Automated Unit Testing in Delphi

Malcolm Groves – malcolm@madrigal.com.au

The Problem

How many times has this happened to you? You have to make some modifications to some code that either wasn't written by you, or that you wrote quite some time ago. So, you spend a bit of time getting (re-)aquainted with the code, make your changes, run the code a few times and check in your changes. A weeks or so later, when QA have finished testing the next build, you get a list of bugs as long as your arm. Who knew those modifications would have such a broad impact? Also, you've got to spend a bunch of time getting to know the code again, and then even more time tracking down the problems.

Chances are, this is a familiar problem. It's also an old problem. Surely if we can solve so many other complex problems, we can do something about this too.

Solution # 1: A smarter compiler

Well, an ideal solution would be a compiler that not only checks that my code is syntactically correct, but that my logic is correct as well. Not asking for much, I know. As impressed as I am by the boffins in Scotts Valley, I'm pretty sure this one isn't going to pop up in the next version of Delphi.

But, pretend for a moment that you had this right now. That your compiler would give you errors when what you wrote wasn't what you meant to write. That it would raise a flag when your business logic was incorrect, or when you changed something that broke some other dependant piece of code. OK, everyone pretending?

How would this change the way you wrote code? Well for starters, as soon as you wrote some incorrect logic, you'd be alerted to the fact. No more having to remember what you were thinking last month when you coded this particular method. You'd know about it straight away, while all of your convoluted logic was still fresh in your mind. Also, the scope of the areas you'd need to look at would be reduced. If your code passed the logic-compiler 5 minutes ago, but now it's raising errors, you can be pretty sure it's something you did in the last five minutes.

In addition, you'd probably be much braver in the way you attack your code. No more would you hear the phrase "It's not broken so why fix it?". We'd all suddenly become Indiana Jones, wading through our code, refactoring with a machete, cutting out great swathes of code and reimplementing more elegantly and concisely, confident that if we've broken anything that something else depends on, we'll know about it as soon as we compile.

I think I'd better calm down. I can hear you all muttering "Indiana Jones? What the hell is this guy on about?" but this compiler is what I want. No, I haven't got it. But I think I have the next best thing.

Solution # 2: Automated Unit Testing

Once we accept the fact that we don't have the compiler outlined above, we need to look at how we can get as many of the same benefits as possible. One method which has been

gaining in popularity in the Java and Smalltalk worlds for a few years is Automated Unit Testing.

Before we get too far into Unit Testing, it's worth looking at what it isn't. Unit Testing is not Black Box testing. Ideally, the tests should be written by the same developer who writes the class being tested. Typically, tests will include some that are designed to workout the class being tested based on knowledge of the internal implementation. This is by no means a requirement, but some people find it beneficial.

Also, Unit Testing is not Functional Testing. We are not testing that the software conforms with the Functional Specification, although it's certainly possible to do some functional testing this way.

What Unit Testing is, however, is a way to define a set of tests for a class in your system. You test the programmatic interface of your object. You put values into properties, call methods, and test that the results you receive are those you expect to receive. So, just as you may have in the past written a simple Delphi application that lets you plug values into a class you've written and display the results, your test case is simply an object that performs those same tasks. But instead of displaying the results for a human to confirm, we write code to automatically confirm that the result is what was expected.

A few years ago, Kent Beck and Erich Gamma released a framework for Automated Unit Testing in Java called JUnit. JUnit has gone on to inspire other Unit Testing frameworks for various languages, including a number for Delphi. Probably the most popular implementation for Delphi is DUnit, an open source, extremely faithful clone of JUnit.

I should say that DUnit, while not terribly complex or difficult to use, is fairly flexible, and so can be used in a lot of different ways. We can’t hope to cover all it’s features, or all the things you could possibly do with it, in a single paper. My aim is to get you familiar enough with the framework so you can start exploring, and give you a few pointers about how to get the most benefit from automated unit testing.

Tests

Perhaps not surprisingly, DUnit is built around the concept of tests. Think about what makes up a test. You need something to test (called a fixture), an action to perform on the fixture, and you need to check whether the results of the action are what you expected.

Let’s think of a simple example. If we have an class called TCustomer, which has properties called Balance and DiscountLevel. DiscountLevel changes based on the current Balance (a Balance of less than $10,000 gives a Discount of 5%, a Balance greater than or equal to $10,000 returns a Discount of based on a more complex calculation involving other properties of the TCustomer).

To test this in DUnit, we could create a TTestCase descendant called TCustomerTestCase. We need something to test (i.e. a fixture), so we’ll declare a private variable in our TCustomerTestCase class of type TCustomer.

Ok, so we have our fixture, next we need a test. In the published section of our TCustomerTestCase class, let’s declare a procedure called TestLowDiscount. In this procedure we’ll set the Balance to a value less than $10,000, and then check that the Discount value is what we expect (i.e. 5%). How do we check this? Well, TTestCase defines a generic Check method, which takes a Boolean parameter, and optionally a message. This

method is kind of like Assert. If the expression passed in the Boolean parameter is True, the check passes, if false, the check has failed and it raises an exception with the optional message.

Similarly, we can declare a few more published procedures to test other aspects of the Discount behaviour, maybe TestHighDiscount, TestZeroBalanceDiscount and TestBoundaryDiscount.

We still haven't created our fixture. The obvious place to do this is in the constructor and destructor of our TCustomerTestCase, but this may not be the best place to do it. Our published Test* methods are going to be called one after the other, and before each is called we want to make sure our fixture is in a consistent state. However, our constructor will only be able to ensure the consistent state of the fixture for the first Test* method. So, TTestCase supplies some virtual methods, Setup and Teardown, which are called before and after (respectively) each Test method. So, in our overridden Setup method, we'll create an instance of our TCustomer and initialise it to reference a particular customer. In our overridden Teardown, we'll destroy it. We can then be fairly sure that every Test method will start with a fixture in a known state.

So we've defined our tests, and ensured that they'll have a consistent starting point. How do we actually run them? Well, we need to register our TTestCase descendant with the DUnit framework, by adding the following line to the initialisation section of its unit:

```
RegisterTest(TCustomerTestCase.Suite);
```

Now that the framework knows about our test case, we need to tell it to run it. Go to the project file for the project which contains our TCustomerTestCase and replace the line:

```
Application.Run;
```

with this line:

```
GUITestRunner.RunRegisteredTests;
```

which kicks off all the test cases which have been registered. In this case, a form will appear and iterate through all the tests, executing each and displaying the results (green for success, pink for failure, and red if any unexpected exception occur). A progress bar will also display the overall results of all tests run (i.e. green only if all tests succeeded, etc). Beware, you also have the option of running your tests as a console application, very handy for including them as part of an automated build system.

We'll look at a few of the other pieces of the framework shortly, but we already have a lot of the pieces we need. We can define tests, run them singly or in multiples, and immediately know if any of the tests fail. Once we fix any failures, and we can successfully run all the tests, we're ready to modify our class, running the tests after each change. If any tests suddenly fail, we can be pretty sure that whatever we just changed caused the problem.

So the developer who is implementing the TCustomer class in this example, can run his text case(s) simply by pressing F9, and know almost immediately whether anything he has added since the last run has broken any of the Test procedures. Also, this developer's test case can be added to the library of test cases for the entire project. This could then be run after every build to ensure that the developers tests have passed, before the build goes to QA, or even as part of an automated build system that does all of this automatically.

TestSuites

If you've stopped reading and gone off to play with this stuff, you might have noticed that the call to RegisterTest doesn't take a parameter of type TTestCase, but rather a parameter of type ITest, an interface. In turn, TTestCase implements ITest.

Why? Well, if all our testing framework knows about is an ITest interface, we can get it to execute things apart from TTestCase descendants. Such as? Well, TTestSuite is another class that implements ITest. A TTestSuite allows you to group a number of objects which implement ITest together, then act on them as a single item (eg. Enable or disable all the ITest implementers as a single group). It's extremely simple to use, you simply create an instance of a TTestSuite object, then add tests to it by calling the AddTest method of the TTestSuite. You can then pass the TTestSuite object into the call to RegisterTest, instead of the TTestCase object.

Just as you can add TTestCase objects to a TTestSuite, you can also add other TTestSuite objects into a TTestSuite, thereby building up a hierarchy of tests.

TestExtensions

In the beginning (no, this isn't about to become a sermon), this was about as far as DUnit went. Certainly extremely useful, but not without areas which could be improved. Thankfully, DUnit is open source, so there are a bunch of people out there improving it and returning their improvements to the community. Some of these improvements take the form of test extensions, and solve some really common problems.

Let's look at the first problem. Let's say in order to test your class, you also need a connection to a database. What we've talked about so far would probably indicate that you should connect to the database in your test cases Setup method and disconnect in the TearDown method. While this will work, this is going to be quite inefficient, with a connection being created before every Test method is run, dropped after each finished, only to be created again almost straight away. Sometime, you want to be able to setup some resources required for your tests once, have them survive across numerous tests or even numerous test suites, and then be cleaned up when you are finished.

This is where the TTestSetup comes in. We can create a descendant of TTestSetup, override it's Setup and TearDown methods to initialise and finalise whatever long-lived test resources we might want, and then create an instance of our TTestSetup, passing in a TTestCase\TTestSuite object to the constructor. The Setup method will be called, then the tests defined in the TTestCase\TTestSuite passed to the constructor will be run, then our TTestSetup object's TearDown method will be called. We can then register our TTestSetup object with the framework, rather than our TTestCase\TTestSuite object.

Another problem might be that you wish to execute a TTestCase\TTestSuite a number of times. The old way to achieve this was to register your TTestSuite repeatedly with the framework. However, a much easier way to achieve this is to use the TRepeatedTest class. Create an instance of TRepeatedTest, passing the TTestCase\TTestSuite object you wish to repeatedly test, and an Integer representing the number of times you wish to repeat it, to the constructor. Registering the TRepeatedTest instance with the framework, rather than the TTestCase, will cause it to be executed the number of times specified.

In case you're interested, both TRepeatedTest and TTestSetup are examples of the Decorator pattern, a way to "attach" extra functionality to an object without changing it's code. It's an extremely useful pattern, and in some way a less sophisticated precursor to Aspect Oriented Programming (AOP).

A Testing Process

It's probably worth spending a bit more time on the How and Why of Automated Unit Testing, rather than just the What.

In clients where we have introduced Automated Unit testing, we typically start them out working a certain way. They inevitably adapt the process once they are comfortable, but here's how we get them started.

Whether they have started their project or not, we first make it a "rule" of the Code Reviews that all new code written must have a test case. So, any new classes get a test case written for them, and any existing code that is touched must have a test case developed for it. Now, the first reaction to this is "Wow, this is going to slow down development", and at first it does. But once developers are comfortable writing them, they usually get to a point where writing the test case and the class takes about the same time as writing the class used to.

How can this work? Well, wherever possible we get developers to write their tests **before** they write the class they are testing. As backward as this sounds, it produces some interesting results.

Firstly, it forces developers to think about what the class needs to do, not how to do it. Put another way, it makes them focus on the interface to the class, not the implementation. They develop the test case based on whatever requirements documentation or specs they have. Once the test case is developed, the developer implements the class until the test case runs without error. The critical point here is that it's very easy to know when you are finished. You're finished when your test cases run without failure. No more gold-plating of classes and adding features that you might need one day. You implement what you need to meet the requirements you have and once your test case runs, you move on to the next task. This is where most of the time savings come from.

The other scenario is bug reports. When a bug is reported, the first thing we do is check-out the corresponding test case and write a test procedure that exposes the bug. Then we can work on the class until our test passes. Not only does it simplify bug reproduction, but your library of test cases becomes more rigorous over time.

The criticisms I often hear about Unit testing is that you can't possibly hope to test all scenario's, and if you could, the time taken to write all those tests would not be feasible. Well, until we have our magic Logic Compiler, we can't expect to cover every test scenario, but that doesn't mean we shouldn't test at all. You have to take a risk-based approach to writing your test cases. Cover the tests that you can reasonably expect will pop up, or that will be completely disastrous if they do pop up, then add to your tests over time as bugs you've missed show up. Cover boundary conditions and other obvious cases, then get on with your job. We've found we can dramatically improve our productivity with this approach, without getting bogged down writing page after page of test cases.

Testing GUI's

Usually when discussing unit testing, the issue of testing GUI's raises it's head. The thing to remember is that you can employ any programming technique you like within your test methods to perform your action or to verify the results. So whether than involves sending Windows messages to a form programmatically to emulate mouse movements, clicks, etc, invoking some some of scripting engine, OLE automation, or whatever, provided you can cause the GUI events to occur, and have some way of verifying the results, it's technically possible to test GUI's with DUnit.

However, personally I think you should be asking yourself whether you should. Plenty of smart people don't agree with me, but I think that there are more efficient, reliable and appropriate tools available to test your application's GUI, and that using DUnit in it's current state to do this usually comes off smelling a little bit like a hack. You either end up writing copious amounts of brittle, complex and hard to maintain code to perform your test, or you end up altering your fixture object's definition to include methods to test the GUI. In the former case, you often then need to test the test or at leats spend an inordinate amount of time writing the test, and in the latter you often end up with methods and properties in your objects interface which probably shouldn't be there.

My advice is, until someone comes up with a way to write brief, maintainable and stable GUI testing code in DUnit, focus on using DUnit in the areas which give you the most bang for your buck, and look to dedicated GUI testing tools such as AQTest or SleuthQASuite to do your GUI testing. Of course, this is just my opinion, and you've no doubt heard the saying comparing opinions to a particular body part, so do whatever you like.

Conclusion

I don't expect to convince you of the benefits of Automated Unit Testing in this paper. I didn't think much of the idea when I first heard it. It wasn't until I was reading "Refactoring" by Martin Fowler, someone who I'd already decided was a pretty smart bloke after "UML Distilled" and "Analysis Patterns", and who basically says that Unit Testing is an essential part of Refactoring, and building reliable code in general, that I was finally convinced to give it a shot. After a couple of days of gradually getting comfortable, I was a convert. It becomes a very natural and intuitive way to work, so at least give it a try and decide for yourselves.

You can find more info at:

http://www.extremeprogramming.org

http://www.junit.org

http://dunit.sourceforge.net/

http://www.martinfowler.com/articles.html

and the latest version of this paper, along with the code samples, is available at http://www.madrigal.com.au