Definition of Terms

The following was written by Robert C. Martin (Uncle Bob) and appears in his blog at <http://blog.cleancoder.com/uncle-bob/2017/05/05/TestDefinitions.html>.

[[](http://blog.cleancoder.com/)](http://blog.cleancoder.com/)

# The Clean Code Blog

by Robert C. Martin (Uncle Bob)

# First-Class Tests.

05 May 2017

I believe it may be my fate to find blogs written by people who have fallen prey to unfortunate disciplines that have led them to give up on unit testing. [This blog](https://techbeacon.com/1-unit-testing-best-practice-stop-doing-it) is just another one of those.

The author tells of how his unit tests are fragile because he mocks out all the collaborators. (sigh). Every time a collaborator changes, the mocks have to be changed. And this, of course, makes the unit tests fragile.

The author goes on to tell us how he abandoned unit testing, and started doing, what he called “System Testing” instead. In his vocabluary, “System Tests” were simply tests that are more end-to-end than “unit tests”.

So first, some definitions. Pardon me for my hubris, but there are so many different definitions of “unit test” and “system test” and “acceptance test” out there that it seems to me someone ought to provide a single authoritative definition. I don’t know if these definitions will stick; but I hope some set of definitions does in the near future.

* Unit Test: A test written by a programmer for the purpose of ensuring that the production code does what the programmer expects it to do. (For the moment we will ignore the notion that unit tests also aid the design, etc.)
* Acceptance Test: A test written by the business for the purpose of ensuring that the production code does what the business expects it to do. The authors of these tests are business people, or technical people who represent the business. i.e. Business Analysts, and QA.
* Integration Test: A test written by architects and/or technical leads for the purpose of ensuring that a sub-assembly of system components operates correctly. These are plumbing tests. They are not business rule tests. Their purpose is to confirm that the sub-assembly has been integrated and configured correctly.
* System Test: An integration test written for the purpose of ensuring that the internals of the whole integrated system operate as planned. These are plumbing tests. They are not business rule tests. Their purpose is to confirm that the system has been integrated and configured correctly.
* Micro-test: A term coined by Mike Hill (@GeePawHill). A unit test written at a very small scope. The purpose is to test a single function, or small grouping of functions.
* Functional Test: A unit test written at a larger scope, with appropriate mocks for slow components.

Given these definitions, the author of the above blog has given up on badly written micro-tests, in favor of badly written functional tests. Why badly written? Because in both cases the author describes these tests as coupled to things that they should not be coupled to. In the case of his micro-tests, he was using too many mocks, and deeply coupling his tests to the implementation of the production code. That, of course, leads to fragile tests.

In the case of his functional tests, the author described them as going all the way from the UI to the Database; and made reference to the fact that they were slow. He cheered the notion that his tests could sometimes run in as little as 15 minutes. 15 minutes is much too long to wait for the kind of rapid feedback that unit tests are supposed to give us. TDDers are not in the habit of waiting for the continuous build system to find out if the tests pass.

Skilled TDDers understand that neither micro-tests, nor functional tests, (nor Acceptance tests for that matter) should be coupled to the implementation of the system. They should (as the blog’s author urges us) be considered as part of the system and “…treated like a first-class citizen: [They] should be treated in the same way as one would treat production code.”

What the blog author does not seem to recognize is that first class citizens of the system should not be coupled. Someone who treats their tests like first class citizens will take great pains to ensure that those tests are not strongly coupled to the production code.

Decoupling micro-tests and functional tests from the production code is not particularly difficult. It does require some skill at software design; and some knowledge of decoupling techniques. Generally, a good dose of OO design and dependency inversion, along with the judicious use of a few Design Patterns (like Facade and Strategy) are sufficient to decouple even the most pernicious of tests.

Unfortunately, too many programmers think that the rules for unit tests are different – that they can be “junk code” written using any ad hoc style that they find convenient. Also, too many programmers have read the books on mocking, and have bought in to the notion that mocking tools are an intrinsic, and necessary, part of unit testing. Nothing could be further from the truth.

I, for example, seldom use a mocking tool. When I need a mock (or, rather, a Test Double) I write it myself. It’s not very hard to write test doubles. My IDE helps me a lot with that. What’s more, writing the Test Double myself encourages me not to write tests with Test Doubles, unless it is really necessary. Instead of using Test Doubles, I back away a bit from micro-testing, and write tests that are a bit closer to functional tests. This too helps me to decouple the tests from the internals of the production code.

The bottom line is that when people give up on unit tests, it’s usually because they haven’t followed the above author’s advice. They have not treated the tests like first-class citizens. They have not treated the tests as though they were part of the system. They have not maintained those tests to the same standards that they apply to the rest of the system. Instead, they have allowed the tests to rot, to become coupled, to become rigid, and fragile, and slow. And then, in frustration, they give up on the tests.

Moral: Keep your tests clean. Treat them as first-class citizens of the system.