Paper TT11

Why Writing Tests Does Save Time in Development

Magnus Mengelbier, Limelogic AB, Malmö, Sweden

ABSTRACT

The Life Science industry spends an unproportionate effort to create unofficial re-usable utilities, such as the SAS macros copied from study-to-study that never reaches the global library. As R is used more broadly across an organization, bespoke packages are soon to follow. An interesting lesson learned is how tests for an R package is managed in the overall development cycle and what roles they can play. We consider different development approaches, from the common develop first and test last to incorporating adaptations of Test Driven Development. An interesting reflection we explore is how writing the tests early in the development cycle can both be used to significantly decrease the development time for utilities and tools written in SAS, R, Python, etc., but also how that can make that SAS macro transition from study to study to the more beneficial study to global library.

INTRODUCTION

Developing standard tools and utilities is not uncommon, driven by reusability, standardization and, more recently, automation. The effort required to maintain these standards is clearly dependent on the functionality and the resources that they rely on, but as these utilities and tools become more complex and integrated into our environments, an increase in the effort needed to maintain them will follow.

We consider a simple moderately complex utility and a user issue to contrast the effort needed for that issue to be resolved without and with tests. Although our example is based on real-life scenarios, they are generalized into a common hypothetical use case to highlight lessons learned and practical points to consider.

The use case considers both the near-term implications of writing tests from the start of development to possible longer-term returns when faced with reported issues. Writing more code may seem counterintuitive at first, but the additional is easily justified considering that we would need to write tests for validation anyway.The introduction explains the purpose and scope of your paper and provides readers with any general information they need to understand your paper.

THE THING

Let us assume that we will develop the thing for this paper. The approach is the same regardless if it is a SAS macro, R function or Python class (), but for convenience we will use the convention of referring to the thing as a SAS macro, i.e. `%thing()`.

SAS macro

```
%macro thing();
 %* some code here ... ;
%mend;
```

R function

```
thing <- function() {
 # some code here ...
}
```

Python class

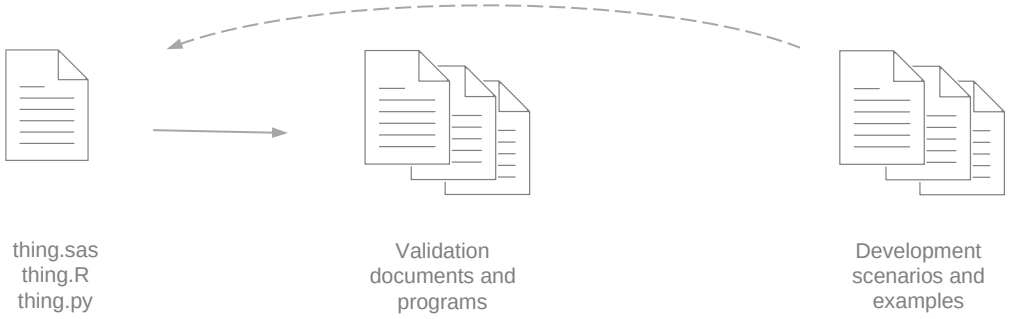
```
class thing:
  # some code here ...
```

SAS as a developer's language is also one in which the community praxis is not in writing general tests during development as it lacks a simple test framework, which makes the reference a good example.

For us, the `%thing()` is a small moderately complex component with a limited set of input parameters but it conveniently relies on an external resource, say a database or a service, to fit the arguments in the paper. We will also assume that it is used for GxP workloads and thus needs to be validated.

GETTING STARTED

Very few of us could write `%thing()` from start to finish in one sitting. We write code in increments, trying out what we have just written, amending as needed until we are happy with a block of code and then moving on to the next part. The important notion is that this development style is very interactive and iterative. Validation at this point is pushed to the future.



How we organize ourselves to develop this way is a very personal approach. Some will have a small program file that contains half-complete scenarios that you can run to test out the code during development. Others will have those scenarios at the bottom of the file for the `%thing()`. Some have a temporary file that they throw away at the end of each development session. Added onto that, if the `%thing()` takes input data, you most probably have small hack-able test data files.

In my case, I will end up with at least two files, the official file containing code for the `%thing()` and one unofficial file containing the development scenarios. That second file is not run from top to bottom, instead I can pick and choose code to run to fulfil my immediate needs, so there is no clear strategy or structure. The code has a very high likelihood of becoming increasingly unstructured over time.

If you have adopted an Agile methodology in your development process, there are additional iterations to this process. You may share a program with example scenarios to those that review your product, and who in turn may also have their own development scenarios. This is turning into a lot of file with a lot of unstructured unorganized code very quickly.

Assume now that the code for `%thing()` is complete, everyone agrees that it fulfils expectations and everyone is conveniently reminded that it needs to be validated. This results in the developer, tester or validator, depending on your validation process, creating one or more test and/or validation programs with associated validation documentation to meet the minimum validation requirements, which are most likely drafted after the code is or very nearly finished. More files and more code, this time a bit more structured.

Once all of tests have passed and been reviewed and signed off, the `%thing()` is released for use, meaning the program code is published to a central location, the test/validation programs and the documents are archived appropriately and users can incorporate `%thing()` in daily activities. Note that the development scenarios are not part of this list since they are seldom official artifacts.

THE ISSUE

A year or two later a user contacts the maintainer with an issue. We will assume that the error message(s) included are not the developers, so the issue is an uncontained/uncontrolled error, e.g. not triggered by an error or sense check in the `%thing()` and thus not obvious. Let us keep in mind that in most organizations, support for `%thing()` is in addition to the maintainer's regular responsibilities, hence below is competing with other tasks you have to get done.

The first step in diagnosing the issue is most probably to replicate the issue that the user has. If you cannot replicate the issue, the root cause can either be associated with that user's permission or how and where the code is executed. Let us assume you can replicate the user's issue and, by the looks of it, it should work. So, the root cause is most probably somewhere in the code.

The next logical step is to identify the failing code block. The code block you have identified is internal logic not seemingly relying on anything. As examples, that could be a DATA step in SAS, a `lapply()` in R or a traversal of some transient data object in Python.

The first thing to notice is that this code block is processing data that only exists internally, i.e. it is not in an external input. After some more digging, the other clue is that the failure seems to be isolated to a particular line of code, but it is not entirely obvious.

You now have two options, update the code to save the failing input data to an externally accessible data file or object that you can review or copy the code block out and try and reconstruct what you think the input data looks like, for both good passing and failing scenarios. If you are the original developer, you may have retained a copy of the development scenarios, so this is about the time to start comparing notes.

Let us now assume you have identified the core issue in that it is data driven. So how did we end up here? We start to meticulously trace back how that data came to be. After walking back multiple steps and redirects, the origin of the troublesome data values is another data block that contains a merge of two data sources and some simple and easily recognizable derivations. As it turns out, one of the data sources is an extract from that external database or service, but the way the code was written would allow the developer to inject a data file/object in place for development purposes. This turns out to be a misdirection and not the cause.

As it turns out, the database or service was available, but it failed to transfer the information that `%thing()` was expecting, hence the recognizable derivation depended on a default value mechanism that the developer implemented but which was incompatible with the scenario that the user reported as part of the issue. The fix is quite straightforward to resolve by introducing one or more error checks for the conditions that led up to the failure to ensure that a value was retrieved from the external resource when the user's scenario is in use.

The developer probably assumed the external database or service would be available, but the values used in the derivation may be optional and not guaranteed to be defined in the database or service, hence the default value. The transfer failure simply meant that the default value was used regardless of if the input value was defined in the database or service.

The question is not that you found the root cause for the user's issue in the end, but how long did it take? Did it take a few hours? A few days? A week? Could a different development approach have avoided the issue entirely or at least reduce the effort to identify the root cause? The obvious answer is yes, but at what cost.

WRITING TESTS

The above meandering effort to identify the root cause and fix the issue could have been significantly reduced by a simple diagnostic program to ensure that the database or service is available. The diagnostic would however not have resolved this particular issue since the root cause was not the availability of the database or service, but that a default value was used. Provided a diagnostic to ensure that the database or service returned sensible results given our user's scenario, that would have been more appropriate, but it would be impractical to create diagnostics for every possible user scenario.

The issues could most likely have been avoided if the user's particular scenario was tested, since it was obviously dependent on a specific input value (not input parameter). We say tested as it is not entirely obvious if the scenario is part of the formal requirements for `%thing()` or it is down to developers prerogative.

What if it turns out that somewhere hidden in the developer scenarios there is code that could have identified the issue? There was the implementation of defaults for a reason and thus likely that the developer did at some point test the derivation given the correct input values.

The additional effort to write tests instead of the developer scenarios is not as substantial as it may seem. If you recall, we most likely write code for the developer scenarios and, at a later stage, create the test programs/code for validation, in many ways duplicating effort. If we instead write tests during development, it is very likely they are comprehensive enough, or near enough with little modification, to also be used as part of our validation effort.

We should note that writing tests during development is a common convention for R and Python packages, but less so for SAS. This is most often down to R and Python having well-established praxis and the availability of test frameworks. A test framework is essentially a utility that permits you to run tests in a structured and isolated way and record the results. We can easily do something similar with test programs in SAS, if you are not able to use a SAS test framework, like SASUnit.

We should acknowledge that there is a practical difference between tests used for development purposes and those that are formal validation checks to ensure that the requirements are met and the implementation of those works as expected. However, the difference is usually in formality, given that tests used in validation are most often constructed with a specific purpose. If we adopt the same mindset for development tests, we can fulfill both with the same effort and no duplication.

Validation is in principle a process that uses tests and test results to document and demonstrate that a function, method or feature is fit for its intended use. It is the collection of tests, and not one test in particular, that fulfil those obligations. In most organizations, we can therefore use the tests written during development for the purpose of

validation as well, just ensuring that the collection of tests covers the desired purpose of validation, also sometimes referred to as coverage.

Using a test-driven development approach could also ensure that during development, any additions, patches or changes does not break the `%thing()`, which is most likely to happen at some point. In retrospect, test-driven development can significantly decrease the time to develop moderate to complex components as it forces the developer to write testable and runnable code.

WRITING TESTABLE CODE

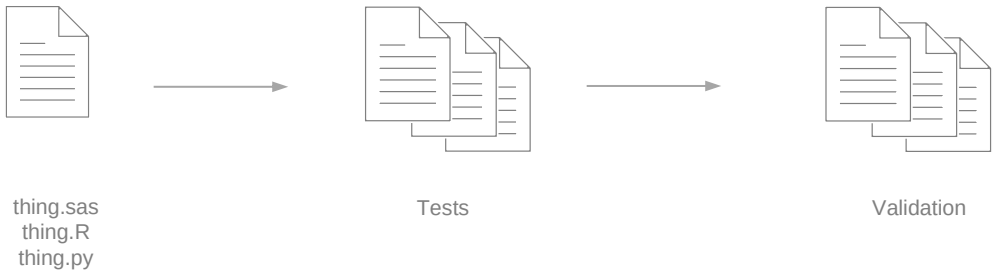
A large part of moving from using development scenarios to a test-driven approach is writing testable code. In most cases, there is very little difference between the two beyond adopting a more structured and reusable approach for the latter.

A key principle is that a test scenario is entirely stand-alone. It can be run any time, in any sequence and with no dependence on previous tests and test results.

A test scenario also follows a very simple set pattern; the test is staged, executed, assessed and cleaned up. In practice, many tests are extremely similar variations on a theme making the actual test development a copy-paste-and-tweak exercise, note not modify. Very seldom would you make any unique one-off extensive modifications to the copied test scenario.

Testable code simply means that you write code so that it is easier to test, i.e. you can reach and test code nested within multiple levels of if-conditions. A simple trick is to isolate the code block within that nest by moving it to an internal macro, function, class or method rather than to try and construct and maintain complex scenarios to navigate the if-conditions. Just simply test the internal macro, function or method directly. This would also permit you to test scenarios that would not otherwise be possible, since they could have been blocked by the nest of if-conditions.

An internal macro, function or method can simply be added to any of the languages we consider. A simple convention could be to name internal SAS macros with a leading underscore, i.e. `%_internal_for_thing()`, using a dot-prefix in the function name and not export the function for R to make it hidden, or simply a smaller class or internal method for Python.



There are several added benefits to writing testable code. The quality of the code improves as the code is tested more often and, as the test scenarios evolve and are expanded with use across studies, projects and global libraries, much more of the code is tested, i.e. increased test coverage, and not just the minimum to satisfy the lowermost threshold in order for it to be called validated. Validation is then no longer an arduous unnecessarily expensive task but rather a documentation exercise to run the tests and document the results. Validation then quickly becomes a common practice.

DÉJÀ VU

As an experiment, let us consider the user issue once more, but now having test scenarios rather than the original development scenarios.

The first step in diagnosing the issue is again most probably to replicate the issue that the user has with the same result as before. It is replicated.

The second step would be to execute the tests to ensure that the code was still working as expected.

It is possible that one or more tests would exist to assess if the external database or service the `%thing()` relies on is available and would return a value the is compatible with the user's scenario. If the value was not retrieved as expected, this test sequence will most probably fail, and we would have identified the cause already.

Let us assume that that lesson was not learned just yet. A second possibility is that tests could exist for the user's particular scenario given the dependency on particular or specific input values. Even if those tests would be entirely isolated on the derivation itself and not result in a failure, it would provide assurance that the derivation algorithm worked as expected given the correct input values.

At this point, we know that the database or service is available, and that the derivation works because we have passing tests. The attention is then drawn to the link between the two. If a test exists for the user scenario to ensure that our expected input value is available in the database, we would have a failing test. Is the root cause identified? Probably not, since our external resource for testing is seldom our production database or service, but it does give us a frame of reference to compare against that can lead to diagnosing the problem.

However, if that test is missing, we are fairly confident it is a true bug. We can construct that missing test scenario to replicate the issue. Once the bug is fixed, we then have a check to ensure that the bug was corrected, and any future updates do not cause the issue to reappear.

If we compare déjà vu with our original effort to identify the root cause and resolve it, using tests can save a great deal of effort. If we already had the test to identify that the expected input value was not retrieved from the database or service, the cause could have been identified with running the tests. In practice, that could be accomplished within an hour.

If a test scenario did not exist, the root cause could have been identified with little additional effort since the existing tests that passed would eliminate many possible issues. The existing tests scenarios can also be used as a reference to what is and what is not normal use. Given the user scenario is normal use, we would naturally expect that at least one test exists.

CONCLUSION

Developing standard utilities is not uncommon, driven by reusability, standardization and, more recently, automation. The effort required to maintain these standards is clearly dependent on the functionality and their dependencies, but as these utilities and tools become more complex and integrated into our environments, an increase in the effort needed to maintain them will follow.

The effort saved in the longer term can be relatively significant with investing additional effort in writing tests during development, a distinct subset we would need to do for validation anyway. Not all macros, functions or methods are representative of the `%thing()`, but writing tests during development does promote cleaner and better coding practices.

Automating GxP tasks and workloads will require more rigorous validation, i.e. testing, so by incorporating test driven development, or a variation thereof, is just a benefit as there is no or little difference between creating and maintaining standard utilities and tools used by users and those incorporated into automated workflows. With good testing strategy, it is just how these tools and utilities are developed and with what ease they integrated into daily tasks.

CONTACT INFORMATION

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

Magnus Mengelbier
magnus.mengelbier@limelogic.com

Limelogic AB
Jungmansgatan 12
211 11 Malmö
Sweden

www.limelogic.com