Use test doubles in Android

When designing the testing strategy for an element or system, there are three related testing aspects:

- **Scope**: How much of the code does the test touch? Tests can verify a single method, the entire application, or somewhere in between. The tested scope is *under test* and commonly refer to it as the *Subject Under Test*, though also the *System Under Test* or the *Unit Under Test*.
- **Speed**: How fast does the test run? Test speeds can vary from milli-seconds to several minutes.
- **Fidelity**: How "real-world" is the test? For example, if part of the code you're testing needs to make a network request, does the test code actually make this network request, or does it fake the result? If the test actually talks with the network, this means it has higher fidelity. The trade-off is that the test could take longer to run, could result in errors if the network is down, or could be costly to use.

See what to test (/training/testing/fundamentals/what-to-test) to learn how to start defining your test strategy.

Isolation and dependencies

When you test an element or system of elements you do it in *isolation*. For example, to test a ViewModel you don't need to start an emulator and launch a UI because it doesn't (or shouldn't) depend on the Android framework.

However, the subject under test might *depend* on others for it to work. For instance, a ViewModel might depend on a data repository to work.

When you need to provide a dependency to a subject under test, a common practice is to create a *test double* (or *test object*). Test doubles are objects that look and act as components in your app but they're created in your test to provide a specific behavior or data. The main advantages are that they make your tests faster and simpler.

Types of test doubles

There are various types of test doubles:

Fake	A test double that has a "working" implementation of the class, but it is implemented in a way that makes it good for tests but unsuitable for production. Example: an in-memory database. Fakes don't require a mocking framework and are lightweight. They are **preferred**.
Mock	A test double that behaves how you program it to behave and that has expectations about its interactions. Mocks will fail tests if their interactions don't match the requirements that you define. Mocks are usually created with a *mocking framework* to achieve all this. Example: Verify that a method in a database was called exactly once.
Stub	A test double that behaves how you program it to behave but doesn't have expectations about its interactions. Usually created with a mocking framework. Fakes are preferred over stubs for simplicity.
Dummy	A test double that is passed around but not used, such as if you just need to provide it as a parameter. Example: an empty function passed as a click callback.
Spy	A wrapper over a real object which also keeps track of some additional information, similar to mocks. They are usually avoided for adding complexity. Fakes or mocks are therefore preferred over spies.
Shadow	Fake used in Robolectric.

Caution: There are conflicting definitions depending on the source. A comprehensive source is *Mocks aren't Stubs* (https://martinfowler.com/articles/mocksArentStubs.html) by Martin Fowler.

Note: When using a library or framework, check with the authors to see if they provide any officially-

supported testing infrastructures, such as fakes, that you can reliably depend on.

Example using a fake

Suppose that you want to unit test a ViewModel that depends on an interface called `UserRepository` and exposes the name of the first user to a UI. You can create a fake test double by implementing the interface and returning known data.

```
object FakeUserRepository : UserRepository {
    fun getUsers() = listOf(UserAlice, UserBob)
}

val const UserAlice = User("Alice")
val const UserBob = User("Bob")
```

This fake `UserRepository` does not need to depend on the local and remote data sources that the production version would use. The file lives in the test source set and will not ship with the production app.

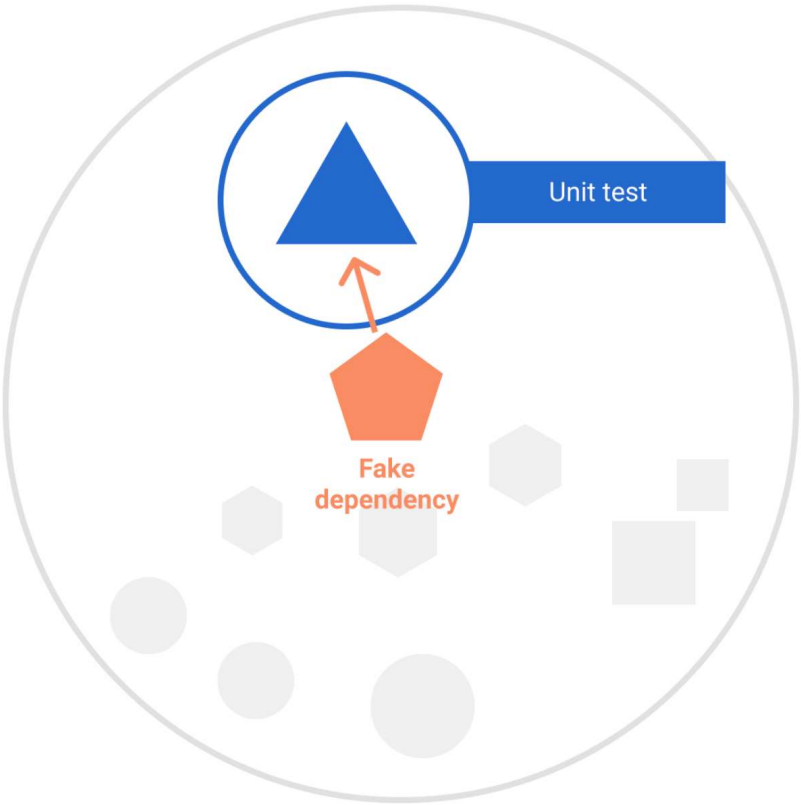

***Figure 1**: A fake dependency in a unit test.*

The following test verifies that the ViewModel correctly exposes the first user name to the view.

```
@Test
fun viewModelA_loadsUsers_showsFirstUser() {
    // Given a VM using fake data
    val viewModel = ViewModelA(FakeUserRepository) // Kicks off data load on ini

    // Verify that the exposed data is correct
    assertEquals(viewModel.firstUserName, UserAlice.name)
}
```

Replacing the `UserRepository` with a fake is easy in a unit test, because the ViewModel is created by the tester. However, it can be challenging to replace arbitrary elements in bigger tests.

Replacing components and dependency injection

When tests have no control over the creation of the systems under test, replacing components for test doubles becomes more involved and requires the architecture of your app to follow a *testable* design.

Even big end-to-end tests can benefit from using test doubles, such as an instrumented UI test that navigates through a full user flow in your app. In that case you might want to make your test *hermetic*. A hermetic test avoids all external dependencies, such as fetching data from the internet. This improves reliability and performance.

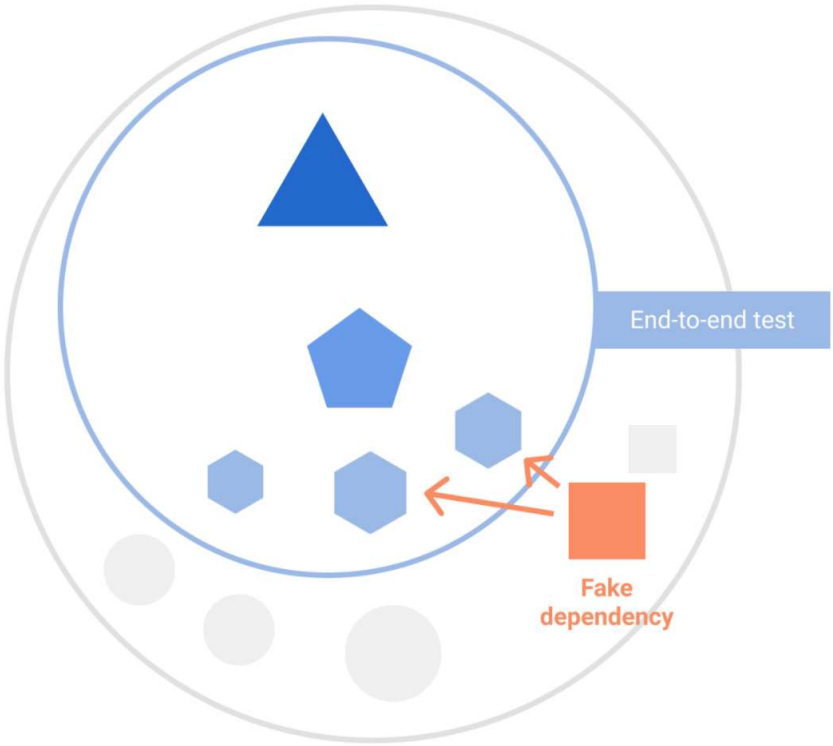

***Figure 2**: A big test that covers most of the app and fakes remote data.*

You can design your app to achieve this flexibility manually, but we recommend using a dependency injection (/training/dependency-injection) framework like Hilt (/training/dependency-injection/hilt-android) to replace components in your app at test time. See the Hilt testing guide. (/training/dependency-injection/hilt-testing)

Robolectric

On Android, you can use the Robolectric (http://robolectric.org/) framework, which provides a special type of test double. Robolectric lets you run your tests on your workstation, or on your continuous integration environment. It uses a regular JVM, without an emulator or device. It simulates inflation of views, resource loading, and other parts of the Android framework with test doubles called *shadows*.

Robolectric is a simulator so it should not replace simple unit tests or be used to do compatibility testing. It provides speed and reduces cost at the expense of lower fidelity in some cases. A good approach for UI tests is to make them compatible with both Robolectric and instrumented tests and decide when to run them depending on the need to test functionality or compatibility. Both Espresso and Compose tests can run on Robolectric.