SUBJECT7
QUALITY IS A CULTURE

# When to use Mocking – Manager's edition

*By Matthew Heusser for Subject7*

During our recent webinar on revisiting the test automation pyramid, an attendee named Stephen asked for some guidance on when mocking was good or bad. The good folks at Subject7 asked me to take a look at the question and offer my perspective.

## Mocking Defined

A strict definition of a mock is a bit of code that replaces an object. Programmers writing code to test an object they are writing might be interested in how that object interacts with the world. They can "mock out" the database or other objects, and test just their code in isolation. This has the side benefit of being much faster, as external components, like databases and file systems, tend to be much slower than pure code running in memory.

Today we'll use a slightly wider version of "mock" to include replacing *any* subsystem that you are dependent on with something stable and predictable. The simplest example of this is probably e-commerce, where we mock out the credit card processing system and the manufacturing resource planning system.

With these subsystems mocked out, an automated test can run end to end without placing an order on the test warehouse system (which might be down for maintenance or being used for other testing which may cause conflicts). If these mocks have an exposed user interface, a GUI test could check if that order number appears in the mock warehouse, verify the credit card was submitted, and even have the fake credit card system return success for one specific credit card number and reject another. This makes it possible to test credit card submissions quickly, internally, without relying on a vendor.

## When to Mock

Most of the reasons to mock are outlined above. At the programmer level, they make code testable without needing to actually have an order in the database — or even a database at all. Having the mock respond to specific hard-coded values makes testing easier and more consistent, and the responses can be faster.

It's easy to see why to mock. Knowing *when* to mock is a little more difficult.

My simple answer is "when you find yourself relying on some external system that you do not really need to test, is tested elsewhere, does not need to be tested every time, or is too slow." If you want to test 10,000 combinations of esoteric orders, and the external credit card validator takes 10 seconds and everything else takes milliseconds … it might be time to consider mocking.

One interesting case is the ability to separate user interface from business logic. Say, for example, that you separate the main logic of the user interface from the GUI itself. That means the code consists of web pages (HTML) and a little glue code (JavaScript) that calls APIs. Suddenly it is possible to mock out the APIs and test the JavaScript incredibly quickly. On the other hand, now we can test the APIs themselves in isolation. And those APIs can be tested a few ways: end to end by a tool like Subject7, or as units, probably mocking out the database.

## When not to Mock

When mocking became popular, we had a strange explosion of tests that, well … didn't really check much. The "tests" would assume how the warehouse would work, the warehouse would change, but the assumptions would not, and we would suddenly have a problem. On more than one project I was known to sing the songs from the Burger King ads of my childhood, "Ain't Nothing Like the Real Thing."

In other words, it's OK to mock. But actually test the software, friend.

Most modern software tests are really "glue": a pretty front-end that connects to a bunch of disconnected back-end services. If you have a staging or pre-production test environment, by all means, test everything integrated together. Likewise, when it comes to code, you can have unit tests that are isolated if there is some compelling reason for it. This can save some time compared to tests that hit an external boundary like the network, the file system, or databases. Just don't forget to actually test it, and don't forget that the mocks have a purpose.

Writing code designed to use mocks will tend to create nice seams in the software so the mocks can be substituted out for the real thing. This tends to make the code more testable and, when it comes time to replace a component system, like a database, with another brand, they can help reduce the switching code.

Mocks can speed you up. Just don't let them trip you up, and you'll be fine.