Unit testing - test driven development.

Michaël Van Canneyt

May 25, 2014

Abstract

Testing new code is a natural thing, done by every programmer. Testing code in a structured manner is also frequently found. Test driven development is taking structured testing to the extreme: An introduction to one of the possibilities for an Object Pascal programmer: DUnit.

1 Introduction

As a software engineer, it is (hopefully) natural to test your code before shipping your software. There are many ways to test code. Manually, through some test program, or using a test framework. Test-Driven Development (TDD) takes this last approach even one step further: the test code is written parallel with (or even before) the actual code to be tested.

The use of test frameworks became well-established starting with the junit test framework for Java: Extremely simple in its design, it is nevertheless very powerful.

Creating unit tests in a test framework offers several advantages:

- The output of the tests is uniform.
- The tests themselves are written largely in the same manner. Test code written in a test framework is recognisable for any programmer familiar with the framework.
- It is easily automated.

The fact that the output is uniform makes it suitable for storing. Using some automation, it is then possible to do automated regression testing, and it becomes doable to do continuous delivery.

Unit testing does not mean testing a "unit" in the Delphi sense. The "unit" in "unit testing" is a single unit of functionality: this can be a method, a procedure or function, but can also be a class or a Delphi unit. Which of these possibilities is chosen depends on the tester.

Dunit is a test framework resembling the junit testing framework, it is shipped with at least Delphi XE and higher. DUnit2 is another test framework for Delphi. DUnit2 can work with Free Pascal and Lazarus. The latter ship with fpcunit, which is to a large degree compatible to DUnit.

In this article the use of DUnit, as distributed with Delphi, is demonstrated and explained.

2 What to test

Normally, unit tests test a single unit of functionality: the test checks whether the functionality behaves as expected, i.e. obeys the 'API contract'. This usually means constructing several kinds of input for an algorithm , and checking whether the output is what is expected. Here input can take various forms: for a function this may be a combination of arguments. It can also be a complete XML or JSON document to be processed by a component, and so on.

Typically, all functions in a unit or all public methods of a class will be tested.

Testing also includes checking for error conditions, for example: checking that taking the square root of a negative number results in an error; or checking how a routine that expects XML behaves in case it gets JSON as input. Checking for errors becomes doubly necessary if the API specifies a series of error codes: in that case the testsuite should check that the proper error code is returned.

Some people take test-driven development to the extreme by first developing the test code, watching it fail, and then implement the actual code till it the tests succeed. The usefullness of way of working is debatable: the interface (contract) may not yet be known in advance; it also supposes that the test code is written correctly. Also, a test may fail because the test code is written wrongly...

Although unit testing of a GUI is possible, it is somewhat awkward; it is difficult to simulate all possible actions of a user. It is also difficult to integrate in an automated build system, which is why unit testing is usually reserved for business logic code - which should normally be separate from GUI logic, a requirement that requires some discipline when working in a RAD environment such as Delphi.

3 Getting started

To demonstrate how DUnit works, a new fictitious component (`TMyComponent`) is created in a new unit (`mycomponent`), in a new package (`newpackage`). The package is in a project group. Part of the component code has been written, and now it is time to test the implementation. So, “Add new project” is chosen, and under “Unit test” the entry ’Test Project’ (figure 1 on page 3)is chosen.

Upon choosing this, the new test project dialog in figure 2 on page 3 is shown. The dialog is self-explaining. The first option ’Source Project’ specifies the project to be tested. This option is important, it because it determines what the other dialog in the dunit package will show when invoked.

The next part of the dialog asks whether the test application should be a GUI or a command-line application. This setting can be changed later at any time. The command-line application is suitable e.g. for integration in a build process or nightly regression testing.

Once the wizard is completed, a new project has been added to the project group. Let’s look at the project source:

```
program mypackageTests;

{$IFDEF CONSOLE_TESTRUNNER}
{$APPTYPE CONSOLE}
{$ENDIF}

uses
  DUnitTestRunner;

{$R *.RES}

begin
```

Figure 1: Starting a new test project

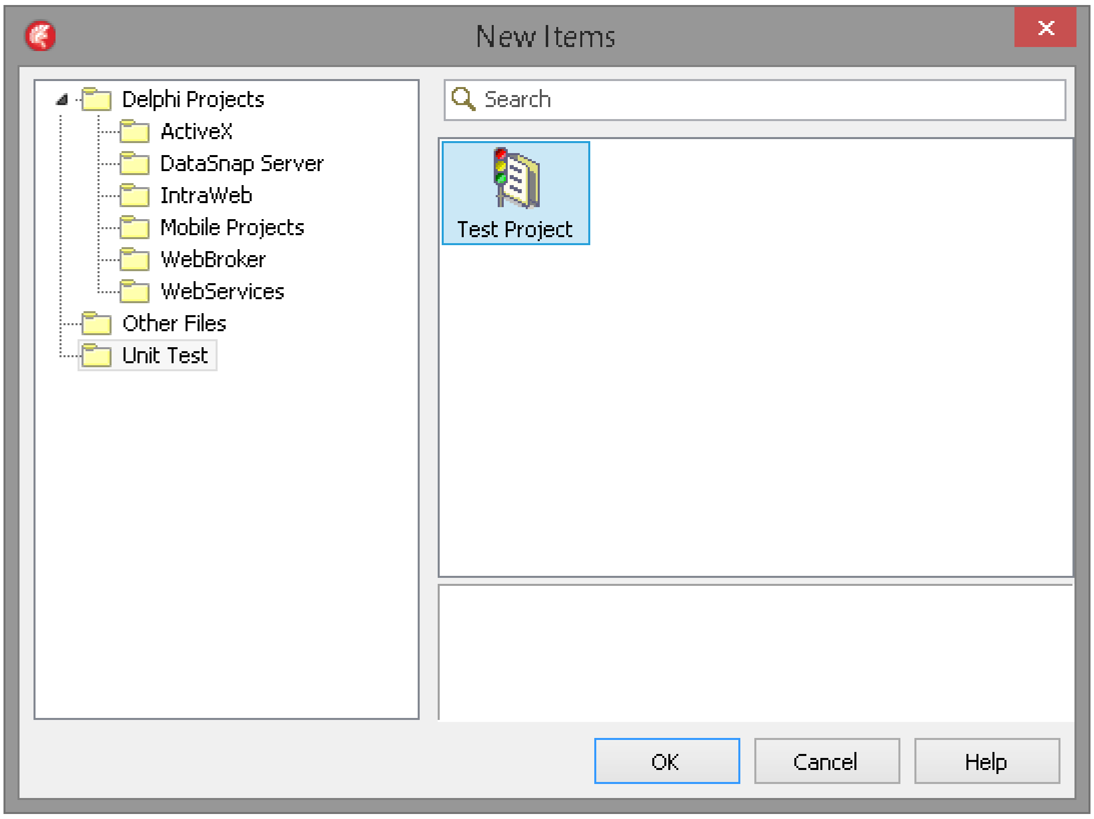
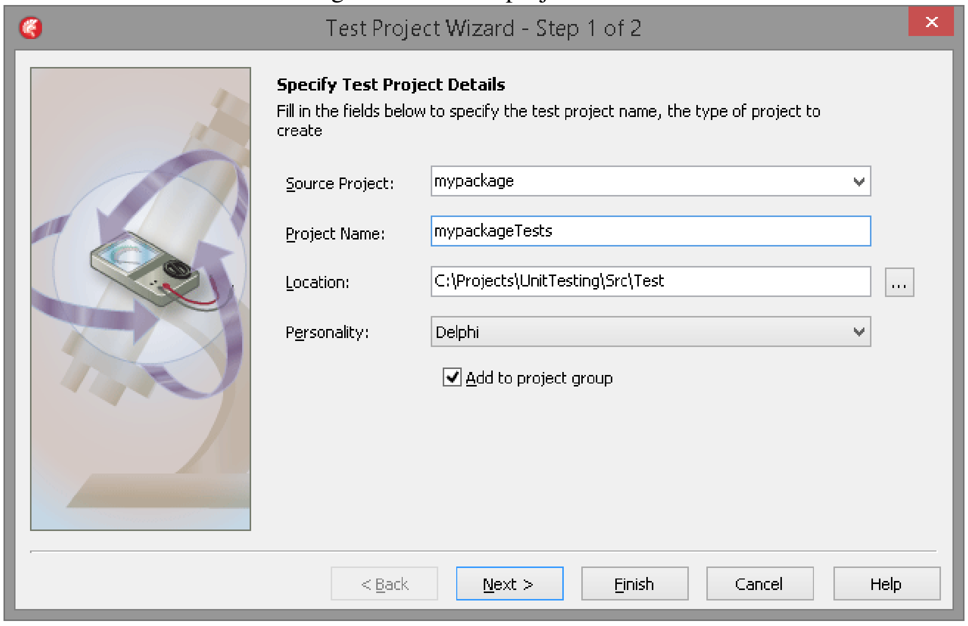


Figure 2: The test project wizard


Test Project Wizard - Step 1 of 2

Specify Test Project Details
Fill in the fields below to specify the test project name, the type of project to create

Source Project: mypackage

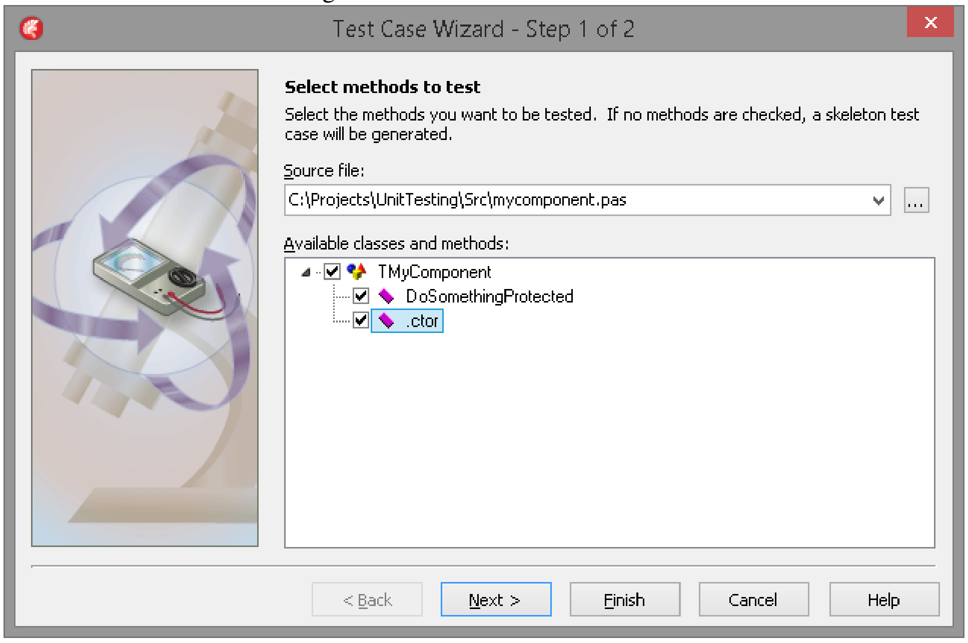
Project Name: mypackageTests

Location: C:\Projects\UnitTesting\Src\Test

Personality: Delphi

[✓] Add to project group

Figure 3: The new test unit wizard



Select methods to test

Select the methods you want to be tested. If no methods are checked, a skeleton test case will be generated.

Source file:

C:\Projects\UnitTesting\Src\mycomponent.pas

Available classes and methods:

- [✓] TMyComponent
  - [✓] DoSomethingProtected
  - [✓] .ctor

```
  DUnitTestRunner.RunRegisteredTests;
end.
```

The call to ’RunRegisteredTests’ reveals that there must be tests registered somewhere: DUnit works with a registry (a list, or actually a tree) of tests.

When the application is run, first all tests must be registered with the test framework. When this is done, the test framework will then run the tests one by one, and collect the test results.

Registering a test with the test framework is normally done in the initialization section of the unit that implements the test, although this is not a requirement. All that is needed is that the test is registered before `RunRegisteredTests` is called.

After creating the test project, no tests of test code exist yet. To add test code, a new unit must be created: From the “File-New-Other” dialog, under “Unit test”, the “Test Case” item must be chosen. This will pop up the ’Test case wizard’ dialog.

This dialog will present a list of all the units found in the “source project” that was specified when the test project was started. When selecting a unit, a list of classes, with their methods is presented, as seen in figure 3 on page 4.

Placing a checkmark in front of all methods that must be tested will create an empty test-method for that method. (the `.ctor` stands for the constructor of the class). When the dialog is done, the wizard generates a new unit with a skeleton test case class declaration. For the case of the `TMyComponent`, this results in the following class:

```
TestTMyComponent = class(TTestCase)
  strict private
    FMyComponent: TMyComponent;
  public
    procedure SetUp; override;
    procedure TearDown; override;
  published
    procedure TestDoSomethingProtected;
    procedure Test.ctor;
  end;
```

DUnit Test cases are organized in test classes, which descend from `TTestCase`. All published procedures are considered tests: the test framework will instantiate the test class at one time or another, and will then call the various published procedures one by one, in the order that they are declared. These procedures cannot have arguments, and should not be functions (i.e. they can not return a result).

At the end of the unit, the following code can be found:

```
initialization
  RegisterTest(TestTMyComponent.Suite);
```

This code registers the test class as a test suite in the DUnit test registry. This registration can be moved to a central routine, or changed in any other way, as long as the test is registered before the call to `RunRegisteredTests`. The registration is done as a testsuite using the standard `Suite` method of the `TTestCase` class, because the class contains actually multiple tests: one per published method, and hence the class forms a testsuite in itself.

Several things can be noted about the code created by the wizard.

1. It declares a field of the type `TMyComponent`, the class that will be tested.
2. It declares 2 standard methods: `SetUp` and `TearDown`. These methods are called before and after each test. They are used to instantiate the `TMyComponent` instance.
3. It declares a published procedure `TestNNN` for each method that was selected in the wizard.
4. The generated code will not compile: Test.ctor is not a valid method name.
5. No regular functions or procedures (functions/procedures that are not methods) can be selected to generate a test case for.
6. Properties cannot be selected either.
7. It allows you to select protected methods, but these are obviously not directly callable from a testcase.

The wizard generates skeleton code, this means it may require some clean-up before it is compilable, depending on the options that were chosen in the dialog.

The generated `Setup` and `Teardown` methods look as follows:

```
procedure TestTMyComponent.SetUp;
begin
  FMyComponent := TMyComponent.Create;
end;

procedure TestTMyComponent.TearDown;
begin
  FMyComponent.Free;
  FMyComponent := nil;
end;
```

The above will not compile, because the `TComponent` constructor requires an `Owner` argument. This is easily corrected. But the code gives a clue as to the meaning of these methods:

- `Setup` is called before each test method, and should set up initial conditions for the test. All tests will test some functionality of `TMyComponent`, so it is logical that `Setup` sets up an instance of the component.
- Similarly, `TearDown` will clean up after the test: it frees the component from memory.

An important aspect of unit testing is that each test must be isolated: it should start with a clean slate. It should not be influenced by the results of other tests, nor should a test require another test to be run first. That is why the `Setup` and `Teardown` methods should always result in a clean, new, situation for each test.

The constructor or destructor of the testcase class itself must not be used to set up the initial situation: a test framework is free to choose whether it will instantiate the test class once for each test method, or once for all test methods. The only guaranteed thing is that the `Setup` and `Teardown` methods are called before and after the test, respectively.

4 Implementing test code

Now that the testcase class is implemented, it is time to write actual test code. The `.ctor` test generated by the wizard is renamed to `TestConstructor`, and filled with the following code:

```
procedure TestTMyComponent.TestConstructor;
begin
  CheckNotNull(FMyComponent.MyStrings,
               'Strings property correctly initialized');
  CheckEquals(0,FMyComponent.MyStrings.Count,'Strings empty.');
end;
```

This code speaks almost for itself. The first line checks that the `MyStrings` property of `TMyComponent` is initialized with a `TStrings` instance. The second line checks that the number of strings in the list is zero. The `CheckNotNull` and `CheckEquals` methods are part of `TAbstractTest`, an ancestor of `TTestCase`. They are declared as follows:

```
procedure CheckNotNull(obj: TObject; msg: string = '');
procedure CheckEquals(expected, actual: integer; msg: string = '');
```

Both methods will test (check) a condition: In the case of `CheckNotNull` this means checking that `Obj` is not equal to `Nil`, and for `CheckEquals` it means verifying that the `actual` value is equal to the `expected` value. Both methods will raise an `ETestFailure` exception if the condition is not met; The failure error message can be specified as the last option.

Overloaded versions `CheckEquals` exists for almost all basic pascal types, just as a `CheckNotEquals` method exists to verify that 2 values are not equal. And similarly to `CheckNotNull`, a `CheckNull` method exists. To compare object or interface instances, instead of `CheckEquals`, `CheckSame` should be used:

```
procedure CheckSame(expected, actual: IUnknown; msg: string = '');
procedure CheckSame(expected, actual: TObject; msg: string = '');
```

Checking whether an instance is of a certain class can be done easily with:

```
procedure CheckIs(AObject: TObject; AClass: TClass; msg: string = '');
```

Some more check methods exist, but the above is the bulk of the methods that will be used in testing code.

Since these testing methods raise an exception, this means that the test method will stop at the first error: One one hand, this is logical since the rest of the method may rely on the fact that the check passed. In the case of the above method, test line line 2 would result in an access violation if the check in the first line didn't raise an exception.

Is the above a useful test, since clearly the component will never function if it is not correctly initialized? This is a matter of opinion, and depends how far one wishes to go in testing. There is one case when testing the constructor is definitely useful: if default values are specified for published properties, it may be a good idea to check that the constructor actually sets the value as declared in the property.

Ultimately, all `Check` methods call the `Fail` method to raise the `ETestFailure` error message. This method is also available in case a test requires a more elaborate test condition, which is not easily expressed as an (in)equality.

```
procedure TestTMyComponent.TestSomethingDifficult;
begin
  If Not SomeVeryDiffcultTestCondition then
    Fail('This test failed, condition not met');
  // More test code here.
end;
```

5 Running the tests

To give some more body to the tests, a public `Clear` method is added to the component, which is supposed to clear the strings, and do some other clearing as well. An additional test is written for this method:

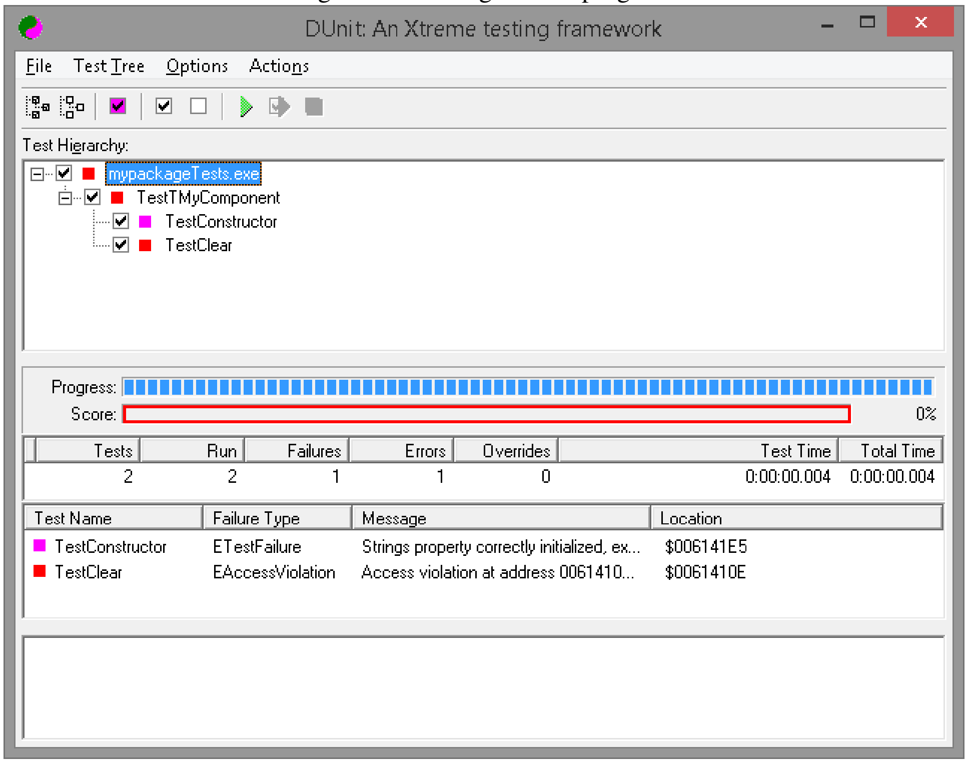
```
procedure TestTMyComponent.TestClear;
begin
  FMyComponent.MyStrings.Add('a string');
  FMyComponent.Clear;
  CheckEquals(0,FMyComponent.MyStrings.Count,'Strings empty after Clear.');
end;
```

Now 2 test methods exist, at least one of which actually tests behaviour, and the test program can be run.

The test runner program shows all available tests in a tree-like structure, each test can be individually selected or deselected: the toolbar contains buttons to select/deselect all tests or just the selected test or testsuite. It also contains a button to select all failed tests - more about this later.

Running the tests using the 'Run Selected Tests' button results in a screen as shown in figure 4 on page 8. The first progress bars will indicate how many tests have been run, and

Figure 4: Running the test program



```
mypackageTests.exe
  TestTMyComponent
    TestConstructor
    TestClear
```

0%

Tests	Run	Failures	Errors	Overrides	Test Time	Total Time
2	2	1	1	0	0:00:00.004	0:00:00.004

Test Name	Failure Type	Message	Location
TestConstructor	ETestFailure	Strings property correctly initialized, ex...	$006141E5
TestClear	EAccessViolation	Access violation at address 0061410...	$0061410E

the second indicates the success/failure rate. The small grid below the progress bars shows a small statistic, and the grid below that one shows failure or error messages.

The test run shows a score of zero: the first test failed, and the second resulted in an error.

There are 3 possible outcomes for a test:

Success The test completed successfully.

Failed The test failed: a condition was not met, and a `ETestFailure` exception was raised.

Error An error occurred during execution of the test. This is not the same as a test failure: any exception (different from `ETestFailure`) that occurs during execution of the test is reported as an error.

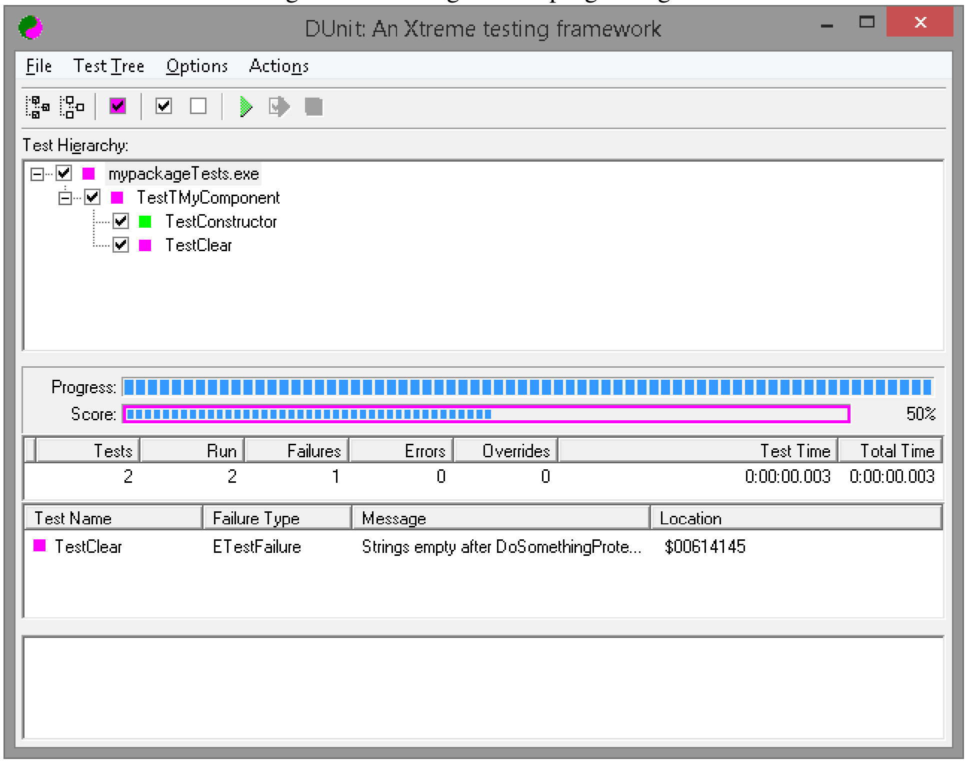
Clicking on a test or an error in the grid with failures and errors, will show some more detail about the failure in the bottom area of the test screen.

That the second test gives an error is normal: it relies on the fact that the `MyStrings` property is properly initialized. The result of the `TestConstructor` test shows that it is not properly initialized. The solution is to adapt the constructor of the component:

```
constructor TMyComponent.Create(AOwner: TComponent);
begin
  inherited;
  FMyStrings:=TStringList.Create;
end;
```

Running the test program again will now show (figure 5 on page 9) that the first test succeeds, and the second fails. Before fixing this, it is a good idea to push the 'Select failed

Figure 5: Running the test program again



mypackageTests.exe
  TestTMyComponent
    TestConstructor
    TestClear

Score: 50%

Tests	Run	Failures	Errors	Overrides	Test Time	Total Time
2	2	1	0	0	0:00:00.003	0:00:00.003

Test Name	Failure Type	Message	Location
TestClear	ETestFailure	Strings empty after DoSomethingProte...	$00614145

tests' button. This will deselect all succeeded tests, leaving only failed tests selected.

The selection of tests is saved across runs of the test program. This means that at the next run of the test program, only the failed test will be selected. When the 'Run selected tests' button is pressed, only tests that previously failed will be run. In some cases this can drastically reduce the test run time.

6 Testing for exceptions

Handling of error conditions form part of the contract (API) of a component. Thus, they must also be tested. Assume the `TMyComponent` call has a `SetCapacity` call that accepts only even arguments. This can be coded as follows:

```
procedure TMyComponent.SetCapacity(const ACapacity: Integer);
begin
  If (ACapacity mod 2)<>0 then
    Raise EMyException.CreateFmt('Capacity %d must be even.',[ACapacity]);
  FCurrentCapacity:=ACapacity;
end;
```

To test this, the following code can be used:

```
procedure TestTMyComponent.TestSetCapacity;

Var
  EC : TClass;
```

```
begin
  // This should work
  FMyComponent.SetCapacity(100);
  CheckEquals(100,FMyComponent.CurrentCapacity,
              '100 is ok');
  EC:=Nil;
  Try
    // This should not work.
    FMyComponent.SetCapacity(99);
  except
    On E : Exception do
      EC:=E.ClassType;
  End;
  If EC=Nil then
    Fail('Expected EMyException, but no exception was raised');
  CheckEquals(EMyException,EC,
              'Exception was raised, but of wrong class.');
end;
```

This becomes awkward to write if a lot of errors must be tested. Fortunately, the same can be achieved with the `CheckException` method;

```
procedure TestTMyComponent.SetOddCapacity;

begin
  FMyComponent.SetCapacity(99);
end;

procedure TestTMyComponent.TestSetCapacity2;

begin
  FMyComponent.SetCapacity(100);
  CheckEquals(100,FMyComponent.CurrentCapacity,'100 is ok');
  CheckException(SetOddCapacity,EMyException,
                 'setting odd capacity results in exception');
end;
```

The `CheckException` method expects a routine that must throw an exception, and the class of this exception. Optionally, an error message may be specified. The routine will be run, and the test will fail in 2 cases:

1. if no exception was raised.
2. when the exception class differs from the expected exception class.

In this case, the routine to run is a method of the test class. This need not be so: a method of the component that is tested can also be called directly, provided it has the correct signature.

7 Testsuites

Each `TTestCase` class is a testsuite in itself. When registered, it appears as a top-level node in the test tree, with all the published methods as leaf nodes below the top-level node.

But what if you want to add more structure to the test tree ? For instance, for a database application, you may want to separate the tests that require database access from the tests that do not need database access, and create 2 trees.

This means that for instance you can run the tests that do not require database access without connecting to the database.

There are several ways to do this. Imagine a test case

```
  TTestMyDBComponent = Class(TTestCase)
  published
    procedure TestRead;
    procedure TestCreate;
    procedure TestUpdate;
    procedure TestDelete;
  end;
```

Then this can be registered in a database testsuite as follows:

```
initialization
  RegisterTest('Database tests',TTestMyDBComponent.Suite);
```

This will create a testsuite called 'Database Tests' if it doesn't exist yet, and will register the `TTestMyDBComponent` testcase in that testsuite. A hierarachy of suites can be created by putting dots in the test name:

```
initialization
  RegisterTest('Database.Local',TTestMyLocalDBComponent.Suite);
  RegisterTest('Database.Remote',TTestMyRemoteDBComponent.Suite);
```

This will create a test suite called 'Database', with 2 sub-suites below it: 'Local' and 'Remote'. Instead of the dot (.) character, a slash or backslash character can also be used to separate the various levels.

An alternative is to create the testsuite manually:

```
Var
  DBTestSuite : TTestSuite;

initialization
  DBTestSuite:=TTestSuite.Create('Database tests');
  DBTestSuite.AddTest(TTestMyDBComponent.Suite);
  RegisterTest(DBTestSuite);
```

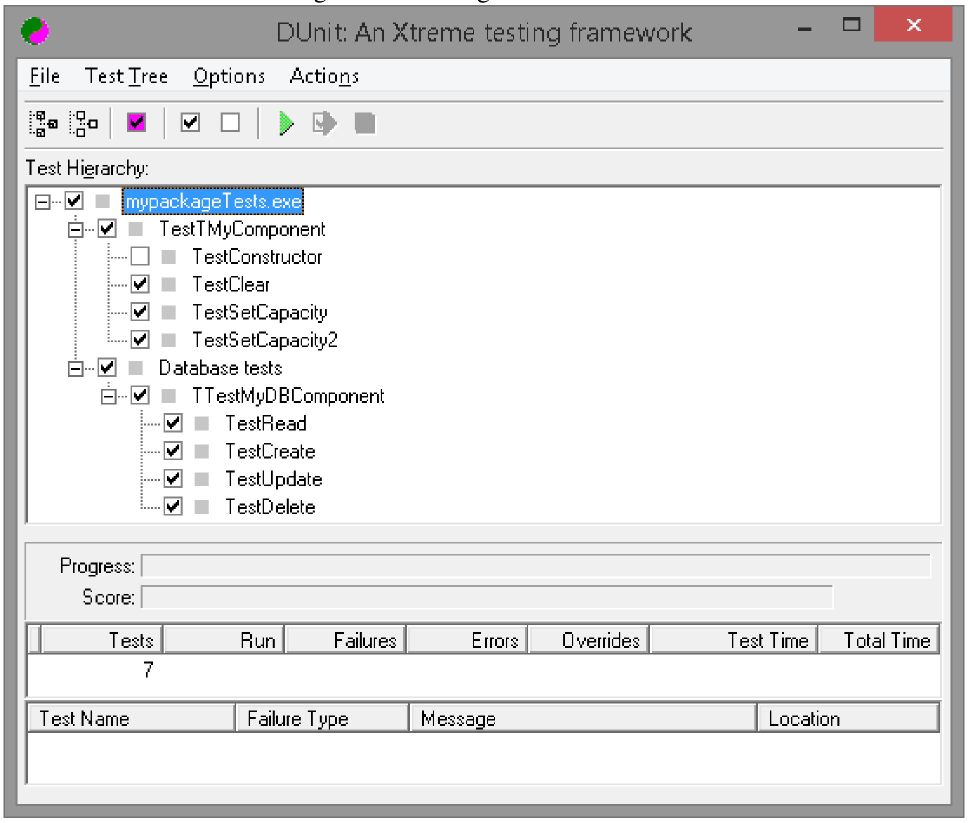
Using the manual way, quite complex test trees can be established. There is a shorthand for this construction:

```
Var
   DBTestSuite : TTestSuite;

initialization
  RegisterTest(TestTMyComponent.Suite);
  DBTestSuite:=TTestSuite.Create('Database tests',[
                               TTestMyDBComponent.Suite
                                ]);
  RegisterTest(DBTestSuite);
```

Whatever the used method, the result is always the same, and is shown in figure figure 6 on page 12.

Figure 6: Dividing tests in testsuites



```
mypackageTests.exe
  TestTMyComponent
    TestConstructor
    TestClear
    TestSetCapacity
    TestSetCapacity2
  Database tests
    TTestMyDBComponent
      TestRead
      TestCreate
      TestUpdate
      TestDelete
```

Tests	Run	Failures	Errors	Overrides	Test Time	Total Time
7						

Test Name	Failure Type	Message	Location

8 One time Setup/Teardown: decorators

There is another reason for using testsuites: the `Setup` and `TearDown` methods are called to create the initial situation for each test. For a database test this would mean connecting to and disconnecting from the database. Obviously, this would adversely affect performance. One could suppose that the constructor of the test case is the place to do so, but since the testsuite does not guarantee how a testcase is instantiated, this is not a good solution. A way is needed to make sure a routine is run once per test(suite).

There are 2 ways to solve this problem. In fact, they amount to the same thing. The first is to use a test decorator. The test decorator can be used to 'Decorate' a test, i.e. it is to be executed whenever a test is executed.

The `TTestDecorator` class in the `TestExtensions` unit does just that. To use it, the `RunTest` method of the test decorator must be overridden:

```
TMyDBTestDecorator = Class(TTestDecorator)
 Procedure RunTest(ATestResult: TTestResult); override;
end;
```

The method `RunTest` then just needs to implement the connect/disconnect:

```
Procedure TMyDBTestDecorator.RunTest(ATestResult: TTestResult);

begin
  ConnectToDatabase;
  try
    Inherited;
  finally
    DisconnectFromDatabase;
  end;
end;
```

And to use this class, instead of registering the test, the test decorator is registered:

```
RegisterTest(TMyDBTestDecorator.Create(TTestMyDBComponent.Suite));
```

A test decorator can contain only one test, but an instance can be created for each test that needs this decoration.

Obviously, the test to be decorated can also be a test suite. So to make sure that the database connection is made only once for all Database related tests, the following test setup can be used.

```
Var
   DBTestSuite : TTestSuite;

initialization
  RegisterTest(TestTMyComponent.Suite);
  DBTestSuite:=TTestSuite.Create('Database tests');
  DBTestSuite.AddTest(TTestMyDBComponent.Suite);
  RegisterTest(TMyDBTestDecorator.Create(DBTestSuite));
```

In this case, a simpler way is to create a descendant of `TTestSuite` and override the testsuite's `RunTest` method:

```
  TDBTestSuite = Class(TTestSuite)
    Procedure ConnectToDatabase;
    Procedure DisconnectFromDatabase;
    Procedure RunTest(ATestResult: TTestResult); override;
  End;
```

The code for the RunTest method is the same. The code for registering the tests is somewhat simpler:

```
Var
   DBTestSuite : TDBTestSuite;

initialization
  RegisterTest(TestTMyComponent.Suite);
  DBTestSuite:=TDBTestSuite.Create('Database tests');
  DBTestSuite.AddTest(TTestMyDBComponent.Suite);
  RegisterTest(DBTestSuite);
```

The end result of all these mechanisms is the same; the use of a test decorator will insert an extra level in the tree, giving a visual indication that a test decorator is used.

9 Conclusion

Anno 2014, the ability to use a test framework should be part of the toolbox of every programmer. The DUnit test framework is simple to use, and offers all the functionality required to run automated tests. The purpose of this article was to introduce the topic, showing that it is not hard to do, and has many benefits. Many areas (mock classes and other techniques used in testing) are not explored, but this can be left for a future contribution.