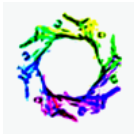
JCSM
Journal of Computational Simulation and Modeling
Volume 1, Issue 1, 2011, pp-01-09
Available online at http://www.bioinfo.in/contents.php?id=47



SOFTWARE TESTING

DINESH CHOUDHARY[1] AND VIJAY KUMAR[2]
[1]Department of Computer Science & Engineering, Kautilya Institute of Technology & Engineering and School of Management, Jaipur, India, dinesh_matwa@yahoo.co.in
[2]Department of Computer Science & Engineering, Stani Memorial College of Engineering & Technology, Jaipur, India, vijay_matwa@yahoo.com

Abstract- Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs.
Testing can never completely establish the correctness of computer software. Instead, it furnishes a criticism or comparison that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.
Over its existence, computer software has continued to grow in complexity and size. Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it presumably must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment. A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed. A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions.
The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Introduction

When a program is implemented to provide a concrete representation of an algorithm, the developers of this program are naturally concerned with the correctness and performance of the implementation. Software engineers must ensure that their software systems achieve an appropriate level of quality. Software verification is the process of ensuring that a program meets its intended specification [Kaner et al., 1993].
One technique that can assist during the specification, design, and implementation of a software system is software verification through correctness proof. Software testing, or the process of assessing the functionality and correctness of a program through execution or analysis, is another alternative for verifying a software system. As noted by Bowen, Hinchley, and Geller, software testing can be appropriately used in conjunction with correctness proofs and other types of formal approaches in order to develop high quality software systems [Bowen and Hinchley, 1995, Geller, 1978]. Yet, it is also possible to use software testing techniques in isolation from program correctness proofs or other formal methods.
Software testing is not a “silver bullet” that can guarantee the production of high quality software systems. While a “correct” correctness proof demonstrates that a software system (which exactly meets its specification) will always operate in a given manner, software testing that is not fully exhaustive can only suggest the presence of flaws and cannot prove their absence. Moreover, Kaner et al. have noted that it is impossible to completely test an application because [Kaner et al., 1993]: (1) the domain of program inputs is too large, (2) there are too many possible input paths, and (3) design and specification issues are difficult to test. The first and second points present obvious complications

and the final point highlights the difficulty of determining if the specification of a problem solution and the design of its implementation are also correct.

Using a thought experiment developed by Beizer, we can explore the first assertion by assuming that we have a method that takes a String of ten characters as input and performs some arbitrary operation on the String. In order to test this function exhaustively, we would have to input 280 Strings and determine if they produce the appropriate output.1 the testing of our hypothetical method might also involved the usage of anomalous input, like Strings consisting of more or less than ten characters, to determine the robustness of the operation. In this situation, the total number of inputs would be significantly greater than 280. Therefore, we can conclude that exhaustive testing is an intractable problem since it is impossible to solve with a polynomial-time algorithm [Binder, 1999, Neapolitan and Naimipour, 1998]. The difficulties alluded to by the second assertion are exacerbated by the fact that certain execution paths in a program could be infeasible. Finally, software testing is an algorithmically unsolvable problem since there may be input values for which the program does not halt [Beizer, 1990, Binder, 1999]. Thus far, we have provided an intuitive understanding of the limitations of software testing. However, Morell has proposed a theoretical model of the testing process that facilitates the proof of pessimistic theorems that clearly state the limitations of testing [Morell, 1990]. Furthermore, Hamlet and Morell have formally stated the goals of a software testing methodology and implicitly provided an understanding of the limitations of testing [Hamlet, 1994, Morell, 1990]. Young and Taylor have also observed that every software testing technique must involve some tradeoff between accuracy and computational cost because the presence (or lack thereof) of defects within a program is an undecidable property [Young and Taylor, 1989]. The theoretical limitations of testing clearly indicate that it is impossible to propose and implement a software testing methodology that is completely accurate and applicable to arbitrary programs [Young and Taylor, 1989]. While software testing is certainly faced with inherent limitations, there are also a number of practical considerations that can hinder the application of a testing technique. For example, some programming languages might not readily support a selected testing approach, a test automation framework might not easily facilitate the automatic execution of certain types of test suites, or there could be a lack of tool support to test with respect to a specific test adequacy criterion. Even though any testing effort will be faced with significant essential and accidental limitations, the rigorous, consistent, and intelligent application of appropriate software testing techniques can improve the quality of the application under development.

Software testing and analysis is an active research area. The ACM/IEEE International Conference on Software Engineering, the ACM SIGSOFT Symposium on the Foundations of Software Engineering, the ACMSIGSOFT International Symposium on Software Testing and Analysis, and the ACM SIGAPP Symposium on Applied Computing's Software Engineering Track are all important forums for new research in the areas of software engineering and software testing and analysis. Other important conferences include: IEEE Automated Software Engineering, IEEE International Conference on Software Maintenance, IEEE International Symposium on Software Reliability Engineering, the IEEE/NASA Software Engineering Workshop, and the IEEE Computer Software and Applications Conference.

There are also several magazines and journals that provide archives for important software engineering and software testing research. The IEEE Transactions on Software Engineering and the ACM Transactions on Software Engineering and Methodology are two noteworthy journals that often publish software testing papers. Other journals include: Software Testing, Verification, and Reliability, Software: Practice and Experience, Software Quality Journal, Automated Software Engineering: An International Journal, and Empirical Software Engineering: An International Journal. Magazines that publish software testing articles include Communications of the ACM, IEEE Software, IEEE Computer, and Better Software (formerly known as Software Testing and Quality Engineering). ACM SIGSOFT also sponsors the bi-monthly newsletter called Software Engineering Notes.

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. [Hetzel88] 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 can not 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.

Software Testing is the process of executing a program or system with the intent of finding errors. [Myers79] Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required

results. [Hetzel88] Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible. [Rstcorp]

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 (yielding 2^64 distinct test cases) 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. [Rstcorp]

An interesting analogy parallels the difficulty in software testing with the pesticide, known as the Pesticide Paradox [Beizer90]: *Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual*. But this alone will not guarantee to make the software better, because the Complexity Barrier [Beizer90] principle states: Software complexity (and therefore that of bugs) *grows to the limits of our ability to manage that complexity*. 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. [Beizer90]

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 (Y2K) 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. [Bugs] 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 [Kaner93], is 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 can not 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. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1 illustrates some of the most frequently cited quality considerations.

Table 1- Typical Software Quality Factors [Hetzel88]

Functionality (exterior quality)	**Engineering (interior quality)**	**Adaptability (future quality)**
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Good testing provides measures for all relevant factors. The importance of any particular factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each relevant factor and thus forcing quality to become tangible and visible. [Hetzel88]

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 can not 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, refers 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 [Kaner93] [Lyu95]

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 [Lyu95]), 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.

Key Concepts

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. [Perry90] It is also termed data-driven, input/output driven [Myers79], or requirements-based [Hetzel88] 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. [Howden87] 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. Specification problems contributes approximately 30 percent of all bugs in software. [Beizer95]

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 [Beizer95] partitions the input domain into regions, and consider 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 [Myers79] requires one or more boundary values selected as representative test cases. The difficulties with domain testing are that incorrect domain definitions in the specification can not 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 [Myers79] or design-based testing [Hetzel88].

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). [Parrington89]

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 can not 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 [Duran84] 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, 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 [Smith90]. 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. [Vokolos98]

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. [Hamlet94] 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. [Hamlet94] 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. [IEEE90] 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.

The problem is lessened in reliability testing and performance testing. In robustness testing, the simple specification and oracle: doesn't crash, doesn't hang suffices. Similar simple metrics can also be used in stress testing.

When to stop testing?

Testing is potentially endless. We can not 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. [Yang95] 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.

Alternatives to testing

Software testing is more and more considered a problematic method toward better quality. Using testing to locate and correct software defects can be an endless process. Bugs cannot be completely ruled out. Just as the complexity barrier indicates: chances are testing and fixing problems may not necessarily improve the quality and reliability of the software. Sometimes fixing a problem may introduce much more severe problems into the system, happened after bug fixes, such as the telephone outage in California and eastern seaboard in 1991. The disaster happened after changing 3 lines of code in the signaling system.

In a narrower view, many testing techniques may have flaws. Coverage testing, for example. Is code coverage, branch coverage in testing really related to software quality? There is no definite proof. As early as in [Myers79], the so-called "human testing" -- including inspections, walkthroughs, reviews -- are suggested as possible alternatives to traditional testing methods. [Hamlet94] advocates inspection as a cost-effect alternative to unit testing. The experimental results in [Basili85] suggests that code reading by stepwise abstraction is at least as effective as on-line functional and structural testing in terms of number and cost of faults observed.

Using formal methods to "prove" the correctness of software is also an attracting research direction. But this method can not surmount the complexity barrier either. For relatively simple software, this method works well. It does not scale well to those complex, full-fledged large software systems, which are more error-prone.

In a broader view, we may start to question the utmost purpose of testing. Why do we need more effective testing methods anyway, since finding defects and removing them does not necessarily lead to better quality. An analogy of the problem is like the car manufacturing process. In the craftsmanship epoch, we make cars and hack away the problems and defects. But such methods were washed away by the tide of pipelined manufacturing and good quality engineering process, which makes the car defect-free in the manufacturing phase. This indicates that engineering the design process (such as clean-room software engineering) to make the product have less defects may be more effective than engineering the testing process. Testing is used solely for quality monitoring and management, or, "design for testability". This is the leap for software from craftsmanship to engineering.

Available tools, techniques, and metrics

There are an abundance of software testing tools exist. The correctness testing tools are often specialized to certain systems and have limited ability and generality. Robustness and stress testing tools are more likely to be made generic.

Mothora [DeMillo91] is an automated mutation testing tool-set developed at Purdue University. Using Mothora, the tester can create and execute test cases, measure test case adequacy, determine input-output correctness, locate and remove faults or bugs, and control and document the test.

NuMega's Boundschecker [NuMega99] Rational's *Purify* [Rational99]. They are run-time checking and debugging aids. They can both check and protect against memory leaks and pointer problems.

Ballista COTS Software Robustness Testing Harness [Ballista99]. The Ballista testing harness is an full-scale automated robustness testing tool. The first version supports testing up to 233 POSIX function calls in UNIX operating systems. The second version also supports testing of user functions provided that the data types are recognized by the testing server. The Ballista testing harness gives quantitative measures of robustness comparisons across operating systems. The goal is to automatically test and harden Commercial Off-The-Shelf (COTS) software against robustness failures.

Relationship to other topics

Software testing is an integrated part in software development. It is directly related to software quality. It has many subtle relations to the topics that software, software quality, software reliability and system reliability are involved.

Related topics

- Software reliability Software testing is closely related to software reliability. Software reliability can be augmented by testing. Also testing can be served as a metric for software reliability.
- Fault injection Fault injection can be considered a special way of testing. Fault injection and testing are usually combined and performed to validate the reliability of critical fault-tolerant software and hardware.
- Verification, validation and certification The purpose of software testing is not only for revealing bugs and eliminate

them. It is also a tool for verification, validation and certification.

Conclusion

- 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.

Annotated References

1. [Ballista99] http://www.cs.cmu.edu/afs/cs/project/edrc-ballista/www/
2. [Basili85] Victor R. Basili, Richard W. Selby, Jr. "Comparing the Effectiveness of Software Testing Strategies",
3. [Beizer90] Boris Beizer, Software Testing Techniques. Second edition. 1990
4. A very comprehensive book on the testing techniques. Many testing techniques are enumerated and discussed in detail. Domain testing, data-flow testing, transactin-flow testing, syntax testing, logic-based testing, etc.
5. [Beizer95] Beizer, Boris, *Black-box Testing: techniques for functional testing of software and systems.* Publication info: New York : Wiley, c1995. ISBN: 0471120944 Physical description: xxv, 294 p.: ill. ; 23 cm.
6. [Duran84] Joe W. Duran, Simeon C. Ntafos, "An Evaluation of Random Testing", IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, July 1984, pp438-443.
7. [Hetzel88] Hetzel, William C., *The Complete Guide to Software Testing, 2nd ed.* Publication info: Wellesley, Mass. : QED Information Sciences, 1988. ISBN: 0894352423.Physical description: ix, 280 p. : ill ; 24 cm.
8. [Howden87] William E. Howden. Functional program Testing and Analysis. McGraw-Hill, 1987.
9. [IEEE90] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE Computer Soc., Dec. 10, 1990.
10. [Kaner93] Cem Kaner, Testing Computer Software. 1993.
11. Discusses the purpose and techniques for software testing.
12. [Lyu95] Michael R. Lyu , Handbook of Software Reliability Engineering. McGraw-Hill publishing, 1995, ISBN 0-07-039400-8
13. [Rational99] http://www.rational.com/products/purify_unix/index.jtmpl
14. [Rstcorp] http://www.rstcorp.com/definitions/software_testing.html
15. [PERRY90] A standard for testing application software, William E. Perry, 1990
16. [Musa97] Software-reliability-engineered testing practice (tutorial);John D. Musa;

References

[1] G. Rothermel, L. Li, and M. Burnett. Testing strategies for form-based visual programs. In Proceedings of the 8th International Symposium on Software Reliability Engineering, pages 96–107, Albuquerque, NM, November 1997.

[2] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In Proceedings of the International Conference on Software Maintenance, pages 179–188, August 1999.

[3] G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J. Harrold. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10):929–948, October 2001a.

[4] Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A methodology for testing spreadsheets. ACM Transactions on Software Engineering and Methodology, 10(1):110–147, January 2001b.

[5] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation. In Proceedings of the 22nd International Conference on Software Engineering, pages 230–239. ACM Press, 2000.

[6] Andrew Sears. Layout appropriateness: A metric for evaluating user interface widget layout. IEEE Transactions on Software Engineering, 19(7):707–719, 1993.

[7] Forrest Shull, Ioana Rus, and Victor Basili. Improving software inspections by using reading techniques. In Proceedings of the 23rd International Conference on Software Engineering, pages 726–727. IEEE Computer Society, 2001.

[8] Ian Sommerville. Software Engineering. Addison-Wesley, 6th edition, August 2000.

[9] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. In Proceedings of the International Symposium on Software Testing and Analysis, pages 158–167. ACM Press, 2000.

[10] T. Tsai and R. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In Proceedings of the 8th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, pages 26–40, 1995.

[11] Timothy K. Tsai and Navjot Singh. Reliability testing of applications on Windows NT. In Proceedings of the International Conference on Dependable Systems and Networks, New York City, USA, June 2000.

[12] Raja Vall´ee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In Proceedings of CASCON 1999, pages 125–135, 1999.

[13] Jeffrey M. Voas. PIE: a dynamic failure-based technique. IEEE Transactions on Software Engineering, 18(8):717– 735, 1992.

[14] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In Third International Conference of Reliability, Quality, and Safety of Software Intensive Systems, May 1997.

[15] Elaine Weyuker. Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering, (12): 1128–1138, December 1986.

[16] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In Proceedings of the Symposium on Testing, Analysis, and Verification, pages 1–10. ACM Press, 1991.

[17] James A. Whittaker. What is software testing? and why is it so hard? IEEE Software, 17(1):70–76, January/February 2000.

[18] James A. Whittaker and Jeffrey Voas. Toward a more reliable theory of software reliability. IEEE Computer, 32(12): 36–42, December 2000.

[19] W.E. Wong. On Mutation and Data Flow. PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, December 1993.

[20] W.E.Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In Proceedings of the 8th International Symposium on Software Reliability Engineering, pages 230–238, November 1997.

[21] Michael Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In Proceedings of the 11th International Conference on Software Engineering, pages 53–62. ACM Press, 1989.

[22] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. ACM Computing Surveys, 29(4):366–427, 1997.