Volume 6, Issue 6, June 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Review of Testing Strategies

[1]K. Vijaya Sai Rachana, [2]K. Sai Prasad Reddy, [3]Prof. K.Nagabhushan Raj
[1] 4th Year,B.Tech(CS), Amrita Vishwa Vidyapeetam, Amritapuri, Kerala, India
[2] Research Scholar, S. K. University, Anantapur, Andhra Pradesh, India
[3] Department of Instrumentation, S.K. University, Anantapur, Andhra Pradesh, India

Abstract-- This paper reviews software Testing Strategies that are used in the area of software development. It points the narrow difference between verification and validation. It elucidates about various testing strategies for conventional software architectures and object oriented software architectures. A software testing should be flexible enough to promote a customized testing approach.

Keywords-Validation, Verification, Unit testing, Integration testing, Conventional software architecture, Object Oriented software testing.

I. INTRODUCTION

A strategy for software testing integrates software test case design methods into a well-planned series of steps that leads to the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation. A software testing strategy should be flexible enough to promote reasonable planning and management tracking as the project progresses. In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defense against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques are helping us to reduce the number of initial errors that are inherent in the code.

II. CONCEPT OF VALIDATION AND VERIFICATION

Testing is a set of activities that can be planned in advance and conducted systematically. Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirement. The minute difference between Verification and validation is:

Verification: Are we building the product right?
Validation: Are we building the right product?

III. A SOFTWARE TESTING STRATEGY FOR CONVENTIONAL SOFTWARE ARCHITECTURE

The software process may be viewed as the spiral model. Initially software requirement analysis is performed to get an idea about the information, functions, behavior, and performance of the software. Moving inward along the spiral, we come to design and then finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction in each turn.

A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit in the source code. It makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection. The next step is integration testing, where the focus is on design and the construction of the software architecture. It addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration. Taking another turn around on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed. It provides final assurance that software meets all functional, behavioral, an performance requirements. Finally, we arrive at system testing, where the software and the other system elements are tested as a whole. It verifies that all elements mesh properly and that overall system function/performance is achieved. To test software, we spiral out along streamlines that broaden the scope of testing with each turn.

A testing strategy that is chosen by most software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described below.

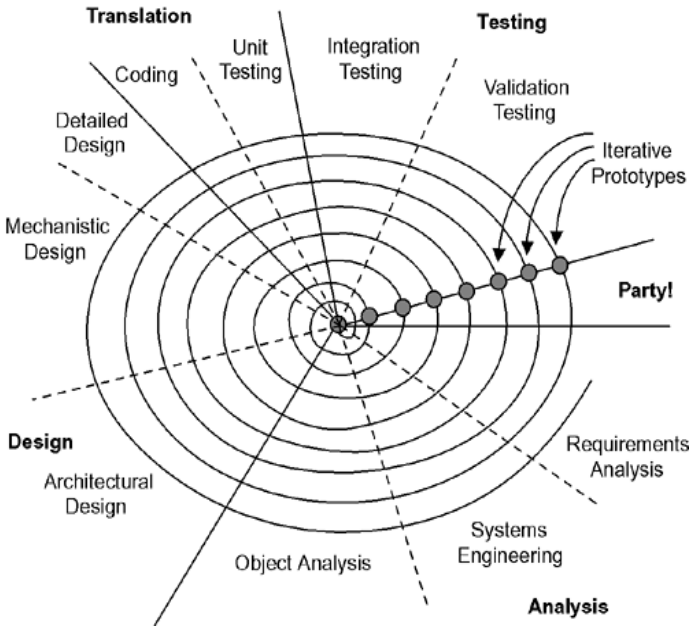

Fig. 1 Spiral model

A. Unit Testing

Unit testing focuses verification effort on the smallest unit of software design—the software components or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

i. **Unit Test Considerations**:

The module interface is tested to ensure that information properly flows into and out of the program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error handling paths are tested.

Test of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained during unit testing.

ii. **Unit Test Procedures:**

Unit testing is normally considered as an adjunct to the coding steps. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test should be coupled with a set of expected results. Because a component is not a stand-alone program, driver and/or stud software must be developed for each unit test. Driver is nothing more than a main program that accepts test cases data, passes such data to the component, and prints relevant results. Stubs serve to replace modules that are subordinate to the component to be tested. A stub or dummy subprogram uses the subordinate module's interface, may do minimal data manipulation, and provides verification of entry and return control to the module undergoing testing.

Drivers and stubs represents overhead. That is, both are software that must be written but not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with simple overhead software. In such cases, complete testing can be postponed until the integration test step(where drivers or stubs are also used).Unit testing is simplified when a component with high cohesion is designed when only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicated and uncovered.

B. Integration Testing

Integration testing is a systematic technique for considering the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration that is, all components are combined in advance. The entire program is tested as a whole. This will definitely lead to chaos. Here correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration converse the above mentioned non incremental approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. There are different incremental integration strategies discussed below:

i. **Top Down Integration:**

Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module. Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

This integration process is performed in five steps:

1) The main module is used as attest driver, and stubs are substituted for all components directly subordinate to the main control module.
2) Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3) Tests are conducted as each component is integrated.
4) On completion of each set of tests, another stub is replaced with the real component.
5) Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues until the entire program is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first.

But in cases, top-down approach may fail. So, there is an approach called bottom-up testing is required.

ii. **Bottom up Integration:**

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules. Because components are integrated from bottom up, processing required for the components subordinates to a given level is always available and the need for stubs is eliminated. Bottom-up integration strategy may be implemented with the following steps:

a) Low-level components are combined into clusters that perform a specific software sub function.
b) A driver is written to coordinate test case input and output.
c) The cluster is tested.
d) Drivers are removed and clusters are combined moving upwards in the program structure.

As integration moves upwards, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

C. Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously work flawlessly. In the context of an integration test strategy, regressive testing is the re-execution of some subset of tests that have been conducted to ensure that changes have not propagated unintended side effects.

Regressive testing may be conducted manually, by re-executing a subset of all test cases or using automated capture tools. Capture tools enable the software engineer to capture test cases and results for subsequent playback and comparison. The regression test suite contains three different classes of test cases.

D. Smoke Testing

Smoke testing is an integration testing approach that is commonly used when software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to access its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

a) Software components that have been translated into code are integrated into a "build". A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
b) A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stoppers" errors that have the highest likelihood of throwing the software project behind schedule.
c) The build is integrated with other builds and the entire product is smoke tested daily. The integration may be top down or bottom up.

IV. A SOFTWARE TESTING STRATEGY FOR OBJECT ORIENTED SOFTWARE ARCHITECTUE

The testing of Object-oriented system presents a set of different challenges. The definition of testing should be broadened to include error detection techniques that are applied to analysis and design models. The completeness and consistency of object-oriented representation must be assessed as they are built. Here unit testing loses some of its meaning and integration strategies changes significantly. In summary both testing strategies and testing tactics must account for the unique characteristics of object oriented software. The overall strategy for object oriented software is identical in philosophy to the one applied for conventional architectures, but differs in approach. We begin with testing in the small and work outward towards testing in the large. As classes are integrated into an object-oriented architecture, a series of regression tests are run to uncover errors due to communication and collaboration between classes and side effects caused by the addition of new classes. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

A. Unit Testing

When Object-oriented software is considered, the concept of the unit testing changes. Encapsulation drives the definition of classes. This means that each class and each instance of a class packages attributes and the operations that manipulate these data. An encapsulated class is usually the focus of unit testing. However, operations within the class are the smallest testable units. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change.

B. Integration Testing

Because object orient software does not have an obvious hierarchical control structure, traditional top down and bottom up integration strategies have little meaning. In addition, integrating operations one at a time into a class is often impossible because of the direct and indirect interactions of the components that make up the class. Therefore, there arise two different integration testing strategies. The first thread-based testing integrates the set of classes required to respond to one input or event of the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, use-based testing begins the construction of the system by testing those classes that use very few server classes. After the independent classes are tested, the next layer of classes, called dependent classes, which uses the independent classes, are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

V. VALIDATION TESTING

Validation testing begins at the culmination of integration testing, when individual components have been exercised, the software is completely assembled as a package, and interfacing errors have been uncovered and corrected. At the validation or system level, the distinction between conventional and object-oriented software disappears. Testing focuses on user-visible actions and user-recognizable output from the system.

An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle.

A. Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; Output that seemed clear to the tester may be unintelligible to a user in the field.

If the software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

The alpha test is conducted at the developer's site by end-users. The software is used in a natural setting with the developer "looking over the shoulder" of typical users and recording errors and usage problems. Alpha test are conducted in a controlled environment.

The beta test is conducted at end-users sites. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The end-users records all problems that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

VI. SYSTEM TESTING

System testing is actually a series of series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. There are different types of system tests. They are:

A. Recovery Testing

Recovery testing is a system test that forces the software to fall in a variety of ways and verifies that recovery is properly performed. If recovery is automatic, reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.

B. Security Testing

Security testing verifies that protection mechanism built into a system will, in fact, protect it from improper penetration.

C. Stress Testing

Stress tests are designed to confront programs with abnormal situations. Example: Special test case may be designed that generate ten interrupts per second, when one or two is average.

D. Performance Testing

Performance testing is designed to test the run-time performance of software within the context of an integrated system. It occurs throughout all steps in the testing process. Performance tests are often coupled with stress testing and

usually requires both hardware and software instrumentation. That is, it is often necessary to measure resource utilization in an extracting fashion.

VII. CONCLUSION

There are many software testing methods, which are different for different types of architecture. This paper provides an overview of few of these testings. No test is superior to any other and these tests are used depending upon the requirements and prevailing conditions. Recent time has shown a shift from traditional models to contemporary models of software development. Selecting the correct testing models can help to deliver the required software product within deadline, meeting quality and cost constraints. Further, more changes and evolutions are continuously being performed to increase the quality of software being developed by improving the methodologies used.

REFERENCES

[1] Frankl P. and Weyuker E. A formal analysis of the fault-detecting ability of testing methods. Software Engineering, IEEE Transactions. 1993.

[2] Chen T. and Yu Y. On the expected number of failures detected by sub domain testing and random testing Software Engineering, IEEE Transactions. 1996

[3] Frankl P. Hamlet D. Littelewood B. and Strigini L., Choosing a testing method to deliver reliability. 19th International conference on Software Engineering, ACM, 1997

[4] Frankl P. Hamlet D. Littelewood B. and Strigini L., Evaluating testing methods by delivered reliability [software]. Software Engineering, IEEE, 1998

[5] Pizza M. and Strigini L. Comparing the effectiveness of testing methods in improving programs: the effect of variations in program quality. In Software Reliability Engineering, 1998, Proceedings. The Ninth International Symposium, IEEE 1998.

[6] Umar Farooq, Evaluation Effectiveness of Software Testing Techniques with Emphasis on Enhancing Software Reliability, 2012

[7] Lott C. , Rombach H. Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques, Empirical Software Engineering, Spring 1997

[8] Li N. and Malaiya Y. On input profile selection for software testing. In Software Reliability Engineering, 1994, 5th International Symposium, IEEE, 1994.

[9] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing techniques. IEEE Transactions on Software Engineering, 13(12):1278–1296, December 1987.

[10] Juristo N. and Vegas S. Functional testing, structural testing and code reading: what fault type do they each detect? Empirical Methods and Studies in Software Engineering, 2003.

[11] Kamsties E. and Lott C. An empirical evaluating of three defect-detection techniques. Software Engineering ESEC‟95, 1995.

[12] Roper M. Wood M and Miller J. An empirical a valuation of defect detection techniques.

