Test your Lazarus/FPC code with FPCUnit

Michaël Van Canneyt

October 9, 2005

Abstract

Testing frameworks for your code are popular: Java has JUnit, Delphi has Dunit and NUnit. Free Pascal and Lazarus could not stay behind, so FPCUnit was implemented. An overview.

Remark: This article was translated to German and published in issue 2/2005 of the Toolbox Magazine (`http://www.toolbox-mag.de/`).

1 Introduction

Lots of people are firm believers in the concept of writing test code for classes or routines, as part of the development effort. This has culminated in a series of test frameworks, similar to the original JUnit test framework.

Basically, the JUnit test framework provides a controlled environment to perform tests of software routines. It does this by providing a few base classes (testcases) of which the tester must make a descendent, and which should include test code for the routines to be tested. These classes are then registered in a test suite registry, and can be run. The output of this run can be analyzed by yet another class. (a 'Listener')

Over time, many implementations of this idea have appeared, for all kinds of programming languages. Also for Delphi (DUnit and NUnit). Since the 1.9.8 version of Free Pascal (and version 0.9.7 beta of Lazarus), such an implementation is also available for Free Pascal: FPCunit.

The implementation was made by Dean Zobec, and is included in the FCL (Free Component Library) shipped with Free Pascal. Vincent Snijders made a nice package to facilitate creating ad running tests for the FPCunit framework in the Lazarus IDE. Both will be discussed here.

2 FPCunit: Units and Classes

The heart of the FPCUnit testing system is implemented in the fpcunit unit. It contains a series of base classes, which make up the test system. The following classes can be found in this unit:

TTest The parent class for `TTestCase` and `TTestSuite`.

EAssertionFailedError This is the basic exception, indicating that a test has failed. It is used by the other classes.

TAssert A class which implements a series of 'Assert' commands.

TTestCase The central class of the fpcunit system: For each set of tests routines, a descendent of `TTestCase` must be written.

TTestResult A class to run a test and store the results of the test run.

TTestSuite A class to register `TTests` with, which can run the tests registered in it. Both `TTestCase` and `TTestSuite` instances can be registered, making the complete collection of tests resemble a Tree-Like structure.

TTestFailure A class to store a failed test result in.

IListener An interface that can be used to be notified of test results.

Of all these classes, only `TTestCase` and `TAssert` are actually important for the tester. All the other classes are needed to register tests, run the tests and communicate results.

The testregistry unit implements a registry for tests using a the `TTestSuite` class. It exposes the `TTestSuite` instance with the `GetTestRegistry` function.

The testreport unit implement a class with the `IListener` interface; A `TXMLResultsWriter` class. This will collect the results of a test run and return a string with the results represented in XML.

The sources of FPCUnit also contain a demo 'consolerunner'. This is a ready-to-run application, which uses the `testregistry` unit to maintain a registry of tests. To make it run some tests, just add the units with the implementation of the tests to the uses clause of the program, compile and run. Obviously, the tests must register themselves with the testregistry in the initialization section of the units.

3 Writing a test case

The central class in `FPCUnit` is `TTestCase`. To implement a set of tests, a descendent of this class must be created. A Test case consists of various procedures, each of which is a test that can be run, independent of the others. When the tests are run, each of these procedures is called automatically, and if an exception is raised during the test, the test is considered failed. If it runs without errors, the test is considered succeeded.

How does the testing framework know which procedures must be called ? Simple, it checks the RTTI information of the class: Only 'published' procedures will be run. These procedures must accept no parameters, and should return no result. In case the test fails, they should simply raise an exception.

So, the most simple test case would be:

```
TMyTestCase = Class(TTestCase)
Published
  Procedure MySillyTest;
end;
```

Which could be implemented as follows:

```
Procedure TMyTestCase.MySillyTest;

begin
  AssertEqual('The compiler cannot count !',2,1+1);
end;
```

The `AssertEqual` method is declared as follows:

```
Class Procedure AssertEquals(const Msg: string;
                             Expected, Actual: integer);
```

What does it do ? It compares `Expected` and `Actual` and raises an `EAssertionFailedError` exception if they are different.

The `AssertEqual` method is overloaded. The `Expected` and `Actual` parameters can be of almost any type: String, Integer, Int64, currency,Double, Boolean, Character, Class, Pointer, Interface and Object.

The message is optional, i.e. overloaded versions without message exist.

There is also a simple `Fail` method:

```
class procedure Fail(const AMessage: String);
```

Which will simply raise an `EAssertionFailedError` exception with the indicated message. This can be used to test try/except statements. For instance, the following method tests the behaviour of `TStringList.Delete` when an attempt is made to delet a non-existing item.

```
procedure TStringListTestCase.TestDeleteNonExistent;

begin
  l.Add(Line1);
  Try
    L.Delete(1);
  except
    Exit;
  end;
  Fail('No exception raised when deleting non-existent item');
end;
```

The above code adds one item to an empty stringlist, and then attempts to delete the second item. Since there is no second item, an exception should be raised by `TStringList`. If the exception is raised, it is caught, and the procedure exits cleanly (test succeeded). If no exception is raised, then the `Fail` statement will be executed, indicating that the test failed.

There are other `Assert` methods to aid in testing, the full list is in the definition of the (soon to be documented) `TAssert` class.

To test the above class, the following unit could be made:

```
unit MyTestCase;

Interface

Uses fpcunit,testregistry;

Type
  TMyTestCase = Class(TTestCase)
  Published
    Procedure MySillyTest;
  end;

Implementation
```

```
Procedure TMyTestCase.MySillyTest;

begin
  AssertEquals('The compiler cannot count !',2,1+1);
end;

Initialization
  RegisterTest(TMyTestCase);
end.
```

To run this test, the unit can be simply added to the uses clause of the `consolerunner` application in the FPC sources. Compiling and running gives the following result:

```
home: >ppc386 -S2 testrunner
home: >./testrunner --all
<testresults>
<testlisting>
<test name="TMyTestCase.MySillyTest">
</test>
</testlisting>
<NumberOfRunnedTests>1</NumberOfRunnedTests>
<NumberOfErrors>0</NumberOfErrors>
<NumberOfFailures>0</NumberOfFailures>
</testresults>
```

The sources accompagnying this article contain some more tests.

4 Housekeeping and Isolation: Setup and TearDown

Now, for real test scenarios, it is likely that a test needs some setting up. For instance in the `TStringList` test above, a stringlist instance is used. When should this stringlist be created and freed ? In the case of a database test: when should the connection be made ? If no special provisions are made, there are only 2 possibilities:

1. In the constructor of the test. However, there is no explicit control over when a `TTestCase` instance is constructed, and no assumptions about it should be made by the programmer of the test.
2. In the test method itself. This would require a lot of duplicate code, even if it was separated out, as probably the same code would have to be executed in each test method:

The latter case would mean that each test would look like this:

```
procedure TStringListTestCase.TestSomething;

begin
  CreateStringlist;
  Try
    DoSomeTest; // Real test code here.
  finally
    FreeStringList;
```

```
  end;
end;
```

This is of course a lot of writing.

No acceptions should be made about the instantiation of the `TTestCase` classes. The following 2 cases, totally opposite, show why using the constructor for resource allocation is a bad idea.

The first case, allocating the stringlist in the constructor, could potentially use an enormous amount of memory: for each method to be tested, a separate instance of the `TTestCase` class could be created and kept in memory during the lifetime of the testsuite. If the stringlist was allocated in the constructor of the testcase, then this would mean that as much stringlists would be allocated as there are test methods. In the case of database tests, this would mean e.g. a huge amount of connections to a database. This is of course not acceptable.

There is a second case, when only 1 instance of the `TTestCase` class is created, and all methods are run one after the other. If the resources are allocated during the creation of the `TTestCase`, this of course would violate an important principle of testing: Isolation. Isolation means that all tests must be able to be run indepenent of each other in any order. Consider the scenario that the stringlist is created in the constructor of the `TTestCase`. If a Stringlist test fails, it could free the stringlist. The next test will then fail - because the stringlist is not there, or because it has the wrong number of items in it. In the case of a database test, a test could cause the database connection to dissapear; The following test would fail, because there is no connection.

Luckily, the designers of the `JUnit` testing framework foresaw this. They added 2 methods virtual methods to `TTestCase`: `Setup` and `TearDown`. These methods are guaranteed to be run before and after each test is run. The actual test method therefore looks roughly as follows:

```
procedure TTestCase.Run;
begin
  SetUp;
  try
    RunTest;
  finally
    TearDown;
  end;
end;
```

So `Setup` is the place to allocate resources, and `Teardown` is the place to free the allocated resources again. In the case of a stringlist test, this means that the implementation of these methods looks as follows:

```
Procedure TStringListTestCase.Setup;
begin
  l:=TStringList.Create;
end;

Procedure TStringListTestCase.Teardown;
begin
  FreeAndNil(l);
end;
```

The field `L` is a field of the `TStringListTestCase` test class.

As it happens, the FPCUnit implementation uses scenario 1 for it's implementation of the unit test framework. But this could be changed in the future.

5 Lazarus support: Installation

There are 2 cooperating packages in the lazarus distribution, which make implementing test cases really easy. To install support for this, the following must be done:

1. Select the menu 'Components', item 'Open package file'.
2. Select the file fpcunittestrunner.lpk. It resides in the directory components/fpcunit under the main Lazarus source directory.
3. Compile the package. Do not install it yet.
4. Using the same menu as before, select the file fpcunitide.lpk, in the directory components/fpcunit/ide.
5. Install this package.

Lazarus should recompile itself, and should restart.

6 Lazarus support: Usage

After the lazarus IDE was compiled, there should be 2 new items under the 'File' - 'New' menu item:

FPCUnitApplication Is a new project type. Selecting this item will create a new project with a GUI form that displays the results of the tests. Additionally, a single unit is created with a TTestCase descendent class, ready to be implemented, and it is registered in the testregister.

FPCUnitTestCase This creates a new unit with a TTestCase descendent.

The new items are shown in figure 1 on page 7.

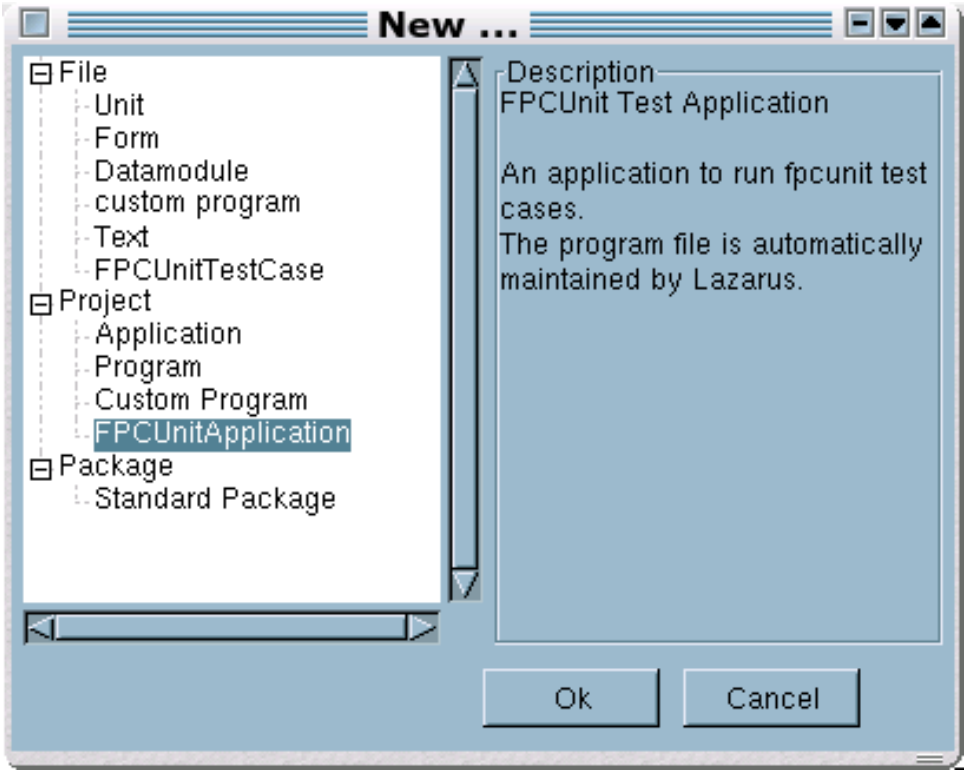
To run existing tests in the lazarus testrunner, it is therefore sufficient to start a new testrunner application, and add the existing units in it. If the test are not registered in these units, then the registration code must still be added. figure 2 on page 8 shows the result of several tests. All registered tests are shown in a tree-like fashion. It is possible to select any node in the tree, all tests in and under that node will be run. When the tests are run, the status of the tests is shown in the tree, and the results are logged. The logged results of the tests can be copied to the clipboard (e.g. for pasting in an email and sending a bug report).

The workings of the GUI form are extremely simple; It implements the IListener interface, and offers a visual way of selecting a test and displaying the test result. The workings are of no importance if your only goal is to write tests, but can be studied if an alternative implementation for a GUI testing application must be made: The code is quite simple and clearly written.

7 Conclusion

The FPCUnit framework is a powerful testing environment. Its inclusion in Lazarus makes it extremely comfortable to use, and people who make a lot of low-level classes

Figure 1: Lazarus new FPCUnit items.



should definitely invest the extra time in designing some tests as they code their classes. The extra time is well-spent, as well-designed tests will show bugs in an early stage. Influence of later changes to the code can be tested by simply re-running the tests. The FPC distribution comes with a lot of demo tests, including a port to pascal of the original JUnit example test. Some of the tests are included in the source code that comes with this article, they can be studied and used as an inspiration for the design of your own testing code. Searching on internet will also result in a huge amount of other test code which can be studied as well.

Figure 2: Lazarus GUI test runner.