Formal Methods of Software Testing and Terminology

Sunil Kumar Scholar-Ph.D. Computer Science ,CMJ University ,Shillong (Meghalaya)
Guide Dr. P.K Yadav, Associate Professor, Govt. College, Kanwali (Rewari)

ABSTRACT
Software provides a complete set of application development tools for building stand-alone, client-server, and Internet-enabled applications. But making it easy to build applications can be a two-edged sword. Not only can developers build powerful, sophisticated applications, but they can also build applications that frustrate users, waste computer resources, and damage the credibility of both the developer. Formal testing will help prevent bad applications from being released. For those unfamiliar with the topic, this paper can serve as a primer or first step in learning about a more formal, rigorous approach to software testing.

KEY WORD: SDLC, SAP, ASD.

INTRODUCTION
I believe that the finger is better pointed at those responsible for training the entry-level developer and, even more so, at those responsible for overseeing the testing and release of their work. In many cases, however, the problem may be a systemic or scheduling failure—overly aggressive schedules to document requirement, design, build, test, and release software may leave too little time for adequate testing and force developers to release code that’s not ready. Assuming that a project has fully collected and clearly documented its business and technical requirements (which might be a stretch, but let’s pretend), a primary cause of failed application software development is lack of a formal requirements-based testing process. “Formal requirements-based testing” may conjure up an image of a lengthy, involved, and minutely detailed process but it doesn’t necessarily have to be like that, although in organizations with mature software engineering practices. In many if not most organizations, formal software testing can easily be tailored to the application being examined and has only two real prerequisites.

- Business requirements and design documents that allow development of a test plan.
- People who understanding how to write and carry out a test plan.

“It is not enough to do your best. You must know what to do and then do your best. W. Edwards Deming”

In this paper, the terms application, program, and system are used rather interchangeably to describe ‘applications software’, which is “a program or group of programs designed for end users to accomplish some task”.

SOFTWARE TESTING—WHAT, WHY, AND WHO

WHAT IS SOFTWARE TESTING?
Software testing is a process of verifying and validating that a software application or program
1. Meets the business and technical requirements that guided its design and development
2. Works as expected.
Software testing also identifies important defects, flaws, or errors in the application code that must be fixed. The modifier “important” in the previous sentence is, well, important because defects must be categorized by severity (more on this later).

During test planning we decide what an important defect is by reviewing the requirements and design documents with an eye towards answering the question “Important to whom?” Generally speaking, an important defect is one that from the customer’s perspective affects the usability or functionality of the application.

Software testing has three main purposes: verification, validation, and defect finding.

1. The verification process confirms that the software meets its technical specifications. A “specification” is a description of a function in terms of a measurable output value given a specific
input value under specific preconditions. A simple specification may be along the line of “a SQL query retrieving data for a single account against the multi-month account-summary table must return these eight fields ordered by month within 3 seconds of submission.”

2. The validation process confirms that the software meets the business requirements.

3. A defect is a variance between the expected and actual result. The defect’s ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

WHY DO SOFTWARE TESTING?

“A clever person solves a problem. A wise person avoids it. Albert Einstein” Why test software? “To find the bugs!” is the instinctive response and many people, developers and programmers included, think that that’s what debugging during development and code reviews is for, so formal testing is redundant at best. But a “bug” is really a problem in the code; software testing is focused on finding defects in the final product.

Software testing answers questions that development testing and code reviews can’t.

- Does it really work as expected?
- Does it meet the users’ requirements?
- Is it what the users expect?
- Do the users like it?
- Is it compatible with our other systems?
- How does it perform?
- How does it scale when more users are added?
- Which areas need more work?
- Is it ready for release?

What can we do with the answers to these questions?

- Know that we’ve satisfied our users’ requirements
- Build a list of desired modifications and enhancements for later versions
- Identify and catalog reusable modules and components
- Identify areas where programmers and developers need training

WHAT DO WE TEST?

First, test what’s important. Focus on the core functionality—the parts that are critical or popular—before looking at the ‘nice to have’ features. Concentrate on the application’s capabilities in common usage situations before going on to unlikely situations. The value of software testing is that it goes far beyond testing the underlying code. It also examines the functional behavior of the application. Behavior is a function of the code, but it doesn’t always follow that if the behavior is “bad” then the code is bad. It’s entirely possible that the code is solid but the requirements were inaccurately or incompletely collected and communicated. It’s entirely possible that the application can be doing exactly what we’re telling it to do but we’re not telling it to do the right thing.

A comprehensive testing regime examines all components associated with the application. Even more, testing provides an opportunity to validate and verify things like the assumptions that went into the requirements, the appropriateness of the systems that the application is to run on, and the manuals and documentation that accompany the application. More likely though, unless your organization does true “software engineering” the focus will be on the functionality and reliability of application itself.

Testing can involve some or all of the following factors. The more, the better.

- Business requirements
- Functional design requirements
- Technical design requirements
- Regulatory requirements
- Programmer code
- Systems administration standards and restrictions
- Corporate standards
WHO DOES THE TESTING?
Software testing is not a one person job. It takes a team, but the team may be larger or smaller depending on the size and complexity of the application being tested. The programmer(s) who wrote the application should have a reduced role in the testing if possible. The concern here is that they’re already so intimately involved with the product and “know” that it works that they may not be able to take an unbiased look at the results of their labors.

Testers must be cautious, curious, critical but non-judgmental, and good communicators. One part of their job is to ask questions that the developers might find not be able to ask themselves or are awkward, irritating, insulting or even threatening to the developers.

- How well does it work?
- What does it mean to you that “it works”?
- How do you know it works? What evidence do you have?
- In what ways could it seem to work but still have something wrong?
- In what ways could it seem to not work but really be working?
- What might cause it to not to work well?

A good developer does not necessarily make a good tester and vice versa, but testers and developers do share at least one major trait—they itch to get their hands on the keyboard. As laudable as this may be, being in a hurry to start can cause important design work to be glossed over and so special, subtle situations might be missed that would otherwise be identified in planning. Like code reviews, test design reviews are a good sanity check and well worth the time and effort.

Testers are the only IT people who will use the system as heavily an expert user on the business side. User testing almost invariably recruits too many novice business users because they’re available and the application must be usable by them. The problem is that novices don’t have the business experience that the expert users have and might not recognize that something is wrong. Testers from IT must find the defects that only the expert users will find because the experts may not report problems if they’ve learned that it’s not worth their time or trouble.

THE V-MODEL OF SOFTWARE TESTING
Software testing is too important to leave to the end of the project, and the V-Model of testing incorporates testing into the entire software development life cycle. In a diagram of the V-Model, the V proceeds down and then up, from left to right depicting the basic sequence of development and testing activities. The model highlights the existence of different levels of testing and depicts the way each relates to a different development phase.

Like any model, the V-Model has detractors and arguably has deficiencies and alternatives but it clearly illustrates that testing can and should start at the very beginning of the project. (See Goldsmith for a summary of the pros and cons and an alternative. Marrik’s articles provide criticism and an alternative.) In the requirements gathering stage the business requirements can verify and validate the business case used to justify the project. The business requirements are also used to guide the user acceptance testing. The model illustrates how each subsequent phase should verify and validate work done in the previous phase, and how work done during development is used to guide the individual testing phases. This interconnectedness lets us identify important errors, omissions, and other problems before they can do serious harm.

Application testing begins with Unit Testing, and in the section titled “Types of Tests” we will discuss each of these test phases in more detail.
The V-Model of Software Testing

Validation and Verification → Work Flow

Two arrows indicate Testing Relationship.
THE TEST PLAN
The test plan is a mandatory document. You can’t test without one. For simple, straight-forward projects the plan doesn’t have to be elaborate but it must address certain items. As identified by the “American National Standards Institute and Institute for Electrical and Electronic Engineers Standard 829/1983 for Software Test Documentation”, the following components should be covered in a software test plan.

Items Covered by a Test Plan

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
<td>Specific people who are and their assignments</td>
<td>Assigns responsibilities and keeps everyone on track and focused</td>
</tr>
<tr>
<td>Assumptions</td>
<td>Code and systems status and availability</td>
<td>Avoids misunderstandings about schedules</td>
</tr>
<tr>
<td>Test</td>
<td>Testing scope, schedule, duration, and prioritization</td>
<td>Outlines the entire process and maps specific tests</td>
</tr>
<tr>
<td>Communication</td>
<td>Communications plan—who, what, when, how</td>
<td>Everyone knows what they need to know when they need to know it</td>
</tr>
<tr>
<td>Risk Analysis</td>
<td>Critical items that will be tested</td>
<td>Provides focus by identifying areas that are critical for success</td>
</tr>
<tr>
<td>Defect Reporting</td>
<td>How defects will be logged and documented</td>
<td>Tells how to document a defect so that it can be reproduced, fixed, and retested</td>
</tr>
<tr>
<td>Environment</td>
<td>The technical environment, data, work area, and interfaces used in testing</td>
<td>Reduces or eliminates misunderstandings and sources of potential delay</td>
</tr>
</tbody>
</table>

REDUCE RISK WITH A TEST PLAN
The release of a new application or an upgrade inherently carries a certain amount of risk that it will fail to do what it’s supposed to do. A good test plan goes a long way towards reducing this risk. By identifying areas that are riskier than others we can concentrate our testing efforts there. These areas include not only the must-have features but also areas in which the technical staff is less experienced, perhaps such as the real-time loading of a web form’s

Contents into a database using complex ETL logic. Because riskier areas require more certainty that they work properly, failing to correctly identify those risky areas leads to a misallocated testing effort.

How do we identify risky areas? Ask everyone for their opinion! Gather information from developers, sales and marketing staff, technical writers, customer support people, and of course any users who are available. Historical data and bug and testing reports from similar products or previous releases will identify areas to explore. Bug reports from customers are important, but also look at bugs reported by the developers
themselves. These will provide insight to the technical areas they may be having trouble in. When the problems are inevitably found, it’s important that both the IT side and the business users have previously agreed on how to respond. This includes having a method for rating the importance of defects so that repair work effort can be focused on the most important problems. It is very common to use a set of rating categories that represent decreasing relative severity in terms of business/commercial impact. In one system, ‘1’ is the most severe and 6’ has the least impact. Keep in mind that an ordinal system doesn’t allow an average score to be calculated, but you shouldn’t need to do that anyway—a defect’s category should be pretty obvious.

1. Show Stopper – It is impossible to continue testing because of the severity of the defect.
2. Critical -- Testing can continue but the application cannot be released into production until this defect is fixed.
3. Major -- Testing can continue but this defect will result in a severe departure from the business requirements if released for production.
4. Medium -- Testing can continue and the defect will cause only minimal departure from the business requirements when in production.
5. Minor– Testing can continue and the defect will not affect the release into production. The defect should be corrected but little or no changes to business requirements are envisaged.
6. Cosmetic– Minor cosmetic issues like colors, fonts, and pitch size that do not affect testing or production release. If, however, these features are important business requirements then they will receive a higher severity level.

WHAT SHOULD A TEST PLAN TEST?
Testing experts generally agree that test plans are often biased towards functional testing during which each feature is tested alone in a unit test, and that the systems integration test is just a series of unit tests strung together. The problem that this approach causes is that if we test each feature alone and then string a bunch of these tests together, we might never find that a series of steps such as Open a document, edit the document, print the document, save the document, edit one page, Print one page, save as a new document doesn’t work. But a user will find out and probably quickly. Admittedly, testing every combination of keystrokes or commands is difficult at best and may well be impossible (this is where unstructured testing comes in), but we must remember that features don’t function in isolation from each other. Users have a task orientation. To find the defects that they will find—the ones that are important to them—test plans need to exercise the application across functional areas by mimicking both typical and atypical user tasks. A test like the sequence shown above is called scenario testing, task-based testing, or use-case testing.

An incomplete test plan can result in a failure to check how the application works on different hardware and operating systems or when combined with different third-party software. This is not always needed but you will want to think about the equipment your customers use. There may be more than a few possible system combinations that need to be tested, and that can require a possibly expensive computer lab stocked with hardware and spending much time setting up tests. Configuration testing isn't cheap, but it’s worth it when you discover that the application running on your standard in-house platform which "entirely conforms to industry standards" behaves differently when it runs on the boxes your customers are using. In a 1996 incident this author was involved in, the development and testing was done on new 386-class machines and the application worked just fine. Not until customers complained about performance did we learn that they were using 286’s with slow hard drives.

A crucial test is to see how the application behaves when it’s under a normal load and then under stress. The definition of stress, of course, will be derived from your business requirements, but for a web-enabled application stress could be caused by a spike in the number
of transactions, a few very large transactions at the same time, or a large number of almost identical simultaneous transactions. The goal is to see what happens when the application is pushed to substantially more than the basic requirements. Stress testing is often put off until the end of testing, after everything else that’s going to be fixed has been. Unfortunately that leaves little time for repairs when the requirements specify 40 simultaneous users and you find that performance becomes unacceptable at 50.

Finally, Marick (1997) points out two common omissions in many test plans—the installation procedures and the documentation are ignored. Everyone has tried to follow installation instructions that were missing a key step or two, and we’ve all paged through incomprehensible documentation. Although those documents may have been written by a professional technical writer, they probably weren’t tested by a real user. Bad installation instructions immediately cause lowered expectations of what to expect from the product, and poorly organized or written documentation certainly doesn’t help a confused or irritated customer feel better. Testing installation procedures and documentation is a good way to avoid making a bad first impression or making a bad situation worse.

### Test Plan Terminology

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Plan</td>
<td>A formal detailed document that describes ♦ Scope, objectives, and the approach to testing, ♦ People and equipment dedicated/allocated to testing ♦ Tools that will be used ♦ Dependencies and risks ♦ Categories of defects ♦ Test entry and exit criteria ♦ Measurements to be captured ♦ Reporting and communication processes ♦ Schedules and milestones</td>
</tr>
<tr>
<td>Test Case</td>
<td>A document that defines a test item and specifies a set of test inputs or data, execution conditions, and expected results. The inputs/data used by a test case should be both normal and intended to produce a ‘good’ result and intentionally erroneous and intended to produce an error. A test case is generally executed manually but many test cases can be combined for automated execution.</td>
</tr>
<tr>
<td>Test Script</td>
<td>Step-by-step procedures for using a test case to test a specific unit of code, function, or capability.</td>
</tr>
<tr>
<td>Test Scenario</td>
<td>A chronological record of the details of the execution of a test script. Captures the specifications, tester activities, and outcomes. Used to identify defects.</td>
</tr>
<tr>
<td>Test Run</td>
<td>A series of logically related groups of test cases or conditions.</td>
</tr>
</tbody>
</table>
TYPES OF SOFTWARE TESTS

The V-Model of testing identifies five software testing phases, each with a certain type of test associated with it.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Guiding Document</th>
<th>Test Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Phase</td>
<td>Technical Design</td>
<td>Unit Testing</td>
</tr>
<tr>
<td>System and Integration Phase</td>
<td>Functional Design</td>
<td>System Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration Testing</td>
</tr>
<tr>
<td>User Acceptance Phase</td>
<td>Business Requirements</td>
<td>User Acceptance Testing</td>
</tr>
<tr>
<td>Implementation Phase</td>
<td>Business Case</td>
<td>Product Verification Testing</td>
</tr>
</tbody>
</table>

Regression Testing applies to all Phases

Each testing phase and each individual test should have specific entry criteria that must be met before testing can begin and specific exit criteria that must be met before the test or phase can be certified as successful. The entry and exit criteria are defined by the Test Coordinators and listed in the Test Plan.

Sample Entry and Exit Criteria for Unit Testing

♦ Business Requirements are at least 80% complete and have been approved to-date
♦ Technical Design has been finalized and approved
♦ Development environment has been established and is stable
♦ Code development for the module is complete
♦ Code has version control in place
♦ No known major or critical defects prevents any modules from moving to
Exit Criteria System Testing
♦ A testing transition meeting has been held and
the developers signed off
♦ Project Manager approval has been received

SYSTEM TESTING

System Testing tests all components and modules that are new, changed, affected by a change, or needed to form the complete application. The system test may require involvement of other systems but this should be minimized as much as possible to reduce the risk of externally-induced problems. Testing the interaction with other parts of the complete system comes in Integration Testing. The emphasis in system testing is validating and verifying the functional design specification and seeing how all the modules work together. The first system test is often a smoke test. This is an informal quick-and-dirty run through of the application’s major functions without bothering with details. The term comes from the hardware testing practice of turning on a new piece of equipment for the first time and considering it a success if it doesn’t start smoking or burst into flame.

Sample Entry and Exit Criteria for System Testing

♦ Unit Testing for each module has been completed and approved; each module is under version control Entry Criteria
♦ An incident tracking plan has been approved
♦ A system testing environment has been established
♦ The system testing schedule is approved an
♦ Application meets all documented business and functional requirements
♦ No known critical defects prevent moving to the Integration Testing Exit Criteria
♦ All appropriate parties have approved the completed tests
♦ A testing transition meeting has been held and the developers signed off.

Integration testing has a number of sub-types of tests that may or may not be used, depending on the application being tested or expected usage patterns.

 Compatibility Testing – Compatibility tests insures that the application works with differently configured systems based on what the users have or may have. When testing a web interface, this means testing for compatibility with different browsers and connection speeds.
Performance Testing – Performance tests are used to evaluate and understand the application’s scalability when. This is particularly important for identifying bottlenecks in high usage applications. The basic approach is to collect timings of the critical business processes while the test system is under a very low load (a ‘quiet box’ condition) and then collect the same timings with progressively higher loads until the maximum required load is reached. For a data retrieval application, reviewing the performance pattern may show that a change needs to be made in a stored SQL procedure or that an index should be added to the database design.

Stress Testing – Stress Testing is performance testing at higher than normal simulated loads. Stressing runs the system or application beyond the limits of its specified requirements to determine the load under which it fails and how it fails. A gradual performance slow-down leading to a non-catastrophic system halt is the desired result, but if the system will suddenly crash and burn it’s important to know the point where that will happen. Catastrophic failure in production means beepers going off, people coming in after hours, system restarts, frayed tempers, and possible financial losses. This test is arguably the most important test for mission-critical systems.

Load Testing – Load tests are the opposite of stress tests. They test the capability of the application to function properly under expected normal production conditions and measure the response times for critical transactions or processes to determine if they are within limits specified in the business requirements and design documents or that they meet Service Level Agreements. For database applications, load testing must be executed on a current production-size database. If some database tables are forecast to grow much larger in the foreseeable future then serious consideration should be given to testing against a database of the projected size. Performance, stress, and load testing are all major undertakings and will require substantial input from the business sponsors and IT staff in setting up a test environment and designing test cases that can be accurately executed. Because of this, these tests are sometimes delayed and made part of the User Acceptance Testing phase. Load tests especially must be documented in detail so that the tests are repeatable in case they need to be executed several times to ensure that new releases or changes in database size do not push response times beyond prescribed requirements and Service Level Agreements.

PRODUCTION VERIFICATION TESTING

Production verification testing is a final opportunity to determine if the software is ready for release. Its purpose is to simulate the production cutover as closely as possible and for a period of time simulate real business activity. As a sort of full dress rehearsal, it should identify anomalies or unexpected changes to existing processes introduced by the new application. For mission critical applications the importance of this testing cannot be overstated. The application should be completely removed from the test environment and then completely reinstalled exactly as it will be in the production implementation. Then mock production runs will verify that the existing business process flows, interfaces, and batch processes continue to run correctly. Unlike parallel testing in which the old and new systems are run side-by-side, mock processing may not provide accurate data handling results due to limitations of the testing database or the source data.

Sample Entry and Exit Criteria for Production Verification Testing

- UAT testing has been completed and approved by all necessary parties.
- Known defects have been documented

Entry Criteria

- Migration package documentation has been completed, reviewed, and approved by the production systems manager.
- Package migration is complete.
- Installation testing has been performed and documented and the results have been signed off. Exit Criteria
- Results of the test show no negative impact to the application.
CONCLUSION
Software testing is a critical element in the software development life cycle and has the potential to save time and money by identifying problems early and to improve customer satisfaction by delivering a more defect-free product. Unfortunately, it is often less formal and rigorous than it should, and a primary reason for that is because the project staff is unfamiliar with software testing methodologies, approaches, and tools. To partially remedy this situation very software testing and terminology. Without adequate testing, however, there is a greater risk that an application will inadequately deliver what was expected by the business users or that the final product will have problems such that users will eventually abandon it out of frustration. In either case, time and money are lost and the credibility and reputation of both the developers.

REFERENCES AND RESOURCES:-
♦ Better Software magazine.
♦ Ergo/Gero H (information on color blindness)
♦ International Institute for Software Testing.
♦ Parasoft Corporation. “Automated Error Prevention”.
♦ Software Engineering Research Network, University of Calgary, Alberta, Canada.
♦ Software Testing Education.
♦ Vanderbilt University, MIS Application Services. “MIS Development Methodology”.