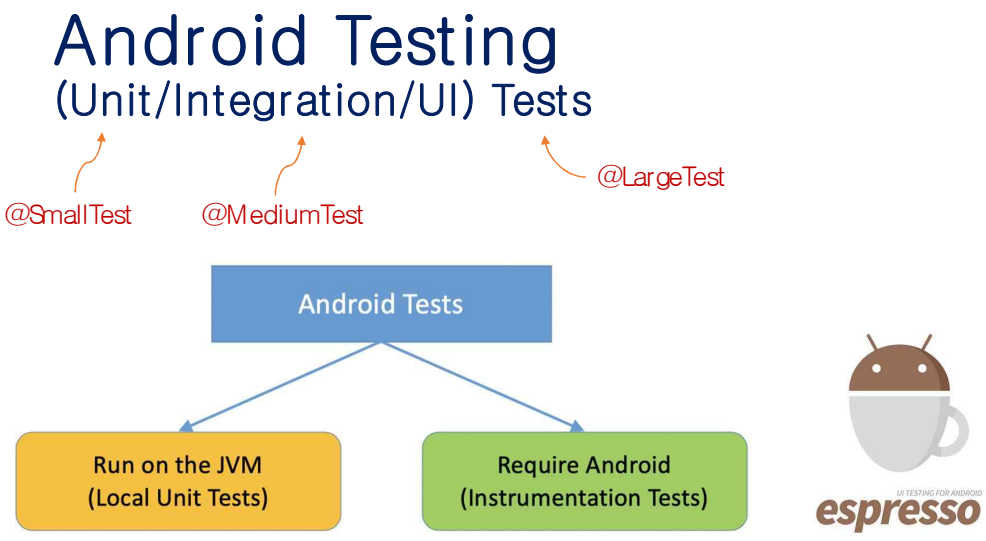
Android App Testing

(Unit/Integration/UI) Tests

Course Contents



Prerequisites

You are expected to have working knowledge of …

- JUnits
- Mockito

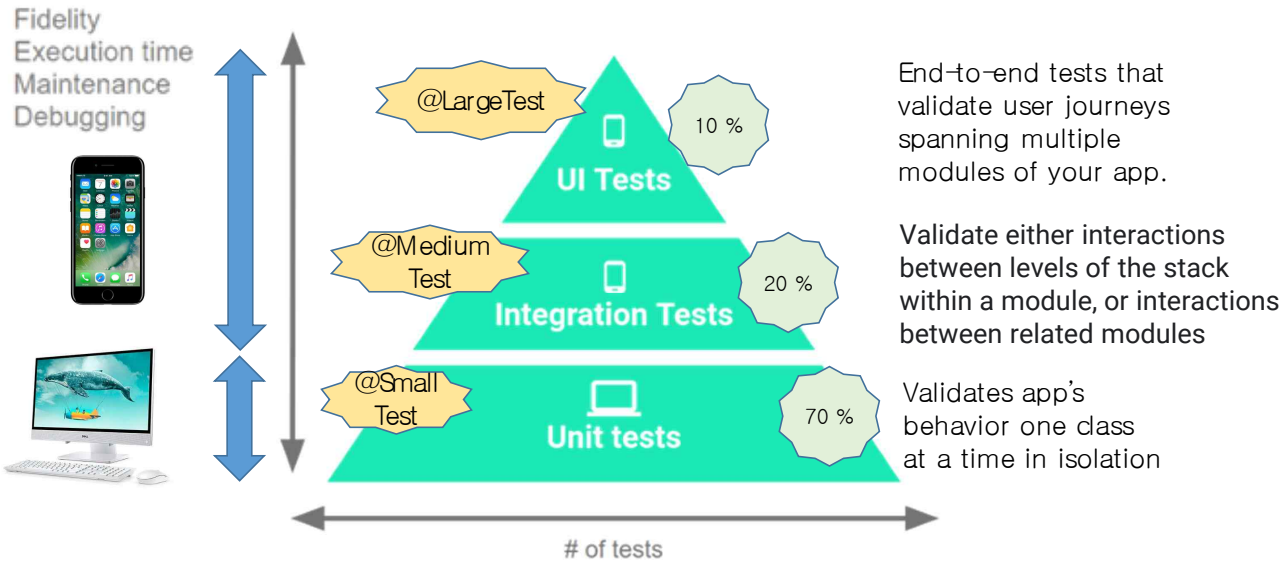
Types of Software Testing

- Integration Testing
- Load Testing
- Acceptance Testing
- Monkey Testing
- Block Box Testing
- White Box Testing
- Performance Testing
- Function Testing
- Alpha Testing
- Beta Testing
- System Testing
- Gorilla Testing
- Acceptance Testing
- Security Testing
- Compatibility Testing
- White Box Testing
- Regression Testing
- Sanity Testing
- Usability Testing
- Negative Testing
- End-to-End Testing
- Stress Testing
- Property-based Testing
- Security Testing
- Volume Testing
- Mutation Testing
- …

Unit/ Integration/ End-to-End Testings

- **Unit Testing**
  - testing individual methods or classes of an application in isolation to confirm that the code is doing things right
- **Integration Testing**
  - checking if different classes are working fine when combined together as a group
- **End-to-End Testing (aka Functional testing)**
  - testing a slice of application's functionality to confirm that the entire application is working as intended

Test Coverage: Levels of Testing Pyramid



Confusing terminology

https://developer.android.com/training/testing/unit-testing

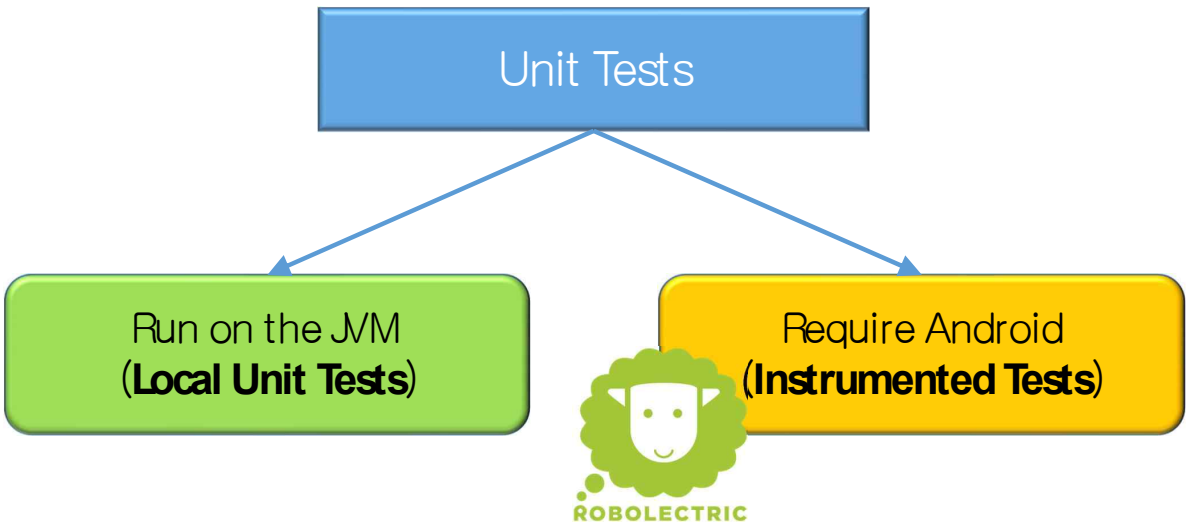
Build effective unit tests

Unit tests are the fundamental tests in your app testing strategy. By creating and running unit tests ...

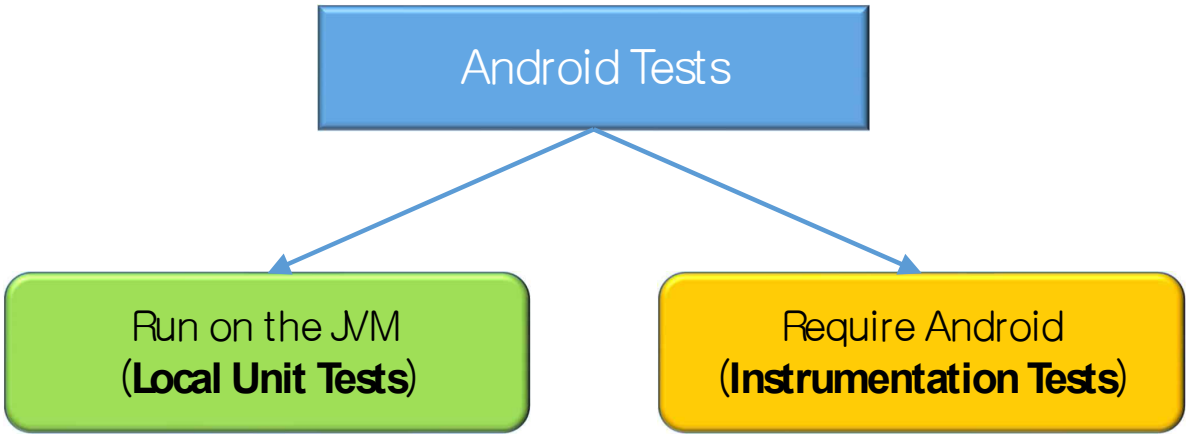
For testing Android apps, you typically create these types of automated unit tests:

- **Local tests:** Unit tests that run on your local machine only. These tests are compiled to run locally on the Java Virtual Machine (JVM) to minimize execution time. If your tests depend on objects in the Android framework, we recommend using Robolectric. For tests that depend on your own dependencies, use mock objects to emulate your dependencies' behavior.
- **Instrumented tests:** Unit tests that run on an Android device or emulator. These tests have access to instrumentation information, such as the `Context` for the app under test. Use this approach to run unit tests that have complex Android dependencies that require a more robust environment, such as Robolectric.

Types of Android Tests



Types of Android Tests: My View



Unit Testing Tools for Android App

- **JUnit4/ JUnit5**
  - normal test assertions
- **Mockito/ MockK**
  - mocking out other classes that are not under test
- **PowerMock**
  - mocking out static classes
- **Robolectric**
  - simulate Android Framework

Instrumented & UI testing Tools for Android

- **Espresso**
  - Used for testing within your app, selecting items, making sure something is visible, etc.
- **UIAutomator**
  - Used for testing interaction between different apps.
- Other tools
  - Appium, Calabash, Robotium, etc.

Why Write Tests?

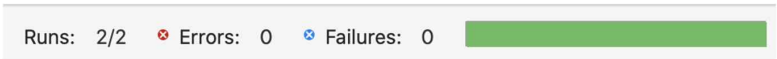
- To find the bug that may exist in our code?

- To test the functionality of our code, i.e., whether our code is working as expected or not.

- To refactor the code for evolution

More on Why Testing?

- Testing forces you to think in a different way and implicitly makes your code cleaner in the process.
- You feel more ***confident*** about your code if it has tests.
- Regression testing is made a lot easier, as automated tests would pick up the bugs first.
- Executable *live* documents!
- Shiny green status bars and cool coverage reports are added bonus!

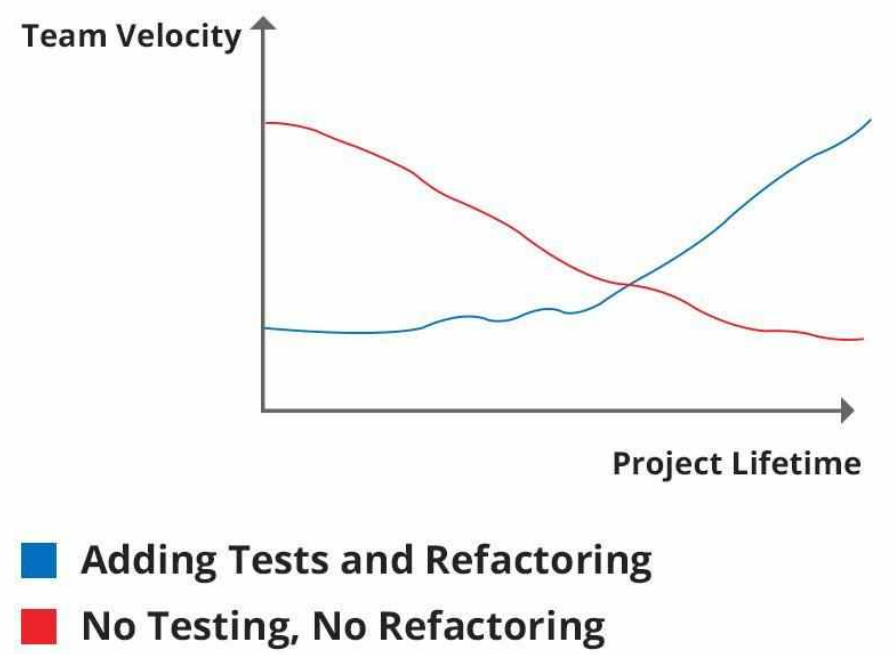


Advantages of Testing

Testing also provides you with the following advantages:

- **Rapid feedback** on failures.
- **Early failure detection** in the development cycle.
- **Safer code refactoring**, letting you optimize code without worrying about regressions.
- **Stable development velocity**, helping you minimize technical debt.

Google



Legacy code is code without tests

"Code without tests is bad code"

– Michael Feathers

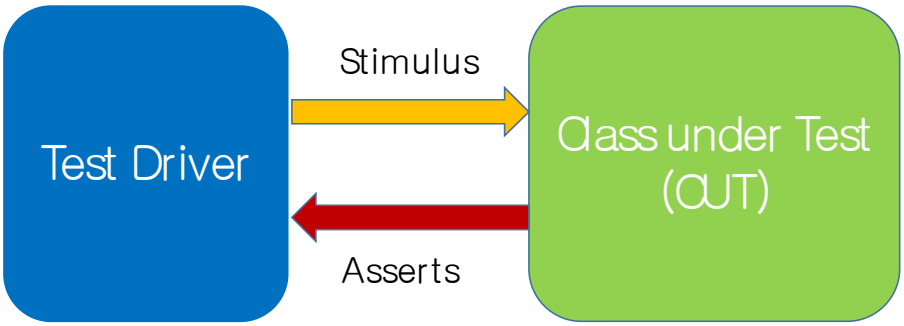
Unit Tests

A unit test is an ***automated test***
that tests a ***unit***
in ***isolation*** from its dependencies.

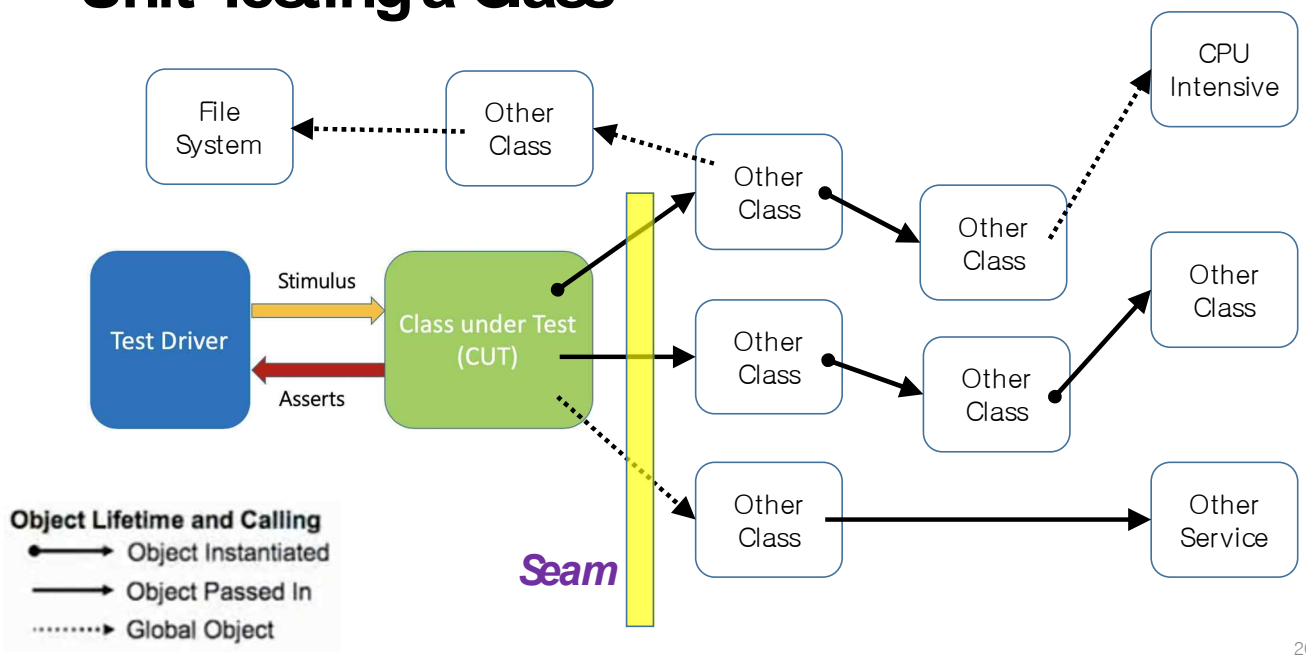
What is a unit?

- "The smallest component that it makes sense to test"
- Unit for testing depends on individual programmers or teams
- Generally, a unit means
  - class or an interface
  - a single method or function.

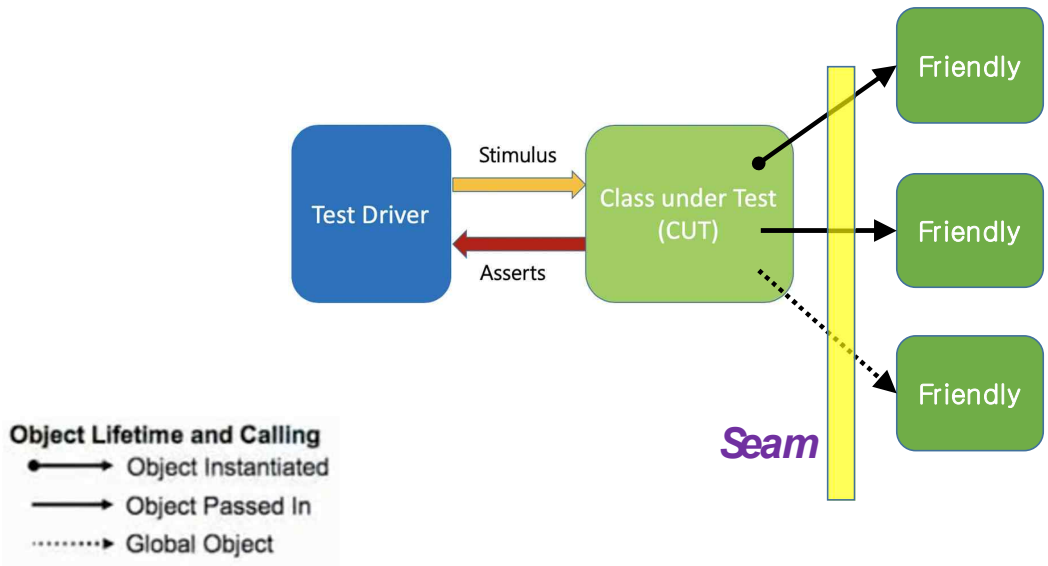
Unit Testing a Class



Unit Testing a Class



Unit Testing a Class



Microsoft C# Peoples

- ***Fake*** –A fake is a generic term which can be used to describe *either a stub or a mock* object. Whether it is a stub or a mock depends on the context in which it's used. So in other words, a fake can be a stub or a mock.
- ***Mock*** –A mock object is a fake object in the system that decides whether or not a unit test has passed or failed. A mock starts out as a Fake until it is asserted against.
- ***Stub*** –A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly. By default, a fake starts out as a stub.

Is this a Mock?

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```


A little Better!

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

To use it as a Mock …

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

- In this case, you are checking a property on the Fake (asserting against it), so in the above code snippet, the `mockOrder` is a Mock.

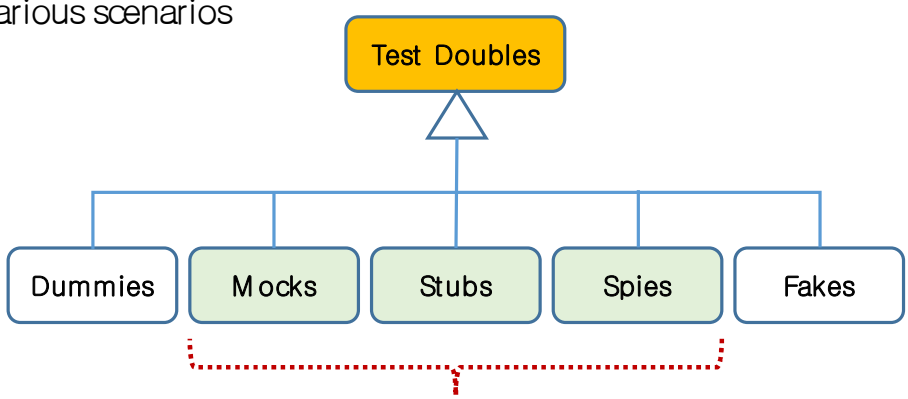
> ⓘ **Important**
>
> It's important to get this terminology correct. If you call your stubs "mocks", other developers are going to make false assumptions about your intent.

The main thing to remember about mocks versus stubs is that mocks are just like stubs, but

- you assert against the mock object, whereas
- you do not assert against a stub.

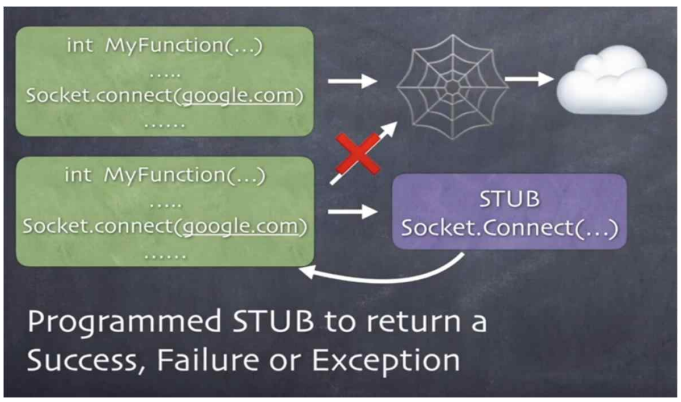
Test Doubles

- Used in lieu of external dependencies
  - DB, Web, API, Library, Network etc.
  - Easy to simulate various scenarios



Stubs

- Generates predefined outputs
- Does not provide validation of how the class uses the dependency
- Used when data is required by the class but the process used to obtain it isn't relevant to what's being tested
- Usually created using a mock framework

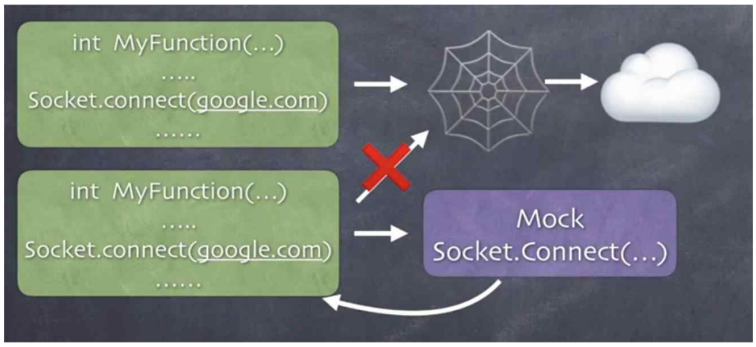


- Returns success, failure or exceptions (as coded)

Checks the behavior of code under test in case of these return values

Mocks

- Mocks replaces external interface
- Mechanism for validating how a dependency is used by the class
- Can provide data required by the class (by stubbing)
- Created by a mocking framework

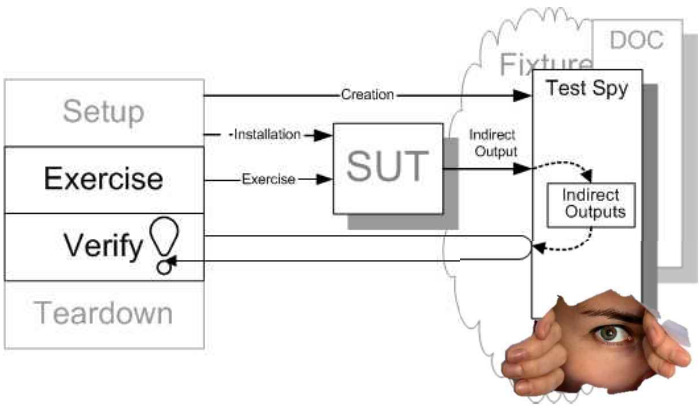


- Mocked function called or NOT?
- How many times it gets called?
- What parameters are passes when it was called?

Right call, Right #of times with Right setup parameters

Spy

- A stand-in for DOC used by SUT
- Creating a spy requires a real object to spy on
- Might be useful for testing legacy code ("partial mock")
- Consider using mocks instead of spies whenever possible.
- Usually created using a mock framework

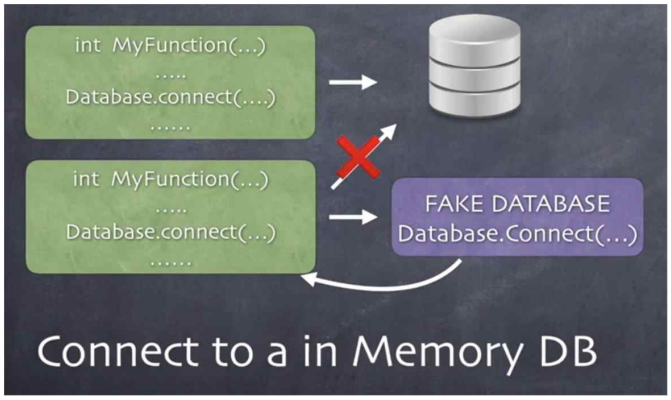


- By default, a spy delegates all method calls to the real object and records what method was called and with what parameters.
- Can selectively stub methods

Like Mocks and Stubs, normally used for **behavior verification** of SUT.

Fakes

- Almost working simplified implementation
- Usually coded directly, without the use of a framework
- Does not provide direct validation of how the class uses dependency
- Used when the class being tested requires a specific logic in the dependency



- Instead of actually going to the internet, it connects to a local (limited) implementation
- Created specifically for this test

Check the behavior with respect to actual (potentially lots of) data it receives.

A Set of Unit Testing Rules

A test is not a unit test if:

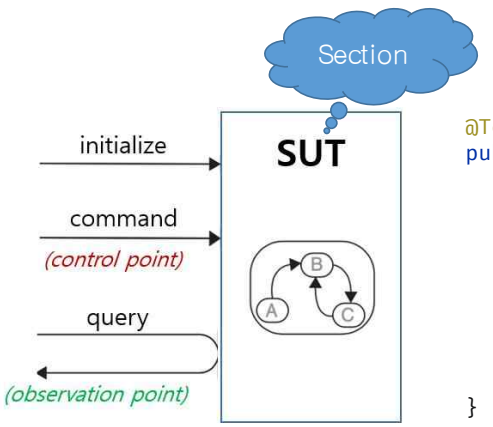
- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

How to do unit testing?

A unit test typically features three different phases (**AAA**):

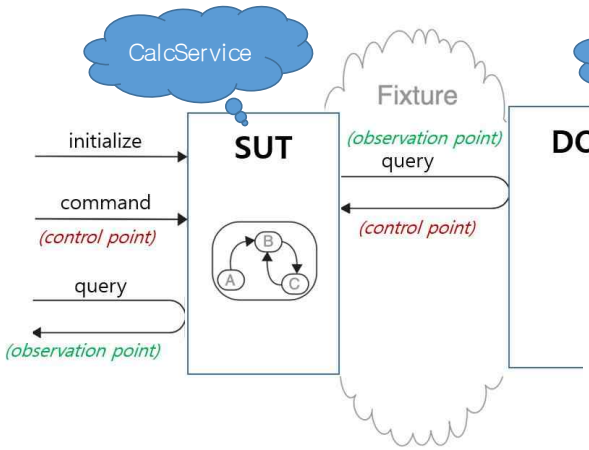
• **Arrange** (Given) • ***Preparation***	**Act** (When) ***Execution***	**Assert** (Then) ***Assertion***
• An SUT initialization • Stubs/Mocks creation • Stubbing and Injection *"Given user is logged in …"*	• An operation to test in a given test *"When user launches app …"* *"When user clicks Log Out …"*	• Received result verification • Mocks verification (if needed) *"Then user sees logout text"*

System Under Test (SUT) vs. Depended-On Objects (DOC)



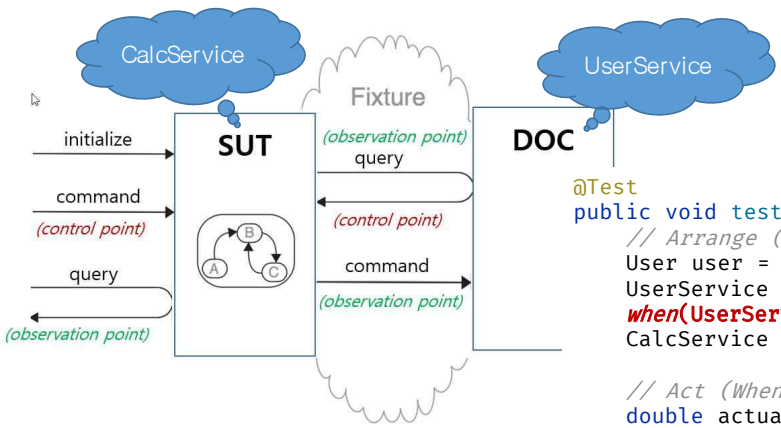

```
@Test
public void should_increment_students_if_a_new_student_enrolled() {
    // Given
    Section section = new Section(course, 2020, Semester.Spring);

    // When
    section.addStudent(student);

    // Then
    List<Student> students = section.getStudents();
    assertThat(students, hasSize(1));
}
```

System Under Test (SUT) vs. Depended-On Objects (DOC)



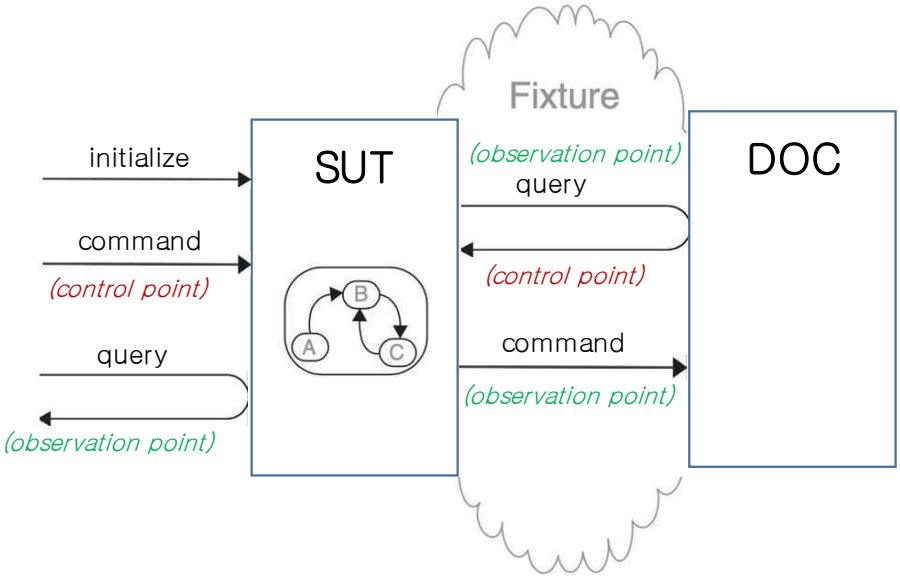

```
@Test
public void testCalc() {
    // Arrange (Given)
    AddService addService = mock(AddService.class);
    when(addService.add(1, 2)).thenReturn(3);
    CalcService calcService = new CalcService(addService);

    // Act (When)
    int actual = calcService.calc(1, 2);

    // Assert (Then)
    assertEquals(6, actual);
    verify(addService).add(1, 2);
}
```


System Under Test (SUT) vs. Depended-On Objects (DOC)




```
@Test
public void testCalc() {
    // Arrange (Given)
    User user = new User(new User("id001"));
    UserService userService = mock(UserService.class);
    when(UserService.find("id001")).thenReturn(user);
    CalcService calcService = new CalcService(UserService);

    // Act (When)
    double actual = calcService.calc("id001");

    // Assert (Then)
    assertEquals(180d, actual, 0.1);
    verify(userService).find(anyString());
    verify(userService).save(user);
}
```

System Under Test (SUT) vs. Depended-On Objects (DOC)



How to write Good Tests?

Unfortunately, well …virtually *nothing*!

There is no secret knowledge of how to write good tests …

…except for a couple of tips about styles and general tactics …

Trouble writing tests?

- The problem's not in your test suite.
- It's in your code.

Common Warning Signs of Hard to Test Codes

- Static Properties and Fields
- Singletons
- Static Methods
- The new Operator
- Work in constructor
- ...

Testing is not Hard!

Testing is Easy in the presence of Good Design.

What makes a good unit test?

- A "good" unit test follows these rules:
  1. **The test only fails when a new bug is introduced into the system or requirements change**
  2. **When the test fails, it is easy to understand the reason for the failure.**

Traits of Good Unit Tests

- **F**ast
- **I**solated/Independent
- **R**epeatable
- **S**elf-Validating
- **T**horough
- Trustworthy

A good rule of thumb is to not trust a test that has never failed

After all, writing tests is just like writing production code …But

- Unit testing code is production code that you will need to maintain, refactor and build upon for years to come.

- The rules that apply for writing good production code do NOT always apply to creating a good unit testing.

- Do not fall into the trap of following best practices for writing production code that are not appropriate for writing unit tests.

The Two Least Known Facts of Unit Testing

1. Test do not share instance data.

```
public class ListTest {
    private List<String> list = new ArrayList<>();

    @Test
    public void testAdd() {
        list.add("Foo");
        Assert.assertEquals(1, list.size());
    }

    @Test
    public void testAdd2Elements() {
        list.add("Baz");
        list.add("Baz");
        Assert.assertequals(2, list.size());
    }
}
```

2. You can have many test classes per model class

- Do not feel compelled to stuff all your tests for **Foo** into **FooTest**.
- Every test that needs a slightly different set up can go into a separate test class.

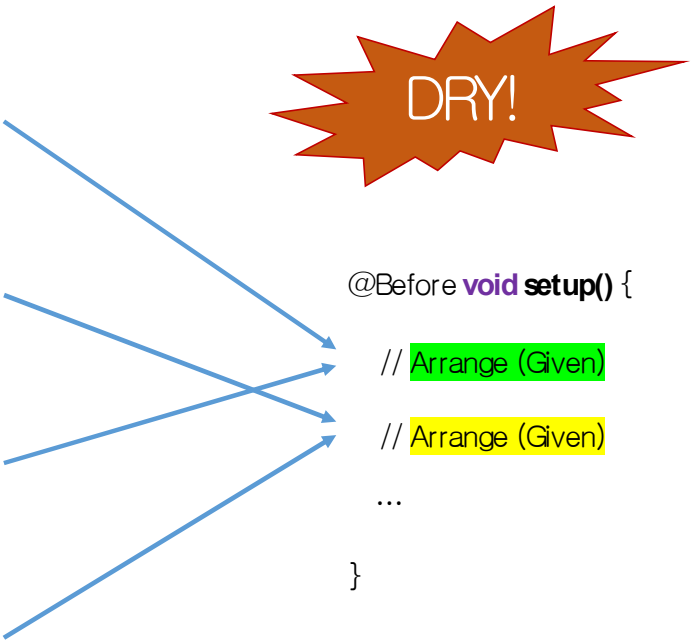
Structure of a JUnit test class

- To test a class named **Foo**, create a test class **FooTest**

```
class FooTest {
  @Test void test() {…}
  …
}
```

```
class FooTest {
  @Test void test1() {…}
  @Test void test2() {…}
  @Test void test3() {…}
  @Test void test4() {…}
  @Test void test5() {…}
  @Test void test6() {…}
  @Test void test7() {…}
  …
}
```

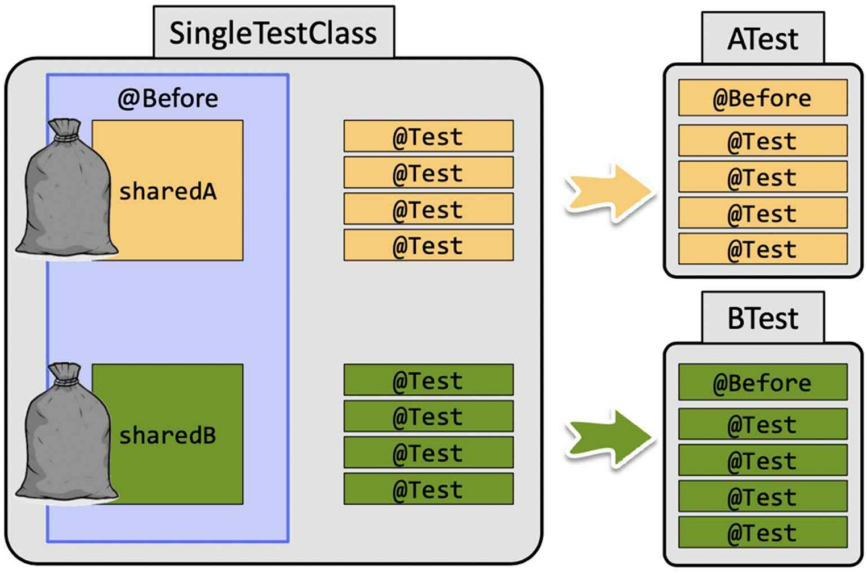
```
@Test void test1() {
  // Arrange (Given)
  ...
}

@Test void test2() {
  // Arrange (Given)
  ...
}

@Test void test3() {
  // Arrange (Given)
  ...
}

@Test void test4() {
  // Arrange (Given)
  ...
}
```




```
@Before void setup() {

  // Arrange (Given)

  // Arrange (Given)

  ...

}
```


Split Test Classes per Production Class

- If it grows, or
- If many tests share different fixtures



Build Local Unit Tests

- Store the source files for local unit tests at ***module-name***/src/test/java/
- In your app's top-level build.gradle file,

```
dependencies {
    // Required -- JUnit 4 framework
    testImplementation 'junit:junit:4.13’

    // Optional -- Mockito framework
    testImplementation 'org.mockito:mockito-core:3.5.5'
}
```