Testing Models

There are various models which have been presented in the past 20 years in the field of Software Engineering for Development and Testing. Let us discuss and explore into few of the famous models.

The following models are addressed:
Waterfall Model.
Spiral Model.
'V' Model.
'W' Model, and
Butterfly Model.

The Waterfall Model
This is one of the first models of software development, presented by B.W. Boehm. The Waterfall model is a step-by-step method of achieving tasks. Using this model, one can get on to the next phase of development activity only after completing the current phase. Also one can go back only to the immediate previous phase.

In Waterfall Model each phase of the development activity is followed by the Verification and Validation activities. One phase is completed with the testing activities, then the team proceeds to the next phase. At any point of time, we can move only one step to the immediate previous phase. For example, one cannot move from the Testing phase to the Design phase.

Spiral Model
In the Spiral Model, a cyclical and prototyping view of software development is shown. Test are explicitly mentioned (risk analysis, validation of requirements and of the development) and the test phase is divided into stages. The test activities include module, integration and acceptance tests. However, in this model the testing also follows the coding. The exception to this is that the test plan should be constructed after the design of the system. The spiral model also identifies no activities associated with the removal of defects.

'V' Model
Many of the process models currently used can be more generally connected by the 'V' model where the 'V' describes the graphical arrangement of the individual phases. The 'V' is also a synonym for Verification and Validation.

By the ordering of activities in time sequence and with abstraction levels the connection between development and test activities becomes clear. Oppositely laying activities complement one another (i.e.) server as a base for test activities. For example, the system test is carried out on the basis of the results specification phase.
The 'W' Model
From the testing point of view, all of the models are deficient in various ways: The Test activities first start after the implementation. The connection between the various test stages and the basis for the test is not clear.

The tight link between test, debug and change tasks during the test phase is not clear.

Why 'W' Model?
In the models presented above, there usually appears an unattractive task to be carried out after coding. In order to place testing on an equal footing, a second 'V' dedicated to testing is integrated into the model. Both 'V's put together give the 'W' of the 'W-Model'.

Butterfly Model of Test Development
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:
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,
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, 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 annunciating.

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.