Paper 124-2011

SAS Software Development with the V-Model

Andrew Ratcliffe, RTSL.eu, United Kingdom

ABSTRACT

Software development is about building useful systems, not generating reams of documents. The V-Model helps the development team apply focus to what documents are useful (and why) and how much content is appropriate for each.

The V-model offers a framework that clarifies the relationships between requirements, specifications, and testing. This paper discusses the benefits of the V-Model in a variety of differing development processes including waterfall and agile.

INTRODUCTION

Developing software can be difficult enough without development managers or project managers demanding we produce reams of documentation too. If there's a clear purpose to writing something down then the task becomes more palatable - especially useful if we don't feel that we're natural authors or writers. The V-Model attempts to put some structure and *purpose* into the documents that we should *consider* producing during the lifetime of our development project. Whilst it is not a process itself, it helps establish a standard process for creating software.

TARGET

We need to produce software that meets the customers' expectations. We can summarise those expectations as:

- Functional – it allows the customers to get their day jobs done
- Operable on a day-to-day basis – the behind the scenes and/or automated tasks can be done effectively
- Maintainable - problems can be fixed, and enhancements can be made
- Cost effective – all of the above can be done at an affordable and justifiable price

To meet those demands we are going to need to test the code. To test it we are going to need to know what it's meant to do. Knowing what it's meant to do implies that the requirements and design are written down somewhere, but we don't need to get into any arguments over documentation; as we'll see later in this paper, comments in code can adequately substitute for formal documents if the comments are well written (this paper is agnostic on the subject of waterfall versus agile, etc.).

We need a development framework that ensures we collect testable requirements and design.

THE SOLUTION

MEET THE V

We can simplify software development as a series of tasks thus:

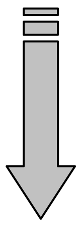
a) customers tell us what they want,

b) we figure-out how we can deliver that,

c) we build it,

d) we make sure we built what we intended to, and

e) the customers check that we've built something that does what they need.



This linear progression can be bent and folded in many ways in order to describe lots of different processes; so, for instance, in iterative projects we may run through the sequence many times, adding extra bits of functionality each time; if we use test driven design we might do some of the steps out of sequence, but at the end of any phase of development we will have completed elements of all five steps a through e. So, in essence, we have to have the steps that we have listed.

The V-model simply takes those five steps and places them onto a V-shape thus:

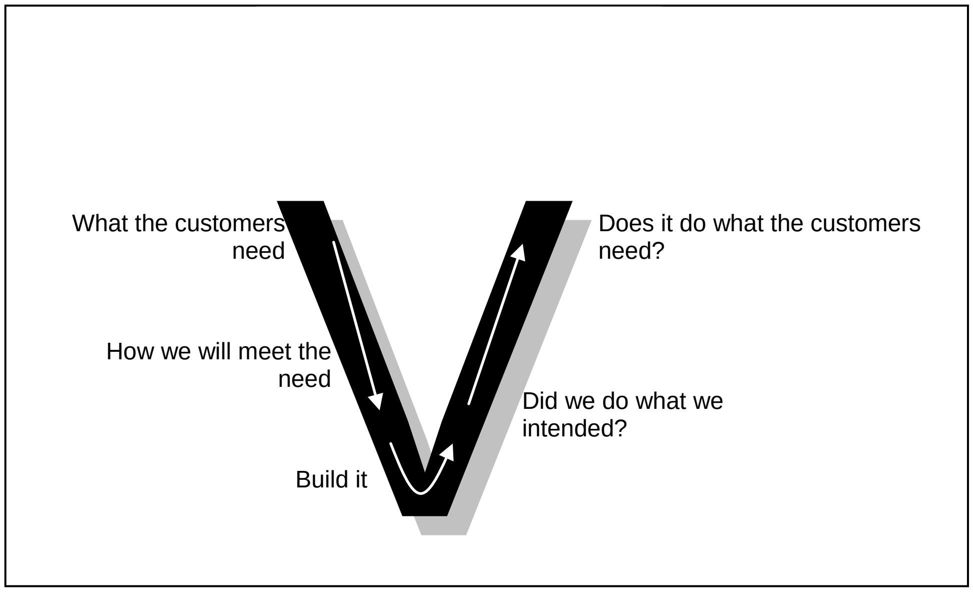

What the customers need
How we will meet the need
Build it
Did we do what we intended?
Does it do what the customers need?

Figure 1. The Basic Flow Through the V

There's a relationship between the elements on the left-hand side. They represent the decomposition of the problem from business-speak into technical speak and ultimately into technical activity (from top to bottom). This is sometimes known as elucidation or *verification*.

We can see that the right-hand side of the V contains the tests of the counterparts on the left-hand side, e.g. "Does it do what the customers needed?" is the test of "What the customers need". This is sometimes known as testing or *validation*.

So the left and right sides of the V capture verification and validation.

Let's expand those activities:

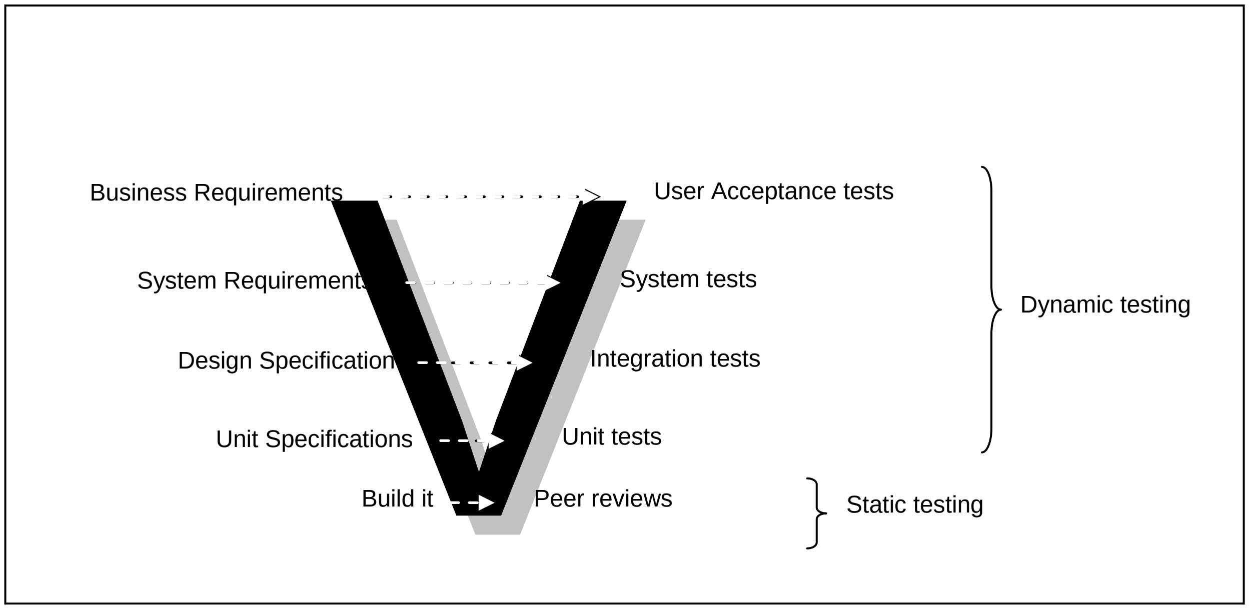


Figure 2. Traceability to testing

The documents are explained in more detail in following sections (and we'll also see that some can be combined to reduce size and effort further), but by highlighting the verification and validation relationships, we can begin to see the purpose of a number of documents. The size and complexity of the project will determine how many physical documents we actually need, but the content will need to be present somewhere.

In summary, there's a transition of requirements down the left-hand side of the V, and there's a testing relationship going across the V.

WHAT IS NEEDED

The **Business Requirements** are a statement by the customer of what the system shall achieve in order to meet the need. These involve both functional and non-functional requirements.

These requirements establish what the ideal system has to perform in terms of the business processes (the day-to-day tasks) of the customer. The Business Requirements are usually collected by a series of interviews with users or customers conducted by a business analyst. Tools used in creating the Business Requirements often include Use Cases and workshops.

The User Acceptance Tests (UAT) are derived from the business requirements.

The **System Requirements** describe the full range of attributes of the system such as the system's functional, physical, interface, performance, data and security requirements. These are derived from the business requirements, typically by the business analyst. The system requirements are used by the business analyst to communicate their understanding of the system back to the customers. The customers must carefully review this reinterpretation of the business requirements because the system requirements will serve as the guideline for the system designers in producing the design specification.

The system tests are derived from the system requirements.

The business and system requirements specify what the system must do. They do not attempt to spell-out how the system will meet these requirements. It's very easy to fall into solution mode and begin to make design decisions but this temptation must be avoided at all costs.

HOW IT CAN BE ACHIEVED

So, the business requirements specify what the customers need (in their own language) and the system requirements echo that back in a technical interpretation of those requirements. Neither of these collections says *how* the system will meet the requirements. That's the job of the specifications.

The **Design Specification** is where we analyse and understand the purpose of the system by studying the system requirements (with reference to the business requirements too). We figure out possibilities and techniques by which the system requirements can be implemented. Clearly, if any of the requirements seem infeasible or are unlikely to be justifiable in terms of the cost of designing and building them, we must revert to the customers. We can discuss and resolve the issues and update the requirements accordingly.

The design specification incorporates the high-level design and contains the general system organisation, data structures, data flow, process flow, etc. It may also hold example reports for increased clarity and understanding. Other technical documentation like entity diagrams and a data dictionary will also be produced as part of the design specification. The design specification will typically include a list of modules, brief functionality of each module, their interface relationships, dependencies, tables, etc.

The unit specifications provide the design of each individual unit (or module). These unit-level designs are sometimes known collectively as the low-level design. The design specification broke the system down into small units and each of them is explained in their respective unit specification so we could start coding straight away. The unit specifications will usually contain a detailed functional logic of the module (in pseudo code), data sets with columns, lengths and formats, all interface details (e.g. stored processes and macros) including inputs and outputs, and a list of messages (including error messages) that the unit may issue.

BUILD IT

We might guess that this is the stage that needs least explanation! We can follow the unit specifications one at a time and build the units one at a time. Or we can get a room full of programmers sufficient for each programmer to independently build one unit!

BE SURE WE BUILT WHAT WE INTENDED

Before we release our lovely new shiny code to the customer, we better check that it works in a manner that matches our specifications.

The first phase of testing should be **Peer Review**. This is typically a static test whereby a fellow programmer will take a look at our code and offer opinion on whether it meets the team's coding guidelines and other standards, and whether it gives the impression of meeting the unit specification. Peer review should not focus on being a destructive critique of our code, it should be constructive and should offer additional benefits such as an opportunity to share knowledge and experience of the overall system and of programming techniques in general.

Unit testing involves checking that each feature in the unit specifications has been implemented successfully in the module. Unit testing is usually white box testing, i.e. it requires knowledge of the code. This should be done by a different programmer but is often done by the coder themselves.

If a unit is reliant on one or more other units then the tester may prepare dummy units so that the emphasis is on testing just the one real unit, or it may be pragmatic to skip to integration testing for some units.

Integration Testing involves executing groups of units in order to test their interaction and to see if they behave as predicted by the Design Specification. Units can only be integration tested once they have successfully been unit tested! Integration tests can be planned in a layered fashion so that they test increasingly large groups of units until the full system is constructed and tested.

Integration testing is focused upon the communication between the units, not their individual behaviour. We wish to test that the units co-exist and cooperate as planned, under error conditions as well as the "happy path".

BE SURE IT DOES WHAT THE CUSTOMERS NEED

The V-Model shows **System Testing** opposing the system requirements. System testing is about checking the system as a whole, not about checking the individual parts of the design. It treats the whole system as one big unit.

System testing can involve a number of specialist types of test to see if all the functional and non-functional requirements have been met. In addition to functional requirements these may include the following types of testing for the non-functional requirements:

- Performance - Are the performance criteria met?
- Volume - Can large volumes of information be handled?
- Stress - Can peak volumes of information be handled?
- Documentation - Is the documentation usable for the system?
- Robustness - Does the system remain stable under adverse circumstances?

Finally, **User Acceptance Testing** (UAT) is an opportunity for the customer to check that the system does what they need. It may sound like system testing but there's a key difference: Systems testing checks that the system that was specified has been delivered, UAT checks that the system delivers what was requested. As its name implies, UAT is done by the users. The users should check that the system fits into the business processes that they choose to employ, and they should check that the documentation and training is adequate and fit for purpose.

DOING WHAT'S APPROPRIATE

It's important we understand the big picture, but it's also important to apply the rule of "appropriateness" to what we do. We just discussed at least eight documents, but it is not necessary to produce eight documents for each separate project that we work on.

It is quite common to deliver one document for the left-side of the V, and another for the right-side. These documents can be known as the Project Specification and the Test Specification, but names and terminology vary.

One of the most common concerns is "how much detail do I need to put into each document". Well, we can't answer that without being clear on what the purpose of the document is. We've now done that: each document typically has two purposes, i.e. to inform the next stage of verification/elucidation and to inform the associated stage of validation/testing. A third audience we should remember is the maintenance programmer (the poor soul who has to fix bugs or add features six months down the line after you've moved to a new project, or employer, or retired). And having established those facts, we can now confidently fill the documents with the appropriate amount of detail, i.e. we should provide the *barely adequate* amount of detail to each.

Barely adequate is a good phrase. Adequate tells us that we've done the job; it means the target has been met. Barely tells that we only just got across the line; there was no wastage or unnecessary effort.

FITTING THE V TO THE PROCESS

The information that the V-Model framework specifies can be collected and stored in a variety of different ways and means. It is by no means the case that each of the stages (such as Business requirements and Design Specification) have to be a separate document. Indeed, we're not saying that any of this has to be typed into a Word Processor. If we're content that the code comments encapsulate some or all of what we've described (and if our coding guidelines permit) then we may not need to produce any documents for elucidation/verification.

It's less easy to see how we can avoid describing what we intend to do for tests outside of a document, and there's usually a demand to keep evidence of successful test runs, but each site is different so we must judge each case on its merits.

TRACEABILITY

By assigning unique IDs to each business requirement and to each system requirement, we can build a "traceability matrix" to check that each business requirement has at least one system requirement that is intended to meet the need. Thus we can use the traceability matrix to be sure that we have covered the system's business requirements in full within the system requirements document, and we can also easily see if system requirements have been added without any associated business requirement (that could scope creep).

We can do the same between system requirements and design specification, between design specification and unit specification, and between unit specification and code. How much of this we actually do will be dependent on the effort versus the benefit for our specific project.

Having assigned unique IDs to each element on the left-side of the V, it's easy to see how we also apply traceability to our testing plans in order to be sure we have tested every element from the left-side of the V.

BACKGROUND AND HISTORY

The creation (invention?) of the V-model seems to be a happy coincidence of events in the US and in Germany. Around the end of the 1980s, two separate software development teams happened upon the same fundamental idea and produced it as the V-model.

The V-model is now in wide use around the world, providing structure and focus to numerous projects.

CONCLUSION

The V-Model is a framework, not a process. It can be used with a wide variety of the popular development processes, from water fall to agile. It does not specify what *documents* are required, it merely provides guidance on what information should be collected, why that information should be collected, and how that information relates to other information (whether it be relating to higher or lower degrees of elucidation/verification, or whether it relates to testing/validation).

Adopting the V-Model framework as part of our development process:

- facilitates greater control due to standardisation of products in the process
- delivers an increase in quality due to the ease with which templates and examples can be produced and due to familiarity with the process's products
- easier cost estimation due to the repeatability of the process

Adopt the V-Model. V is for victory!

APPENDIX A. TESTING – SOME SAS® SOFTWARE SPECIFICS

This paper is presented in the Coders' Corner stream of SAS Global Forum 2011. This appendix contains some SAS coding tips related to the unit testing phase.

Testing an individual unit such as a macro is not restricted to checking that the unit provides the functionality required; testing should also examine whether the unit has been well-built and follows coding guidelines. Those coding guidelines might include requirements to clean-up the environment after execution. In practice, this can mean that each unit should delete temporary elements that were created, such as data sets & library assignments, and no unintended creation of other objects (such as global macro variables).

The following test harness (THARNESS) macro will execute a specified macro and report upon whether the macro-under-test created (or removed) any macro variables and/or library assignments. As such, the THARNESS macro is a skeleton and forms an example of what can be done to automate some elements of unit testing. The principles demonstrated by the macro are easily adopted for other elements of the environment, etc.

THE TEST HARNESS DEFINITION

Key features of the THARNESS macro are as follows:

Echoes all of its parameters upon initiation	Using `%put _local_`, the macro echoes the values of all of its parameters to the log at initiation. This ensures that the values of those parameters taking default values (and hence not specified in the calling program) are known to anybody inspecting the log
Deletes its own temporary WORK data sets	In keeping with the implied coding guidelines, the THARNESS macro deletes its own WORK data sets prior to termination. This is facilitated by the fact that the names of all of the macro's WORK data sets are prefixed with _THARNESS_ (achieved generically with _&sysmacroname._). By using the same prefix, the data sets can be deleted generically at the end of the macro by specifying _THARNESS_: on PROC DATASETS' DELETE statement. However, this functionality is optional, based upon the THARNESS macro's TIDY parameter.
Internal workings are hidden	OPTIONS NONOTES is used to hide the test harnesses activities from the log. This allows the reader of the log to focus upon a) execution of the macro-under-test, and b) the results from THARNESS. Ideally, the initial value of the NOTES option would be captured prior to changing it to NONOTES, and then subsequently restored to its original value. The GETOPTION function can be used for this purpose. Currently the code simply specifies OPTIONS NOTES in order to "restore" it.
Limited to global macro variables and library assignments	The functionality of the skeleton/example macro is limited to reporting upon whether the macro-under-test created (or removed) any macro variables and/or library assignments. The principles demonstrated by the macro are easily adopted for other elements of the environment, etc.
Basic approach is to take snapshot of environment before and after	A snapshot of relevant elements of the environment is taken before and after execution of the macro-under-test. In practice this means taking copies of sashelp.vmacro and sashelp.vslib. The comparison of the before and after images are then done in DATA steps and reported using PUT statements. This is more flexible than using PROC COMPARE, for example.

The macro definition is listed below.

```
%macro tharness(testmacro =
               ,tidy      = y
               );
  %put &sysmacroname: Parameters received by this macro are:;
  %put _local_;
  %put ;

  options nonotes;
  /********************************/
  /* Collect the BEFORE image(s) */
  /********************************/
  Data work._&sysmacroname._macroBefore;
    Set sashelp.vmacro;
    Where scope not in ("&sysmacroname",'AUTOMATIC');
  Run;

  Data work._&sysmacroname._vslibBefore;
    Set sashelp.vslib;
  Run;
  options notes;
```

```
/********************/
/* Execute the test */
/********************/
%put &sysmacroname: Starting execution of code to be tested;
%&testmacro;
%put &sysmacroname: Execution of code to be tested has finished;
%put ;

/******************************/
/* Collect the AFTER image(s) */
/******************************/
options nonotes;
Data work._&sysmacroname._macroAfter;
  Set sashelp.vmacro;
  Where scope not in ("&sysmacroname",'AUTOMATIC');
Run;

Data work._&sysmacroname._vslibAfter;
  Set sashelp.vslib;
Run;

/*************************/
/* Summarise the results */
/*************************/
%put &sysmacroname: Summarising results;
%put ;

/* Macro Variables */
%put &sysmacroname: Summary of results for macro variables:;
proc sort data=work._&sysmacroname._macroBefore;
  by scope name;
run;

proc sort data=work._&sysmacroname._macroAfter;
  by scope name;
run;

data _null_;
  merge work._&sysmacroname._macrobefore (in=before)
        work._&sysmacroname._macroafter (in=after)
        end=finish;
  by scope name;
  retain IssueFound 0;
  if before and not after then
  do;
    put "&sysmacroname: Macro variable has been removed: " scope= name=;
    IssueFound=1;
  end;
  else if not before and after then
  do;
    put "&sysmacroname: Macro variable has been added: " scope= name=;
    IssueFound=1;
  end;
  if finish and not IssueFound then
    put "&sysmacroname: No macro variable issues found";
run;
%put ;

/* Library Assignments */
%put &sysmacroname: Summary of results for library assignments:;
proc sort data=work._&sysmacroname._vslibBefore;
  by libname;
run;
```

```
  proc sort data=work._&sysmacroname._vslibAfter;
    by libname;
  run;

  data _null_;
    merge work._&sysmacroname._vslibbefore (in=before)
          work._&sysmacroname._vslibafter (in=after)
          end=finish;
    by libname;
    retain IssueFound 0;
    if before and not after then
    do;
      put "&sysmacroname: Library assignment has been removed: " libname=;
      IssueFound=1;
    end;
    else if not before and after then
    do;
      put "&sysmacroname: Library assignment has been added: " libname=;
      IssueFound=1;
    end;
    if finish and not IssueFound then
      put "&sysmacroname: No library assignment issues found";
  run;
  %put ;

  %put &sysmacroname: End of results summary;

  %if %upcase(%substr(&tidy,1,1)) eq Y %then
  %do;
    proc datasets lib=work nolist;
      delete _&sysmacroname._: ;
    quit;
    options notes;
  %end;

%mend tharness;
```

THE TEST HARNESS EXECUTION

The following code uses the Test Harness to execute the macro-under-test, i.e. %BestCodeEver.

```
/* Execute the harness on the macro-under-test: %BestCodeEver */
%tharness(testmacro=BestCodeEver);
```

THE TEST HARNESS LOG OUTPUT

The log output shown below illustrates how the Test Harness is able to report upon the newly-created global macro variable (NLOBS) and library assignment. Reference to the design of the unit may show that the creation of the global macro variable NLOBS was intended functionality but the assignment of the library may not be (perhaps it was intended as a temporary assignment and should have been cleared prior to the termination of the macro-under-test).

```
141        %tharness(testmacro=BestCodeEver);
THARNESS: Parameters received by this macro are:
THARNESS TESTMACRO BestCodeEver
THARNESS TIDY y

THARNESS: Starting execution of code to be tested

nlobs=19
NOTE: The data set WORK.COUNT has 1 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

```
NOTE: Libref NFIND_ME was successfully assigned as follows:
      Engine:        V9
      Physical Name: C:\SAS\Config\Lev1\SASApp
NLOBS=2.7481588701
THARNESS: Execution of code to be tested has finished

THARNESS: Summarising results

THARNESS: Summary of results for macro variables:
THARNESS: Macro variable has been added: scope=GLOBAL name=NLOBS

THARNESS: Summary of results for library assignments:
THARNESS: Library assignment has been added: libname=NFIND_ME

THARNESS: End of results summary
```

AUTHOR BIOGRAPHY

Andrew is Managing Director of RTSL.eu, a leading European SAS specialist consultancy. Having first used SAS in 1983, Andrew's experience and knowledge covers breadth as well as depth. Andrew shares his experience through his www.NoteColon.info blog.

CONTACT INFORMATION

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

Name: Andrew Ratcliffe
Enterprise: Ratcliffe Technical Services Limited (RTSL.eu)
Address: Willow Close, Bexley, Kent, DA5 1QY, United Kingdom
Work Phone: +44-1322-525672
Web: www.RTSL.eu / www.NoteColon.info