# Mocking in Unit Tests

One of the key components of writing unit tests is to remove the dependencies your system has and replacing it with an implementation you control. The most common method people use as the replacement for the dependency is a mock, and mocking frameworks exist to help make this process easier.

Many frameworks and articles use different meanings for the differences between test doubles. A test double is a generic term for any "pretend" object used in place of a real one. This term, as well as others used in this page are the definitions provided by Martin Fowler. The most commonly used form of test double is Mocks, but there are many cases where Mocks perhaps are not the best choice and Fakes should be considered instead.

## Stubs

Stub allows you to have predetermined behavior that substitutes real behavior. The dependency (abstract class or interface) is implemented as a stub with a logic as expected by the client. Stubs can be useful when the clients of the stubs all expect the same set of responses, e.g. you use a third party service. The key concept here is that stubs should never fail a unit or integration test where a mock can. Stubs do not require any sort of framework to run, but are usually supported by mocking frameworks to quickly build the stubs. Stubs are commonly used in combination with a dependency injection frameworks or libraries, where the real object is replaced by a stub implementation.

Stubs can be useful especially during early development of a system, but since nearly every test requires its own stubs (to test the different states), this quickly becomes repetitive and involves a lot of boilerplate code. Rarely will you find a codebase that uses only stubs for mocking, they are usually paired with other test doubles.

Stubs do not require any sort of framework to run, but are usually supported by mocking frameworks to quickly build the stubs.

```
# Python test example, that creates an application
# with a dependency injection framework an overrides
# a service with a stub

class StubTestCase(TestBase):
    def setUp(self) -> None:
```

```
        super(StubTestCase, self).setUp()
        self.app.container.service_a.override(StubService())

    def test_service():
        service = self.app.container.service_a()
        self.assertTrue(isinstance(service, StubService))
```

### Upsides

- Do not require any framework, easy to set up.

### Downsides

- Can involve rewriting the same code many times, lots of boilerplate.

## 2 Mocks

Fowler describes **mocks** as pre-programmed objects with expectations which form a specification of the calls they are expected to receive. In other words, mocks are a replacement object for the dependency that has certain expectations that are placed on it; those expectations might be things like validating a sub-method has been called a certain number of times or that arguments are passed down in a certain way.

Mocking frameworks are abundant for every language, with some languages having mocks built into the unit test packages. They make writing unit tests easy and still encourage good unit testing practices.

The main difference between a mock and most of the other test doubles is that mocks do **behavioral verification**, whereas other test doubles do **state verification**. With behavioral verification, you end up testing that the implementation of the system under test is as you expect, whereas with state verification the implementation is not tested, rather the inputs and the outputs to the system are validated.

The major downside to behavioral verification is that it is tied to the implementation. One of the biggest advantages of writing unit tests is that when you make code changes you have confidence that if your unit tests continue to pass, that you are making a relatively safe change. If tests need to be updated every time because the behavior of the method has changed, then you lose that confidence because bugs could also be introduced into the test code. This also increases the development time and can be a source of frustration.

For example, let's assume you have a method that you are testing that makes 5 web service calls. With mocks, one of your tests could be to check that those 5 web service calls were made. Sometime later the API is updated and only a single web service call needs to be made. Once the system code is changed, the unit test will fail because it expects 5 calls and not 1. The test needs to be updated, which results in lowered confidence in the change, as well as potentially introduces more areas for bugs to sneak in.

Some would argue that in the example above, the unit test is not a good test anyway because it depends on the implementation, and that may be true; but one of the biggest problems with using mocks (and specifically mocking frameworks that allow these verifications), is that it encourages these types of tests to be written. By not using a mock framework that allows this, you never run the risk of writing tests that are validating the implementation.

### Upsides to Mocking

- Easy to write.
- Encourages testable design.

### Downsides to Mocking

- Behavioral testing can present problems with maintainability in unit test code.
- Usually requires a framework to be installed (or if no framework, lots of boilerplate code)

## Fakes

- Memory database for testing
- Not suitable for production
- Does take more time

**Fake** objects actually have working implementations, but usually take some shortcut which may make them not suitable for production. One of the common examples of using a Fake is an in-memory database - typically you want your database to be able to save data somewhere between application runs, but when writing unit tests if you have a fake implementation of your database APIs that are store all data in memory, you can use these for unit tests and not break abstraction as well as still keep your tests fast.

Writing a fake does take more time than other test doubles, because they are full implementations, and can have their own suite of unit tests. In this sense though, they increase confidence in your code even more because your test double has been thoroughly tested for bugs before you even use it as a downstream dependency.

Similarly to mocks, fakes also promote testable design, but unlike mocks they do not require any frameworks to write. Writing a fake is as easy as writing any other implementation class. Fakes

can be included in the test code only, but many times they end up being "promoted" to the product code, and in some cases can even start off in the product code since it is held to the same standard with full unit tests. Especially if writing a library or an API that other developers can use, providing a fake in the product code means those developers no longer need to write their own mock implementations, further increasing re-usability of code.

### Upsides to Fakes

- No framework needed, is just like any other implementation.
- Encourages testable design.
- Code can be "promoted" to product code, so it is not wasted effort.

### Downsides to Fakes

- Takes more time to implement.

## Best Practices

To keep your mocking efficient, consider these best practices to make your code testable, save time and make your test assertions more meaningful.

### 1 Dependency Injection

If you don't keep testability in mind from the beginning, once you start writing your tests, you might realize you have to do a time-intensive refactor to make the code unit testable. A common problem that can lead to non-testable code in certain languages such as C# is not using dependency injection. Consider using dependency injection so that a mock can easily be injected into your Subject Under Test (SUT) during a unit test.

More information on using dependency injection can be found here.

### 2 Assertions

When it comes to assertions in unit tests you want to make sure that you assert the right things, not necessarily lots of things. Some assertions can be inefficient and not give you the confidence you need in the test result. When you are mocking a client or configuration and your method passes the mock result directly as a return value without significant changes, consider not

asserting on the return value. Because if you do, you are mainly asserting whether you set up the mock correctly. For a very simple example, look at this class:

```
public class SearchController : ControllerBase {

   public ISearchClient SearchClient { get; }

   public SearchController(ISearchClient searchClient)
   {
      SearchClient = searchClient;
   }

   public String GetName(string id)
   {
      return this.SearchClient.GetName(id);
   }
}
```

When testing the `GetName` method, you can set up a mock search client to return a certain value. Then, it's easy to assert that the return value is, in fact, this value from the mock.

```
mockSearchClient.Setup(x => x.GetName(id))
   .ReturnsAsync("myResult");
var result = searchController.GetName(id);
Assert.Equal("myResult",result.Value);
```

But now, your method could look like this, and the test would still pass:

```
public String GetName(string id)
{
   return "myResult";
}
```

set Mock search with return value assertion in your - logic instead of return value

Similarly, if you set up your mock wrong, the test would fail even though the logic inside the method is sound. For efficient assertions that will give you confidence in your SUT, make assertions on your logic, not mock return values. The simple example above doesn't have a lot of logic, but you want to make sure that it calls the search client to retrieve the result. For this, you can use the verify method to make sure the search client was called using the right parameters even though you don't care about the result.

```
mockSearchClient.Verify(mock => mock.GetName(id), Times.Once());
```

This example is kept simple to visualize the principle of making meaningful assertions. In a real world application, your SUT will probably have more logic inside. Pieces of glue code that have as little logic as this example don't always have to be unit tested and might instead be covered by

integration tests. If there is more logic and a unit test with mocking is required, you should apply this principle by verifying mock calls and making assertions on the part of the mock result that was modified by your SUT.

## Callbacks

It can be time-consuming to set up mocks if you want to make sure they are being called with the right parameters, especially if the parameters are complex. To make your testing more efficient, consider using callbacks to make assertions on the parameters after a method was called. Often you don't care about all the parameters but only a few, or even only parts of them if the parameters are also objects. It's easy to make a small mistake in the creation of the parameter, like missing an attribute that the actual method sets, and then your mock won't be called, even though you might not care about this attribute at all. To avoid this, you can define only the most relevant parameters to differentiate between method calls and use an `any`-statement for the others. In this example, the method has a complex search options parameter which would take a lot of time to set up manually. Since you only care about 2 attributes in the search options, you use an `any`-statement and store the options in a callback for later assertions.

```
var actualOptions = new SearchOptions();

mockSearchClient
   .Setup(x =>
      x.Search(
         "[This parameter is most relevant]",
         It.IsAny<SearchOptions>()
      )
   )
   .Returns(mockResults)
   .Callback<string, SearchOptions>((query, searchOptions) =>
     {
       actualOptions = searchOptions;
     }
   );
```

Since you want to test your method logic, you should care only about the parts of the parameter which are influenced by your SUT, in this example, let's say the search mode and the search query type. So, with the variable you stored in the callback, you can make assertions on only these two attributes.

```
Assert.Equal(SearchMode.All, actualOptions.SearchMode);
Assert.Equal(SearchQueryType.Full, actualOptions.QueryType);
```

This makes the test more explicit since it shows which parts of the logic you care about. It's also more efficient since you don't have to spend a lot of time setting up the parameters for the mock.

## Conclusion

Using test doubles in unit tests is an essential part of having a healthy test suite. When looking at mocking frameworks and using test doubles, it is important to consider the future implications of integrating with a mocking framework from the start. Sometimes certain features of mocking frameworks seem essential, but usually that is a sign that the code itself is not abstracted enough if it requires a framework.

If possible, starting without a mocking framework and attempting to create fake implementations will lead to a more healthy code base, but when that is not possible the onus is on the technical leaders of the team to find cases where mocks may be overused, rely too much on implementation details, or end up not testing the right things.

---

Last update: August 26, 2024