Unit Testing and TestComplete 2.0: One Possible Framework

TestComplete White Paper Series - Volume II, Number I - 7/3/2003.

Copyright ©2003 by Robert K. Leahey and AutomatedQA, Corp. All rights reserved.

Table of Contents

Purpose of this Paper 1

Introduction 3

Overview 5

Tutorial 9

Step 1 9

Step 2 9

Step 3 10

Step 4 10

Step 5 11

Step 6 12

Step 7 15

Step 8 15

Step 9 16

Step 10 17

Step 11 17

Step 12 18

Step 13 18

Summary 21

Sidebars 23

A Testing Smorgasbord 23

Why Unit Test in TestComplete? 23

Code Listings 25

Code Listing 1 25

Code Listing 2 25

Code Listing 3 26

Code Listing 4 27

Symbol Reference 35

Classes 35

TtsUnitTestingClass 35

TtsUnitTestingClass.LoggingEnabled 36

TtsUnitTestingClass.SendToCodeSite 36

TtsUnitTestingClass.CompareFiles 36

TtsUnitTestingClass.CurrentTestMessage 37

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

TtsUnitTestingClass.EvaluateEquality 37
TtsUnitTestingClass.EvaluateForNil 37
TtsUnitTestingClass.FinalizeMessage 38
TtsUnitTestingClass.InitializeMessage 38
TtsUnitTestingClass.InternalTest 38
TtsUnitTestingClass.LogError 38
TtsUnitTestingClass.LogErrorEx 38
TtsUnitTestingClass.LogMessage 38
TtsUnitTestingClass.LogMessageEx 39
TtsUnitTestingClass.LogWarning 39
TtsUnitTestingClass.LogWarningEx 39
TtsUnitTestingClass.NotEqualError 40
TtsUnitTestingClass.StartTestSet 40
TtsUnitTestingClass.StoragesComparisonError 40
TtsUnitTestingClass.Test 40
TtsUnitTestingClass.TestedClassName 40
Files 41
tsUnitTestingClass.pas 41
Index 43

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Purpose of this Paper

This paper is the first of a series designed to provide a focused look at various parts of TestComplete 2.0 by AutomatedQA Corp. This paper in particular provides one of many possible solutions to the question of how to perform unit testing (for Delphi applications) with TestComplete. While this paper stands alone, if you wish to follow along with the tutorial, you will need TestComplete version 2.x and Delphi. You will also need a working knowledge of these tools. While this document contains some introductory topics, it still assumes that you know your way around both TestComplete and Delphi.

Please note once again that this is not the only way to do unit testing with TestComplete. Nor is it even AutomatedQA's design — I began work on this framework prior to joining AutomatedQA (and while it was still AQTest.) I created it primarily because I wanted all my tests to be performed together and the results to all be recorded to the TestComplete log together. Also because I wanted a unified, object-based framework with which to conduct my unit tests. TestComplete's online help contains information about creating self-testing applications, another way of performing unit tests.

One last thought — TestComplete 3.0 is coming and it contains new features in the area of unit testing; this paper is currently only for TestComplete 2.0. I'll write a new paper to deal specifically with TestComplete 3.0 and unit testing.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Introduction

When I first began using TestComplete (AQTest it was called at that time) one of the first things I wanted to figure out was how to best do unit testing with this tool that was obviously strong at automating functional and regression testing. (If you're not familiar with the various types of testing, see the sidebar, A Testing Smorgasbord (see page 23).) This is not an uncommon goal; there seem to be many who have the same interest — the fact that you are reading this means that you probably share it. Even though TestComplete 2.0 is stronger in unit testing, I still use this framework for a variety of reasons. This paper presents the framework that I currently use for performing my unit testing in TestComplete. For a discussion on why to use TestComplete at all for unit testing, see the sidebar, Why Unit Test in TestComplete? (see page 23) There are many different possible approaches to using TestComplete for unit testing, (I went through 3 different major revisions of this framework before settling on this one, and in fact, there are elements of this design which borrow from Eric Holton's article on unit testing in AQTest) so don't think that the approach presented here is the AutomatedQA-authorized "Best Practices" approach. This is merely what I am currently using, and it will probably have changed by the time you read this. Additionally, the approach that I take in this framework has been dictated by the type of programming I've been doing during the time I've been using TestComplete. It's entirely possible that the base class presented here will not meet your needs. With that possibility in mind, I've made the base class as extensible as possible. Finally, remember that this framework is not an alternative to, but rather a supplement to TestComplete's concept of Connected, self-testing applications.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Overview

The arguments for unit testing are manifold and well documented, but are not presented here. If you're reading this, it's assumed that you are aware of the benefits of unit testing. The constraints that precipitated the development of this framework include the following:

- Since I use TestComplete for the automation of some of my functionality testing and all of my regression testing, I wanted to integrate my unit testing into this automated process, keeping one entry point for all my automated testing.
- I wanted my code to test itself, like a TestComplete "self-testing application", yet I wanted the process to be entirely automated — kicked off from a script in TestComplete. Thus I needed my solution to work as both an Open Application and as a Connected Application.
- I wanted my unit testing code to exist as an objectified framework (a set of classes) rather than as occasional calls, de-centralized throughout my application code.
- I wanted my unit testing classes to operate within the unit of the classes they were testing (real unit testing).
- I wanted my descendant unit testing classes to be independent of TestComplete and its COM objects. Only the base class (see page 35) should know how to send messages to the TestComplete Log.
- The entire framework should be conditionally declared so that no part of it would be compiled into the release version of my applications.
- In short, I wanted my tests to operate as displayed in figure 1.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Figure 1 — my preferred automated unit testing scenario

With all that in mind, I designed my current framework around a base class, TtsUnitTestingClass (see page 35). You can see the declaration for this class in code listing 1 (see page 25). Basically, this class functions as an evaluator, a communicator and an all-around organizer. It provides methods for evaluating equality and for communicating with the TestComplete log. In addition, it organizes the process of performing unit tests on a class by informing the TestComplete log object when tests are beginning and ending, and by keeping track of how many cases have been run in a "test set" — more on that later.

This class is the sole inhabitant of the unit tsUnitTestingClass.pas (see page 41) which can be found in its entirety in code listing 4 (see page 27). Note that the prefix "ts" denotes the name of my private consulting company (Thoughtsmithy) prior to joining up with AutomatedQA.

This base class requires some explanation, so we'll begin with a couple of simple class diagrams to better display the relationships involved. Notice the simple relationship diagram of figure 2.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Figure 2 — a simple framework diagram

In the upper left corner, you'll see the TtsUnitTestingClass (see page 35) abstract base class. It is responsible for talking to the TestComplete Log object and for comparing values. Descending from TtsUnitTestingClass (see page 35) are two example testing classes (TmyCustomObject_Tester and TmyOtherObject_Tester) charged with testing classes (TmyCustomObject and TmyOtherObject) on a one-to-one basis. This is a convention I choose to follow for clarity and manageability: I create one testing class descendant for each class to be tested, even if there is more than one tested class in the same unit. In this arrangement, in order to test TmyCustomObject, you would define your TmyCustomObject_Tester class (which resides in the same unit as TmyCustomObject), and declare a set of methods which test the methods of TmyCustomObject. In figure 3 you can see the areas of responsibility of the objects in this framework. Note that when a TestComplete test run is ready to begin unit testing, a TestComplete test script starts the process.

Figure 3 — a more detailed view of the framework

In figure 3 this is represented by a TestComplete script clicking *Some UI Element*. Note that this could also be accomplished by simply calling a method within the tested application — more on that later. Whichever method we choose, we create our unit testing classes (TmyTester in the example) and call their Test (see page 40) methods. The base class' Test method in turn calls InternalTest (see page 38) (which we have overridden) where all our testing methods get called to test our tested class. Why the redundant methods (Test and InternalTest)? I simply do this to allow TtsUnitTestingClass (see

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

page 35) to send a message to the TestComplete Log object before and after calling InternalTest; these messages indicate the beginning and ending of the testing class' test sets. During unit testing, our test class sends results to TestComplete's Log object (via TtsUnitTestingClass (see page 35)). When we're all done with our unit tests, TestComplete (which has been patiently waiting for a sign that unit testing is complete) resumes control and begins running its automated Functional and Regression tests.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Tutorial

The best way to understand this framework is to see it in action. To do so, we will create a very simple set of unit tests for an existing demo project.

TestComplete comes with several sample applications for tutorials and examples of features. Our tutorial will be utilizing one of the TestComplete sample applications. If you installed TestComplete to the default location, the application we'll be using can be found in C:\Program Files\Automated QA\TestComplete\Samples\Open Apps\OrdersDemo\Delphi.

Step 1

Copy project to new location

Description

Since we'll be altering the source code for this project, it would be best to create a copy of the source code so that we don't lose the original. Create a copy of the entire project directory by copying and pasting (within Windows Explorer) the directory C:\Program Files\Automated QA\TestComplete\Samples\Open Apps\OrdersDemo\Delphi. You should now have a folder named C:\Program Files\Automated QA\TestComplete\Samples\Open Apps\OrdersDemo\Copy of Delphi. From our copied folder, open the Orders.dpr project in Delphi.

Step 2

Add conditional unit to the UOrderFrm uses clause

Description

Within this project, the unit UOrderFrm contains a class for which we can create some tests: TDateParser. It is a simple class, but that's perfect for this tutorial. We will create a descendant of TtsUnitTestingClass (see page 35) that will test the TDateParser class. To add our testing class to TDateParser's unit, we'll need to add tsUnitTestingClass to the unit's uses clause. But because we don't want any of our testing code to be compiled into our release build, we'll need to conditionally include the unit in the uses clause.

Open UOrderFrm.pas and add the following text to the uses clause, just before the final semicolon:

```
{$IFDEF UNITTESTING}, tsUnitTestingClass{$ENDIF}
```

The uses clause should now look like this:

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls, Mask, ExtCtrls, ComCtrls{$IFDEF UNITTESTING}, tsUnitTestingClass{$ENDIF};
```

Notice that only when UNITTESTING is defined in the project's options will the tsUnitTestingClass unit will be correctly compiled into the project.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Step 3

Add testing class to unit

Description

Next we will begin adding our testing class to the UOrderFrm unit. Remember that we want our unit testing classes in the same units as the classes they are to test. This is, in part, so that we can take advantage of Delphi's class scoping flaw (allowing classes in the same unit to see each other's private code) to gain access to our tested class' private methods. We've already added our unit to the uses clause, so now let's define our testing class.

Just below the TDateParser declaration, in the interface section, add the following class stub:

```
{$IFDEF UNITTESTING}
TDateParser_Tester = class(TtsUnitTestingClass)
end;
{$ENDIF}
```

Notice that TDateParser_Tester descends from TtsUnitTestingClass (see page 35) and it follows the naming convention of Txxx_Tester — where 'xxx" is the class to be tested, in this case, TDateParser.

Step 4

Override InternalTest and TestedClassName

Description

TtsUnitTestingClass (see page 35) defines two abstract methods that we need to implement: InternalTest and TestedClassName. TestedClassName is simply the name of the class that this class will be testing. InternalTest is the business end of this testing class; it's where we will call all of our testing methods.

In the class declaration we created in the last step, add the declarations for these two methods. The result should look like this:

```
TDateParser_Tester = class(TtsUnitTestingClass)
  protected
    procedure InternalTest; override;
    function TestedClassName: String; override;
end;
```

Scroll down to the bottom of the implementation section of the unit, and add the following:

```
{$IFDEF UNITTESTING}
function TDateParser_Tester.TestedClassName: String;
begin

end;

procedure TDateParser_Tester.InternalTest;
begin

end;
{$ENDIF}
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Notice that once again we conditionally enclose the code so that this class only appears in our testing version of the application.

For TestedClassName, I typically make a call to ClassName rather than returning a string constant. To do so, add the following line to the implementation of TestedClassName:

```
Result := TDateParser.ClassName;
```

As I said, InternalTest is where we'll actually call all of our testing methods. For this tutorial we'll only have 2 testing methods, and they'll be discussed shortly.

But first…

Step 5

Add tested objects

Description

In this framework, I always try to test fresh, new instances of the tested class (rather than an instance that already exists in the application under test, for example.) This affords me the comfort of knowing that the object under test has not been altered without my knowledge.

With that in mind, add the following code (in red) to the class declaration:

```
TDateParser_Tester = class(TtsUnitTestingClass)
  private
    FTestedObject: TDateParser;
    procedure InitTestObjects;
    procedure ClearTestObjects;
  protected
    procedure InternalTest; override;
    function TestedClassName: String; override;
end;
```

FTestedObject is the class instance that we'll use for testing. InitTestObjects creates a new instance of FTestedObject while ClearTestObjects frees the tested object.

Here is the code for these two new methods:

```
procedure TDateParser_Tester.InitTestObjects;
begin

  if (FTestedObject <> nil) then
    FTestedObject.Free;

  FTestedObject := TDateParser.Create;

end;

procedure TDateParser_Tester.ClearTestObjects;
begin
  FreeAndNil(FTestedObject);
end;
```

Make sure to add this code within the conditional block in the implementation section of the unit.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Step 6

Add testing methods

Description

TDateParser has two methods which we'll test for this example: ParseDate and SetDateString.

By convention, I usually add testing methods on a one-to-one basis for methods to be tested. For example, to test TDateParser.ParseDate, I'll create a method TDateParser_Tester.Test_ParseDate. So with that in mind, add the following two methods to TDateParser_Tester:

To the class declaration, add:

```
  private
    procedure Test_ParseDate;
    procedure Test_SetDateString;
```

In the implementation section (inside the conditional $IFDEF block) add:

```
procedure TDateParser_Tester.Test_ParseDate;
begin

end;

procedure TDateParser_Tester.Test_SetDateString;
begin

end;
```

Next we'll write the actual code for these testing methods, but some elaboration is required; I'll discuss the code as we add it. Find the implementation of Test_ParseDate that you just added. First remember the obvious, that this method, Test_ParseDate is here for the purpose of testing the ParseDate method of TDateParser. To that end, we will pass in to ParseDate known values, and examine the results. We know what the results should be, so if they're different, we'll send an error to the log.

Take a look at TDateParser.ParseDate. Basically when this method is called, the given date is broken into its elements, which are stored in private fields, and then the method calls UpdateStrings which adds the date elements to a string list. To test this method, we'll pass in a date, and then read the strings in the list to make sure they are correct. For instance, if we pass in the date, "3/15/1980", the resulting strings should be "15", "3", "March" and "1980".

What follows, then, is the code for Test_ParseDate.

For local variables, add:

```
var
  lsResult: String;
  lDate: TDate;
```

lsResult is a string in which we'll store resultant values to be compared against our baseline. Likewise, lDate is simply a TDate to hold our testing values.

Next, add the following two lines of code to the body of Test_ParseDate:

```
  InitTestObjects;
  StartTestSet('Testing ParseDate');
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

InitTestObjects we discussed earlier, it simply creates an instance of our tested class, but StartTestSet needs some explanation. I've tried to automate the messaging functionality of the base class as much as possible, and to that end, I've designed the class to expect sets of tests within a test method. For instance, you might want to put a calculation through several test cases (standard cases, boundary cases, divide by zero, etc.) without the overhead of having to define a new message for the TestComplete log every time you run a case. This framework handles that for you in the form of test sets. By calling StartTestSet you're telling the base class that you want to start a net set of cases, all with the same basic message ("Testing myCalculation" for example). The base class keeps track of a test index, incrementing and appending that index onto your message each time you run a case. So if you call StartTestSet('Testing myCalculation'); at the start of a set of cases, and then call EvaluateEquality 3 times, the resulting test log will have the following messages (assuming your tests pass):

Testing myCalculation (1) passed.
Testing myCalculation (2) passed.
Testing myCalculation (3) passed.
You can also see an example in figure 4.

Message	Priority	Time	Descript...
Executing Activate	Normal	6:44:48 AM	
Executing RadioButton.Click	Normal	6:44:49 AM	
Executing Activate	Normal	6:44:49 AM	
Beginning unit tests for TbscCalculator.	**Normal**	**6:44:50 AM**	
Testing BattingAve (1) passed	Normal	6:44:50 AM	
Testing BattingAve (2) passed	Normal	6:44:50 AM	
Testing BattingAve (3) passed	Normal	6:44:50 AM	
Testing BattingAve (4) passed	Normal	6:44:50 AM	
Testing TotalBases (1) passed	Normal	6:44:50 AM	
Testing TotalBases (2) passed	Normal	6:44:50 AM	
Testing TotalBases (3) passed	Normal	6:44:50 AM	
Testing TotalBases (4) passed	Normal	6:44:50 AM	
Testing TotalBases (5) passed	Normal	6:44:50 AM	
Testing TotalBases (6) passed	Normal	6:44:50 AM	
Testing SluggingPercentage (1) passed	Normal	6:44:50 AM	
Testing SluggingPercentage (2) passed	Normal	6:44:50 AM	
Testing OnBasePercentage (1) passed	Normal	6:44:50 AM	
Testing OnBasePercentage (2) passed	Normal	6:44:50 AM	
Testing OnBasePercentage (3) passed	Normal	6:44:50 AM	
Testing OnBasePercentage (4) passed	Normal	6:44:50 AM	
Testing OPS (1) passed	Normal	6:44:50 AM	
Testing OPS (2) passed	Normal	6:44:50 AM	
Testing InitialRunsCreated (1) passed	Normal	6:44:50 AM	
Unit tests for TbscCalculator completed.	**Normal**	**6:44:50 AM**	
Executing Click	Normal	6:44:50 AM	

:2 :0 :50 :42 :0

Figure 4 — an example test log

So, by calling StartTestSet at the beginning of each set of test cases, you can simplify and automate the process of message generation for your testing class.

Now we'll add the next two lines of code to Test_ParseDate:

```
{**** test 12/30/1899 ****}
lDate := EncodeDate(1899, 12, 30);
FTestedObject.ParseDate(lDate);
```

This should be fairly self-explanatory: We store in lDate a baseline value of December 30, 1899, then we call the method we are trying to test, ParseDate. The tested object should now decode the date and store 4 string representations of the date in its string list (day, month, month name and year). To

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

check this, we'll get the elements out of the string list and compare them to the values that we know they should be. The next two lines of code are:

```
lsResult := FTestedObject[0];
EvaluateEquality('30', lsResult);
```

lsResult stores the value of the first element in the tested object's string list. We know it had better be "30", representing the 30th day of December, 1899, so we call EvaluateEquality, passing in "30" as the expected value, and lsResult as the actual value. Remember that EvaluateEquality will compare these two values and if they are the same, send a log message indicating that the test passed. If they are different, a log error will be generated.

We want to test the other 3 generated values, so add the following code next:

```
lsResult := FTestedObject[1];
EvaluateEquality('12', lsResult);

lsResult := FTestedObject[2];
EvaluateEquality('December', lsResult);

lsResult := FTestedObject[3];
EvaluateEquality('1899', lsResult);
```

These lines will test the values of the month, month name and year values in the string list.

We'll want to test more than one date, so add these lines for another test case:

```
{**** test 1/1/2001 ****}
lDate := EncodeDate(2001, 1, 1);
FTestedObject.ParseDate(lDate);

lsResult := FTestedObject[0];
EvaluateEquality('1', lsResult);

lsResult := FTestedObject[1];
EvaluateEquality('1', lsResult);

lsResult := FTestedObject[2];
EvaluateEquality('January', lsResult);

lsResult := FTestedObject[3];
EvaluateEquality('2001', lsResult);
```

Note that each time we call EvaluateEquality our test case index is incremented, so each message to the log has a new index. This is handy when one test case fails and you're trying to find out which one it was.

The last thing to add in our test method is a call to clean up our mess:

```
ClearTestObjects;
```

ClearTestObjects frees our tested object.

The final version of this method should look like code listing 2.

Next you can implement Test_SetDateString with the code found in code listing 3.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Step 7

Add testing methods to InternalTest

Description

The next thing we need to do is add our testing methods to InternalTest so that they'll be called when our tests are run.

Find the method body of InternalTest which we added earlier to the implementation section and add to it calls to our two testing methods.

That method should now look like this:

```
procedure TDateParser_Tester.InternalTest;
begin
  Test_ParseDate;
  Test_SetDateString;
end;
```

Now, when we call TDateParser_Tester.Test, InternalTest will in turn be called, thereby executing our two test methods.

Step 8

Add UnitTestingComplete property to main form

Description

Since the whole point of this framework is to allow TestComplete to run our unit tests as part of the automated testing whole, we need a way to tell TestComplete when unit testing is finished. When the unit tests are done, TestComplete can move on with other testing we have scripted. To do this, I normally create a public Boolean property in my application's main form called UnitTestingComplete (conditionally compiled in, of course — it won't exist in the release version.) Simply stated, TestComplete watches this property after it starts the unit testing process; when UnitTestingComplete is True, TestComplete can continue with its other duties.

Until now, we've worked only in the UOrderFrm unit. Now open up the MainForm in UMain.pas. We don't need to add anything to the uses clause, since we're already using the unit which contains our testing class.

Scroll down to the private section of the TMainForm declaration and add the following lines just after the declaration of FDirty:

```
  {$IFDEF UNITTESTING}
    FUnitTestingComplete: Boolean;
  {$ENDIF}
```

Now continue down to the public section and add this code after the declaration of FileName:

```
  {$IFDEF UNITTESTING}
    property UnitTestingComplete: Boolean read FUnitTestingComplete write
FUnitTestingComplete;
  {$ENDIF}
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

We need to initialize this property so that it will be False when we start the unit tests. Find the FormCreate method implementation, and add these lines:

```
{$IFDEF UNITTESTING}
  FUnitTestingComplete := False;
{$ENDIF}
```

TestComplete will be able to see this property and our test scripts will monitor it so as to know when it is safe to move on.

Step 9

Add DoUnitTests method to Main form

Description

Next we need to add the method that TestComplete will call to start the unit testing process. Note that our implementation here is an example of how we can take multiple approaches to this process. I used to call the DoUnitTests method from a UI element — perhaps a button click. TestComplete would start the automated testing process, launch the tested application and then click a button to kick off the unit testing. This approach is described in figures 1 and 3 at the beginning of this paper. However, now I do things a little differently; now I just have TestComplete call DoUnitTests directly. Either way works just fine, and since I still have to call DoUnitTests somewhere in my application code (to force Delphi to link in the code) there's very little difference ultimately.

To add the DoUnitTests method, find the public section of the TMainForm class declaration and add the following lines after the declaration of the ChangeOrder method:

```
  {$IFDEF UNITTESTING}
    procedure DoUnitTests;
  {$ENDIF}
```

In the implementation section of the unit, add the following method implementation:

```
{$IFDEF UNITTESTING}
procedure TMainForm.DoUnitTests;
var
  lTester: TDateParser_Tester;
begin

  if not (FUnitTestingComplete) then
    begin
      lTester := TDateParser_Tester.Create;

      try
        lTester.Test;

      finally
        lTester.Free;
        FUnitTestingComplete := True;
      end;
    end;
end;
{$ENDIF}
```

This method is fairly simple: we check FUnitTestingComplete to make sure we only run these tests once, then we create an instance of the tester, call its Test method, free the instance and then finally set

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

FUnitTestingComplete to True.

Step 10

Call DoUnitTests

Description

As I mentioned, if we don't call DoUnitTests somewhere in our application, Delphi's ever-diligent linker will refuse to link in the method, making it impossible for TestComplete to call it. As a result, whether we use TestComplete to call DoUnitTests directly, or we click on some button which in turn calls DoUnitTests is immaterial — we wind up with the same code in the application.

Pick some UI event (I chose Records_NewOrderClick) and add:

```
{$IFDEF UNITTESTING}
  DoUnitTests;
{$ENDIF}
```

Now the method will be included in your testing build. Note that even though DoUnitTests will be called anytime you click the chosen button, it doesn't matter since FUnitTestingComplete prevents the unit testing from being executed more than once.

Step 11

Define "UNITTESTING" and build

Description

We're just about done here — one more thing to do in Delphi and then we'll move to TestComplete. We need to compile the application with 'UNITTESTING' defined so that all our unit testing code will be built in. In Delphi, go to Project|Options and select the Directories/Conditionals tab. In the Conditional Defines field, add UNITTESTING and click OK (see figure 5.) Now build the application. Assuming all is well, TestComplete can now perform automated unit testing on your application.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Figure 5 — Conditional defines in the Project Options dialog.

Step 12

Create a new TestComplete project

Description

Finally we can do some work in the product in question. Launch TestComplete and select File|New|Project…

In the New Project dialog, select Delphi, DelphiScript.

Specify a location for your new project (C:\Program Files\Automated QA\TestComplete\Projects\Tutorials is good), and then type "UnitTestingPaperDemo" into the Project Name field. Finally, click OK (see figure 6.)

Figure 6 — TestComplete's New Project dialog box

You now have a new project with which to see our unit testing in action.

We need to add the application to our Tested Application list, so select File|Tested Applications…, click Add and then add the compiled Orders.exe to the list. Click OK when you're done.

Step 13

Make the call

Description

And now the moment we've all been waiting for — alter Unit1 to look as follows:

```
procedure Test; forward;

procedure Main;
begin
  try
    TestedApps.RunAll;
    Test;
  except
    Log.Error('Exception', ExceptionMessage);
  end;
end;

procedure Test;
begin
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  Sys.Process('Orders').MainForm.DoUnitTests;

  while not(Sys.Process('Orders').MainForm.UnitTestingComplete) do
    Sys.Delay(1000);

  //Do some other testing

end;
```

That's it. Main is the project's main routine, so when you start the test project, Main will launch our tested application and call Test which will in turn call the main form's DoUnitTests method. Our unit tests will run, while TestComplete waits, then when they're done, TestComplete moves on. Of course, in this particular case there's nothing for TestComplete to do afterward, but in a real-world case, you would execute your other testing scripts after the unit tests are completed.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Summary

As you can see, this unit testing framework is rather straightforward and achieves the goal providing automated unit testing through TestComplete while minimizing the amount of contact each class has TestComplete's objects; coalescing most of the common functionality into a single unit testing base class.

As I've said, this is a framework in constant development. It does what I've needed, but as my needs continue to change, no doubt this design will as well. I welcome any suggestions you may have and you are free to contact me with questions or suggestions at robertl@automatedqa.com.

Note that most of the code in this paper can be found in the Code Listings (see page 25) section.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Sidebars

A Testing Smorgasbord

A discussion of various types of testing.

Description

Are you unfamiliar with the general concepts of software testing? Then perhaps a quick overview is in order. Testing in general can be defined as "comparing output against an expected standard." As it says in the TestComplete online help:

…the basic test sequence is —

- Define expected output
- Feed corresponding input
- Gather output
- Compare to expected output
- Call for attention if the comparison fails

There are generally three major types of testing: unit, functional and regression.

Unit testing (or module testing) tests the interface between units (modules, functions, libraries, classes, etc.) and the rest of the code, as well as the user or the system (where this applies). In other words, unit testing evaluates objects in the problem domain (stuff the user doesn't see.)

Functional testing tests the interface between the application on one side, and the rest of the system and users on the other side. In other words, functional testing evaluates objects in the presentation domain (stuff the user can see.)

Regression testing is based on the idea that tests repeat. To perform a regression test on something means to run the same test you ran prior to the last modification, and check that you get the same result you got and accepted then. Thus, each test builds support for the next run of tests. In other words, regression testing makes sure that your latest changes didn't break anything and especially that they didn't re-break something that was once broken but had been fixed.

TestComplete is an unusual tool in that it handles all three types of testing well.

Why Unit Test in TestComplete?

Reasons why TestComplete is well suited to unit testing.

Description

More than once I've shown TestComplete to adherents of Extreme Programming and gotten the response that it looks like overkill. "Why do I need all that when I have DUnit?" If all you want to do is unit testing, then this is a very good question. TestComplete would still be a decent way to unit test, but there are a good many features that would go unused. But the question I would ask is, why do you think your QA job is finished when you've finished unit testing? There seems to be a continuum in the area of belief in where the responsibility for functional and regression testing lies. Some feel that it is entirely the responsibility of human testers, while others feel it should be entirely automated. Those

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

who feel that unit testing is all they need to worry about are typically in the former camp, believing that functional and regression testing should all be handled by humans. Personally, I feel the answer lies in the middle. While there should always be humans involved in functional and regression testing, I think that more than half of functional testing should be automated and that the vast majority of regression should be handled by automation. Studies have shown that any time humans are engaged in performing the same tasks repeatedly (like regression testing) concentration (and satisfaction) drops off considerably. I'd much rather automate all that and free my human testers to find new problems.

With all that being said, one of the best reasons to consider TestComplete for unit testing is the ability to combine all three types of testing into one automated whole. I can automatically run all of my unit testing, almost all of my regression testing and much of my functional testing at 3:00 in the morning (when no one's around) and have a unified result log (and a full history of result logs) waiting for me when I arrive in the morning. In addition, the automated nature of this unified suite of tests means that all my tests will be verified every time; I never have to worry about whether some set of unit testing cases were skipped accidentally.

Overkill? Well, what would you rather be doing at 3:00 in the morning?

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Code Listings

Code Listing 1

The TtsUnitTestingClass (see page 35) declaration.

Description

```
TtsUnitTestingClass = class (TObject)
private
  FLoggingEnabled: Boolean;
  FSendToCodeSite: Boolean;
  FTestIndex: Integer;
  FTestSetMessage: String;
protected
  procedure CompareFiles(const aFileName1, aFileName2, aMsg: String); virtual;
  function CurrentTestMessage: String; virtual;
  procedure EvaluateEquality(aExpected, aActual: Double); overload; virtual;
  procedure EvaluateEquality(aExpected, aActual: Integer); overload; virtual;
  procedure EvaluateEquality(const aExpected, aActual: String); overload; virtual;
  procedure EvaluateEquality(aExpected, aActual: TComponent); overload; virtual;
  procedure EvaluateEquality(aExpected, aActual: TObject; const aExpectedName,
            aActualName: String); overload; virtual;
  procedure EvaluateForNil(aObject: TObject; const aObjectName: String); virtual;
  function FinalizeMessage: String; virtual;
  function InitializeMessage: String; virtual;
  procedure InternalTest; virtual; abstract;
  procedure LogError(const aMessage: String); virtual;
  procedure LogErrorEx(const aMessage, aMessageEx: String; aPriority, aFontStyle,
            aFontColor, aColor: Integer); virtual;
  procedure LogMessage(const aMessage: String); virtual;
  procedure LogMessageEx(const aMessage, aMessageEx: String; aPriority,
            aFontStyle, aFontColor, aColor: Integer); virtual;
  procedure LogWarning(const aMessage: String); virtual;
  procedure LogWarningEx(const aMessage, aMessageEx: String; aPriority,
            aFontStyle, aFontColor, aColor: Integer); virtual;
  procedure NotEqualError(const aExpected, aActual, aMsg: String); virtual;
  procedure StartTestSet(const aTestSetMsg: String); virtual;
  procedure StoragesComparisonError(const aItemName1, aItemName2, aCompareType,
            aMsg, aErrorText: String); virtual;
  function TestedClassName: String; virtual; abstract;
public
  constructor Create;
  procedure Test;
  property LoggingEnabled: Boolean read FLoggingEnabled write FLoggingEnabled;
  property SendToCodeSite: Boolean read FSendToCodeSite write FSendToCodeSite;
end;
```

Code Listing 2

The implementation of TDateParser_Tester.Test_ParseDate.

Description

```
procedure TDateParser_Tester.Test_ParseDate;
var
  lsResult: String;
  lDate: TDate;
begin

  InitTestObjects;
  StartTestSet('Testing ParseDate');
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  {**** test 12/30/1899 ****}
  lDate := EncodeDate(1899, 12, 30);
  FTestedObject.ParseDate(lDate);

  lsResult := FTestedObject[0];
  EvaluateEquality('30', lsResult);

  lsResult := FTestedObject[1];
  EvaluateEquality('12', lsResult);

  lsResult := FTestedObject[2];
  EvaluateEquality('December', lsResult);

  lsResult := FTestedObject[3];
  EvaluateEquality('1899', lsResult);

  {**** test 1/1/2001 ****}
  lDate := EncodeDate(2001, 1, 1);
  FTestedObject.ParseDate(lDate);

  lsResult := FTestedObject[0];
  EvaluateEquality('1', lsResult);

  lsResult := FTestedObject[1];
  EvaluateEquality('1', lsResult);

  lsResult := FTestedObject[2];
  EvaluateEquality('January', lsResult);

  lsResult := FTestedObject[3];
  EvaluateEquality('2001', lsResult);

  ClearTestObjects;

end;
```

Code Listing 3

The implementation of TDateParser_Tester.Test_SetDateString.

Description

```
procedure TDateParser_Tester.Test_SetDateString;
var
  lsResult: String;
begin

  InitTestObjects;
  StartTestSet('Testing SetDateString');

  {**** test 2/14/2002 ****}
  FTestedObject.DateString := '2/14/2002';
  lsResult := FTestedObject.FDateString;
  EvaluateEquality('2/14/2002', lsResult);

  lsResult := FTestedObject[0];
  EvaluateEquality('14', lsResult);

  lsResult := FTestedObject[1];
  EvaluateEquality('2', lsResult);

  lsResult := FTestedObject[2];
  EvaluateEquality('February', lsResult);

  lsResult := FTestedObject[3];
  EvaluateEquality('2002', lsResult);
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  {**** test 4/17/1970 ****}
  FTestedObject.DateString := '4/17/1970';
  lsResult := FTestedObject.FDateString;
  EvaluateEquality('4/17/1970', lsResult);

  lsResult := FTestedObject[0];
  EvaluateEquality('17', lsResult);

  lsResult := FTestedObject[1];
  EvaluateEquality('4', lsResult);

  lsResult := FTestedObject[2];
  EvaluateEquality('April', lsResult);

  lsResult := FTestedObject[3];
  EvaluateEquality('1970', lsResult);

  ClearTestObjects;

end;
```

Code Listing 4

The tsUnitTestingClass.pas (see page 41) unit.

Description

```
{*****************************************************************************
**
**  tsUnitTestingClass
**  As of: 5/27/2003
**  ©2002 Thoughtsmithy
**  ©2003 AutomatedQA Corp
**
*****************************************************************************}



unit tsUnitTestingClass;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, TCClient, TCConnect{$IFDEF CODESITE}, csIntf{$ENDIF};

type
  TtsUnitTestingClass = class (TObject)
  private
    FLoggingEnabled: Boolean;
    FSendToCodeSite: Boolean;
    FTestIndex: Integer;
    FTestSetMessage: String;
  protected
    procedure CompareFiles(const aFileName1, aFileName2, aMsg: String); virtual;
    function CurrentTestMessage: String; virtual;
    procedure EvaluateEquality(aExpected, aActual: Double); overload; virtual;
    procedure EvaluateEquality(aExpected, aActual: Integer); overload; virtual;
    procedure EvaluateEquality(const aExpected, aActual: String); overload;
virtual;
    procedure EvaluateEquality(aExpected, aActual: TComponent); overload; virtual;
    procedure EvaluateEquality(aExpected, aActual: TObject; const aExpectedName,
aActualName:
            String); overload; virtual;
    procedure EvaluateForNil(aObject: TObject; const aObjectName: String); virtual;
    function FinalizeMessage: String; virtual;
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
    function InitializeMessage: String; virtual;
    procedure InternalTest; virtual; abstract;
    procedure LogError(const aMessage: String); virtual;
    procedure LogErrorEx(const aMessage, aMessageEx: String; aPriority,
aFontStyle, aFontColor,
            aColor: Integer); virtual;
    procedure LogMessage(const aMessage: String); virtual;
    procedure LogMessageEx(const aMessage, aMessageEx: String; aPriority,
aFontStyle, aFontColor,
            aColor: Integer); virtual;
    procedure LogWarning(const aMessage: String); virtual;
    procedure LogWarningEx(const aMessage, aMessageEx: String; aPriority,
aFontStyle, aFontColor,
            aColor: Integer); virtual;
    procedure NotEqualError(const aExpected, aActual, aMsg: String); virtual;
    procedure StartTestSet(const aTestSetMsg: String); virtual;
    procedure StoragesComparisonError(const aItemName1, aItemName2, aCompareType,
aMsg, aErrorText:
            String); virtual;
    function TestedClassName: String; virtual; abstract;
  public
    constructor Create;
    procedure Test;
    property LoggingEnabled: Boolean read FLoggingEnabled write FLoggingEnabled;
    property SendToCodeSite: Boolean read FSendToCodeSite write FSendToCodeSite;
  end;


implementation


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.Create}

constructor TtsUnitTestingClass.Create;
begin
  inherited Create;

  FLoggingEnabled := True;
  FSendToCodeSite := False;

end; { TtsUnitTestingClass.Create() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.CompareFiles}

procedure TtsUnitTestingClass.CompareFiles(const aFileName1, aFileName2, aMsg:
String);
var
  lsErrorText: String;
  lsMsg: String;
begin

  if (aMsg = '') then
    lsMsg := CurrentTestMessage

  else
    lsMsg := aMsg;

  if Files.Compare(aFileName1, aFileName2) then
    LogMessage(lsMsg + ' passed')

  else
    begin
      lsErrorText := Files.LastError;
      StoragesComparisonError(aFileName1, aFilename2, 'Files', lsMsg, lsErrorText);
    end;
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  if (aMsg = '') then
    Inc(FTestIndex);

end; { TtsUnitTestingClass.CompareFiles() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.CurrentTestMessage}

function TtsUnitTestingClass.CurrentTestMessage: String;
begin
  Result := Format(FTestSetMessage + ' (%d)', [FTestIndex]);
end; { TtsUnitTestingClass.CurrentTestMessage() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateEquality}

procedure TtsUnitTestingClass.EvaluateEquality(aExpected, aActual: Double);
begin
  EvaluateEquality(FloatToStrF(aExpected, ffFixed, 15, 3),
      FloatToStrF(aActual, ffFixed, 15, 3));
end; { TtsUnitTestingClass.EvaluateEquality() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateEquality}

procedure TtsUnitTestingClass.EvaluateEquality(aExpected, aActual: Integer);
begin

  if (aExpected <> aActual) then
    NotEqualError(IntToStr(aExpected), IntToStr(aActual), CurrentTestMessage)

  else
    LogMessage(CurrentTestMessage + ' passed');

  Inc(FTestIndex);

end; { TtsUnitTestingClass.EvaluateEquality() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateEquality}

procedure TtsUnitTestingClass.EvaluateEquality(const aExpected, aActual: String);
begin

  if (aExpected <> aActual) then
    NotEqualError(aExpected, aActual, CurrentTestMessage)

  else
    LogMessage(CurrentTestMessage + ' passed');

  Inc(FTestIndex);

end; { TtsUnitTestingClass.EvaluateEquality() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateEquality}

procedure TtsUnitTestingClass.EvaluateEquality(aExpected, aActual: TComponent);
begin

  if (aExpected <> aActual) then
    NotEqualError(aExpected.Name, aActual.Name, CurrentTestMessage)

  else
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
    LogMessage(CurrentTestMessage + ' passed');

  Inc(FTestIndex);

end; { TtsUnitTestingClass.EvaluateEquality() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateEquality}

procedure TtsUnitTestingClass.EvaluateEquality(aExpected, aActual: TObject; const
aExpectedName,
        aActualName: String);
begin

  if (aExpected <> aActual) then
    NotEqualError(aExpectedName, aActualName, CurrentTestMessage)

  else
    LogMessage(CurrentTestMessage + ' passed');

  Inc(FTestIndex);

end; { TtsUnitTestingClass.EvaluateEquality() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.EvaluateForNil}

procedure TtsUnitTestingClass.EvaluateForNil(aObject: TObject; const aObjectName:
String);
begin

  if (aObject <> nil) then
    LogError('Unexpected value: ' + CurrentTestMessage + ' - ' + aObjectName + '
should be nil, but is not')

  else
    LogMessage(CurrentTestMessage + ' passed');

  Inc(FTestIndex);

end; { TtsUnitTestingClass.EvaluateForNil() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.FinalizeMessage}

function TtsUnitTestingClass.FinalizeMessage: String;
begin
  Result := 'Unit tests for ' + TestedClassName + ' completed.';
end; { TtsUnitTestingClass.FinalizeMessage() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.InitializeMessage}

function TtsUnitTestingClass.InitializeMessage: String;
begin
  Result := 'Beginning unit tests for ' + TestedClassName + '.';
end; { TtsUnitTestingClass.InitializeMessage() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogError}

procedure TtsUnitTestingClass.LogError(const aMessage: String);
begin
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendError(aMessage)

    else
  {$ENDIF}
      Log.Error(aMessage);

end; { TtsUnitTestingClass.LogError() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogErrorEx}

procedure TtsUnitTestingClass.LogErrorEx(const aMessage, aMessageEx: String;
aPriority, aFontStyle,
        aFontColor, aColor: Integer);
begin

  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendError(aMessage)

    else
  {$ENDIF}
      Log.Error(aMessage, aMessageEx, aPriority, aFontStyle, aFontColor, aColor);

end; { TtsUnitTestingClass.LogErrorEx() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogMessage}

procedure TtsUnitTestingClass.LogMessage(const aMessage: String);
begin

  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendMsg(aMessage)

    else
  {$ENDIF}
      Log.Message(aMessage);

end; { TtsUnitTestingClass.LogMessage() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogMessageEx}

procedure TtsUnitTestingClass.LogMessageEx(const aMessage, aMessageEx: String;
aPriority,
        aFontStyle, aFontColor, aColor: Integer);
begin

  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendMsg(aMessage)

    else
  {$ENDIF}
      Log.Message(aMessage, aMessageEx, aPriority, aFontStyle, aFontColor, aColor);

end; { TtsUnitTestingClass.LogMessageEx() }
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogWarning}

procedure TtsUnitTestingClass.LogWarning(const aMessage: String);
begin

  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendWarning(aMessage)

    else
  {$ENDIF}
      Log.Warning(aMessage);

end; { TtsUnitTestingClass.LogWarning() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.LogWarningEx}

procedure TtsUnitTestingClass.LogWarningEx(const aMessage, aMessageEx: String;
aPriority,
        aFontStyle, aFontColor, aColor: Integer);
begin

  if FLoggingEnabled then
  {$IFDEF CODESITE}
    if FSendToCodeSite then
      CodeSite.SendWarning(aMessage)

    else
  {$ENDIF}
      Log.Warning(aMessage, aMessageEx, aPriority, aFontStyle, aFontColor, aColor);

end; { TtsUnitTestingClass.LogWarningEx() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.NotEqualError}

procedure TtsUnitTestingClass.NotEqualError(const aExpected, aActual, aMsg:
String);
begin
  LogError(Format('Unexpected value: %s; expected: %s, actual: %s', [aMsg,
aExpected, aActual]));
end; { TtsUnitTestingClass.NotEqualError() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.StartTestSet}

procedure TtsUnitTestingClass.StartTestSet(const aTestSetMsg: String);
begin

  FTestIndex := 1;
  FTestSetMessage := aTestSetMsg;

end; { TtsUnitTestingClass.StartTestSet() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.StoragesComparisonError}

procedure TtsUnitTestingClass.StoragesComparisonError(const aItemName1,
aItemName2, aCompareType,
        aMsg, aErrorText: String);
begin
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
  LogError(Format('A %s comparison failed: %s; %s did not match %s. Error: %s',
      [aCompareType, aMsg, aItemName1, aItemName2, aErrorText]));
end; { TtsUnitTestingClass.StoragesComparisonError() }


{:-----------------------------------------------------------------------------
-*TtsUnitTestingClass.Test}

procedure TtsUnitTestingClass.Test;
begin

  LogMessageEx(InitializeMessage, '', pmNormal, fmBold, clNavy, clMoneyGreen);
  InternalTest;
  LogMessageEx(FinalizeMessage, '', pmNormal, fmBold, clNavy, clMoneyGreen);

end; { TtsUnitTestingClass.Test() }




end.
```

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Symbol Reference

Classes

TtsUnitTestingClass

TtsUnitTestingClass is the base class for all testing classes in this TestComplete unit testing framework.

Class Hierarchy

TObject
TtsUnitTestingClass

```
TtsUnitTestingClass = class(TObject)
```

File

tsUnitTestingClass (see page 41)

Description

Derive a class from TtsUnitTestingClass to create a testing class that is responsible for testing one specific class. The new testing class should reside within the same unit as the class to be tested.

Descendant classes should override the abstract methods InternalTest (see page 38) and TestedClassName (see page 40).

InternalTest (see page 38) should perform all the unit tests for the tested class.

TestedClasName should return the name of the tested class.

Members

Properties

Property	Description
LoggingEnabled (see page 36)	Enables or disables the sending of messages to the log.
SendToCodeSite (see page 36)	Determines whether log messages will be sent to Raize CodeSite instead of to the TestComplete Log.

Methods

Method	Description
CompareFiles (see page 36)	Compares two files and logs the result.
CurrentTestMessage (see page 37)	Used in the support of test sets and messaging.
EvaluateEquality (see page 37)	Compares two items and logs the result.
EvaluateEquality (see page 37)	Compares two items and logs the result.
EvaluateEquality (see page 37)	Compares two items and logs the result.
EvaluateEquality (see page 37)	Compares two items and logs the result.
EvaluateEquality (see page 37)	Compares two items and logs the result.
EvaluateForNil (see page 37)	EvaluateForNil can be called when an object should be nil. Pass the object reference that should be nil and a name for that object.
FinalizeMessage (see page 38)	Returns the message that is sent to the Log when the unit tests end.
InitializeMessage (see page 38)	Returns the message that is sent to the Log when the unit tests begin.
InternalTest (see page 38)	Called by Test (see page 40) for class-specific testing.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

LogError (see page 38)	Calls Log.Error.
LogErrorEx (see page 38)	Calls Log.Error with full parameter list.
LogMessage (see page 38)	Calls Log.Message.
LogMessageEx (see page 39)	Calls Log.Message with full parameter list.
LogWarning (see page 39)	Calls Log.Warning.
LogWarningEx (see page 39)	Calls Log.Warning with full parameter list.
NotEqualError (see page 40)	Called by EvaluateEquality (see page 37) methods to log an error.
StartTestSet (see page 40)	Initializes internal fields for a new test set.
StoragesComparisonError (see page 40)	Called by the CompareFiles (see page 36) method to log a file comparison error.
Test (see page 40)	Starts the unit testing process for descendant classes.
TestedClassName (see page 40)	Returns a string representation of the classname of the class under test.

Legend

	protected
	virtual
	abstract

TtsUnitTestingClass.LoggingEnabled

Enables or disables the sending of messages to the log.

```
property LoggingEnabled: Boolean;
```

Description

LoggingEnabled determines whether the messages generated by calls to EvaluateEquality (see page 37) will actually be sent to the Log object (or to CodeSite if SendToCodeSite (see page 36) is True).

TtsUnitTestingClass.SendToCodeSite

Determines whether log messages will be sent to Raize CodeSite instead of to the TestComplete Log.

```
property SendToCodeSite: Boolean;
```

Description

SendToCodeSite provides the option of sending the unit testing messages to CodeSite rather than to the Log object. Set this property to true to redirect all messages to CodeSite.

Defaults to False.

TtsUnitTestingClass.CompareFiles

Compares two files and logs the result.

```
procedure CompareFiles(const aFileName1: String; const aFileName2: String; const
aMsg: String); virtual;
```

Parameters

Parameters	Description
`const aFileName1: String`	a file to be compared; this file can either be on disk on in the *Files* storage.
`const aFileName2: String`	a 2nd file to be compared.
`const aMsg: String`	a message to be sent to the TestComplete Log object.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Description

Call CompareFiles to use TestComplete's Files object to perform a comparison.

CompareFiles calls Files.Compare using aFileName1 and aFileName2 (see the TestComplete online help for more information on the Files object.) If Files.Compare returns false, CompareFiles gets the latest error message from the Files object and posts it as part of the error message sent to the log.

Notes

The aMsg parameter is optional. If it is empty, CurrentTestMessage (see page 37) is used instead, allowing this test to be part of a test set (see StartTestSet (see page 40)). The test index is only incremented if aMsg is empty and CurrentTestMessage (see page 37) is used as part of the logged message.

See Also

StartTestSet (see page 40)

TtsUnitTestingClass.CurrentTestMessage

Used in the support of test sets and messaging.

```
function CurrentTestMessage: String; virtual;
```

Description

When one of the EvaluateEquality (see page 37) methods sends a message (or error) to the log, the result of CurrentTestMessage will be added to the message. See StartTestSet (see page 40) for more information.

See Also

StartTestSet (see page 40)

TtsUnitTestingClass.EvaluateEquality

Compares two items and logs the result.

```
procedure EvaluateEquality(aExpected: Double; aActual: Double); virtual; overload;
procedure EvaluateEquality(aExpected: Integer; aActual: Integer); virtual;
overload;
procedure EvaluateEquality(const aExpected: String; const aActual: String);
virtual; overload;
procedure EvaluateEquality(aExpected: TComponent; aActual: TComponent); virtual;
overload;
procedure EvaluateEquality(aExpected: TObject; aActual: TObject; const
aExpectedName: String; const aActualName: String); virtual; overload;
```

Description

Call EvaluateEquality to do a comparison of two elements. If the elements are equivilent, a message is posted to the log indicating a passed test. If the elements are inequivilent, an error is posted to the log.

TtsUnitTestingClass.EvaluateForNil

```
procedure EvaluateForNil(aObject: TObject; const aObjectName: String); virtual;
```

Description

EvaluateForNil can be called when an object should be nil. Pass the object reference that should be nil and a name for that object.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

TtsUnitTestingClass.FinalizeMessage

```
function FinalizeMessage: String; virtual;
```

Description

Returns the message that is sent to the Log when the unit tests end.

TtsUnitTestingClass.InitializeMessage

```
function InitializeMessage: String; virtual;
```

Description

Returns the message that is sent to the Log when the unit tests begin.

TtsUnitTestingClass.InternalTest

Called by Test (see page 40) for class-specific testing.

```
procedure InternalTest; virtual; abstract;
```

Description

Override InternalTest for your descendant classes to provide class-specific testing. The overridden InternalTest should call testing methods that perform unit tests particular to the class under test.

TtsUnitTestingClass.LogError

Calls Log.Error.

```
procedure LogError(const aMessage: String); virtual;
```

Description

Passes aMessage on to the TestComplete log object by calling Log.Error. Descendant classes should call LogError rather than calling Log.Error directly.

See Also

LogErrorEx (see page 38), LogMessage (see page 38), LogMessageEx (see page 39), LogWarning (see page 39), LogWarningEx (see page 39)

TtsUnitTestingClass.LogErrorEx

Calls Log.Error with full parameter list.

```
procedure LogErrorEx(const aMessage: String; const aMessageEx: String; aPriority:
Integer; aFontStyle: Integer; aFontColor: Integer; aColor: Integer); virtual;
```

Description

Passes aMessage and the other parameters on to the TestComplete log object by calling Log.Error. Descendant classes should call LogErrorEx or LogError (see page 38) rather than calling Log.Error directly.

See Also

LogError (see page 38), LogMessage (see page 38), LogMessageEx (see page 39), LogWarning (see page 39), LogWarningEx (see page 39)

TtsUnitTestingClass.LogMessage

Calls Log.Message.

```
procedure LogMessage(const aMessage: String); virtual;
```


©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Description

Passes aMessage on to the TestComplete log object by calling Log.Message. Descendant classes should call LogError (see page 38) rather than calling Log.Error directly.

See Also

LogError (see page 38), LogErrorEx (see page 38), LogMessageEx (see page 39), LogWarning (see page 39), LogWarningEx (see page 39)

TtsUnitTestingClass.LogMessageEx

Calls Log.Message with full parameter list.

```
procedure LogMessageEx(const aMessage: String; const aMessageEx: String;
aPriority: Integer; aFontStyle: Integer; aFontColor: Integer; aColor: Integer);
virtual;
```

Description

Passes aMessage and the other parameters on to the TestComplete log object by calling Log.Message. Descendant classes should call LogMessageEx or LogMessage (see page 38) rather than calling Log.Message directly.

See Also

LogError (see page 38), LogErrorEx (see page 38), LogMessage (see page 38), LogWarning (see page 39), LogWarningEx (see page 39)

TtsUnitTestingClass.LogWarning

Calls Log.Warning.

```
procedure LogWarning(const aMessage: String); virtual;
```

Description

Passes aMessage on to the TestComplete log object by calling Log.Warning. Descendant classes should call LogWarning rather than calling Log.Warning directly.

See Also

LogError (see page 38), LogErrorEx (see page 38), LogMessage (see page 38), LogMessageEx (see page 39), LogWarningEx (see page 39)

TtsUnitTestingClass.LogWarningEx

Calls Log.Warning with full parameter list.

```
procedure LogWarningEx(const aMessage: String; const aMessageEx: String;
aPriority: Integer; aFontStyle: Integer; aFontColor: Integer; aColor: Integer);
virtual;
```

Description

Passes aMessage and the other parameters on to the TestComplete log object by calling Log.Warning. Descendant classes should call LogWarningEx or LogWarning (see page 39) rather than calling Log.Warning directly.

See Also

LogError (see page 38), LogErrorEx (see page 38), LogMessage (see page 38), LogMessageEx (see page 39), LogWarning (see page 39)

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

TtsUnitTestingClass.NotEqualError

Called by EvaluateEquality (see page 37) methods to log an error.

```
procedure NotEqualError(const aExpected: String; const aActual: String; const
aMsg: String); virtual;
```

Description

NotEqualError is called by the EvaluateEquality (see page 37) methods when an Actual value does not match an Expected value. The error sent to the TestComplete Log appears as, "Unexpected value: Testing DoSomething (n) expected: SomeValue, actual: SomeOtherValue" where "Testing DoSomething (n)" is the result of CurrentTestMessage (see page 37).

See Also

CurrentTestMessage (see page 37), EvaluateEquality (see page 37)

TtsUnitTestingClass.StartTestSet

Initializes internal fields for a new test set.

```
procedure StartTestSet(const aTestSetMsg: String); virtual;
```

Description

Causes the testing object to reset the TestIndex and sets the TestSetMessage to aTestSetMsg. StartTestSet should be called each time a new functionality is to be tested; this way, each time EvaluateEquality (see page 37) is called, TestIndex is incremented and a message is sent to the log as:

"Testing DoSomething (n): passed." where "n" is the current value of TestIndex.

TtsUnitTestingClass.StoragesComparisonError

Called by the CompareFiles (see page 36) method to log a file comparison error.

```
procedure StoragesComparisonError(const aItemName1: String; const aItemName2:
String; const aCompareType: String; const aMsg: String; const aErrorText: String);
virtual;
```

Description

NotEqualError (see page 40) is called by the CompareFiles (see page 36) method when Files.Compare returns False; this can happen either when the files do not match or when there is a real error.

See Also

CompareFiles (see page 36)

TtsUnitTestingClass.Test

Starts the unit testing process for descendant classes.

```
procedure Test;
```

Description

Call Test in order to begin unit testing for the current object. Test in turn calls the abstract method InternalTest (see page 38).

Descendant classes should override InternalTest (see page 38) in order to describe a set of test methods to be called.

TtsUnitTestingClass.TestedClassName

Returns a string representation of the classname of the class under test.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

```
function TestedClassName: String; virtual; abstract;
```

Description

Override TestedClassName for your descendant classes to return the name of the class under test. This value is used by the base class in messaging.

Files

tsUnitTestingClass.pas

Classes

Class	Description
TtsUnitTestingClass (see page 35)	TtsUnitTestingClass is the base class for all testing classes in this TestComplete unit testing framework.

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.

Index

A

A Testing Smorgasbord 23

C

Code Listing 1 25

Code Listing 2 25

Code Listing 3 26

Code Listing 4 27

Code Listings 25

I

Introduction 3

O

Overview 5

P

Purpose of this Paper 1

S

Sidebars 23

Step 1 9

Step 10 17

Step 11 17

Step 12 18

Step 13 18

Step 2 9

Step 3 10

Step 4 10

Step 5 11

Step 6 12

Step 7 15

Step 8 15

Step 9 16

Summary 21

T

tsUnitTestingClass.pas 41

TtsUnitTestingClass 35

- CompareFiles 36
- CurrentTestMessage 37
- EvaluateEquality 37
- EvaluateForNil 37
- FinalizeMessage 38
- InitializeMessage 38
- InternalTest 38
- LogError 38
- LogErrorEx 38
- LoggingEnabled 36
- LogMessage 38
- LogMessageEx 39
- LogWarning 39
- LogWarningEx 39
- NotEqualError 40
- SendToCodeSite 36
- StartTestSet 40
- StoragesComparisonError 40
- Test 40
- TestedClassName 40

Tutorial 9

W

Why Unit Test in TestComplete? 23

©2003 by Robert K. Leahey and AutomatedQA Corp. All rights reserved.