The Butterfly Model for Test Development

Stephen D. Morton
Applied Dynamics International
morton@adi.com
www.adi.com

There is a dichotomy between the development and testing of software. This schism is illustrated by the plethora of development models employed for planning and estimating the development of software as opposed to the scarcity of valid test development models. At first glance, the same models which serve to underlay the software development process with forethought and diligence appear to be adequate for the more complex task of planning, developing, and executing adequate verification of the application.

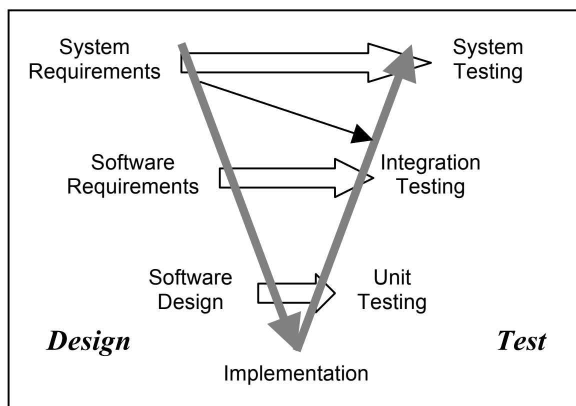
Unfortunately, software development models were not intended to encapsulate the vagaries of software verification and validation, the two main goals of software testing. Indeed, software development models can be antithetical to the effective testing of software. It lies in the hands of software testing professionals, therefore, to define an effective model for software test development that complements and completes any given software development model.

One such test development model is the Butterfly Model, which I will explore in some detail in this paper. It should be understood that the butterfly model is neither separate nor integrated with the development model, but instead is a monitoring and modifying factor in the completion of the development model. While this may seem arbitrary and self-contradictory, it is my hope that the elaboration of the butterfly model presented herein will both explain and justify this statement.

In this paper I will present a modified view of the ubiquitous "V" software development model. On top of this modified model I will superpose the butterfly model of test development. Finally, I will reconcile the relationship between the models, clarifying the effects of each on the other and identifying the information portals germane to both, together or separately.

The Standard V Software Development Model

Nearly everyone familiar with modern software development knows of the standard V development model, depicted below.



In this standardized image of the V development model, both the design and test phases of development are represented as linear processes that are gated according to the specific products of specific activities. On the design side, system requirements beget software requirements, which then beget a software design, which in turn begets an implementation.

On the test side of development, the software design begets unit tests. Similarly, software requirements beget integration tests (with a

© 2001 Applied Dynamics International

little help from the system requirements). Finally, system requirements beget system tests. Acceptance testing, being the domain of the end user of the application, is deliberately omitted from this view of the V model.

It should be understood that the V model is simply a more expressive rearrangement of the waterfall model, with the waterfall's time-line component mercifully eliminated and abstraction of the system indicated by the vertical distance from the implementation. The V model is correct as far as it goes, in that it expresses most of the lineage required for the artifacts of successful software development. From an application development point of view, this depiction of the model is sufficient to convey the source associations of the major development cycle artifacts, including test artifacts.

Unfortunately, the application development viewpoint falls well short of the software test development vantage required to create and maintain effective test artifacts.

Rigor of Model Enforcement

Before launching into a discussion of the shortfalls of the V software development model, a side excursion to examine the appropriate level of rigor in enforcing the model is warranted. It needs to be recognized from the start that not all applications will select to implement the V model in the same manner. Generally, deciding on how rigidly the model must be followed is largely a product of understanding the operational arena of the application.

For example, any certification requirements attached to the application will dictate the rigor of the model's implementation. Safety critical software in the commercial aerospace arena, for example, undergo an in-depth certification review prior to being released for industry use. Applications in this arena therefore tailor their implementation of the V model toward fulfillment of the objectives listed for each segment of the process in RTCA/DO-178B, the Federal Aviation Administration's (FAA's) selected guidelines for certification.

Similarly, medical devices containing software that affects safety must be developed using a version of the model that fulfills the certification requirements imposed by the Food and Drug Administration (FDA). As automotive embedded controller software continues to delve into applications that directly affect occupant safety (such as actuator based steering), it can be expected that some level of certification requirement will be instituted for that arena, as well.

Other arenas do not require anything approaching this level of rigor in their process. If the application cannot directly cause injury or the loss of life, or trigger the financial demise of a company, then it can most likely follow a streamlined version of the V model.

Web applications generally fall into this category, as do many e-commerce and home-computing applications. In fact, more applications fall into the second category than the first. That doesn't exempt them from the need to follow the model, however. It simply modifies the parameters of their implementation of the model.

Where the V Model Leaves Off

The main issue with the V development model is not its depiction of ancestral relationships between test artifacts and their design artifact progenitors. Instead, there are three facets of the V model's that are incomplete and must be accounted for. Just as in software development, we must define the problem before we can attempt to solve it.

First, the V model is inherently a linear expression of a nonlinear process. The very existence of the spiral model of software development should be sufficient evidence of the nonlinearity of software development, but this point deserves further examination.

© 2001 Applied Dynamics International

Software design artifacts, just like the software program they serve, are created and maintained by people. People make mistakes. The existence of software testers bears witness to this, as does the amount of buggy software that still seems to permeate the marketplace, despite the best efforts of software development and testing professionals. When mistakes are found in an artifact, the error must be corrected. The act of correction, in a small way, is another iteration of the original development of the artifact.

The second deficient aspect of the V model is its implication of unidirectional flow from design artifacts into test artifacts. Any seasoned software developer understands that feedback within the development cycle is an absolute necessity. The arrows depicting the derivation of tests from the design artifacts should in reality be two headed, although the left-pointing arrowhead would be significantly smaller than the right-pointing head.

While test artifacts are *generally* derived from their corresponding design artifacts, the fact that a test artifact must be so derived needs to be factored in when creating the design artifact in the first place. Functional requirements must be testable – they must be stated in such a manner as to be conducive to analysis, measurement, or demonstration. Vague statement of the requirements is a clear indicator of trouble down the road. Likewise, software designs need to be complete and unambiguous. The implementation methodology called for in the software design must be clear enough to drive the definition of appropriate test cases for the verification and validation of that design.

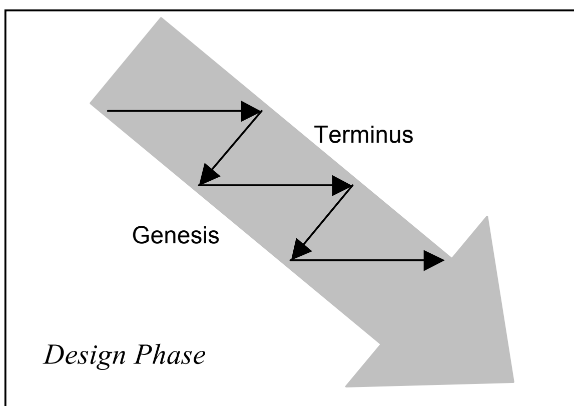
If the implementation itself is to be part of the test ancestry, then it, too, must be concise and complete, with adequate commentary on the techniques employed in its construction but without ambiguity or self-contradiction.

It should be noted that this discussion of the second deficient aspect of the V model is predicated on a rigorous enforcement of the model's dictates, such as is required for most aerospace applications. For less rigorous instances of the model, the absolutes listed above may not apply. This issue will be discussed further later in this paper.

The third deficient aspect of the V software development model is its encapsulation of test artifact ancestry solely within the domain of the design artifacts. As stated above, test artifacts are *generally* derived from their corresponding design artifacts. There are a multitude of other sources that must be touched upon to ensure success in generating a "complete" battery of tests for the software being developed.

A Closer View

The first issue mentioned with regard to the V development model is its essential linearization of a nonlinear process – software development. This problem is one of perception, really, or perhaps perspective. The root cause can be found in the fact that the V software development model is a simplified visualization tool that illustrates a complex and interrelated process. A more detailed view of a segment of the design leg (*which* segment is immaterial) is shown below.



In this expanded view of the design leg of the V, the micro-iterative feedback depicted by the small black arrows within the overall gray feed-forward thrust are visible. Each micro-iteration represents the accumulation of further

data, application of a lesson learned, or even the bright idea someone dreamed up while singing in the shower. The point to be made here is this: The general forward-leaning nature of the legs of the V tends to disguise the frenzied iterations in thought, specification, and development required to create a useful application.

There are critical points along the software development stream that must be accounted for in any valid test development model. For example, any time there is a handoff of an artifact (or part of an artifact), the transacted artifact must be analyzed with respect to its contents and any flow-down effects caused by those contents [MARI99]. In the expanded view of the V development model shown above, the left edge of the broad arrow represents the genesis of a change in the artifact under development. This edge, where new or modified information is being introduced, is the starting point for all new micro-iterations. The right edge of the broad arrow is the terminus for each micro-iteration, where the new or modified information is fully incorporated in the artifact.

It should be further understood that micro-iterations can be independent of each other. In fact, most significant software development incorporates a maelstrom of independent micro-iterations that ebb and flow both concurrently and continuously throughout the overall development cycle.

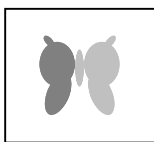
The spiral model of software development, which many consider to be superior to the V model, is founded on an explicit understanding of the iterative nature of software creation. Unfortunately, the spiral model tends to be expressed on a macro scale, hiding the developmental perturbations needed for the production of useful design and test artifacts.

The Butterfly Model

Now that we have rediscovered the hidden micro-iterations in a successful process based on the V model, we need to understand the source of these perturbations. Further, we need to understand the fundamental interconnectedness of it all, to borrow an existential phrase.

Butterflies are composed of three pieces – two wings and a body. Each part represents a piece of software testing, as described hereafter.

Test Analysis



The left wing of the butterfly represents test analysis – the investigation, quantization, and/or re-expression of a facet of the software to be tested. Analysis is both the byproduct and foundation of successful test design. In its earliest form, analysis represents the thorough pre-examination of design and test artifacts to ensure the existence of adequate testability, including checking for ambiguities, inconsistencies, and omissions.

Test analysis must be distinguished from software design analysis. Software design analysis is constituted by efforts to define the problem to be solved, break it down into manageable and cohesive chunks, create software that fulfills the needs of each chunk, and finally integrate the various software components into an overall program that solves the original problem. Test analysis, on the other hand, is concerned with validating the outputs of each software development stage or micro-iteration, as well as verifying compliance of those outputs to the (separately validated) products of previous stages.

Test analysis mechanisms vary according to the design artifact being examined. For an aerospace software requirement specification, the test engineer would do all of the following, as a minimum:

© 2001 Applied Dynamics International

- Verify that each requirement is tagged in a manner that allows correlation of the tests for that requirement to the requirement itself. (Establish Test Traceability)
- Verify traceability of the software requirements to system requirements.
- Inspect for contradictory requirements.
- Inspect for ambiguous requirements.
- Inspect for missing requirements.
- Check to make sure that each requirement, as well as the specification as a whole, is understandable.
- Identify one or more measurement, demonstration, or analysis method that may be used to verify the requirement's implementation (during formal testing).
- Create a test "sketch" that includes the tentative approach and indicates the test's objectives.

Out of the items listed above, only the last two are specifically aimed at the act of creating test cases. The other items are almost mechanical in nature, where the test design engineer is simply checking the software engineer's work. But all of the items are germane to test analysis, where any error can manifest itself as a bug in the implemented application.

Test analysis also serves a valid and valuable purpose within the context of software development. By digesting and restating the contents of a design artifact (whether it be requirements or design), testing analysis offers a second look – from another viewpoint – at the developer's work. This is particularly true with regard to lower-level design artifacts like detailed design and source code.

This kind of feedback has a counterpart in human conversation. To verify one's understanding of another person's statements, it is useful to rephrase the statement in question using the phrase "So, what you're saying is…". This powerful method of confirming comprehension and eliminating miscommunication is just as important for software development – it helps to weed out misconceptions on the part of both the developer and tester, and in the process identifies potential problems in the software itself.

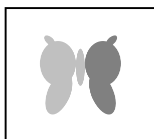
It should be clear from the above discussion that the tester's analysis is both formal and informal. Formal analysis becomes the basis for documentary artifacts of the test side of the V. Informal analysis is used for immediate feedback to the designer in order to both verify that the artifact captures the intent of the designer and give the tester a starting point for understanding the software to be tested.

In the bulleted list shown above, the first two analyses are formal in nature (for an aerospace application). The verification of system requirement tags is a necessary step in the creation of a test traceability matrix. The software to system requirements traceability matrix similarly depends on the second analysis.

The three inspection analyses listed are more informal, aimed at ensuring that the specification being examined is of sufficient quality to drive the development of a quality implementation. The difference is in how the analytical outputs are used, not in the level of effort or attention that go into the analysis.

Test Design

Thus far, the tester has produced a lot of analytical output, some semi-formalized documentary artifacts, and several tentative approaches to testing the software. At this point, the tester is ready for the next step: test design.



The right wing of the butterfly represents the act of designing and implementing the test cases needed to verify the design artifact as replicated in the

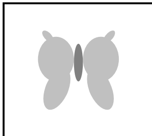
implementation. Like test analysis, it is a relatively large piece of work. Unlike test analysis, however, the focus of test design is not to assimilate information created by others, but rather to implement procedures, techniques, and data sets that achieve the test's objective(s).

The outputs of the test analysis phase are the foundation for test design. Each requirement or design construct has had at least one technique (a measurement, demonstration, or analysis) identified during test analysis that will validate or verify that requirement. The tester must now put on his or her development hat and implement the intended technique.

Software test design, as a discipline, is an exercise in the prevention, detection, and elimination of bugs in software. Preventing bugs is the primary goal of software testing [BEIZ90]. Diligent and competent test *design* prevents bugs from ever reaching the implementation stage. Test design, with its attendant test analysis foundation, is therefore the premiere weapon in the arsenal of developers and testers for limiting the cost associated with finding and fixing bugs.

Before moving further ahead, it is necessary to comment on the continued analytical work performed during test design. As previously noted, tentative approaches are mapped out in the test analysis phase. During the test design phase of test development, those tentatively selected techniques and approaches must be evaluated more fully, until it is "proven" that the test's objectives are met by the selected technique. If all tentatively selected approaches fail to satisfy the test's objectives, then the tester must put his test analysis hat back on and start looking for more alternatives.

Test Execution



In the butterfly model of software test development, test execution is a separate piece of the overall approach. In fact, it is the smallest piece – the slender insect's body – but it also provides the muscle that makes the wings work. It is important to note, however, that test execution (as defined for this model) includes *only* the formal running of the designed tests. Informal test execution is a normal part of test design, and in fact is also a normal part of software design and development.

Formal test execution marks the moment in the software development process where the developer and the tester join forces. In a way, formal execution is the moment when the developer gets to take credit for the tester's work – by demonstrating that the software works as advertised. The tester, on the other hand, should already have proactively identified bugs (in both the software and the tests) and helped to eliminate them – well before the commencement of formal test execution!

Formal test execution should (almost) never reveal bugs. I hope this plain statement raises some eyebrows – although it is very much true. The only reasonable cause of *unexpected* failure in a formal test execution is hardware failure. The software, along with the test itself, should have been through the wringer enough to be bone-dry.

Note, however, that *unexpected* failure is singled out in the above paragraph. That implies that some software tests will have *expected* failures, doesn't it? Yes, it surely does!

The reasons behind expected failure vary, but allow me to relate a case in point:

In the commercial jet engine control business, systems engineers prepare a wide variety of tests against the system (being the FADEC – or Full Authority Digital Engine Control) requirements. One such commonly employed test is the "flight envelope" test. The flight envelope test essentially begins with the simulated engine either off or at idle with the real controller (both hardware and software) commanding the situation. Then the engine is spooled up and taken for a simulated ride throughout its defined operational domain – varying altitude, speed, thrust,

© 2001 Applied Dynamics International

temperature, etc. in accordance with real world recorded profiles. The expected results for this test are produced by running a simulation (created and maintained independently from the application software itself) with the same input data sets.

Minor failures in the formal execution of this test are fairly common. Some are hard failures – repeatable on every single run of the test. Others are soft – only intermittently reaching out to bite the tester. Each and every failure is investigated, naturally – and the vast majority of flight envelope failures are caused by test stand problems. These can include issues like a voltage source being one twentieth of a volt low, or slight timing mismatches caused by the less exact timekeeping of the test stand workstation as compared to the FADEC itself.

Some flight envelope failures are attributed to the model used to provide expected results. In such cases, hours and days of gut-wrenching analytical work go into identifying the miniscule difference between the model and the actual software.

A handful of flight envelope test failures are caused by the test parameters themselves. Tolerances may be set at unrealistically tight levels, for example. Or slight operating mode mismatches between the air speed and engine fan speed may cause a fault to be intermittently annunciated.

In very few cases have I seen the software being tested lay at the root of the failure. (I did witness the bugs being fixed, by the way!)

The point is this – complex and complicated tests can fail due to a variety of reasons, from hardware failure, through test stand problems, to application error. Intermittent failures may even jump into the formal run, just to make life interesting.

But the test engineer understands the complexity of the test being run, and anticipates potential issues that may cause failures. In fact, the test is *expected* to fail once in a while. If it doesn't, then it isn't doing its job – which is to exercise the control software throughout its valid operational envelope. As in all applications, the FADEC's boundaries of valid operation are dark corners in which bugs (or at least potential bugs) congregate.

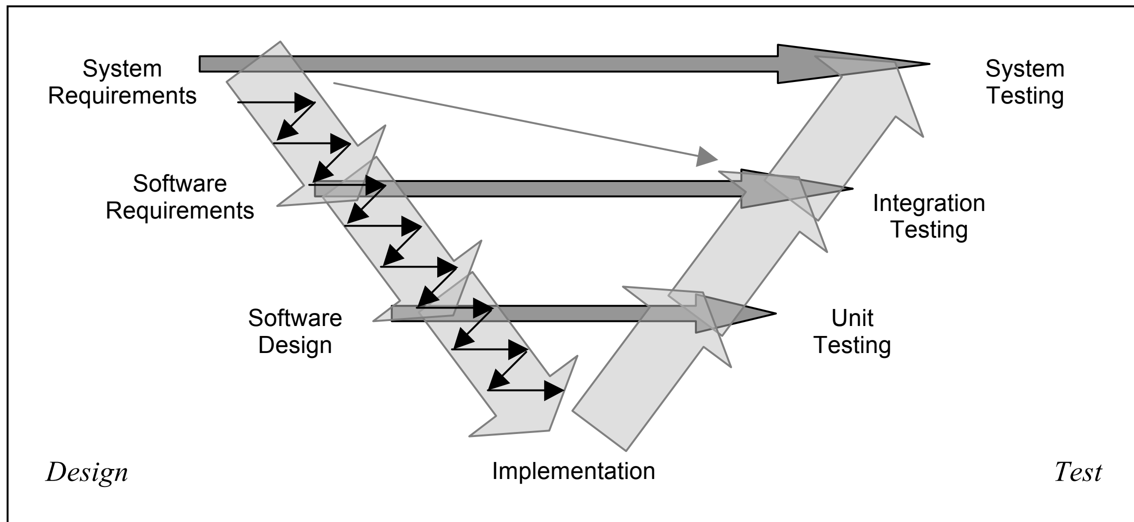
It was mentioned during our initial discussion of the V development model that the model is sufficient, from a software development point of view, to express the lineage of test artifacts. This is because testing, again from the development viewpoint, is composed of only the body of the butterfly – formal test execution. We testers, having learned the hard way, know better.

A Swarm of Testing

We have now examined how test analysis, test design, and test execution compose the body of the butterflies in this test development model. In order to understand how the butterfly model monitors and modifies the software development model, we need to digress slightly and reexamine the V software development model itself.

In Figure 1, not only have the micro-iterations naturally present in the design cycle been included, but the major design phase segments (characterized by their outputs) have been separated into smaller arrows to clearly define the transition point from one segment to the next. The test side of the V has been similarly separated, to demarcate the boundaries between successful *formal execution* of each level of testing.

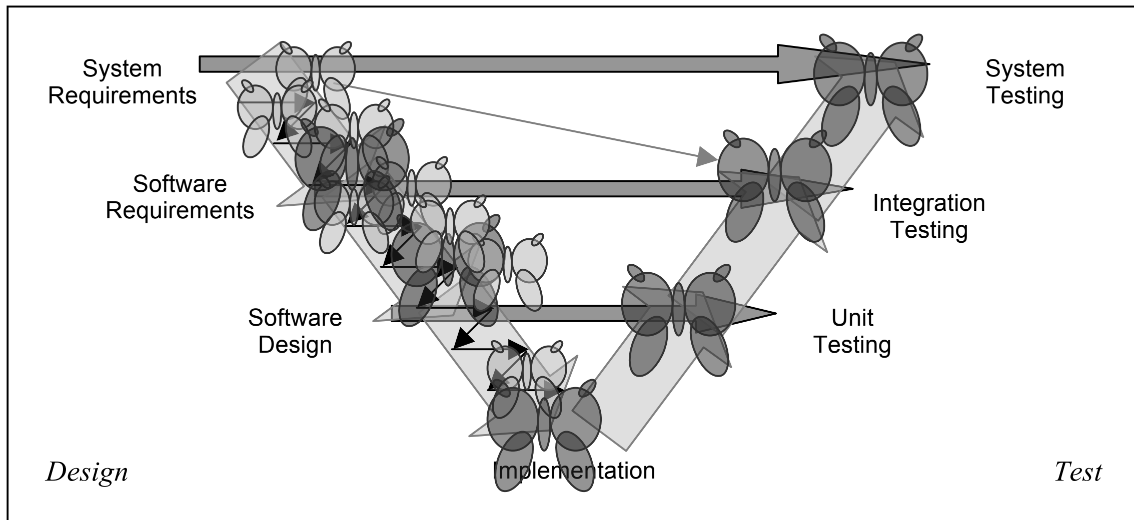
Figure 1. Complete Expanded V Development Model View



No micro-iterations on the test side of the V are shown in this depiction, although there are a few to be found – mostly around the phase segment transitions, where test execution documentary artifacts are formulated and preserved. The relative lack of micro-iterations on the test side of the V is due to the fact that it represents only the formal running of tests – the leg work of analysis and design are done elsewhere. The question, therefore, is: Where?

The answer to this all-important question is shown in Figure 2.

Figure 2. Illustration of the Butterfly Test Development Model



At all micro-iteration termini, and at some micro-iteration geneses, exists a small test butterfly. These tiny test insects each contribute to the overall testing effort, encapsulating the test analysis and design required by whatever minor change is represented by the micro-iteration.

Larger, heavier butterflies spring to life on the boundaries between design phase segments. These larger specimens carry with them the more formal analyses required to transition from one segment to the next. They also answer the call for coordination between the tests designed as part of their smaller brethren. Large butterflies also appear at the transition points between test phase segments, where documentary artifacts of test execution are created in order to claim credit for the formal execution of the test.

A single butterfly, by itself, is of no moment – it cannot possibly have much impact on the overall quality of the application and its tests. But a swarm of butterflies can blot out the sun, affecting great improvement in the product's quality. The smallest insects handle the smallest changes, while the largest tie together the tests and analyses of them all.

The right-pointing lineage arrows, which show the roots of each test artifact in its corresponding design artifact, point to the moment in the software development model where the analysis and design of tests culminate in their formal execution.

Butterfly Thinking

"A butterfly flutters its wings in Asia, and the weather changes in Europe." This colloquialism offers insight into the chaotic (in the mathematical sense of the word) nature of software development. Events that appear minor and far removed from relevance can have a profound impact on the software being created. Many seemingly minor and irrelevant events are just that – minor and irrelevant. But some such events, despite their appearance, are not.

Identifying these deceptions is a key outcome of the successful implementation of the butterfly model. The following paragraphs contain illustrations of this concept.

Left Wing Thinking

The FADEC must assert control over the engine's operation within 300 msec of a power-on event.

This requirement, or a variant of it, appears in every system specification for a FADEC. It is important because it specifies the amount of time available for a cold-start initialization in the software.

The time allotted is explicit. No more than three tenths of a second may elapse before the FADEC asserts itself.

The commencement of that time period is well defined. The nearly vertical rising edge of the FADEC power signal as it moves from zero volts (off) to the operational voltage of the hardware marks the start line.

But what the heck does "assert control" mean?

While analyzing this requirement statement, that question should jump right off the written page at the tester. In one particular instance, the FADEC asserted control by crossing a threshold voltage on a specific analog signal coming out of the box. Unfortunately, that wasn't in the specification. Instead, I had to ask the senior systems engineer, who had performed similar tests hundreds of times, how to tell when the FADEC asserted itself.

In other words, I couldn't create a test sketch for the requirement because I couldn't determine what the end point of the measurement should be. The system specification assumed that the reader held this knowledge, although anyone who was learning the ropes (as I was at that point) had no reasonable chance of knowing. As far as I know, this requirement has never been elaborated.

As a counterpoint example, consider the mass-market application that, according to the verbally preserved requirements, had to be "compelling". What the heck is "compelling", and how does one test for it?

In this case, it didn't matter that the requirement was ill suited for testing. In fact, the testers' opinions on the subject weren't even asked for. But the application succeeded, as evidenced by

© 2001 Applied Dynamics International

the number of copies purchased. Customers found the product compelling, and therefore the project was a success.

But doesn't this violate the "must be testable" rule for requirements? Not really. The need to be "compelling" doesn't constitute a *functional* requirement, but is instead an *aesthetic* requirement. Part of the tester's analysis should weed out such differences, where they exist.

Right Wing Thinking

Returning to our power-up timing example, how can we measure the time between two voltage-based events? There are many possibilities, although most can't handle the precision necessary for a 300 msec window. Clocks, watches, and even stopwatches would be hideously unreliable for such a measurement.

The test stand workstation also couldn't be used. That would require synchronization of the command to apply power with the actual application of power. There was a lag in the actual application of power, caused by the software-driven switch that had to be toggled in the test stand's circuitry. Worse yet, detection of the output voltage required the use of a digital voltmeter, which injected an even larger amount of uncertainty into the measurement.

But a digital oscilloscope attached to a printer would work, provided that the scope was fast enough. The oscilloscope was the measurement device (obviously). The printer was required to "prove" that the test passed. This was, after all, an application subject to FAA certification.

As a non-certification counter example, consider the product whose requirements included the following statement:

Remove unneeded code where possible and prudent.

In other words, "Make the dang thing smaller". The idea behind the requirement was to shrink the size of the executable, although eliminating unnecessary code is usually a good thing in its own right. No amount of pleading was able to change this requirement into a quantifiable statement, either.

So how the heck can we test for this? In this case, the tester might rephrase the requirement in his or her mind to read:

The downloadable installer must be smaller than version X.

This provides a measurable goal, albeit an assumed one. More importantly, it preserves the common thread between the two statements, which is that the product needed to shrink in size.

Body Thinking

To be honest, there isn't all that much thought involved in formally executing thoroughly prepared test cases. The main aspect of formal execution is the collection of "evidence" to prove that the tests were run and that they passed. There is, however, the need to analyze the recorded evidence as it is amassed.

For example, aerospace applications commonly must be unit tested. Each individual function or procedure must be exercised according to certain rules. The generally large number of modules involved in a certification means that the unit testing effort required is big, although each unit test itself tends to be small. Naturally, the project's management normally tries to get the unit testing underway as soon as possible to ensure completion by the "drop-dead" date for unit test completion implied in the V model.

© 2001 Applied Dynamics International

As the established date nears, the test manager must account for every modified unit. The last modification of the unit must predate the configured test procedures and results for that unit. All of the tests must have been peer reviewed prior to formal execution. And all of the tests must have passed during formal execution.

In other words, “dot the I’s and cross the T’s”. It is largely an exercise in bookkeeping, but that doesn’t diminish its importance.

The Swarm Mentality

To better illustrate the swarm mentality, let’s look at an unmanned rocket project that utilized the myriad butterflies of this model to overwhelm bugs that could have caused catastrophic failure. This rocket was really a new version of an existing rocket that had successfully blasted off many, many times.

First, because the new version was to be created as a change to the older version’s software, a complete and thorough system specification analysis was performed, comparing the system specs for both versions. This analysis found that:

- The old version contained a feature that didn’t apply to the new version. A special extended calculation of the horizontal bias (B_H) that allowed for late-countdown (between five and ten seconds before launch) holds to be restarted within a few minutes didn’t apply to the new version of the rocket. B_H was known to be meaningless after either version left the launch pad, but was calculated in the older version for up to 40 seconds after liftoff.
- The updated flight profile for the new version had not been included in the updated specification, although this omission had been agreed to by all relevant parties. That meant that discrepancies between the early trajectory profiles between the two versions were not available for examination. The contractors building the rocket didn’t want to change their agreement on this subject, so the missing trajectory profile information was marked as a risk to be targeted with extra-detailed testing.

Because of the fairly serious questions raised in the system requirements analysis, the test engineers decided to really attack the early trajectory operation of the new version. Because this was an aerospace application, they knew that the subsystems had to be qualified for flight prior to integration into the overall system. That meant that the inertial reference system (SRI) that provided the raw data required to calculate B_H would work, at least as far as it was intended to.

But how could they test the interaction of the SRI and the calculation of B_H? The horizontal bias was also a product of the rocket’s acceleration, so they knew that they would have to at least simulate the accelerometer inputs to the control computer (it is physically impossible to make a vibration table approach the proper values for the rocket’s acceleration).

If they had a sufficiently detailed SRI model, they could also simulate the inertial reference system. Without a detailed simulation, they’d have to use a three-axis dynamic vibration table. Because the cost of using the table for an extended period of time was higher than the cost of creating a detailed simulation, they decided to go with the all simulation approach.

In the meantime, a detailed analysis of the software requirements for both versions revealed a previously unknown conceptual error. Every exception raised in the Ada software automatically shut down the processor – whether the exception was caused by a hardware or software fault!

The thinking behind this problem was that exceptions should only address random hardware failures, where the software couldn’t hope to recover. Clearly, software exceptions were possible, even if they were improbable. So, the exception handling in the software spec was updated to differentiate between hardware and software based exceptions.

Examining the design of the software, the test engineers were amazed to discover that the horizontal bias calculations weren't protected for Operand Error, which is automatically raised in Ada when a floating point real to integer conversion exceeds the available range of the integer container. B_H was involved just such a conversion!

The justification for omitting this protection was simple, at least for the older version of the rocket. The possible values of B_H were physically limited in range so that the conversion couldn't ever overflow. But the newer version couldn't claim that fact, so the protection for Operand Error was put into the new version's design. Despite the fact that this could put the 80% usage goal for the SRI computer at risk, the possibility that the computer could fail was simply too great.

Finally, after much gnashing of teeth, the test engineers convinced the powers that be to completely eliminate the prolonged calculation of horizontal bias because it was useless in the new version. The combined risks of the unknown trajectory data, the unprotected conversion to integer, and the money needed to fund the accurate SRI simulation was too much for the system's developers. They at last agreed that it was better to eliminate the unnecessary processing, even though it worked for the previous version.

As a result, the maiden demonstration flight for the Ariane 5 rocket went off without a hitch.

That's right – I have been describing the findings of the inquiry board for the Ariane 5 in light of how a full and rigorous implementation of the butterfly model would have detected, mitigated, or eliminated them [LION96].

Ariane 4 contained an extended operation alignment function that allowed for late-countdown holds to be handled without long delays. In fact, the 33rd flight of the Ariane 4 rocket used this feature in 1989.

The Ariane 5 trajectory profile was never added to the system requirements. Instead, the lower values in the Ariane 4 trajectory data were allowed to stand.

The SRI computers (with the deficient software) were therefore never tested to the updated trajectory telemetry.

The missing Operand Error exception handling for the horizontal bias therefore never occurred during testing, causing the SRI computer to shut down.

The flawed concept of all exceptions being caused by random hardware faults was therefore never exposed.

SRI 1, the first of the dual redundant components, therefore halted on an Operand Error caused by the conversion of B_H in the 39th second after liftoff. SRI 2 immediately took over as the active inertial reference system.

But then SRI 2 failed because of the same Operand Error in the following data cycle (72 msec in duration).

And therefore, Ariane 5 self destructed in the 42nd second of its maiden voyage – all for lack of a swarm of butterflies.

The Butterfly Model within the V Model Context

The butterfly model of test development is not a component of the V software development model. Instead, the butterfly test development model is a superstructure imposed atop the V model that operates semi-independently, in parallel with the development of software.

The main relationship between the V model and the butterfly swarm of testing activity is timing, at least on the design side of the V. Test development is driven by software development, for software is what we are testing. Therefore, the macro and micro iterations of software development define the points at which test development activity is both warranted and required. The individual butterflies must react to the iterative software development activity that spawned them, while the whole of the swarm helps to shape the large and small perturbations in the software design stream.

On the test side of the V, the relationship is largely reversed – the software milestones of the V model are the results of butterfly activity on the design side. The differences between the models give latitude to both the developer and the tester to envision the act of testing within their particular operational context. The developer is free to see testing as the culmination of their development activity. The tester is likewise free to see the formal execution of testing as the end of the line – where all of the analytical and test design effort that shepherded the software design process is transformed into the test artifacts required for progression from development to delivery.

But the butterfly model does not entirely fall within the bounds of the V model, either. The third issue taken with the standardized V model stated that the roots of software testing lay mainly within the boundaries of the software to be tested. But proper performance of test analysis and design require knowledge outside the realm of the application itself.

Testers in the butterfly model require knowledge of testing techniques, tools, methodologies, and technologies. Books and articles about test theory are hugely important to the successful implementation of the butterfly model. Similarly, software testing conferences and proceedings are valuable resources.

Testers in this test development model also need to keep abreast of technological advancements related to the application being developed. Trade journals and periodicals are valuable sources for such information.

In the end, the tester is required to not only know the application being tested, but also to understand (at some level) software testing, valid testing techniques, software testing tools and technologies, and even a little about human nature.

Next Steps

The butterfly model of test development is far from complete. The model as described herein is a first step toward a complete and usable model. Some of the remaining steps to finish it include:

- Creating a taxonomy of test butterflies that describes each type of testing activity within the context of the software development activity it accompanies.
- Correlating the butterfly taxonomy with a valid taxonomy of software bugs (to understand what the butterflies eat).
- Formally defining and elaborating the “objectives” associated with various testing activities.
- Creating a taxonomy of “artifacts” to better define the parameters of the model’s execution.
- Expanding visualization of the model to cover the spiral development model.
- Defining the framework necessary to achieve full implementation of the model.
- Identifying methods of automating significant portions of the model’s implementation.

Summary

The butterfly model for software test development is a semi-dependent model that represents the bifurcated role of software testing with respect to software development. The underlying

© 2001 Applied Dynamics International

realization that software development and test development are parallel processes that are separate but complementary is embodied by the butterfly model's superposition atop the V development model.

Correlating the V model and butterfly model requires understanding that the standard V model is a high-level view of software development that hides the myriad micro-iterations all along the design and test legs of the V. These micro-iterations are the core of successful software development. They represent the incorporation of new knowledge, new requirements, and lessons learned – primarily during the design phase of software development, although the formation of test artifacts also includes some micro-iterative activity.

Tiny test butterflies occupy the termini of these micro-iterations, as well as some of their geneses. Larger, more comprehensive butterflies occupy phase segment transition points, where the nature of work is altered to reach toward the next goal of the software's development.

The parts of the butterfly represent the three legs of successful software testing – test analysis, test design, and formal test execution. Of the three, formal execution is the smallest, although it is the only piece explicitly represented in the V model. Test analysis and test design, ignored in the V model, are recognized in the butterfly model as shaping forces for software development, as well as being the foundation for test execution.

Finally, the butterfly model is in its infancy, and there is significant work to do before it can be fully described. However, the visualization of a swarm of testing butterflies darkening the sky while they steer software away from error injection is satisfying– at last we have a physical phenomena that represents the ephemeral act of software testing.

References

BEIZ90	Boris Beizer, *Software Testing Techniques (2/e)*, International Thomson Computer Press, 1990.
MARI99	Brian Marick, "New Models for Test Development", Reliable Software Technologies, 1999, Available online at http://www.testing.com/writings/new-models.pdf.
LION96	Professor J.L. Lions, Chairman of the Board, "ARIANE 5 Flight 501 Failure. Report by the Inquiry Board", 1996, Available online at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html.

Acknowledgements

Special thanks are due Elisabeth Hendrickson, who helped me see more of the world beyond the aerospace boundaries I "grew up" in. Her review and suggestions have been immeasurably helpful in framing this theory in its proper context.

© 2001 Applied Dynamics International