

Paper 004-31

Drawkcab Gnimmarginop: Test-Driven Development with FUTS

Jeff Wright, ThotWave Technologies LLC, Cary, NC

ABSTRACT

One of the practices of Extreme Programming is Test-Driven Development (TDD), also known as Test-First Design. This style of development emphasizes an approach that is backwards to many programmers: writing tests before working code. Interestingly, this approach has found application in both agile methodologies and validation-intensive programming environments. TDD involves creating automated and repeatable unit tests, typically using a testing framework such as JUnit for Java and NUnit for Microsoft .NET languages. This paper will describe FUTS, a Framework for Unit Testing SAS® programs in the spirit of JUnit and similar tools. Unit testing with FUTS is described, and an example of TDD in SAS is presented. In addition, pragmatic advice is given for dealing with challenges that commonly occur with automated testing.

INTRODUCTION

One aspect of the universe of Foundation SAS® users that is interesting is how there are two distinct, enthusiastic communities: users of SAS for individual analytics, and users of SAS to create production applications.

Software development with SAS is the concern of this paper, which is thus aimed squarely at the second group of people. One of the key differences between writing SAS code for individual analysis versus writing a production application is testing. There is just an entirely different level of robustness required for an application that serves a collection of (possibly technology-phobic) users, versus a program that is only used by its author. This paper introduces FUTS, a Framework for Unit Testing SAS programs. It will begin by examining the JUnit framework for unit testing in Java, which is the inspiration for FUTS. Similar to JUnit, FUTS contains a library of assertions and a way to organize and run test programs. Writing and running tests in FUTS will be explained.

In previous papers (Nelson 2004a, Nelson 2004b), we described the assertion library for FUTS and explained its use in production code for testing for error conditions. This paper goes into more detail on *Test-Driven Development* (TDD), which involves writing dedicated test programs. A brief example of TDD with FUTS will be presented to give a flavor for how TDD can work in a data-intensive application, such as most SAS programmers encounter.

Automated testing does not mean testing happens automatically. Certain challenges tend to arise in creating automated tests. We will inventory those challenges and describe practical strategies for dealing with them.

Finally, some concluding thoughts will be offered over where FUTS and TDD are, in our humble opinion, valuable.

ThotWave expects to release FUTS under an open source license by the time you read this paper. Please see our website, www.thotwave.com, for more information.

AUTOMATED TESTING FOR LANGUAGES BESIDES SAS

Most of the attention on automated unit testing and test-driven development has been spurred by the JUnit test framework for Java (<http://www.junit.org>). JUnit provides a way to write individual test cases and suites of tests to run Java code and verify its results. As reflected in the name, JUnit was intended to support *unit testing*. Unit testing may be defined as "a process of testing the individual subprograms, subroutines, or procedures in a program" (Myers 1979). Since unit testing is frequently performed by the developer herself, JUnit was written in Java for Java programmers.

Here is a sample JUnit test method from an introductory tutorial (Beck):

```
1 public void testEquals() {
2     Money m12CHF= new Money(12, "CHF");
3     Money m14CHF= new Money(14, "CHF");
4
5     Assert.assertTrue(!m12CHF.equals(null));
6     Assert.assertEquals(m12CHF, m12CHF);
7     Assert.assertEquals(m12CHF, new Money(12, "CHF"));
8     Assert.assertTrue(!m12CHF.equals(m14CHF));
9 }
```

Let's walk through this sample to see what is happening. This fragment of code defines a Java method named *testEquals* that tests the *equals()* method of a class named *Money*. Defining an *equals()* method to check for equivalence is a standard practice for Java classes. To prepare for the tests, two *Money* objects are created with values of 12 CHF (Swiss Franks) and 14 CHF in lines 2 and 3. Lines 5 through 8 represent tests on these objects. The *Assert* class is a utility supplied with JUnit so that test results can be checked. The assertions in the above code fragment verify that the *equals()* method in the *Money* class works as intended. For example, the test in line 7 asserts that the object named *m12CHF* equals a new object with an amount of 12 CHF.

An assertion describes a condition that should be true. If the condition is not true, some sort of error is generated. In the case of JUnit, the test is marked as failing. JUnit provides a variety of assertions, such as two objects are equal, two objects are the same object, a reference is null or not null, or some expression evaluates to *true*.

A method written like this is a single test in JUnit. JUnit provides a way to organize individual tests into suites, execute a suite, and report on results. A test is considered to pass if no errors are generated and no assertions fail. After executing a suite, JUnit reports how many tests were run, how many tests had failed assertions, and how many tests generated errors (threw exceptions).

You can imagine how a conscientious Java developer can write test cases to exercise the code she is writing. Because these tests are automated, they can be run at any time to verify that the code is still working the way it used to (*regression testing*). Later in this paper we will explore the concept of using a test framework more proactively in development: test-driven development.

Because of the popularity (and simplicity!) of JUnit, not to mention it being licensed as open source, testing enthusiasts have created work-alike unit testing frameworks for a large number of programming environments (see <http://www.xprogramming.com/software.htm>). For Microsoft .Net, there is *NUnit* (<http://www.nunit.org/>). Besides opening up automated unit testing to another realm of programmers, NUnit illustrates that unit testing tools should leverage the facilities and idioms of the programming language that they support. So while the object model and assertion library in NUnit are highly similar to JUnit, NUnit provides support for things that are unique to the .Net environment such as attributes and assemblies.

Another testing tool that is worth mentioning is *Fit* (<http://fit.c2.com/wiki.cgi?IntroductionToFit>). Fit is similar to JUnit in that it is an open source tool for automated testing, but the emphasis is quite different. Fit is intended to address acceptance testing rather than unit testing. The web-based interface to Fit is all about allowing users to specify expected results and review actual test results. This is done in a table format that is easy for non-technical people to understand.

Naturally there are commercial testing tools as well. These tools tend to address different aspects of testing, such as record/playback GUI testing and load testing. Those areas are outside the topic of this paper.

To summarize, automated unit testing frameworks are tools that allow software developers to write their own tests to verify their own code. The two main components of a testing framework are:

- An assertion library for checking expected results
- Some way to organize and execute tests

FUTS DESCRIBED

Inspired by the tools described in the previous section, ThotWave has created FUTS, a Framework for Unit Testing SAS programs. This begs the question: in the SAS world, what does *unit* mean? Programming languages usually provide a way to break down a complex task into manageable chunks. This process is called *modularization*. Base SAS provides at least two ways to do this:

- A separate file of SAS code can be referenced from another program using a %INCLUDE statement.
- A SAS macro allows a block of SAS code to be called repeatedly, including logic capabilities such as parameters, conditions, and loops.

So a unit in SAS might be a %INCLUDE file or a macro. We can also use the term *module* as a generic reference for a chunk of SAS code. FUTS is designed to enable test cases for SAS modules, both %INCLUDE files and macros.

In comparison to languages such as Java, a significant distinguishing characteristic of SAS programs is that they are *data-intensive*. SAS programs typically crunch through large amounts of data, perhaps in complex and arcane structures, and perhaps creating more data as output. FUTS is designed to support that reality.

WRITING FUTS TESTS

A FUTS test is just a SAS program that uses assertions to check that some module of SAS code is working as intended.

Let's start with an example. Part of the FUTS library is a macro %obs that returns the number of observations in a data set. Here is a simple test for the %obs macro:

```
1 data obstest;
2     x=1; output;
3     x=2; output;
4 run;
5 %let NOBS = %obs(obstest);
6 %assert_equal(2, &NOBS)
```

The DATA step starting on line 1 sets up a WORK data set with two observations. In line 5 the %obs macro is called, and its return value is assigned to &NOBS. Line 6 contains an assertion that &NOBS should be 2. That's all there is to it!

In general, a test case will:

- Set up any required input data.
- Call the SAS module being tested.
- Make one or more assertions about the results of the module.
- Tear down any input or output data that could interfere with other tests. This is to undo the *set up* that we did at the start of the test. It is not always necessary to undo our set up, but sometimes it is important to insure a clean start for other tests, or to not be wasteful of resources like disk space.

When an assertion turns out to be false, we say that it generates an event. These events are important to us, because they mean that a test has failed. How do you know if a failure occurred? FUTS uses the standard SAS log convention that a message starting with "ERROR" indicates that something has failed. So a passing test is one that produces a log without any ERROR messages.

The library that comes with FUTS includes the following macros:

| Macro | Description |
|---------------------------|--|
| %assert_compare_equal | Generates an event if two data sets are not the same, as compared by PROC COMPARE. |
| %assert_empty | Generates an event if a data set has any observations; no event if data set does not exist. |
| %assert_equal | Generates an event if two arguments are not equal. |
| %assert_exist | Generates an event if a data set does not exist. |
| %assert_fexist | Generates an event if a file identified by fileref does not exist. |
| %assert_filecompare_equal | Generates an event if two text files identified by name are not equal. |
| %assert_fileexist | Generates an event if a file identified by name does not exist. |
| %assert_not_compare_equal | Generates an event if two data sets are the same, as compared by PROC COMPARE. |
| %assert_not_empty | Generates an event if a data set has no observations, or if the data set does not exist. |
| %assert_not_equal | Generates an event if two arguments are equal. |
| %assert_not_exist | Generates an event if a data set does exist. |
| %assert_not_fexist | Generates an event if a file identified by fileref exists. |
| %assert_not_fileexist | Generates an event if a file identified by name exists. |
| %assert_not_null | Generates an event if the argument is null (has zero length). |
| %assert_not_zero | Generates an event if the argument is zero. |
| %assert_null | Generates an event if the argument is not null (has non-zero length). |
| %assert_sym_compare | Generates an event if the arguments do not meet the comparison criteria. |
| %assert_zero | Generates an event if the argument is non-zero. |
| %exist | Verifies the existence of a SAS data set. |
| %expect_error | Registers an expected error type, so that an assertion event will not generate a notification. |
| %fexist | Verifies the existence of a file by fileref. |
| %fileexist | Verifies the existence of a file by name. |
| %generate_event | Generates a notification that an assertion has failed. |
| %obs | Returns the observation count for a data set. |

RUNNING FUTS TESTS

Since a FUTS test is just a SAS program that makes calls to a macro library, you can use whatever is your preferred SAS programming environment for writing and running tests. In this respect, FUTS is compatible with Display Manager, Enterprise Guide, or batch submit. All you have to do is to check the log file for the absence of ERROR messages -- something you should already be doing!

However, if you are making use of FUTS to write a large number of tests for a set of SAS modules, you will probably want a way to run all of your tests in batch. FUTS includes a perl script, `futs.pl`, for this purpose. You can point `futs.pl` to your source code, and it will locate all your tests, run them one by one, and report how many tests were executed and how many failed.

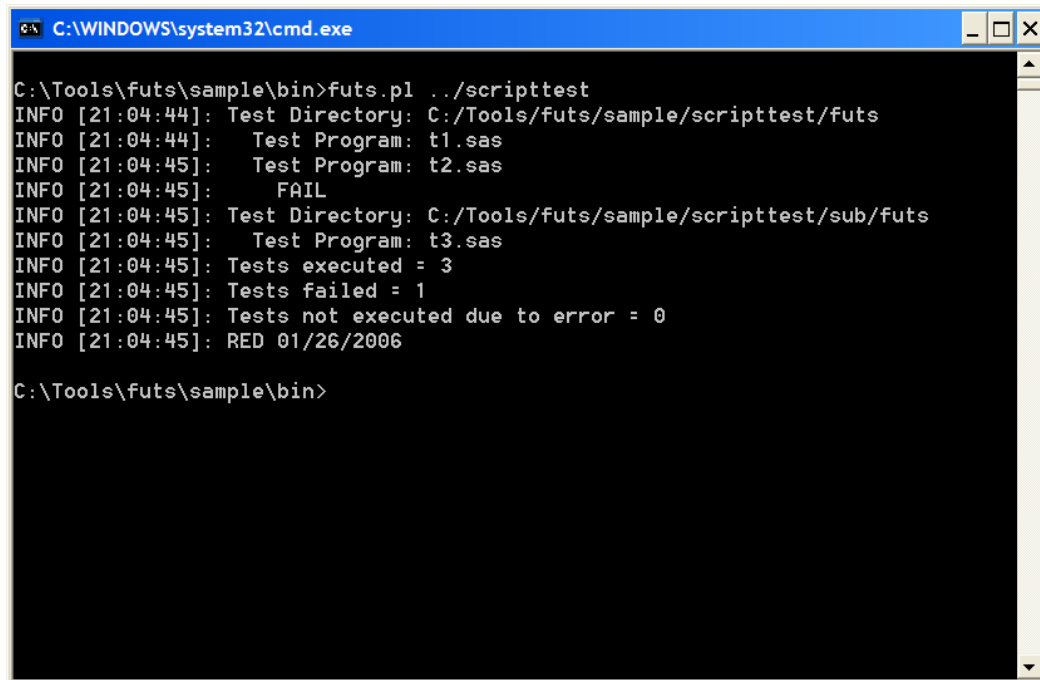
FUTS uses a directory naming convention to find your test cases. Any SAS program in a directory named *futs* is expected to be a test case. In a large project, there may be an entire tree of SAS source code with many directories. A typical practice would be to create a *futs* subdirectory under each directory containing SAS code. If you pass `futs.pl` the top-level directory for the project, it will identify every *futs* directory in the tree.

There is one special rule about finding test cases. If there are files named *setup.sas* or *teardown.sas* in a *futs* directory, these files will not be run as test cases. Instead, these files will be run before and after each test case. This is to allow you to put in reusable code to set up and tear down any test data for your test cases.

The `futs.pl` script expects two environment variables to be present, in order to tell it how to run SAS:

- **SASROOT:** The `SASROOT` environment variable should be set to the SAS installation directory that contains the SAS executable.
- **SASOPTS:** The `SASOPTS` environment variable should contain any desired command line options for running SAS. For example, you probably want to include the `-config` option tell SAS which config file to use for system options for your project. The `futs.pl` script will also add some command line options to SAS for you that are required for batch execution.

When `futs.pl` runs, it produces a running log of directories and test cases. At the end, it summarizes the number of test cases run, failed, and failed to execute due to some system problem. It also gives an overall status of either GREEN (all tests passed) or RED (some failure occurred). This is a nod to the GUI test runners for JUnit and NUnit which produce a colored progress bar to summarize test status.



```

C:\WINDOWS\system32\cmd.exe

C:\Tools\futs\sample\bin>futs.pl ../scripttest
INFO [21:04:44]: Test Directory: C:/Tools/futs/sample/scripttest/futs
INFO [21:04:44]: Test Program: t1.sas
INFO [21:04:45]: Test Program: t2.sas
INFO [21:04:45]: FAIL
INFO [21:04:45]: Test Directory: C:/Tools/futs/sample/scripttest/sub/futs
INFO [21:04:45]: Test Program: t3.sas
INFO [21:04:45]: Tests executed = 3
INFO [21:04:45]: Tests failed = 1
INFO [21:04:45]: Tests not executed due to error = 0
INFO [21:04:45]: RED 01/26/2006

C:\Tools\futs\sample\bin>

```

Also, the exit status of the `futs.pl` script is zero when all tests pass, or non-zero when tests fail. This feature is to enable scheduled execution of the test suite with notification if any tests failed.

TEST-DRIVEN DEVELOPMENT

At this point you may be thinking, "This is all interesting, but I'm not actually a software tester, I'm a developer." Now we turn our attention to *Test-Driven Development* (TDD).

TEST-DRIVEN DEVELOPMENT CASE STUDY

Test-Driven Development reverses the normal process of programming. Instead of writing some code and testing it, in TDD you begin by writing a test, then write the code to make the test pass. This backward approach is the inspiration for the *Drawkcab Gnimmarginop* title for this paper. Interestingly, this test-driven methodology is most popular among the practitioners of Extreme Programming (XP), which is more widely known for informality than formality.

TDD is best understood by walking through a case study. Suppose we have been asked to create a report on the Residential Energy Consumption Survey conducted by the U.S. Department of Energy (<http://www.eia.doe.gov/emeu/recs/recs2001/publicuse2001.html>). This survey has produced data files describing households and their use of electric power, natural gas, and heating oil. In this case, we want to present average

electric power consumption by census region and number of bedrooms per house. The following screen shot shows the report that we will ultimately produce:

| Census Region | Number of Bedrooms | Avg Power Consumption (kwh) |
|---------------|--------------------|-----------------------------|
| Midwest | 0 | 3,054 |
| | 1 | 5,091 |
| | 2 | 7,733 |
| | 3 | 10,336 |
| | 4 | 13,084 |
| | 5 | 14,180 |
| | 6 | 25,285 |
| Northeast | 0 | 4,762 |
| | 1 | 4,039 |
| | 2 | 6,160 |

For this assignment, we will be using the coding standards for our organization, which dictate things like project directory structure and coding style. Our coding standards state that we should modularize reports by breaking them down into files that are called with `%include`. That will work well with TDD, which encourages us to program in small increments.

Our first step is surely to bring the source data into SAS. The RECS data that we need is in two text files in a comma-separated (CSV) format. The files come with codebooks that explain the variables in each file. We decide to start with `datafile2001.txt`, which contains the basic household profile.

We decide that our module to bring the data into SAS will be called `get_data.sas`. We already know that the data has 4,822 records, so we write the following as our first test:

```
1 %include "&SASAPP/reporting/include/get_data.sas";
2 %let EXPECTED_OBS_COUNT = 4822;
3 %assert_exist(housing_unit)
4 %let NOBS = %obs(housing_unit);
5 %assert_equal(&EXPECTED_OBS_COUNT, &NOBS)
```

This test calls our not-yet-written `get_data` module (line 1) and asserts that we expect it to create a WORK data set `housing_unit` (line 3) that will have 4,822 observations (line 5).

Now that we have a test, we need to write the code to make the test pass. We remember that we have used PROC IMPORT to read CSV files on some past projects, so we dig up some of that old code, and after a little copy and editing we create our `get_data.sas` with the following:

```

1 proc import datafile="&DROP_DIR/datafile2001.txt"
2   out=housing_unit
3   dbms=csv
4   replace;
5   getnames=yes;
6 run;

```

We run our test, inspect the log, and find no errors. So far so good!

TDD encourages us to

- First, write the simplest code that could possibly work.
- Then, once the tests are running green, take stock and consider whether any of our code could be improved by *refactoring* (fancy word for revising).
- If there are more features to implement, start again by creating another test.

When we look at the PROC IMPORT in line 1, we remember that part of our coding standard is to not hard code the names of any externally supplied files in our programs. Instead, the coding standard says that we should use a FILENAME statement in the autoexec to define a logical reference to the file. So we add the FILENAME to the autoexec, and edit get_data to use the file reference. Now we run the test again and see that it still passes. Using a similar process, we can expand our test program to assert that the second file with power consumption is also read in, and that a merged data set containing data from both files is prepared.

Now that we have our data in SAS, we need to perform the analysis. The survey data represents random samples of individual households. Each record has a weight that represents the sampling weight for that observation. So we want to create a weighted average, and to use census region and number of bedrooms as classification variables. Creating a test case to verify this statistic is going to be a challenge because we have thousands of records. But what if we parameterize our analysis module so that we can feed it a contrived input instead of the real data? This contrived input can be small enough to manually calculate the statistic, to verify the analysis module.

Since we have PROC IMPORT fresh in our mind, we decide to use Microsoft Excel to create a CSV file with our contrived input data. This looks like the following:

| | A | B | C | D | E | F | G | H |
|----|---------|----------|---------|-----|---|---|---|---|
| 1 | REGIONC | BEDROOMS | NWEIGHT | KWH | | | | |
| 2 | 1 | 3 | 1 | 10 | | | | |
| 3 | 1 | 3 | 2 | 13 | | | | |
| 4 | 1 | 2 | 1 | 8 | | | | |
| 5 | 2 | 4 | 1 | 15 | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |
| 8 | | | | | | | | |
| 9 | | | | | | | | |
| 10 | | | | | | | | |
| 11 | | | | | | | | |
| 12 | | | | | | | | |

With this size data set, we can easily determine that the weighted average of KWH (kilowatt hours of electricity) for each combination of census region= and number of bedrooms.

A Test-Driven Development practitioner working in a language like Java or Visual Basic would write some assertions at this point with the weighted average data. However, we decide to be pragmatic about the realities of data-intensive programming such as we are doing. We write a test driver that prepares our simulated data from Excel and calls the analysis module. We then begin interactively writing the PROC MEANS steps in the analysis module, inspecting results as we go until we get the expected answer. At this point, we save the output of the PROC MEANS as a benchmark data set to use in the test case, and modify our test to verify that the analysis module is producing the same output as our benchmark. The final test looks like this:

```

1  /* create mock input data set for analysis
2  */
3  proc import
4  datafile="&SASAPP./ ... /futs/testdata/merged_consumption_01.csv"
5      out=merged_consumption_01
6      dbms=csv
7      replace;
8      getnames=yes;
9  run;
10 %let PREPARED_DATA=merged_consumption_01;
11
12 /* run unit under test
13 */
14 %include "&SASAPP/reporting/include/analyze_data.sas";
15
16 %assert_exist(rptdata.consumption_summary)
17 %assert_compare_equal(
18     base=bench.analysis_ds_01,
19     compare=rptdata.analysis_ds)

```

To recap:

- The PROC IMPORT starting in line 3 reads our test data that we created in Excel into a WORK data set.
- The name of the WORK data set is assigned to the variable &PREPARED_DATA.
- The analysis module assumes the data to analyze is identified as &PREPARED_DATA. It is called in line 14.
- The test asserts that an analysis data set is created (line 16) and that it is identical to our benchmark that we saved (lines 17-19).

Finishing the report is a matter of creating a module to display the analysis data set with appropriate formatting and titles, and creating a main program that calls all the individual modules.

This has been a very brief attempt to give a flavor of what Test-Driven Development is like in action. For an extended case study in Java see (Astels 2003).

OBSERVATIONS ON TEST-DRIVEN DEVELOPMENT

Based on the glimpse of the practice of Test-Driven Development provided by the preceding section, we can make some observations on the characteristics of this methodology:

- **Small steps.** TDD encourages you to write code in small increments. This means that you are always a short time away from a working state, and you are much less likely to fall into a black hole of debugging for hours and hours. The philosophy is that it's better to make "1 step forward, another step forward" rather than "2 steps forward, 1 step back."
- **Lots of test code.** With TDD, you may write as much test code as production code (measured in lines of code).

- **Design for testing.** TDD influences the way you design your code. It encourages you to modularize and to think carefully about the expected behavior of the code you write. Because of the influence on design, this approach has also been called *Test-First Design*.
- **Built-in documentation.** The test cases created by TDD augment the documentation for a project. This documentation aspect is particularly significant for library code that is intended to be reused. The test cases provide a body of sample code to refer to how to use the module.
- **Courage to change.** The body of automated tests produced by TDD gives a developer more courage to make changes and enhancements. There is less risk to adding an enhancement if you have a quick way of verifying that you did not break existing functionality. As an aside, the courage to make changes is further amplified by use of a good version control tool.
- Bottom line: TDD is not really a testing methodology. It is a programming methodology.

This last point helps us to fit TDD into the concept of a *Software Development Lifecycle* (SDLC). Automated unit testing belongs as part of the construction phase, where we are writing code. The tests we are writing are not part of the production code base, they are separate programs that check the programmers' work. This is not to say that the test programs should be run once and thrown away. One of the payoffs of a suite of automated tests is using it to make sure enhancements don't break existing functionality: TDD is also beneficial during maintenance.

TESTING PROBLEMS AND SOLUTIONS

In this section, we consider some specific challenges that arise in attempting to write automated unit tests, and give suggestions on how to address them.

TEST INPUT DATA

Most SAS programs are data intensive. It may seem obvious, but if your input data is not in a known state, your test results will also be unknown. Some of the data-related challenges that we may face as unit test authors are:

- The data may not be available, or not be finalized, at the time of programming. For example, in clinical trials programming, the statistical programmers normally do not have the option to wait until all data is received and finalized by data management before starting their coding.
- The data may be stored in a large database not under our control.
- The data may be stored in a complex data model involving many data sets or tables.
- There may be huge volumes of data that make testing slow and make it impractical to create multiple test cases.

There is no silver bullet to easily overcome these issues. Managing test data is probably the biggest obstacle for automated testing of data-intensive programs. You simply must have input data in a known state to make meaningful assertions about the output of analyzing that data. Here are some strategies that may help your particular situation:

- In one of the examples presented earlier in this paper, a DATA step was used to create some input data. Often programming can be done against simulated data, if it is in the proper structure.
- Another example in this paper used a comma-separated file to provide input data. Often users respond well to being asked to supply sample data in the form of an Excel spreadsheet.
- If your SAS program is working with a database such as Oracle, it may be possible to have your organization devote a database (or a schema) to test data. You may need some backup/restore procedure to reset the database to a known state along with your test database.
- Alternatively, if you are using a SAS/ACCESS@ libname engine for accessing your database, you may be able to simulate the database using a SAS data set. This approach takes advantage of the ability of SAS/ACCESS to make many information stores look like a data set. However, there are usually subtle differences. If you simulate a database with SAS data sets, you will need to do some verification with the actual database along with your simulated database.
- One principle of *design for testability* is to parameterize your inputs. We just touched on one level of parameterization: a libref. You may also consider referring to your input data using a macro variable or parameter. That way you can "point" your code to several different input data sets.

- The *setup* and *teardown* programs can be used to help manage test data. Recall that *setup.sas* is run before each test case. This routine may be a useful place to create test data, read test data from external files, or issue LIBNAME statements. Similarly, *teardown.sas* is run after each test case, and can be used to clean up the work done in *setup.sas*.

RESULT BENCHMARKS

Managing output can present similar challenges to input data. In the simplest case, the output of a SAS macro could be a single number or test string. More frequently, the SAS program or macro that we want to test will produce either data sets or files as output. What kind of assertions can we write about these results?

If the program that you are testing creates a SAS data set, you can use `%assert_compare_equal` to compare your program's output to a *benchmark* data set. A practitioner of TDD might go so far as to create the benchmark up front, using SAS interactively to create it by hand, or a DATA step to create it programmatically, or some other mechanism. Alternatively, you could use the program you are testing to create some output, and then copy that output to serve as the benchmark. You probably want to do some other independent verification of the benchmark, such as manual inspection, before taking this approach.

A file-comparison-to-benchmark strategy can work with output files other than data sets as well. If a SAS program produces a file containing a report, you can compare that output to a benchmark copy.

As with input data, parameterizing your output data with librefs, filerefs, macro variables, or macro parameters makes it easier to write tests.

INCONSISTENT OUTPUT

One challenge that sometimes occurs with comparing test output results to a benchmark file is that the output may change for every execution. The most common example of this phenomenon is when the output contains a timestamp or date of when the program was run. It is pretty common for reports to include this kind of timestamp in their output. In this kind of program, the comparison to the benchmark will fail every time because the timestamp keeps changing.

There are a couple of strategies that can be applied to allow such a program to become testable:

- You could parameterize the timestamp. If the timestamp is supplied to the program or macro, rather than being generated as the current system time, you can set it to a known value and the test becomes repeatable. For example, a report program could look for a macro variable `&REPORT_TIMESTAMP` to include in the output. For a test run, this variable could be set in the test program. For a production run, the value could be set to the current system time in an autoexec file.
- Alternatively, you could write a specialized comparison assertion that knows to ignore certain differences. The ignore logic could be based on line number or pattern matching: for example: ignore differences on line 3, or ignore differences on lines that contain the string "Report generated on".

BINARY OUTPUT

Sometimes the output of a SAS program is a binary file. For example, the program may create a chart as a GIF file. In principle there is nothing to prevent you from doing a binary file comparison to a benchmark file. However, if the comparison fails, you will need to resort to manual comparison in an appropriate file viewing tool to debug the differences. Also, the previously mentioned issue of Inconsistent Output can be more troublesome in a binary file.

In the discussion of issues with input data, we noted that SAS/ACCESS is great for testing because libname access hides (most) differences between different types of information stores. In a similar way, SAS Output Delivery System (ODS) provides some help, because it hides (many) differences between different types of output files.

In order to use ODS to our advantage with testing, we need to modularize our program so that ODS options are decoupled from presentation logic such as PROC GCHART and PROC REPORT. We can then run unit tests to the LISTING, in text form, whereas the production operation can be to formats such as PDF or RTF that are not as friendly to file comparison.

TESTING ERROR CASES

If we are trying to write reliable, production-quality code, we will be very concerned with testing for error conditions and handling those gracefully. For example, in our sample application for reporting on household energy consumption, we probably want to check to make sure that our input data file is present. We can do that with the following assertion:

```
%assert_fexist(rawhous, TYPE=MISSING_FILE, LEVEL=ERROR, ABORT=YES)
```

This assertion states that we are expecting the fileref *rawhous* to correspond to a file that really exists. If the file does not exist, we want to generate an event with a type of *MISSING_FILE* and a level of *ERROR*, and we want to abort the program.

Note this: up to now, we have been discussing assertions in the context of separate test programs. However, this example shows an application of the *assert* API in production code. The *assert* API is actually highly useful for this purpose (Nelson 2004a, Nelson 2004b). ThotWave has even created an entire application (*thinking data*® Event System) for managing runtime events from asserts in production code.

Using asserts for production error checking presents an interesting dilemma. If error checking is that important to us, we would like to write unit tests that prove that the error checking works. However, FUTS thinks that a test has failed if it sees an *ERROR* message in the log. But that is exactly what we want to happen!

The *%expect_error* macro was created for just this purpose. This macro lets FUTS know that we are expecting an error event of a certain type. When the next assertion failure occurs, the type is compared to the expected type. If this failure was expected, no notification is created, and hence the test passes.

CONCLUSIONS

In this paper, we have presented a brief introduction to Test-Driven Development (TDD). One characteristic that distinguishes most SAS programming from the world of object-oriented languages such as Java is the extent to which SAS applications are data-intensive. While there are challenges to following a TDD methodology with data-intensive problems, it is possible to do. Qualitatively, the areas where we see the biggest payoff to TDD are:

- **Quality-critical applications.** Applications where quality is critical, such as clinical trials, can benefit from a rigorous approach to testing.
- **Efforts to create highly reusable macros or include files.** A TDD approach helps ensure coverage of different usage scenarios, and provides a lot of sample code for how the reusable modules are called.
- **Code that will have a long life.** Any code that is going to be maintained for a long time may benefit from a suite of automated tests, because it reduces the risk and testing time associated with enhancements.
- **Complex applications with many interdependent modules.** TDD helps decrease overall debugging time, because individual modules are tested independently before integration.

One could ask if the benefits of TDD on productivity and quality have been quantitatively measured. Some studies have been performed (Janzen 2005), but the results are not yet conclusive. Perhaps future studies will take qualitative factors more into account in setting up study conditions.

In this paper we have also presented FUTS, a Framework for Unit Testing SAS programs. FUTS includes a macro library of assertions for verifying results and a perl script for running suites of test programs in batch. You can use FUTS to implement automated tests in SAS using either a test-first or a test-last methodology. Also, the FUTS assertion library may be used for testing error conditions in production code as well as in dedicated test programs. Please see the ThotWave web site (www.thotwave.com) for directions on downloading FUTS.

REFERENCES

Ambler, Scott W. "Introduction to Test Driven Development (TDD)". <http://www.agiledata.org/essays/tdd.html>.

Astels, David. 2003. *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall PTR.

Beck, Kent and Eric Gamma. "JUnit Test Infected: Programmers Love Writing Tests".
<http://junit.sourceforge.net/doc/testinfected/testing.htm>.

(extreme programming) "Software Downloads". <http://www.xprogramming.com/software.htm> (see Unit Testing heading).

"Introduction to Fit". <http://fit.c2.com/wiki.cgi?IntroductionToFit>.

Janzen, David and Hossein Saiedian. 2005. "Test-Driven Development: Concepts, Taxonomy, and Future Direction". *IEEE Computer*, September, 2005: 43-50.

"JUnit, Testing Resources for Extreme Programming". <http://www.junit.org>.

Myers, Glenford J. 1979. *The Art of Software Testing*. New York, NY: John Wiley & Sons.

Nelson, Greg, Danny Grasse, and Jeff Wright. 2004a. "Automated Testing and Real-time Event Management: An Enterprise Notification System." *Proceedings of the Twenty-ninth Annual SAS Users Group International Conference*, Montreal, Canada, 228-29.

Nelson, Greg. 2004b. "SASUnit: Automated Testing for SAS." *PharmaSUG 2004 Proceedings*, San Diego, CA, DM10.

"NUnit Home". <http://www.nunit.org>.

ACKNOWLEDGMENTS

Thanks very much to my colleagues at ThotWave for their contributions to FUTS and suggestions for this paper, particularly Greg Nelson, Richard Phillips, and Danny Grasse.

CONTACT INFORMATION

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

Jeff Wright
ThotWave Technologies, LLC
E-mail: jwright@thotwave.com
Web: <http://www.thotwave.com>

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

thinking data® is registered trademark of ThotWave Technologies, LLC.

Other brand and product names are trademarks of their respective companies.