Volume 3, Issue 1, January 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Study on Software Testing

H.S. Samra
Punjab Polytechnic College
Punjab, India

Abstract— ***Testing is an important and critical part of the software development process, on which the quality and reliability of the delivered product strictly depend. Testing is not limited to the detection of "bugs" in the software, but also increases confidence in its proper functioning and assists with the evaluation of functional and non-functional properties. Testing related activities encompass the entire development process and may consume a large part of the effort required for producing software. In this paper we provide a comprehensive overview of software testing, from its definition to its organization, from test levels to test techniques, from test execution to the analysis of test cases effectiveness. Emphasis is more on breadth than depth: due to the vastness of the topic, in the attempt to be all-embracing, for each covered subject we can only provide a brief description and references useful for further reading.***

Keywords: V& V, Unit Testing, Integration Testing.

I. INTRODUCTION

Software Testing is the process of executing a program or system with the intent of finding errors 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 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. 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.

II. OBJECTIVE

A. *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. 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, is the purpose of debugging in programming phase.

B. *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. 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.

C. 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.

III. TEST LEVELS

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process are unit-, integration-, and system testing that is distinguished by the test target without implying a specific process model.

A. Unit Testing

Unit testing deals with testing a unit as a whole. This would test the interaction of many functions but confine the test within one unit. The exact scope of a unit is left to interpretation. Supporting test code, sometimes called scaffolding, may be necessary to support an individual test. This type of testing is driven by the architecture and implementation teams. This focus is also called black-box testing because only the details of the interface are visible to the test. Limits that are global to a unit are tested here.

In the construction industry, scaffolding is a temporary, easy to assemble and disassemble, frame placed around a building to facilitate the construction of the building. The construction workers first build the scaffolding and then the building. Later the scaffolding is removed, exposing the completed building. Similarly, in software testing, one particular test may need some supporting software. This software establishes an environment around the test. Only when this environment is established can a correct evaluation of the test take place. The scaffolding software may establish state and values for data structures as well as providing dummy external functions for the test. Different scaffolding software may be needed from one test to another test. Scaffolding software rarely is considered part of the system.

Sometimes the scaffolding software becomes larger than the system software being tested. Usually the scaffolding software is not of the same quality as the system software and frequently is quite fragile. A small change in the test may lead to much larger changes in the scaffolding.

Internal and unit testing can be automated with the help of coverage tools. A coverage tool analyses the source code and generates a test that will execute every alternative thread of execution. It is still up to the programmer to combine these tests into meaningful cases to validate the result of each thread of execution. Typically, the coverage tool is used in a slightly different way. First the coverage tool is used to augment the source by placing informational prints after each line of code. Then the testing suite is executed generating an audit trail. This audit trail is analysed and reports the per cent of the total system code executed during the test suite. If the coverage is high and the untested source lines are of low impact to the system's overall quality, then no more additional tests are required.

B. Integration Testing

Objective of Integration testing is to make sure that the interaction of two or more components produces results that satisfy functional requirement. In integration testing, test cases are developed with the express purpose of exercising the interface between the components. Integration testing can also be treated as testing assumption of fellow programmer. During the coding phase, lots of assumptions are made. Assumptions can be made for how you will receive data from different components and how you have to pass data to different components.

During Unit Testing, these assumptions are not tested. Purpose of unit testing is also to make sure that these assumptions are valid. There could be many reasons for integration to go wrong, it could be because Interface Misuse . A calling component calls another component and makes an error in its use of interface, probably by calling/passing parameters in the wrong sequence. Interface Misunderstanding A calling component makes some assumption about the other components behavior which are incorrect.

Integration Testing can be performed in three different ways based on the from where you start testing and in which direction you are progressing.

i. Big Bang Integration Testing

In big bang Integration testing, individual modules of the programs are not integrated until everything is ready. This approach is seen mostly in inexperienced programmers who rely on 'Run it and see' approach. In this approach, the program is integrated without any formal integration testing, and then run to ensures that all the components are working properly.

ii. Top Down Integration Testing

In this approach testing is conducted from main module to sub module. if the sub module is not developed a temporary program called STUB is used for simulate the sub module. A 'Stub' is a piece of software that works similar to a unit which is referenced by the Unit being tested, but it is much simpler that the actual unit. A Stub works as a 'Stand-in' for the subordinate unit and provides the minimum required behaviour for that unit. A Stub is a dummy procedure, module or unit that stands in for an unfinished portion of a system.

iii. Bottom Up Integration Testing

In this approach testing is conducted from sub module to main module, if the main module is not developed a temporary program called DRIVERS is used to simulate the main module. A 'Driver' is a piece of software that drives (invokes) the Unit being tested. A driver creates necessary 'Inputs' required for the Unit and then invokes the Unit. Driver passes test cases to another piece of code. Test Harness or a test driver is supporting code and data used to provide an environment for testing part of a system in isolation. It can be called as as a software module which is used to invoke a module under test and provide test inputs, control and, monitor execution, and report test results or most simplistically a line of code that calls a method and passes that method a value.

iv. Hybrid Integration testing

Testing technique which combines top-down and bottom-up integration techniques in order leverage benefits of these kind of testing. It is usually performed by the testing teams

C. System Testing

System Testing' is the next level of testing. It focuses on testing the system as a whole. How does System Testing fit into the Software Development Life Cycle? In a typical Enterprise, 'unit testing' is done by the programmers. This ensures that the individual components are working OK. The 'Integration testing' focuses on successful integration of all the individual pieces of software (components or units of code). Once the components are integrated, the system as a whole needs to be rigorously tested to ensure that it meets the Quality Standards. Thus the System testing builds on the previous levels of testing namely unit testing and Integration Testing. Usually a dedicated testing team is responsible for doing 'System Testing'.

The following steps are important to perform System Testing:

Step 1: Create a System Test Plan
Step 2: Create Test Cases
Step 3: Carefully Build Data used as Input for System Testing
Step 4: Execute the test cases
Step 5: Fix the bugs if any and re test the code
Step 6: Repeat the test cycle as necessary

a) System Test Case

A Test Case describes exactly how the test should be carried out. The System test cases help us verify and validate the system. The System Test Cases are written such that:

The format of the System Test Cases (figure 1) may be like all other Test cases as illustrated below:

- Test Case ID
- Test Case Description:
- What to Test?
- How to Test?
- Input Data
- Expected Result
- Actual Result

Sample Test Case Format:

Test Case ID	W to Test?	How to Test?	Input Data	Expected Result	Actual Result

Figure 1. Shows Test Case Format.

Additionally the following information may also be captured:
a) Test Suite Name
b) Tested By
c) Date
d) Test Iteration (The Test Cases may be executed one or more times)

4. *Software Integration Testing*

Integration testing (sometimes called Integration and Testing, abbreviated "I&T) is the activity of software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Integration testing is executed on individual software modules that are combined and tested. Principle of integration testing is to validate functional, performance and reliability.

IV. CONCLUSION

Clearly, software doesn't have to be 100% bug free. In fact, one of the hardest problems with testing is to know when to stop. If your company puts a team of testers on a project, and they spend four weeks on the finished product, they may find a lot of bugs the first week, some the second week, few the third week, and none the fourth week. But just because they found no bugs in the fourth week doesn't mean there are none. There is no practical way to prove that any piece of real world software is devoid of bugs, even a well-tested piece of software. In addition, functionality for expert users of software often doesn't get tested as well as the basic functionality, because testers are rarely expert users. No one wants to get a reputation for software that is not robust in the eyes of their expert users, because expert users have an impact on the usage habits of novice users. If these users get upset, your entire user base could slowly migrate to another product, even if you tested it fairly thoroughly!

Testing is generally considered costly and a nuisance. But as we have just seen, it is a necessary nuisance. The goal for most companies should be to do the best job testing possible and to minimize the costs. The idea that seems to work best is "test early and test often." Robustness isn't a module that can be bolted onto the side of a preexisting system -- it is far more cost-effective to develop robust software if you strive for this quality from day one. Similarly, the more software is tested, the more bugs will be found.

REFERENCES

[1]. *Software Testing Foundations*, Andreas Spillner, Tilo Linz, Hans Schaefer, Shoff Publishers and Distributors
[2]. *Software Testing: Principles and Practices* by Srinivasan D and Gopalswamy R, PearsonEd, 2006
[3]. *Foundations of Software Testing* by Aditya P. Mathur – Pearson Education custom edition 2000
[4]. *Testing Object Oriented Systems: models, patterns and tools*, Robert V Binder, Addison Wesley, 1996
[5]. *Software Engineering* – A practitioner's approach by Roger S. Pressman, 5th Edition, McGraw Hill
[6]. The art of software testing by GJ Myers, Wiley.
[7]. Exploratory Testing, Cem Kaner, Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, November 2006
[8]. *Software Testing* by Jiantao Pan, Carnegie Mellon University
[9]. Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development – Writing Test Cases", Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
[10]. Software errors cost U.S. economy $59.5 billion annually, NIST report
[11]. Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons.

