Unit Testing with Dependencies

"Code without tests is broken by design"

This is a quote by Jacob Kaplan-Moss, a core developer of the Django Web framework, which is written in Python. He emphasizes how important testing is. *Testing gives you predictability and confidence.* You can have confidence in the software continuing to work as it should. Of course, bugs will sneak in, but with good testing, they will not make it to production.

Mock Objects for Testing

One thing I didn't talk about in the lectures is *mocking* in testing. Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. During testing, the mock objects are used to replace dependencies (other APIs or interfaces) that cannot be controlled under normal testing conditions, or would be too resource-heavy for reasonable testing purposes.

A few examples of things the code might depend on that need to be tested, but can't under normal circumstances:

- Time-related tests. If the code uses time-related interfaces such as the current date/time, it will need to be tested for things like daylight savings changes or leap years and leap seconds. Date/time issues become even more complicated if we have to handle different time zones.
- Error scenarios. Perhaps the code is supposed to handle certain network errors - how can we test this in a unit test when we can't predict when there will be a network error?
- Slow interactions. For example, getting data from a big, complicated database where each test would need to spend a lot of time and resources creating a specific scenario.

In each of these examples, it is difficult or impossible to be sure that you can test all the scenarios and special cases with the normal unittest methods.

In addition, if a function being tested does rely on a dependency, if something goes wrong with the dependent interface, we only know our test failed, but not the reason why. It would be possible that the function being tested works just fine, but the test fails because of the dependency's failure.

This is where *mocking* comes in. The tests will add a "mock object" that will be executed instead of the dependency to give the function being tested the answer it needs to be able to test the special cases and have confidence that when the dependency is working correctly, the function we are testing also works correctly.

You will also see the terms *stub* and *fake* used; the terminology is a little fuzzy, but they are all for the same purpose, namely, to provide a way to test specific situations without having to be concerned about dependencies.

As a simple example, consider the error scenario mentioned above. Say we want to test a function called `get_data` that involves accessing data across a network, and it has functionality to handle the situation where there is a network error. Most of the unittests for `get_data` can be created normally. In order to test the edge case of the network error, we would create a *mock object* that represents the network call. This mock object only exists for this test case. When the test case is run, the mock object is called instead of the real network call, and it returns the network error condition we are testing. Note that our function `get_data` has no knowledge that it isn't performing a real network communication.

In Python, the `unittest` module has a sub-module `mock` for testing. It allows you to replace some of your dependencies

used by the system under test with mock objects. The details of implementing mocking are beyond the scope of this class, but you should be aware of the existence and importance of mocking in testing.