

# A proactive Approaches in Software Testing.

Er. Amandeep Kaur

Student. In CSE department, Gurukul Vidhapeeth  
BANUR, Patiala, Punjab, India

## Abstract:

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we cannot completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality.

## Introduction

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Software testing can be stated as the process of validating and verifying that a software program/application/product:

1. meets the requirements that guided its design and development;
2. works as expected;
3. Can be implemented with the same characteristics.
4. satisfies the needs of stakeholders

Software testing, depending on the testing method employed, can be implemented at any time in the development process. Traditionally most of the test effort occurs after the requirements have been defined and the coding process has been completed, but in the agile approaches most of the test effort is on-going. As such, the methodology of the test is governed by the chosen software development methodology.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear -- generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects -- or bugs -- will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable -- and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but his time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing. Testing is usually performed for the following purposes

- **To improve quality.**

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. Bugs can kill. Bugs can cause disasters. The so-called year 2000 bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. Being correct, the minimum requirement of quality, means performing as required under specified circumstances. Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible to make a moderately complex program correct the first time. Finding the problems and get them fixed the purpose of debugging in programming phase.

- **For Verification & Validation (V&V)**

Just as topic Verification and Validation indicated, another important purpose of testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test.

We cannot test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors -- functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space.

Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests cannot validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests, or negative tests, refer to the tests aiming at breaking the software, or showing that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests.

A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development.

- **For reliability estimation**

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program, testing can serve as a statistical sampling method to gain failure data for reliability estimation.

Software testing is not mature. It still remains an art, because we still cannot make it a science. We are still using the same testing techniques invented 20-30 years ago, some of which are crafted methods or heuristics rather than good engineering methods. Software testing can be costly, but not testing software is even more expensive, especially in places that human lives are at stake. Solving the software-testing problem is no easier than solving the Turing halting problem. We can never be sure that a piece of software is correct. We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct either.

## Testing Methods

### Taxonomy

There is a plethora of testing methods and testing techniques, serving multiple purposes in different life cycle phases. Classified by purpose, software testing can be divided into: correctness testing, performance testing, reliability testing and security testing. Classified by life-cycle phase, software testing can be classified into the following categories: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing. By scope, software testing can be categorized as follows: unit testing, component testing, integration testing, and system testing .

## Correctness testing

Correctness is the minimum requirement of software, the essential purpose of testing. Correctness testing will need some type of oracle, to tell the right behavior from the wrong one. The tester may or may not know the inside details of the software module under test, e.g. control flow, data flow, etc. Therefore, either a white-box point of view or black-box point of view can be taken in testing software. We must note that the black-box and white-box ideas are not limited in correctness testing only.

- **Black-box testing**

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. It is also termed data-driven, input/output driven, or requirements-based testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing -- a testing method emphasized on executing the functions and examination of their input and output data. The tester treats the software under test as a black box -- only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

It is obvious that the more we have covered in the input space, the more problems we will find and therefore we will be more confident about the quality of the software. Ideally we would be tempted to exhaustively test the input space. But as stated above, exhaustively testing the combinations of valid inputs will be impossible for most of the programs, let alone considering invalid inputs, timing, sequence, and resource variables. Combinatorial explosion is the major roadblock in functional testing. To make things worse, we can never be sure whether the specification is either correct or complete. Due to limitations of the language used in the specifications (usually natural language), ambiguity is often inevitable. Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem: it is not possible to specify precisely every situation that can be encountered using limited words. And people can seldom specify clearly what they want -- they usually can tell whether a prototype is, or is not, what they want after they have been finished. A specification problem contributes approximately 30 percent of all bugs in software.

The research in black-box testing mainly focuses on how to maximize the effectiveness of testing with minimum cost, usually the number of test cases. It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Partitioning is one of the common techniques. If we have partitioned the input space and assume all the input values in a partition is equivalent, then we only need to test one representative value in each partition to sufficiently cover the whole input space. Domain testing partitions the input domain into regions, and considers the input values in each domain an equivalent class. Domains can be exhaustively tested and covered by selecting a representative value(s) in each domain. Boundary values are of special interest. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification cannot be efficiently discovered.

Good partitioning requires knowledge of the software structure. A good testing plan will not only contain black-box testing, but also white-box approaches, and combinations of the two.

- **White-box testing**

Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called glass-box testing, logic-driven testing or design-based testing .

There are many techniques available in white-box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white-box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once (statement coverage), traverse every branch statements (branch coverage), or cover all the possible combinations of true and false condition predicates (Multiple condition coverage).

Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code -- code that is of no use, or never get executed at all, which cannot be discovered by functional testing.

In mutation testing, the original program code is perturbed and many mutated programs are created, each contains one fault. Each faulty version of the program is called a mutant. Test data are selected based on the effectiveness of

failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is too computationally expensive to use. The boundary between black-box approach and white-box approach is not clear-cut. Many testing strategies mentioned above, may not be safely classified into black-box testing or white-box testing. It is also true for transaction-flow testing, syntax testing, finite-state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad -- it may contain any requirement including the structure, programming language, and programming style as part of the specification content.

We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward: they are randomly chosen. Study in indicates that random testing is more cost effective for many programs. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. One can also obtain reliability estimate using random testing results based on operational profiles. Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies.

### **Performance testing**

Not all software systems have specifications on performance explicitly. But every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes: resource usage, throughput, and stimulus-response time and queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources. Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage. The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark -- a program, workload or trace designed to be representative of the typical system usage.

### **Reliability testing**

Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black-box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on reliability information advocates that the primary goal of testing should be to measure the dependability of tested software.

There is agreement on the intuitive meaning of dependable software: it does not fail in unexpected or catastrophic ways. Robustness testing and stress testing are variances of reliability testing based on this simple criterion.

The robustness of a software component is the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions. Robustness testing differs with correctness testing in the sense that the functional correctness of the software is not of concern. It only watches for robustness problems such as machine crashes, process hangs or abnormal termination. The oracle is relatively simple, therefore robustness testing can be made more portable and scalable than correctness testing. This research has drawn more and more interests recently, most of which uses commercial operating systems as their target, such as the work in [Koopman97] [Kropp98] [Ghosh98] [Devale99] [Koopman99].

Stress testing, or load testing, is often used to test the whole system rather than the software alone. In such tests the software or system are exercised with or beyond the specified limits. Typical stress includes resource exhaustion, bursts of activities, and sustained high loads.

### **Security testing**

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to find vulnerabilities.

## Testing automation

Software testing can be very costly. Automation is a good way to cut down time and cost. Software testing tools and techniques usually suffer from a lack of generic applicability and scalability. The reason is straight-forward. In order to automate the process, we have to have some ways to generate oracles from the specification, and generate test cases to test the target software against the oracles to decide their correctness. Today we still don't have a full-scale system that has achieved this goal. In general, significant amount of human intervention is still needed in testing. The degree of automation remains at the automated test script level.

### When to stop testing?

Testing is potentially endless. We cannot test till all the defects are unearthed and removed -- it is simply impossible. At some point, we have to stop testing and ship the software. The question is when.

Realistically, testing is a trade-off between budget, time and quality. It is driven by profit models. The pessimistic and unfortunately most often used approach is to stop testing whenever some or any of the allocated resources -- time, budget, or test cases -- are exhausted. The optimistic stopping rule is to stop testing when either reliability meets the requirement, or the benefit from continuing testing cannot justify the testing cost. This will usually require the use of reliability models to evaluate and predict reliability of the software under test. Each evaluation requires repeated running of the following cycle: failure data gathering -- modeling -- prediction. This method does not fit well for ultra-dependable systems, however, because the real field failure data will take too long to accumulate.

### Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:
  - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
  - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
  - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

- A regression testing technique is to have a standard set of tests, which cover existing functionality that result in persistent tabular data, and to compare pre-change data to post-change data, where there should not be differences, using a tool like diffkit. Differences detected indicate unexpected functionality changes or "regression".

These are tools for software testing automation from development to execution..

- Win Runner
- Load Runner
- Test Director
- Silk Test
- Test Partner

## Conclusions

- Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. It is nowhere near maturity, although there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience and intuition, together with proper techniques.
- Testing is more than just debugging. Testing is not only used to locate defects and correct them. It is also used in validation, verification process, and reliability measurement.
- Testing is expensive. Automation is a good way to cut down cost and time. Testing efficiency and effectiveness is the criteria for coverage-based testing techniques.
- Complete testing is infeasible. Complexity is the root of the problem. At some point, software testing has to be stopped and product has to be shipped. The stopping time can be decided by the trade-off of time and budget. Or if the reliability estimate of the software product meets requirement.

- Testing may not be the most effective method to improve software quality. Alternative methods, such as inspection, and clean-room engineering, may be even better.

**References:**

- [1] Mike Lutz, Rochester Institute of Technology, "Testing Tools", IEEE, 0740-7459/90/0500/0053, May 1990
- [2] PERRY90: A standard for testing application software, William E. Perry, 1990
- [3] IEEE90: IEEE Standard Glossary of Software Engineering Terminology
- [4] Bertrand Meyer, "Seven Principles of Software Testing,"
- [5] IEEE (1998). IEEE standard for software test documentation. : IEEE. ISBN 0-7381-1443-X.