The Theory of Software Testing

Article *in* Intelligent Transportation Systems Journal · June 2012

CITATIONS
4

READS
15,269

1 author:

Adtha Lawanna
Assumption University of Thailand
27 PUBLICATIONS **31** CITATIONS

SEE PROFILE

The Theory of Software Testing

Adtha Lawanna

Department of Information Technology, Faculty of Science and Technology
Assumption University, Bangkok, Thailand
E-mail: <adtha@scitech.au.edu>

Abstract

Software testing is the process of testing bugs in lines of code of a program that can be performed by manual or automation testing. The theory of software testing involves problem definitions of testing such as test team, failure after testing, manual testing, uncertainty principle, participation, and incorrect test case selection. This article shows the details of a critical part of software testing, which is how to test the performance of new software and the entire system. The outcome of this article is the whole picture of three phases for software testing as follows: preliminary testing, testing and user acceptance testing.

Keywords*: Automation testing, verification, validation, user acceptance testing.*

1. Introduction

Software testing in the Software Development Life Cycle (SDLC) is the process of executing and evaluating a program with the intent of finding faults (Myers 1979). In general, it focuses on any activity aimed at evaluating an attribute or capability of a program or system and finding that it meets its specification requirements (Hetzel 1988). In fact, a software system is not unlike other physical systems where inputs are received and outputs are produced (Pan 1999). However, software does not suffer from physical changes. Generally, it will not change until modifications, upgrades, or user requirement changes take place (Pan 1999). Therefore, once the software is released, the design faults or bugs will be buried in and remain latent until activation (Pan 1999). All the possible values are required to be verified and tested, but perfect testing is impossible to be made (Humphrey 1995). If a testing failure occurs during a preliminary step and the code is released, the program may now work for a test case that it may not have been intended to work for previously (Pan 1999). But its behavior on pre-debugging test cases that it dealt with before can no longer be trusted (Pan 1999). To respond to this possibility, the testing has to be restarted. The expense of this step often tends to be discouraging (Pan 1999). In response to this, verification can be used to test or check items, including code, for consistency and conformance by evaluating the results against pre-specified requirements. Debugging implies that testing should intentionally aim to make things go wrong to find if things occur when they should not or things do not occur when they should. The validation involves the system correctness e.g., it is the process of checking whether the requirement that should be specified is what the user really wants. This means that the validation process checks whether the invented system is what the customer wants and needs to be included in it, while the verification process checks whether that system is invented correctly (Fischer 1977). Therefore, both verification and validation are necessary as distinct elements of any testing activity. This article shows three phases of software testing in order to consider possible requirements which can affect the abilities of the whole system.

2. The Theory of Software Testing

2.1 Problems in Software Testing

This article is concerned with five problems found in software testing as follows. The first problem is concerned with the limitations of the testing team. Insufficiency may be the result of limited resources, lack of

training of the individual team members, or issues with the leadership. Also, suitable testing means may not be available to the team. The second problem is failure of the test maintenance. This is because the specification changes of the requirements result in abnormally long reverse times. The third problem is with manual testing. The software testing team is busy making manual testing instead of building new test specifications, or modifying old ones to fit a new or changed requirement. The forth problem is about the uncertainty principle. Sometimes, the uncertainty is in the exact testing conditions as well as how to view the condition for replication. The fifth problem is in selecting the right tests. The software testing cannot address some of the essential aspects of the software testing application or system when testing only some part of the required functions or choosing only the expected interactions when executing a fault-tolerant application.

2.2 Processes in Software Testing

The processes of verification and validation are discussed below. Briefly, verification makes the product right, but validation makes the right product. Many testers use white-box testing in these processes. It deals with the investigation of internal logic and structure of the code. Some important processes of white-box testing are briefly described. Data Flow Testing is the process that can define and use program variables (Horgan and London 1991). Loop Testing exclusively concerns the validity of loop constructs. Branch Testing tests true and false values given to compound conditions appearing in different branch statements (Jorgensen 2002). Control Flow Testing is a structural testing that applies the program's control flow as a model and selects a set of test paths through the program (Rapps and Weyuker 1985). Basis Path Testing allows the test suite designer to build a logical complexity measure of procedural design and then applies this measure for determining a basic set of execution paths (Clarke *et al*. 1989). Besides this, some testers use black-box testing instead of white-box testing for the examination of the fundamental aspects of a system with little regard to its internal logical structure (Beizer 1995). There are several typical processes of black-box testing. Equivalence Partitioning can remove the number of test cases and divides a software unit into partitions of data from which test cases can be determined (Spillner *et al*. 2007). Boundary Value Analysis concerns the testing at boundary values such as minimum, maximum, just inside and outside boundaries, error values and typical values. Cause-Effect Graph creates a relation between the causes and effects (Nursimulu and Probert 1995). Fuzz Testing determines implementation bugs and uses malformed data injection in an automated session (Miller *et al*. 1990). There are also other processes of black-box testing. Regression Testing reruns some of the selected test cases to ensure that the modified software system still has the functionality as required (Ball 1998). Pattern Testing is a new type of automated testing which can verify the good application for its design or architecture and patterns (Glaser and Strauss 1967). Orthogonal Array Testing is a systematic, statistical way of software testing which can be applied in user interface testing, system testing, regression testing, configuration testing and performance testing. Matrix Testing states the status report of the project. With Manual Testing, a tester defines manual test operations to test software without the aim of test automation. Accordingly, this testing is a laborious activity that uses a tester possessing a certain set of qualities e.g., to be smart, hard working, observant, creative, speculative, innovative, open-minded, resourceful, and skillful. Automated Testing runs the program being tested, using the proper input and evaluating the output against the expectation before testing. Automated Testing needs a test suite which is generated by test case generator, no human intervention is required. Examples of Automated Testing Tools are regression testing, unit testing, automated functional testing and test management. Regression Testing refers to retesting the unchanged parts of the application. In the test suite, test cases could be re-executed in order to check whether new changes have not produced any new bugs and the previous functionality of an application is still preserved. This test can be constructed in a new build when there are significant

changes in original functionalities or even a single debugging. This article focuses on three main techniques which are described as follows. Retest-All Testing is one of the techniques for regression testing in which all the tests in the existing functionality or test suite could be re-executed. A drawback of this technique is that it is very expensive as it needs huge resources and time (Smith and Robson 1992). With Regression Test Selection, instead of retesting the entire test suite, it is better to choose some test cases to be run. One of the advantages of this technique is that reusable test cases can be used in succeeding regression cycles. Prioritization of Test Case produces scheduling over test cases in an order that improves the ability of regression testing (Elbaum *et al*. 2000). On the other hand, for Manual Testing, testers use Unit Testing/ Module Testing instead of Automated Testing. A unit is the smallest testable component in the software system. Unit Testing is introduced to verify that the lowest independent function in the software is working fine. The testable unit is tested to determine whether it works correctly when isolated from the remainding code (Rothermel and Kinneer 2004). In addition, Integration Testing can: provide the intended functionality when modules are built to interact with each other; and validate such modules. While individual classes can be invented and tested correctly, bugs may occur due to their faulty interaction. Unit Testing concerns Integration Testing which involves the consideration of: the states of multiple modules concerned at the same time in an interaction; and the state of a single object of a module (Meyers 1979). The last strategy is grey box testing, it combines the concepts of white-box and black-box testing to test an application having partial knowledge of the internal structure and at the same time having knowledge of fundamental aspects of the system (Barnett *et al*. 2003).

2.3. Integration Test Approaches

A mixture of the following approaches can be applied to develop the integration test plan. Big-Bang Integration Approach is a type of integration testing in which software elements and hardware elements are combined all at once rather than in stages. In this approach, individual modules of the programs are not integrated until determination of modules is made. This approach is not suitable mostly for inexperienced programmers who rely on running tests (Sage and Palmer 1990). With Bottom-Up Integration Approach, each subclass is invented and tested separately and after this the entire program is tested. A subclass may consist of many modules through well-defined interfaces. The primary objective of this approach is that each subclass needs to test the interfaces among different modules forming the subsystem. The test cases must be carefully selected to exercise the interfaces in all possible manners. Top-Down Integration Approach is an incremental integration testing that begins by testing the top level module. After this, it adds in lower level modules one by one. It uses stubs to simulate lower level modules. A Stub is a special code arrangement that can stimulate the behavior of a well-designed and existing module which is not yet constructed or developed (Cleve and Zeller 2005). Mixed Integration Approach (sandwiched approach or hybrid approach) combines top-down and bottom-up testing approaches. In this approach, drivers (calling programs) and stubs (called programs) are used by testers wherever the programs are incomplete.

2.4 Efficiency Testing

Testers concern performance testing which provides information about their application regarding stability, speed, and scalability to stakeholders. The performance testing will find whether or not their software meets stability, speed, and scalability requirements. Next, they concern recovery testing which is defined as how well the system should recover from system crashes, hardware failures, and other catastrophic problems. Many computer-based systems must recover from bugs and resume operation within a pre-specified time. A system can be fault tolerant, which means that processing faults must not cause the overall system function to cease. However, a system failure needs to be corrected within a specified period. The program is verified under test stores of data files in the correct directories. Accordingly, sufficient space is handled and unexpected

terminations resulting from lack of storage are avoided. This is external storage as opposed to internal storage. After this, procedure testing is used to provide detailed instructions for executing one or more test cases. The last testing, User Acceptance Testing (UAT) is the formal testing proposed to find whether a software system meets its acceptance criteria. It helps the buyer to find whether to reject the system or not. Moreover, acceptance testing is proposed to find whether software is suitable for use or not. Beside this, it involves the factors related to business environment. Alpha Testing is conducted when any type of new software or version of old software (called alpha version) is released. In addition, alpha versions are tested by some specific groups of users, those who are chosen by the software developer. They test and check whether all the features that are provided work properly or not. Beta Testing is conducted after alpha testing. Versions of the software refer to beta versions that are released to limited groups of people. It can ensure the product has a few bugs or faults. Sometimes, beta versions are produced to the public to increase the feedback area and to increase the number of future users.

Unfortunately, software testing is one of the most difficult processes in SDLC. Testers can test software with different testing techniques. Therefore, this article proposes software testing phases as an alternative option for testers.

3. Design of Software Testing Phases

This section is the result of studying the theory of software testing. Software testing can be divided into three main phases: preliminary testing, testing and user acceptance testing.

3.1 Preliminary Testing Phase

Preliminary testing phase is conducted especially for testers to clarify the specification requirements of the customer. According to this, software testing can fulfill the needs of both testers and customer. Fig. 1 describes the preliminary testing phase as follows.

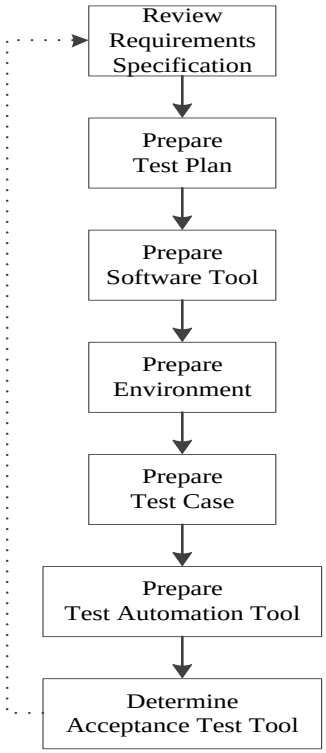

Fig.1. Preliminary testing.

Review Requirements Specification: This step is important because many customers initially provide insufficient information (e.g., size and cost of the project). Therefore, the basic document that is needed is called a requirements specification. It is a description of what the system should do.

Prepare Test Plan: This step provides a document describing the resources, schedule of testing activities, scope, and testing approach (e.g., items, features, tasks, environment, skills of tester independence, design techniques criteria, and risks)

Prepare Software Tool: This step must be set up when the different modules in the product, hardware, and operating system are known.

Prepare Test Environment: This step identifies production environment elements that must be tested and creates a test environment relevant to these elements (e.g., business applications, administrative tools, computer hardware, databases, services, security system, applications, and network systems).

Prepare Test Case: This step provides a document that explains an input and an expected response of an application feature. A

test case must contain particulars such as name of test case, input data requirements, test case identifier, objective, test setup, steps and expected output.

Prepare Test Automation Tool: This step provides planning of a test technique on how to automate software testing. For instance, test cases are executed for regression testing.

Determine Acceptance Test Tool: This step can provide acceptance test tool for software testing to meet the requirements specification.

3.2. Testing Phase

The testing phase is a separate phase which is conducted by a different test team after the implementation is completed. The testing technique is selected based on the perspective of the test team. In this article, the testing phase is divided into three steps as shown in Fig. 2: independent verification, independent validation and testing.

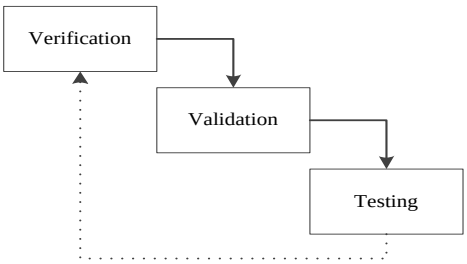

Fig. 2. Testing phase.

Verification: The verification activities include technical reviews, software inspections, walkthroughs, and efforts to: check the software requirements (e.g., they are traceable to the requirement specifications), check the design components (they are traceable to the software), check conduct requirements, perform unit testing, perform integration testing, perform system testing, check the acceptance testing, and audit. One can say that it is to determine the right thing, which concerns the testing of the implementation of right process, e.g., to determine whether the software is properly made.

Validation: It checks whether the developed software adheres to the user requirements.

Testing: It is a useful technique for both verification and validation. Obviously, other techniques useful for verification are: static analysis, reviews, inspections and walkthroughs. Other techniques useful for validation are prototyping and early release.

3.3 User Acceptance Testing (UAT) Phase

It would be most important to complete the UAT as shown in Fig. 3 to ensure that the system, which is to be implemented, is working correctly.

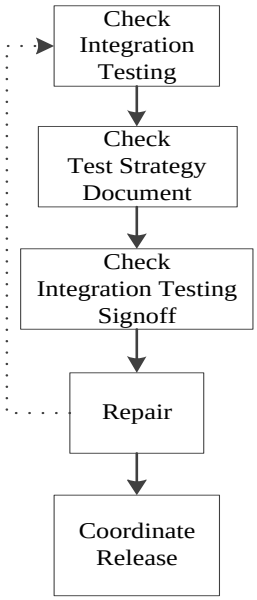

Fig. 3. User acceptance testing.

Check Integration Testing: When placing several units together, one has to conduct integration testing to ensure that the unit integration has not produced any errors.

Check Test Strategy Document: This document is prepared at the planning stage when the user requirements are determined. The strategy document is communicated to the test team for testing the software release.

Check Integration Testing Signoff: This step covers the acceptance for the whole of the system testing. An agreement for the defects is reached. This is a form, signed by the project manager, which indicates that the documentation, system testing, and training materials have satisfied all tests within acceptable margins.

Repair: This step is conducted to have stable performance based on technical and functional specifications.

Coordinate Release: This step is the last step before the release of new software to the market.

The three phases are designed to describe all activities when dealing with software

testing. According to this, software testers can take the benefit of the design to implement the ideas of software testing and address the critical problems described in Section 2.

4. Conclusion

In summary, software testing is not only the process of a team tester to: determine the bugs and report the bugs to software developers; and fix the bugs to develop new quality software. The critical problems of testing must also be considered, including the structure of source codes and the whole system. However, there is no one to guarantee the use of best methods in software testing. Therefore, many researchers are still working on it. In future research, the focus will be on software maintenance. This is because software testing may result in high costs before feeding into the maintenance system.

5. References

Ball, T. 1998. On the limit of control flow analysis for regression test selection. Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA), Clearwater Beach, FL, USA, 2-5 March 1998, pp. 134-42.

Barnett, M.; Grieskamp, W.; Kerer, C.; Schulte, W.; Szyperski, C.; Tilmann, N.; and Watson, A. 2003. Serious specification for composing components. Proc. 6th ICSE Workshop on Component-based Software Engineering: Automated Reasoning and Prediction, Portland, OR, USA, 3-4 May 2003. 6 pages.

Clarke, L.A.; Podgurski, A.; Richardson, D.J.; and Zeil, S.J. 1989. A formal evaluation of data flow path selection criteria. IEEE Trans. Software Eng. 15(11): 1,318-32.

Cleve, H.; and Zeller, A. 2005. Locating causes of program failures. Proc. 27th ACM Int. Conf. Software Eng. (ICSE), St. Louis, MO, USA, 15-21 May 2005, pp. 342-51.

Elbaum, S.; Malishevsky, A.G.; and Rothermel, G. 2000. Prioritizing test cases for regression testing. Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA), Portland, OR, USA, 22-25 August 2000, pp. 102-12.

Fischer, K.F. 1977. A test case selection method for the validation of software maintenance modifications. Proc. IEEE Int. Computer Software and Application Conference (COMPSAC), Chicago, IL, USA, 8-11 November 1977, pp. 421-6.

Glaser, B.G.; and Strauss A.L. 1967. The Discovery of Grounded Theory: Strategies for Qualitative Research. Aldine, Chicago, IL, USA.

Hetzel, W.C. 1988. The Complete Guide to Software Testing. 2nd ed. QED Information Sciences, Inc., Wellesley, MA, USA.

Horgan, J.R.; and London, S. 1991. Data flow coverage and the C language. Proc. 4th ACM Symp. on Testing, Analysis, and Verification (TAV 4), Victoria, BC, Canada, 8-9 October 1991, pp. 87-97.

Humphrey, W. S. 1995. A Discipline for Software Engineering. Addison Wesley, New York, NY, USA.

Jorgensen, P.C. 2002. Software Testing: A Craftsman's Approach. 2nd ed. CRC Press, New York, NY, USA. Chapter 6.

Miller, B.P.; Fredriksen, L.; and Bryan, S. 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM, 33(12): 32-44.

Myers, G.J. 1979. The Art of Software Testing. John Wiley & Sons, New York, NY, USA.

Nursimulu, K.; and Probert, R.L. 1995. Cause-effect graphing analysis and validation of requirements. Proc. Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON), IBM Press, Toronto, Ontario, Canada, 7-9 November 1995. p. 46.

Pan, J. 1999. Software testing. Student Report. Available:<http://www.ece.cmu.edu/~koopman/des_s99/sw_testing>.

Rapps, S.; and Weyuker, E.J. 1985. Selecting software test data using data flow information. IEEE Trans. Software Eng. 11(4): 367-75.

Sage, A.P.; and Palmer, J.D. 1990. Software Systems Engineering. John Wiley & Sons, New York, NY, USA.

Spillner, A.; Linz T.; and Schaefer, H. 2007. Software Testing Foundations. A Study Guide for the Certified Tester Exam. Foundation Level, ISTQB compliant. Rocky Nook Inc., Santa Barbara, CA, USA.