Progress®

Unit Testing: The Complete Guide

Everything you need to know when starting a Blazor, ASP.NET Core, .NET 5, Xamarin or desktop project

EBOOK

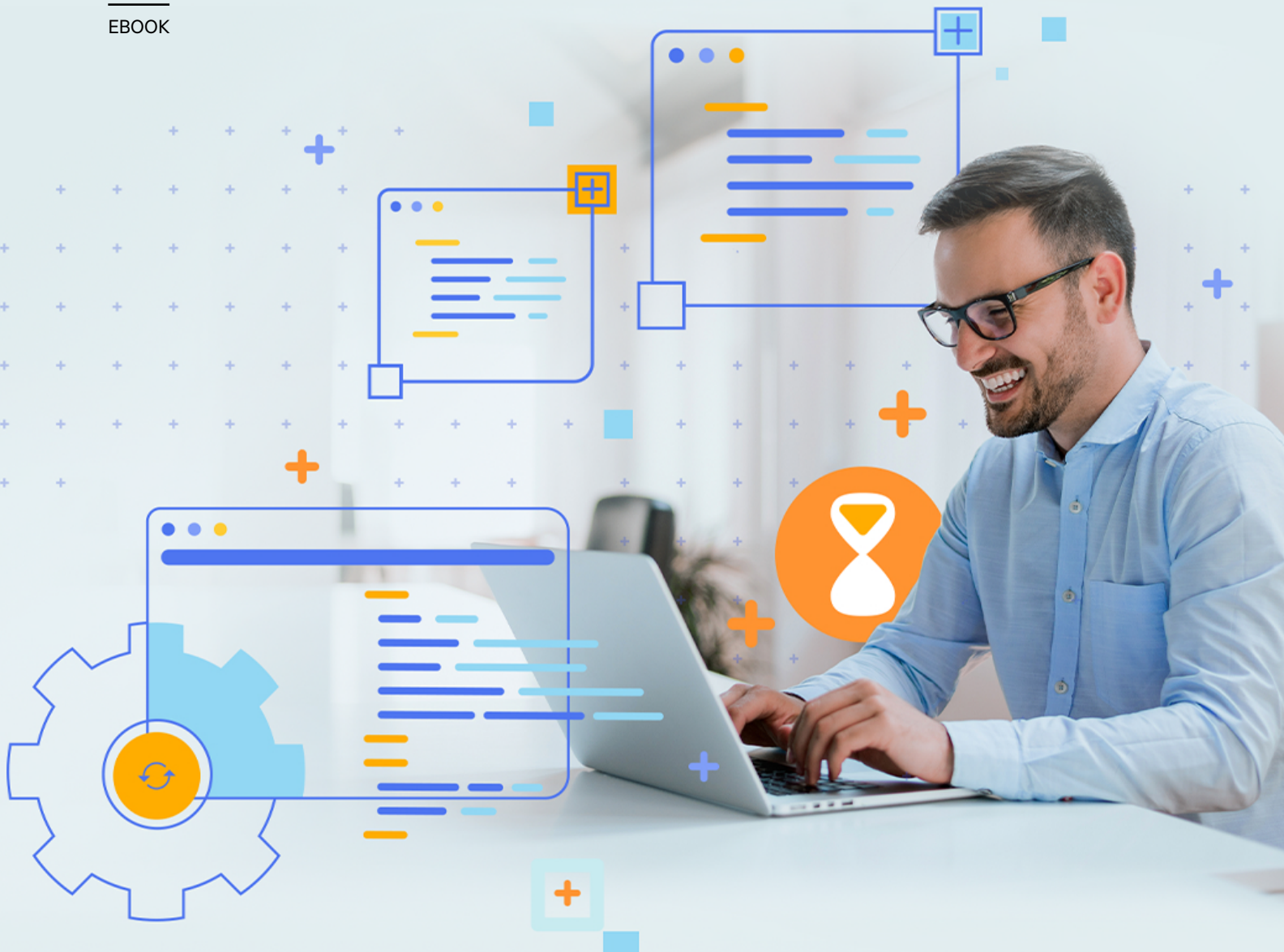


Table of Contents

Introduction / 3

Fundamentals of Unit Tests / 3

Advantages of Unit Testing / 4

Disadvantages and Limitations of Unit Testing / 7

Best Practices / 9

What is Mocking? / 15

Mocking Frameworks Explained / 17

Executing the Tests / 19

Introduction to Code Coverage / 19

When to Choose Your Tooling / 22

Conclusion / 23

Introduction

We all know how quickly we can give up on an application if we often encounter bugs in it as one of the most important criteria for choosing an application is quality. One of the essential methods for improving application quality is writing a full set of unit tests. Comprehensive testing is even more crucial if you are starting a new project on one of the latest technologies, such as Blazor, ASP.NET Core, or are looking into .NET 5. In this eBook, I've tried to cover everything you need to know about writing unit tests.

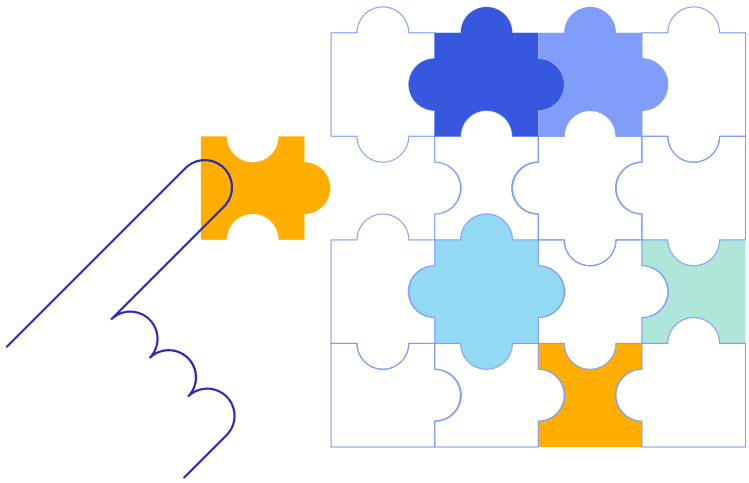
This eBook should be helpful to you whether you're exploring the fundamentals for the first time, refreshing your knowledge, or are in a role where you're guiding other developers or managing projects.

Let's dive right in.

Fundamentals of Unit Tests

What Is a Unit of Code?

A unit of code is the smallest piece of code that can be tested. In .NET languages, a unit typically represents a method. In C# 7.0, the smallest piece of code, strictly speaking, could be the local function. You could compare a unit to a single piece of a large jigsaw puzzle—you need to logically stack them all together to build a functional application.



What Is Unit Testing?

Unit Testing is a practice in software development for validating that a unit of code will behave as intended and the tested code will produce the same result every time the test is executed.

A unit test should only test the functionality of a single method, also called method under test or system under test (SUT). This is easier said than done due to different dependencies in the method and how they are controlled. Such dependencies include arguments, using the results of calling other methods, creation of new class instances and working with them, etc. The behavior of these dependencies should be controlled and they should produce control data.

I would say that anything that is not the actual logic of the method under test should be strictly controlled. This allows the method under test to be tested in complete isolation from its dependencies.

Unit testing provides numerous benefits, including finding bugs early, speeding up development, preventing regression bugs, understanding usage of code and more. Below, I will provide a detailed explanation of each advantage and disadvantage of unit testing.

Advantages of Unit Testing

Unit testing is essential to fast and agile development. It can help teams experience multiple short and long-term gains, the biggest ones being improved overall software quality and customer satisfaction. Here is a list of key advantages of unit testing.

Speed Up Development

The process of writing a complete set of unit tests makes you carefully consider the input, output, error conditions and overall architecture of the code. As a result, we developers can catch bugs and flaws in the code very early in the development process.

Imagine for a moment the unit tests were not comprehensive or, even worse, they weren't created in the first place. In such scenarios, you would make an implementation, use your manual testing abilities to find problems, fix them and iterate. In the process, you are likely to introduce an old problem with a slightly different execution path.

It's easy to get stuck in an endless and unproductive cycle. Unit tests can save you a lot of time and effort as they do a remarkable job of catching regressions.

Prevent Regressions

Regression bugs are bugs that are introduced after a change is made to the software and did not exist prior to that change. Bug fixes and new features often cause regression bugs to break existing functionalities. A well-designed and comprehensive unit test suite will prevent you or anyone on your team from breaking functionality that previously worked.

Test in Isolation

One of the biggest advantages of unit testing is that it allows you to test the method logic in isolation from its dependencies. In this way, when a bug is introduced, only a specific unit test will fail and indicate that there is an issue and where that issue is located. You will be able to immediately understand what the underlying problem is and fix it.

Isolation also allows all unit tests to be executed in random order. This is important because tests are a dynamic system—you add new tests, change or delete existing ones and keeping test cases independent will eliminate dependencies between tests. If done incorrectly, some of the tests may fail during some runs and pass in others. Debugging these types of failures is very hard and time consuming.

The main takeaway here is that the order in which the unit tests are executed should not affect the outcome of the unit test.

Code Modification, Refactoring and Maintenance

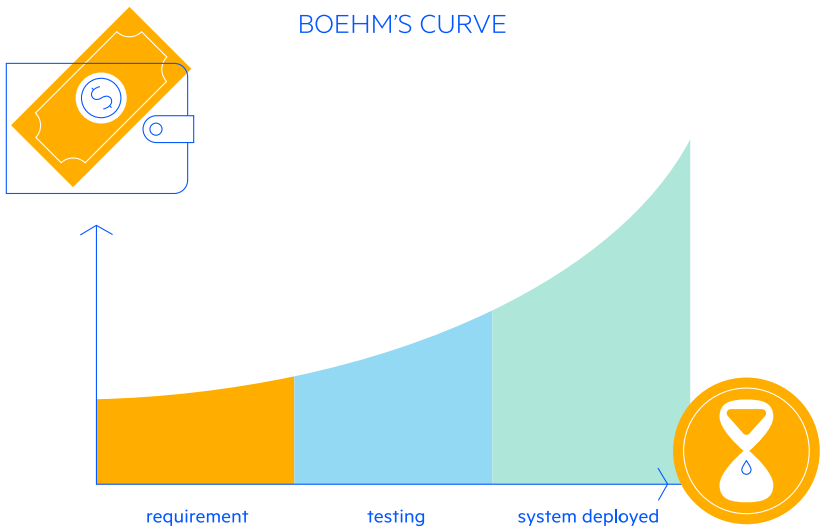
Once you have covered the bulk of your software by unit tests, you can rest assured you have covered all the bases for delivering quality applications on time. This will allow you to confidently make any changes to the software, knowing it will still work as expected, and keep bugs and regressions out of releases.

Reduce the Cost of Fixing Bugs

The cost of finding and fixing a bug during the initial stages of software development is far lower than doing it once the software is in production.

This is because testing is context dependent. When you are engaged in the process of implementing new functionality, your mind is quicker and better at catching and debugging a problem than doing it days or weeks later when it needs to rebuild the context from scratch. You can save yourself days of work and frustration fixing a piece of code you've grown unfamiliar with by fixing bugs as soon as you find them, which could take just a few hours.

Bugs that go unresolved late in the process can have significantly negative implications for your business, such as customer dissatisfaction, damaged reputation, payment of contractual penalties or even a lawsuit.



Safely Upgrade Third-Party Libraries

Quite often, software depends on third-party libraries. We know all too well they come with their own set of advantages and vulnerabilities. Having a complete set of unit tests allows you to upgrade any third-party library without worrying that they will break due to a defect outside of your control.

Updating third-party technology is particularly important when new features, significant improvements and bug fixes are released that will benefit your project. The more often you do it, the easier it will be to manage the updates.

Resolve Customer Complaints

It's never pleasing when a customer discovers a bug in your software. Nevertheless, it happens. What we as software developers can do is to identify the part of the code which is causing the issue and write unit tests in accordance with the customer scenario. This will allow you to weed out the bug and make sure that it doesn't bother your customers again.

Understand Usage of Code

As a developer, I often explore code that I see for the first time. It can be challenging to read and tackle code you didn't write—I am not sure what is the most convenient way to call a method or if I should use already existing helpers to construct the proper arguments for a given class constructor. In those cases, I check how the unit tests are written—they always give me valuable information.

Execute Tests Faster

Unit tests are lightweight and faster to execute compared to other types of tests like integration tests or UI test.

Automate Your Unit Tests

Today, unit testing is a key component of the automated continuous integration process. Continuous integration provides a consistent approach towards quality software and ensures any adjustments to the code are tested, no matter what. As a result, the software is more reliable and easier to maintain, and customers are satisfied.

Disadvantages and Limitations of Unit Testing

While the advantages of unit testing are manifold, there are some drawbacks. I will explore a few notable disadvantages and limitations of unit testing related to both its nature and implementation.

Skipping Execution

You may have the greatest set of unit test in the world, but they will be useless if not executed regularly throughout the development process.

It's Time-Consuming

Writing a thorough set of unit test requires a lot of time. Take for example a simple if-statement with one line of code that contains a Boolean condition. You must write at least two tests to cover the possible outcomes—one for true and one for false. What about three lines of code containing three different Boolean conditions? Then the number of required unit tests grows rapidly to meet the different combinations and scenarios.

Integration Errors

Unit tests are great when it comes to testing small chunks of software. However, since they do not interact with any peripheral units, they are not able to catch integration errors.

Hard to Isolate

Not every method can be easily isolated from its dependencies. Hard-to-isolate dependencies include external resources such as .NET Framework, .NET Core, third-party libraries, databases, services and more.

Other dependencies that are problematic involve the creation of new instances, access of internal or private APIs, access to static APIs, usage of events, delegates, etc. The good news is that dependencies can be solved by using a mocking framework, which I will explain later.

Software Changes May Require Unit Test Changes

When a significant refactoring is made, changes to the unit tests could be expected as the tested API is changed.

There are some well-known best practices I follow, which could help you reduce the effect of those disadvantages. Let's look at them.

Best Practices

Best practices are a set of guidelines that represent the most efficient or prudent course of action for a given topic.

Test Naming Convention

Why naming conventions matter? Following a naming convention provides consistency, structure and guidance that will make your and your team's work so much easier. For example, it is far easier to understand what you have broken by just looking at the name of the failed unit test instead of debugging it.

Pick a naming conversion for your tests that is easy to understand, easy to read and stick with it. One of the popular conventions to name your unit tests is to map the method under test, the state under test and the expected behavior. Like this:

MethodName_StateUnderTest_ExpectedBehavior

Test Naming

The test names should be descriptive and explicit

Test names should express a specific requirement

I like to use:

`Method_Scenario_Expected`

a.k.a

`UnitOfWork_StateUnderTest_ExpectedBehavior`

```
public void Sum_NegativeNumberAsFirstParam_ExceptionThrown()
public void Sum_SimpleValues_Calculated()
public void Parse_SingleToken_ReturnsEqualTokenValue()
```

Sometimes the name can become too long and incomprehensible. In such cases, it is useful to write self-explanatory tests.

Self-Explanatory Tests

Writing short, self-explanatory tests is a very important practice. It is as important as writing self-documented code. If you look at a test and you can't understand what it is designed to do and why it has been created, then this test has little value. No developer should be forced to debug a unit test in order to understand what the test is testing.

Self-Documenting Code Examples

```
public static List<int> FindPrimes(int start, int end)
{
    List<int> primesList = new List<int>();
    for (int num = start; num <= end; num++)
    {
        bool isPrime = IsPrime(num);
        if (isPrime)
        {
            primesList.Add(num);
        }
    }

    return primesList;
}
```

A good practice to write self-explanatory test is to use the Arrange-Act-Asser (AAA) pattern.

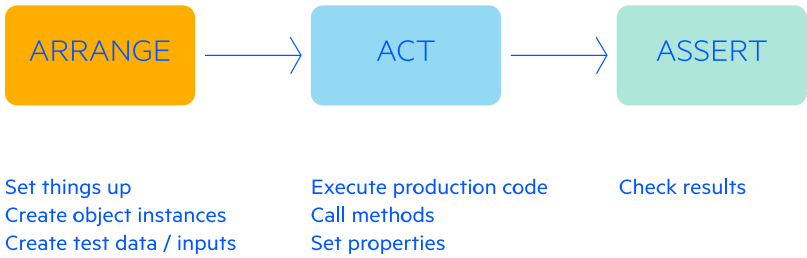
Arrange-Act-Assert Pattern

The AAA pattern makes your test look simpler and more structured by dividing it in three subsections, as explained below:

- **Arrange:** This is the section of the test where you will setup all the required test objects and prepare the prerequisites for your test.
- **Act:** This is the section where the actual work will be performed.
- **Assert:** This is the section where all verifications of expectations and results will be performed.

The division into subsections is done by adding a comment line containing the name of the subsection.

When you see such a test, you immediately understand what the purpose of each part of the test is. Still, the different parts could contain code that is difficult to understand, especially the Arrange section.



Easy to Setup Test Objects

Sometimes, preparing the dependencies required for the test could become cluttered and hard to understand. Try to limit the cluttering of test objects in the Arrange section. This will improve the readability of the test and help you understand faster what the test is doing. As a result, you will be able to resolve broken tests faster.

Following the above practices will improve the readability of your tests but will not prevent them from failing if they interfere with each other.

Avoid Test Interdependence

Unit tests should not interfere with each other in any way. Every test should be responsible for the setup and cleanup of its test objects.

A commonly suggested practice for avoiding test interdependencies is to write unit tests only for the public API.

Write Unit Tests Only for the Public API

Writing unit tests only for the public API is a popular recommendation I do not completely agree with. Let me explain why.

Imagine the following scenario: You work for a company that develops a document processor, like MS Word. The product can be delivered to the client as a library as well as a web and desktop application.

Your team is responsible only for the development of the style system. The style system decides if a specific letter should be rendered bold or italic, what font and font size to apply, etc.

You want all your APIs to be internal and not directly accessible by the end user. The end user will be granted access through a set of commands provided by another team.

This begs the question: "Should you write unit tests for the style system?" To me, the answer is straightforward—yes, you should write those unit tests.

And this is just one example. Oftentimes, exposing the entire API is just not feasible due to the nature and complexity of the project.

Test Driven Development (TDD)

Another beneficial practice I like is test-driven development.

Here is the Wikipedia definition of TDD:

> **"Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the code is improved so that the tests pass. This is opposed to software development that allows code to be added that is not proven to meet requirements."**
>
> **Wikipedia**

This process consists of five different phases:

First, Write the Tests

Write the tests from the perspective of a client who will work with the API. Add a few empty interfaces and classes and don't overthink the feature implementation. Focus on how to make the API easier to use.

This will force you to think about the feature details, what cases should be covered and how the dependencies would be organized.

Run All Tests and Verify They Fail

Execute the tests and verify that all tests are failing. Why is this important? Because in this way you will verify that the required behavior is not implemented and there are no flawed tests that will always pass no matter the code.

Write the New Code

Now the implementation can begin. Add the new code so that only a specific unit test is satisfied. Don't dive into full implementation. Just write the minimum amount of code required for the test to pass.

Run the Tests

After your chosen test passes, run all the tests. This will ensure that the new code meets the test requirements and does not break any other functionality.

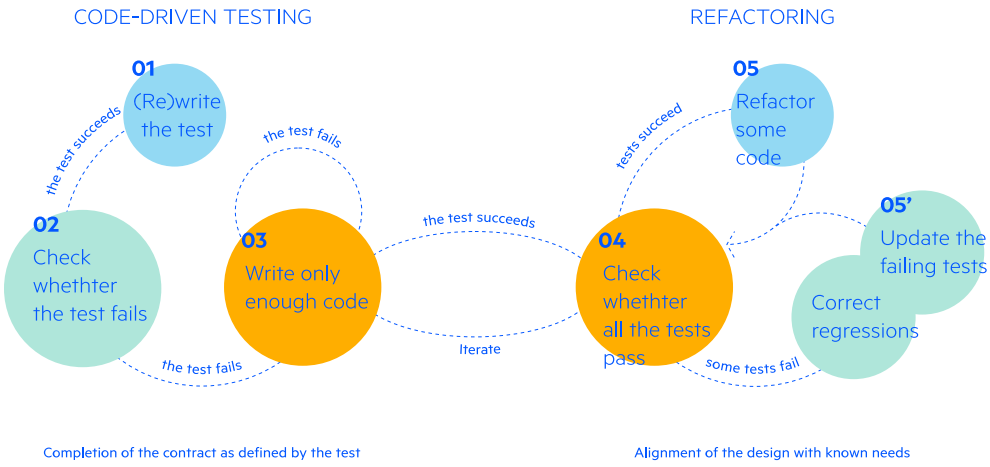
If this is not the case, fix the problems until all tests start passing.

Refactor the Code

In this phase, you can make significant or minor refactoring to meet code quality or architecture standards. The goal is to improve the code and fix all the shortcuts you have made.

Remember, the tests that passed should continue to do so after the refactoring. Look for strange patterns such as tests that previously failed and now start passing for no obvious reason. Investigate such cases.

A side effect of the refactoring can be a change in the way certain methods are used, which require you to rewrite some of the unit tests to reflect the changes made in the software.



Repeat the process until each test has passed and the quality of the code is acceptable.

Here are the main benefits of using TDD:

#1 Useful Public API

When you write your test, focus first on consuming that portion of the code which improves the quality of the public API.

#2 Easy to Read Code

Because refactoring is an essential step of TDD, your code will be much cleaner and easier to understand.

#3 Smoother Code Design

Another benefit of refactoring is the smoother code design.

#4 Easy to Understand and Maintain

When a code is easy to read and with good design, it is usually easier to maintain it too.

#5 Better Organized Dependencies

The dependencies of your classes and methods will contain only what is necessary. This typically leads to better naming, structure and organization.

Quite frequently, you will write a fake implementation of an interface dependency to isolate the tested method from that dependency. This is called mocking.

In the following chapter, I explain what mocking is and why it constitutes a best practice.

What is Mocking?

Mocking is a process employed in unit testing to simulate external dependencies. The purpose of mocking is to isolate and focus on the code being tested and not on the behavior or state of external dependencies. In mocking, dependencies are replaced by closely controlled replacement objects that mimic the behavior of the real ones.

To further explain this, I will give an example of how a car crash test is made. Instead of a real human, a dummy doll is used to test how the car crash will impact the human body.

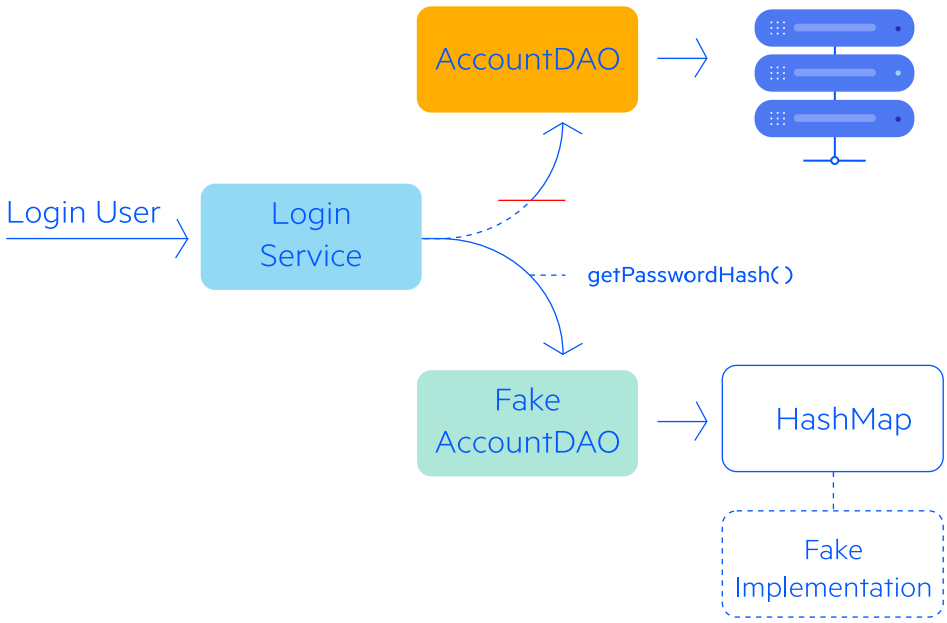
You will set expectations about how the car crash will impact the dummy doll and, after the crash, you will compare the data from the sensors inside the dummy doll to your expectations.

The dummy doll represents a mock of a human.

In unit testing, there are different types of mock objects:

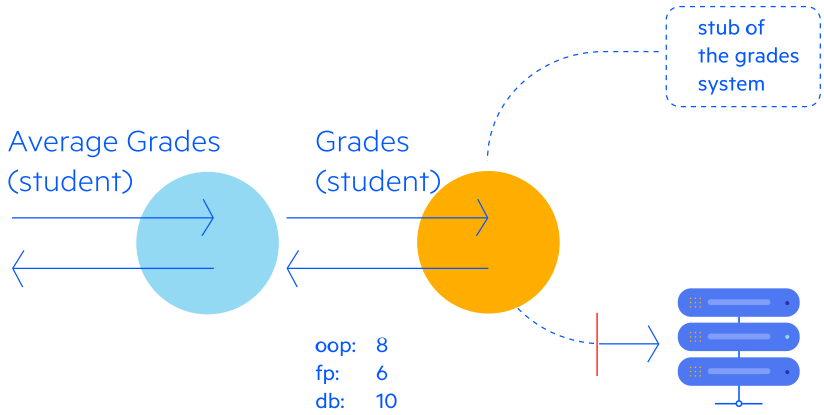
Fake Objects

A Fake is an object that will replace the actual code by implementing the same interface but without interacting with other objects. Usually, the Fake is hard coded to return fixed results. To test for different use cases, a lot of Fakes must be introduced. The problem with Fakes is that when an interface has been modified, all Fakes implementing this interface should be modified as well.



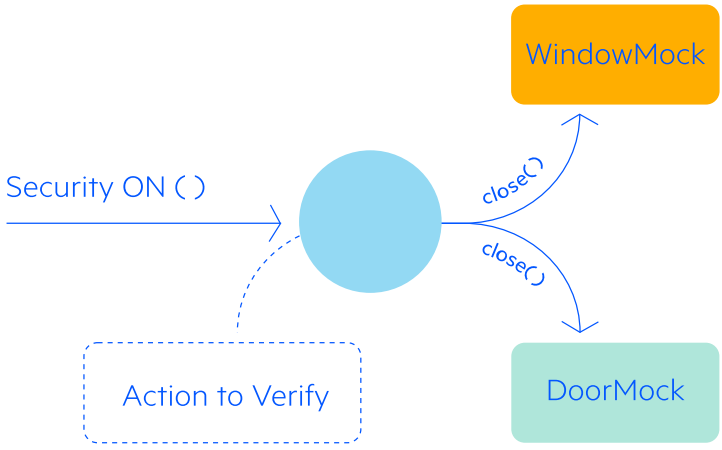
Stub Objects

A Stub is an object that will return a specific result based on a specific set of inputs and typically will not respond to anything outside of what is programed for the test.



Mock Objects

A Mock is a much more sophisticated version of a Stub. It will still return values like a Stub does, but it can also be programmed with expectations in terms of how many times each method should be called, in which order and with what data.



In the beginning, it can be hard to grasp the differences between the various types of mock objects. The good news is—you don't have to. A mocking framework will automate even the most complex implementation for creating a mock for you.

Mocking Frameworks Explained

Creating mock objects and managing them manually is time-consuming and frankly boring.

To boost your productivity and focus on the important things, I suggest you use a mocking framework.

A mocking framework like Progress® Telerik® JustMock will generate and manage the expectations of a replacement mock object for you. This allows you to write faster, concise, isolated tests. By isolated I mean your tests will be isolated from other tests and from the dependencies of the method under test.

I can hear you ask, "But how does a mocking framework make unit tests more concise?"

Well, because you will need less lines of code for writing the mock objects yourself. There won't be any "physical" mock objects. Frequently, only few lines of code per dependency will be required. This is the real benefit of using a mocking framework. You simply write less code and achieve better isolation.

More importantly, the mocking framework will create the correct type of mock object for you, so you don't have to worry about it.

Using a mocking tool will boost your productivity and let you focus on what is truly important like the overall architecture and usability, managing a comprehensive set of unit tests, etc.

In my opinion, a mocking framework is a must-have tool for writing unit tests.

Not All Mocking Frameworks Are Equal

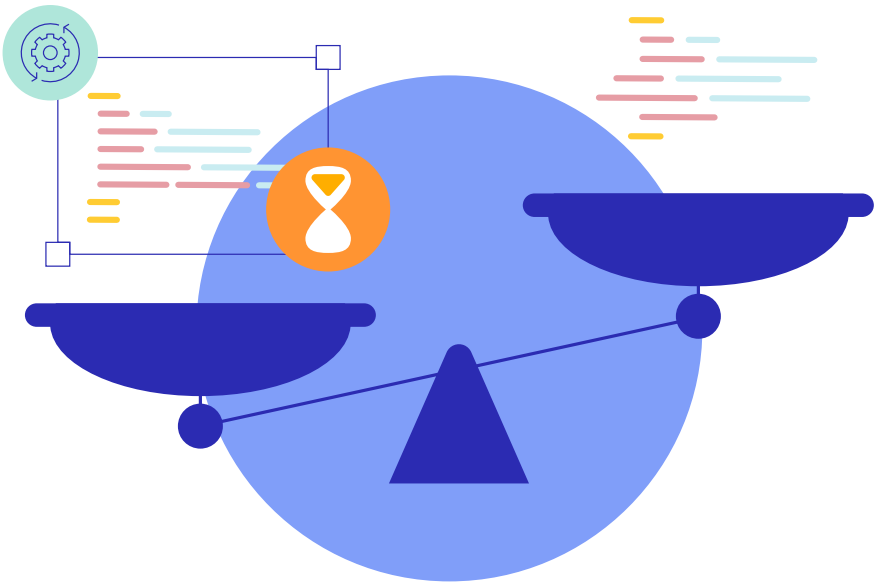
Most mocking frameworks have limitations as to what they can mock. The limitation comes from the way the mock object is created, which is through inheritance. The mock object can inherit only public APIs like virtual methods, abstract classes and interfaces. In addition, the APIs could provide all kinds of different expectations for the mock along with the validations to those expectations.

This functionality may be enough for your project as many companies have internal projects where all APIs can be public as they won't be consumed by a third party. However, if your development team is dealing with internal, private logic, static calls or a dependency on a third-party library, then Telerik JustMock is a better suited solution.

JustMock helps you advance the unit testing of C# devs. Here, the mocking is done through a CLR profiler and some technical wizardry. In essence, when the CLR tries to execute a method, the profiler will check if you have set a different behavior for that method and if so, your behavior will be executed.

This approach powers the rich capability of JustMock to mock just about anything—sealed classes, internal, private and static methods, LINQ queries, extension methods, third party libraries and the .NET itself.

Now that you have written your unit tests and used a mocking framework to isolate them, let's look at how to perform them.



Executing the Tests

The execution of tests is done by a Test Runner. The Test Runner communicates with the unit test framework to understand which classes contain unit tests in order to execute them. Most of the unit testing frameworks include test runners and they vary from simple command line runners to graphical interfaces.

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 complicated and very laborious. I hope nobody does it manually these days.

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 development team's level of expertise. Here are some of the most popular unit testing frameworks:

NUnit
xUnit
MS Test V2

Okay, so you have your unit tests and you execute them, but do you know if they are enough to ensure that the code quality is acceptable for you? You will probably need some measurement to understand this. This measurement is code coverage.

Introduction to Code Coverage

Code coverage is a metric that can help you understand what percentage of the source code is executed during a test run. Higher code coverage value means that more of your code is covered by unit tests and suggests that the defects would be less than a counterpart with lower code coverage.

It's a very useful metric that can help you assess the quality of your unit tests suite.

How is code coverage calculated?

Code coverage tools will use one or more coverage criteria to measure what percentage of your code is executed during a test run. Here are some of the most common criteria:

Function or method coverage: Measures how many of the methods have been called.

Statement coverage: Measures how many of the statements in the program have been executed.

Branch coverage: Measures how many of the branches have been executed. Branch is a decision-making code like if/or case statement.

To better explain it, consider a given if statement. Have both the true and the false branches been executed?

Condition coverage: Measures how many of the Boolean subexpressions have been tested for both true and false values.

Multi-condition coverage: Measures how many of the combinations in a conditional decision are tested. To illustrate this, consider the following if statement:

```
If(x||y)
```

For this example, the possible values are the following:

```
X=true, Y=true

X=false, Y=true

X=true, Y=false

X=false, Y=false
```

100% multi-condition coverage for this if statement means that all possible combinations are included in your unit tests.

Data coverage: This type of coverage is also known as parameter value coverage and measures how many of the common values of a parameter are used for testing a method with parameters.

Consider a method with a single string parameter. The common values for that string parameter would be null, empty string, whitespace, tab, newline, single-byte-string, double-byte-string.

Does 100% code coverage equal no bugs at all?

Sadly, no.

Most of the available code coverage tools do not provide the full set of coverage criteria. Especially the multi-condition and data coverage.

Another issue is that the code coverage does not provide information if all possible combinations of routes have been executed. This means that, if a unit test executes a method coming from one route, it will pass successfully. However, if it comes from another route it could potentially fail.

Does 100% code coverage provide real business value to clients?

My opinion is that it doesn't. Why?

Well, if you consider you have 95% code coverage, will the remaining 5% prevent major problems? If so, why?

Typically, when writing unit tests, you should strive to first cover the most common customer scenarios and then proceed with the rest.

Covering your code with 95% code coverage implies that all major and even side scenarios are covered. The other 5% are often scenarios so far away from the day-to-day use of the software that a small fraction of the customers may encounter them. And even if they run into those scenarios, the chance of their having a critical problem is relatively small. Not impossible, but small.

Bear in mind that in most cases 100% code coverage does not include multicondition and data coverage. The pursuit of that 100% could be considered overreaching.

Let me be clear—that doesn't mean that you shouldn't use code coverage. But when you do use it, both the pros and cons need to be considered.

When to Choose Your Tooling

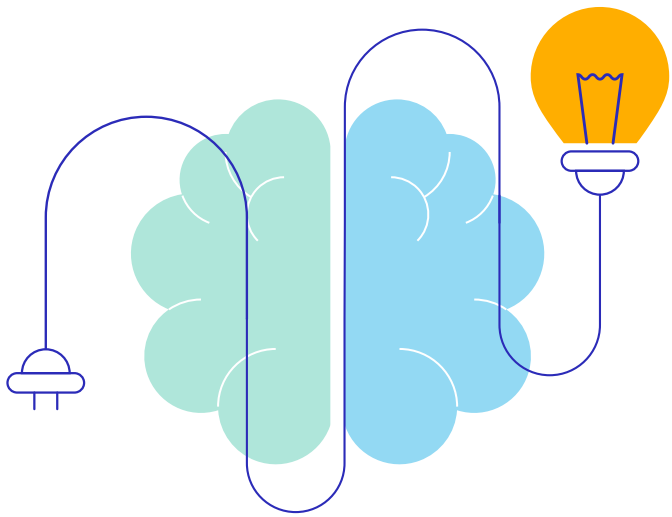
In my opinion, it is a good practice to evaluate and choose your tooling at the initial phase of your new project. This is the time when you will have more freedom and you will have more freedom and an easier time getting senior management's approval to change your tooling. However, this simple rule should not stop you from migrating to another tool if you are not satisfied with what you have, or if another more advanced tool emerges in a later phase of the project.

Conclusion

Here are the key insights I'd like to leave you with:

- Unit tests are a great practice for ensuring the quality of your code. Write more of them and deliver high quality software.
- A mocking framework is a must-have tool for isolating the code under test from any dependencies. So please stop wasting your time writing inefficient tests and find the mocking framework that will make your life easier.
- Code coverage is a useful metric for assessing the quality of your unit test suite. Use it wisely.

Happy testing!



Get Started Today

Take our mocking library for a spin – try out the latest version today with a FREE trial.

Try Telerik JustMock

Telerik JustMock is the fastest, most flexible and complete mocking solution for crafting unit tests. With JustMock you can easily isolate your testing scenario and focus on the logic you want to verify. It comes with an intuitive API with better discoverability that is easy to learn, use and allows for natural expression of mocking isolation concepts. The JustMock fluent interface facilitates fast feature discovery and provides options only valid to the current context resulting in higher productivity, better code, increased product quality and faster delivery.

About the author

Mihail Vladov is a Software Engineering Manager at Progress. He has more than a decade of experience with software and product development and is passionate about good software design and quality code. Mihail helped develop the WPF controls suite and Document Processing libraries, which are used by thousands of developers. Currently, he is leading the JustMock team. In his free time, he loves to travel and taste different foods. You can find Mihail on LinkedIn or Twitter.

To learn more visit:
www.telerik.com/products/mocking.aspx

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying strategic business applications. We enable customers and partners to deliver modern, high-impact digital experiences with a fraction of the effort, time and cost.Progress offers powerful tools for easily building adaptive user experiences across any type of device or touchpoint, leading data connectivity technology, web content management, business rules, secure file transfer and network monitoring. Over 1,700 independent software vendors, 100,000 enterprise customers, and two million developers rely o nProgress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.

Rev 2020/10 RITM0088515

Worldwide Headquarters

Progress, 14 Oak Park,
Bedford, MA 01730 USA
Tel: +1-800-477-6473

www.progress.com

facebook.com/progresssw
twitter.com/progresssw
youtube.com/progresssw
linkedin.com/company/progress-software

Progress