PhUSE 2010

Paper RG01

A Case Study in Using Unit Testing as a Method of Primary Validation

Ross Farrugia, Roche Products Ltd, Welwyn, UK

ABSTRACT

We piloted an innovative alternative to double programming for (high risk deliverable) validation, on a Phase II study. This was using unit testing, a method more commonly thought of as a developer testing tool. The pilot was successful and I'd like to present our findings and advice for anyone looking to consider this approach in future. I will explain how our implementation worked in practice, and offer conclusions on where we found best suited to use the method, as well as many considerations and benefits that we have learned from the experience. I hope to encourage this idea to be applied by other companies in the future.

INTRODUCTION

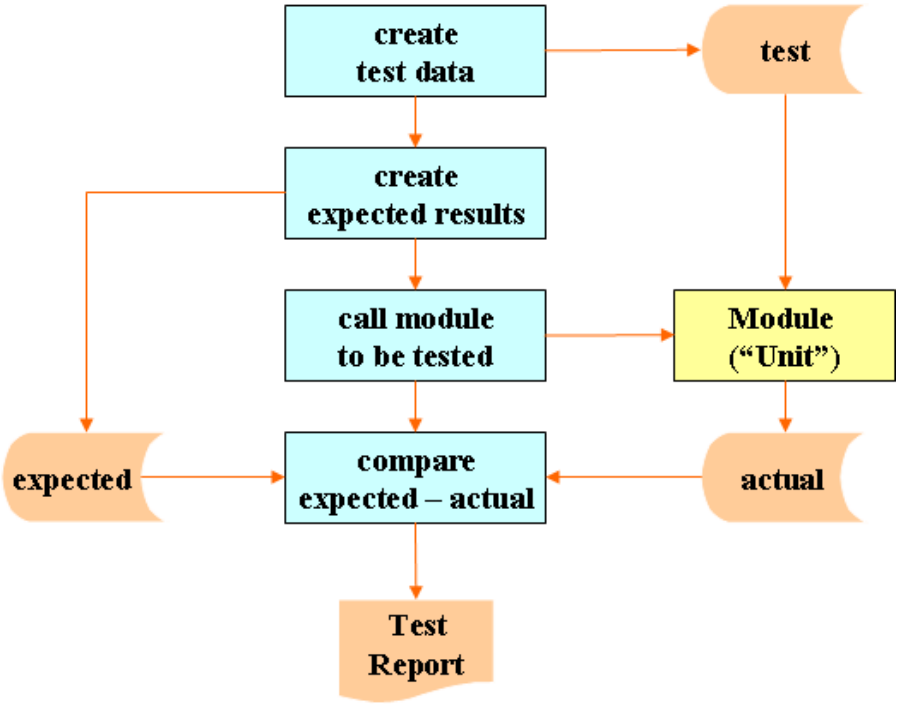
This paper will cover the following:

- A brief overview of unit testing.
- The idea of unit testing as a developer testing tool versus this new initiative as a primary validation tool.
- The proposal we had of how to implement unit testing as a primary validation method for a Phase II oncology study, and our planned areas of usage.
- The steps we put in place to ascertain whether the implementation would prove successful.
- Where we found most appropriate to use this method.
- Techniques which we adopted in our first line programming to enable more efficient unit testing.
- In practice how we applied the new process, and how we developed our unit test programs.
- Some real study examples of unit test programs.
- Our findings, both negatives and positives of the approach used in our case study. As well as further benefits found in later studies using the same ideas, but currently in a more early stage of reporting.
- Conclusions which will include considerations to be aware of and clear advantages that can be gained.

WHAT IS UNIT TESTING?

Unit testing is a method of establishing confidence that a module of code (the “unit”) meets requirements. This is done firstly by creating test data cases that aim to cover the range of expected data scenarios. From the requirements you can then create the expected results from your code when this test data is used as input. Then if you run the module of code using this test data you can see the actual results and compare these with the results you expected. Any difference in the compare will give a clear signal of any inaccuracy in the code, or of course an accidental error in your expected outcomes, which can easily be resolved.

The below flow diagram gives a good idea of this process and the steps involved:



UNIT TEST DESIGN

The first step in producing a unit test program is to thoroughly review your requirements, and once this is done consider the individual features and logic involved. Then start to plan test data cases that can cover each of these. The easiest way to do this is to split them into three categories:

- Regular Cases – These should cover any possible standard everyday data scenarios.
- Boundary Cases – These should cover any scenarios where data points involved in the derivation are set as close as possible or possibly equal.
- Special Cases – These should cover any potential non-standard data scenarios that can occur, for example missing data points.

How you fit your test cases into these categories will vary dependent on the requirements, but I will show one example for a simple study day calculation macro. If the requirement was say *"study day = eventdt – refdt (and if this value is greater than zero then add 1 day)"*, then below is an example of the test data cases that could be used.

ID	Values Category	Test Case	Independent Variables		Result
			refdt	eventdt	sday
01	Regular	refdt before eventdt	15JAN2006	25JAN2006	11
02			15JAN2006	10JUN2006	147
03			15JAN2006	12APR2008	819
04		eventdt before refdt	25JAN2006	15JAN2006	-10
05			10JUN2006	15JAN2006	-146
06			12APR2008	15JAN2006	-818
07	Special	character	"ref"	25JAN2006	missing
08			15JAN2006	"event"	missing
09			"ref"	"event"	missing

10		missing	missing	25JAN2006	missing
11			15JAN2006	missing	missing
12			missing	missing	missing
13	Boundary	refdt just before eventdt	15JAN2006	16JAN2006	2
14			15JAN2006	17JAN2006	3
15			31DEC2005	01JAN2006	2
16		eventdt just before refdt	16JAN2006	15JAN2006	-1
17			17JAN2006	15JAN2006	-2
18			01JAN2006	31DEC2005	-1
19		eventdt = refdt	01JAN2006	01JAN2006	1

Once this is done, you can work out your expected outcomes of the derivation when this data is input (which is shown in the above 'Result' column). This step offers another confirmation of your understanding of the requirements, and can often improve the quality of programming specifications, as some of the data scenarios may have not been considered initially. You can even involve a statistician at this stage to review your test data cases and expected outcomes, which can verify that the understanding of the specification writer matches that of the statistical analysis plan.

The test data cases and the expected outcomes are the two initial parts of the unit test program involved in the design. This can be done as soon as requirements are stable, possibly in parallel with the programming of the module of code.

UNIT TEST EXECUTION

The unit tests can be included all in one program, consisting of a data step inputting your test data cases first into a dataset, and then your expected outcomes into another dataset. Then you just need a step calling the macro module you are testing using the test data cases as input and storing the actual results of this. Finally a compare procedure is needed to check your expected outcomes against the actual results. This can be written to an output file for proof of formal testing.

The program for the previous example may look like the below (though note I've only included the first 2 test data cases):

```
    *** Read test cases ***;

data testcases;
    infile datalines dsd;
    input testcase refdt:datetime20. eventdt:datetime20.;
datalines;
1,15JAN2006:00:00:00,25JAN2006:00:00:00
2,15JAN2006:00:00:00,10JAN2006:00:00:00
…etc
;
run;

    *** Prepare expected results ***;

data expected.sday;
    infile datalines dsd;
    input testcase sday;
datalines;
1,11
2,147
…etc
;
run;
```

```
      *** Execute Study Day Macro using test cases ***;

%sday(source=testcases, target=actual.sday);

      *** Test Actual versus Expected Results ***;

proc printto file=sday_test.report;
run;

proc compare base=expected.sday compare=actual.sday;
run;
```

It should be noted that some macro modules may have multiple input datasets, in this case you would need to create test cases for each. Similarly a macro may create more than one output dataset, so then you would need multiple expected outcomes datasets.

Also you may possibly have to create repeats of the expected and actual results to test different macro flexibilities, if there are macro parameters that can be changed in calls. You even may need to create different test data cases to fully test different flexibilities, but for efficiency purposes this should be avoided.

Even if the above scenarios are faced you still just need one unit test program per module of code you are trying to validate. The separate calls and test data cases can just read one after the other in your program.

An example of some of these scenarios can be seen later in the paper under the section titled 'A More Complex Example Program'.

PLANNED AND ACTUAL USAGE ON OUR PILOT STUDY

Most commonly the method of unit testing has been considered more as a tool for developer testing (i.e. informal checking of your own code before you pass it on for full validation). Our project team decided we would pilot an approach to our validation strategy where we would employ unit testing as our primary quality checking method for certain high risk deliverables, to be performed by a peer. An oncology Phase II study was chosen for this pilot. To ascertain the benefit of the novel approach we decided to still double program the deliverables in addition to unit testing. This way we had a control group to test our results against.

As is clear from the definition of unit testing above, this technique works most efficiently with programs that are split in a modular fashion, e.g. a large analysis dataset split into smaller macro modules, with each one holding a reduced number of derivations. This then means the individual macros can be unit tested, but also some form of integration testing is then vital to ensure when they are called together that the overall program does match the requirements. We proposed a code review would be sufficient here, mainly just to check the order of the macro calls is correct, and the calls themselves have the correct input parameters (where the macros had flexibilities). Also any code falling outside of the macro calls would have to be checked, for example merges between datasets created by the macros.

It should be mentioned we only planned to double program the programs as a whole, so not the individual modules of code. In the example of a large analysis dataset this means we double programmed the requirements for this study (i.e. the dataset as a whole), but not every flexibility of the individual macro modules called within the dataset program, these were just code reviewed.

In actuality due to time constraints we did only choose one key high risk and high complexity analysis dataset to be unit tested. This was split into 12 individual macros, and these macros were given project-level flexibilities. We did this by considering past study requirements (and coming studies that we were aware of). As a by-product of this we were then getting project-level macros that could be picked up and re-used for future studies to come, so we could see an instantaneous long-term time saving being achieved by using this modular approach to programming alone.

We did unit test a few more project-level macros we created, in addition to this analysis dataset, but the main use was this one critical deliverable for this pilot study.

RESULTS OF THE PILOT STUDY

The bottom-line of the results of this study was that unit testing did find updates needed in the modules of code that double programming missed. There are two main reasons for this:

Firstly, at the time of testing we were working on an ongoing database, so at the time we found issues in the code which related to data scenarios we had yet to see in our clinical data. We would have expected (or hoped) double programming would have picked these up eventually when the data scenarios did occur, but this may have been in the later stages of reporting nearing final data. This in mind, we were grateful to have found these early and up front due to unit testing so that we could avoid any late surprises.

Secondly, some project-level macros were given flexibilities to be ready for future studies, and unit testing found issues in these sections of the code, which weren't needed specifically for our pilot study. As mentioned earlier, we were only double programming the whole analysis dataset (i.e. although this program consisted of many macro calls,

we were only double programming those calls of these macros which were relevant to this study, and therefore not testing the full flexibilities of the macros themselves). The code review of the individual macros had not picked these issues up. We would have anticipated finding these required updates when we came to the future study reporting, but finding these early was extremely beneficial, as changes to these macros at a later stage would have then required some form of regression testing or backward compatibility testing to ensure the previous reporting event where they were used would be unaffected.

In all, unit testing gave us confidence that our programs were worthy of their validated status for all future data scenarios we could expect to receive and regardless of which flexibilities were called from our macros.

EFFICIENCY OF THE METHOD

We saw during our pilot study that initially unit testing did take longer for our programmers to carry out, compared to double programming. I'd estimate at this being double the time spent in the early stages. Though much of this was down to it being a new concept, and often we found thanks to unit testing the programmers were spending increased time questioning the specifications more thoroughly, and with this their understanding of the requirements grew (not to mention the quality of the specifications themselves). So some of this we accepted as time valuably spent. Towards the end of the study we found that, with familiarity of the new method, the time cost was more or less the same as double programming.

A further efficiency that has to be considered though is the idea that with modular programming, and macros with project-level flexibilities, you should expect to see a programming time saving on future studies. And also within the current study we saved resource by not having to make later updates to the analysis dataset program. Often this would occur due to new data scenarios occurring for which the original code maybe was not robust enough to deal with correctly.

INVALID DATA

One benefit of double programming was that it did pick up some unacceptable data issues that unit testing had missed. These scenarios would never have been received in final data and were only there as a result of the ongoing database, which our data managers were still cleaning. But still it was useful that double programming found these. Although this was by chance, as these invalid scenarios were not covered in the programming specifications so the programmer and QC'er could have coded the same way just as easily. Also many statistical programming departments have some form of data quality checking tool, which is far more appropriate to use here rather than relying on double programming to find these data issues.

An important point nonetheless is raised though, and that is that unit testing relies on the test data cases that you put in as the user, so you won't be testing the code on any other scenarios, including those of invalid data. It also stresses the importance of putting time into the quality of your test data cases as these need to represent the boundaries of all possible valid data scenarios.

POST-STUDY FINDINGS

During reporting of our next study we did find an issue in one of the project-level macros that was not chosen for unit testing. It was double programmed only and there was a valid data scenario that hadn't occurred on the pilot study, but now did. This error was spotted in the double programming that was repeated for the new study, but it meant that we had to go back and update the macro. This involved regression testing to ensure the updates made had no negative impact on the previous study reporting, which had an extra time cost. We then decided it was worth unit testing this macro so that this will not occur again in the future.

BENEFITS AND CONSIDERATIONS

As with any new method there will always be things you need to consider before adopting this for your project team. This section will highlight where we saw the real advantage in using unit testing, balanced against the extra thought that needs to be weighed up.

BENEFITS

The most important benefits are explained in the results section above, namely unit testing not being data dependent. For every new batch of data received there is no need to re-run your QC program, and there is no risk of an error being later found in your program. Once the program is validated it remains validated. Of course the exception to this is if a new data scenario arises that has been missed in test data cases (see considerations below).

Also from above there is the benefit of having macros available for later re-use with all flexibilities tested, but this is more a benefit related to the idea of modular programming.

The final point briefly mentioned in the results section would be the increased quality of specifications, which is really noticeable. Instead of programming against the current data and querying specifications for the observed data scenarios only, unit testing encourages the programmer to openly imagine all possible data cases, and this is a great way to find holes in requirements.

A further advantage would be that the QC'er does not necessarily need to have a great knowledge of the programming language used. Even for the most complex derivations, the unit test program can be set up using a simple template of code. Even a statistician with limited programming experience could then actually set these up quite comfortably. Or another way the statistician could be brought in would just be to review the test data cases and

agree the expected outcomes. This offers increased understanding for them of the data scenarios possible, as well as a verification of our understanding of their statistical analysis plan.

CONSIDERATIONS

When considering unit testing as a primary method of validation on your study, it is worth from the offset thinking of whether your first line programming strategy is set up to better suit the method. As I've already explained in this paper we used a modular programming approach and I have no doubt this benefits unit testing. This may be a slower approach during initial set-up of programs but the ease of adapting and re-usability of macro modules should make this time back in future studies. When using modular programming for say an analysis dataset program, you may unit test the individual macro modules, but also you need to remember that some form of integration testing is then needed on the program as a whole, but as mentioned before we found code review could be sufficient here.

If macro modules are developed to be called within your programs then you can save time when unit testing by limiting the flexibilities of the macro to only those requirements you know it will ever be needed for. This then means your unit test program will need to test less different calls to the macro. A very simplified example would be if you had a macro that takes a date variable and adds X days to it. You may know up front that you will only ever want to use this macro to add days and not subtract them from that date. Then in this case you limit the flexibility of the macro to only accept X as a positive integer. In this case it saves you having to include a call with a negative number in your unit test program.

With experience you will find that whether or not to use this method versus double programming in terms of efficiency does depend on the derivation you have. For example, if a derivation requires ten different input datasets then you would need a lot of test data cases. Similarly, if a macro has ten different parameters offering a host of different flexibilities then you are likely to need to test many different calls of the macro which each would need new expected outcome datasets created. In these cases, double programming would seemingly be more appropriate.

Another consideration our programmers fed back was that stable requirements will definitely aid unit testing efficiency, as late changes to derivation rules often take more time to update in a unit test program compared to a double program. Purely because the test data cases have to be re-checked to ensure they are still robust enough, and then all the expected outcomes have to be re-traced to see what will change under the new rules. This can be time consuming.

Finally, the most important potential risk in my opinion to be aware of is that unit testing is only as good as the test data cases you can think of. This means it is an activity more suited to someone of experience in your project that knows the data well, so that the appropriate scenarios are thought of that require testing.

A MORE COMPLEX EXAMPLE PROGRAM

I'd like to share another example now for a macro with a slightly more difficult derivation and some flexibilities. This will now require testing of multiple calls to the macro. I'll include the whole unit test program, but split into sub-sections with comments to explain the reasoning behind each step, and some tips I've learned from the experience of using this method.

MACRO REQUIREMENTS

The macro (named %g_neodt) was built for calculating a cut-off date of when a patient had breast cancer surgery, and the specification for this stated that from the macro we need to:

- Calculate the neo-adjuvant period cut-off datetime as the first valid trial medication start date (variable MTBEGDT from dataset MEDT) for a non-missing treatment (variable MTTXT from dataset MEDT) with a valid dose (i.e. variable MTTDD from dataset MEDT is greater than zero) AFTER first non-missing surgery date (variable TRBEGDT from dataset MEDO where variable TRGOAL is equal to 'ACTUAL PRIMARY SURGERY FOR BC').
- If patient receives surgery, but has no trial medication after this date, then use the maximum of the start date of surgery and the primary study cut-off date (macro input parameter &CUTDT; default is the variable DMCUTDT) from the dataset DEMO.
- If patient receives no surgery at all then populate with primary study cut-off date.
- The new variable will be named by the input parameter &OUTDT (default is DMNEODT), and this will be merged onto the dataset DEMO.

(Note: this derivation may seem fairly easy to just double program, but it was chosen here just to give an easily understandable example of a unit testing program. You would visualize greater benefit from using unit testing if this derivation was very difficult to program, but I just wanted an easy example so the tips given below will be easily comprehended)

MACRO PARAMETERS

The following table shows the macro parameters and their default values for this macro. Note: all datasets mentioned in the above specification were made flexible.

OPTION NAME	MANDATORY/ OPTIONAL	DEFAULT	OPTION DESCRIPTION

dsin	Mandatory	<none>	Name of the input dataset in demographic structure
dsin_medt	Mandatory	<none>	Name of the trial medications input dataset
dsin_medo	Mandatory	<none>	Name of the surgery input dataset
cutdt	Optional	DMCUTDT	Input primary study cut-off date variable
goal	Optional	ACTUAL PRIMARY SURGERY FOR BC	The value of the variable TRGOAL which identifies the surgery record
outdt	Optional	DMNEODT	Surgery cut-off datetime variable name
dsout	Mandatory	<none>	Output dataset name

Note: we knew that our variable TRGOAL was always going to be the variable we used to identify surgery records, so we had no need to add this input variable as a new macro parameter. Likewise for the variables MTBEGDT, MTTXT and TRBEGDT, which we were happy to keep fixed, as we'd always use these.

BEFORE YOU START

It is important to identify up front the various dependencies of the macro module, so you know exactly which datasets and variables you need to create test data cases for. Also ensure you review the requirements thoroughly and have a clear idea of how you will work out your expected outcomes.

Then it is worth considering whether all flexibilities of the macro actually need to be unit tested. For example, for this macro I deemed it easy enough to code review the input/output dataset name parameters, but decided to unit test the rest. This is really up to you though, you could just as easily unit test all and less code review would then be needed.

This is the stage you will decide on how many calls of the macros will need to be tested, and often you can combine tests for different flexibilities into one call, thus reducing the number of different expected outcome datasets you'll have to create. Though if two macro parameters are closely linked in the derivation then it may be worth testing in separate calls as it could be difficult otherwise to decipher which one any errors come from. And then you may have to look back to the first line code to identify the exact issue.

At this point consider whether you can make any test data cases robust enough to be used for multiple calls, to save you time thinking of more than one set of test data cases. This time-saver is shown in my example program below, but is not always possible and sometimes multiple sets of input test data cases are inevitable.

UNIT TEST PROGRAM

Firstly, I'd always start with testing the macro with default options. At the top of the program include a description of your test data cases, so that the unit test program is standalone (i.e. descriptions of test data cases do not need to be documented anywhere else), and also this makes the program more easily picked up by another programmer at a later stage if ever needed. For this example, these descriptions are shown below. I find it useful to include a hint of the expected outcome of the derivation along with these, as this saves time when you come to create the expected outcomes dataset. It is worth splitting up regular, boundary and special cases for clarity.

*** Test Data Cases descriptions ***;

* Regular Cases;

* pt 1001: Patient has 2 valid records in MEDT both after surgery, with valid surgery date and valid DMCUTDT
(Outcome=Take earliest MEDT date);
* pt 1002: Patient has 2 valid records in MEDT only 1 after surgery, with valid surgery date and valid DMCUTDT
(Outcome=Take MEDT date after surgery);
* pt 1003: Patient has 2 valid records in MEDT neither after surgery, with valid surgery date before valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1004: Patient has 2 valid records in MEDT neither after surgery, with valid surgery date after valid DMCUTDT
(Outcome=Take date of surgery);

* Special Cases (Here we just need to think of the possible special cases (i.e. missing or invalid values) we can get in the variables involved in the derivation. However, only include those that are possible. For example, MTBEGDT and MTTDD in this derivation are always numeric, so I did not need to test any character values for these);

* pt 1005: Patient has 2 records in MEDT, 1 with missing start date, 1 after valid surgery date, with a valid DMCUTDT
(Outcome=Take valid MEDT date after surgery);
* pt 1006: Patient has 2 records in MEDT, both with missing start dates. Valid surgery date before valid DMCUTDT

```
(Outcome=Take DMCUTDT);
* pt 1007: Patient has 2 records in MEDT, 1 with missing treatment, 1 after valid surgery date, with a valid
DMCUTDT
(Outcome=Take valid MEDT treatment after surgery);
* pt 1008: Patient has 2 records in MEDT, both with missing treatment. Valid surgery date before valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1009: Patient has 2 records in MEDT, 1 with zero dose, 1 after valid surgery date, with a valid DMCUTDT
(Outcome=Take valid MEDT dose after surgery);
* pt 1010: Patient has 2 records in MEDT, both with zero dose. Valid surgery date before valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1011: Patient has 2 valid records in MEDT, with missing surgery date and valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1012: Patient has 2 valid records in MEDT, with no surgery record and valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1013: Patient has 2 valid records in MEDT both after valid surgery date, and missing DMCUTDT
(Outcome=Take earliest MEDT date);
* pt 1014: Patient has 2 valid records in MEDT, with no surgery record and missing DMCUTDT
(Outcome=Missing);
* pt 1015: Patient has no record in MEDT, with valid surgery date before valid DMCUTDT
(Outcome=Take DMCUTDT);

* Boundary Cases
* pt 1016: Patient has 1 valid record in MEDT 1 day after surgery, with valid surgery and valid DMCUTDT
(Outcome=Take MEDT date);
* pt 1017: Patient has 1 valid record in MEDT 1 day before surgery, with valid surgery after valid DMCUTDT
(Outcome=Take surgery date);
* pt 1018: Patient has 1 valid record in MEDT 1 second later same day of surgery, with valid surgery and valid
DMCUTDT
(Outcome=Take MEDT date);
* pt 1019: Patient has 1 valid record in MEDT 1 second earlier same day of surgery, with valid surgery after valid
DMCUTDT
(Outcome=Take surgery date);
* pt 1020: Patient has 1 valid record in MEDT before surgery, with valid surgery 1 day after valid DMCUTDT
(Outcome=Take surgery date);
* pt 1021: Patient has 1 valid record in MEDT before surgery, with valid surgery 1 day before valid DMCUTDT
(Outcome=Take DMCUTDT);
* pt 1022: Patient has 1 valid record in MEDT before surgery, with valid surgery later same day as valid
DMCUTDT
(Outcome=Take surgery date);
* pt 1023: Patient has 1 valid record in MEDT before surgery, with valid surgery earlier same day as valid
DMCUTDT
(Outcome=Take DMCUTDT);
```

Next in the program you start to build your input test data case datasets, as shown below. In this example I've made these robust enough to be used for both calls of the macro tested in this example program. Once you've seen the different calls below, it would be worth coming back to these test data cases so you can see how they've been tailored to cover both. This saves a lot of effort, as building the test data cases is often the most time consuming part of unit testing.

```
*** Test Data Cases from MEDT ***;

data MEDT;
   infile datalines dsd;
   input PT MTBEGDT:datetime20. MTTXT:$200. MTTDD;
datalines;
1001,01DEC2008:00:00:00,DRUG A,100
1001,23DEC2008:00:00:00,DRUG A,100
1002,01DEC2008:00:00:00,DRUG A,100
```

```
1002,23DEC2008:00:00:00,DRUG A,100
1003,01DEC2008:00:00:00,DRUG A,100
1003,23DEC2008:00:00:00,DRUG A,100
1004,01DEC2008:00:00:00,DRUG A,100
1004,23DEC2008:00:00:00,DRUG A,100
1005,                   ,DRUG A,100
1005,23DEC2008:00:00:00,DRUG A,100
1006,                   ,DRUG A,100
1006,                   ,DRUG A,100
1007,01DEC2008:00:00:00,        ,100
1007,23DEC2008:00:00:00,DRUG A,100
1008,01DEC2008:00:00:00,        ,100
1008,23DEC2008:00:00:00,        ,100
1009,01DEC2008:00:00:00,DRUG A,
1009,23DEC2008:00:00:00,DRUG A,100
1010,01DEC2008:00:00:00,DRUG A,
1010,23DEC2008:00:00:00,DRUG A,
1011,01DEC2008:00:00:00,DRUG A,100
1011,23DEC2008:00:00:00,DRUG A,100
1012,01DEC2008:00:00:00,DRUG A,100
1012,23DEC2008:00:00:00,DRUG A,100
1013,01DEC2008:00:00:00,DRUG A,100
1013,23DEC2008:00:00:00,DRUG A,100
1014,01DEC2008:00:00:00,DRUG A,100
1014,23DEC2008:00:00:00,DRUG A,100
1016,23DEC2008:14:30:00,DRUG A,100
1017,23DEC2008:14:30:00,DRUG A,100
1018,23DEC2008:14:30:00,DRUG A,100
1019,23DEC2008:14:30:00,DRUG A,100
1020,23DEC2008:14:30:00,DRUG A,100
1021,23DEC2008:14:30:00,DRUG A,100
1022,23DEC2008:14:30:00,DRUG A,100
1023,23DEC2008:14:30:00,DRUG A,100
;

*** Test Data Cases from MEDO ***;

data MEDO;
   infile datalines dsd;
   input PT TRBEGDT:datetime20. TRGOAL:$80.;
datalines;
1001,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1001,01JUL2009:00:00:00,PRIMARY SURGERY
1002,05DEC2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1002,06DEC2008:00:00:00,PRIMARY SURGERY
1003,01JUL2009:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1003,01JUN2008:00:00:00,PRIMARY SURGERY
1004,01JUL2009:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1004,02JUL2009:00:00:00,PRIMARY SURGERY
1005,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1006,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1007,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1008,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1009,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1010,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1011,                   ,ACTUAL PRIMARY SURGERY FOR BC
1013,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
```

```
1015,01JUN2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1016,22DEC2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1017,24DEC2008:00:00:00,ACTUAL PRIMARY SURGERY FOR BC
1018,23DEC2008:14:29:59,ACTUAL PRIMARY SURGERY FOR BC
1019,23DEC2008:14:30:01,ACTUAL PRIMARY SURGERY FOR BC
1020,01JUL2009:11:00:00,ACTUAL PRIMARY SURGERY FOR BC
1021,01JUL2009:11:00:00,ACTUAL PRIMARY SURGERY FOR BC
1022,01JUL2009:11:00:00,ACTUAL PRIMARY SURGERY FOR BC
1023,01JUL2009:11:00:00,ACTUAL PRIMARY SURGERY FOR BC
;

*** Test Data Cases from DEMO ***;

data DEMO;
   infile datalines dsd;
   input PT DMCUTDT:datetime20. NEWDT:datetime20.;
datalines;
1001,15JUL2009:00:00:00,01JAN2009:00:00:00
1002,15JUL2009:00:00:00,01JAN2009:00:00:00
1003,15JUL2009:00:00:00,01JAN2009:00:00:00
1004,01MAY2009:00:00:00,03JUL2009:00:00:00
1005,15JUL2009:00:00:00,15JUL2009:00:00:00
1006,15JUL2009:00:00:00,15JUL2009:00:00:00
1007,15JUL2009:00:00:00,15JUL2009:00:00:00
1008,15JUL2009:00:00:00,15JUL2009:00:00:00
1009,15JUL2009:00:00:00,15JUL2009:00:00:00
1010,15JUL2009:00:00:00,15JUL2009:00:00:00
1011,15JUL2009:00:00:00,15JUL2009:00:00:00
1012,15JUL2009:00:00:00,15JUL2009:00:00:00
1013,                  ,
1014,                  ,
1015,15JUL2009:00:00:00,15JUL2009:00:00:00
1016,01NOV2008:00:00:00,01NOV2008:00:00:00
1017,01NOV2008:00:00:00,01NOV2008:00:00:00
1018,01NOV2008:00:00:00,01NOV2008:00:00:00
1019,01NOV2008:00:00:00,01NOV2008:00:00:00
1020,30JUN2009:00:00:00,30JUN2009:00:00:00
1021,02JUL2009:11:00:00,02JUL2009:11:00:00
1022,01JUL2009:10:59:59,01JUL2009:10:59:59
1023,01JUL2009:11:00:01,01JUL2009:11:00:01
;
```

Now you need to create your expected outcomes dataset. I'd advise once you've done this just to take a quick run through and see you agree with any derived values, as it easy to make copy and paste errors or typo when compiling unit test programs. Ideally, you want to be sure when the program is ran that any issues coming out in the compare at the end are QC issues with the first line program, rather than errors in your own unit test program.

```
*** Expected Results for DEMO output dataset ***;

data EXPECTED.DEMO2;
   infile datalines dsd;
   input PT DMCUTDT:datetime20. NEWDT:datetime20. DMNEODT:datetime20.;
datalines;
1001,15JUL2009:00:00:00,01JAN2009:00:00:00,01DEC2008:00:00:00
1002,15JUL2009:00:00:00,01JAN2009:00:00:00,23DEC2008:00:00:00
1003,15JUL2009:00:00:00,01JAN2009:00:00:00,15JUL2009:00:00:00
1004,01MAY2009:00:00:00,03JUL2009:00:00:00,01JUL2009:00:00:00
1005,15JUL2009:00:00:00,15JUL2009:00:00:00,23DEC2008:00:00:00
```

```
1006,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1007,15JUL2009:00:00:00,15JUL2009:00:00:00,23DEC2008:00:00:00
1008,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1009,15JUL2009:00:00:00,15JUL2009:00:00:00,23DEC2008:00:00:00
1010,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1011,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1012,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1013,                   ,                   ,01DEC2008:00:00:00
1014,                   ,                   ,
1015,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1016,01NOV2008:00:00:00,01NOV2008:00:00:00,23DEC2008:14:30:00
1017,01NOV2008:00:00:00,01NOV2008:00:00:00,24DEC2008:00:00:00
1018,01NOV2008:00:00:00,01NOV2008:00:00:00,23DEC2008:14:30:00
1019,01NOV2008:00:00:00,01NOV2008:00:00:00,23DEC2008:14:30:01
1020,30JUN2009:00:00:00,30JUN2009:00:00:00,01JUL2009:11:00:00
1021,02JUL2009:11:00:00,02JUL2009:11:00:00,02JUL2009:11:00:00
1022,01JUL2009:10:59:59,01JUL2009:10:59:59,01JUL2009:11:00:00
1023,01JUL2009:11:00:01,01JUL2009:11:00:01,01JUL2009:11:00:01
;
```

Finally, just execute the macro call and then proc compare the resulting dataset against your expected outcomes. Any differences shown will then lead to your QC findings of the first line macro code. You can also add a print procedure to an output file if you want to keep formal evidence of the testing.

```
*** Execute macro with default options ***;

%G_NEODT(dsin=DEMO, dsin_medt=MEDT, dsin_medo=MEDO, dsout=DEMO2);

*** Test Actual versus Expected Results ***;

proc compare base=DEMO2 compare=EXPECTED.DEMO2; id PT; run;
```

Then beneath this in the program can be the second set of expected outcomes for the new macro call testing the flexibilities. In this example we planned to test the macro parameters goal, cutdt and outdt. As the special and boundary cases had been tested already above, I deemed these flexibilities easily could be tested by just using the regular cases (i.e. PTs 1001 – 1004). Hence why in the test data cases for MEDO I only needed to create differing values of TRGOAL for these patients. Same for NEWDT in the DEMO test data cases. If you look back at the input test data cases with the new flexibilities you should see why the expected outcomes for the derived variable have now changed.

```
*** Expected Results when goal=PRIMARY SURGERY, cutdt=NEWDT and outdt=DMSURDT for DEMO
output dataset ***;

data EXPECTED.DEMO3;
   infile datalines dsd;
   input PT DMCUTDT:datetime20. NEWDT:datetime20. DMSURDT:datetime20.;
datalines;
1001,15JUL2009:00:00:00,01JAN2009:00:00:00,01JUL2009:00:00:00
1002,15JUL2009:00:00:00,01JAN2009:00:00:00,23DEC2008:00:00:00
1003,15JUL2009:00:00:00,01JAN2009:00:00:00,01DEC2008:00:00:00
1004,01MAY2009:00:00:00,03JUL2009:00:00:00,03JUL2009:00:00:00
1005,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1006,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1007,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1008,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1009,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1010,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1011,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
```

```
1012,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1013,                  ,                  ,
1014,                  ,                  ,
1015,15JUL2009:00:00:00,15JUL2009:00:00:00,15JUL2009:00:00:00
1016,01NOV2008:00:00:00,01NOV2008:00:00:00,01NOV2008:00:00:00
1017,01NOV2008:00:00:00,01NOV2008:00:00:00,01NOV2008:00:00:00
1018,01NOV2008:00:00:00,01NOV2008:00:00:00,01NOV2008:00:00:00
1019,01NOV2008:00:00:00,01NOV2008:00:00:00,01NOV2008:00:00:00
1020,30JUN2009:00:00:00,30JUN2009:00:00:00,30JUN2009:00:00:00
1021,02JUL2009:11:00:00,02JUL2009:11:00:00,02JUL2009:11:00:00
1022,01JUL2009:10:59:59,01JUL2009:10:59:59,01JUL2009:10:59:59
1023,01JUL2009:11:00:01,01JUL2009:11:00:01,01JUL2009:11:00:01
;

*** Execute macro for when goal=PRIMARY SURGERY, cutdt=NEWDT and outdt=DMSURDT for DEMO
output dataset ***;

%G_NEODT(dsin=DEMO, dsin_medt=MEDT, dsin_medo=MEDO, cutdt=NEWDT, goal=%STR("PRIMARY
SURGERY"), outdt=DMSURDT, dsout=DEMO3);

*** Test Actual versus Expected Results ***;

proc compare base=DEMO3 compare=EXPECTED.DEMO3; id PT; run;

*** End of unit test program ***;
```

CONCLUSION

The main conclusion I have to offer from this paper is that unit testing can be a very valuable alternative to double programming for validation of high risk deliverables. However, it is not the ideal method to implement in all cases. As our team's experience grew of unit testing on our pilot study we found that there are advantages and disadvantages of using each method, and in the end we found that certain programs were more suitable for unit testing, and others for double programming. In future we will use these learnings to use a combination of the two methods, which I believe will increase our overall reporting quality and efficiency. I hope that through this paper there are enough tips and guidelines for other project teams to implement this approach to their benefit as we have so far, and as we will continue to do so.

ACKNOWLEDGMENTS

I'd like to give thanks to:

- Frederik Malfait and Ryan Copping (for review of my paper)
- Cheryl Kubsch (for re-use of some training material to contribute to this paper)
- Hinal Patel and Francis Kendall (for allowing and encouraging this pilot to go ahead)

CONTACT INFORMATION

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

Ross Farrugia
Roche Products Ltd
6 Falcon Way, Shire Park
Welwyn Garden City, UK / AL7 1TW
Work Phone: 01707 365918
Email: ross.farrugia@roche.com