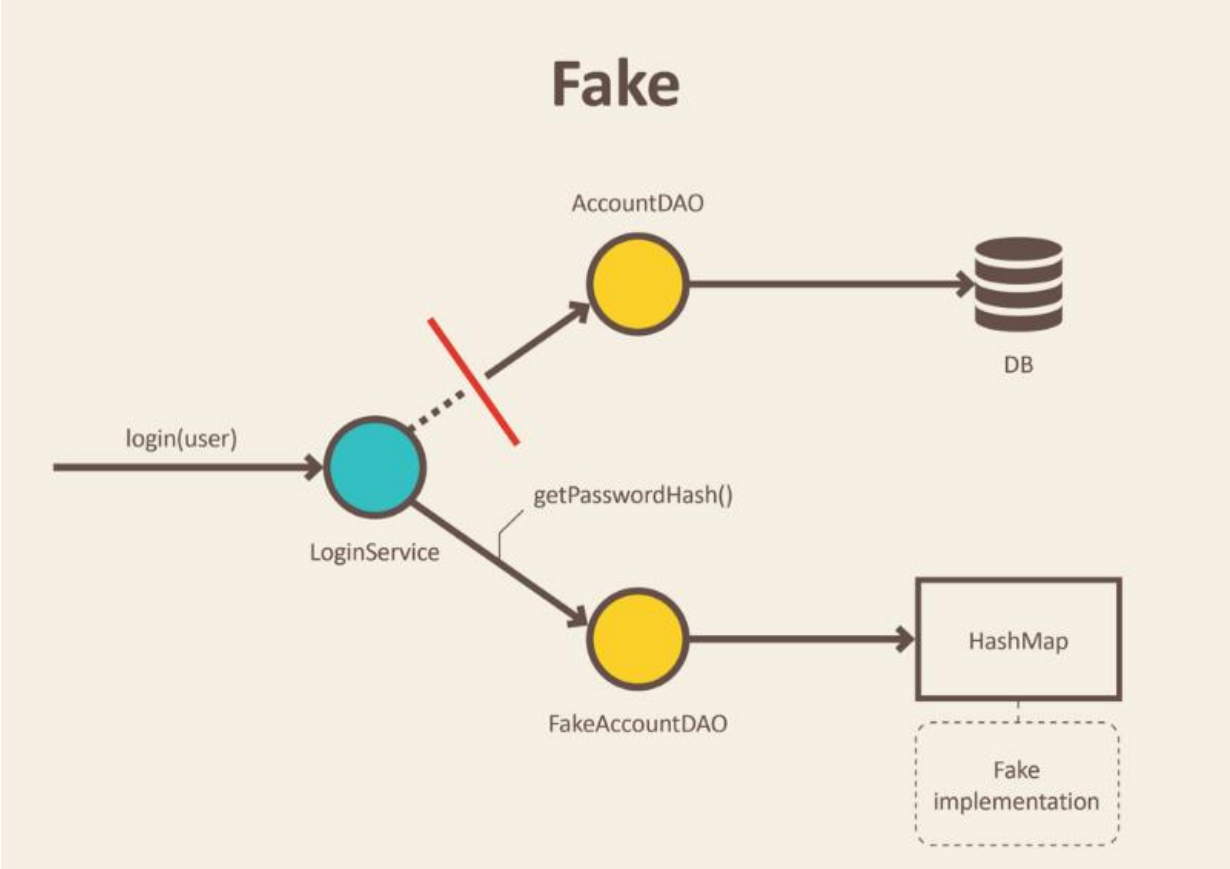
- A test double is an object that can stand in for a real object in a test
- Similar to how a stunt double stands in for an actor in a movie.



Fake

- ***Fakes are objects that have working implementations, but not same as production one. Usually they take some shortcut and have simplified version of production code.***

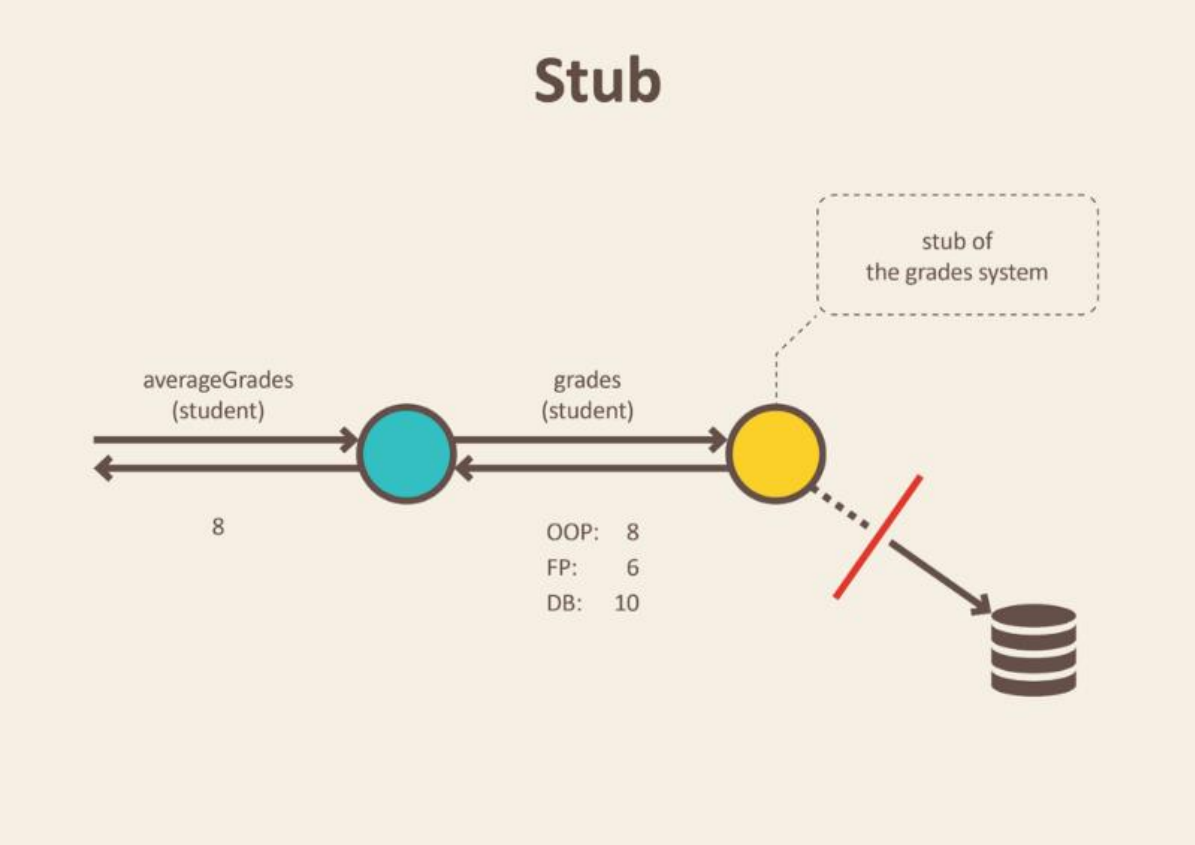
- Fakes can be used when you can't use a real implementation in your test. (e.g. if the real implementation is too slow or it talks over the network).



Stub

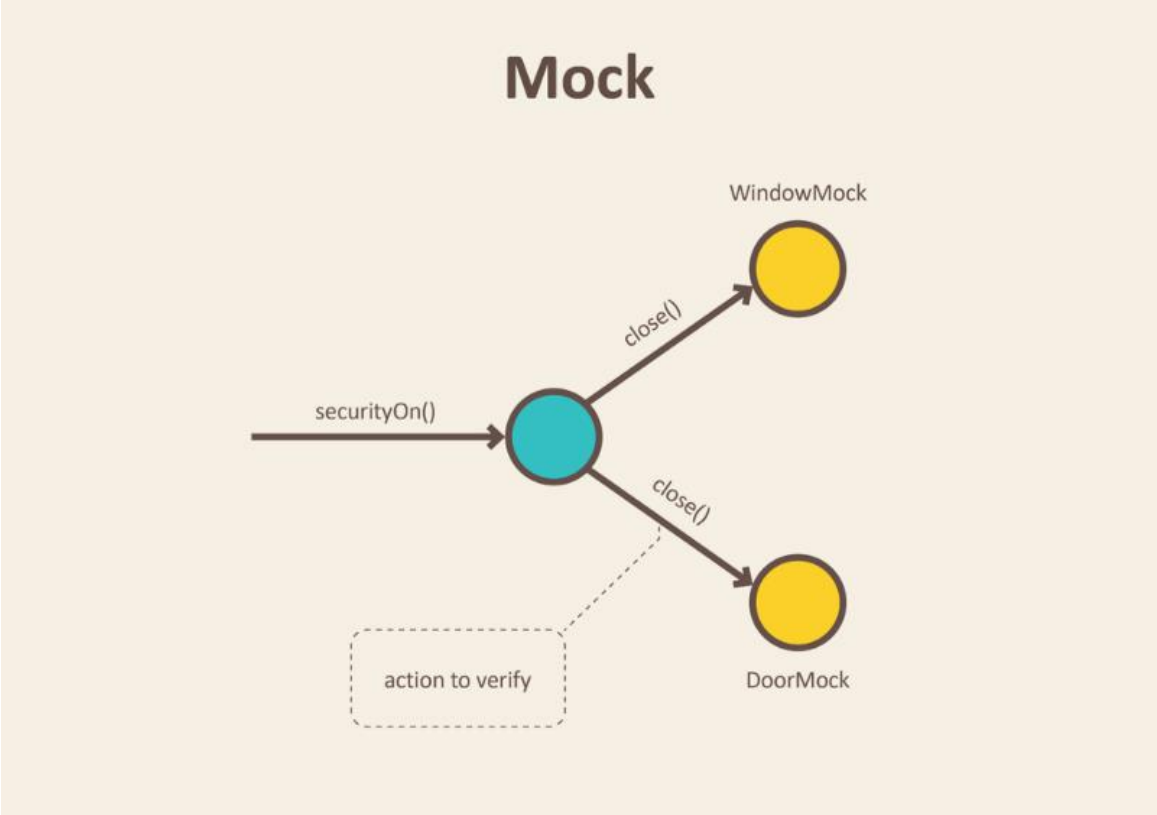
- ***Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.***

- A stub has no logic, and only returns what you tell it to return. Stubs can be used when you need an object to return specific values in order to get your code under test into a certain state.



Mock

- ***We use mocks when we don't want to invoke production code or when there is no easy way to verify, that intended code was executed.***
- ***We don't want to send e-mails each time we run a test. Moreover, it is not easy to verify in tests that a right email was send. Only thing we can do is to verify the outputs of the functionality that is exercised in our test. In other worlds, verify that e-mail sending service was called.***



Collaborators

A Collaborator Object is a unit of code whose responsibility is to orchestrate the invocation of other units.

- Creating dependencies for unit testing. It is common in unit tests to mock or stub collaborators of the class under test so that the test is independent of the implementation of the collaborators.

Example

```
public class LoginUseCaseSync {

    public enum UseCaseResult {...}

    /**
     * Collaborators
     */
    private final LoginHttpEndpointSync mLoginHttpEndpointSync;
    private final AuthTokenCache mAuthTokenCache;
    private final EventBusPoster mEventBusPoster;

    public LoginUseCaseSync(LoginHttpEndpointSync loginHttpEndpointSync,
                            AuthTokenCache authTokenCache,
                            EventBusPoster eventBusPoster) {
        mLoginHttpEndpointSync = loginHttpEndpointSync;
        mAuthTokenCache = authTokenCache;
        mEventBusPoster = eventBusPoster;
    }
```

Unit test Class Body

Each unit test class must contain a setup method. The object name of the class that is under test will be named as SUT which stands for System Under Test.

Template of class body

```
/**
 * Created by Anjan Debnath on 7/19/2018.
 * Copyright (c) 2018, W3 Engineers Ltd. All rights reserved.
 */
@RunWith(MockitoJUnitRunner.class)
public class LocatorServiceTest {

    constants

    helper fields

    LocatorService SUT;

    @Before
    public void setup() throws Exception {

        //Creating an similar object and share for all @Test
        SUT = new LocatorService();

    }

    @Test
    public void test1() {
        ArgumentCaptor<Point> ac = ArgumentCaptor.forClass(Point.class);
        SUT.geoLocate(ac.capture());
    }

    helper methods

    helper classes
```

Unit test name pattern

<unitOfWork>_<stateUnderTest>_<expectedBehaviour>

- unitOfWork = Method Name
- stateUnderTest = what is testing
- expectedBehaviour = return value of the test

Example

Reverse_emptyString_emptyStringReturned()

Method body

@Test
public void test_customerIdPassed_nameReturned() {

//arrange

//action

//assertion

}

Arrange

On this part we will arrange all type of task that will needed to verify or sub.

Action

Method that is under system test.

Assertion

All type of assertion related tasks

Example

```
@Test
public void test_customerIdPassed_nameReturned() {

    //arrange
    Customer customer = new Customer();
    customer.setFirstName(FIRST_NAME);
    customer.setLastName(LAST_NAME);

when(dbManagerMock.findCustomer(CUSTOMER_ID)).thenReturn(
customer);
    //action
    String name = SUT.findName(CUSTOMER_ID);
    //assert
    assertThat(name, is(equalTo(CUSTOMER_NAME)));
}
```

Object vs Data Structure

Object

Expose Behavior

Hide implementation detail

Should be injected into other objects

Need to be unit tested explicitly.

Data Structure

Expose Data

No Behavior

No need to be substituted with test doubles

Reference

- https://hackernoon.com/objects-vs-data-structures-e380b962c1d2
- https://www.codingblocks.net/podcast/objects-vs-data-structures/

Test Driven Development TDD

Please follow Uncle Bob's TDD rule to properly maintain your unit test.