New Models for Test Development

Brian Marick
Testing Foundations[1]
marick@testing.com
www.testing.com

A software testing model summarizes how you should think about test development. It tells you how to plan the testing effort, what purpose tests serve, when they're created, and what sources of information you use to create them. A good model guides your thinking; a bad one warps it.

I claim that most software testing models are bad ones.

They're bad because they're mere embellishments on software *development* models. People think hard and argue at length about how to do software development. They invent a model. Then they add testing as an afterthought.

A testing model has to be driven by development – after all, we're testing their work. But when the testing model is an afterthought, it's driven in the wrong way. It connects to development activities in the places that are easiest to describe, not those that give testing the most leverage. It builds upon what developers *ought* to do, not upon what they always do even when they're not following the rules. It lumps together activities that have no essential connection and separates others that belong together. It suffers from all the flaws of hasty thinking.

The result is ineffective and (especially) inefficient testing. Ineffective testing misses bugs. Inefficient testing wastes money.

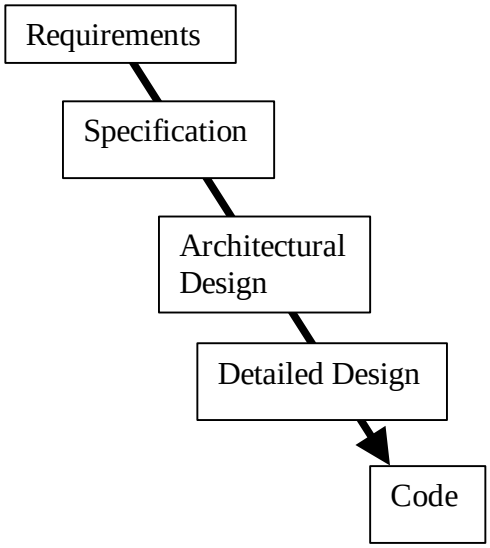
In this paper, I'll do two things. First, I'll attempt to demolish a bad model, the quite popular "V model". In the process, I hope to banish the phrases "unit testing" and "integration testing" from our vocabularies. Second, I'll describe a model I think is better. But my primary purpose is not to claim I have the right model – it's too early for that – but to describe important requirements for test models that will replace mine. Those requirements are summarized at the end of the paper.

What's wrong with the V model?

I will use the V model as my example of a bad model. I use it because it's the most familiar.

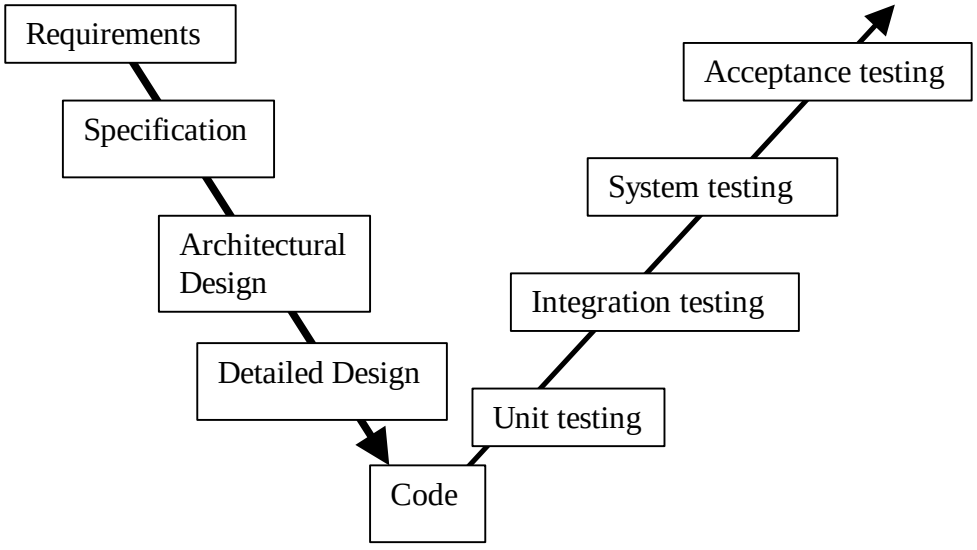
[1] This paper was written while I was an employee of Reliable Software Technologies (www.rstcorp.com). I have been granted permission to reproduce it.

A typical version of the V model begins by describing software development as following the stages shown here:



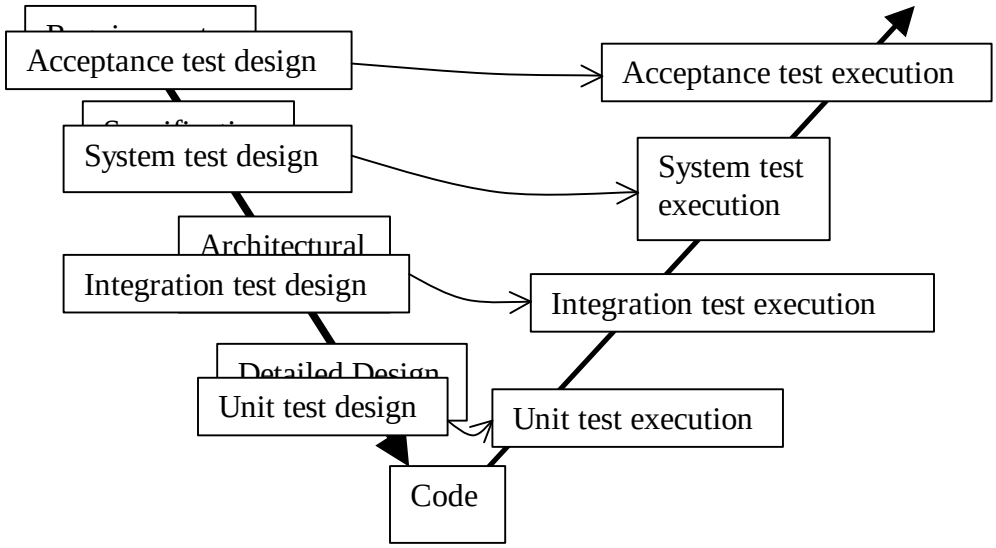
That's the age-old waterfall model. As a development model, it has a lot of problems. Those don't concern us here – although it is indicative of the state of testing models that a development model so widely disparaged is the basis for our most common testing model. My criticisms also apply to testing models that are embellishments on better development models, such as the spiral model [Boehm88].

Testing activities are added to the model as follows:



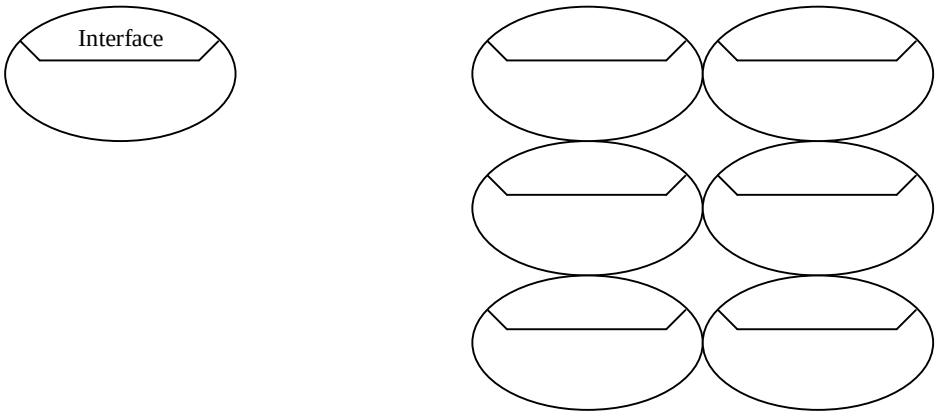
Unit testing checks whether code meets the detailed design. Integration testing checks whether previously tested components fit together. System testing checks if the fully integrated product meets the specification. And acceptance testing checks whether the product meets the final user requirements.

To be fair, users of the V model will often separate test design from test implementation. The test design is done when the appropriate development document is ready. That looks like this:



This model, with its appealing symmetry, has led many people astray. I'll concentrate on the confusion caused at the unit and integration levels.

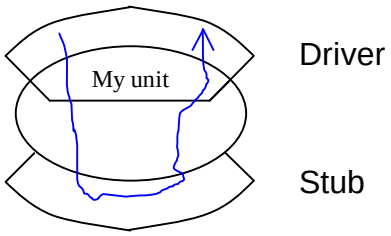
Here, I show a picture of a unit and of an aggregation of units, which I will call a *subsystem*.



There's always some dispute over how big a unit should be (a function? a class? a collection of related classes?) but that doesn't affect my argument. For my purposes, a unit is the smallest chunk of code that the developers can stand to talk about as an independent entity.

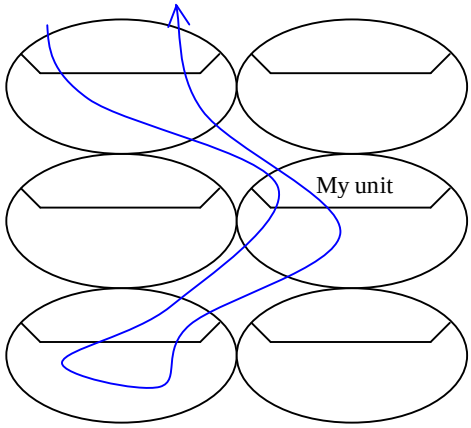
The V model says that someone should first test each unit. When all the subsystem's units are tested, they should be aggregated and the subsystem tested to see if it works as a whole.

So how do we test the unit? We look at its interface as specified in the detailed design, or at the code, or at both, pick inputs that satisfy some test design criteria, feed those inputs to the interface, then check the results for correctness. Because the unit usually can't be executed in isolation, we have to surround it with stubs and drivers, as shown at the right. The arrow represents the execution trace of a test.



That's what most people mean when they say "unit testing".

I think that approach is sometimes a bad idea. The same inputs can often be delivered to the unit through the subsystem, which thus acts as both stub and driver. That looks like the picture to the right.

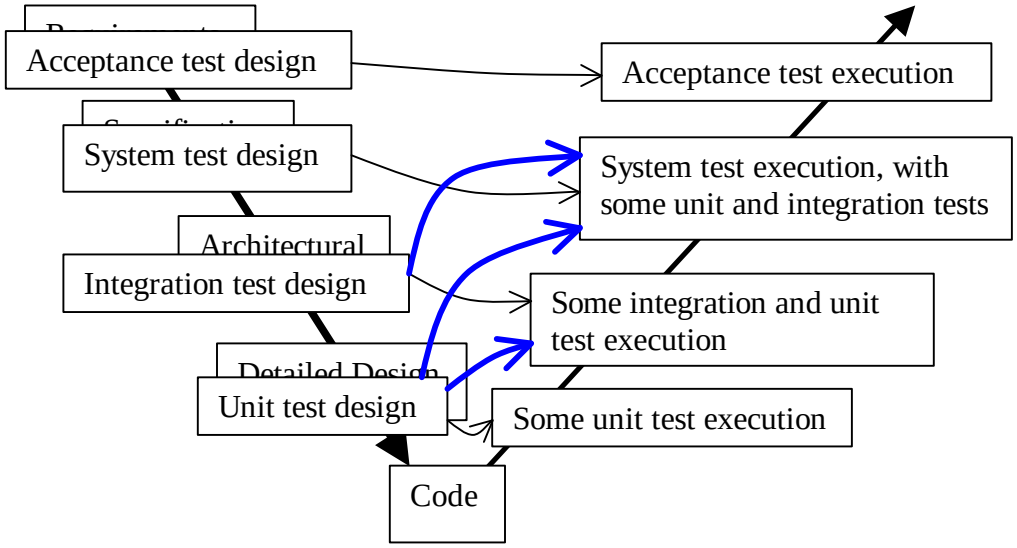


The decision about which approach to take is a matter of weighing tradeoffs. How much would the stubs cost? How likely are they to be maintained? How likely are failures to be masked by the subsystem? How difficult would debugging through the subsystem be? If tests aren't run until integration, some bugs will be found later. How does the estimated cost of that compare to the cost of stubs and drivers? And so forth.

The V model precludes these questions. They don't make sense. Unit tests get executed when units are done. Integration tests get executed when subsystems are integrated. End of story. It used to be surprising and disheartening to me how often people simply wouldn't think about the tradeoffs – they were trapped by their model.

Therefore, **a useful model must allow testers to consider the possible savings of deferring test execution**.

A test designed to find bugs in a particular unit might be best run with the unit in isolation, surrounded by unit-specific stubs and drivers. Or it might be tested as part as the subsystem – along with tests designed to find integration problems. Or, since a subsystem will itself need stubs and drivers to emulate connections to other subsystems, it might *sometimes* make sense to defer both unit and integration tests until the whole system is at least partly integrated. At that point, the tester is executing unit, integration, and system tests through the product's external interface. Again, the purpose is to minimize total lifecycle cost, balancing cost of testing against cost of delayed bug discovery. The distinction between "unit", "integration", and "system" tests begins to break down. In effect, you have this picture:[2]

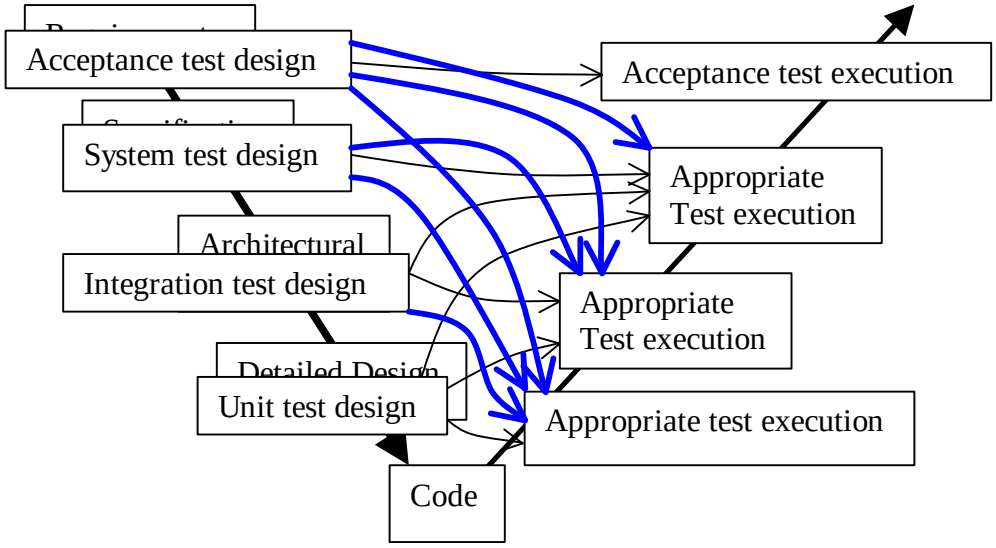


It would be better to label each box on the right "execution of appropriate tests" and be done with it.

What of the left side? Consider system test design, an activity driven by the specification. Suppose that you knew that two units in a particular subsystem, working in conjunction, implemented a particular statement in the specification. Why not test that specification statement just after the subsystem was integrated, at the same time as tests derived from the design? If the statement's implementation depends on nothing outside the subsystem, why wait until the whole system is available? Wouldn't finding the bugs earlier be cheaper?

In the previous diagram, we had arrows pointing upward (effectively, later in time). It can also make sense to have arrows pointing downward (earlier in time):

[2] I'm leaving acceptance test execution alone. It's usually done by a customer. Since it's an activity of a different group, I exclude it from this discussion.



In that case, the boxes on the left might be better labeled "Whatever test design can be done with the information available at this point".

Therefore, **when test design is derived from a description of a component of the system, the model must allow such tests to be executed before the component is fully assembled.**

I have to admit my picture is awfully ugly – all those arrows going every which way. I have two comments about that:

1. We are not in the business of producing beauty. We're in the business of finding as many serious bugs as possible as cheaply as possible.
2. The ugliness is, in part, a consequence of assuming that the order in which developers produce system description documents, and the relationships among those documents, is the mighty oak tree about which the slender vine of testing twines. If we adopt a different organizing principle, things get a bit more pleasing. But they're still complicated, because we're working in a complicated field.

The V model fails because it divides system development into phases with firm boundaries between them. It discourages people from carrying testing information across those boundaries. Some tests are executed earlier than makes economic sense. Others are executed later than makes sense.

Moreover, it discourages you from combining information from different levels of system description. For example, organizations sometimes develop a fixation on "signing off" on test designs. The specification leads to the system test design. That's reviewed and signed off. From that point on, it's done. It's not revised unless the specification is. If information relevant to those tests is uncovered later – if, for example, the architectural design reveals that some tests are redundant – well, that's too bad. Or, if the detailed design reveals an internal boundary that could easily be incorporated into existing system tests, that's tough: separate unit tests need to be written.

Therefore, **the model must allow individual tests to be designed using information combined from various sources.**

And further, **the model must allow tests to be redesigned as new sources of information appear.**

A different model

Let's step back for a second. What is our job?

There are times when some person or group of people hands some code to other people and says, "Hope you like it." That happens when the whole project puts bits on a CD and gives them to customers. It also happens within a project:

- One development team says to other teams, "We've finished the XML enhancements to the COMM library. The source is now in the master repository; the executable library is now in the build environment. The XARG team should now be unblocked – go for it!"
- One programmer checks in a bug fix and sends out email saying, "I fixed the bug in allocAttList. Sorry about that." The other programmers who earlier stumbled over that code can now proceed.

In all cases, we have people handing **code** to **other people**, possibly causing them **damage**. Testers intervene in this process. Before the handoff, testers execute the code, find bugs (the damage), and ask the question, "Do you really want to hand this off?" In response, the handoff may be deferred until bugs are fixed.

This act is fundamental to testing, regardless of the other things you may do. If you don't execute the code to uncover possible damage, you're not a tester.

Our test models should be built around the essential fact of our lives: code handoffs.

Therefore, **a test model should force a testing reaction to every code handoff in the project.**

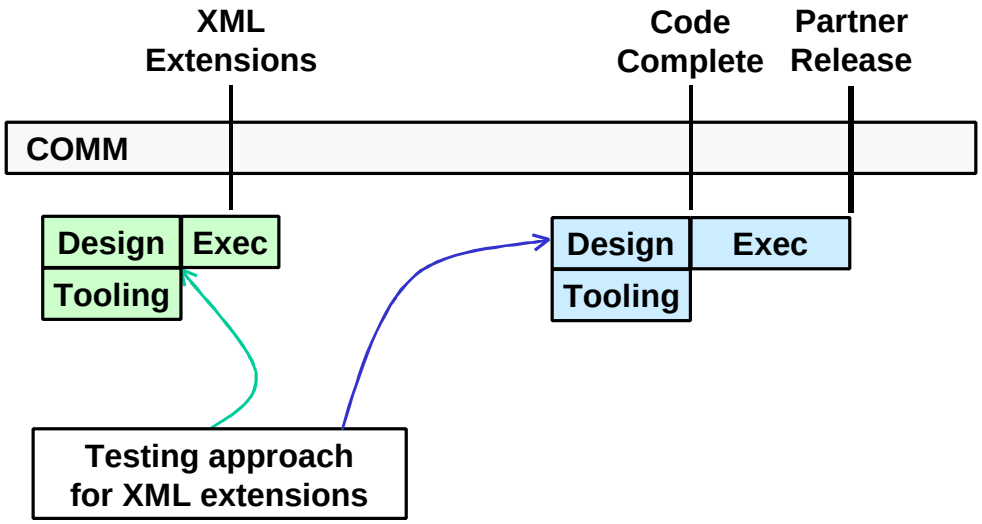
I'll use the XML-enhanced COMM library as an example. That's a handoff from one team to the rest of the project. Who could be damaged?

- It might immediately damage the XARG team, who will be using those XML enhancements in their code.
- It might later damage the marketing people, who will be giving a demonstration of the "partner release" version of the product at a trade show. XML support is an important part of their sales pitch.
- Still later, it might damage a partner who adopts our product.

We immediately have some interesting test planning questions. The simple thing to do would be to test the XML enhancements completely at the time of handoff. (By "completely," I mean "design as many tests for them as you ever will.") But maybe some XML features aren't required by the XARG team, so it makes sense to test them through the integrated partner release system. That means moving some of the XML-inspired testing to a later handoff. Or we might move it later for less satisfying reasons, such as

that other testing tasks must take precedence in the near term. The XARG team will have to resign itself to stumbling over XML bugs for a while.

Our testing plan might be represented by a testing schedule that annotates the development timeline:



Based on our understanding of the change, we've committed to doing some testing around the time the XML extensions are handed off. Test design and test support work precedes test execution. Other XML testing has been deferred to the project-wide "Code Complete" milestone, which is when all of the subsystems are integrated together and the whole product is stabilized to create the version for the trade show.

For clarity, two things aren't shown at the Code Complete milestone:

- There's a great deal of other test execution (and design and tooling) that happens there. It was deferred from other handoffs of subsystems other than COMM. Moreover, there are tests for the milestone's specific dangers. For example, there might be a set of tests that runs through the marketer's demo script, including all deviations she might inadvertently make. The goal is that she will not stand in front of an audience of one thousand and be the first person to ever try a particular sequence of inputs.
- Some of the first handoff XML tests will be re-executed at the Code Complete milestone.

My point is that test planning is made up of hard decisions about staffing, machine resources, allocation of time to test design, the amount of test support code to produce, which tests should be automated, and so forth. All those hard decisions should be driven by information about what's new in individual handoffs. If there were only one handoff, you'd work your way backwards from it:

1. Analyze risks. Who could be damaged by this change, and in what ways?
2. Decide on a test approach that addresses the specific risks.

3. Estimate the test design and implementation cost and schedule.
4. At the appropriate point in the project schedule, put the plan into action:
   A. Begin designing tests…
   B. … while also designing and creating any test support code…
   C. … and quite likely executing some tests before all have been designed.

Since there's more than one handoff, there's a potentially complex interplay between the planning driven by each handoff. Staffing has to be balanced. Test support code and tools have to be shared among tasks. You must consider to what extent tests designed for earlier handoffs will need to be re-executed on later ones.[3] And so forth.

That sounds complicated. It sounds like there's too much to keep track of, too much that could be overlooked. It may seem as if I'm asking you to perform the thinking process required by an IEEE 829 [IEEE98] Test Plan for each handoff, then merge all those thoughts into one test plan that spreads handoff-specific testing tasks across the project.

Yes and no. Thinking takes time. Too much planning can lead to diminishing returns. At some point, you need to stop planning and start acting. For example, you cannot write – or even think of – an elaborate test plan for each bug fix, even though a bug fix is a handoff.

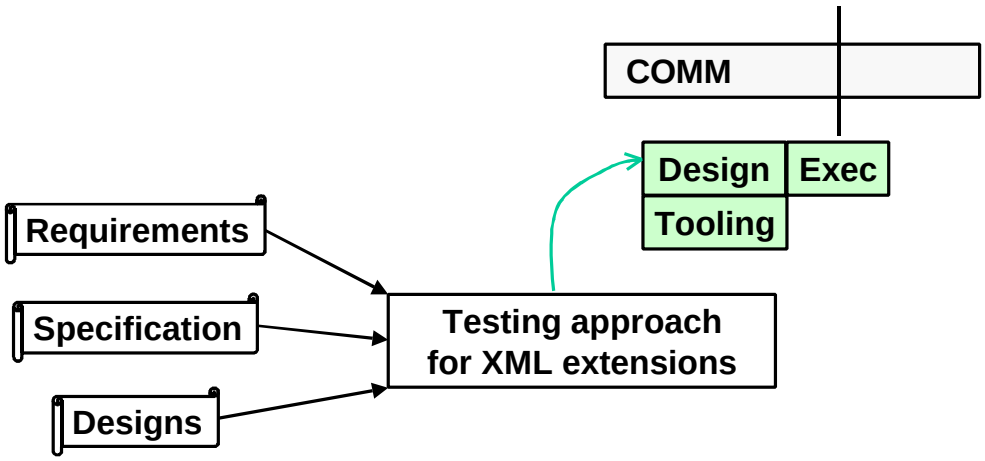
But bug fixes are a fact of life. The overall project test plan should address them. It should insist that you have a default procedure for all bug fixes. That procedure should include a build verification ("smoke test") suite that's run against each proposed fix. Effort should be devoted to thinking through what makes a good suite. Too often, build verification suites are slapped together haphazardly.

To be realistic, a model must allow some rote behavior ("do the following things for any handoff of type *X*") while encouraging just enough examination of what's special about particular handoffs. The smaller the risk of the handoff, the more rote the behavior.

A model that is explicit about the fundamental realities of testing simply must work better than a model that ignores them, that abstracts away all the real complexity of your job. As another example, there's the matter of project documentation.

I haven't mentioned requirements, specification, or design documents yet. Instead, the discussion has been driven by a list of changes in a handoff. What's the role of those documents? They're used like this:

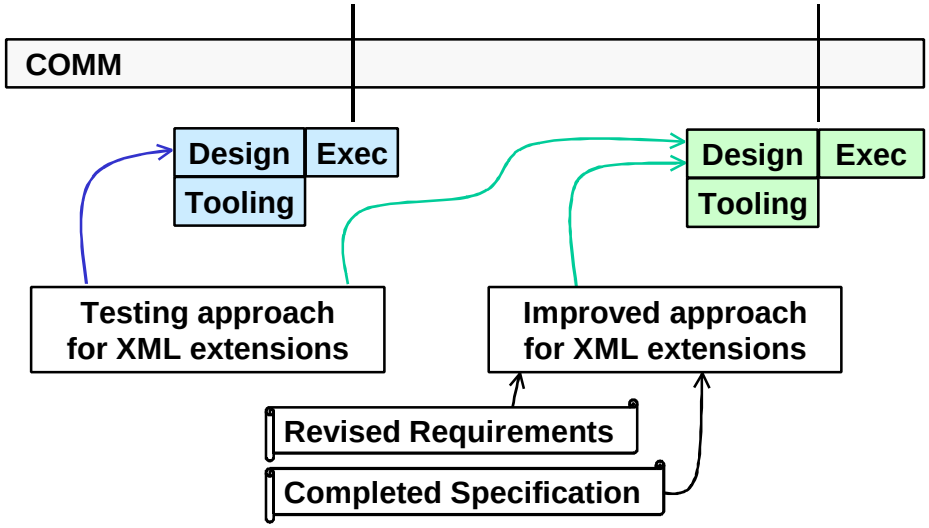
[3] Tests should be re-executed on a later handoff if the changes made as a part of that handoff are likely to cause bugs that the re-executed tests would find. Predicting which tests have value when re-executed is an interesting and tough problem. It's nowhere near as simple as "repeat them all" – the economics of that strategy aren't good. See [Marick98].



Documents guide how you react to a handoff's changes. If you had a *good* requirements document, it would be a description, perhaps indirect, of the problem the product solves. That would help you analyze risk. A *good* specification would describe the system's behavior. That would help you convert your test approach into specific tests. A *good* architectural design would help you understand the ramifications of a change: what other parts of the system might be affected? What tests need to be re-executed?

I don't see good documents very often. A requirements document is likely to be a marketing feature checklist. A specification is user documentation, delivered after the code is done. Design documents don't exist.

That's OK. By concentrating on the list of changes in a handoff, I've decoupled the testing process from the software design process. If the XML additions to COMM are poorly described, I'll design the best tests I can with what I know. If, later in the project, the XML-relevant user documentation becomes available, I'll add more testing effort to a later handoff. If the marketing requirements change – as they so often do – I'll add or remove testing effort later. All that looks like this:



Therefore, **the test effort may be degraded by poor or late project documentation, but it should not be blocked entirely.**

A savvy tester doesn't trust the documentation anyway. After all, the whole point of testing is that people make mistakes. Well, didn't people write those documents?

Since "official" documents are weak, **the test model must explicitly encourage the use of sources of information other than project documentation during test design.**

Testers should talk to developers, users, marketers, technical writers, and anyone else who can give clues about better tests. Testers should attempt to immerse themselves in the culture that builds up around any technology. For example, I would expect testers working with XML to stay current with the World Wide Web Consortium's XML links (http://www.w3.org/XML/) and other XML sites and mailing lists, even including quirky ones like Dave Winer's DaveNet / Scripting News (http://www.scripting.com/). These should not be "side channels," unacknowledged sources of information. They should be sources that are planned and scheduled.

The executing tests are themselves a useful source of information. Good testers read bug reports carefully because they teach about weaknesses in the system. In particular, they often give a feel for the implications of architectural decisions that the official architectural design cannot. The act of executing tests should yield at least a trickle of new test ideas; it's a poor model that doesn't take that into account.

Therefore, **the test model must include feedback loops so that test design takes into account what's learned by running tests.**

The real complexity in our jobs is that all planning is done under conditions of uncertainty and ignorance. The code isn't the only thing that changes. Schedules slip. New milestones are added for new features. Features are cut from the release. During development, everyone – marketers, developers, and testers – comes to understand better what the product is really *for*. Given all that, how can the first version of the test plan possibly be right?

Therefore, **the model must require the test planner to take explicit, accountable action in response to dropped handoffs, new handoffs, and changes to the contents of handoffs.**

Summary

The V model is fatally flawed, as is any model that:

1. ignores the fact that software is developed in a series of handoffs, where each handoff changes the behavior of the previous handoff,
2. relies on the existence, accuracy, completeness, and timeliness of development documentation,
3. asserts a test is designed from a single document, without being modified by later or earlier documents, or
4. asserts that tests derived from a single document are all executed together.

I have sketched – but not elaborated – a replacement model. It organizes the testing effort around code handoffs or milestones. It takes explicit account of the economics of testing: that the goal of test design is to discover inputs that will find bugs, and that the goal of test implementation is to deliver those inputs in *any* way that minimizes lifecycle costs. The model assumes imperfect and changing information about the product. Testing a product is a learning process.

In the past, I haven't thought much about models. I ostensibly used the V model. I built my plans according to it, but seemed to spend a lot of my time wrestling with issues that the model didn't address. For other issues, the model got in my way, so I worked around it.

I hope that thinking explicitly about requirements will be as useful for developing a testing model as it is when developing a product. I hope that I can elaborate on the model presented in this paper to the point that it provides as much explicit guidance as the V model *seems* to.

Acknowledgements

James Bach first made me realize that test plans have to take handoffs into account. Noel Nyman and Johanna Rothman made helpful comments on a draft. Kamesh Pemmaraju and Carol Stollmeyer do not let me get away with hand-waving explanations. They want to know how ideas will work, in detail, in real life, on real projects. I sense they have only begun to force me to think things through. I look forward to it.

References

[Boehm88]
Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May, 1988.

[IEEE98]
"IEEE Standard for Software Test Documentation," IEEE Std 829-1998, The Institute of Electrical and Electronics Engineers, 1998.

[Marick98]
Brian Marick, "When Should a Test be Automated?" Proceedings of International Quality Week, May, 1998.
ftp://ftp.rstcorp.com/pub/papers/automate.pdf

A checklist of model requirements

These requirements are grouped according to testing activity. They were presented in a different order in the text.

A test model should:

1. force a testing reaction to every code handoff in the project.
2. require the test planner to take explicit, accountable action in response to dropped handoffs, new handoffs, and changes to the contents of handoffs.
3. explicitly encourage the use of sources of information other than project documentation during test design.
4. allow the test effort to be degraded by poor or late project documentation, but prevent it from being blocked entirely.
5. allow individual tests to be designed using information combined from various sources.
6. allow tests to be redesigned as new sources of information appear.
7. include feedback loops so that test design takes into account what's learned by running tests.
8. allow testers to consider the possible savings of deferring test execution.
9. allow tests of a component to be executed before the component is fully assembled.