Unit Testing and Mocking Explained

Unit Testing for Software Craftsmanship

Unit testing is becoming more and more popular in software development. However, the numerous frameworks, tools, and development processes can be confusing. This page is designed to help:

* Shed light on the different terms, practices and concepts related to unit testing and mocking
* Guide developers who are in the early stages of using unit tests

What is a unit of source code?

A unit of source code is the smallest part of a code that can be tested. In .NET languages, this is usually a method or a class.

What is unit testing?

Unit testing is the process through which units of source code are tested to verify if they work properly. Performing unit tests is a way to ensure that all functionalities of an application are working as they should. Unit tests inform the developer when a change in one unit interferes with the functionality of another. Modern unit testing frameworks are typically implemented using the same code used by the system under test. This enables a developer who is writing application code in C# to write their unit tests in C# as well.

*See the*[*Understanding Unit Testing video*](https://www.telerik.com/videos/justmock/understanding-unit-testing)*for additional information.*

What is a unit testing framework?

Unit testing frameworks are developed for the purpose of simplifying the process of unit-testing. Those frameworks enable the creation of Test Fixtures, which are .NET classes that have specific attributes enabling them to be picked up by a Test Runner.

Although it is possible to perform unit tests without such a framework, the process can be difficult, complicated and very manual.

There are a lot of unit testing frameworks available. Each of the frameworks has its own merits and selecting one depends on what features are needed and the level of expertise of the development team. Some examples of unit testing frameworks include:

* + CMSTest
  + xUnit
  + NUnit
  + MbUnit/Gallio
  + MSpec

What is mocking?

Mocking is a process used in unit testing when the unit being tested has external dependencies. The purpose of mocking is to focus on the code being tested and not on the behavior or state of external dependencies. In mocking, the dependencies are replaced by closely controlled replacements objects that simulate the behavior of the real ones. There are three main types of replacement objects - stubs, fakes and mocks.

**Fakes:** Fakes replace the actual code by implementing the same interface but without interacting with other objects. The problem Fakes introduce is they are hard coded to return fixed results and in order to test for different use cases a lot of Fakes must be introduced. Tests become hard to understand and maintain since the tests require a lot of Fakes as well as “else” statements to cover all possible statements.

**Stubs:** Stubs are similar to Fakes because they return prerecorded answers to calls. However, the difference is that by using a mocking framework you can create the Stub in the test with a minimal amount of code. Thus, it becomes clear how the dependency will respond and how the tested system should behave.

**Mocks:** Mocks have all the advantages of Stubs but also provide the option to specify behavior by setting an expectation on how many times a method must be executed. While Stubs are simple substitutes, Mocks are substitutes that can verify usage.

*For more information on the differences between stubs, fakes and mocks read the*[*Fakes, Stubs and Mocks*](http://blogs.telerik.com/justteam/posts/12-05-31/fakes-stubs-and-mocks.aspx)*blog post.*

*More practical examples on how to get started mocking can be found in the*[*Doing Your First Mock*](http://blogs.telerik.com/justteam/posts/12-05-14/doing-your-first-mock.aspx)*blog post*

What is a mocking framework?

Mocking frameworks are used to generate replacement objects like Stubs and Mocks. Mocking frameworks complement unit testing frameworks by isolating dependencies but are not substitutes for unit testing frameworks. By isolating the dependencies, they help the unit testing process and aid developers in writing more focused and concise unit tests. The tests also perform faster by truly isolating the system under test.

What is a test runner?

Where the unit testing framework adds facilities to create unit tests, the tests still need to be executed to verify the behavior of the system under test. Test Runners execute unit tests. Most of the unit testing frameworks include test runners and they vary from simple command line runners to graphical interfaces.

Some tools, such as [JustCode, a Visual Studio developer productivity tool](https://www.telerik.com/products/justcode.aspx) are able to run many types of unit tests.

*Additional information on the difference between testing frameworks, mocking tools and test runners can be found in the*[*Why Mocking Matters*](http://blogs.telerik.com/justteam/posts/12-10-10/why-mocking-matters.aspx)*blog post.*

Code First - Test Eventual Development

Code first is the most popular and widely used methodology. It implies first writing the code and afterwards writing a test that ensures the code’s functionality. Even developers that usually practice other methodologies sometimes use this methodology when they are writing simple, one-off applications.

Test First - Test-Driven Development (TDD)

The test first methodology implies writing the unit test before writing the actual code. This process forces the developer to think about the unit’s interface and expected results. Whether or not the code is written correctly is apparent immediately when using this methodology.

Test-driven development is based on the test first methodology but with the introduction of refactoring. TDD has the following prescribed set of steps:

* 1. Write an automated unit test which will initially fail for the new functionality being implemented. Use mock objects to stand in for dependencies.
  2. Write the minimum code needed to make the test pass. The code does not have to be perfect, it will be refined later.
  3. Refactor (or clean up) the code or add additional user cases to the test
  4. Ensure the tests still pass.
  5. Repeat

The purpose of refactoring and rerunning the tests is to help the developer deliver better code quality. If the refactoring step is skipped, there may be some repeated code, thus the [DRY (Don’t Repeat Yourself) principle](http://en.wikipedia.org/wiki/Don%27t_repeat_yourself) will be violated. If the tests are not continually re-run, a broken test can occur as a result of new code or previous refactoring.

Since TDD requires that each code unit is preceded by a test describing its functionality, using this methodology means following the YAGNI (You Aren’t Going to Need It) principle or ensuring that each functionality added is really needed.

Behavior-Driven Development (BDD)

Behavior driven development is based on TDD but it also addresses some of the issues that using TDD poses—where to start, what to test, how much to test in a cycle and what to call in the tests. BDD focuses on the behavioral requirements of the tested units.

BDD uses a ubiquitous language to facilitate communication between developers and stakeholders.

The following are the ways BDD modifies the unit testing process:

* 1. Using automated testing for feedback and regression testing.
  2. Using ‘when’ to describe an action under test.
  3. Using ‘should’ to describe behavior expected from the code.
  4. Using ‘ensure’ to describe responsibilities outside the scope of the code tested.
  5. Using mock objects to stand in for dependencies.

*Learn more about TDD, BDD and the SOLID principles in the*[*Why SOLID Matters*](http://blogs.telerik.com/justteam/posts/12-09-25/why-solid-matters.aspx)*blog post.*

Suggested Unit Testing Framework

.NET developers can take advantage of the numerous free unit testing frameworks available for download including MSTest (part of Visual Studio), NUnit, MbUnit and xUnit. The latter three are open source and have both a graphical and a command line test runners.

*For more on how to get started with those tools read our blog post*[*Getting Started with NUnit, NUnit.Should, and JustMock*](http://blogs.telerik.com/blogs/posts/12-10-05/getting-started-with-nunit-nunit-should-and-justmock.aspx)*.*

Suggested Mocking Framework

Mock objects can be created and maintained manually, but this process is very time consuming and ultimately unproductive.  
A complete mocking framework like [JustMock](https://www.telerik.com/products/mocking.aspx) enables developers to focus solely on testing the system under test and forget about the distracting mocking details. Mock objects are created automatically in memory when the tests are run based on the simple configuration in the unit test. There are no “physical” mock objects that have to be maintained as the project changes.

JustMock goes even further and does not force the developer to distinguish between “mocks” and “stubs.” With JustMock, there is a single, easy-to-use API for all of possible mocking needs. Behavioral specification is done though optional assertion calls, not through different construction techniques of the replacement objects.

JustMock has a free edition, [JustMock Lite](https://www.telerik.com/justmock/free-mocking/), as well, which provides all the functionality of JustMock except elevated mocking. JustMock Lite is suitable for mocking with loosely coupled code and projects designed for testability. However, if the development team is dealing with legacy or tightly coupled code, JustMock, the full edition, is the best option, since elevated mocking is designed for testing tightly coupled code that is hard to test otherwise.

Suggested Test Runner

The [JustCode Unit Test Runner](https://www.telerik.com/products/justcode/unit-test-runner.aspx) helps you run, debug, filter and organize your unit tests with ease. It supports the multiple.NET unit testing frameworks including MSTest, xUnit, NUnit 2.5, MbUnit 2.4, MbUnit3, and MSpec as well as two JavaScript testing frameworks QUnit and Jasmine.