White Paper

embarcadero

Unit Testing in Delphi

By Nick Hodges

for

Embarcadero Technologies

February 2014

Introduction

Unit testing is an important topic. Very important. Unit testing can mean the difference between a codebase that is clean and maintainable and one that is impossible to maintain and imbued with "RADitis". (RADitis is also known as "OnClick Programming, or more precisely, coupling all your logic to your user interface.) All too often, Delphi apps catch a bad dose of RADitis. Luckily, Unit Testing is both the cure for and vaccine against this malady.

Why is unit testing such an effective vaccine? Because it forces you to do a number of things. It forces you to write clean modular code. It forces you to ensure that your code works before you even try to link it up into an application. It forces you to build a regression suite. And probably most importantly, it leads you, no, shoves you down the path of well ordered, decoupled, modular code. If your code is testable, it is by definition properly modular and uncoupled. That's the very definition of "Clean Code."

And Delphi developers are lucky. They have had for many years the venerable DUnit framework for doing unit testing. And recent advancements in the language have enabled a new framework for Delphi developers, DUnitX. DUnitX takes advantage of attributes to make building tests and test classes easier than ever before.

This Whitepaper discusses the why's and how's of Unit Testing in Delphi. It will make an impassioned plea for doing unit testing. It will show off the powerful language features that can really supercharge your unit tests and the proper techniques for writing good, easy to maintain unit tests. If this white paper can't get you writing tests, well, nothing can.

So here we go.

Unit Testing and Isolation Frameworks

> *Am I suggesting 100% test coverage? No, I'm demanding it. Every single line of code that you write should be tested. Period.*
>
> *I don't want management to mandate 100% test coverage. I want your conscience to mandate it as a point of honor.*
>
> *How about: if you have 100%, you can refactor savagely whenever you want with confidence.*

I took the above quotes from a Twitter conversation that Uncle Bob Martin of Eighth Light had with some of his followers. He is pretty passionate about code coverage and unit testing. We all might not be quite as serious as Uncle Bob, but we all definitely should be at least a little passionate about unit testing.

And that passion should stem from a desire to write good code. I'm going to make the case that the only way you can call your code "good" is if it is fully unit tested. If you write and deliver code that isn't completely unit tested, you should feel like you are walking down the street naked. You should feel exposed and like everyone is looking at you. You should believe firmly that your code will fail miserably unless it is completely covered by unit tests.

SO WHAT IS UNIT TESTING?

Unit testing is the testing of code to ensure that it performs the task that it is meant to perform. It tests code at the very lowest level possible — the individual methods of your classes. It is the key to writing clean, maintainable code. If you concentrate on writing code that is easily testable, you can't help but end up with decoupled, clean, high-quality code that is easy to maintain. What's not to like?

But sometimes there are questions over the definition of terms when it comes to unit testing. For instance, what, exactly, is a "unit"? What does "mocking" mean? How do I know whether I actually am doing unit testing? First, we'll cover what these terms all mean, and then we'll take a look at a simple example that illustrates the way to go about writing tests and code together.

WHAT IS A "UNIT"?

The first question that comes up when discussing unit testing is, well, what is a unit? You can't do unit testing without knowing what a unit is.

When it comes to unit testing, I view a "unit" as any discrete module of code that can be tested in isolation. It can be something as simple as a stand-alone routine (think `StringReplace` or `IncMonth`), but normally it will be a single class and its methods. A class is the basic, discrete code entity of modern languages. In Delphi, classes (and records which are conceptually very similar) are the base building blocks of your code. They are the data structures that, when used together, form a system.

In the world of unit testing, that class is generally referred to as the "Class Under Test (CUT)" or the "System Under Test (SUT)." You'll see those terms used extensively — to the point where it is strongly recommended that you use CUT as the variable name for your classes being tested.

> ***Definition**: A unit is any code entity that can be tested in isolation, usually a class.*

AM I ACTUALLY DOING UNIT TESTING?

So when you are doing unit testing, you are generally testing classes (And for the sake of the discussion, that will be the assumption hereafter.). But the key thing to note is that when unit testing a class, you are unit testing the given class and only the given class. Unit testing is always done in isolation — that is, the class under test needs to be completely isolated from any other classes or any other systems. If you are testing a class and you need some external entity, then you are no longer unit testing. A class is only "testable" when its dependencies can be "faked" and thus tested without any of its real external dependencies. So if you are running what you think is a unit test, and that test needs to access a database, a file system, or any other external system, then you have stopped doing unit testing and you've started doing integration testing.

One thing I want to be clear about: There's no shame in doing integration testing. Integration testing is really important and should be done. Unit testing frameworks are often a very good way to do integration testing. I don't want to leave you with the impression that because integration testing is not unit testing, you shouldn't be doing it — quite the contrary. Nevertheless, it is an important distinction. The point here is to recognize what unit tests are and to strive to write them when it is intended to write them. By all means, write integration tests, but don't write them in lieu of unit testing.

Think of it this way: Every unit test framework — DUnit and DUnitX included — creates a test executable. If you can't take that test executable and run it successfully on your mother's computer in a directory that is read only, then you aren't unit testing anymore.

> ***Definition**: Unit testing is the act of testing a single class in isolation, completely apart from any of its actual dependencies.*
>
> ***Definition**: Integration testing is the act of testing a single class along with one or more of its actual external dependencies.*

WHAT IS AN ISOLATION FRAMEWORK?

Commonly, developers have used the term "mocking framework" to describe code that provides faking services to allow classes to be tested in isolation. However, as we'll see below (and discuss more fully later in this paper), a "mock" is actually a specific kind of fake class, along with stubs. Thus, it is probably more accurate to use the term "Isolation Framework" instead of "Mocking Framework." A good isolation framework will allow for the easy creation of both types of fakes — mocks and stubs.

Fakes allow you to test a class in isolation by providing implementations of dependencies without requiring the real dependencies.

> ***Definition**: An isolation framework is a collection of code that enables the easy creation of fakes.*
>
> ***Definition**: A Fake Class is any class that provides functionality sufficient to pretend that it is a dependency needed by a class under test. There are two kind of fakes — stubs and mocks.*

If you really want to learn about this stuff in depth, I strongly recommend you read *The Art of Unit Testing: With Examples in .Net* by Roy Osherove. For you Delphi guys, don't be scared off by the C# examples — this book is a great treatise on unit testing and gives plenty of descriptions, proper techniques, and definitions of unit testing in far more detail than I will do here. Or you can listen to Roy talk to Scott Hanselman on the Hanselminutes podcast at http://hanselminutes.com/169/the-art-of-unit-testing-with-roy-osherove.

If you really want to get super geeky, get a hold of a copy of *xUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros. This heavy tome is a tour de force of unit testing, outlining a complete taxonomy of tests and testing patterns. It's not for the faint of heart, but if you read that book, you'll know everything there is to know and then some.

Stubs

A stub is a class that does the absolute minimum to appear to be an actual dependency for the Class Under Test. It provides no functionality required by the test, except to appear to implement a given interface or descend from a given base class. When the CUT calls it, a stub usually does nothing. Stubs are completely peripheral to testing the CUT, and exist purely to enable the CUT to run.

A typical example is a stub that provides logging services. The CUT may need an implementation of, say, `ILogger` in order to execute, but none of the tests care about the logging. In fact, you specifically don't want the CUT logging anything. Thus, the stub pretends to be the logging service by implementing the interface, but that implementation actually does nothing. Its implementing methods might literally be empty. Furthermore, while a stub might return data for the purpose of keeping the CUT happy and running, it can never take any action that will fail a test. If it does, then it ceases to be a stub, and it becomes a "mock."

> ***Definition**: A stub is a fake that has no effect on the passing or failing of the test and that exists purely to allow the test to run.*

Mocks

Mocks are a bit more complicated. Mocks do what stubs do in that they provide a fake implementation of a dependency needed by the CUT. However, they go beyond being a mere stub by recording the interaction between itself and the CUT. A mock keeps a record of

all the interactions with the CUT and reports back, passing the test if the CUT behaved correctly, and failing the test if it did not. Thus, it is actually the Mock, and not the CUT itself, that determines if a test passes or fails.

Here is an example — say you have a class `TWidgetProcessor`. It has two dependencies, an `ILogger` and an `IVerifier`. In order to test `TWidgetProcessor`, you need to fake both of those dependencies. However, in order to really test `TWidgetProcessor`, you'll want to do two tests — one where you stub `ILogger` and test the interaction with `IVerifier`, and another where you stub `IVerifier` and test the interaction with `ILogger`. Both require fakes, but in each case, you'll provide a stub class for one and a mock class for the other.

Let's look a bit closer at the first scenario — where we stub out `ILogger` and use a mock for `IVerifier`. The stub we've discussed — you either write an empty implementation of `ILogger`, or you use an isolation framework to implement the interface to do nothing. However, the fake for `IVerifier` becomes a bit more interesting — it needs a mock class. Say the process of verifying a widget takes two steps: first the processor needs to see if the widget is in the system, and then, if it is, the processor needs to check if the widget is properly configured.

Thus, if you are testing the `TWidgetProcessor`, you need to run a test that checks whether `TWidgetProcessor` makes the second call if it gets `True` back from the first call. This test will require the mock class to do two things: first, it needs to return `True` from the first call, and then it needs to keep track of whether or not the resulting configuration call actually gets made. Then it becomes the job of the mock class to provide the pass/fail information — if the second call is made after the first call returns `True`, then the test passes; if not, the test fails. This is what makes this fake class a mock: The mock itself contains the information that needs to be checked for the pass/fail criteria.

> ***Definition**: A mock is a fake that keeps track of the behavior of the Class Under Test and passes or fails the test based on that behavior.*

Most isolation frameworks include the ability to do extensive and sophisticated tracking of exactly what happens inside a mock class. For instance, mocks can not only tell if a given method was called, it can track the number of times given methods are called and the parameters that are passed to those calls. They can determine and decide if something is called that isn't supposed to be, or if something isn't called that is supposed to be. As part of the test setup, you can tell the mock exactly what to expect, and to fail if that exact sequence of events and parameters is not executed as expected. Stubs are fairly easy and straightforward, but mocks can get rather sophisticated.

Later, we'll take a closer look at the Delphi Mocks Framework found at https://github.com/VSoftTechnologies/Delphi-Mocks. It takes advantage of some cool new RTL features that first appeared in Delphi XE2. It's also a very generous and awesome gift to the Delphi community from Vincent Parrett, who makes the excellent FinalBuilder tool. (http://www.finalbuilder.com). If you have XE2 or newer and are doing unit testing, you

should get Delphi Mocks and use it. If you don't have XE2 or above and are doing unit testing, you should upgrade so you can start using this very valuable isolation framework.

But again, the whole point here is to test your classes in isolation; you want your CUT to be able to perform its duties without any outside, real, external dependencies.

Thus, a final definition:

> ***Definition**: Unit testing is the testing of a single code entity when isolated completely from its dependencies.*

WHY DO UNIT TESTING?

I find that there is a lot of resistance to doing unit testing. Many developers seem to view it as a waste of time or as effort that will merely delay the completion of a project under deadline. They feel that they can't get any benefit from it. I couldn't disagree more. Here's why.

UNIT TESTING WILL FIND BUGS

Whether you do Test Driven Development and write your tests first, write your tests as you go along, or write tests long after the code has been written, unit testing will find bugs. When you write a full suite of tests that define what the expected behavior is for a given class, anything in that class that isn't behaving as expected will be revealed.

UNIT TESTING WILL KEEP BUGS AWAY

A complete and thorough suite of unit tests will help to ensure that any bugs that creep into your code will be revealed immediately. Make a change that introduces a bug, and your tests can reveal it the very next time you run your tests. If you find a bug that is outside the realm of your unit test suite and you can write a test for it to ensure that the bug never returns.

UNIT TESTING SAVES TIME

This is the most controversial notion about unit testing. Most developers believe that writing unit tests takes more time than it saves. I don't believe this - in fact, I argue the opposite. Writing unit tests helps ensure that your code is working as designed right from the start. Unit tests define what your code should do, and thus you won't spend time writing code that does things that it shouldn't do. Every unit test becomes a regression test, ensuring that things continue to work as designed while you develop. They help ensure that subsequent changes don't break things. They help ensure that what you write the first time is the right thing. All of these benefits save time both in the short and the long term.

And if you think about it, you already test your code while you are writing it. Maybe you write a simple console app. At the very least you compile and see it running. No one checks in code that they don't believe works, and you have to do *something* to make yourself think that it works. Spend that time writing unit tests, and you'll have decoupled working coding with a suite of regression tests.

UNIT TESTING GIVES PEACE OF MIND

Having a full, complete, and thorough set of tests that cover the complete functionality of your code might be difficult to achieve, but having it will give you peace of mind. You can run all those tests and know that your code works as it is supposed to. You can refactor and change the code knowing that if you break anything, you'll know right away. Knowing the state of your code, that it works, and that you can update and improve it without fear is a very good thing.

All code ages, but you can keep your code from ever truly becoming legacy code. There are a number of ways to define legacy code, but one way is "Code you are afraid to touch". If your code has unit tests, it is really hard for it to become legacy code. Many of us have that chunk of code that we are afraid to touch — but with unit testing, you'll not have that kind of code. Unit testing removes that fear of touching code. In *Working Effectively with Legacy Code*, author Michael Feathers goes so far as to define legacy code as any code that doesn't have unit tests. Want to avoid your code becoming legacy code? Write unit tests for it.

UNIT TESTING DOCUMENTS THE PROPER USE OF A CLASS

One of the benefits of unit testing is that your tests can define for subsequent developers how the class should be used. Unit tests become, in effect, simple examples of how your code works, what it is expected to do, and the proper way to use the code being tested. Consumers of your code can look to your unit tests for information about the proper way to make your code do what it is supposed to do.

UNIT TESTING IN DELPHI

For many years, the main unit testing framework has been DUnit. (http://dunit.sourceforge.net/) Based on the original Delphi RTTI, it's been the mainstay of Delphi unit testing for a decade or more. DUnit is a complete xUnit testing solution, but while venerable and well-used, it is at this point somewhat outdated.

Recently a new framework has appeared called DUnitX, which can be found at https://github.com/VSoftTechnologies/DUnitX. DUnitX leverages attributes and Delphi's new

RTTI to make test classes and methods more flexible and easy to manage. As of this writing, DUnitX is quite usable but still under development.

DUnit

DUnit was originally published in the late 1990s and was written by Juanco Añez. He followed the original pattern set forth for all xUnit frameworks defined by Kent Beck. DUnit is an open source project that can be found on SourceForge. (http://dunit.sourceforge.net/)

Starting with Delphi 2005, DUnit has shipped with RAD Studio and has been integrated into the IDE. Developers can use the IDE expert to automatically generate test cases for a given Delphi unit.

DUnit is based on the "old" RTTI built into Delphi. Tests are based on a class called `TTestCase`. To use `TTestCase`, you create a descendant class and add published methods as tests. The framework then uses RTTI to find all the published methods and execute them.

Testing is done using a set of overloaded functions that start with `CheckXXX`, such as `CheckEquals` or `CheckNotEquals`. They take as parameters an expected value, an actual value, and an optional message parameter that lets you provide information for output in the event of a test failure. The `Check` calls raise an internal exception if the test fails, and in this way tests can be tracked as passing or failing. `TTestCase` also included virtual methods called `Setup` and `TearDown` that allow you to do as their names say — set up things for testing and then "tear down" anything you created in `Setup`. `Setup` and `TearDown` are run once for every test, allowing you to ensure that each test is run with a "clean slate."

DUnitX

DUnitX, on the other hand, utilized the new RTTI to allow any class you want to contain tests. Using the `[TestFixture]` attribute on a class allows it to contain methods that are tests. Any method on such a class can be labeled as a test by attaching the attribute `[Test]` to it. Setup and tear down is done by decorating any method with `[Setup]` and `[TearDown]` attributes. In addition, the framework allows for test fixture level setup and tear down with the `[SetupFixture]` and `[TearDownFixture]` attributes. These last two are called once per test run. DUnitX is also an open source project that can be found on GitHub at https://github.com/VSoftTechnologies/DUnitX.

As I said, as of this writing, DUnitX is still in development. It can be used to create Console applications for visual output. In addition, you can integrated it into your continuous integration solution by using the XML output. A graphical output is currently in the works.

General Rules for Unit Tests

Test one thing at a time in isolation

This is probably the baseline rule to follow when it comes to unit tests. All classes should be tested in isolation. They should not depend on anything other than mocks and stubs. They shouldn't depend on the results of other tests. They should be able to run on any machine. You should be able to take your unit test executable and run it on your mother's machine while it isn't even connected to the Internet.

Follow the AAA rule: Arrange, Act, Assert

When it comes to unit testing, AAA stands for "Arrange, Act, Assert." It is a general pattern for writing individual tests to make them more readable and effective. First, you "Arrange." In this step, you set things up to be tested. You set variables, fields and properties to enable the test to be run, as well as define the expected result. Then you "Act" — that is, you actually call the method that you are testing. Finally, you "Assert" — call the testing framework to verify that the result of your "Act" is what was expected. Follow the AAA principle, and your test will be clear and easy to read.

Here's an example of a test that follows the AAA rule:

```
procedure TDateUtilsOfTests.TestDateOf;
var
 Expected: TDateTime;
 Actual: TDateTime;
 Input: TDateTime;
begin
 //Arrange
 Input := EncodeDateTime(1944, 1, 2, 3, 4, 5, 6);
 Expected := EncodeDate(1944, 1, 2);
 //Act
 Actual := DateOf(Input);
 // Assert
 Assert.AreEqual(Expected, Actual);
end;
```

Write simple, "right down the middle" tests first

The first tests you write should be the simplest tests — the "happy path," They should be the ones that easily and basically illustrate the functionality you are trying to write. If you are writing an addition algorithm, the early tests that you write should make sure that your code can do 2 + 2 = 4. Then, once those tests pass, you should start writing the more complicated tests (as discussed below) that test the edges and boundaries of your code.

WRITE TESTS THAT TEST THE EDGES

Once the basics are tested and you know that your basic functionality works, you should test "the edges." That is, you should test what happens if an overflow occurs. What if values go to zero or below zero? What if they go to `MaxInt`? If you doing something with geometry, what if you try to create an arc of 361 degrees? What happens if you pass in an empty string? A string that is 2GB in size? A good set of tests will explore the outer edges of what might happen to a given method.

TEST ACROSS BOUNDARIES

Unit tests should test both sides of a given boundary. If you are building some tests for `TDateTime`, try testing one second before midnight and one second after. Test across the date value of `0.0`. If you are dealing with a structure that holds a rectangle, then test what happens to points inside and outside the rectangle. What about above or below? To the left or right? Above and to the right? Below and to the left? Moving across boundaries are places where your code might fail or perform in unpredictable ways.

IF YOU CAN, TEST THE ENTIRE SPECTRUM

If it is practical, test the entire set of possibilities for your functionality. If it involves an enumerated type, test the functionality with every one of the items in the enumeration. It might be impractical to test every possible string or every integer, but if you can test every possibility, do it.

IF POSSIBLE, COVER EVERY CODE PATH

This one is challenging as well, but if your code is designed for testing, and you make use of a code coverage tool, you can ensure that every line of your code is covered by unit tests at least once. Delphi has a very good code coverage tool available for your use — https://code.google.com/p/delphi-code-coverage/. Use it in concert with your unit tests. Covering every code path won't guarantee that there aren't any bugs, but it surely gives you valuable information about the state of every line of code.

WRITE TESTS THAT REVEAL A BUG, THEN FIX IT

This is a powerful and useful technique. If you find a bug, write a test that reveals it. Then, you can easily fix the bug by debugging the test. Then you have a nice regression test to make sure that if that bug comes back for any reason, you'll know right away. It's really easy to fix a bug when you have a simple, straightforward test to run in the debugger.

A side benefit here is that you've "tested your test." Because you've seen the test fail and then have seen it pass, you know that the test is valid in that it has proven to work correctly. This makes it an even better regression test.

MAKE EACH TEST INDEPENDENT OF EACH OTHER

Tests should never depend on each other. If your tests have to be run in a certain order, then you need to change your tests. Instead, you should make proper use of the Setup and TearDown features of your unit testing framework to ensure each test is ready to run individually. Unit tests frameworks don't guarantee that tests are going to be run in any particular order, and if your tests depend on tests running in a specific order, then you may find yourself with some subtle, hard to track down bugs in your tests themselves. Make sure each test stands alone, and you won't have this problem.

WRITE ONE ASSERT PER TEST

As a general rule, you should write one assert per test. If you can't do that, then refactor your tests so that your SetUp and TearDown events are used to correctly create the environment so that each test can be run individually. If your tests require elaborate setup and you feel the need to run multiple tests and call `Check/Assert` multiple times, then you need to create a new test case class and utilize its `Setup/TearDown` feature for that particular test case, enabling you to create multiple tests for the specific situation.

NAME YOUR TESTS CLEARLY. DON'T BE AFRAID OF LONG NAMES.

Since you are doing one assert per test, each test can end up being very specific. Thus, don't be afraid to use a long, complete test name. It is better to have `TestDivisionWhenNumPositiveDenomNegative` than `DivisionTest3`. A long complete name lets you know immediately what test failed and what exactly what the test was trying to do. Long, clearly named tests also can document your tests. For example, a test named `DivisionByZeroShouldThrowException` documents exactly what the code does when you try to divide by zero.

TEST THAT EVERY RAISED EXCEPTION IS IN FACT RAISED.

If your code raises exceptions, then write tests to ensure that every exception you raise in fact gets raised when it is supposed to. Both DUnit and DUnitX have the ability to test for an exception being raised, so you should use that feature to ensure that every exception your code raises is indeed raised under the proper circumstances.

AVOID THE USE OF CHECKTRUE OR ASSERT.ISTRUE

Avoid checking for a Boolean condition. For instance, instead of checking if two things are equal with `CheckTrue` or `Assert.IsTrue`, use `CheckEquals` or `Assert.AreEqual` instead. Why? Because this:

```
CheckTrue(Expected, Actual)
```

will report something like "*Some test failed: Expected True but actual result was False*". That doesn't tell you anything. Instead, use `CheckEquals`:

```
CheckEquals(Expected, Actual)
```

which will tell you the actual values involved, such as "*Some test failed: Expected 7 but actual result was 3.*"

CONSTANTLY RUN YOUR TESTS

Run your tests while you are writing code. Your tests should run fast, enabling you to run them after even minor changes. If you can't run your tests as part of your normal development process then something is going wrong — unit tests are supposed to run almost instantly. If they aren't, it's probably because you aren't running them in isolation.

RUN YOUR TESTS AS PART OF EVERY AUTOMATED BUILD

Just as you should be running your tests as you develop, they should also be an integral part of your continuous integration process. A failed test should mean that your build is broken. Don't let a failing test linger — consider it a build failure and fix it immediately.

TEST DRIVEN DEVELOPMENT

Probably the most controversial facet of unit testing is the question "When should I write my tests?" This question is controversial because the rise of unit testing coincided with the rise of the notion of Test Driven Development, or TDD. TDD says that you should write unit tests first, before anything else, and that your tests should drive your code and your design.

Some people object to TDD because they say that adding the writing of tests will add time to the project and make it either late or longer. Proponents say that the time spent up front will save more time in the long run because your code will work right the first time and you'll have a set of tests to prove it. I'm not here to settle this debate or even to engage much in it. Whether you decide to test first, test in the middle, or test after, I just want you to write tests.

The basic idea behind TDD is the notion of "Red, Green, Refactor." In other words, write tests that fail, write code that makes the tests pass, refactor your code so it is clean, and start again. The tests you write should define the correct behavior you want from your code. You then write code to make the tests pass. You then can improve — refactor — the code to improve it, all the while making sure your tests still pass. Once you are satisfied, you move on to the next requirement of your code, starting with a new test.

> *The "Red/Green" part comes from unit testing tools that show failed tests in red and passing tests in green. Unit test proponents are happy when they "see green" in their testing tool.*

You repeat this cycle until your code is complete and all requirements are met. At the end, you have code that can be proven, via your unit tests, to do what it was defined and required to do. The tests then also serve as a regression suite to ensure that any subsequent changes you make don't break the designed, required functionality.

That's a very quick overview of unit testing. For a more in depth view, I recommend the seminal work on the subject by Kent Beck called *Test Driven Development: By Example*. In this book, Beck lays out the entire case for TDD and takes you through the entire process. It is a must read for anyone interested in unit testing and TDD.

Although it's a complete change of development mindset, when you use TDD, the tendency is to write decoupled code: you are writing the tests first, and then you must write the code. If you write coupled code, you cannot run the test; therefore, you will design it in a decoupled way so that it can be safely tested. Another great advantage lies in the Refactor phase — in that phase you will see new ways to write your code (and learn a lot). Usually in this phase you will see a lot of code smells (http://en.wikipedia.org/wiki/Code_smell) and rewrite your code to remove them. After some time, your code is more decoupled and clean.

A Basic Example

So far all we've had is a lot of theory, so how about a little practice?

Let's say we have a requirement to write a class that does basic mathematical actions. It needs to add, subtract, multiply, divide, and provide a Power function (that is, x raised to the y). For simplicity's sake, we'll assume all the math will be done with integers.

And here's what we are going to do:

1. Write a test
2. Write code until the application compiles but the test fails
3. Write code until the test passes
4. Refactor until we are happy with our code
5. Go to step 1

And we'll do this until we are done and our class does everything it is supposed to do and we have tests that prove it.

So following TDD, the first thing we need to do is to write tests. We'll use the DUnitX testing suite because it is easier to use in our examples. Here's a class that will do a very basic test of our addition algorithm:

```
unit uCalculatorTests;

interface

uses
   uCalculator
  , DUnitX.TestFramework
  ;

type
 [TestFixture]
 TCalculatorTests = class
 private
  Expected, Actual: integer;
  Calculator: TCalculator;
  [Setup]
  procedure Setup;
  [TearDown]
  procedure TearDown;
 public
  [Test]
  procedure TestSimpleAddition;
 end;

implementation

procedure TCalculatorTests.Setup;
begin
 Calculator := TCalculator.Create;
end;

procedure TCalculatorTests.TearDown;
begin
  Calculator.Free;
end;

procedure TCalculatorTests.TestSimpleAddition;
begin
 Expected := 4;
 Actual := Calculator.Add(2, 2);
 Assert.AreEqual(Expected, Actual, 'The calculator thinks that 2 + 2 is
not 4!');
end;
```

```
end.
```

This is our first exposure to DUnitX, so some discussion is in order. First, the uses clause includes `DUnitX.TestFramework`. That unit is all you should need to include when building tests. It includes all the classes and interfaces you need to declare and write unit tests using the framework.

Next, you can declare a class as a "test fixture" by tagging it with the `[TestFixture]` attribute. A Test Fixture is a class that contains tests. Any class will do — it doesn't matter what the class descends from, as long as it contains that attribute, it will be scanned for tests. This makes it pretty easy to write tests, as you don't have to descend from a particular class as you do in the DUnit framework.

In order to declare an actual test, you add a method to your Test Fixture that has the `[Test]` attribute attached to it. By convention, I like to make these methods `public`. Any method that has the `[Test]` attribute will be run by the DUnitX framework.

It is also common to declare a set of common fields in the `private` section of the Test Fixture. Very often, test methods will all require a similar set of variables, and it makes sense to declare them as class fields. In the above case, there are two variables, `Expected` and `Actual`, which will likely be used by every test, so they are declared in common.

We are going to be testing the `TCalculator` class, so naturally we'll need an instance of `TCalculator`. Each test should run completely independent of each other, and so we'll want to create a new instance for every test. We could do that in each `[Test]` method, but that would be tedious. Unit testing frameworks normally provide a means for simplifying this repetitive process via the notion of “Setup” and “TearDown.” Via the aptly named attributes `[Setup]` and `[TearDown]`, you can write code that will be called for every test. Thus, in our case, we can create and destroy an instance of `TCalculator` for each test. Any method tagged with the `[Setup]` attribute will be run at the start of every test, and any method tagged with the `[TearDown]` attribute will be run at the end of every test.

Though not shown here, you can also do Setup and TearDown at the Test Fixture level by declaring methods on the Test Fixture with the `[SetUpFixture]` and `[TearDownFixture]` attributes. Any methods declared with these attributes will be run at the creation and destruction of the Test Fixture, respectively. You should only have one of each of these attributes in your class.

In order for an actual test to be run, you must call a method of the `Assert` class. `Assert` has a number of class methods on it that do the actual testing. The `Assert` class is defined in the `DUnitX.TestFramework` unit. You can see it there. `Assert` allows you to check most anything and has many overloads to ensure that you can test any data type.

All of the methods take an optional final parameter of type `string`. This allows you to add your own message on to the call to `Assert` so that you can explain the exact nature of the

error based on the test. You can use this string to uniquely identify the test error and completely explain why it failed. This makes finding and fixing failing tests much easier. Don't be afraid to provide a complete, specific message explaining the problem.

Okay, back to our example. Now here's the fun part: Our code won't compile. It claims to use a unit called `uCalculator`, but that unit is completely empty right now. We haven't written any code for it yet because we wrote our test first. And it's a really simple test — checking to see if our calculator can figure out that two and two is four. Not a tough test, but a critical one. If that doesn't work, then nothing will.

So, in our TDD example here, our first step is done: we've written a test. The second step: get the code to compile.

In order to do that, we'll create a `TCalculator` class with an `Add` method:

```
unit uCalculator;

interface

type
 TCalculator = class
  function Add(x, y: integer): integer;
 end;

implementation

{ TCalculator }

function TCalculator.Add(x, y: integer): integer;
begin

end;

end.
```

Okay, so now the code compiles. It doesn't do anything, and our test still fails, but it compiles. Step Two is complete.

For Step Three, we'll write code until our test passes. The first thing that we need to do is to register our test class. In order for DUnitX to do its magic, it needs to know that our Test Fixture is available for testing. Thus, in the initialization section of our test unit we'll add the following:

```
initialization
 TDUnitX.RegisterTestFixture(TCalculatorTests);
```

Next, we'll update our `DPR` file to actually run DUnitX code. First, we'll add these three units to the uses clause:

```
DUnitX.TestFramework,
DUnitX.Loggers.Console,
DUnitX.Windows.Console,
```

`DUnitX.TestFramework` is the unit that contains all the code for DUnitX's main interface.

`DUnitX.Loggers.Console` contains a class that implements `ILogger`. The `ILogger` interface is the one that provides the output of the tests. In this case, the unit includes the `TDUnitXConsoleLogger` class that outputs the test results to the console.

`DUnitX.Windows.Console` contains a cool class that provide colors to the console, allowing us to do things like output green results for passing tests, red results for failing tests, and purple for setup code.

Then we have to declare three variables:

```
var
 Runner: ITestRunner;
 Logger: ITestLogger;
 Results: ITestResults;
```

`Runner` is the interface that will actually run the tests. `Logger` is the reference to the console runner discussed above. And `Results` is the interface that will hold the results of the tests. We won't actually do anything with the results in our simple example, but if you wanted to log or otherwise record the results somewhere, you could do this using the `Results` variable.

Then, we'll use those variables in the `DPR` file:

```
try
 //Create the runner
 Runner := TDUnitX.CreateRunner;
 Runner.UseRTTI := True;
 //tell the runner how we will log things
 Logger := TDUnitXConsoleLogger.Create;
 Runner.AddLogger(Logger);
 //Run tests
 Results := Runner.Execute;
 System.Write('Done.. press the <Enter> key to quit.');
 System.Readln;
except
 on E: Exception do
  System.Writeln(E.ClassName, ': ', E.Message);
end;
```

Basically we create and use the runner. This is fairly boilerplate code, so I won't discuss it too much. It just runs the tests, displays the output in the console, and waits for you to press the `Enter` key.

The real fun is that the code will now compile and run. Sure, the test fails, but we have more code to write. We'll keep writing until the test passes.

And making the test pass is really easy. How about this:

```
function TCalculator.Add(x, y: integer): integer;
begin
 Result := 4;
end;
```

Now, we run our test, and it passes. Yay!

At this point, let's add another test.

```
procedure TCalculatorTests.AddingOneAndOneShouldReturnTwo;
begin
 Expected := 2;
 Actual := Calculator.Add(1, 1);
 Assert.AreEqual(Expected, Actual, 'The calculator thinks that 1 + 1 is not
2!');
end;
```

Now we'll run both tests, and — uh-oh — that new test doesn't pass. Looks like it's time to refactor by changing the code in our `Add` function to fix it.

In order to make both of the tests pass, we need to actually do an addition algorithm. How about this one:

```
function TCalculator.Add(x, y: integer): integer;
begin
 Result := (x + 6 + y - y + y - 6) * 2 div 2;
end;
```

(I know, it's ridiculous, but it works. Bear with me....)

Yay! Now we have passing tests! Step Three complete!

So let's keep refactoring. Clearly the above algorithm will work just fine — it gets the correct result for addition. But I'm guessing that a better one exists.

How about this:

```
function TCalculator.Add(x, y: integer): integer;
begin
 Result := (x + y - y + y) * 2 div 2;
end;
```

If we change the algorithm to the above, our test still passes. We've made changes, and our tests still pass, so we know that our changes are safe. How about we add a test to test if we can properly add zero. That seems like a bit of an edge case:

```
procedure TCalculatorTests.TestAddingZero;
begin
 Expected := 4;
 Actual := Calculator.Add(4, 0);
 Assert.AreEqual(Expected, Actual, 'The calculator thinks that 4 + 0 is
not 4!');
end;
```

Running that results in all of our tests passing.

How about adding negative numbers:

```
procedure TCalculatorTests.TestAddingTwoNegativeNumbers;
begin
 Expected := -4;
 Actual := Calculator.Add(-2, -2);
 Assert.AreEqual(Expected, Actual, 'The Add function failed to realize
that -2 + -2 is -4');
end;
```

That passes, as well. So now we can move on to Step Four with some confidence. (I'll bet you can think of some other tests that might fit in here. What happens when you add a positive and a negative? What about sums that cross zero? Does all that work? I'll leave that as an exercise for the reader.)

Our addition algorithm is proving to be effective, but you aren't satisfied. It looks a little goofy, and you go to that Math PhD you know, and he suggests that you use the following instead:

```
function TCalculator.Add(x, y: integer): integer;
begin
 Result := x + y + 1;
end;
```

That looks a little simplistic, but a PhD is a PhD, and so you add the refactored test to your code. But uh oh! It fails all the tests. There must be a problem with the new algorithm. That's not good.

But what is good is that you know right away that there is an issue, and you have a set of tests that can be easily debugged to find the problem. No finding the bug after delivery to QA, no stepping through complex application code to find the issue — just a quick debugging session in a ready-made test to find the issue. In our case, that pesky "+ 1" is the issue. Remove that, and lo and behold, it works! All the tests pass! Step Four is now complete! WooHoo!

Step Five says to start the whole process over. In our case, we would a test for the `Subtract` method, see it fail, write code until it passes, refactor to our satisfaction, and then start in on `Multiply` and `Divide`. A simple example, yes, but it should give you a taste of what the process should be like and how you can use the simple five step pattern to write tests and code together.

And of course, the end result is a class that meets requirements and has a load of unit tests that can be used as regression tests and that give confidence that you can refactor and alter your code if desired without fear of things breaking. Your changes might break things, but you'll know immediately because you are running your tests constantly as you debug. Your tests can reveal bugs as you write them, and thus you can fix them immediately.

Now this example is a simple one. The tests all lack something that would probably be very prevalent in your real code — dependencies. As previously noted above, the key thing that you need to do when unit testing is test classes *in isolation*, meaning that the tests for the class should not be dependent on anything. As discussed above, dependencies need to be faked, and so in the next section, we'll look at how you can create fake classes to take the place of real dependencies in order to properly isolate your classes for proper testing.

Testing with an Isolation Framework

In the previous section, we looked at the whys and hows of unit testing. We went through a very simple example of how to do Test Driven Development. But as I mentioned, it was simple — too simple, in fact for real world use.

Good developers know that they should constantly be aware of the notion of testing code in isolation, and the decoupling of code via Dependency Injection. So now, it's time for it all to come together. The proper use of Dependency Injection should make testing your classes in isolation a piece of cake. I exhorted you to create a testing executable that could run on any machine without any specific dependencies present. But if you are going to do that, you'll have to substitute your actual dependencies with ones that don't actually couple to anything "real." If only there was a way to do that simply and easily.

Of course there is a way to do all that: in this case, it's Isolation Frameworks. We discussed them a bit in the last section, talking about the difference between Stubs and Mocks and how you can use them to test your classes in isolation. I promised that in this section we'd dive deeper into unit testing by showing how to use fakes — that is, stubs and mocks — to be able to test almost any code with any dependencies.

A Quick Review

Okay, first, let's do a quick review. First, an isolation framework is a set of classes that enables you to provide fake dependencies for your classes. There are two kinds of fakes — stubs and mocks. Stubs do nothing other than the very minimum to replace their real counterpart. A test success or failure won't depend on a stub, and generally, a stub will do nothing. A test should never fail because of a stub. A mock, however, is a fake representation that can and should provide feedback and information to a given test. In fact, the reason to use a mock instead of a stub is to interact with the Class Under Test and provide the ability to fail the test. That is the big distinction between a mock and a stub: a mock can fail a test where a stub never should.

Isolation Frameworks for Delphi

While there were mocking frameworks for Delphi before the introduction of virtual interfaces and generics, a true and complete isolation framework really wasn't available until XE2. The Delphi Mocks Framework (https://github.com/VSoftTechnologies/Delphi-Mocks) (as noted, it is common to call Isolation Frameworks "Mock Frameworks") is an open source project built by none other than Vincent Parrett of FinalBuilder fame — yes, the same guy that leads the development of the DUnitX project (He's a busy guy....).

Getting Started

In their simplest form, a fake object is an alternate implementation of a class that provides "fake" responses to method calls. For example, you have an interface `ICustomer` that has a method `GetCustomerName`, and normally, that call goes to the production database and gets the name (a simple, unlikely example, I know, but you get the idea). So to avoid the call to the database, you just create `MockCustomer` and implement its call to `GetCustomerName` and have it return "George Jetson" every time. This enables you to test the class that is using `ICustomer` without having to hit the database at all.

But that can get a bit clumsy. What if you want to return different values based on different inputs? What if you find a bug based on specific input or output, and you want to create a unit test for that specific case? What if you don't want to return anything at all because you are testing something different, but need to call to `GetCustomerName`? Then a specially created fake class as described above gets harried, complicated, and hard to maintain.

Enter an Isolation Framework

What if we could have a framework that would allow us to implement any interface and define easily and exactly what the inputs and outputs should be? That would be cool. This

would enable you to easily create a fake object that can respond to method calls in defined ways in a flexible, easy to set up manner. This is what an isolation framework does.

Obviously something this flexible needs some powerful language features. Such a framework would have to be able to flex to dynamically implement an interface. It would have to be able to dynamically recognize method calls and respond accordingly. Fortunately, as of XE2, Delphi is up to the task. Delphi XE2 introduced the `TVirtualInterface` class that lets you dynamically implement any interface at run-time. Combine that with the new RTTI, and you have the ability to build a very powerful mocking framework.

A Simple Stub

The Delphi Mocks Framework can easily create a simple stub for you to use to stub out a dependency that you don't need for a given test. For instance, say you had a class as follows:

```
type
 TDollarToGalleonsConverter = class
 private
  FLogger: ILogger;
 public
  constructor Create(aLogger: ILogger);
  function ConvertDollarsToGalleons(aDollars: Double; aExchangeRate:
Double): Double;
 end;


 function
TDollarToGalleonsConverter.ConvertDollarsToGalleons(aDollars,
aExchangeRate: Double): Double;
 begin
  Result := aDollars * aExchangeRate;
  FLogger.Log(Format('Converted %f dollars to %f Galleons', [aDollars,
Result]));
 end;

 constructor TDollarToGalleonsConverter.Create(aLogger: ILogger);
 begin
  inherited Create;
  FLogger := aLogger;
 end;
```

It has a single dependency of type `ILogger`, which passed to it via constructor injection. Normally, the logger will make an entry in the database every time you convert Dollars to Galleons, but you don't want that to happen for fake transactions that will happen when you run your unit tests. Instead, you'd like to simply ignore the logging as part of testing of the conversion. In other words, you want a stub for the `ILogger` interfaces.

First, let's look at `ILogger`:

```
type
 ILogger = interface(IInvokable)
 ['{B571A28D-A576-4C83-A0D3-CB211435CDEA}']
  procedure Log(aString: string);
 end;
```

This is a typical interface with one exception — it augments the `IInvokable` interface. `IInvokable` is simply an interface that has the `{M+}` switch turned on. This is required by the Delphi Mocks Framework in order for it to stub or mock and interface. It is probably a good idea to descend all of your interfaces from `IInvokable` for this reason.

Once you have your class ready to test, here's a test that you can write that creates a stub for `ILogger` and lets you focus on testing the conversion process:

```
procedure TDollarToGalleonConverterTest.TestPointFiveCutsDollarsinHalf;
var
 Expected: Double;
 Actual: Double;
 TempConverter: TDollarToGalleonsConverter;
 TempLogger: TMock<ILogger>;
begin
 //Arrange
 TempLogger := TMock<ILogger>.Create;
 TempConverter := TDollarToGalleonsConverter.Create(TempLogger);
 try
  Expected := 1.0;
  //Act
  Actual := TempConverter.ConvertDollarsToGalleons(2, 0.5);
  //Assert
  Assert.AreEqual(Expected, Actual, 'Converter failed to convert 2
dollars to 1 galleon');
 finally
  TempConverter.Free;
 end;
end;
```

Creating the stub is as simple as creating an instance of `TMock<ILogger>`. `TMock` is a generic record that takes an interface. The interface has to have the `{M+}` compiler flag attached to it. You can either add the directive yourself or descend all your interfaces from `IInvokable` as mentioned above. Once a `TMock<T>` is created, it assumes the role of the interface type passed to it. Thus, you can pass the resulting type to the constructor of `TDollarToGalleonConverter` class and none shall be the wiser. If the class calls methods on the logger, they will be ignored by the fake instances of `ILogger`.

Thus, that is all there is to creating a simple stub interface for use in our tests. You can now test `TDollarToGalleanCoverter.ConvertDollarsToGalleons` in isolation, apart from any specific implementation of a logger.

TESTING THE LOGGER

Now you've written tests to ensure that the currency converter works as it should. But you still have something else to test — that is, does the logger actually do what it is supposed to do when you make a conversion? You expect (remember that word) that the logger will make a single call to the `Log` method whenever a conversion occurs, but how can you check to make sure that actually happens and that the class using the Logger is using it correctly?

This is where the second kind of fake comes in — mocks. We discussed before that Mocks are a fake class that actively takes part in a unit test and can actually fail a test. In our case we want a mock that will make sure that the `Log` method will be called and that the correct thing will be logged.

Consider the following test:

```
  procedure
TDollarToGalleonConverterTest.TestThatLoggerIsProperlyCalled;
  var
   Expected: Double;
   Actual: Double;
   TempConverter: TDollarToGalleonsConverter;
   TempLogger: TMock<ILogger>;
   Input: Double;
   TempMessage: string;
  begin
   //Arrange
   TempLogger := TMock<ILogger>.Create;
   Input := 2.0;
   Expected := 1.0;
   TempMessage := Format('Converted %f dollars to %f Galleons', [Input,
Expected]);
   TempLogger.Setup.Expect.Once.When.Log(TempMessage);
   TempConverter := TDollarToGalleonsConverter.Create(TempLogger);
   try
    //Act
    Actual := TempConverter.ConvertDollarsToGalleons(Input, 0.5);
    //Assert
    TempLogger.Verify();
   finally
    TempConverter.Free;
   end;

  end;
```

Some things to note:

- The setup for the mock must be done before the mock is sent to the Class Under Test.

- The setup code for the mock is using a fluent interface, where each call leads to the next. One could read the code as "*For the TempLogger, do the setup by expecting that the method will be called once, and when it is, it will be passed a given string*"
- The call to `Verify` is the assertion that you are making for the test. This is how a mock can fail the test. If the things that were expected didn't happen, `Verify` will raise an exception and fail the test.
- You can set up as many expectations for a given mock as you want. You can expect that things will happen `Once`, `Never`, `AtLeastOnce`, `Exactly` a specified number of times, `AtLeast` a number of times, `AtMost` a number of times, `Between` a number of times, or `Before` and `After` other methods. In each case, you can also specify exactly what parameters are passed `When` that specific call is made.
- In other words, you can `Expect` whatever you want to have happen to the class during the test, and if those specified things don't happen in the way you say that they should happen, then the test will fail.
- The `When` property returns an instance of the interface itself, so that is the point where you will make the call to the method of the interface you want to test. There you pass all the parameters as they would be passed in the test itself.

STUBS THAT DO STUFF

Sometimes in the process of testing you need your stub to return a value from a function call. As your tests run, you know that your CUT will be calling a function on your stub, and you want it to behave in a predetermined way. No problem — Delphi Mocks allows you to tell your stubs to return values of functions. Instead of calling `Expect`, you can call `WillReturn`, passing to it an expected value as well as the actual method call with parameters via the `When` method.

Consider the following interface and implementing class:

```
unit uCreditCardValidator;

interface

uses
   SysUtils;

type
 ICreditCardValidator = interface(IInvokable)
 ['{68553321-248C-4FD4-9881-C6B6B92B95AD}']
  function IsCreditCardValid(aCreditCardNumber: string): Boolean;
  procedure DoNotCallThisEver;
 end;

 TCreditCardValidator = class(TInterfacedObject, ICreditCardValidator)
  function IsCreditCardValid(aCreditCardNumber: string): Boolean;
  procedure DoNotCallThisEver;
```

```
 end;

 ECreditCardValidatorException = class(Exception);

implementation

uses
   Dialogs;

function TCreditCardValidator.IsCreditCardValid(aCreditCardNumber:
string): Boolean;
begin
 // Let's pretend this calls a SOAP server that charges $0.25 everytime
 // you use it.

 // For Demo purposes, we'll have the card be invalid if it the number
7 in it
 Result := Pos('7', aCreditCardNumber) <= 0;
 WriteLn('Ka-Ching! You were just charged $0.25');
 if not Result then
 begin
  raise ECreditCardValidatorException.Create('Bad Credit Card! Do not
accept!');
 end;
end;

procedure TCreditCardValidator.DoNotCallThisEver;
begin
 // This one will charge the company $500! We should never
 // call this!
end;

end.
```

This code (pretends) to validate credit cards. It “charges” your company $0.25 every time you use it, so it is a dependency that performs an action that you need in order to test the class that uses it, but one that you obviously don't want to have run every time your tests run.

Here's a class that uses it:

```
unit uCreditCardManager;

interface

uses
   uCreditCardValidator
  , SysUtils
  ;

type
 TCreditCardManager = class
```

```
 private
  FCCValidator: ICreditCardValidator;
 public
  constructor Create(aCCValidator: ICreditCardValidator);
  function CreditCardIsValid(aCCString: string): Boolean;
  function ProcessCreditCard(aCCString: string; aAmount: Double):
Double;
 end;

 EBadCreditCard = class(Exception);

implementation

function TCreditCardManager.CreditCardIsValid(aCCString: string):
Boolean;
begin
 inherited;
 Result := FCCValidator.IsCreditCardValid(aCCString);
end;

function TCreditCardManager.ProcessCreditCard(aCCString: string;
aAmount: Double): Double;
begin
 if CreditCardIsValid(aCCString) then
 begin
  // Charge the card
  Result := aAmount;
 end else
 begin
  Result := 0.0;
 end;
end;

constructor TCreditCardManager.Create(aCCValidator:
ICreditCardValidator);
begin
 inherited Create;
 FCCValidator := aCCValidator;
end;

end.
```

The `TCreditCardManager` class has a dependency on `ICreditCardValidator`. But if you want to test `TCreditCardManager`, you don't want to depend on the real implementation of `ICreditCardValidator` because that wouldn't allow your tests to run in isolation. However, when you test `TCreditCardManager`, you need the `ICreditCardValidator` to behave in a certain way — either accept or reject the card. You can tell the stub exactly what to do.

Here's the test for a passing card:

```
procedure
TestTCCValidator.TestCardChargeReturnsProperAmountWhenCardIsGood;
var
 CCManager: TCreditCardManager;
 CCValidator: TMock<ICreditCardValidator>;
 GoodCard: String;
 Input: Double;
 Expected, Actual: Double;
begin
 //Arrange
 GoodCard := '123456';
 Input := 49.95;
 Expected := Input;
 CCValidator := TMock<ICreditCardValidator>.Create;
 CCValidator.Setup.WillReturn(True).When.IsCreditCardValid(GoodCard);

 CCManager := TCreditCardManager.Create(CCValidator);
 try
  //Act
  Actual := CCManager.ProcessCreditCard(GoodCard, Input)
 finally
  CCManager.Free;
 end;
 // Assert
 Assert.AreEqual(Expected, Actual);
end;
```

Here, we call `WillReturn(True)` on the stub, allowing the credit card manager to return the proper amount charged. Next, we can tell the stub to return `False` for the validation and make sure that the credit card manager returns zero in that case. The stub then doesn't affect whether the test passes or fails, it merely exists purely for the purpose of letting the actual test run in the way that you want it to run.

Here is the test to ensure that the credit card manager returns $0.00 when the card is invalid:

```
procedure TestTCCValidator.TestCardChargeReturnsZeroWhenCCIsBad;
var
 CCManager: TCreditCardManager;
 CCValidator: TMock<ICreditCardValidator>;
 GoodCard: String;
 Input: Double;
 Expected, Actual: Double;
begin
 //Arrange
 GoodCard := '777777'; // 7 in a card makes it bad.....
 Input := 49.95;
 Expected := 0;
 CCValidator := TMock<ICreditCardValidator>.Create;
 // Tell the stub to make it a bad card
 CCValidator.Setup.WillReturn(False).When.IsCreditCardValid(GoodCard);
```

```
 CCManager := TCreditCardManager.Create(CCValidator);
 try
  //Act
  Actual := CCManager.ProcessCreditCard(GoodCard, Input)
 finally
  CCManager.Free;
 end;
 // Assert
 Assert.AreEqual(Expected, Actual);
end;
```

DEPENDENCIES THAT DO EXPECTED THINGS

Stubs are fakes that do nothing, or at least don't do anything that can cause the test to fail. In fact, a Stub might be defined as a fake for which you have no declared expectations. Mocks, on the other hand, are fakes that do cause tests to fail and which do have declared expectations. Often the reason for that failure is that your class under test interacts with a dependency in an unexpected way.

Mocks allow you to ensure that the interactions with a dependency are what is expected. `TMock<T>` allows you to state what the expectations are for a given interaction with a dependency and then to verify that those expectations were met.

For instance, here is a simple class that manages a mailing list. It can take names and email addresses, and send out single or bulk emails (Okay, it can't do anything at all like that — it only pretends to do that. But this is a simple demo.). Anyway, here's the class:

```
TEmailListManager = class
private
 FEmailSender: IEmailSender;
public
 constructor Create(aEmailSender: IEmailSender);
 procedure RegisterNewPerson(aName: string; aEmailAddress: string);
end;

...

constructor TEmailListManager.Create(aEmailSender: IEmailSender);
begin
 inherited Create;
 FEmailSender := aEmailSender;
end;

procedure TEmailListManager.RegisterNewPerson(aName, aEmailAddress:
string);
begin
 // Insert person and email address into database
 // Then send a confirmation email
 FEmailSender.SendMail(aEmailAddress, 'Thanks for signing up'!);
```

```
end;
```

This class merely pretends to keep track of people who sign up for a mailing list. The important thing here is the code in the `RegisterNewPerson` method, where it uses its dependency to send a single email confirming that a person has signed up.

If you want to test this class, you don't actually want it to send out emails — you want to fake that part. But when you test adding someone to the database (note that I have for simplicity's sake left out the dependency which would do that) , you want to be sure that the dependency actually does make a call to the code that would send out that email. This is, again, what Mocks do — they validate that sort of thing.

So in order to do that, you'd write the following test:

```
procedure TEmailManagerTester.TestAddingPersonSendingOneEmail;
var
 CUT: TEmailListManager;
 MockSender: TMock<IEmailSender>;
 StubSL: TMock<TStringList>;
begin
 // Arrange

 MockSender := TMock<IEmailSender>.Create;
 MockSender.Setup.Expect.Once.When.SendMail(TestEmail, TestMessage);
 MockSender.Setup.Expect.Never.When.SendBulkEmail;

 CUT := TEmailListManager.Create(MockSender);
 try
  // Act
  CUT.RegisterNewPerson('Marvin Martian', TestEmail);
 finally
  CUT.Free;
 end;

 // Assert
 MockSender.Verify();

end;
```

Here are some things to note:

- A mock can set up any number of expectations for a given test. In this case, `MockSender` creates two expectations that need to be met for the test to pass.
- First, it expects that the `SendMail` procedure will be called `Once` and only once. If it is called more than once or not at all, the test will fail.
- Second, it ensures that the `SendBulkEmail` method is `Never` called during the process of signing someone up. If it is called, then the test will fail.

- In order to determine whether or not the test passes, a call is made to `MockSender.Verify`. If the expectations are met, then the call does nothing. If an expectation is not met, then the call will raise an exception, resulting in a failing test.

In this way, you can make sure that your Class Under Test is doing the proper things with your dependencies without actually having to create a real dependency.

DEPENDENCIES THAT RAISE EXCEPTIONS

Often, your dependencies will raise exceptions, and you need to ensure that your testing class handles that properly. Here's an interface and an implementing class that raises an exception when you try to validate a bad widget. In this case, we need a Mock, because we want to let the test pass or fail depending on whether or not the dependency raises the exception as expected. Basically, any time you are using a fake and you can call the `Verify` method, you are using a mock.

So for example, consider this unit:

```
unit uDependencyRaisesObjection;

interface

uses
   SysUtils
  ;

type
 IWidget = interface(IInvokable)
  function IsValid: Boolean;
 end;

 TWidget = class(TInterfacedObject, IWidget)
 public
  function IsValid: Boolean;
 end;

 IWidgetProcessor = interface(IInvokable)
  procedure ProcessWidget(aWidget: IWidget);
 end;

 TWidgetProcessor = class(TInterfacedObject, IWidgetProcessor)
 public
  procedure ProcessWidget(aWidget: IWidget);
 end;

 EInvalidWidgetException = class(Exception);

implementation
```

```
procedure TWidgetProcessor.ProcessWidget(aWidget: IWidget);
begin
 try
  if aWidget.IsValid then
  begin
   WriteLn('Widget has been properly processed');
  end;
 except
  On E: EInvalidWidgetException do
  begin
   WriteLn('IsValid failed to validate the widget');
  end;
 end;
end;

function TWidget.IsValid: Boolean;
begin
 // Just for demo purposes, lets say that 1 in 100 widgets are bad
 // But then again, we'll never call this code because it will be
mocked out
 Result := Random(100) >= 99;
 if not Result then
 begin
  raise EInvalidWidgetException.Create('Bad Widget! Bad, bad widget!');
 end;
end;

end.
```

In it you should notice that the `TWidget.IsValid` call has the potential to raise an exception. The `TWidgetProcessor` class uses `TWidget`, and thus should know what to do in the case of `TWidget` raising an `EInvalidWidgetException` exception. The `TWidgetProcessor.ProcessWidget` method has logic to properly handle the exception, but we want to test that, right?

In order to do that, we need to create a mock dependency based on `IWidget` that will raise the exception and make sure that `TWidgetProcessor` correctly handles it.

```
procedure TTestWidgetProcessor.TestBadWidgetRaisedException;
var
 CUT: IWidgetProcessor;
 MockWidget: TMock<IWidget>;
begin
 // Arrange
 MockWidget := TMock<IWidget>.Create;
 MockWidget.Setup.WillRaise(EInvalidWidgetException).When.IsValid;
 CUT := TWidgetProcessor.Create;
 // Act
 CUT.ProcessWidget(MockWidget);
 // Assert
```

```
    MockWidget.Verify();
  end;
```

This test is similar to the one in the previous section, except it establishes that a given exception type — `EInvalidWidgetException` — will be raised when `IsValid` is called. The test also makes sure that IsValid is only called once — no sense in letting code be called more than it need be.

Now you might look at that code above and say "But wait, you called `IsValid`, and yet it wasn't valid.". Well, yes — but remember, sometimes `IsValid` can raise an exception, and that is what you are telling the mock to do. In essence, the above code says "*Call `IsInvalid`, raise an exception, and see if `TWidgetProcessor` handles the exception correctly. If it does, pass the test. If it doesn't, fail the test.*" Again, this is an example of a Mock determining whether or not a test fails.

ONLY ONE MOCK PER TEST

A given test should only ever have one mock per test. For instance, if you have a class that has three dependencies, you should always test that class with zero or one mocks and three or two stubs. You should never have two mocks because then you'd have two different ways for the tests to fail. Having only one mock means that the test can only fail one way. No matter how many dependencies you have, only one of them should be a Mock, and the rest should be stubs.

EXPECTATION PARAMETERS MUST MATCH

When you set an expectation and make a call to the interface, you must pass in parameters. Those parameters must then be used exactly as "expected" because if they are not, the Mock won't properly verify. In other words, those parameter values are expected just as much as the behavior itself. So for instance, if you set the following expectation

```
SomeMock.Setup.Expect.Once.When.Add(5, 9);
```

then that expectation will not be met if you call the actual test with

```
MyAdder.Add(4, 3);
```

The parameters must match exactly. If the parameter is a reference type, the reference must be exactly the same one — a different reference will cause the mock not to verify.

Conclusion

That should give you a good overview of Isolation Frameworks. Isolation Frameworks exist to allow you to test your classes in isolation. If you've used proper Dependency Injection, then testing your classes in isolation along with an Isolation Framework should be very easy. Stubs allow you to test classes by providing expectation-less fake classes and interfaces. Mocks allow you to test your class's interaction with its dependencies by providing numerous ways to verify that your Class Under Test behaved as expected when tested — all without having to create real dependencies.

Next Steps

Learn more about Delphi and download the latest software at www.embarcadero.com/delphi

About The Author

This white paper has been written for Embarcadero Technologies by Nick Hodges. Nick Hodges has been a part of the Delphi community from the very beginning. He is an original Delphi 1 beta tester, a former member of TeamB, an Advisory Board member for the annual Borland Conference, a frequent conference speaker, a blogger and author of numerous articles on a wide range of Delphi topics.

Nick has a BA in Classical Languages from Carleton College and an MS in Information Technology Management from the Naval Postgraduate School. In his career he has been a busboy, a cook, a caddie, a telemarketer (for which he apologizes), an Office Manager, a high school teacher, a Naval Intelligence officer, a software developer, a product manager, and a software development manager. In addition, he is a former Delphi Product Manager and Delphi R&D Team Manager. He lives with his family in Gilbertsville, PA.

About Embarcadero Technologies

Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance, and accelerate innovation. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.