PhUSE 2009

Paper TU07

Take me for a test drive

Christof Binder, F. Hoffmann-La Roche, Basel, Switzerland
Dante diTommaso, F. Hoffmann-La Roche, Basel, Switzerland

ABSTRACT

Writing SAS® code is one of the main tasks of a clinical programmer. Making sure a program is doing what it should is probably the most important step in completing that task. Without correct behavior, little else matters. This paper establishes the mental framework to include testing as a natural part of the development process and presents several techniques to achieve this.

INTRODUCTION

Writing code is only one part of the process of getting a program from the idea to a productive state, but it is the one stage that many clinical SAS programmers consider the most important. Structured testing on the other hand is too often considered a 'nice to have' luxury, time permitting.

Additional complexity is added when, as often happens, existing code has to be changed. As with new programs, structured or even automated testing rarely is in place for existing code. It can be challenging to modify such code to implement a new feature without breaking any of the existing functionality.

In this paper several approaches to testing are discussed along with the rationale for why it should be applied and a discussion of the benefits of automated tests.

DIFFERENT SCENARIOS OF TESTING

Writing tests for your programs can range from trivial to really hard depending on the complexity of the program and the environment in which the program is running.

STANDALONE COMPONENT

For a standalone component there is a well established methodology for testing. It has grown mostly from the Java development community and is often centered around the so-called JUnit test setup. The basic idea around it is to have a separate framework that calls your component and compares the result with an expected result that you provide.

The JUnit framework can initialize the test environment to the state that the components require (e.g., system pathnames, non-production test data, etc.). The basic idea of this framework has been ported to many other languages including SAS. For further information see the FUTS framework by ThotWave Technologies, LLC (see **References**).

The beauty of this approach is that it enables the idea of writing your tests first and having your program match the specified outcome bit by bit. This is often referred to as "test-first" or "test-driven" development, and protects already programmed functionality by reporting test cases that fail.

In the SAS world this kind of component can be a macro with little or no interaction with the current environment (i.e. global and local symbol tables). In the companion paper **PO11**, the authors provide an extended discussion and demonstration of building tests for a standalone component (see **References**).

COMPONENTS WITH SOME INTERACTION

Working with a component that has some dependencies on the current environment requires a modified approach. If the component expects macro variables or a specific data set structure, then the test itself can prepare these pieces.

The approach is the same as with a standalone component, with the extra step of creating the required inputs. Note that when running a sequence of tests, these inputs should be disposed of properly after each test has been run. In the Java world this can be handled within the JUnit framework that provides setup and teardown methods. In the SAS world a similar effect can be achieved by allocating temporary work areas that get cleaned up after termination of the program.

COMPONENTS WITH LOTS OF INTERACTION

Sometimes it is necessary to develop or modify a component that interacts heavily with and modifies the current environment. A lot of global macro variables are used, datasets are read and written, other files included, *et cetera*. In such situations, it is often impractical to create independent test cases that set up the entire required, and guarantee that this test environment matches the run-time environment. Building such tests is time consuming, and therefore unjustifiable since any change to related, prior functionality (upstream functionality) could quietly render the tests invalid. One option to test this kind of component it to use a "seam"; a place where you can inject new code without actually changing the code you need to test. This can be done using a combination of preprocessing code and manipulation of the search path for SASAUTOS.

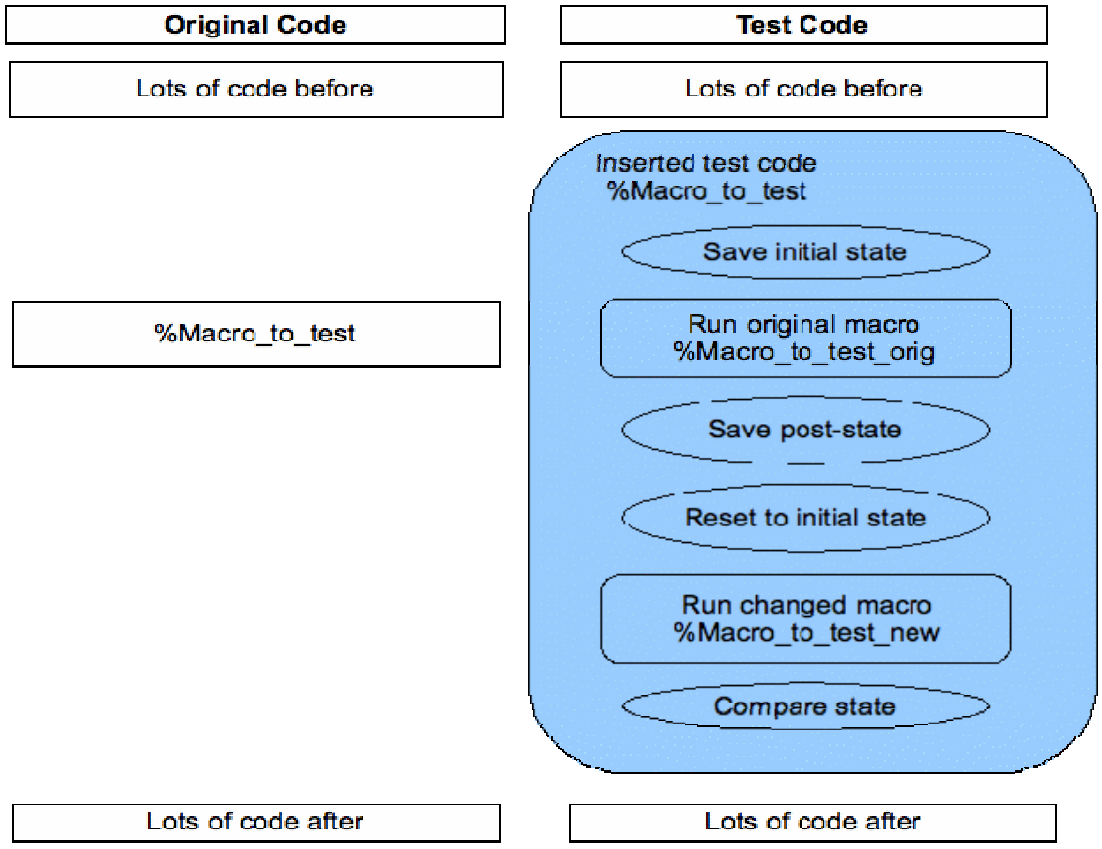

Figure 1: *Diagram of testing given complex interactions*

For example, consider the process for updating the macro `%macro_to_test`, which is too intermingled with other code to be pulled out easily into a unit test. Several steps are necessary:

1. Prepend your test location to the SASAUTOS setting. This can be done in the autoexec.sas or at the beginning of your test script.
2. Create a macro with the same name in the test location. This macro will do the following basic steps
    1. Copy the current (productive) macro definition to a temporary area and rename it ot p_<macroname>.
    2. Copy the development version of the macro to the same location calling it d_<macroname>.
    3. Save the state of the system. This includes all the datasets, macro variables etc. that the macro under testing can change – but not all the other state information that the macro may depend upon
    4. run the productive version, save the output
    5. reset the state of the system
    6. run the dev version
    7. save the output
    8. compare the two outputs and report for success or failure

This can be achieved by a mix of perl-scripts and SAS code. The important part in the renaming step is that not just the file-name of the macro to test is renamed, but also the definition of the macro in the file itself, because SAS does not like the call of a different macro with the same name from within a macro. See Figure 1.

CONCLUSION

In this paper we have shown that there are several options to test new or existing code. Depending on the complexity of the code and the degree to which it relays on the state of the environment, different approaches are appropriate. The most important thing to relay on is that the code you deploy is actually tested and that you are confident that it solves the problem at hand correctly. Testing can lead to more stable, better structured code and avoid time-consuming iterations with the QA function.

REFERENCES

FUTS: http://www.thotwave.com/products/futs.jsp

diTommaso, Dante, and Binder, Christof (PhUSE 2009 - PO11), *Concepts Learned from our Programming Cousins*, http://www.lexjansen.com/cgi-bin/xsl_transform.php?x=phuse09&s=phuse&c=phuse#po.

CONTACT INFORMATION

Christof Binder
Statistical Applications Manager
F. Hoffmann-La Roche Ltd.
Methodology and Innovation
Malzgasse 30
CH-4052, Basel, Switzerland
Tel: +41 61 687-4286
christof.binder@roche.com

Dante DiTommaso
Statistical Applications Manger, Team Lead
F. Hoffmann-La Roche Ltd.
Methodology and Innovation
Malzgasse 30
CH-4052, Basel, Switzerland
Tel: +41 61 687-4314
dante.di_tommaso@roche.com