PhUSE 2014

Paper AD01

Patchwork – A Creative Approach to Revise Existing Validation Methods

Anja Feuerbacher, F. Hoffmann - La Roche, Basel, Switzerland

ABSTRACT

The validation of generic scripts and macros can be a slow and laborious process, frequently seen as a hindrance to the smooth flow of implementing or changing software. With the knowledge that this is an essential part of the drug development process, the key step is the risk assessment of the impact the changes will have and this risk decision in turn leads into the depth of validation that is required. Over the lifetime of a script or macro, many regenerations of the code are likely to occur, each with their own validation requirements. This constant updating can lead to the validation packages containing a lot of unclear, inconsistent, duplicated, or even erroneous tests, so that the execution is very cumbersome and can take several days to complete.

Sounds familiar, right? In this paper I want to show an impressive example how the idea of a "validation patchwork" helped me to revise the validation package – unravelling what had become a misshapen blanket, picking out the good pieces, cleaning them and setting them back together to form a nicely framed wall hanging. And this is far from the end of the story - merely the beginning of rummaging through the house to review and redesign the entire interior.

INTRODUCTION

Patchwork - what does that actually mean? Patchwork is a centuries-old technique reutilising textile remnants which are sewn together into a larger design, using different patterns and all kinds of fabrics. Over the centuries a variety of traditional patterns evolved, which are still in use today. However, in the past decades patchwork has also become a modern art with artists designing elaborate pictures. They make use of traditional patterns and variations of these, or use no pattern at all.

Validation and verification ensure that a program fulfills its requirements and functions properly. In case of modifications it must be guaranteed that no errors are introduced into a working program. A modification can be the removal of an error that has not been detected before, the addition or change of functionality, or the removal of functionality that is never used. There are a wide range of validation techniques, including regression testing, which will be used as the method of choice in my example. "Regression testing is the re-execution of a subset of tests that have already been conducted, to ensure that recent changes to the software have not propagated unintended side effects." [Futrell, p. 724]

Within the scope of this paper is the validation and verification of generic UNIX scripts and SAS® macros, hereafter called "components", which are independent of particular projects, and may be utilized by different functions.

The following sections will explain the validation approach in general, the revision of the validation package for a specific component, and consequences of these revisions on the validation process. The paper is designed as a Q&A, always following the "patchwork" motif:

- How do you validate a component? - How does this relate to patchwork?
- Why should you revise your validation packages? - Why should you revamp anything?
- Give me an example: How did you do it? - Looking at a particularly misshapen blanket
- What was the outcome? - Comparing the old and new design
- Were there consequences for the validation process in general? - How will you look at the remaining interior?

HOW DO YOU VALIDATE A COMPONENT?

Before we can look at the details of validation and verification, we must first get an idea of how a component is created or changed. When a requestor defines the requirements for a new component or modifications to an existing one, the developer identifies new components and evaluates the extent of modifications and the impact on related components. After a risk assessment and the selection of the appropriate validation methodology, the developer creates the code or implements the change. Then the developer and the reviewer perform and document validation and verification tasks as described for the relevant validation methodology.

The validation methodology for a component is determined by a risk assessment, which has to be done for all components, and is associated with the component itself. The risk assessment of a component is not intended to indicate the degree of risk that a change may incur. The three basic input parameters to the risk assessment process are:

- *Likelihood of failure* (low, medium, high)
- *Business impact* (low, medium, high)
- *Probability of failure detection* (obvious, likely, only by chance)

The resulting risk assessment value leads to the indicated validation methodology. In the case of low overall risk the methodology selected may not require a validation package. For medium and high overall risk the selected validation methodology must describe the steps that lead to a reusable validation package for a component. The methodology for medium- and high-risk components will be covered below.

The validation methodology applicable for medium- and high-risk components is not intended to provide details for each and every action required to validate a particular component. Rather, it outlines the general process, terminology, and the purpose of related documents. The focus is on ensuring that a component continues to provide existing functionality despite additions and modifications and that backward compatibility can be guaranteed (this is known as regression testing). How this is actually done is described in the related documents: the program design document and the validation instructions document. The program design document should contain sufficient detail of the requirements to validate the testing package against it. It is envisaged that this document will become a fundamental part of the process, and as such it should be ensured that the original requirements are described, as well as all future changes, although it may be acceptable simply to refer to the change request. The validation instructions document contains a detailed set of actions that have to be performed to ensure proper validation and verification of a component. This includes, but is not limited to creating the environment for requirements and regression testing, and which results are expected.

Being responsible for producing a new version of a component, the developer creates or modifies the program design document, performs the risk assessment and creates or modifies the validation instructions document. For the requirements phase certification new tests are designed and executed, and added to the validation instructions document together with details for related regression phase verification. Both, the developer and the validator run the entire validation package from a clean start, commencing with the earliest tests and working through to the newly added tests. When QA passed the validator documents the successful execution of the validation package.

The same procedures have to be followed in the case that a new version of a program adds no new or changed functionality. This will include performing a full regression testing run as above, as well as documenting why no new testing is required.

How is the above related to patchwork? We can look at the topic from a different angle and view it as a validation patchwork. We will find an assembly of patterns (tests) sewn tightly together. Some patterns may be similar (tests for similar requirements), others are unique. All patterns use different shapes of fabric. You can add patterns as needed (new requirements), but you cannot separate one without the risk of unraveling the rest (full regression test ensures backward compatibility). Let us assume a hypothetical household with lots of stuff made of patchwork. As each component in the computing environment has its own volume of validation, we will find smaller and larger patchwork items (components requiring less or more validation). Examples include blankets, cushions, wall hangings, table cloths, or pot holders.

WHY SHOULD YOU REVISE YOUR VALIDATION PACKAGES?

A generic component is designed for a certain purpose and is generally used by many people. A validation package is connected to a particular component and thus fulfills a single purpose. The amount of validation done, and thus the size of the validation package, basically depends on the complexity of the component for which the validation package is designed. Over the years a component may have been enhanced by many programmers clearing bugs and adding functionality. As a result many programmers have added many tests to the related validation package. There are various reasons, why such a validation package should be revised:

- *Similar tests*: Similar modifications to a component lead to similar additions to the validation package, so the package may contain many very similar tests.
- *Duplicates*: It can happen that the same test was added twice or even more to be executed in different areas. For example, a SAS component can be executed in a development, QA and production area. The component which initiates the SAS runtime environment has a proper validation package to ensure consistency throughout these areas. It can therefore be expected that a component running successfully in one area will also run in the others. However, when following the process the tests still have to be performed and thus work is duplicated.

- *Unclear or inconsistent instructions*: Although the validation instructions document is based on a template that has to be used for all tests, programmers may still have different perceptions of how to use it or how to phrase their tests. Thus, the template may have been used in different ways, and some tests may not be well-written. That means that some tests are not easily understood and thus not easy to perform.
- *People make mistakes*: It can happen that a test does not work in the way it is described. For example, a step may need to be performed with administrator privileges although the instructions say otherwise. Or the instructions contain a typo in the directory that has to be set-up for executing a test. The actual settings must first be clarified before such a test can be executed successfully.
- *Manual testing*: It is a matter of course that regression testing implies automated testing, because the intention is to execute the validation package upon each modification of a component. However, it is a matter of fact that many tests are manual.
- *Software versions*: In a statistical computing environment with SAS as the standard tool many components are designed to help SAS programming. A component and validation package developed with a particular SAS version may not function properly with another version. When several versions of the SAS software are in use a generic component must support all. In which case the validation package must take care of the various versions, too.
- *Unneeded functionality:* Some functionality of a component may be out of use, and respective testing only adds workload without value. When backward compatibility is not an issue the programming team may want to remove that functionality, in which case related tests must be removed as well.

Taken by itself each of the above is a nuisance. In a large validation package that has evolved over many years with many programmers involved, you may find all of these nuisances in various combinations. At some point you will likely begin to experience that developers are reluctant to implement even the tiniest change to the component. For the modification is no problem at all, but the validation is! Only when a change request can no longer be ignored or a bunch of change requests accumulates, the effort of validation will be considered worthwhile. When the validation package is in such a state, you should consider revising it, for reinventing the wheel is quite expensive.

When we look at the patchwork items in our hypothetical household, we will find many small items in different shapes (small validation packages). Some are similar, because they were created by the same person, others are not. There will also be quite a few large blankets (large validation packages), where you can clearly see the contributions of many people in the various sizes and designs of distinct patterns. Many patterns fit well into the original layout of the blanket with slight variations of the same pattern (similar tests). However, there are also patterns which stick out and become a nuisance, because they are badly sewn in or the fabric is damaged (people make mistakes) or they require some stitching every time you want to use that blanket (manual testing). And for some patterns you wonder why on earth they have been included in the first place (duplicates). Usually, when a blanket is full of flaws, you just throw it away. Imagine that we talk about a blanket that has been created and modified for generations. You want to keep some of the nice pieces and you still need a wall-hanging to cover that ugly spot in the living room.

GIVE ME AN EXAMPLE: HOW DID YOU DO IT?

The benefits of revising a validation package can best be shown on a component that is widely used in many statistical programming environments using SAS: autoexec.sas is often standardised to support any type of project and thus has to be maintained continuously in a carefully thought out manner to permit consistent usage throughout the whole environment. In this example, the validation package had grown to a misshapen blanket with all the flaws described above. Programmers were reluctant to modify autoexec.sas because of the immense validation effort. The overall goal for the revision was to simplify the validation, including but not limited to automating it as much as possible.

The first step to revise the validation package for autoexec.sas was a thorough review of the corresponding validation instructions document to get a feel for the target. During this step, tests for different requirements where highlighted in different colors. It soon became apparent that most tests could be grouped into categories, for example tests for libnames, filenames or global variables.

The next step was to set up a new document outlining all the categories and then cut and paste each test to the respective category. Tests covering multiple categories were copied to each related category. Such a simple task seems hardly worth mentioning, but it was a big step forward: The result immediately revealed (a) duplicates which could be deleted so that only unique tests remained, and (b) similarities. Having similar tests grouped together made both the need and the potential for automation obvious. For example, most tests for libname assignments varied only in names, paths, and access privileges, while the actual test was always the same.

The idea of a metadata approach emerged and further revision to the instructions document aimed to support this as far as possible. While keeping records of all required values and options in a simple text file, similar tests were combined into unique tests. For example, 19 tests for libname assignments could be reduced to 3 unique tests which were adapted to allow for a consistent processing of the collected metadata.

The interim result was the reduction of 36 to 16 tests. Of these, 8 more were deleted because after careful consideration of possible consequences (i.e., none) the programming team agreed on the removal of a certain functionality from the component.

The previous steps automatically cleared away many unclear or inconsistently written tests. The remainder of tests were then aligned to be clear and consistent throughout the new instructions document.

After downsizing the actual amount of validation and harmonising the instructions, it was time to implement appropriate automation. For two of the tests the supporting code was retained, though in one place a minor error had to be corrected. For another test the previously manual instructions could be transformed into a script which replaced a simple visual check by an automated one.

All tests for libname, filename, and global variable assignments should follow the metadata approach. For many of these values the old validation package already used a metadata file and scripts testing all at once in different locations. It was important to keep the testing in different locations, as study projects may have different individual settings which all have to be covered by autoexec.sas. However, it was felt that a separation into the categories as described above would be better for maintenance and so the files were split and customised to cover one category at a time. While implementing and optimising these scripts, it became apparent that different SAS versions had to be supported not only by the component autoexec.sas itself but also by the automatic validation scripts. To be pragmatic each test just loops through the supported SAS versions, though some nice SAS 9.2 code to process the metadata could not be used because it was not compatible with SAS 8.2. At that time we found that another test was required to check that format catalogues are properly supported by autoexec.sas with respect to the various supported SAS versions. In addition, autoexec.sas of course should execute without errors or warnings, which was only tested in one or the other test as a side issue, so this became a new test as well.

The revised validation package underwent exhaustive review and discussions. The outcome was that the package was optimised in a way that all but two tests can be run in the same way. Code to run them together was added to the validation instructions document and can just be copied and pasted to the UNIX session. In its current design the instructions document enables the programmer to easily adjust the tests to new or modified functionality, or remove tests for functionality out of use.

Again, we look at our misshapen blanket, that we do not want to keep as such any longer. At first we should simply undo the seams between the patterns and sort them. We will find many patterns that are more or less similar; we can pick out a few we want to keep. We may have to clean them and repair one or the other seam within the pattern, but basically they are fine. However, there will also be patterns where we like the fabrics but the pattern is not nice. In that case we can unravel the patterns, select the best pieces of each and sew them together into new patterns. When we are happy with the individual patterns, we can arrange them loosely, because over time we want to be able to pick out one or the other for cleaning, repairing or exchange. Then we check whether or not that spot on the wall is covered properly, and decide on a nice frame.

WHAT WAS THE OUTCOME?

Table 1 shows a comparison of the old and revised validation package for autoexec.sas. Eight of the nine automated tests can be executed together, leading to an overall execution time of about 45 minutes. While these are run in the background, the programmer can execute the ninth automated test and the one manual test and then continue with some other task. So, the benefits of the revised package clearly are that all but one test can be executed automatically, as well as the tremendous time saving.

	Old validation package	Revised validation package	Change
# of pages in instructions document	42	13	-69%
# of tests	36	10	-72%
- manual	23	1	-96%
- automatic	13	9	-31%
# of files supporting the process	126	33	-74%
Execution time	several hours	less than 1 hour	nameless relief!

Table 1: Comparison of the old and revised validation package

Stepping back from our new wall hanging we see a nice small picture. While in the end our blanket was a pain to use and look at, we can from now on enjoy our most loved pieces in a new design and framework.

WERE THERE CONSEQUENCES FOR THE VALIDATION PROCESS IN GENERAL?

In the midterm the work described in this paper lead to the review of other validation packages, that should be scaled down to unburden the maintenance of the existing environment. In addition, the programming group agreed that automation is crucial for validation when regression testing is the technique of choice.

For the long term the programming team was invited to discuss the whole process in order to shape the future. The goal is to create a validation concept that will be independent of particular tools to be fit for any environment.

The exercise of reutilising our misshapen blanket was a major task, but it was definitely worthwile. The experiences of having it done thoroughly this time will be of great value for renewing other items as well. In the future we may even think about entirely different techniques and utensils to create new patchwork items as needed.

CONCLUSION

When the validation for a component is in the critical state where programmers are reluctant to implement change requests just because the validation is too cumbersome, a programming group should think about their validation concept. It may be a good exercise to revise a validation package and get a feel for the actual needs.

Even for a small component a programmer will never know when a point is reached where testing becomes a burden. When testing is done automatically from the beginning, supporting unique tests, and using a reasonable metadata approach, the whole process is less errorprone and will better support program development and modification. This is even more essential for components of high complexity.

REFERENCES

[Futrell] Robert T. Futrell; Donald F. Shafer; Linda I. Safer (2002):Quality Software Project Management. Prentice Hall

ACKNOWLEDGMENTS

I thank Oliver 'Ollie' Rees for reviewing my abstract and paper, Reinhold Koch for an excellent review of my work that helped me to create a well-wrought validation package, and Bruce Rogers for a great discussion about patchwork quilts that helped me to find a start for writing this paper.

I thank Ian Whatmough and Stephen Frikert for supporting my job rotation from Statistical Programming to Systems Development and Support.

Despite some differences I would also like to thank Monika Engemann for introducing me to the faszinating world of patchwork and quilting which has become an important part of my life.

RECOMMENDED READING

Computer Science and Technology. Validation, Verification, and Testing for the Individual Programmer: Martha A. Branstad, John C. Cherniavsky, W. Richards Adrion. National Bureau of Standards, Depertment of Commerce. Washington, DC. February 1980. Library of Congress Catalog Card Number: 80-600005

CONTACT INFORMATION

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

Anja Feuerbacher
F. Hoffmann - La Roche
CH-4070 Basel

Email: anja.feuerbacher(at)roche.com