Paper 002-2008

Automated Unit Testing for SAS® Applications

David A. Scocca, Rho, Inc., Chapel Hill, NC

ABSTRACT

As your software application grows larger and more complex, so do your testing requirements. Unfortunately the complexity of that testing grows even more rapidly than the scope of your application. Incorporating low-level unit testing into the development process can help ease your pain. A system of automated unit testing and test-driven development allows you to continually test new code as you write it, to easily check interactions between new or modified code and the rest of your application, and to produce a robust and well-tested product for your customer.

SCLUnit is a SAS/AF® implementation of the JUnit testing framework for Java development which supports the development and automation of unit tests for a SAS application developed using the SAS Component Object Model (SCOM). The process of test development in SCLUnit illustrates general application development and testing principles which can be applied to applications written with other SAS products or in other development environments.

UNIT TESTING

Unit testing is the developer's testing, built around the underlying architecture of the application rather than the user interface, and distinct from the formalized and scripted user testing that is performed on the finished application. The unit tests focus on the parts of the application rather than the whole; unit tests are run during and throughout the development cycle and are not restricted to a separate testing phase (although a complete run of unit tests can also be part of a validation process).

DEFINITIONS

A **unit** is the lowest level at which you can reasonably test your code. In an object-oriented application written in SAS/AF or Java, this will often be a single class; in an application built in Base SAS® it could be a stored macro or set of macros, or a single callable block of SAS code.

A **unit test** is code that checks whether a particular unit does what it is supposed to do; the set of tests for a unit can test the operation of the unit more exhaustively than it could be tested as part of the overall application.

EXAMPLE

Assume your application is a clinical trials data management system and the unit is a stored macro used as part of a reporting module. Specifically, the macro will be invoked to populate a temporary data set with identifying data for all queries that you have sent to the clinical sites but for which you have not yet received replies.

Some of the things you could test for this macro:

- does it run without errors?
- does the temporary data set include all the records it should?
- do all the records included in the temporary data set represent queries that have been sent and have not been returned?
- do all the records excluded from the temporary data set represent queries that are not yet sent or that have already been returned?
- does it run correctly when all sent queries have been returned and when no queries have been sent?
- does it behave as expected when the temporary data set already exists?

By testing the macro separately from the reporting module as a whole, you can compare an expected result table to the source tables; an intermediate table is generally easier to check than a final saved report.

AUTOMATED TESTING FRAMEWORKS

For unit tests to be useful you have to run them frequently, so they have to be easy to run. Any manual step required is an obstacle that makes it that much easier to skip the test than to perform it. A unit testing framework provides an

environment for running tests automatically and includes tools to make it easier for you to set up, develop, and run your unit tests.

Even the relatively simple example above suggests a number of tasks that are likely to be commonly used across a wide variety of tests:

- defining necessary SAS libraries
- creating test data
- checking for SAS error conditions encountered when the code runs
- comparing table structures and observations within tables
- cleaning up after the test

Anything that prevents you from running the tests as often as possible makes it easier for code to go untested and errors to creep in. Having a good test framework and integrating the test development process into the application development process makes unit testing more effective.

TEST-FIRST PROGRAMMING

The concepts of automated unit testing and "test-first programming" were popularized by the Extreme Programming movement (Beck 2000; Wells 2006). The idea behind test-first programming is that you should write the unit test code before you write the application code that passes the test. Writing a method becomes a three-step process:

1. define the starting state, the inputs, and the corresponding results or output
2. create test code that sets up the starting state, sends the input or invokes the method, and checks that the output is correct; this test will initially fail because the method is not yet written
3. write the method that makes the test code run successfully

Strictly following the test-first sequence can be an austere discipline, especially when you are first implementing a new component. A friendlier alternative is to develop the methods and the test cases together and to have the implementation of the tests a necessary part of incorporating a new component into your application.

In either case, it is a good idea to maintain the test code and the functional code together as part of the code base of the application; for any build of the application you should be able to identify the associated test code and know that the tests have run successfully on the units incorporated in that build.

APPLICATION DESIGN FOR TESTING

The idea of designing your application architecture to support your testing plan sounds a bit like the proverbial tail wagging the dog. But a unit-test-friendly application architecture has advantages in addition to the benefits inherent in frequent and thorough testing.

- The code for the application is organized in distinct units.
- Units have clearly defined areas of responsibility.
- Units have specific and well-defined requirements, expectations, and interfaces (instantiation and method calls in the case of an object; a macro invocation in the case of a macro).
- Common tasks are implemented in units which other parts of the application can invoke as needed. This approach avoids repeated or duplicate code and improves maintainability; when the code needs to be changed or extended, you can focus on the appropriate unit rather than needing to make changes throughout the application.
- To the extent possible, user interface code is separated from the rest of the program logic.

THE APPLICATION OBJECT

In a SAS/AF application, you can create an object which is responsible for "running" the entire application. If the main interface is a frame from which the user invokes parts of the application from menus, toolbar buttons, or controls on the frame, all the command-processing code can be done by the application's main controller object, yielding. The simplest command-driven main frame code would look like:

```
INIT:
   declare MyApplication myApp=_new_ MyApplication();
```

```
  return;
MAIN:
   do ;
      declare Char thisCommand=word(1,'u');
      call nextCmd();
      if thisCommand ne '' then
         _frame_._sendEvent('Command Issued', thisCommand);
     end;
  return;
TERM:
   myApp._term();
  return ;
```

where the MyApplication object is responsible for initial setup of the application and for creating an event handler that will process the user commands.

An unexpected benefit is that having an application object can make your application “scriptable”; you can write an SCL program other than the main frame which invokes the application and sends it a command, or which invokes methods on other application objects. In effect, your test cases will be sample scripts that provide a model for programmatically controlling the parts of your application.

TESTING AS SPECIFICATION

The development of a test case can serve to clarify the specification and requirements of a feature; test case code can be an alternate or reverse-engineered version of the feature being implemented. A medical coding application might include a unit that takes a clinical data set, a specification, and a standardized dictionary and produces a coded data set in which verbatim terms in the clinical table are mapped to dictionary terms based on a combination of dictionary matching and rules developed by medical coding specialists.

Given a particular source table, standardized dictionary, and set of rules, the unit produces a coded table; the test code for that unit uses it to produce this coded table and verifies that it is consistent with the specified process by which dictionary matches and specialist-defined rules are to be applied. The logic of the test case resembles that of the unit itself and serves as an additional level of verification that it functions as required. It also provides a baseline so that if a portion of the unit needs to be rewritten—by adding caching routines to improve performance, for example—the test case verifies that the unit still produces the correct results.

JUNIT AND SCLUNIT

Among the earliest popular unit testing frameworks was JUnit, a unit testing framework written in Java and used with Java development (Object Mentor 2008). Because the test cases are an extension of the framework itself, the framework and the application are logically written in the same language; the term xUnit is used to refer to JUnit-style frameworks implemented in other languages.

SCLUnit is a SAS Component Language (SCL) and SAS/AF implementation of the xUnit framework; the object-oriented design and fairly simple class structure of JUnit make it easy to translate into other languages. Don Hopkins did the initial conversion of JUnit 3.2 to SCLUnit 1.0; I worked with Don to finish the system and enhance the interface for developing and running tests.

For Base SAS both the Framework for Unit Testing SAS programs (FUTS) and the Alert System provide xUnit-based frameworks using the SAS macro facility (Wright 2006; Hopkins 2004).

The following examples will specifically discuss SCLUnit, but the test case structure is highly similar across different implementations of the xUnit framework, particularly in an object-oriented environment like Java or SCL.

THE TEST CASE CLASS

The main structure for implementing tests in SCLUnit is the **TestCase** class. To create a set of tests, you create a class which is a descendent of the parent **TestCase** class; you implement the following methods specific to the unit you are testing:

- a set of one or public test methods whose names begin with the word “test” and which take no parameters
- a method named **setup()** which is run before each test method
- a method named **tearDown()** which is run after each test method

A test of a unit implementing a user authentication system might consist of:

- a **setup()** method which creates a test data set of application users
- a **tearDown()** method which deletes the test data set
- a **testValidUserAllowed()** method which makes sure that a valid user is approved by the user authentication system
- a **testBadUserDenied()** method which makes sure that the user authentication system does not approve an invalid user

When you run a specific test case class, the test framework first builds a list of the test methods, then iterates over the list. For each test method the framework creates a separate instance of the test case object and invokes the built-in **run()** method. For a single specified test method, the **run()** method calls **setup()**, then invokes the test method, and finally runs **tearDown()**. It returns to the framework an indication of whether the test method passed or failed.

Because each test method is run by a completely separate instance of the test case object, each test method will run independently of other test methods and other test cases.

HOW IT WORKS: REFLECTION

Like JUnit, SCLUnit works by taking advantage of a feature called **reflection**. In a programming environment like Java or SAS/AF, **reflection** refers to the ability to make a run-time determination of the nature and capabilities of objects and classes, and to write code with method invocations for which the specific method name can be dynamically set at run time.

SCLUnit invokes the **_getMethods()** method of the Class metaclass (SASHelp.FSP.Class) to determine the test methods for a given test case class. After the names of the test methods have been determined, SCLUnit uses the venerable CALL SEND routine to invoke the test methods.

ASSERTIONS, FAILURES, AND ERRORS

How does an individual test method pass or fail? In addition to the built-in methods for running the tests, the TestCase class has a series of built-in assertion methods. When you call an assertion method, the specified condition is checked. If the assertion is valid the test case continues running; if the assertion is not valid, the test case halts and reports a **failure**.

The test framework also wraps the test method's execution in an exception-handling block, and it includes an implementation of the program halt class to generate exceptions in the event of run-time errors. If a run-time error is encountered, the test case will halt and report an **error**; for a test to pass, all assertions must be correct and the code must run without errors.

SCLUnit implements the following assertion methods:

assertTrue(*theExpression***)**: halts the test case and fails if the expression provided does not resolve to "true" (a nonzero value). If your expression should resolve to "false," use **assertTrue(**not (*theExpression*)**)** to reverse the assertion.

assertEquals(*expectedValue*, *actualValue***)**: halts the test case and fails if the expected value and the actual value are not equal. The **assertEquals()** method is overloaded to work with either a pair of character values or a pair of numeric values.

assertEqualsIgnoreCase(*expectedValue*, *actualValue***)**: a variation on **assertEquals()** which allows case-insensitive comparison of character values.

assertNotNull(*theObject***)** or **assertNotNull(***theList***)**: halts the test case and fails if the object or list is undefined (null).

assertNull(*theObject***)** or **assertNull(***theList***)**: halts the test case and fails if the object or list is defined (non-null).

assertSame(*expectedObject*, *actualObject***)** or **assertSame(***expectedList*, *actualList***)**: halts the test case and fails if the object or list identifiers are not identical.

Each **assert…()** method has an additional optional parameter in the final position that can contain a message to explain what it means when the assertion fails.

So for the user authentication system defined above, you might have a test case class that looks like:

```
Class MyApp.MainAppTest.UserAuthenticationModuleTest.Class
```

```
   Extends SCLUnit.SCLUnit.TestCase;

* Attributes ;
Private UserAuthenticationModule userAuth / (AutoCreate='N');

* Method definitions ;
setup: Public Method / (state='O');
   _super();
   /* code to create test table of valid users */
   /* USERA and USERB are valid, USER1 and USER2 are not */
  endMethod;

tearDown: Public Method / (state='O');
   if listLen(userAuth) gt 0 then userAuth._term();
   /* code to delete test user table */
   _super();
  endMethod;

testValidUserAllowed: Public Method;
   userAuth = _new_ UserAuthenticationModule();
   assertNotNull(userAuth, 'Instantiate failed');
   assertTrue(userAuth.userIsValid('USERA'),
      'Valid user mistakenly denied');
endMethod;

testBadUserDenied: Public Method;
   userAuth = _new_ UserAuthenticationModule();
   assertNotNull(userAuth, 'Instantiate failed');
   assertTrue(not (userAuth.userIsValid('USER1')),
      'Bad user mistakenly accepted');
endMethod;
```

This test would pass if it ran without errors and if all assertions were successful. It would report a failure if an assertion was not satisfied—if USERA were mistakenly left out of the test user table, causing **userIsValid()** to return false instead of the expected true. It would encounter an error if an unexpected halt occurred—if user table creation failed because the library referred to a read-only folder.

OBTAINING AND RUNNING SCLUNIT

The most recent version of SCLUnit is version 3.0, a complete rewrite of the application under SAS 9.1 which was finished in January 2008. A copy of SCLUnit may be obtained from the author or from the SCLUnit page at SASCommunity.org.

SCLUnit consists of two SAS catalogs, **SCLUnit** (which contains the main testing framework) and **TestRunner** (which contains the interface for running tests). To install, store the catalogs in a convenient location and define the SAS library SCLUNIT to point to the folder with the catalogs.

The following AFAPPLICATION (AFA) command invokes the SCLUnit test running screen:

```
afa c=SCLUnit.TestRunner.UI.frame
```

You can add a button to the SAS toolbar to make SCLUnit available with a single click.

RUNNING TESTS

Running tests in the SCLUnit framework is straightforward, which makes it easy to test continuously during development. Invoking SCLUnit brings up a screen (figure 1) which allows you to select a test; you then click on a traffic-light button to execute the selected test case.

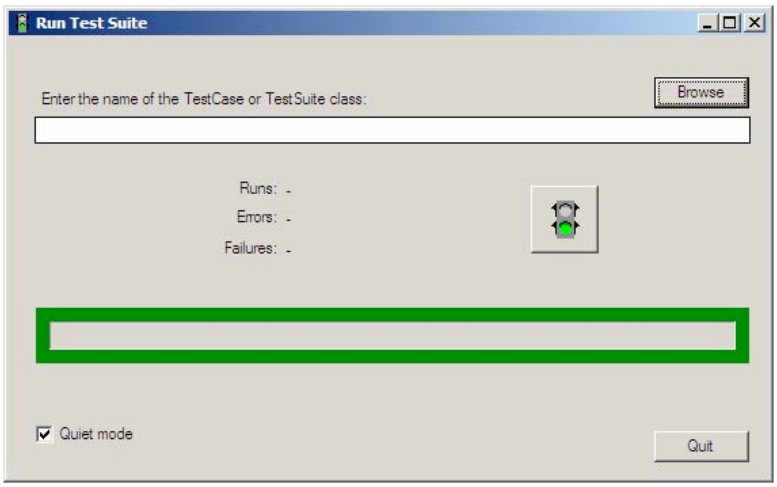

Figure 1: The Main SCLUnit Interface

As the test case runs it updates the progress bar on the frame; when the test is complete the frame presents the counts of passed tests, failed tests, and run-time errors encountered. The borders and traffic light control are colored green if all tests are passed and red if any failures or errors are encountered. Figures 2 through 5 show the SCLUnit interface while running and after completing both successful and unsuccessful tests.

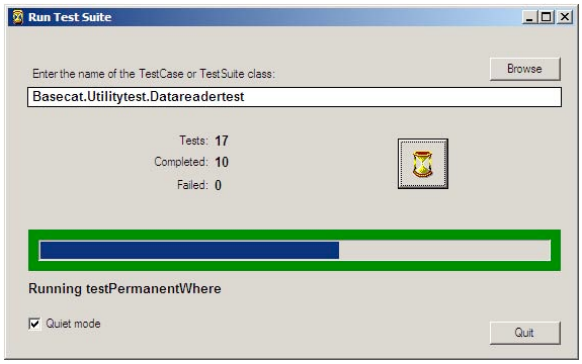

Figure 2: Test run in progress, OK so far

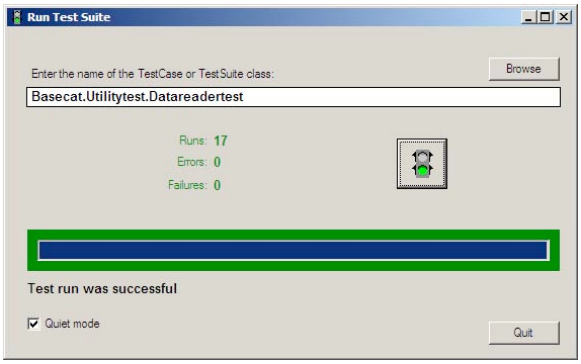

Figure 3: Test run completed successfully

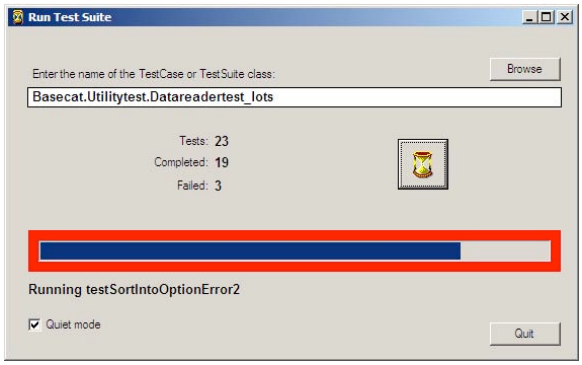

Figure 4: Test running with errors encountered

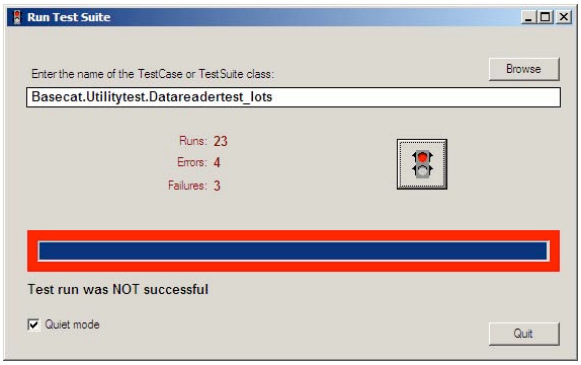

Figure 5: Test completed, errors and failures

Green is good, and red is trouble: if your tests are running successfully all the indicators stay green, and when something goes wrong the bright red catches your attention.

The ease of running tests in a framework like SCLUnit means unit testing can be a part of your normal code modification cycle. The ideal is to keep testing continuously through the development process. If you make a change that has a side effect of breaking another part of your software, continuous testing will alert you to the issue almost immediately. You will know which incremental change caused the problem to appear, and you can understand and address the problem quickly because the relevant code is fresh in your mind.

REPORTING RESULTS

During day-to-day development, the on-screen indicators of the SCLUnit or JUnit interface let you know that your tests are working. If you are developing validated software and need to include your automated test results in a validation package to document the testing, you will need more than a screen grab of a green-bar result.

SCLUnit extends the JUnit test-running framework by using the SAS log to write more extensive information about the tests that you run. In "quiet" mode, the log will contain the name of the test case classes and test methods that run; in verbose mode every assertion—passed or failed—will be written to the log.

EXTENDING THE TESTCASE CLASS

Each specific test case object is a descendent of the original **TestCase** class; you can provide application-specific extensions by creating a subclass of **TestCase** that implements those extensions and defining your test cases as instances of that subclass. So instead of defining a test case directly from the framework:

```
Class MyApp.MainAppTest.UserAuthenticationModuleTest.Class
      Extends SCLUnit.SCLUnit.TestCase;
```

you can define a subclass with methods that extend the test case:

```
Class MyApp.TestingTools.MyTestCase.Class
      Extends SCLUnit.SCLUnit.TestCase;
```

and then define your specific test cases based on that subclass:

```
Class MyApp.MainAppTest.UserAuthenticationModuleTest.Class
      Extends MyApp.TestingTools.MyTestCase;
```

Table comparisons can be simplified with extensions to the test case. For tables accessed through the OPEN() function, you can implement the following methods in a subclass of **TestCase** to replace multiple assertions with a single method call:

checkVariableDefinitionsMatch(*dataSetID1*, *dataSetID2*, *variableName*): checks that the specified variable is present and identically defined in both tables.

doubleFetch(*dataSetID1*, *dataSetID2*): retrieves the next row from both tables simultaneously; returns true if there is a next row, false if both tables return end-of-file, and fails if one table returns end-of-file while the other returns a next row.

checkRowValuesMatch(*dataSetID1*, *dataSetID2*): compares all values in the current row of both tables; fails if any values differ.

Building upon those methods, the following two methods incorporate the built-in OPEN() and CLOSE() functions and take table names as arguments:

checkStructuresMatch(*dataSetName1*, *dataSetName2*): opens both tables and checks that they have the same columns, identically defined.

checkDataSetsMatch(*dataSetName1*, *dataSetName2*): opens both tables, checks that the structures match, checks that the number of observations match, and then retrieves rows and checks that the data values match.

The KEEP=, DROP=, and WHERE= data set options are powerful tools in setting up table comparisons. If you want to check that the variables ID and KEY are identically defined in two tables while ignoring all other variables, you can invoking the **checkStructuresMatch()** method using the KEEP= data set option:

```
checkStructuresMatch('work.TestData1(keep=ID Key)',
                     'work.TestData2(keep=ID Key)');
```

Similarly, you can ignore specific variables or specific observations. If you want to compare the audit trail generated during a test to an expected set of audit trail entries while excluding the variable containing the record timestamps, you can use DROP=:

```
checkDatasetsMatch('work.expectedAuditTrail(drop=ModifiedDT)',
                   'auditLib.theAuditTrail(drop=ModifiedDT)');
```

If you want to check only the records in the audit trail generated by the test user, you can filter using WHERE=:

```
checkDatasetsMatch('work.expectedAuditTrail',
   'auditLib.theAuditTrail(where=(UserID="TESTUSER"))');
```

TEST SUITES

A **TestSuite** object allows you to run multiple test cases in sequence; running the suite will provide counts of passed and failed tests across all contained test cases. Test suites are good for testing related components; a suite containing all tests in your application allows you to test everything in a single step.

The SCLUnit user interface shown above allows you to run both test cases and test suites; you can select a subclass of **TestCase** or of **TestSuite** and the framework will run all associated test methods.

HOW IT WORKS: INTERFACES AND RECURSION

A test case is a single class which extends the built-in **TestCase** class of SCLUnit. A test suite is an extension of the built-in **TestSuite** class, a separate part of the framework which runs tests by iterating over test methods within a single test case, over multiple test cases, and over multiple test suites.

Test cases and test suites are implemented by using an **interface**, which is a specific collection of method names and signatures. If a class **supports** an interface, that means that the class implements all the methods that are part of that interface. (SAS uses the keyword “supports” while Java uses “implements”.) Both **TestCase** and **TestSuite** support an interface named **Test**—and **TestSuite** represents a collection of objects which support the **Test** interface.

A test suite can then runs tests on each object in its collection without needing to know whether the specific object is another test suite or an individual test case. If the object is another test suite, then when the parent suite is run it will recursively invoke the tests on the child suite.

CREATING A TEST SUITE

You create a test suite by defining a class which extends the built-in **TestSuite** class by adding a method named **suite()**. This method creates **TestSuite** objects for all the test case classes you want to combine into the suite. The following sample code creates a test suite named **DataSetSuite**:

```
Class MyApp.UtilityTest.DataSetSuite
      Extends SCLUnit.SCLUnit.TestSuite;

   * Method definitions ;
   suite: Public Method
      return = TestSuite;
      declare TestSuite suite = _new_ TestSuite();
      suite.addTest(
         _new_ TestSuite(loadClass("MyApp.UtilityTest.DataCreatorTest")));
      suite.addTest(
         _new_ TestSuite(loadClass("MyApp.UtilityTest.DataReaderTest")));
      suite.addTest(
         _new_ TestSuite(loadClass("MyApp.UtilityTest.DataUpdaterTest")));
      suite.addTest(
         _new_ TestSuite(loadClass("MyApp.UtilityTest.DataUtilityTest")));
      return suite;
     endMethod;
EndClass;
```

When you run **DataSetSuite**, it will run all the test methods in the four test case classes that are specified in the class definition.

YOUR DEVELOPMENT AND TESTING ENVIRONMENT

The advantage to an automated testing framework is that it makes testing easier to do. When tests are easy to run, you will test more frequently and more consistently.

STORING TEST CODE

A common practice with test-driven development is to keep the test code alongside the main application code. For each catalog containing one or more class entries, create a test catalog in the same folder. If your application’s classes are stored in **MyApp.Utility**, store the associated test cases in **MyApp.UtilityTest**.

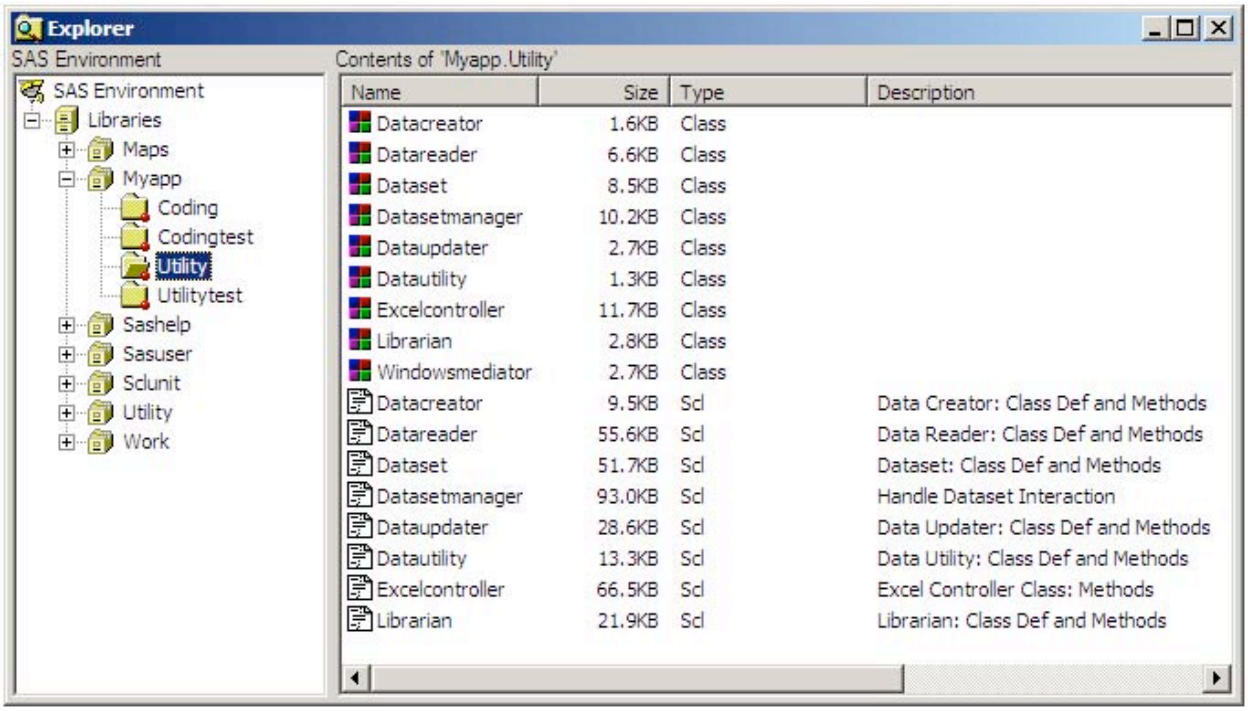

Figure 6: Application code is…

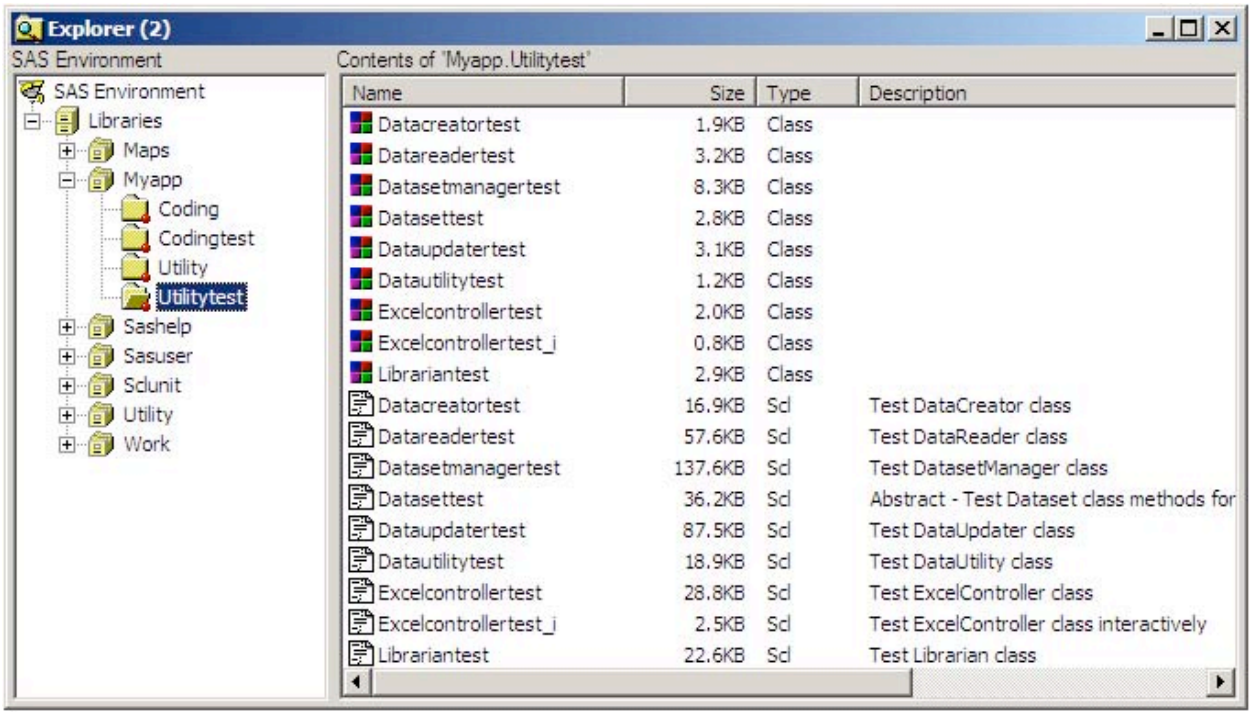

Figure 7: …stored alongside test code

Similarly, test class names are derived from the name of the class they test. So to test **MyApp.Utility.DataReader**, create the test case **MyApp.UtilityTest.DataReaderTest**.

THE TEST FACTORY OBJECT

Applications depend on a great deal of state, while testing is ideally stateless. Your test cases need to create not just their own test data but often a directory structure in which everything will go. Another useful application-specific extension to the test framework is a "factory" object to provide the structures needed for the application to run. To test a data management system, the factory object:

- identifies a temporary directory in which to create test folders, files, and tables
- creates a project folder structure in the temporary directory that matches the folder structure used by the application
- creates and populates the various data sets used by the application as requested by the test case
- deletes all temporary data sets and folders at termination

The factory object can be instantiated as needed by any test case.

ADDITIONAL TESTING CONSIDERATIONS

TESTING EXCEPTION HANDLING

You can test situations in which a unit is supposed to throw exceptions by handling the exception in the test case code. If you expect the method **doThis()** of the object testObject to throw an exception of class **MyException**, you can address this with a test case:

```
do ;
   declare MyException myEx ;
   testObject.doThis() ;
   assertTrue(0, 'Exception not thrown') ;
   catch myEx ;
      assertTrue(1, 'Catch block reached') ;
   endCatch ;
  end ;
```

If the exception is not thrown, calling **assertTrue()** on a value of zero will cause the test to fail.

ABSTRACT TEST CASES AND TEST CASE INHERITANCE

The class structure of your test cases can resemble the class structure of your applications. If you have an abstract class, you can create an associated abstract test case class descended from the **TestCase** class and containing methods that will be used to test the classes descended from the abstract class.

For example, if an abstract **AuditTrail** class is the ancestor of specific **KeyingAuditTrail** and **MasterAuditTrail** classes, you can implement common methods for testing in an abstract **AuditTrailTest** class. Then the **KeyingAuditTrailTest** and **MasterAuditTrailTest** test classes can be defined as descendents of the **AuditTrailTest** class.

Similarly, if you have a **DataUpdater** class that accesses a table in update mode and is a subclass of a **DataReader** class that uses input mode to access the table, you can create the **DataUpdaterTest** class as a descendent of the **DataReaderTest** class; when you run tests for the child class, all the tests of the parent class will also run.

INTERACTIVE TESTS

Often you need to test cases in which your application requires some kind of user input or response. Such tests may be necessary to fully test a unit while making up only a small portion of the unit's functionality. You face a choice: either require user interaction while your tests run (which will making running the tests a longer and more tedious task), or exclude the interactive parts of the unit from the testing process (which will reduce your test case's coverage).

Test case inheritance provides a way to work around this issue. Assume your user authorization module has a **deleteUser()** method to remove an authorized user, and that if the method is called for the current user it presents a message box indicating that you cannot delete yourself and leaves the authorized user list unchanged.

When you create the test class, include a method which performs the test and which would require the user to dismiss the message box, but give that method a name which does not begin with the word "test". Because the name does not begin with "test", the method will not be treated as a test method when the test case is executed.

```
Class MyApp.MainAppTest.UserAuthenticationModuleTest
      Extends SCLUnit.SCLUnit.TestCase;
[...]
   checkDoNotDeleteSelf: Public Method ;
      /* create userAuth and get current userID */
      assertTrue(userAuth.userIsValid(currentUser));
      userAuth.deleteUser(currentUser);
```

```
      * Verify that the user has not really been deleted ;
      assertTrue(userAuth.userIsValid(currentUser));
   endMethod ;
[...]
```

Now you can define an interactive version of the test class which extends the primary version and adds a properly-named test method that simply calls the method defined on the parent class:

```
Class MyApp.MainAppTest.UserAuthenticationModuleTest_I
      Extends UserAuthenticationModuleTest;

   * Method definitions ;
   testDoNotDeleteSelf: Public Method ;
      checkDoNotDeleteSelf() ;
     endMethod ;
EndClass  ;
```

You can choose whether to run **UserAuthenticationModuleTest** (the non-interactive version) or **UserAuthenticationModuleTest_I** (the interactive version), but the actual test code is implemented in the original **UserAuthenticationModuleTest** class. Inheritance lets you create two almost-identical classes so you can choose which meets your needs in a particular round of testing.

BUILDING ON EXISTING OBJECTS

Particularly if your application includes utility objects, the test case for an application component can use other components from the application to set up test conditions. If you have a unit that is responsible for defining the SAS libraries used by the application, the test cases for units that access data in those libraries can call the application's library-defining code to allocate libraries for the test data.

TESTING MODELS

If you develop subclasses of SAS-provided models for use in frame components, the model classes can be instantiated separately from the visible viewer components. You can develop a test case in the SCLUnit framework that will instantiate and test the methods of your customized model class without requiring the presence of a frame or a viewer object.

OBJECT TERMINATION

Because a test method halts when an assertion is not satisfied, test code must be careful to terminate objects that it instantiates even when the test fails. In the example test case above for the user authentication module, the object userAuth is not declared locally inside a test method but is instead defined as a class attribute of the test case class. This allows the **tearDown()** method to call **_term()** to terminate the object; since **tearDown()** runs after each test, pass or fail, this guarantees that the object is properly terminated.

Sometimes you may be uncertain whether an object was successfully instantiated or whether it was previously terminated; invoking **_term()** on a nonexistent object causes the calling program to halt. The solution to this problem is that every object identifier also functions as the identifier of a nonempty SCL list. The LISTLEN() function returns the length of a list; and will return -1 if the list is undefined. So the line:

```
if listLen(userAuth) gt 0 then userAuth._term();
```

will terminate the userAuth object if it exists but will not call **_term()** unnecessarily.

LESSONS FROM TESTING

Automated testing is a particularly good tool for discovering certain kinds of problems with an application. Some of the things that testing has been useful in locating include:

- Cases in which objects are not properly terminated. Garbage collection routines will eventually dispose of objects, but explicitly terminating an object when you are done with it is more efficient.
- Cases in which you generate an additional instance of an object instead of re-using an existing instance, particularly when the AUTOCREATE option is improperly set for a class attribute that is a specific object type.

CONCLUSION

Automated unit testing can play a major role in the development and validation of robust applications. Unit testing frameworks simplify test development and help you build testing into your application development cycle.

REFERENCES

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

Hopkins, Don. 2004. “Sounding the Trumpet: Effective Failure Notification.” *2004 SESUG Proceedings*, Nashville TN.

Object Mentor. 2008. “JUnit: Resources for Test Driven Development.” <http://www.junit.org/>.

Wells, Don. 2006. “Extreme Programming: A Gentle Introduction.” <http://www.extremeprogramming.org/>.

Wright, Jeff. 2006. “Drawkcab Gnimmargorp: Test-Driven Development with FUTS.” *Proceedings of the 31st Annual SAS Users Group International Conference*, San Francisco CA, 004-31.

ACKNOWLEDGMENTS

SCLUnit was developed at Rho, Inc. to support development of the Rho Clinical Trials Data Management System (RhoDMS).

Don Hopkins initially developed version 1.0 of SCLUnit using JUnit 3.2 and helped build SCLUnit into a mature testing tool. The current SCLUnit version 3.0 is a large-scale rewrite of the earlier versions.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David A. Scocca
Rho, Inc.
6330 Quadrangle Drive, Suite 500
Chapel Hill, NC 27517
E-mail: Dave_Scocca@RhoWorld.com