Ensure Long-term Code Quality and Prevent Future Issues with Unit Testing

by Ian Duckworth

Intro

“We can’t make the change you’re looking for; it’s too risky.” “If you change code there, who knows what the impact will be?” “The software is just too fragile to address any tech debt right now.” If you have heard these statements (or something similar), you have no doubt been frustrated by poorly-written software. Having poorly-written software doesn’t necessarily mean that the developers who wrote the software are bad or incompetent; it often simply means that over time, if care isn’t taken to perpetually ensure quality, standards tend to slip as new patterns become both available and viable.

This slippage can then have a cumulative effect in which production bugs spook managers and users alike as the software gets more and more locked down. What you eventually get is an unmaintainable product that nobody is happy with. This is obviously a bad situation to be in, but one that can be avoided.

There is no silver bullet to make software function perfectly, especially software that has gotten into such a poorly-performing state. However, there is hope. Significantly decreasing testing cycle times and adding in new testing cycles to give better and faster feedback to developers can greatly reduce the fragility of an application and decrease time to market for new features. A first step in accomplishing this goal is by implementing a unit test suite for the application.

What is Unit Testing?

Unit testing is the concept of writing code to test code. Every time a method is created, the implication is that at least one other method or dependency is relying on that method to do exactly what it was written to do. If that method is changed, how can one be sure that all of its dependencies still function as intended? Unit testing solves this conundrum by programmatically testing all the supported flows through the method.

Since the tests are programmatic, they run quickly (when compared to tests that hit a database and call out to external dependencies) and provide prompt feedback regarding whether or not a change broke a supported flow through a method. This creates a much less fragile application that can be refactored and maintained as business requirements change and better ways of doing things are discovered.

Since developers get near-immediate feedback on whether or not a change is causing breaks, the back and forth that can occur between developers and QA in the absence of unit tests is reduced. Additionally, unit tests are part of the code base of their applications, which means that developers can run the tests before checking code in, decreasing the burden on functional and end-to-end testing. Simply put, at a high level, having a highly-functioning unit test suite helps software gain and/or keep a competitive advantage.

Precursors for Unit Testing

Unfortunately, getting a code base into a state where it can be unit tested can take some preparation, especially if the code base is older and has been touched by numerous developers and/or teams. Here is what needs to happen to get a code base into a state where unit tests can be effectively written:

Dependency Injection

Dependency injection has two primary advantages. The first is that it creates a more functional code base because it forces all dependencies to be declared upfront since they will be injected directly into the constructor. This ensures that any developer looking at the class constructor can easily tell which other classes it relies on for the work it is going to perform. The second advantage is mocking (which will be discussed later in this document).

Composition Over Inheritance

This point directly aligns with dependency injection; it is the principle that inheritance should only be done in few select cases. For the vast majority of dependencies, each should be an independent class with an associated interface that is registered with a dependency injection provider which then injects into the constructor of a dependant service. This principle also has the effect of making applications more functional. See this blog post for a more in-depth explanation of this principle.

Single Responsibility Principle

Every method should do exactly one piece of work. What a “piece of work" means exactly is somewhat variable based on the business need of the application. However, if an application has a method called *CallServiceAndUpdateDatabase()*, it is likely that it is violating this principle since the implication is that the method calls a service and then updates a database all in one method.

This doesn't mean that a method can't have any decision logic, but be very wary of nested decision logic and excessive looping within a method. To correct the previous example, it would be best to have two methods called *CallService()* and *UpdateDatabase()* with a method that handles calls to both methods and manages the input and output.

Finally, going back to the previous points, each of these methods should probably be encapsulated in its own service and injected. See this blog post for more information on how the single responsibility principle should be applied.

Writing a Useful Unit Test

Many organizations strive for unit test related metrics such as 100% unit test coverage or a certain percentage of tests passing. These are admirable goals, but they are very misguided in practice and can produce more of a "checkbox" mentality rather than providing tangible value.

This is because most methods in a code base have more than one possible outcome, so 100% test coverage means that a large portion of the business logic is left untested (since this metric would indicate to a developer that there is a 1:1 mapping of unit tests to methods, which is frequently not the case).

In addition, depending on how a project is structured, there are some methods that might not need to be unit tested at all, such as methods that provide only a wrapper for third-party functionality, or methods that only access the database (which is discussed more in the next section).

Instead of any unit test related metrics, the focus should be that all business logic is tested. While that may sound simple, it can be somewhat complicated because many code bases have business logic spread out haphazardly. It is always a good idea to make sure that business logic is encapsulated in one or two layers of an API.

Modular Project Structure

Choosing a proper project structure has a huge impact on the quality of unit tests and the ease of which they can be written. While a project can be validly structured in many different ways, below is a tried and true pattern that supports unit testing:

Controller Layer

This is the first layer of the application and provides an entry point for all API endpoints. It controls any sort of exception handling and routing with regards to what should be returned. This layer should contain business logic, but only logic as it pertains to what should be returned based on the result of the processing.

For example, if there is an API method to get a user based on an email address and an email address is passed in that doesn't exist in the database, a 204 HTTP status code should be returned. Whatever the return value, this logic would be done at the controller level.

Helper Layer

This layer sits below the controller layer, and each helper is injected into one or more controllers. This layer contains all of the business logic of the application with the exception of anything routing related (which is done in the controller layer, as described above). This layer will be the primary focus of any unit testing effort.

Data Access Layer

This layer abstracts away any database calls. No business logic should be done in this layer; it is strictly a pass-through for the database. This implies that no unit tests need to be written for any methods in this layer (unless it includes stored procedure calls and one would like to ensure that the stored procedures are called with the proper parameters).

This structure is great for simple APIs. It can also work well for larger and more complicated APIs by tweaking the structure slightly. One such tweak is to add a pseudo layer that sits parallel to the helper layer—a layer that is sometimes called the service layer. This layer can contain business logic that is not directly related to a controller/helper/data layer directly, but is instead used by multiple helpers. It supports reusability and makes testing much easier since it can be unit tested independently.

Sample Unit Tests

Let’s look at a sample controller. This controller has a helper injected into it, and the helper has a method called *DoWork* on it which takes in a string and returns an instance of *TestModel*.

Note that all code samples here are written in C# and are using ASP.NET. However, the principles still apply to any object-oriented language used to write an API.

The following code samples are taken from my unit-test-example [project on GitHub](). The repository can be cloned locally and the code referenced below can be viewed, debugged, modified, etc.

```
    [RoutePrefix(“v1/example”)]
    public class TestController : ApiController
    {
        private readonly ITestHelper _testHelper;
        public TestController(ITestHelper testHelper)
        {
            _testHelper = testHelper;
        }

        public IHttpActionResult GetTest(string input)
        {
            try
            {
                TestModel result = _testHelper.DoWork(input);
                if (result == null)
                    return Content(HttpStatusCode.NoContent,
                        $”No content found for input: {input}”);

                return Ok(result);

            } catch (Exception ex)
            {
                return Content(HttpStatusCode.InternalServerError,
ex.Message);
            }
        }
    }
```

If we were to run this through in a production-like environment, the first thing that would happen is that the try/catch block is entered followed by TestHelper. DoWork. However, in a unit-testing setup, this is not what is desired. The desire would be to only test the controller method (TestController.GetTest). This can be accomplished through a mocking framework. A mocking framework will allow us to create a mocked version of a dependency and pass it in through the constructor when the controller is created.

```
    [TestFixture]
    public class Tests
    {
        private TestController _testController;
        private Mock<ITestHelper> _testHelperMock;

        [SetUp]
        public void SetUp()
        {
            _testHelperMock = new Mock<ITestHelper>();
            _testController = new TestController(_testHelperMock.
Object);
        }
    }
```

Note that we are using Moq here to create a mocked version of TestController's only dependency, ITestHelper (other mocking frameworks for .NET include, but are not limited to, NSubstitute and FakeItEasy). We are then passing the mocked object into the constructor. Through Moq, we can manipulate the TestHelper. DoWork method to do whatever we want, including what it returns and if it throws an exception.

Also note that we are using NUnit here as our testing framework. Decorating the SetUp method with the SetUp attribute ensures that the Setup method will be run before each unit test. This means that any mocks that were set up in a previous unit test will be destroyed. In other words, this pattern ensures that each test has a clean slate.

Going through our TestController.GetTest(string input) method, it becomes evident that there are three possible flows through that method, each of which is driven by what happens in the TestHelper.DoWork(string input) method:

1. If the DoWork method throws an exception, a 500 should be returned with the exception message text.

2. If the DoWork method returns null, a 204 (no content) should be returned.
3. If the DoWork method returns an instance of TestModel, a 200 should be returned and the object returned from DoWork will be returned by GetTest.

This means that to test our business logic in this particular controller method, we will actually need three unit test methods. First, let's test what happens if DoWork returns null. We can do this by using Moq to manipulate what the DoWork method returns. We will now force it to return null:

```
    [TestFixture]
    public class Tests
    {
        private TestController _testController;
        private Mock<ITestHelper> _testHelperMock;

        [SetUp]
        public void SetUp()
        {
            _testHelperMock = new Mock<ITestHelper>();
            _testController = new TestController(_testHelperMock.
Object);
        }

        [Test]
        public void TestGet_Null()
        {
            const string input = “input”;

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).
Returns(default(TestModel));

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<NegotiatedContentResult<string>>(result);
            var response = result as NegotiatedContentResult<string>;

            Assert.True(response.StatusCode == HttpStatusCode.
NoContent);
            Assert.True(response.Content == $”No content found for
input: {input}”);
        }
    }
```

Every mocking framework has a slightly different syntax, but the idea is always the same. In the “arrange” section of this test, we are telling the mocked object that whenever anyone calls DoWork and passes in “input” to return null. When TestHelper.DoWork is called (in the unit test) we will ensure that null is returned by DoWork, and thus a 204 should be returned by TestController.GetTest in our controller. If TestController.GetTest does not return a 204 with the proper message, the unit test will fail.

Next, let’s go over what should happen if TestHelper.DoWork throws an exception. In this case, according to our controller code, it should return a 500 and the exception text should be the content returned.

```
    [TestFixture]
    public class Tests
    {
        private TestController _testController;
        private Mock<ITestHelper> _testHelperMock;

        [SetUp]
        public void SetUp()
        {
            _testHelperMock = new Mock<ITestHelper>();
            _testController = new TestController(_testHelperMock.
Object);
        }

        [Test]
        public void TestGet_Null()
        {
            const string input = “input”;

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).
Returns(default(TestModel));

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<NegotiatedContentResult<string>>(result);
            var response = result as NegotiatedContentResult<string>;

            Assert.True(response.StatusCode == HttpStatusCode.
NoContent);
            Assert.True(response.Content == $”No content found for
input: {input}”);
        }
        [Test]
        public void TestGet_Exception()
        {
            const string input = “input”;
            const string exception = “exception”;

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).Throws(new 
Exception(exception));

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<NegotiatedContentResult<string>>(result);
            var response = result as NegotiatedContentResult<string>;

            Assert.True(response.StatusCode == HttpStatusCode.
InternalServerError);
            Assert.True(response.Content == exception);
        }
    }
```

The final thing that we need to do is check the happy path. What happens if TestHelper.DoWork returns a valid instance of UserModel? If this occurs, no exception is thrown and the instance of UserModel is not null, thus a 200 is returned with the instance of UserModel as the object’s content.

```
   [TestFixture]
    public class Tests
    {
        private TestController _testController;
        private Mock<ITestHelper> _testHelperMock;

        [SetUp]
        public void SetUp()
        {
            _testHelperMock = new Mock<ITestHelper>();
            _testController = new TestController(_testHelperMock.
Object);
        }

        [Test]
        public void TestGet_Null()
        {
            const string input = “input”;

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).
Returns(default(TestModel));

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<NegotiatedContentResult<string>>(result);
            var response = result as NegotiatedContentResult<string>;

            Assert.True(response.StatusCode == HttpStatusCode.
NoContent);
            Assert.True(response.Content == $"No content found for
input: {input}");
        }

        [Test]
        public void TestGet_Exception()
        {
            const string input = "input";
            const string exception = "exception";

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).Throws(new
Exception(exception));

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<NegotiatedContentResult<string>>(result);
            var response = result as NegotiatedContentResult<string>;

            Assert.True(response.StatusCode == HttpStatusCode.
InternalServerError);
            Assert.True(response.Content == exception);
        }

        [Test]
        public void TestGet_Ok()
        {
            const string input = "input";

            //arrange
            _testHelperMock.Setup(x => x.DoWork(input)).Returns(new
TestModel());

            //act
            var result = _testController.GetTest(input);

            //assert
            Assert.
IsInstanceOf<OkNegotiatedContentResult<TestModel>>(result);
        }
    }
```

Those three unit tests now cover every flow through the TestController.GetTest method, thus all business logic contained in that controller is tested and all dependencies are mocked out. This ensures that the unit tests will never fail because of an issue outside of the scope of the method under test. However, the unit tests will fail if someone, for example, modifies the controller to start returning a 404 instead of a 204 when TestHelper.DoWork returns null. This is good because an external dependency is probably depending on the 204 being returned, so that change should cause the unit test to fail.

Even better is the fact that immediate feedback is given to the developer that the change is causing an error. This allows the developer to fix the problem immediately without having to go through any functional or end-to-end testing cycles (or worse, fix a production bug). This is why setting your code base up to follow both the composition over inheritance and the single responsibility principles is so important. If the responsibilities get muddled, the ability to test a single “unit” of work within the code base becomes difficult, if not outright impossible.

Challenges

The main challenge in creating a highly functioning unit test suite is not the writing of the tests themselves, but getting your code base into a state where it can be unit tested. This can be especially challenging since the applications with the largest need for unit tests are oftentimes older applications where the principles and patterns mentioned above have not been followed properly (if at all). Selling the idea of a large refactor to an organization can be a daunting task.

However, it is worth it in the long run because while a refactor of this nature will certainly cause some pain in the short term, it will allow for changes to the application to be made significantly more quickly and in a less error-prone fashion in the long term.

A real enterprise application that Levvel developed for a client had 190 fully-mocked unit tests. This provided full coverage for all business logic in the API (including controllers and helpers as well as extension methods and other miscellaneous functionality), and they took between 2.5 and 3 seconds to run. Because the unit test suite was so fast and so inclusive, it meant that developers could get features done faster with little to no fear that their changes were breaking other business logic (as long as all unit tests were in a passing state). This had the effect of significantly reducing QA cycle times since bugs became a rarity. The end state for this particular application was that code got out to production faster than for other applications in the organization and production defects were very uncommon.

This effectively-instantaneous feedback regarding if a change broke any business logic in any other part of an application is indispensable to any organization regardless of size or type of business. I have personally never seen a client displeased with the long-term results when a proper unit test suite is implemented.

In order to sell the idea of a unit test centric refactor, it is necessary to show the organization the many benefits that will be seen in the medium to long term while highlighting the comparatively short amount of time it takes to set them up. The unit testing itself can present some challenges of its own, particularly regarding how exactly to test one unit of code. This document previously mentioned the use of a mocking framework; using a mocking framework is absolutely essential to having a highly-functioning unit test suite. At no time should the unit tests ever access a database, call an external dependency, or do anything outside the scope of the method itself. To have a unit test do so violates the entire idea of unit testing (while simultaneously making your tests flaky and/or long-running in the case that an external dependency which you have no control over is not functioning properly).

This is not to say that testing external dependencies is bad or unnecessary (in fact, to ensure a successful release, these things absolutely have to be tested), but instead that the unit test suite is the wrong venue to test such things.

Testing dependencies is part of integration testing and testing the data access layer is part of functional testing (as well as integration testing). If anyone says that unit testing is a bad practice because the tests take too long to run or are flaky, it is almost certain that the person saying these things has never been exposed to properly created unit tests. If someone claims that unit testing is flawed because it doesn't cover integration or functional testing, that person almost certainly is misunderstanding what unit testing is for.

Unit testing is best used for immediate feedback for a developer so the developer can see what effect their code change had on the application. This should greatly ease the burden on a QA team. However, a good unit test suite is never a replacement for functional and integration testing.

Getting Buy-in from Management

This idealized state certainly sounds nice, and while it is absolutely attainable, it may take a marked culture shift at an organization to truly make it a reality. It demands that all developers follow a series of patterns that they may not know or understand at the current time; this requires training and explanation. Additionally, it may require refactors of fragile code bases that are left relatively exposed. This all creates a difficult sell to management since many people who are involved in the ultimate decision will only see the upfront cost without much consideration to the cost savings and revenue generation (via getting business features out the door much faster and much more accurately). This is where selling the idea to management becomes very important.

One well-used and tested technique is to bide one's time until the organization decides they want to create a new code base. That code base can then become a pilot for this new way of thinking. A set of strong developers who can implement the appropriate principles and write the appropriate unit tests as the application is being developed should be selected. As time passes, the immediate team should notice that, as the business requirements are inevitably tweaked and modified, the team is very agile in their ability to respond without introducing unintended consequences.

This, hopefully, will build up some goodwill with management which can be parlayed into a larger conversation about the state of the other applications which are not so agile. The pilot application can then be used as a baseline for gathering real and hard data about performance, both regarding how easy it is to test and in how good the team's throughput is (this data can be aggregated through your tracking software; JIRA, Rally, etc.). Finally, this hard data can be used to show that:

1. The idea of a highly-functional unit test suite is not just an unattainable ideal, but one that, with the right people and the right guidance, can easily become

a reality.

2. The long-term benefit of a refactor and a unit-test writing effort far outweighs the initial investment.
3. The right people already exist within the organization who can spearhead such an effort.

Conclusion

While the idea of unit testing has been around for years, it is always surprising how few organizations effectively leverage the power of unit tests. If the desire is to influence an organization to create a culture of unit testing, make sure to learn and implement the principles discussed in this document first. Develop a rapport with management before such an idea is pitched, and remember that someone in the business is generally going to want to hear things in terms of dollars and cents. Anyone's best bet for influencing a culture change of this nature is to get the business on board and to sell it as a beneficial and necessary long-term investment. The ultimate payoff is both lowered cost of the testing effort and significantly improved time to market for business ideas and requirements.

About Levvel

Levvel helps clients transform their business with strategic consulting and technical execution services. We work with your IT organization, product groups, and innovation teams to design and deliver on your technical priorities.

Our App Dev team is made up of technology agnostic enthusiasts with a wide array of knowledge in popular modern and legacy languages such as Java, .NET, Ruby on Rails, Node.js & JavaScript, Python, and PHP.

We firmly believe that mentoring can be integrated with delivery. Our main focus is on saving our partners as much as possible on the lifetime-total-cost of ownership and maintainability of their systems. For more information, contact us at hello@levvel.io.