International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 4 Issue 3, April 2020 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Introduction to Software Testing

Durgesh Raghuvanshi

B Tech, Department of Computer Science,
IILM Academy of Higher Learning, Greater Noida, Uttar Pradesh, India

ABSTRACT

In this era of computing, the ideas and techniques of software testing have become essential knowledge for all software developers. A software developer can expect to use the concepts presented in this book many times during his or her career. This chapter introduces the subject of software testing by describing the activities of a test engineer, defining a number of key terms, and then explaining the central notion of test coverage. Software is a key ingredient in many of the devices and systems that pervade our society. Software defines the behavior of network routers, financial networks, telephone switching networks, the Web, and other infrastructure of modern life. Software is an essential component of embedded applications that control exotic applications such as airplanes, spaceships, and air traffic control systems, as well as mundane appliances. This paper provides an introduction to fundamental concepts of software testing and software product should only be released after it has gone through a proper process of development, testing and bug fixing. Testing looks at areas such as performance, stability and error handling by setting up test scenarios under controlled conditions and assessing the results. This is why exactly any software has to be tested. It is important to note that software is mainly tested to see that it meets the customers' needs and that it conforms to the standards. It is the usual norm that software is considered of good quality if it meets the user requirements.

KEYWORDS: *automation, test engineer, software activity, infeasibility, subsumption, bugs, etc*

How to cite this paper: Durgesh Raghuvanshi "Introduction to Software Testing" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-4 | Issue-3, April 2020, pp.797-800, URL: www.ijtsrd.com/papers/ijtsrd25102.pdf

IJTSRD25102

INTRODUCTION

Software is a series of instructions for the computer that perform a particular task, called a program; the two major categories of software are system software and application software. The system software is made up of control programs. Application software is any program that processes data for the user (spreadsheet, word processor, payroll, etc.).

A software product should only be released after it has gone through a proper process of development, testing and bug fixing. Testing looks at areas such as performance, stability and error handling by setting up test scenarios under controlled conditions and assessing the results. This is why exactly any software has to be tested. It is important to note that software is mainly tested to see that it meets the customers' needs and that it conforms to the standards. It is the usual norm that software is considered of good quality if it meets the user requirements. All software problems can be termed as bugs. A software bug usually occurs when the software does not do what it is intended to do or does something that it is not intended to do. Flaws in specifications, design, code or other reasons can cause these bugs. Identifying and fixing bugs in the early stages of the software is very important as the cost of fixing bugs grows over time. So, the goal of a software tester is to find bugs and find them as early as possible and make sure they are fixed.

Testing is context-based and risk-driven. It requires a methodical and disciplined approach to finding bugs. A good software tester needs to build credibility and possess the attitude to be explorative, troubleshooting, relentless, creative, diplomatic and persuasive.

As against the perception that testing starts only after the completion of the coding phase, it actually begins even before the first line of code can be written. In the life cycle of the conventional software product, testing begins at the stage when the specifications are written, i.e. from testing the product specifications or product spec. Finding bugs at this stage can save huge amounts of time and money.

Once the specifications are well understood, you are required to design and execute the test cases. Selecting the appropriate technique that reduces the number of tests that cover a feature is one of the most important things that you need to take into consideration while designing these test cases. Test cases need to be designed to cover all aspects of the software, i.e. security, database, functionality (critical and general) and the user interface. Bugs originate when the test cases are executed. As a tester you might have to perform testing under different circumstances, i.e. the application could be in the initial stages or undergoing rapid changes, you have less than enough time to test, the product might be developed using a life cycle model that does not support much of formal testing or retesting. Further, testing using different operating systems, browsers and the configurations are to be taken care of.

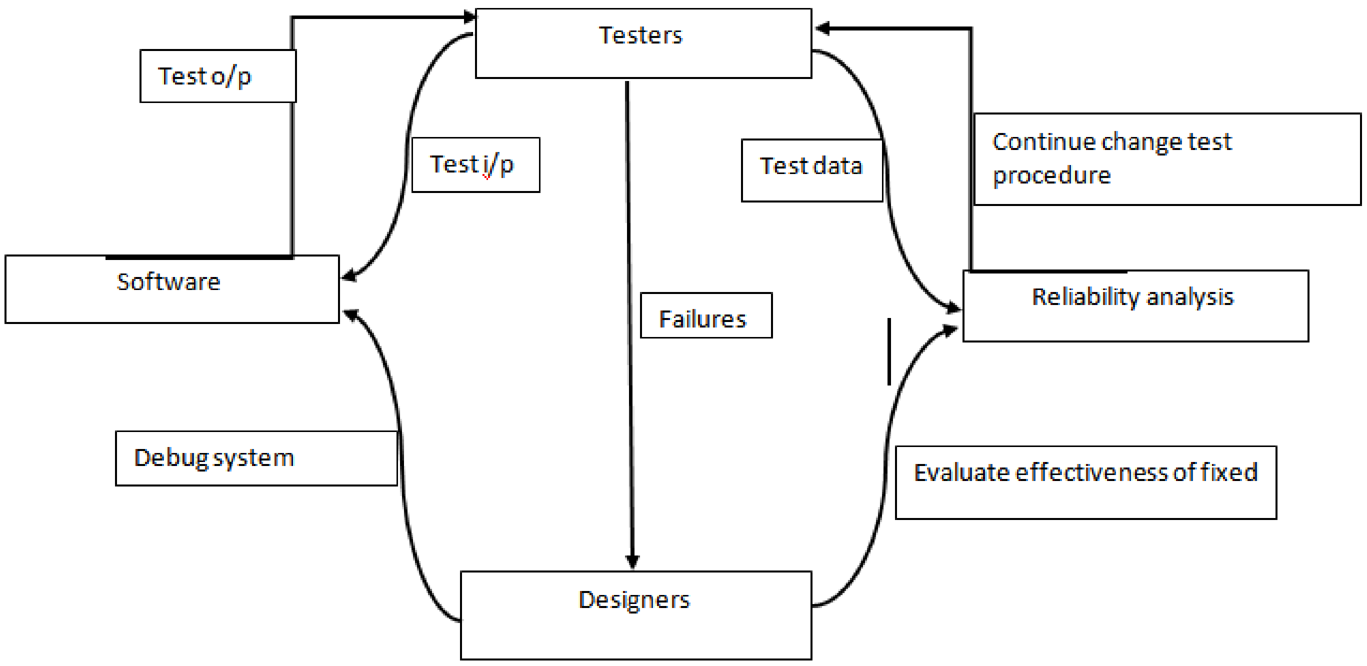
Reporting a bug may be the most important and sometimes the most difficult task that you as a software tester will perform. By using various tools and clearly communicating to the developer, you can ensure that the bugs you find are fixed.

Using automated tools to execute tests, run scripts and tracking bugs improves the efficiency and effectiveness of your tests. Also, keeping pace with the latest developments in the field will augment your career as a software test engineer.

Objectives and important terms of Software Testing

The objective of testing is to discover the residual design errors before delivering to the customer.

- Bug: A software bug may be defined as a coding error that causes an unexpected defect, fault or flaw. In other words, if a program does not perform as intended, it is most likely a bug.
- Error: A mismatch between the program and its specification is an error in the program.
- Defect: Defect is the variance from the desired product attribute (it can be a wrong, missing or extra data). It can be of two types – Defect from the product or variance from customer/user expectations. It is a flaw in the software system and has no impact until it affects the user/customer and operational system. 90% of all the defects can be caused by process problems.
- Failure: A defect that causes an error in operation or negatively impacts a user/ customer.
- Quality Assurance: Is oriented towards preventing defects. Quality Assurance ensures all parties concerned with the project adhere to the process and procedures, standards and templates and test readiness reviews.
- Quality Control: quality control or quality engineering is a set of measures taken to ensure that defective products or services are not produced and that the design meets performance requirements.
- Verification: Verification ensures the product is designed to deliver all functionality to the customer; it typically involves reviews and meetings to evaluate documents, plans, code, requirement, and specifications; this can be done with checklists, issues lists, walkthrough, and inspection meetings.
- Validation: Validation ensures that functionality, as defined in requirements, is the intended behavior of the product; validation typically involves actual testing and takes place after verifications are completed.
- Software Reliability Estimation: The objective of testing is to discover the residual design errors before delivering to the customer. The failure data during the testing process are taken in down to estimate the software reliability. The testing process may function with regular feedback from the reliability analysis to the testers and designers.



Most common errors in the software

- User Interface Errors: Missing/Wrong Functions Doesn't do what the user expects, Missing information, Misleading, Confusing information, Wrong content in Help text, Inappropriate error messages. Performance issues - Poor responsiveness, Can't redirect output, Inappropriate use of keyboard
- Error Handling: Inadequate - protection against corrupted data, tests of user input, version control; Ignores – overflow, data comparison, Error recovery – aborting errors, recovery from hardware problems.
- Boundary related errors: Boundaries in loop, space, time, memory, mishandling of cases outside the boundary.
- Calculation errors: Bad Logic, Bad Arithmetic, Outdated constants, Calculation errors, incorrect conversion from one data representation to another, Wrong formula, incorrect approximation.
- Initial and Later states: Failure to - set data item to zero, to initialize a loop control variable, or re-initialize a pointer, to clear a string or flag, Incorrect initialization.
- Control flow errors: Wrong returning state assumed, Exception handling based exits, Stack underflow/overflow, Failure to block or unblock interrupts, Comparison sometimes yields the wrong result, Missing/wrong default, and Data Type errors.

- Errors in Handling or Interpreting Data: Un-terminated null strings, overwriting a file after an error exit or user abort.
- Race Conditions: Assumption that one event or task finished before another begins, Resource races, Tasks starts before its prerequisites are met, Messages cross or don't arrive in the order sent.
- Load Conditions: Required resources are not available, No available large memory area, Low priority tasks not put off, doesn't erase old files from mass storage, and doesn't return unused memory.
- Hardware: Wrong Device, Device unavailable, Underutilizing device intelligence, Misunderstood status or return code, Wrong operation or instruction codes.
- Source, Version and ID Control: No Title or version ID, Failure to update multiple copies of data or program files.
- Testing Errors: Failure to notice/report a problem, Failure to use the most promising test case, corrupted data files, Misinterpreted specifications or documentation, Failure to make it clear how to reproduce the problem, Failure to check for unresolved problems just before release, Failure to verify fixes, Failure to provide summary report.

Testing levels based on software activity
Tests can be derived from requirements and specifications, design artifacts, or the source code. A different level of testing accompanies each distinct software development activity: Acceptance Testing – assess software with respect to requirements. System Testing – assess software with respect to architectural design. Integration Testing – assess software with respect to subsystem design. Module Testing – assess software with respect to detailed design. Unit Testing – assess software with respect to implementation. A typical scenario for testing levels and how they relate to software development activities by isolating each step. Information for each test level is typically derived from the associated development activity. Indeed, the standard advice is to design the tests concurrently with each development activity, even though the software will not be in an executable form until the implementation phase. The reason for this advice is that the mere process of explicitly articulating tests can identify defects in design decisions that otherwise appear reasonable. Early identification of defects is by far the best means of reducing their ultimate cost. Note that this diagram is not intended to imply a waterfall process. The synthesis and analysis activities generically apply to any development process. The requirements analysis phase of software development captures the customer's needs. Acceptance testing is designed to determine whether the completed software, in fact, meets these needs. In other words, acceptance testing probes whether the software does what the users want. Acceptance testing must involve users or other individuals who have strong domain knowledge. The architectural design phase of software development chooses components and connectors that together realize a system whose specification is intended to meet the previously identified requirements System testing is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole. This level of testing usually looks for design and specification problems. It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team. The subsystem design phase of software development specifies the structure and behavior of subsystems, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of the previously developed software. Integration testing is designed to assess whether the interfaces between modules(defined below)in a given subsystem have consistent assumptions and communicate correctly. Integration testing must assume that modules work correctly. Some testing literature uses the terms integration testing and system testing interchangeably; in this book, integration testing does not refer to testing the integrated system or subsystem. Integration testing is usually the responsibility of members of the development team.

Automation of Test Activities
Software testing is expensive and labor intensive. Software testing requires up to 50% of software development costs, and even more for safety-critical applications. One of the goals of software testing is to automate as much as possible thereby significantly reducing its cost, minimizing human error, and making regression testing easier. Software engineers sometimes distinguish revenue tasks, which contribute directly to the solution of a problem, from excise tasks, which do not. For example, compiling a Java class is a classic excise task because, although necessary for the class to become executable, compilation contributes nothing to the particular behavior of that class. In contrast, determining which methods are appropriate to define a given data abstraction as a Java class is a revenue task. Excise tasks are candidates for automation; revenue tasks are not. Software testing probably has more excise tasks than any other aspect of software development. Maintaining test scripts, rerunning tests, and comparing expected results with actual results are all common excise tasks that routinely consume large chunks of test engineer's time. Automating excise task serves the test engineer in many ways First eliminating excise tasks eliminate drudgery, thereby making the test engineers job more satisfying. Second, automation free suptimeto focus on the fun and challenging parts of testing, namely the revenue tasks. Third, automation can help eliminate errors of omission, such as failing to update all the relevant files with the new set of expected results. Fourth, automation eliminates some of the variances in test quality caused by differences in individual's abilities. Many testing tasks that defied automation in the pastare now candidates for such treatment due to advances in technology. For example, generating test cases that satisfy the given test requirement is typically a hard problem that requires intervention from the test engineer. However, there are tools, both research, and commercial, that automate this task to varying degrees.

Software testing limitations and terminology
One of the most important limitations of software testing is that testing can show only the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable. Testers often call a successful (or effective) test one that finds an error. While this is an example of level 2 thinking, it is also a characterization that is often useful and that we will use later in this book. There to f this section presents a number of terms that are importations of ware testing and that will be used later in this book. Most of these are taken from standards

documents, and although the phrasing is ours, we try to be consistent with the standards. Useful standards for reading in more detail are the IEEE Standard Glossary of Software Engineering Terminology, DOD-STD-2167A andMIL-STD-498 from the US Department of Defense, and the British Computer Society's Standard for Software Component Testing. One of the most important distinctions to make is between validation and verification.

- Validation: The process of evaluating software at the end of software development to ensure compliance with intended usage.
- Verification: The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase.

Verification is usually a more technical activity that uses knowledge about individual software artifacts, requirements, and specifications. Validation usually depends on domain knowledge; that is, knowledge of the application for which the software is written. For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots.

CONCLUSION

Software testability is an important notion distinct from software testing. In general, software testability is an estimate or measurement of a conditional probability, namely, assuming that a given software artifact contains a fault, how likely is it that testing will reveal that fault. We are all familiar with software development projects where, despite extensive testing, faults continue to be found. Testability gets to the core of how easy or hard it is for faults to escape detection – even from well-chosen test suites. Testing for emergent properties presents special challenges. This section offers high-level guidance for engineers faced with testing systems where safety and/or security play an important role. Emergent properties arise as a result of collecting components together into a single system. They do not exist independently in any particular component. Safety and security are classic emergent properties in system design. For example, the overall safety of an airplane is not determined by the control software by itself, or the engines themselves, or by any other component by itself. Certainly, the individual behavior of a given component may be extremely important with respect to overall safety but, even so, the overall safety is determined by the interaction so fall of these components when assembled into a complete airplane. In other words, an airplane engine is neither safe nor unsafe considered by itself because an airplane engine doesn't fly by itself. Only complete airplanes can fly, and hence only complete airplanes can be considered safe or unsafe with respect to flying. Likewise, the security of a web application is not determined by the security of a back-end database server by itself, or by a proxy server by itself, or by the cryptographic systems used by themselves, but by the interactions of all of these components.

REFERENCE

[1] CSCI 5828: Foundations of Software Engineering Lecture 05 — 01/31/2012

[2] Pressman Robert software engineering and practitioner approach 5th edition mc graw hill,2000

[3] IEEE software, www.computer.org/software/

[4] Communications of the ACM, www.acm.org

[5] Borland cooperation: www.borland .com

[6] Royce, Walker, software project management: a unified approach, Australia