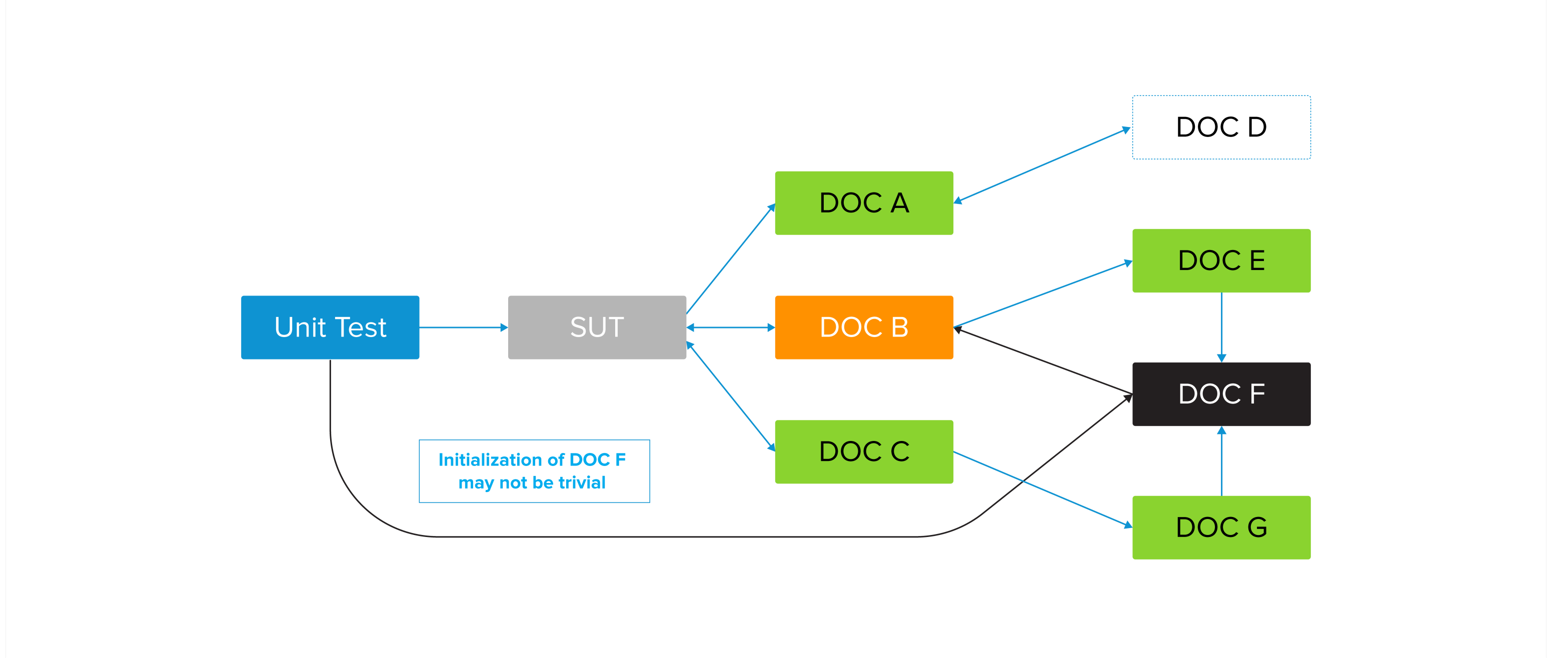
Unit Testing C/C++ Code: When to Mock

Posted on July 26, 2018 , in Unit Testing, C/C++ Testing



Written by

Miroslaw Zielinski

Product Manager for Parasoft's embedded testing solutions, Miroslaw's specialties include C/C++, RTOSes, static code analysis, unit testing, managing software quality for safety critical applications, and software compliance to safety standards.

Share this article:

Unit testing is all about testing the isolated units. In this post, we look at 7 times to mock, including some useful questions so you can guide yourself through C and C++ unit testing.

How much unit test isolation do we need? This is a recurring and important question commonly debated when developing unit tests for C and C++. And I'm not talking here about the isolation from the fellow developer, sitting next to us in the open space and drumming the rhythm of the music from his headphones (which, by the way, is also very important when we want to create good quality code). I'm talking about the isolation of the tested code from its surrounding environment – its so-called collaborators.

Before I continue, let me just clear one thing up: when discussing stubbing and mocking for C and C++ languages, usually there is a line drawn between C and C++ because of the differences in the language layer reflected in the complexity, capabilities, and expectations regarding typical mocking frameworks. With Parasoft C/C++test, the situation is slightly different because most of the framework capabilities are available for both languages. So, when discussing this subject I will be giving either a C or C++ unit test example, and unless I specifically mark something as supported only for C or C++, you should always assume that specific functionality is provided for both languages.

To isolate or not to isolate?

On the one hand, common sense dictates that we should not isolate unless we have a good reason for it. In the end, testing the real collaborators only increases our penetration of the code base. Why should we give up some extra code coverage and possibility of finding a bug? Well, it appears there are some good reasons for it – we will discuss it soon.

On the other hand, an orthodox unit tester will argue that unit testing is about testing the isolated units and it should stay what it is. Testing with real collaborators is the domain of integration phase. It is a well-known fact that by including the real collaborators into the test scope, we make our tests noisier. Tests relying on real collaborators will be reacting not only to the changes in the tested code, but also to changes in dependent components. Noisier tests make the maintenance process more expensive and generate a lot of distraction. Long term, this distraction usually becomes the main reason to abandon your unit testing practice.

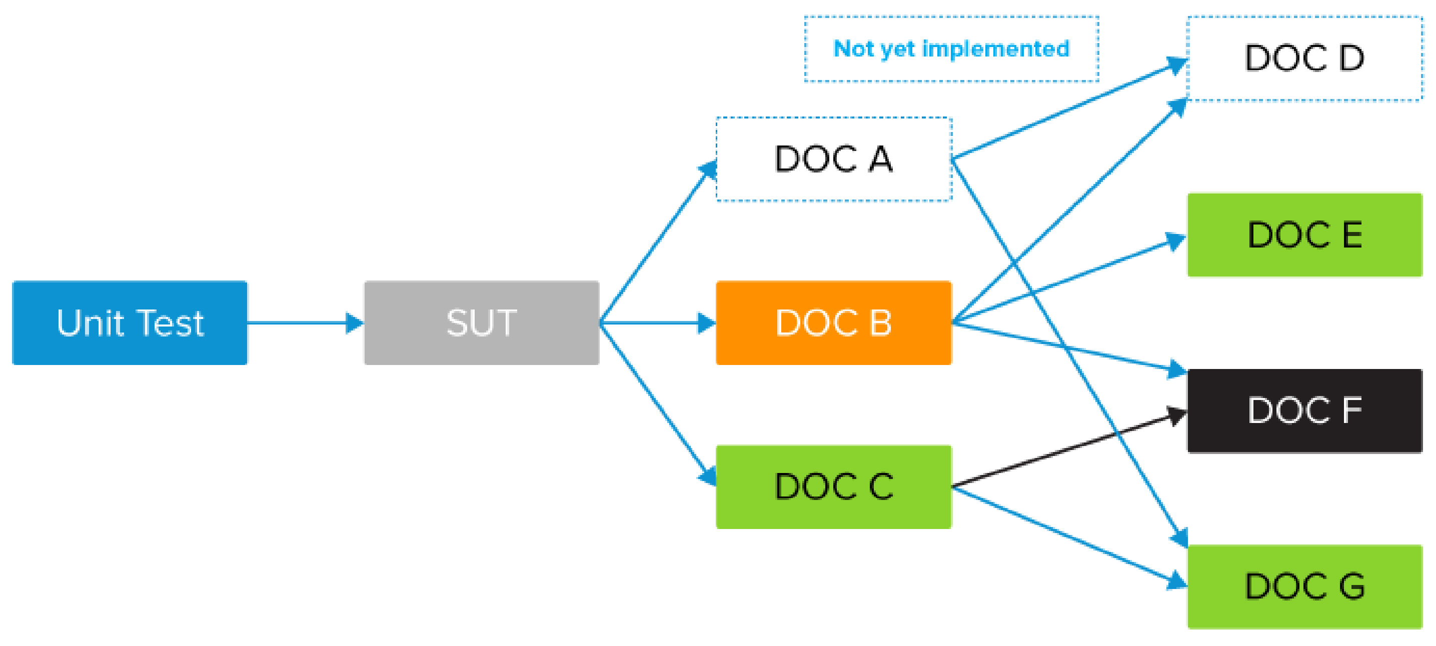
So what's the strategy for isolating the tested code? Given the above, it is difficult to formulate one good rule for determining which collaborators need to be mocked in order to provide suitable isolation of the tested code. From the perspective of testing efficiency and effectiveness, both "isolate as much as you can" and "avoid unit test isolation if possible" approaches have advantages and disadvantages.

3 Straightforward Reasons to Mock

Here are a few more-obvious situations:

1. Collaborator not yet implemented or still under development

This is a simple one. We do not have a choice, and we need a mock implementation. The diagram below illustrates this typical unit test environment (SUT – system under test, DOC – dependent component/collaborator):



Related Posts

Unit Testing

Using Stubs in Integration-Level Testing

Unit Testing

Test Smarter Not Harder: Shift Testing Left and Right with Test Impact Analysis

3 Practical Ways to Future-Proof Your IoT Devices

Unit Testing

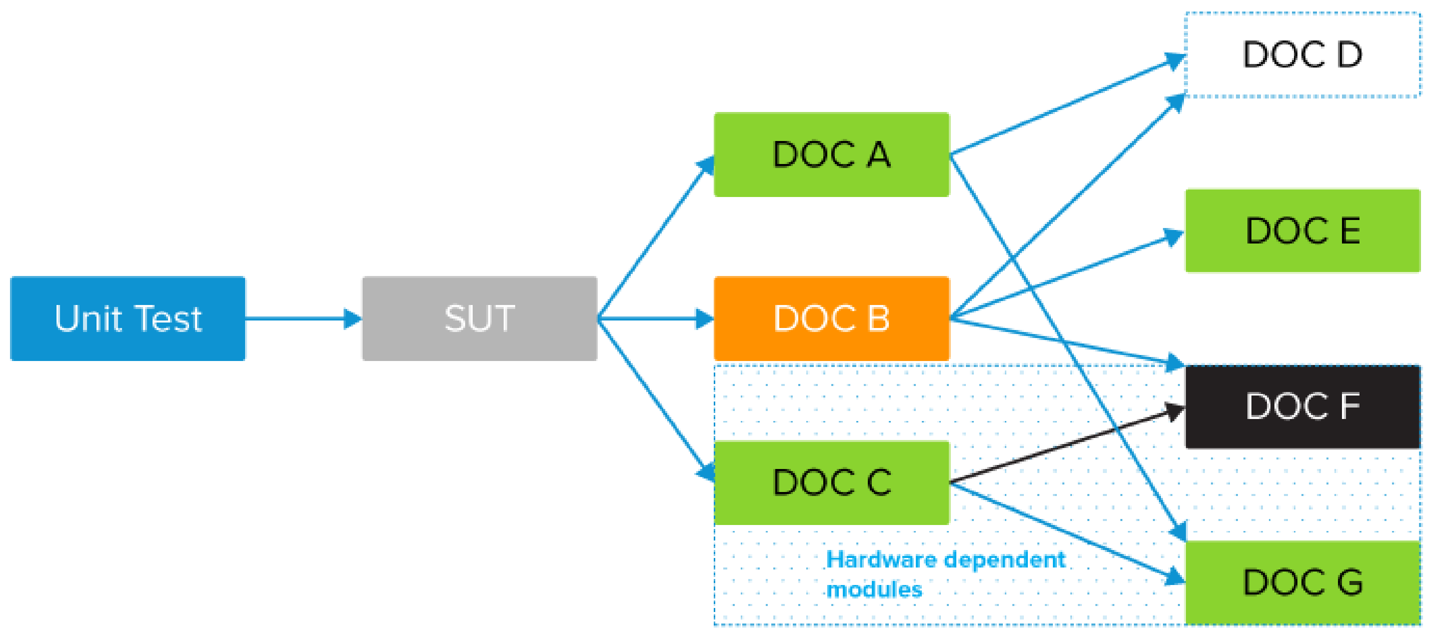
3 Practical Ways to Future-Proof Your IoT Devices

New releases of Parasoft Jtest & Parasoft dotTEST 10.4.2

Unit Testing

New releases of Parasoft Jtest & Parasoft dotTEST 10.4.2

developers, hardware independence of unit tests is an important aspect which allows high level of test automation and execution without a need of hardware. A good example here would be a unit under test interacting with GPS hardware, expecting a certain sequence of localization coordinates being provided to compute velocity. Although it's a good idea that we exercise more, I can't imagine testers running around with a device in order to simulate movement, just to generate the required test inputs, any time a unit testing session is required. To that end, this example illustrates just how late in the development lifecycle GPS testing of a device would be if hardware independence wasn't possible during development.



3. Fault injection

Injecting errors on purpose is a common scenario in testing. This might be used, for example, to test that memory allocation has failed, or to see if a hardware component has failed. Some developers try to stimulate the real collaborator in the test initialization phase, so that it responds with an error when called from the tested code. For me, this is not practical and is usually too much hassle. A test-specific, fake implementation, which simulates a fault, is usually much better choice.

4 Less-Straightforward Reasons to Mock

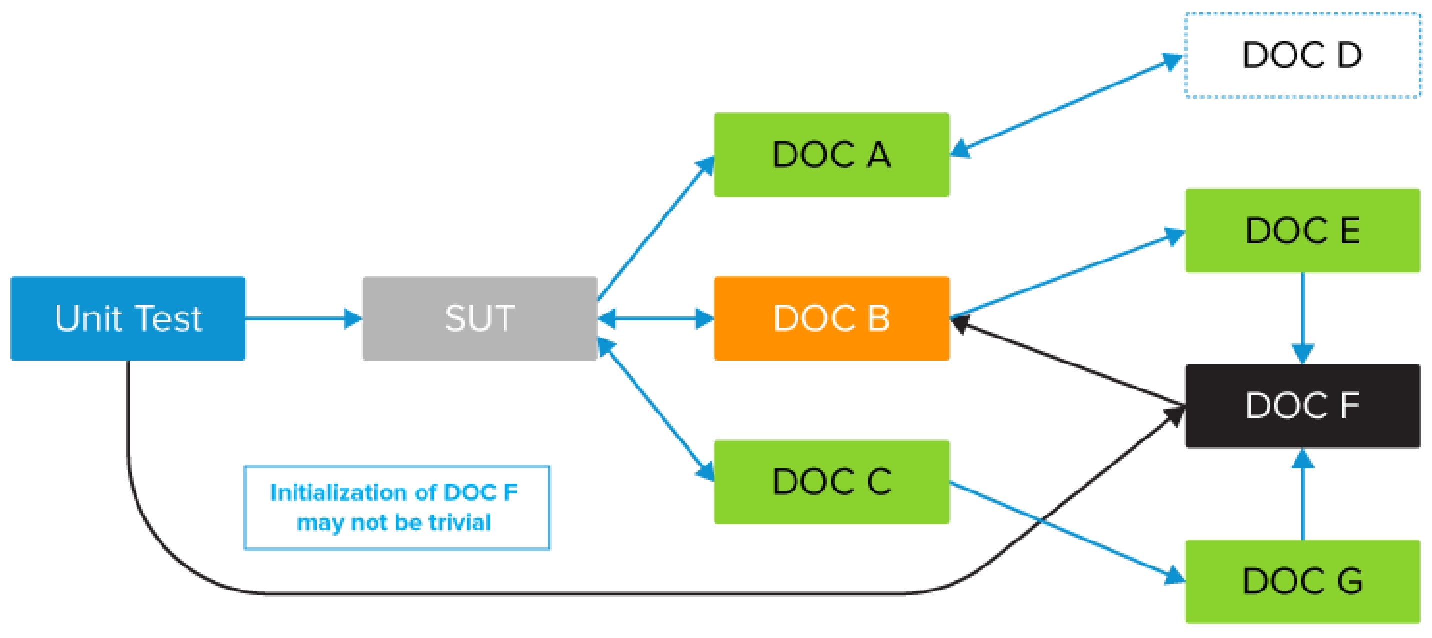
Besides these obvious cases where a mocked collaborator is always desired, there are some other, subtler situations when fake collaborators are a good choice. If our testing process suffers from any of the problems listed below, it is an indication that better isolation of the tested code is required.

1. Unit tests are not repeatable

It is difficult to implement stable tests that depend on a dependency that is volatile. What usually happens in such cases is we receive different test results without changing the tested code or tests' code. Transience may be an effect of relying on system calls or depending on an external signal which cannot be controlled from inside the test. A classic example is the system clock – if a test scenario requires reacting at certain points in time, then automation is difficult to achieve without mocked collaborators that have full control over indirect inputs to the tested code.

2. Test environments are difficult to initialize

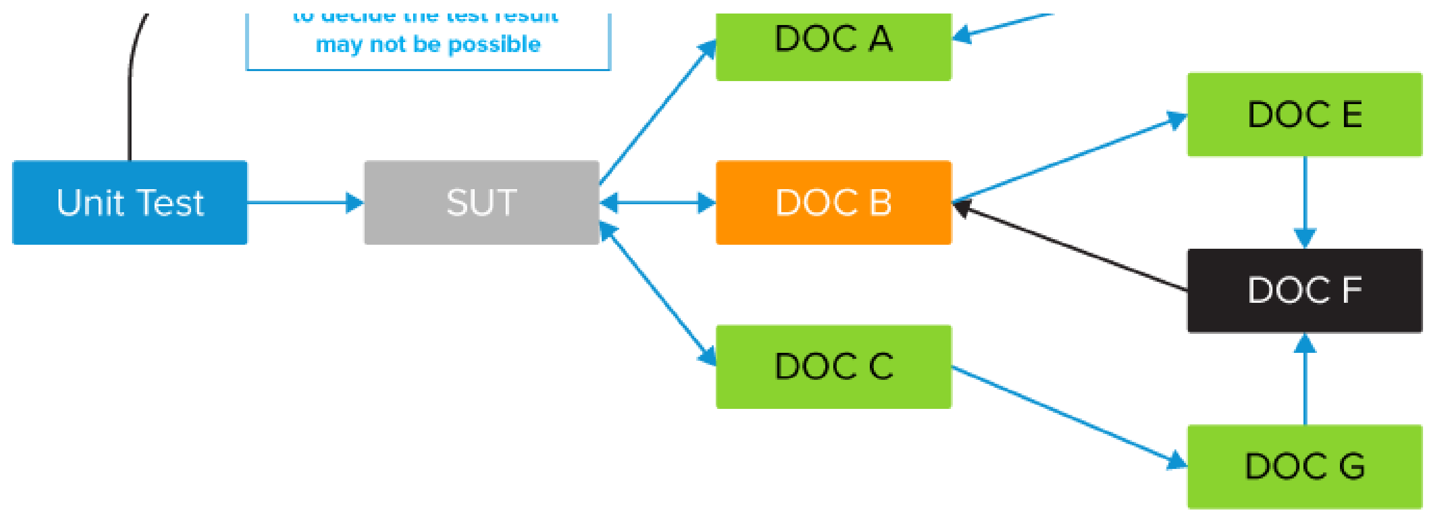
Initializing the test environment can be very complex. Simulating the real collaborators so that they provide reliable inputs to the tested code may be a daunting task, if not impossible.



Components are often interrelated, and when trying to initialize one specific module, we may end up initializing half of the system. Replacing the real collaborators with A fake implementation reduces the complexity of test environment initialization.

3. Test status is difficult to determine

In many cases, determining the test verdict requires checking the state of the collaborator after the test is executed. With real collaborators, it's often impossible because there is no suitable access method in the real collaborator interface to query the state after the test.



Replacing a real collaborator with a mock usually fixes the problem, and we can extend fake implementation with all kinds of access methods to determine the test result.

4. Tests are slow

There are cases when a response from the real collaborator can take a considerable amount of time. It is not always clear when the delay becomes unacceptable and when isolation is required. Is a 2-minute delay in a test run acceptable or not? It's often desirable to be able to run test suites as quickly as possible, perhaps after each code change. Large delays due to interactions with real collaborators can make test suites too slow to be practical. Mocks of these real collaborators can be faster by several orders of magnitude and bring the tests execution time to the acceptable level.

Useful questions to determine whether or not to mock

So, when writing a new C or C++ unit test and deciding about using original collaborators or mocked implementations, consider the following four questions:

1. Is the real collaborator a source of risk for the stability of my tests?
2. Is it difficult to initialize the real collaborator?
3. Is it possible to verify the state of the collaborator after the test, to decide the test status?
4. How long will it take for the collaborator to respond?

If we know the collaborator well enough to answer all these questions, then it's an easy decision one way or the other. If not, then I would suggest starting with the real collaborator and trying to answer these four questions as you go. In practice, this is the approach that most test-driven development (TDD) practitioners apply in their daily work. It means you need take due care of your test cases and review them carefully until they become stable.

Most often, unit test isolation is complicated by the dependencies of the unit under test. There are clearly desirable cases where mocking a dependent component is needed, but also more subtle situations as well. In some cases it isn't clear cut, and depends on the risk and uncertainty that a dependency has in the test environment.