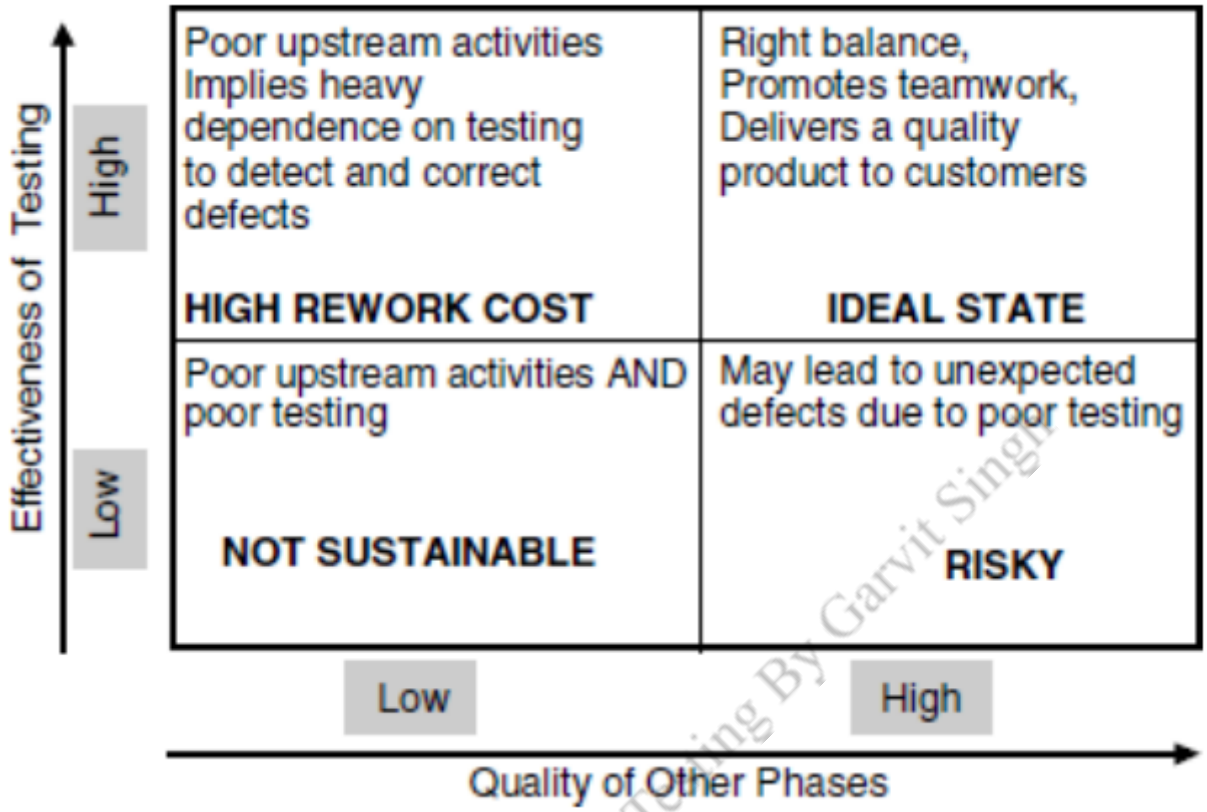
Software Testing

Principles, Practices & Techniques

By Garvit Singh

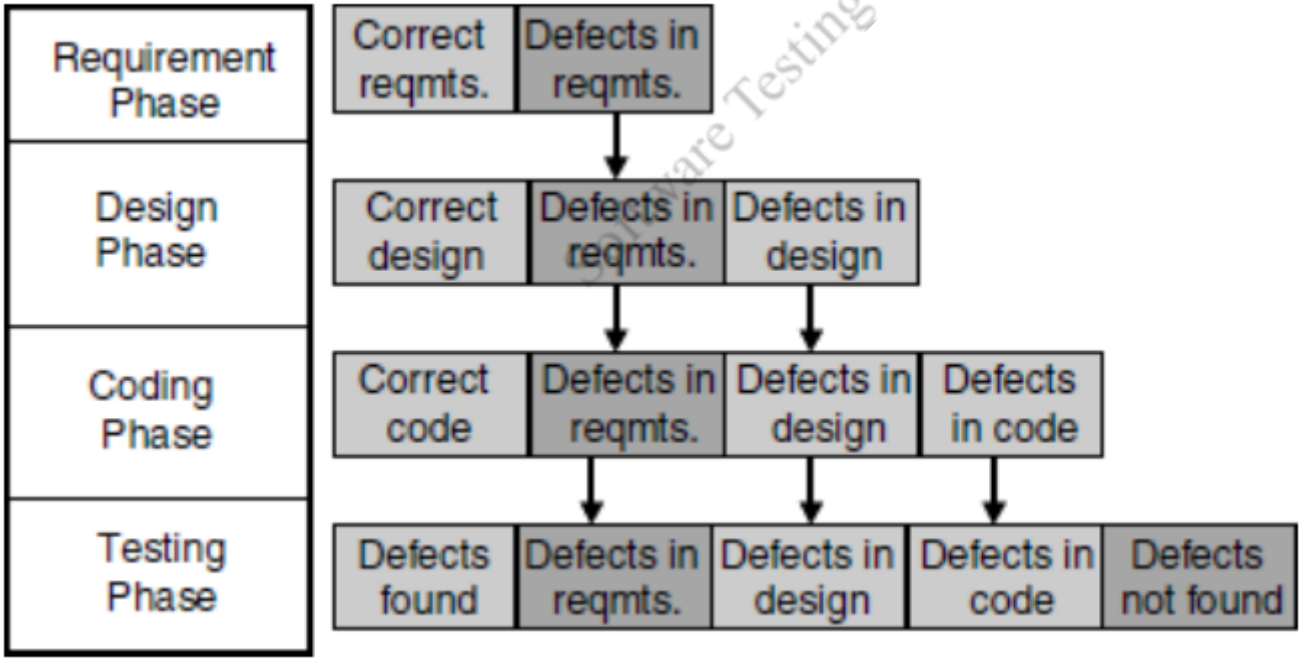
The Fundamental Principles Of Testing

1. The goal of testing is to find defects before customers find them out.
2. Exhaustive testing is not possible; program testing can only show the presence of defects, never their absence.
3. Testing applies all through the software life cycle and is not an end-of-cycle activity.



4. Understand the reason behind the test.
5. Test the tests first.
6. Tests develop immunity and have to be revised constantly.
7. Defects occur in convoys or clusters, and testing should focus on these convoys.

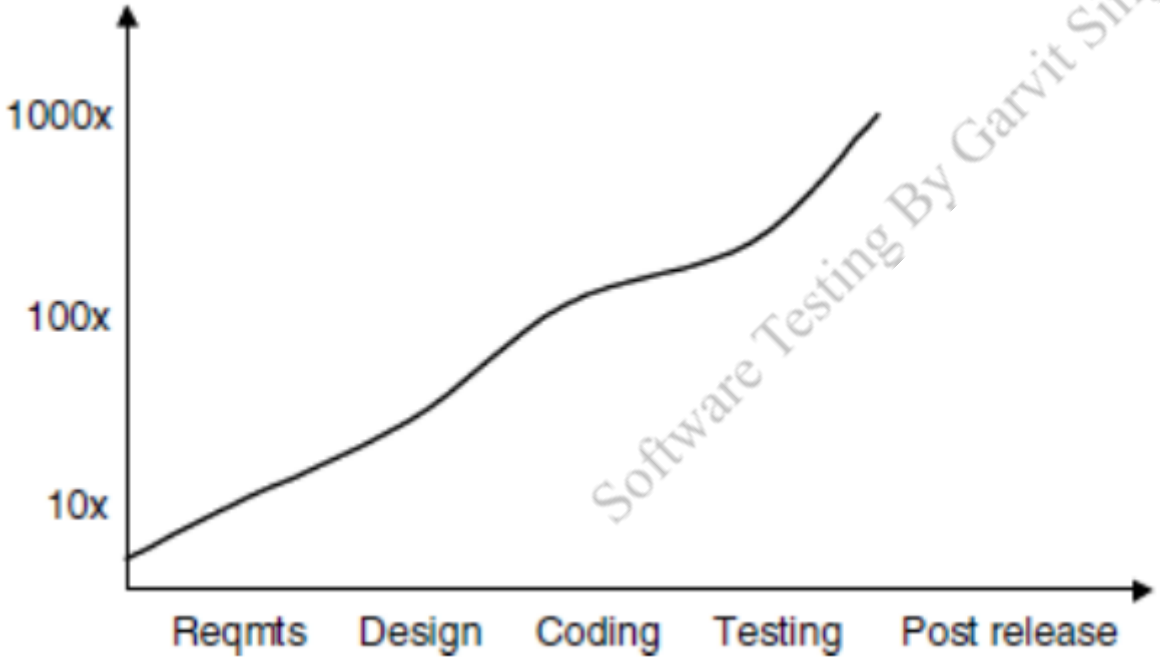
8. Testing encompasses defect prevention.
9. Testing is a fine balance of defect prevention and defect detection.
10. Intelligent and well-planned automation is key to realizing the benefits of testing.
11. Testing requires talented, committed people who believe in themselves and work in teams.



12. The cost of building a product and the number of defects in it increase steeply with the number of defects allowed to seep into the later phases of software development.
13. Costs due to defects in software have a compounding effect. The cost to fix them increases exponentially as they progress from requirements phase, through the design phase, coding phase, testing phase and post release phase. The factor can be in thousands. This makes identifying defects in an early phase crucial for a product's profitable operations.
14. A defective test is more dangerous than a defective product.
15. We can choose to execute only a subset of tests that cover a maximum number of possible errors. This is due to the limited amount of time to test the product. We can never be 100% sure that there are no defects left out.
16. Smaller the lag time between defect injection and defect detection, lesser are the unnecessary costs. Thus, it becomes

essential to catch the defects as early as possible.

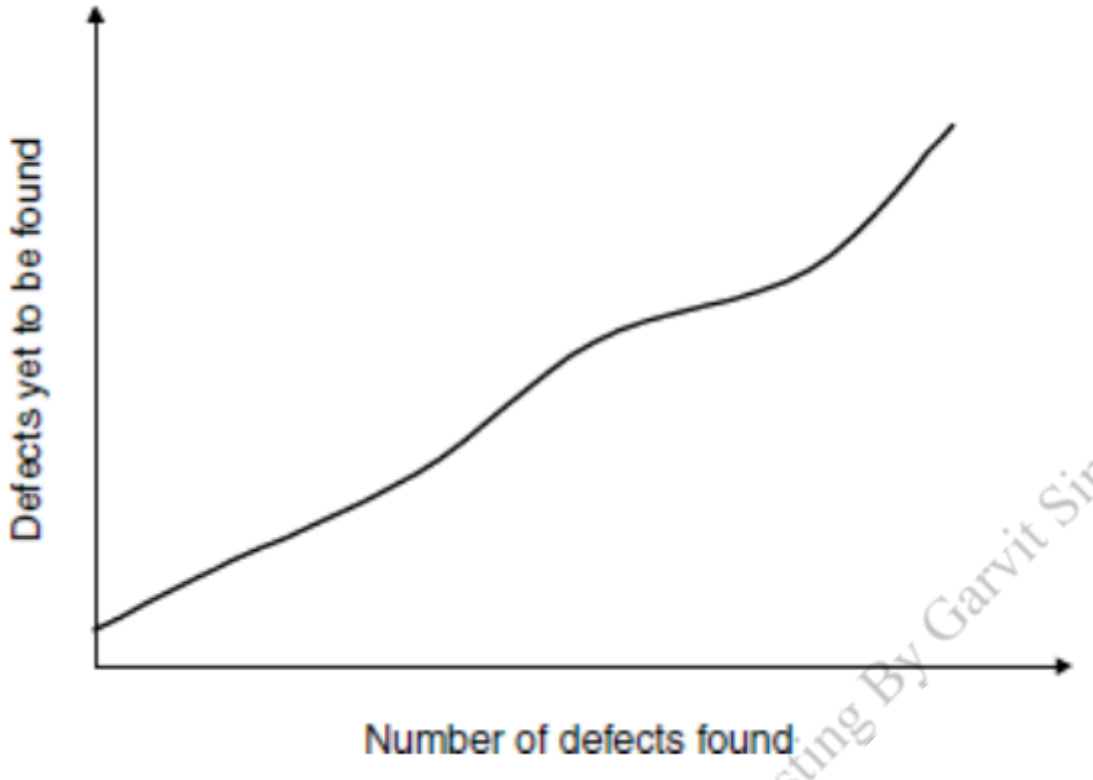
17. A defect introduced during the requirements phase that makes it to the final release may cost as much as a thousand times the cost of detecting and correcting the defect during requirements gathering itself.



18. Testing requires asking about and understanding what you are trying to test, know what the correct outcome is, and why you

are performing any test. If we carry out tests without understanding why we are running them, we'll end up running inappropriate tests that do not address what the product should do.

19. It may even turn out that the product has been modified to make sure that tests are run successfully, even if the product doesn't meet the intended purpose. This is a recipe for disaster after the final release.
20. Defects develop immunity against test cases. As and when we write new test cases and uncover new defects in the product, other defects that were 'hiding' underneath show up. Tests have to be constantly revised to tackle new defects.
21. The probability of existence of more errors in a section of program is proportional to the number of errors already found in that section. The number of defects yet to be found increases with the number of defects uncovered.

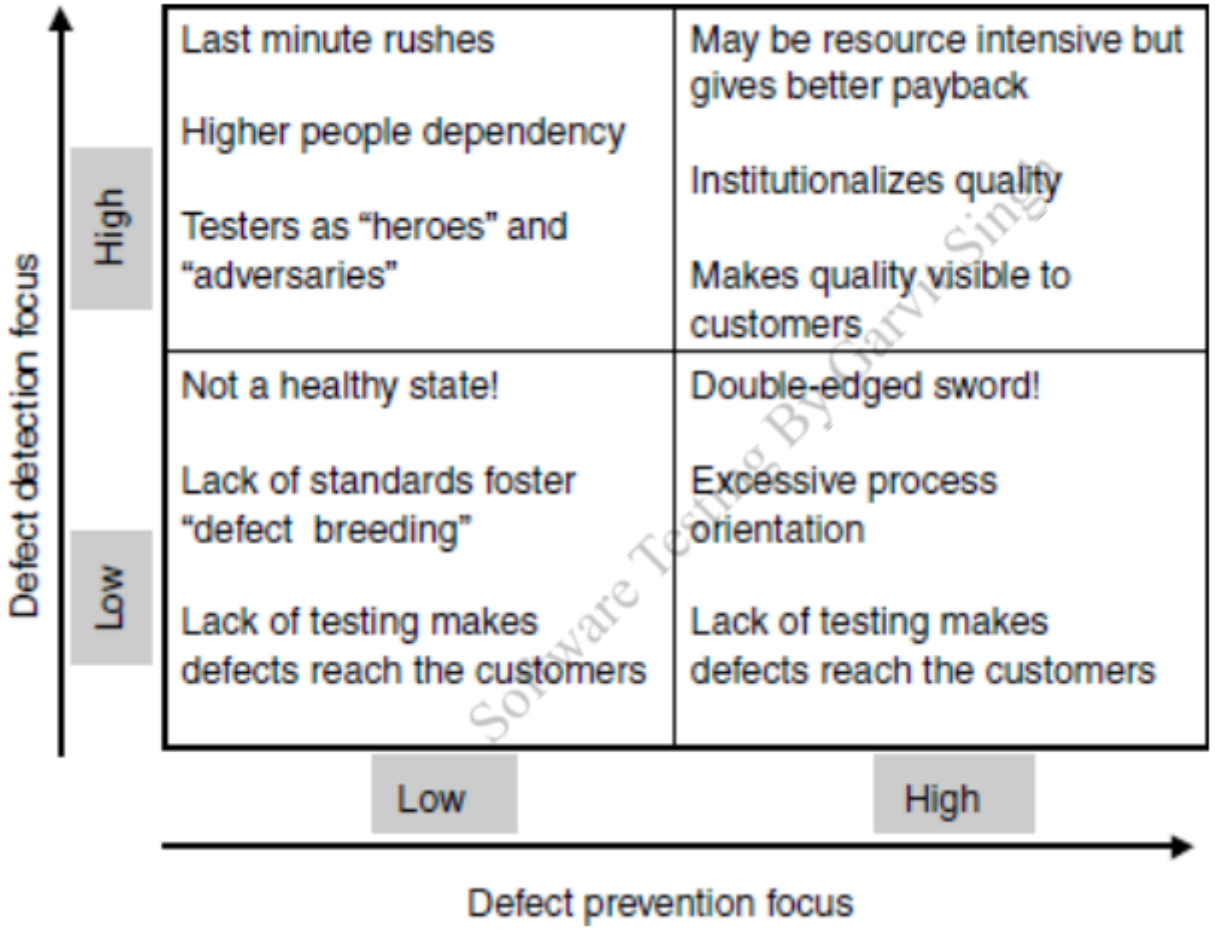


22. Testing can only find a part of defects that exist in a cluster. Fixing a defect may introduce another defect to the cluster. A fix for one defect generally introduces some instability and necessitates another fix. All these fixes produce side effects that eventually cause a convoy of defects in certain parts of product.

The long term solution in such a case is to re-architecture the design and rewrite the code.

23. Testers have to work with development engineers to make sure that the root cause of the defects are addressed. Defect prevention is a part of tester's job. They have to strike a balance between defect prevention and defect detection.
24. Defect detection and correction is called Quality Control, while defect prevention is called Quality Assurance. These are two methods to achieve quality. Testing is traditionally considered a Quality control activity. Quality assurance is associated with process models like CMM, CMMI, ISO 9001 etc.
25. These two function should not be viewed as mutually exclusive, 'either-or' choices. Often organisations swing from one extreme to another, one rooting for defect prevention(quality assurance) and other rooting for defect detection(quality control). This should be avoided. Quality control and assurance are

supplementary activities, and should be done in a right mix to develop a quality product. This grid describes the relationship between quality control and quality assurance.



26. Defect prevention is process focussed and defect detection is product focused. Defect detection acts as an extra check to

augment the effectiveness of defect prevention.

27. Quality is institutionalized with consistently high focus on both defect prevention and detection. Although, an organisation may have to allocate sufficient resources for sustaining a high level of both Quality control and quality assurance activities.
28. The relative emphasis placed on the defect detection and prevention will vary on the type of product, closeness of release date and resources available. Making a conscious choice of balance will enable an organisation to produce better quality products. It is important not to over-emphasize one of these at the expense of the other.
29. Automation in Software Testing : Some Points to Consider

- Automated testing requires careful planning, evaluation and training. Automation may not produce immediate results.

- Expecting immediate results from automation can bring dissapointment and lead people to blame automation, instead of objectively looking at their level of preparedness for automation in terms of planning, evaluation and training.
- A failure in doing so leads to organisations reverting back to manual testing. It is like a double edged sword.
- Know first why you want to automate and what you want to automate, before recommending automation just for automation's sake.
- Evaluate multiple tools before choosing one as being most appropriate for you needs.
- Try to choose tools to match your needs, rather than changing your needs to match the tool's capabilities.
- Train people first before expecting them to be productive.
- Do not expect overnight returns from automation.

Software Development Life Cycle Models

6 Phases of a Software Project

- Requirements gathering and analysis
- Planning
- Design
- Development or Coding
- Testing
- Deployment and maintenance

Requirement Gathering & Analysis

- The specific requirements of the software to be built are gathered and documented.
- A product marketing team within a software product organisation specifies the requirements of multiple potential

customers.

- It is important to ensure that the right requirements are captured at every stage.
- The requirements get documented in the form of a System Requirements Specification(SRS) document. This document acts as the bridge between the customer and the designers who build the product.

Planning

- To come up with schedule, scope, resource requirements for a release.
- A plan explains how the requirements will be met and by which time.
- It needs to take into account what requirements will and will not be met for the current release, decide on the scope of the

project, look at resource availability, and come out with a set of milestones and a release date.

- Planning is done for both development and testing. At the end of this stage, both the documents of project plan and test plan are released.

Design

- Figure out how to satisfy the requirements laid out in the SRS document.
- The design phase produces a representation that will be used by the next phase of development. It serves two purposes.
- First, it should be possible to verify that all the requirements are satisfied.
- Second, it should give sufficient information for the development phase to proceed with the coding and implementation of the system.

- Design is split into two parts - High Level Design and Low Level Design.
- The design stage produces the System Design Description(SDD) document that will be used by development teams to produce the programs that implement the design using code.

Development or Coding

- Design acts as the blueprint for actual coding to begin.
- Programs are coded in the chosen programming language.
- It produces the software that meets the requirements the design was meant to satisfy.
- Product documentation is created in this phase.

Testing

- As the programs are coded, they are also tested consistently.

- After coding is nearing completion, the product is subjected to testing.
- Testing is the process of exercising the software product in pre-defined ways to check if the behaviour is the same as the expectations.
- Defects, as many as possible, are identified and removed before shipping the product.

Deployment & Maintenance

- After the testing, the product is given to customers who deploy it in their environments.
- This is where discrepancies can arise between the actual behaviour of the product and what was expected. Such discrepancies could end up as product defects, which need to be corrected.

- The product now enters the maintenance phase, where the product is maintained or changed to satisfy the changes that arise from customer expectations, environmental changes etc.
- Maintenance is made up of 3 components - Corrective maintenance, Adaptive maintenance, Preventive maintenance
- Corrective maintenance is about fixing customer-repoted problems.
- Adaptive maintenance can be like making the software run on a newer version of an operating system.
- Preventive maintenance is about changing the code to avoid a potential security hole that is not covered by the operating system.

Difference In Quality Assurance & Quality Control

For each software feature, the *expected behaviour* is characterized by a set of test cases. Each test case is characterized by :

1. The environment under which the test case is to be executed.
2. Inputs that should be provided for that test case.
3. How these inputs should get processed.
4. What changes should be produced in the internal state or environment.
5. What outputs should be produced.

The *actual behaviour* of a given software for a given test case, under a given set of inputs, in a given environment and in a given internal state is characterized by :

1. How these inputs actually get processed.

2. What changes are actually produced in the internal state or environment.
3. What outputs are actually produced.

If the *actual behaviour* and *expected behaviour* are indentical in all their characteristics, then that test case is said to be passed. If not, then the given software is said to have a *defect* on that test case.

Two methods for ensuring that the chances of the product meeting its requirements are increased - Quality Assurance and Quality Control.

Quality Assurance	**Quality Control**
1. Concentrates on the process of producing the products.	1. Concentrates on specific products after they have been built.
2. Defect prevention oriented.	2. Defect detection and correction oriented.

Quality Assurance	Quality Control
3. Usually done throughout the life cycle.	3. Usually done after the product is built.
4. This is usually a staff function.	4. This is usually a line function.
5. Ex - Reviews and Audits.	5. Ex - Software testing at various levels.

Quality Control

- Attempts to build a product and test it for expected behaviour after it is built.
- If the actual behaviour and expected behaviour are not similar, then fixes and rebuilds are done.
- This iteration is repeatedly done till expected behaviour of the product matches the actual behaviour for the scenarios tested.

- Works on the product rather than processes, focused on defect detection and correction.

Quality Assurance

- Attempts defect prevention by concentrating on the process of producing the product rather than working on defect detection/correction after the product is built.
- To ensure production of better quality code, a quality assurance process may mandate coding standards to be followed by all programmers.
- Quality assurance tends to apply to all the products that use a process.
- QA continues throughout the life of the product, it is everybody's responsibility, hence it is a staff function.
- In contrast, the responsibility for Quality Control is usually localized to a quality control team.

Difference In Testing Verification & Testing Validation

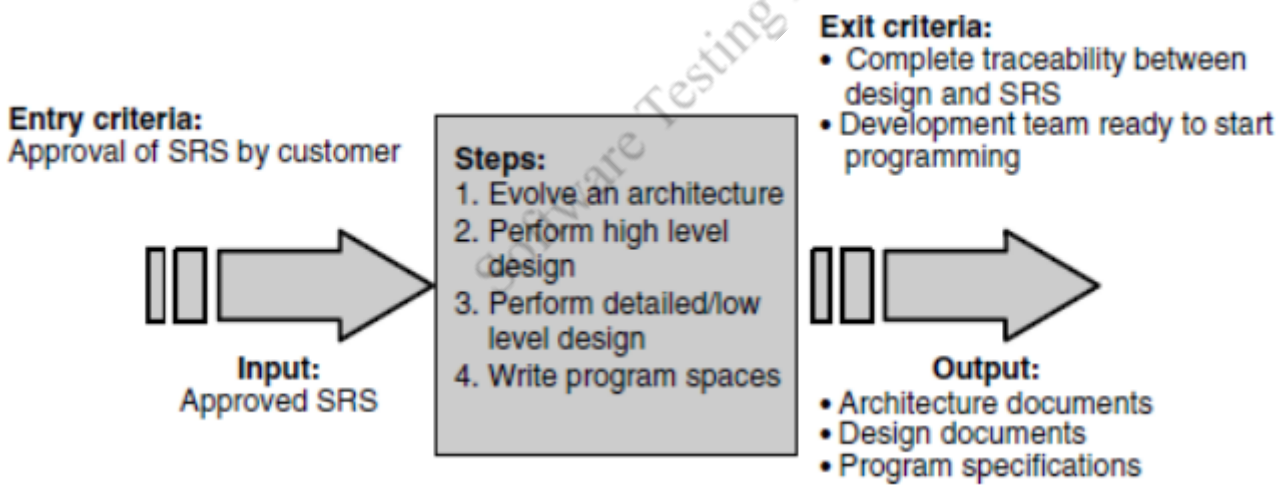
Testing Verification	Testing Validation
1. Quality Assurance = Verification	1. Quality Control = Validation = Testing
2. Verification is the process of evaluating a system or component to determine wether the products of a given phase satisfy the conditions imposed at the start of the phase.	2. Validation is the process of evaluating a system or component during or at the end of the developmental process to determine wether it satisfies specified requirements.
3. Verification takes care of activities to focus on the questions like '*Are we building the product right?*'	3. Validation takes care of a set of activities that address questions like '*Are we building the right product?*'
4. To build the product right, certain activities or conditions or	4. To build the right product, certain activites are carried

Testing Verification	Testing Validation
procedures are imposed at the beginning of life cycle. These activities are considered '*proactive*' as their purpose is to prevent defects before they take shape. The process activities in each phase is termed as verification.	out during various phases to validate wether the product is built as per specifications. These activites are considered '*reactive*'.
5. Ex - Requirements review, design review, code review etc.	5. Ex - Unit testing, Integration testing, System testing etc.

Process Models To Represent Phases

A process model is a way to represent each phase of a software project and is characterized by :

1. Entry criteria specifies when the phase can be started. Inputs for the phase are also included. A clear entry criteria makes sure that a phase doesn't start prematurely.
2. Tasks or steps that need to be carried out in that phase.



3. Verification specifies methods of checking that the tasks have been carried out correctly. Helps in defect prevention and

delays the time gap between defect injection and defect detection.

4. Exit criteria specifies conditions under which one can consider the phase as done. Outputs are also included.
   This model is known as the **E**ntry **T**ask **V**erification E**x**it or ETVX Model.

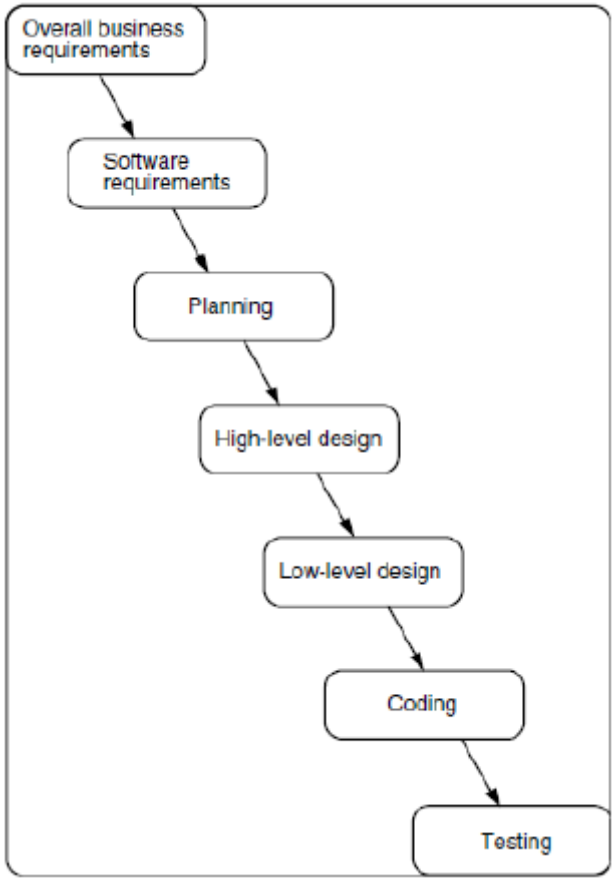
Life Cycle Models

A life cycle model describes how the phases combine together to form a complete project or life cycle. Such a model is characterized by the following attributes :

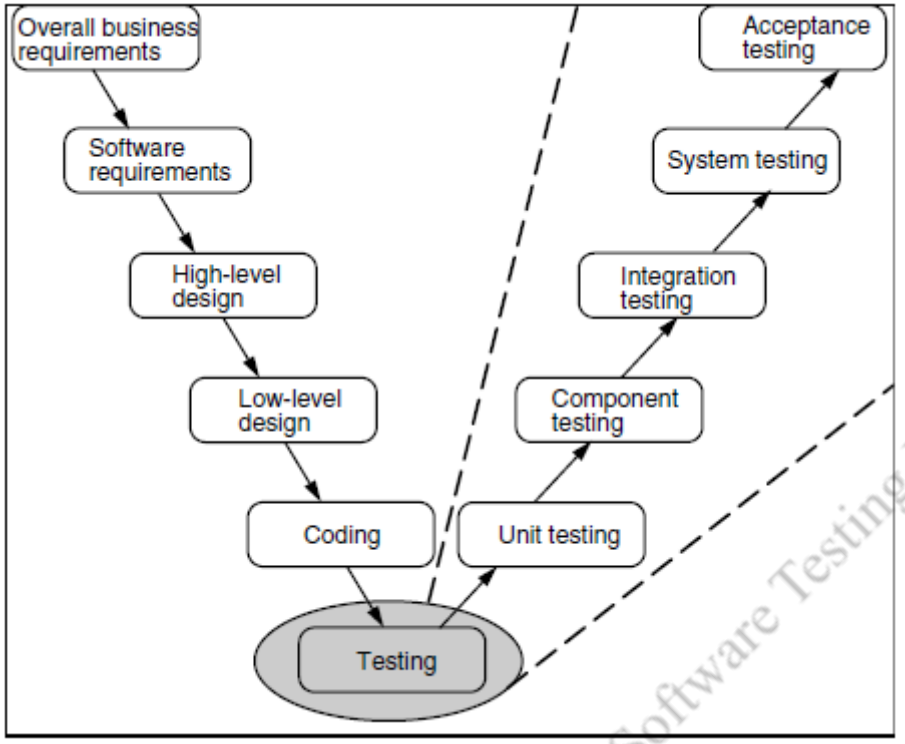
1. The activities performed
2. The deliverables from each activity.
3. Methods of validation of the deliverables.
4. The sequence of activities.
5. Methods of verification of each activity, including the mechanism of communication amongst the activities.

Some Life Cycle Models include :

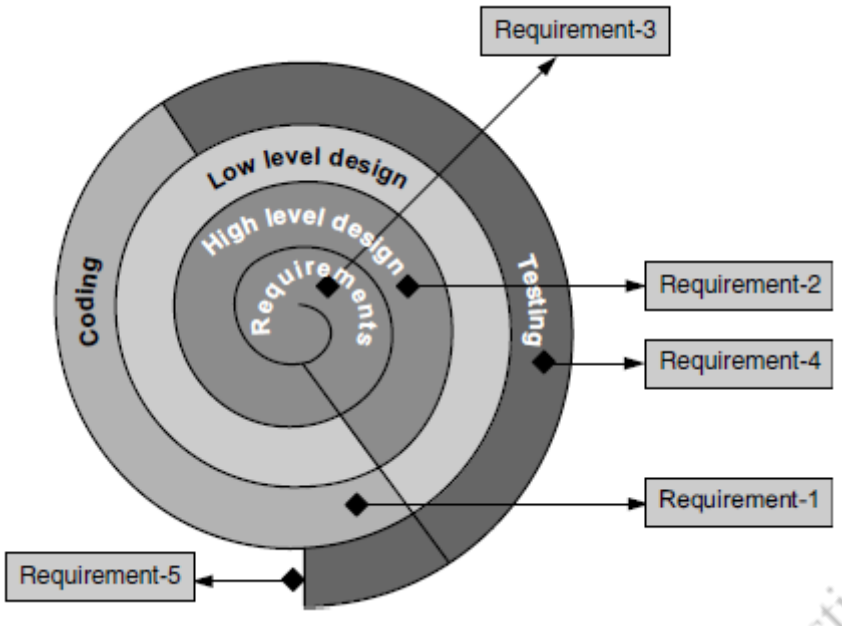
1. Waterfall Model



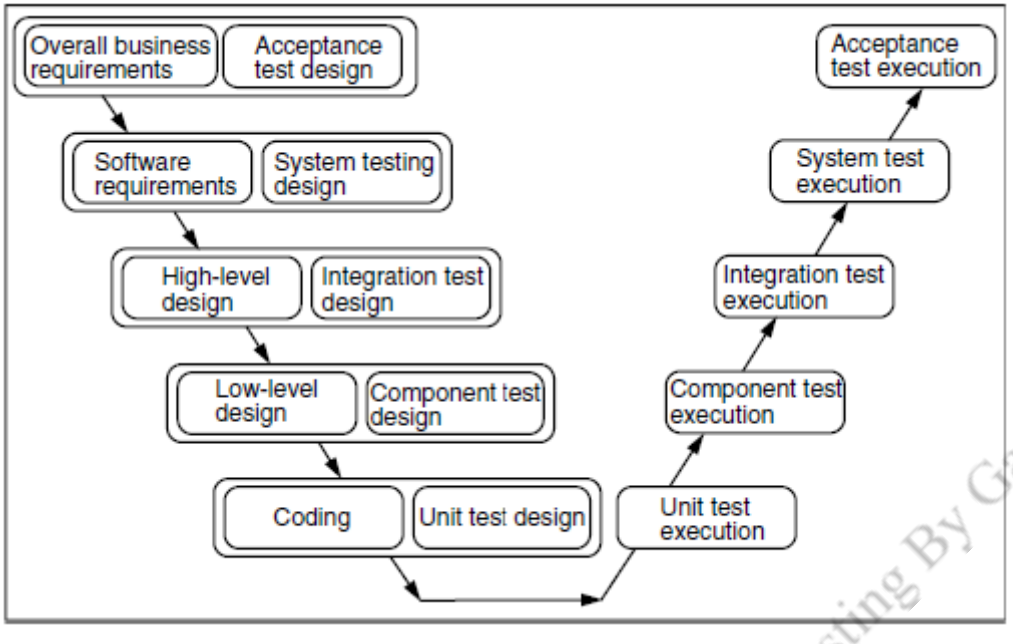
2. Prototyping Model
3. Rapid Application Development(RAD) Models



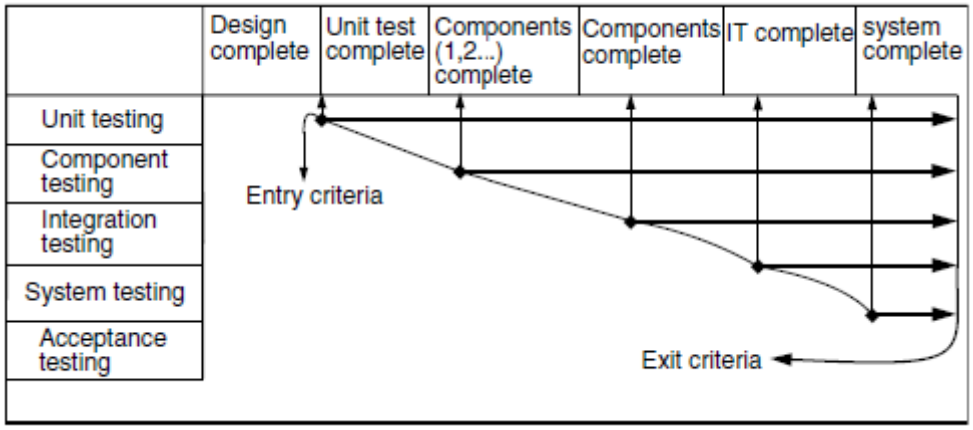
4. Spiral or Iterative Model



5. The V Model

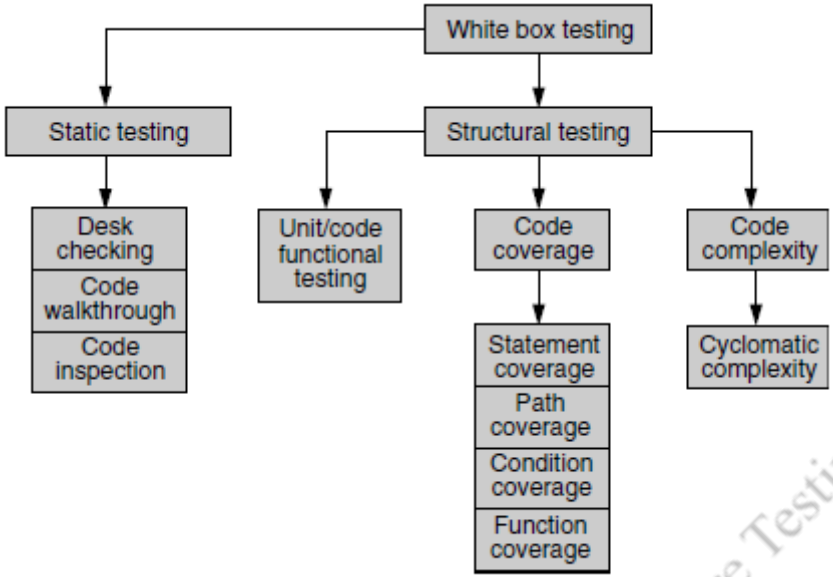


6. Modified V Model



Types Of Testing

White Box Testing



- White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality.
- Also known as *clear box*, or *glass box* or *open box* testing.

- Takes into account the program code, code structure and internal design flow.
- White box testing is done to check various paths in the code and make sure they are exercised correctly.
- Knowing which code paths should be exercised for a given test enables making necessary changes to ensure that appropriate paths are covered.
- WBT is classified into two type - *Static & Structural* Testing

Static Testing

Requires only the source code of the product, not the binaries or executables. Can be done manually by humans or by using specialized tools. Static testing doesn't involve executing the programs but involves select people going through the code to find out wether :

1. The code works according to functional requirement.
2. The code has been written in accordance with the design developed earlier in the project life cycle.
3. The code for any functionality has been missed out.
4. The code handles errors properly.

Static Testing By Humans

Relies on people reading the program code to detect errors rather than computers executing the code to find errors. The advantages include :

1. Sometimes humans can find errors that machines cannot. This can happen when the program has no errors in syntax and runs error free, but has logical errors, like making use of the wrong variable.
2. By making multiple people read the program, we can get multiple perspectives and have more problems identified than a computer could.
3. A human evaluation of code can compare it against the specifications or design and thus ensure it does what is intended to do. This may not always be possible when a computer runs a test.

4. A human evaluation can detect many problems at one go and can even try to identify root causes of the problems. Reactive testing only identifies and fixes the symptoms, not the root cause. Fixing the root cause by human evaluation can fix multiple defects at one go.
5. Computer resources can be saved by making people test the code. This might come at the expense of human resources.
6. A proactive method like static testing minimizes the delay in identification of the problems. The sooner a defect is identified and corrected, lesser is the cost of fixing the defect.
7. Finding defects later in the cycle puts immense pressure on the programmers as they have to fix defects with less time to spare.

Methods to achieve static testing by humans in increasing order of formalism :

1. Desk Checking of Code

2. Code Walkthrough
3. Code Review
4. Code Inspection

Desk Checking

Done manually by authors of the code, this is a method to verify portions of code for correctness. The code is compared with the design specifications to make sure the code does what it is intended to do. Desk checking relies completely on the programmer's diligence and skills. This method is characterized by:

1. No structured method or formalism to ensure completeness.
2. No maintaining of a log or checklist.

Advantages

1. Programmer who knows the code and the programming language well can read and understand his own code very well.
2. There are few schedulings and logistics over-heads as it is done by one individual.

3. The defects are detected and corrected with minimum time delay.

Disadvantages

1. A developer is not the best person to detect problems in his or her own code. They may be tunnel visioned and have blind spots to certain types of problems.
2. Developers prefer to write new code rather than do any testing.
3. This method is essentially person dependent and informal, and thus may not work consistently across all developers.

Code Walkthrough

Walkthroughs are group-oriented and less formal than inspections. The line drawn between walkthroughs and inspections is very thin and varies from one organisation to another.

In walkthroughs, a set of people look at the program code and raise questions for the author. The author explains the logic of the code and answers the questions. If the author is unable to answer some questions, he or she takes those questions and find their answers.

Formal Inspection

Code Inspection, also called Fagan Inspection, is a method with high degree of formalism. The focus is to detect all faults, violations, and other side-effects. The number of defects detected increases by :

1. Demanding thorough preparation before an inspection/review.
2. Enlisiting multiple diverse views.
3. Assigning specific roles to the multiple participants.
4. Going sequentially through the code in a structured manner.

A formal inspection takes place only when the author has made sure the code is ready for inspection by performing basic desk checking and walkthroughs. There has to be a level of readiness in the code before an inspection meeting is arranged. There are four roles in the inspection :

1. *Author*, who writes the code.

2. *Moderator*, who is expected to run the inspection formally according to the process.
3. *Inspectors*, there are typically many of them, and together they review the code.
4. *Scribe*, who takes detailed notes during the inspection meeting and circulates them to the inspection team after the meeting.

Challenges in conducting formal inspections:

1. *Time consuming*, since the process calls for preparation as well as formal meetings.
2. The *logistics* and *scheduling* can become an issue since multiple people are involved.
3. It is not possible to go through every line of code.
4. It may also not be necessary to subject the entire code to formal inspection.

Portions of code can be classified on the basis of their criticality or complexity as 'High', 'Medium' or 'Low'. High or medium complex or critical code should be subjected to formal inspections. The low complexity code can do with walkthroughs or even desk checking.

Static Analysis Tools

Static analysis tools can find errors such as:

1. Wether there are unreachable codes.
2. Variables declared but not used.
3. Mismatch in definition and assignment of values to variables.
4. Illegal or error prone typecasting of variables.
5. Use of non-portable or architecture dependent programming constructs.
6. Memory allocated but not having corresponding statements for freeing the memory.
7. Calculation of cylomatic complexity.

These static analysis tools can also be considered as an extension of compliers as they use the same concept and implementation to locate errors. A good compiler is also a static analysis tool.

Code Review Checklist

Every organization develops it's own code review checklist. In multi-product organizations, the checklist may be at two levels, first an organization wide checklist that will include issues such as organization coding standards, documentation standards, second, a product or project specific checklist that addresses issues specific to the product. The checklist is as follows:

Data Item Declaration Related

- Are the names of the variables meaningful?
- If the programming language allows mixed case names, are there variable names with confusing use of lower case letters and capital letters?
- Are the variables initialized?
- Are there similar sounding names?

- Are all the common structures, constants and flags to be used defined in a header file rather than in each file separately?

Data Usage Related

- Are values of right data types being assigned to the variables?
- Is the access of data from any standard files, repositories, or databases done through publicly supported interfaces?
- If pointers are used, are they initialized correctly?
- Are bounds to array subscripts and pointers properly checked?
- Has the usage of similar looking operators(like = and == or & and &&) checked?

Control Flow Related

- Are all the conditional paths reachable?
- Are all the individual conditions in a complex condition separately evaluated?

- If there is a nested IF statement, are the THEN and ELSE parts properly delimited?
- In the case of a multi-way branch like SWITCH/CASE statements, is a default clause provided? Are the breaks after each CASE appropriate?
- Is there any part of code that is unreachable?
- Are there any loops that will never execute?
- Are there any loops where the final condition will never be met and hence cause the program to go into an infinite loop?
- What is the level of nesting of the conditional statements? Can the code be simplified to reduce complexity?

Standards Related

- Does the code follow the coding conventions of the organization?

- Does the code follow any coding conventions that are platform specific?

Style Related

- Are unhealthy programming constructs being used in the program?
- Is there usage of specific idiosyncrasies of a particular machine architecture or a given version of an underlying product?
- Is sufficient attention being paid to readability issues like indentation of code?

Miscellaneous

- Have you checked for memory leaks?

Documentation Related

- Is the code adequately documented, especially where the logic is complex or the section of code is critical for product functioning?
- Is appropriate change history documented?
- Are the interfaces and the parameters thereof properly documented?

Structural Testing

Structural testing takes into account the code, code structure, internal design, and how they are coded. These tests are run by the computer on the built product, unlike static testing which is tested by humans.

It involves running the actual product against some pre-designed test cases. Further classified into:

1. Unit/Code Functional Testing
2. Code Coverage Testing
3. Code Complexity Testing

Unit/Code Functional Testing

These are some quick checks that a developer performs before subjecting the code to more extensive code coverage testing or code complexity testing.

1. Initially, the developer can perform certain obvious tests, knowing the input variables and the corresponding output values. Repeat this test for multiple input values. This can also be done prior to formal reviews in static testing.
2. For more complex logic or conditions, the developer can build a '*debug version*' of the product by putting intermediate print statements and making sure the program is passing through the right loops and iterations the right number of times. The print statements are removed after the defects are removed.
3. Another approach is to do the initial test to run the product under a debugger or an IDE. These tools allow single stepping of instructions.
   These fall more into the '*debugging*' category rather than testing.

Code Coverage Testing

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by the testing. The percentage of code covered by a test is found by adopting a technique called *instrumentation* of code.

Instrumentation rebuilds the product, linking the product with a set of libraries provided by the tool vendors. This instrumented code can monitor and keep an audit of what portions are covered.

Code coverage testing is made up of the following types of coverage:

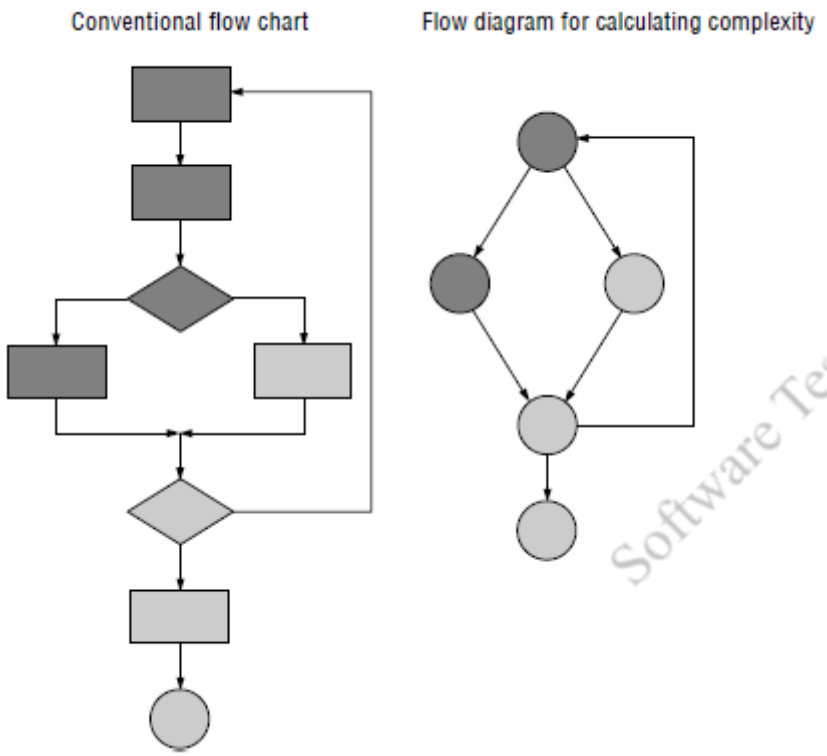
1. Statement coverage
2. Path coverage
3. Condition coverage
4. Function coverage

Few other uses of code coverage testing include:

1. Performance analysis and optimization
2. Resource usage analysis
3. Checking of critical sections or concurrency related parts of code.
4. Identifying memory leaks.
5. Dynamically generated code.

Code Complexity Testing

Cyclomatic complexity is a metric that quantifies the complexity of a program. A program is represented in the form of a *flow graph*, which consists of *node* and *edges*.



A standard flow chart can be converted into a flow graph with these steps:

1. Identify the predicates or decision points in the program, which are usually boolean statements or conditions.
2. Ensure that the predicates are simple, containing no 'and/or'. Complex predicates have to be broken down into simple predicates.
3. Combine all sequential statements into a single node.
4. When a set of sequential statements are followed by a simple predicate, combine all the sequential statements and the predicate check into one node and have two edges emanating from this node. Such nodes with two edges emanating from them are called *predicate nodes*.
5. Make sure that all edges terminate at some node, add a node to represent all the sets of sequential statements at the end of the program.

Cyclomatic Complexity = Edges(E) - Node(N) + 2 = Number of Predicate Nodes + 1

Complexity	What It Means
1 - 10	Well written code, testability is high, cost/effort to maintain is low.
10 - 20	Moderately complex, testability is medium, cost/effort to maintain is medium.
20 - 40	Very complex, testability is low, cost/effort to maintain is high.
> 40	Not testable, any amount of money/effort to maintain may not be enough.

Challenges In White Box Testing

- White box testing requires sound knowledge of the program code and the programming language.
- Developers, in general do not like to perform testing functions, both static and structural.
- Programmers may not 'find time' due to timeline pressures.
- Human tendency of a developer being unable to find the defects in his or her own code.
- Fully tested code may not correspond to realistic scenarios.
- This doesn't mean white box testing is ineffective. These challenges can be overcome by involving other types of testing along with white box testing.

Black Box Testing

What is Black Box Testing?

- Black box testing is done to know the external functionality of what the product should do.
- Doesn't look at the program code but looks at the product from an external perspective.
- Done without the knowledge of internals of system under test.
- Done from the customer's viewpoint.
- The test engineer engaged in BBT only knows set of inputs and expected outputs, and is unaware of how these inputs are processed internally by the product.
- Convenient to administer because they use the complete finished product and do not require any knowledge of its construction.

Why Black Box Testing?

BBT helps in overall functionality verification of the system under test.

1. Done based on requirements. Helps in identifying any incomplete, inconsistent requirements or issues involved when the system is tested as a complete entity.
2. Addresses the stated requirements as well as implied requirements. All requirements are not stated explicitly, but are deemed implicit.
3. Encompasses the end user perspectives.
4. Handles valid and invalid inputs.

When To Do Black Box Testing?

- BBT requires involvement of the testing team right from the beginning of the software project life cycle.
- Testers get involved right from requirements gathering phase.
- Test scenarios and test data are prepared during the test construction phase of the test life cycle, when the software is in design phase.
- Once the code is ready and delivered for testing, test execution is done.
- All the test scenarios developed during the construction phase are executed.
- A subset of these test scenarios are selected for regression testing.

How To Do Black Box Testing?

BBT exploits specifications to generate test cases in a methodical way to avoid redundancy and to provide better coverage. The techniques include :

1. Requirements Based Testing
2. Positive Testing
3. Negative Testing
4. Boundary Value Analysis
5. Decision Tables
6. Equivalence Partitioning
7. State Based or Graph Based Testing
8. Compatibility Testing
9. User Documentation Testing
10. Domain Testing

Requirements Based Testing

- Requirements testing deals with validating the requirements given in the Software Requirements Specification(SRS) of the software product.
- Explicit requirements are stated and documented clearly in the SRS. Implied or implicit requirements are those that are not documented but assumed to be incorporated in the system.
- Precondition is a detailed review of the requirements specifications to ensure they are consistent, correct, complete and testable.
- The review ensures that some implied requirements are converted and documented as explicit requirements, making the testing more effective.
- All explicit and implied requirements are collected and documented as '*Test Requirements Specification(TRS)*'.

- The TRS documented is used for Requirements based testing.
- Requirements are tracked by a '*Requirements Traceability Matrix(RTM)*'. An RTM traces all the requirements from their beginning through design, development and testing.
- The matrix evolves through the life cycle of the project. Each requirement is given a unique id along with a brief description, which are taken from the Requirements specification.
- The requirements are named with a naming convention, which depends on each organization.
- Each requirement is assigned a requirement priority, classified as high, medium or low. Tests for high priority requirements will get precedence over low priority ones, which ensures that the functionality with high risk or high stakes is tested earlier, and the defects fixed at the earliest.

- The RTM contains columns like '*test conditions*', '*test case IDs*' and '*phase of testing*'.
- A requirement is subjected to multiple phases of testing - unit, component, integration, system testing etc.

The RTM helps in identifying the relationship between the requirements and test cases. The following combinations are possible:

1. One to one
2. One to many
3. Many to one
4. Many to many
5. One to none

The Requirements Traceability Matrix or RTM provides a wealth of information on various test metrics:

1. Requirements addressed priority wise helps addressing the high priority requirements first.
2. Number of test cases requirement wise
3. Total number of test cases prepared.

Once the test cases are executed, the test results can be used to collect metrics such as:

1. Total number of test cases passed.
2. Total number of test cases failed.
3. Total number of defects in requirements.
4. Number of requirements completed.
5. Number of requirements pending.

These metrics can also be displayed graphically to better visualize what we are dealing with.

Positive Testing

- Positive testing tries to prove that a given product does what it is supposed to do.
- When a test case verifies the requirements of the product with a set of expected output, it is called positive test case.
- The purpose of positive testing is to prove that the product works as per specification and expectation.
- A product delivering an error when it is expected to give an error, is also a part of positive testing.
- Positive testing is done to verify the known test conditions.

Negative Testing

- Negative testing is done to show that the product does not fail when an unexpected input is given.
- The purpose of negative testing is to try and break the system with unknowns.
- Covers scenarios for which the product is not designed and coded. The input values may not have been represented in the specification of the product. These are termed as unknown conditions.
- It is important to know for testers to know the negative situations that may occur at end-user level so that the application can be tested and made foolproof.
- A negative test would be a product not delivering an error when it should or delivering an error when it should not.

The difference between Positive and negative testing is in their coverage. For positive testing, if all documented requirements and test conditions are covered, then coverage can be said to be 100 percent. In contrast, there is no end to negative testing, and 100 percent coverage is impractical. Negative testing requires a high degree of creativity among the testers to cover as many unknowns as possible to avoid failure.

Boundary Value Analysis

- Most of the defects in software product hover around *conditions* and *boundaries*. Boundaries means the 'limits' of values of the various variables.
- BVA is useful for catching defects that happen at boundaries. It believes that the density of defects is more towards boundaries.
- Boundary Value Analysis is useful to generate test cases when the input(or output) data is made up of clearly indentifiable boundaries or ranges.
- Another instance where boundary value testing is extremely useful in uncovering defects is when there are internal limits placed on certain resources, variables, or data structures.
- Look for any kind of gradation or discontinuity in data values which affect computation - the discontinuities are the boundary values, which require thorough testing.

- Look for internal limits on resources.
- Look for documented limits on hardware resources.
- BVA applies for white box testing as well. Internal data structures like arrays, stacks etc need to be checked for boundary or limit conditions. The way linked lists behave in the beginning and ending have to be tested thoroughly.
- Boundary values and decision tables help identify the test cases that are most likely to uncover defects. A generalization of both these concepts is the concept of *equivalence classes*.

Decision Tables

A *decision table* lists the various decision variables, the conditions assumed by each of the decision variables, and actions to take in each combination of conditions.

The steps in forming a decision table are as follows:

1. Identify the decision variables.
2. Identify the possible values of each of the decision variables.
3. Enumerate the combinations of the allowed values of each of the variables.
4. Identify the cases when values assumed by a variable are immaterial for a given combination of other input variables. Represent such variables by 'dont care' symbol, which is usually the greek character *phi*.

5. For each combination of values of decision variables, list out the action or expected result.
6. From a table, listing in each but the last column a decision variable. In the last column, list the action item for the combination of variables in that row, including dont cares, as appropriate.

Equivalence Partitioning

- Equivalence partitioning is a software testing technique that involves identifying a small set of representative input values that produce many different outputs conditions as possible.
- Reduces the effort involved in testing and increases coverage.
- The set of input values that generate one single expected output is called a *partition*.
- When the behavior of the software is the same for a set of values, then the set is termed as an *equivalence class* or a *partition*.
- One representative sample from each partition, also called member of equivalence class is picked up for testing.
- The benefits of using equivalence classes : choosing a minimal set of input values that are truly representative of the entire spectrum and uncovering a higher number of defects.

- Equivalence partioning is useful to minimize the number of test cases when the input data can be divided into distinct sets, where the behaviour or outcome of the product within each member of the set is the same.

The equivalence partitions table has columns:

1. Partitions definition
2. Type of input(valid/invalid)
3. Representative test data for that partition
4. Expected results

The steps to prepare an equivalence partitions table are:

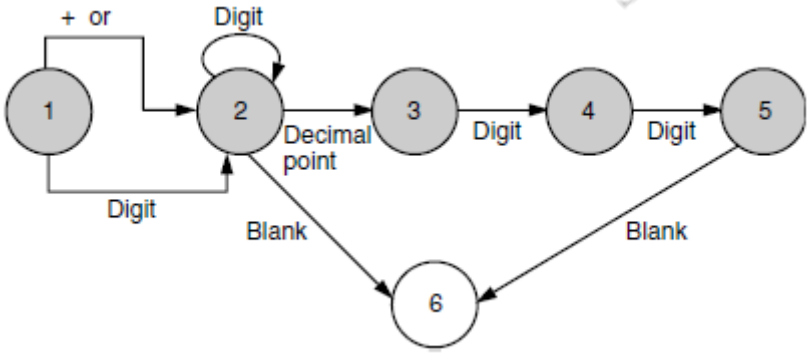
1. Choose criteria for doing the equivalence partitioning(range, list of values etc).
2. Identify the valid equivalence classes based on the above criteria.

3. Select a sample data from that partition.
4. Write the expected results based on the requirements given.
5. Identify special values, if any, and include them in the table.
6. Check to have expected results for all the cases prepared.
7. If the expected result is not clear for any particular test case, mark it and escalate for corrective actions.

State Based or Graph Based Testing

State or graph based testing is useful in situations where:

1. The product under test is a language processor(like a compiler), wherein the syntax of the language automatically lends itself to a state machine.
2. Workflow modeling, where, depending on the current state and appropriate combinations of input variables, specific workflows are carried out, resulting in new output and state.
3. Dataflow modeling, where the system is modeled as a set of dataflow, leading from one state to another.



A general outline for using state based testing methods with respect to language processors is :

1. Identify the grammar for the scenario.
2. Design test cases corresponding to each valid state-input combination.
3. Design test cases corresponding to the most common invalid combinations of state-input.

Compatibility Testing

Compatibility Testing(CT) is done to ensure that the product features work consistently with different infrastructure components is called compatibility testing. Requires high degree of effort as there are a large number of parameter combinations.

The parameters that generally affect the comptability of the product are:

1. Processor and number of processors.
2. Architecture(32bit, 64bit and so on).
3. Resource availability(RAM, disk space, network card etc).
4. Equipment that the product is expected to work with.
5. Operating system.
6. Middle-tier infrastructure components such as web server, application server, network server.

7. Backend components such as database servers.
8. Services that require special hardware-cum-software solutions like clusters, load balancers etc.
9. Any software used to generate product binaries.
10. Various technological components used to generate components(SDK, JDK etc).

A *compatibility matrix* is created. It has its columns as various parameters the combinations of which have to be tested. Each row represents a unique combination of a specific set of values of the parameters.

Techniques for CT include:

1. Horizontal Combination
2. Intelligent Sampling

Compatibility testing of a product involving parts of itself can be classified into two types:

1. Backward compatibility testing
2. Forward compatibility testing

User Documentation Testing

User documentation covers all the manuals, user guides, installation guides, setup guides, read me file, software release notes, and online help that are provided along with the software to help the end user to understand the software system.

User documentation testing has two objectives:

1. To check if what is stated in the document is available in the product.
2. To check if what is there in the product is explained correctly in the document.

Benefits of user documentation testing include:

1. User documentation testing aids in highlighting problems overlooked during reviews.
2. Ensures consistency of documentation and product.

3. Results in less difficult calls for the support staff.
4. New programmers and testers who join a project group can use the documentation to learn the external functionality of the product.
5. Customers need less training and can proceed more quickly to advanced training and product usage if the documentation is of high quality and is consistent with the product.

Domain Testing

- Domain testing goes beyond white box and black box testing, where we do not even look at the specifications of a software product but are testing the product, purely based on domain knowledge and expertise in the domain of application.
- This testing approach requires critical understanding of the day to day business activities for which the software is designed.
- Domain testing is an extension of black box testing.
- The test engineers performing this type of testing are selected because they have in-depth knowledge of the business domain.
- Domain testing exploits the tester's domain knowledge to test the suitability of the product to what the users do on a typical day.
- Domain testing is the ability to design and execute test cases that related to the people who will buy and use that software.

- Domain testing involves testing the product, not by going through the logic built into the product. The business flow determines the steps, not the software under test. This is also called 'business vertical testing'.
- Domain testing is done after all components are integrated and after the product has been tested by other black box techniques.

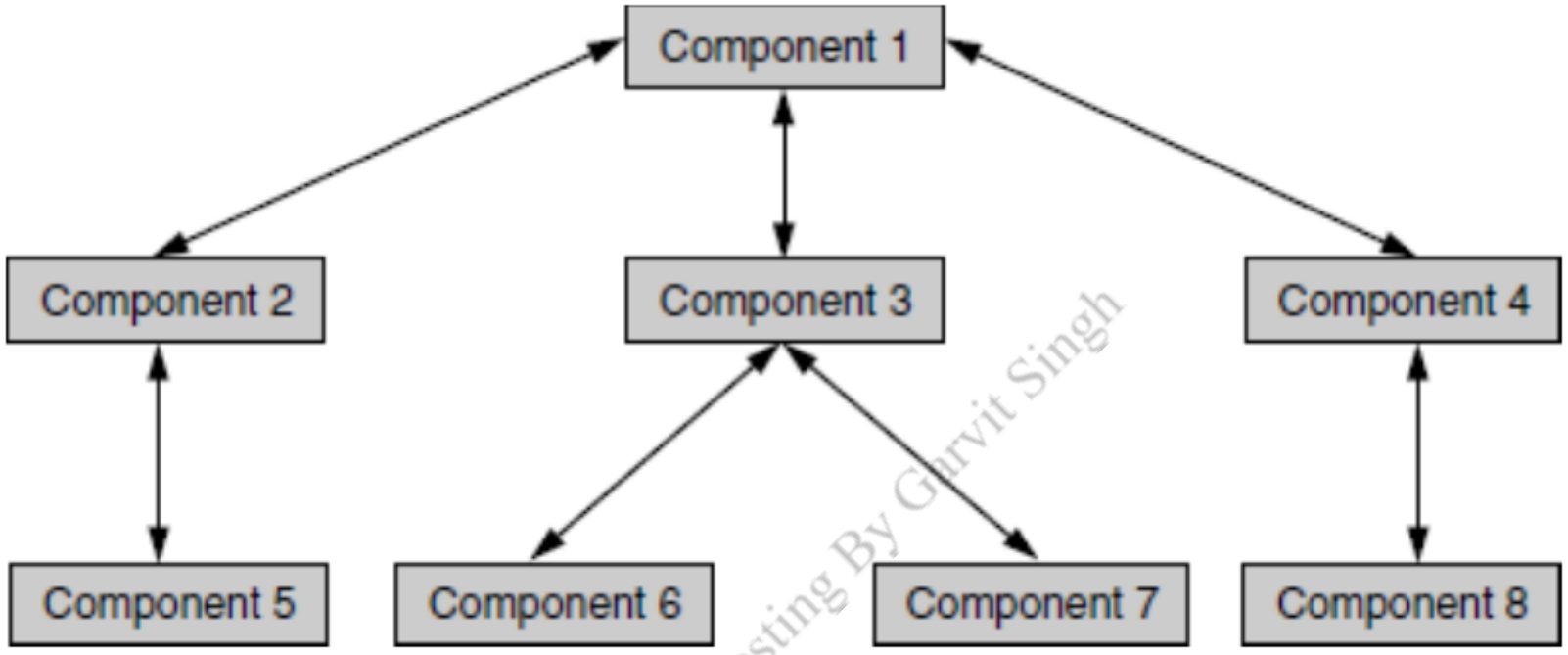
Scenarios	**Most Effective Black Box Testing Technique**
Output values dictated by certain conditions depending upon values of input variables.	Decision Tables
Input values in ranges, with each range exhibiting a particular functionality.	Boundary Value Analysis

Scenarios	Most Effective Black Box Testing Technique
Input values divided into classes, with each class exhibiting a particular functionality.	Equivalence Partitioning
Checking for expected and unexpected input values.	Positive & Negative Testing
Workflows, process flows, or language processors.	Graph or State Based Testing
To ensure that requirements are tested and met properly.	Requirements Based Testing
To test domain expertise rather than product specification.	Domain Testing
To ensure that the documentation is consistent with the product.	Documentation Testing

Integration Testing

- A system is made up of multiple components or modules comprising hardware and software.
- Integration is defined as the set of interactions among components.
- Testing the interaction between the modules and interaction with other systems externally is called integration testing.
- Integration testing is done to make sure that the different components fit together.
- The final round of integration involving all components is called Final Integration Testing (FIT), also known as System Integration.
- Integration is both a phase and a type of testing.
- 4 orders of integration testing : Top-down integration, Bottom-up integration, Bi-directional integration, System integration.

Top-Down Integration



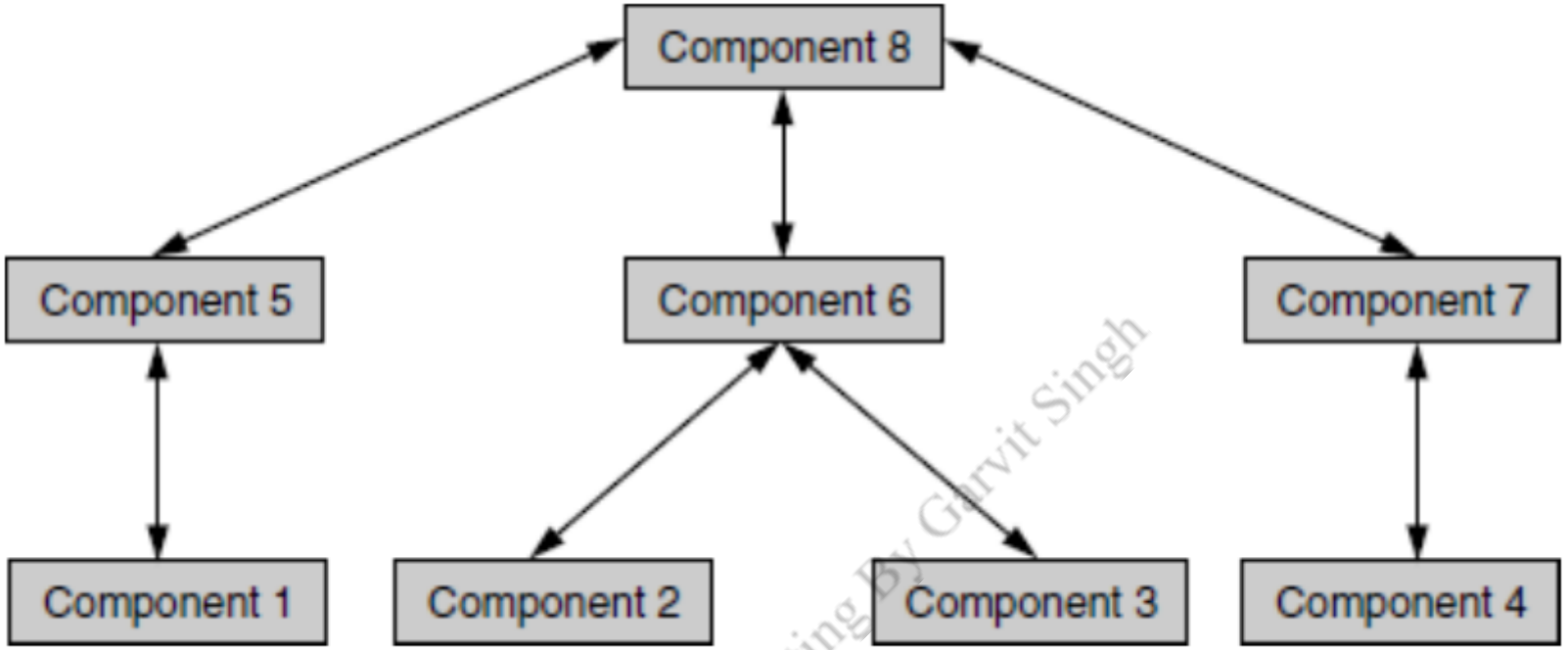
Step Interfaces tested

1. 1-2
2. 1-3
3. 1-4
4. 1-2-5

5. 1-3-6
6. 1-3-6-(3-7)
7. (1-2-5)-(1-3-6-(3-7))
8. 1-4-8
9. (1-2-5)-(1-3-6-(3-7))-(1-4-8)

- The integration starts with testing the interface between component 1 and component 2, then component 1 and 3 and so on.
- To optimize the number of steps in integration testing, steps 6 and 7 can be combined and executed as a single step.
- Similarly, steps 8 and 9 also can be combined and tested in a single step.

Bottom-Up Integration



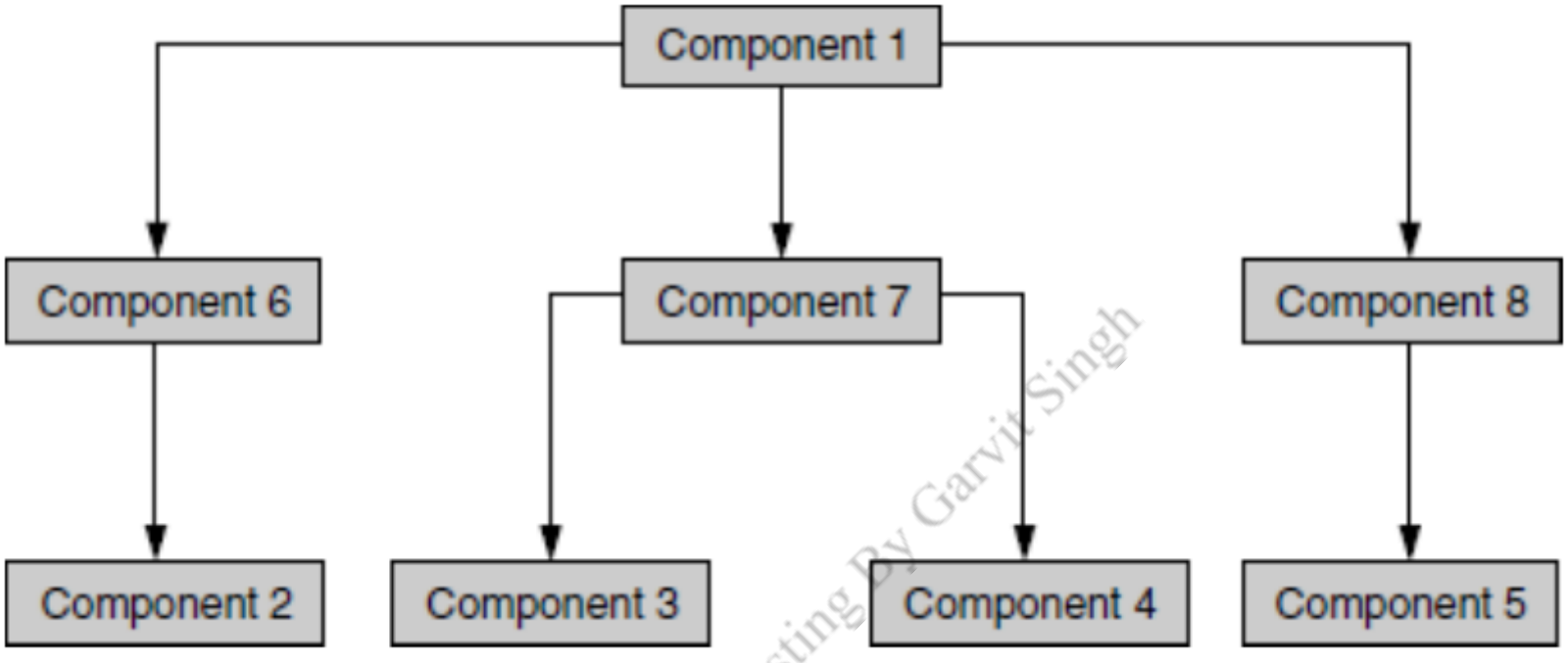
Step Interfaces tested

1. 1-5
2. 2-6, 3-6
3. 2-6-(3-6)
4. 4-7

5. 1-5-8
6. 2-6-(3-6)-8
7. 4-7-8
8. (1-5-8)-(2-6-(3-6)-8)-(4-7-8)

- Bottom-up integration is just the opposite of top-down integration, where the components for a new product development become available in reverse order, starting from the bottom.

Bi-Directional Integration



Step Interfaces tested

1. 6-2
2. 7-3-4
3. 8-5
4. (1-6-2)-(1-7-3-4)-(1-8-5)

- Bi-directional integration is a combination of the top-down and bottom-up integration together to derive integration steps.
- Comes handy when migrating from a two-tier to a three-tier environment.

System Integration

System integration means that all the components of the system are integrated and tested as a single unit.

Integration testing, which is testing of interfaces, can be divided into two types:

1. Components or sub-system integration
2. Final integration testing or system integration

Instead of integrating component by component and testing, this approach waits till all components arrive and one round of integration testing is done.

This approach is also called *big-bang integration.* It reduces testing effort and removes duplication in testing.

Big bang integration is ideal for a product where the interfaces are stable with less number of defects.

While this approach saves time and effort, it is also not without disadvantages:

1. When a failure or defect is encountered during system integration, it is very difficult to locate the problem, to find out in which interface the defect exists. The debug cycle may involve focusing on specific interfaces and testing them again.
2. The ownership for correcting the root cause of the defect may be a difficult issue to pinpoint.
3. When integration testing happens in the end, the pressure from the approaching release date is very high. This pressure on the engineers may cause them to compromise on the quality of the product.
4. A certain component may take an excessive amount of time to be ready. This precludes testing other interfaces and wastes time till the end.

Choosing Integration Method

Factors	Suggested Integrated Method
Clear requirements and design.	Top-down
Dynamically changing requirements, design, architecture.	Bottom-up
Changing architecture, stable design.	Bi-directional
Limited changes to existing architecture with less impact.	Big bang

Integration Testing as a Phase of Testing

- Starts from the point where two components can be tested together, to the point where all the components work together as a complete system, delivering the product functionality.
- It tests how different components work in isolation and also when they are connected to each other.
- Focuses predominantly on defects that arise due to combing various components.
- Focuses on interfaces as well as usage flow.
- When the functionality of different components are combined and tested together for a sequence of related operations, they are called *scenarios.*

Scenario Testing

- Scenario testing is a planned activity to explore different usage patterns and combine them into test cases called scenario test cases.
- Scenario testing is defined as a “set of realistic user activities that are used for evaluating the product.”
- It is also defined as the testing involving customer scenarios.
- Two methods to create scenarios : System Scenarios & Use-case/Role-based Scenarios

System Scenarios

System scenario is a method whereby the set of activities used for scenario testing covers several components in the system. Follow these approaches :

1. Story line - Develop a story line that combines various activities of the product that may be executed by an end user.
2. Life cycle/state transition - Consider an object, derive the different transitions/modifications that happen to the object, and derive scenarios to cover them.
3. Deployment/implementation stories from customer - Develop a scenario from a known customer deployment/implementation details and create a set of activities by various users in that implementation.
4. Business verticals - Visualize how a product/software will be applied to different verticals and create a set of activities as

scenarios to address specific vertical businesses.

5. Battle ground - Create some scenarios to justify that “the product works” and some scenarios to “try and break the system” to justify “the product doesn't work.”
6. These approaches have to be used in combination, not in isolation

Use Case Scenarios

A use case scenario is a stepwise procedure on how a user intends to use a system, with different user roles and associated parameters.

Defect Bash

1. Defect bash is an ad hoc testing where people performing different roles in an organization test the product together at the same time.
2. Ad hoc testing means that the testing is unplanned.
3. The testing by all the participants during defect bash is not based on written test cases.
4. What is to be tested is left to an individual's decision and creativity.

Good Practices Encouraged By Defect Bash

1. Enabling people cross boundaries and test beyond assigned areas.
2. Bringing different people performing different roles together in the organization for testing. Testing isn't for testers alone.
3. Letting everyone in the organization use the product before delivery.
4. Bringing fresh pairs of eyes to uncover new defects.
5. Bringing in people who have different levels of product understanding to test the product together randomly.
6. Let testing doesn't wait for lack of/time taken for documentation.
7. Enabling people to say “system works” as well as enabling them to “break the system”.

All the activities in the defect bash are planned activities, except for what to be tested. It involves several steps:

1. Choosing the frequency and duration of defect bash.
2. Selecting the right product build.
3. Communicating the objective of each defect bash to everyone.
4. Setting up and monitoring the lab for defect bash.
5. Taking actions and fixing issues.
6. Optimizing the effort involved in defect bash.

System Testing

- System testing is defined as a testing phase conducted on the complete integrated system, to evaluate the system compliance with its specified requirements.
- It is done after unit, component, and integration testing phases.
- The testing conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and nonfunctional aspects is called system testing.
- System testing brings out issues that are fundamental to design, architecture, and code of the whole product.
- System testing is the only phase of testing which tests the both functional and non-functional aspects of the product.
- Two types - Functional & Non-Functional.

What is a System?

- A system is a complete set of integrated components that together deliver product functionality and features.
- A system can also be defined as a set of hardware, software, and other parts that together provide product features and solutions.

Why is System Testing Done?

For these reasons:

1. Provide independent perspective in testing
2. Bring in customer perspective in testing
3. Provide a “fresh pair of eyes” to discover defects not found earlier by testing
4. Test product behavior in a holistic, complete, and realistic environment
5. Test both functional and non-functional aspects of the product
6. Build confidence in the product
7. Analyze and reduce the risk of releasing the product
8. Ensure all requirements are met and ready the product for acceptance testing.

Functional Testing vs Non Functional Testing

Testing Aspects	Functional Testing	Non Functional Testing
Involves	Product features and functionality	Quality factors
Tests	Product behavior	Behavior and experience
Result conclusion	Simple steps written to check expected results	Huge data collected and analyzed
Results varies due to	Product implementation	Product implementation, resources, and configurations
Testing focus	Defect detection	Qualification of product
Knowledge required	Product and domain	Product, domain, design, architecture, statistical skills

Testing Aspects	Functional Testing	Non Functional Testing
Failures normally due to	Code	Architecture, design, and code
Testing Phase	Unit, component, integration, system	System
Test case repeatability	Repeated many times	Repeated only in case of failures and for different configurations
Configuration	One-time setup for a set of test cases	Configuration changes for each test case

Functional System Testing

- Functional testing is performed at different phases, two problems arise : *Duplication* & *Gray area*.
- *Duplication* refers to the same tests being performed multiple times.
- *Gray area* refers to certain tests being missed out in all the phases.
- Gray areas in testing happen due to lack of product knowledge, lack of knowledge of customer usage, and lack of co-ordination across test teams.

Ways system functional testing is performed:

1. Design/architecture verification
2. Business vertical testing
3. Deployment testing
4. Beta testing
5. Certification, standards, and testing for compliance.

Design/Architecture Verification

Helps in validating the product features that are written based on customer scenarios and verifying them using product implementation

Certain test cases are identified and moved to earlier phases of testing like integration or component testing to catch defects early and avoid any major defects later.

Guidelines used to reject test cases:

1. Is this focusing on code logic, data structures, and unit of the product? If yes, then it belongs to unit testing.
2. Is this specified in the functional specification of any component? If yes, then it belongs to component testing.
3. Is this specified in design and architecture specification for integration testing? If yes, then it belongs to integration testing.

4. Is it focusing on product implementation but not visible to customers? This is focusing on implementation - to be covered in unit/component/integration testing.
5. Is it the right mix of customer usage and product implementation? Customer usage is a prerequisite for system testing.

Business Vertical Testing

Overview

- General-purpose products like workflow automation systems are designed to cater to various businesses and services.
- Business vertical testing involves adapting and testing the product for different industry verticals such as insurance, banking, and asset management.

Customization in Business Vertical Testing

- The product's procedures are altered to align with specific business processes.
- User objects (e.g., clerk, officer) are created and associated with operations to customize the product for different business needs.

- Role-based operations are implemented to ensure certain tasks are performed by specific user roles.

Terminology Considerations

- The product adapts its terminology to match industry-specific language.
- For example, an email in insurance might be referred to as a *claim*, while in a loan processing system, it may be called a *loan application*.

Importance of Understanding Business Processes

- The product needs to comprehend the intricacies of business processes to effectively customize its workflow for different verticals.
- Customization features are crucial to accommodate the unique requirements of various business domains.

Syndication in Business Vertical Testing

- Some tasks related to business verticals may be handled by solution integrators or service providers.
- Licensing agreements may involve changing product names, company names, and copyrights to reflect the identity of the solution integrators or service providers.

Testing Approaches

- Business vertical testing can be performed through simulation or replication.
- Simulation involves assuming requirements and testing the business flow
- Replication involves obtaining customer data and customizing the product accordingly.

Scenario Testing in Integration and System Phases

- Integration testing creates business vertical scenarios, focusing on interfaces and interactions.
- System testing evaluates business verticals in a real-life customer environment, considering customization, terminology, and syndication aspects.

Deployment Testing

1. Overview
    - System testing is the final phase before product delivery.
    - Ensures that customer deployment requirements are met, assessing short-term success or failure based on customer satisfaction.
2. Offsite Deployment Testing
    - Conducted to simulate customer deployment requirements.
    - Aims to ensure that the product is ready for customers who await it.
3. Onsite Deployment Testing
    - Conducted after the release of the product at customers' locations.

- Involves collaboration between the product development organization and the organization using the product.

Stages of Onsite Deployment Testing

Stage 1

- Mirrored deployment machines with similar configurations are set up.
- Actual data from the live system is taken, and user operations are rerun on the mirrored deployment machine.
- Verifies whether the enhanced or similar product can perform existing functionality without affecting users.
- Intelligent recorders may be used to maintain identical mirrored and live systems regarding business transactions.

Stage 2

- Mirrored system from Stage 1 is made a live system running the new product.
- Regular backups are taken, and alternative methods are used to record incremental transactions.
- Recorded transactions from the mirrored system are preserved with the old live system as a fallback.
- If no failures are observed in this stage for an extended period, the onsite deployment is considered successful, and the old live system is replaced by the new system.

Beta Testing

Sending the product that is under test to the customers and receiving the feedback is called beta testing.

During the entire duration of beta testing, there are various activities that are planned and executed according to a specific schedule. This is called a *beta program*.

Some of the activities involved in the beta program are as follows:

1. Collecting the list of customers and their beta testing requirements along with their expectations on the product.
2. Working out a beta program schedule and informing the customers.
3. Sending some documents for reading in advance and training the customer on product usage.

4. Testing the product to ensure it meets 'beta testing entry criteria'.
5. Sending the beta product to the customer and enable them to carry out their own testing.
6. Collecting the feedback periodically from the customers and prioritizing the defects for fixing.
7. Responding to customer feedback with product fixes or documentation changes and closing the communication loop with the customers in a timely fashion.
8. Analyzing and concluding whether the beta program met the exit criteria.
9. Communicate the progress and action items to customers and formally closing the beta program.
10. Incorporating the appropriate changes in the product.

Certification, Standards and Testing for Compliance

- A product needs to be certified with the popular hardware, operating system, database, and other infrastructure pieces. This is called certification testing.
- There are many standards for each technology area and the product may need to conform to those standards.
- This is very important as adhering to these standards makes the product interact easily with other products.
- Testing the product to ensure that these standards are properly implemented is called *testing for standards*. Once the product is tested for a set of standards, they are published in the *release documentation*.
- Testing the product for contractual, legal, and statutory compliance is one of the critical activities of the system testing team.

Non Functional Testing

- Non-Functional testing is similar to that of functional testing but differs from the aspects of complexity, knowledge requirement, effort needed, and number of times the test cases are repeated.
- Repeating non-functional test cases involves more time, effort, and resources, the process for non-functional testing has to be more robust and stronger than functional testing to minimize the need for repetition.
- This is achieved by having more stringent entry/exit criteria, better planning, and by setting up the configuration with data population in advance for test execution.

Set Up The Configuration

- Two ways setup is done : simulated environment and real-life customer environment.
- Setting up a scenario that is exactly real-life is difficult.
- Simulated setup is used for non-functional testing where actual configuration is difficult to get.
- In order to create a “near real-life” environment, details of customer's hardware setup, deployment information and test data are collected in advance.

Come up with Entry/Exit Criteria

Entry and exit criteria are decided for various test cases based on parameters like Maximum limits, response time, throughput, latency, failures per iteration, failures per test duration, stressing the system beyond limits and so on.

Balancing Key Resources

4 key resources - CPU, Disk, Memory and Network. They need to be judiciously balanced to enhance the quality factors of the product.

Basic assumptions that can be made about resources and configuration:

1. The CPU can be fully utilized as long as it can be freed when a high priority job comes in.
2. The available memory can be completely used by the product as long as the memory is relinquished when another job requires memory.
3. The cost of adding CPU or memory is not that expensive as it was earlier.
4. The product can generate many network packets as long as the network bandwidth and latency is available and does not cost much.

5. More disk space or the complete I/O bandwidth can be used for the product as long as they are available. While disk costs are getting cheaper, IO bandwidth is not.
6. The customer gets the maximum ROI only if the resources such as CPU, disk, memory, and network are optimally used. Software has to be designed intelligently for optimal performance.
7. Graceful degradation in non-functional aspects can be expected when resources in the machine are also utilized for different activities in the server.
8. Predictable variations in performance or scalability are acceptable for different configurations of the same product.
9. Variation in performance and scalability is acceptable when some parameters are tuned, as long as we know the impact of adjusting each of those tunable parameters.

10. The product can behave differently for non-functional factors for different configurations such as low-end & high-end servers as long as they support return on investment. This in fact motivates the customers to upgrade their resources.
11. Once such sample assumptions are validated by the development team and customers, then non-functional testing is conducted.

Scalability Testing

- To find out the maximum capability of the product parameters.
- High resources required.
- A high-end configuration is selected and the scalability parameter is increased step by step to reach the maximum capability.
- When the requirements from the customer are more than what design/architecture can provide, the scalability testing is suspended, the design is reworked, and scalability testing resumed to check the scalability parameters.
- Having a highly scalable system that considers the future requirements of the customer helps a product to have a long lifetime.
- Failures during scalability test include the system not responding, system crashing etc.

- Scalability tests help in identifying the major bottlenecks in a product.
- Sometimes the underlying infrastructure such as the operating system or technology can also become bottlenecks. In that case, the product organisation is expected to work with the OS vendor to resolve the issues.
- Scalability tests are performed on different configurations to check the product's behavior. For each configuration, data are collected and analyzed.
- The demand of resources tends to grow exponentially when the scalability parameter is increased.
- The scalability reaches a saturation point beyond which it cannot be improved. This is called the maximum capability of a scalability parameter. Even though resources may be available, product limitation may still not allow scalability.

- This is called a product bottleneck. Identification of such bottlenecks and removing them in the testing phase as early as possible is a basic requirement for resumption of scalability testing.

Reliability Testing

Done to evaluate the product's ability to perform its required functions under stated conditions for a specified period of time or for a large number of iterations.

Product reliability is achieved by focusing on the following activities:

1. Defined engineering processes.
2. Review of work products at each stage.
3. Change management procedures.
4. Review of testing coverage.
5. Ongoing monitoring of the product.

A *reliability tested product* will have the these characteristics:

1. No errors or very few errors from repeated transactions.
2. Zero downtime.

3. Optimum utilization of resources.
4. Consistent performance and response time of the product for repeated transactions for a specified time duration.
5. No side-effects after the repeated transactions are executed.

Stress Testing

- Stress testing is done to evaluate a system beyond the limits of specified requirements or resources, to ensure that system does not break.
- Done to find out if the product's behavior degrades under extreme conditions and when it is denied the necessary resources, like insufficient memory, inadequate hardware etc.
- The product is over-loaded deliberately to simulate the resource crunch and to find out its behavior.
- It is expected to gracefully degrade on increasing the load, but the system is not expected to crash at any point of time during stress testing.
- The process, data collection, and analysis required for stress testing are very similar to those of reliability testing.
- The difference lies in the way the tests are run.

- Reliability testing is performed by keeping a constant load condition till the test case is completed, the load is increased only in the next iteration of the test case.
- In stress testing, the load is generally increased through various means such as increasing the number of clients, users, and transactions till and beyond the resources are completely utilized.
- When the load keeps on increasing, the product reaches a stress point when some of the transactions start failing due to resources not being available.
- The failure rate may go up beyond this point.
- To continue the stress testing, the load is slightly reduced below this stress point to see whether the product recovers and whether the failure rate decreases appropriately.

- This exercise of increasing/decreasing the load is performed two or three times to check for consistency in behavior and expectations.
- The time required for the product to quickly recover from those failures is represented by MTTR (Mean time to recover).

Guidelines to select the tests for stress testing:

1. *Repetitive tests* - Executing repeated tests ensures that at all times the code works as expected.
2. *Concurrency* - Concurrent tests ensure that the code is exercised in multiple paths and simultaneously.
3. *Magnitude* - This refers to the amount of load to be applied to the product to stress the system.
4. *Random variation* - Stress testing depends on increasing/decreasing variable load.

Interoperability Testing

Interoperability testing is done to ensure the two or more products can exchange information, use
information, and work properly together.

Guidelines that help in improving interoperability.

1. *Consistency of information flow across systems* - When an input is provided to the product, it should be understood consistently by all systems.
2. *Changes to data representation as per the system requirements* - When two different systems are integrated to provide a response to the user, data sent from the first system in a particular format must be modified or adjusted to suit the next system's requirement.
3. *Correlated interchange of messages and receiving appropriate responses* - When one system sends an input in the form of a

message, the next system is in the waiting mode or listening mode to receive the input.

4. *Communication and messages* - When a message is passed on from a system A to system B, if any and the message is lost or gets garbled the product should be tested to check how it responds to such user requesting him to wait for sometime until it recovers the connection.
5. *Meeting quality factors* - When two or more products are put together, there is an additional requirement of information exchange between them.

Regression Testing

What is Regression Testing?

- Regression testing is done to ensure that changes work as designed and do not have any unintended side-effects.
- Regression tests acknowledge that new fixes can cause new 'side effects' and can also cause some older defects to appear.
- The challenge in designing and running regression tests centers around designing the right tests to combat new defects introduced by the immunity acquired by a program against old test cases.
- Regression testing is done to ensure that enhancements or defect fixes made to the software works properly and does not affect the existing functionality.

- Regression testing is important in today's context since software is being released very often to keep up with the competition and increasing customer awareness.
- It is essential to make quick and frequent releases and also deliver stable software.
- Regression testing enables that any new feature introduced to the existing product does not adversely affect the current functionality.
- Regression testing follows *selective re-testing technique*.
- Whenever the defect fixes are done, a set of test cases that need to be run to verify the defect fixes are selected by the test team.
- An impact analysis is done to find out what areas may get impacted due to those defect fixes.

- Based on the impact analysis, some more test cases are selected to take care of the impacted areas.
- Since this testing technique focuses on reuse of existing test cases that have already been executed, the technique is called selective re-testing.

Types of Regression Testing

What is a build?

A build is an aggregation of all the defect fixes and features that are present in the product.

Two types of regression testing:

1. Regular regression testing.
2. Final regression testing.

Regular Regression Testing

- *Regular regression testing* is done between test cycles to ensure that defect fixes done and functionality that was working with the earlier test cycles continues to work normally.
- Regular regression testing can use more than one product build for the test cases to be executed.

Final Regression Testing

- A “final regression testing” is done to validate the final build before release.
- The Configuration Management(CM) engineer delivers the final build with the media and other contents exactly as it would go to the customer.
- The test cycle is conducted for a specific period of time, which is called *cook time*.
- Cook time is necessary as some defects show up only after running the product build for some time.
- The final regression test cycle is more critical than any other type or phase of testing, as the build which is being tested is the one which will go directly to the customer.

When to do Regression Testing?

Perform regression testing when:

1. A reasonable amount of initial testing is already carried out.
2. A good number of defects have been fixed.
3. Defect fixes that can produce side-effects are taken care of.
4. Regression testing may also be performed periodically, as a pro-active measure.

How to do Regression Testing?

A well-defined methodology for regression testing is very important as this among is the final type of testing that is normally performed just before release. If regression testing is not done right, it will enable the defects to seep through and may result in customers facing some serious issues not found by test teams.

1. Performing an initial “Smoke” or “Sanity” test
2. Understanding the criteria for selecting the test cases
3. Classifying the test cases into different priorities
4. A methodology for selecting test cases
5. Resetting the test cases for test execution
6. Concluding the results of a regression cycle

Initial 'Smoke' or 'Sanity' Test

Smoke testing consists of:

1. Identifying the basic functionality that a product must satisfy.
2. Designing test cases to ensure that these basic functionality work and packaging them into a smoke test suite.
3. Ensuring that every time a product is built, this suite is run successfully before anything else is run.
4. If this suite fails, escalating to the developers to identify the changes and perhaps change or roll back the changes to a state where the smoke test suite succeeds.

Criteria for Selecting the Test Cases

Criteria to select test cases for regression testing are as follows:

1. Include test cases that have produced the maximum defects in the past.
2. Include test cases for a functionality in which a change has been made.
3. Include test cases in which problems are reported.
4. Include test cases that test the basic functionality or the core features of the product which are mandatory requirements of the customer.
5. Include test cases that test the end-to-end behavior of the application or the product.
6. Include test cases to test the positive test conditions.
7. Includes the area which is highly visible to the users.

Classifying Test Cases

Priority-0

- These test cases can be called sanity test cases which check basic functionality and are run for accepting the build for further testing.
- They are also run when a product goes through a major change.
- These test cases deliver a very high project value to both to product development teams and to the customers.

Priority-1

- Uses the basic and normal setup and these test cases deliver high project value to both development team and to customers.

Priority-2

- These test cases deliver moderate project value.
- They are executed as part of the testing cycle and selected for regression testing on a need basis.

Methodology for Selecting Test Cases

This methodology takes into account the criticality and impact of defect fixes after test cases are classified into several priorities:

Case 1

- If the criticality and impact of the defect fixes are low, then it is enough that a test engineer selects a few test cases from test case database (TCDB), and executes them.
- These test cases can fall under any priority (0, 1, or 2).

Case 2

- If the criticality and the impact of the defect fixes are medium, then we need to execute all Priority-0 and Priority-1 test cases.
- If defect fixes need additional test cases from Priority-2, then those test cases can also be selected and used for regression testing.

- Selecting Priority-2 test cases in this case is desirable but not necessary.

Case 3

- If the criticality and impact of the defect fixes are high, then we need to execute all Priority-0, Priority-1 and a carefully selected subset of Priority-2 test cases.

Some Other Methodologies

Regress all

- All priority 0, 1, and 2 test cases are rerun.
- This means all the test cases in the regression test bed/suite are executed.

Priority based regression

- All priority 0, 1, and 2 test cases are run in order, based on the availability of time.
- Deciding when to stop the regression testing is based on the availability of time.

Regress changes

- Code changes are compared to the last cycle of testing and test cases are selected based on their impact on the code.

Random regression

- Random test cases are selected and executed.

Context based dynamic regression

- A few Priority-0 test cases are selected, and based on the context created by the analysis of those test cases after the execution and outcome, additional related cases are selected for continuing the regression testing.

Resetting the Test Cases for Regression Testing

Resetting of test cases is not done often, but if done, these points should be kept in mind:

1. When there is a major change in the product.
2. When there is a change in the build procedure which affects the product.
3. Large release cycle where some test cases were not executed for a long time.
4. When the prodcut is in the final regression test cycle with a few selected test cases.
5. Where there is a situation, that the expected results of the test cases could be quite different from the previous cycles.
6. The test cases relating to defect fixes and production problems need to be evaluated release after release.In case they are found to be working fine, they can be reset.

7. Whenever existing application functionality is removed, the related test cases can be reset.
8. Test cases that consistently produce a positive result can be removed.
9. Test cases relating to a few negative test conditions, which are not producing any defects, can be removed.

Concluding the Results of Regression Testing

Result 1

If the result of a particular test case was a pass using the previous builds and a fail in the current build, then regression has failed. A new build is required and the testing must start from scratch after resetting the test cases.

Result 2

If the result of a particular test case was a fail using the previous builds and a pass in the current build, then it is safe to assume the defect fixes worked.

Result 3

If the result of a particular test case was a fail using the previous builds and a fail in the current build and if there are no defect fixes for this particular test case, it may mean that the result of this test case should not be considered for the pass percentage. This may

also mean that such test cases should not be selected for regression.

Result 4

If the result of a particular test case is a fail using the previous builds but works with a documented workaround and if you are satisfied with the workaround, then it should considered as a pass for both the system test cycle and regression test cycle.

Result 5

If you are not satisfied with the workaround, then it should be considered as a fail for a system test cycle but may be considered as a pass for regression test cycle.

Best Practices in Regression Testing

1. Regression can be used for all types of releases.
2. Mapping defect identifiers with test cases improves regression Quality.
3. Create and execute regression test bed daily.
4. Ask your best test engineer to select the test cases.
5. Detect defects, and protect your product from defects and defect fixes.

Performance Testing

- The testing performed to evaluate the response time, throughput, and utilization of the system, to execute its required functions in comparison with different versions of the same product or a different competitive product is called performance testing.
- Performance testing is complex and expensive due to large resource requirements and the time it takes.
- A good number of defects that get uncovered during performance testing may require design and architecture change.
- Performance test cases are repetitive in nature.
- These test cases are normally executed repeatedly for different values of parameters, different load conditions, different configurations, and so on.

- Performance testing is a laborious process involving time and effort.
- Not all operations/business transactions can be included in performance testing.
- High priority test cases are implemented before others.

Factors Governing Performance Testing

1. Throughput - The capability of the system or the product in handling multiple transactions.
2. Response Time - Delay between the point of request and the first response from the product.
3. Network Latency - Delay in response times due to network.
4. Product Latency - Delay in response times due to product.
5. Tuning - The product performance is enhanced by setting different values to the parameters(variables) of the product, operating system, and other components.
6. Benchmarking - Competitive products are compared against each other to analyse the strengths and weaknesses.
7. Capacity Planning - Exercise to find out what resources and configurations are needed.

Performance Testing is done to ensure that a product:

- Processes the required number of transactions in any given interval (throughput).
- Is available and running under different load conditions (availability).
- Responds fast enough for different load conditions (response time).
- Delivers worthwhile return on investment for the resources - hardware and software and deciding what kind of resources are needed for the product for different load conditions (capacity planning).
- Is comparable to and better than that of the competitors for different parameters (competitive analysis and benchmarking).

Methodology For Performance Testing

Performance testing involves the steps:

1. Collecting requirements
2. Writing test cases
3. Automating performance test cases
4. Executing performance test cases
5. Analyzing performance test results
6. Performance tuning
7. Performance benchmarking
8. Recommending right configuration for the customers(Capacity Planning)

Collecting Requirements

- Two types of requirements - generic & specific.
- *Generic requirements* are those that are common across all products in the domain.
- *Specific requirements* are those that depend on implementation for a particular product and differ from one product to another in a given domain

Some resources for deriving performance requirements:

1. Performance compared to the previous release of the same product.
2. Performance compared to the competitive products.
3. Performance compared to absolute numbers derived from actual need.
4. Performance numbers derived from architecture and design.

Writing Test Cases

A test case for performance testing should have the following details:

1. List of operations or business transactions to be tested.
2. Steps for executing those operations/transactions.
3. List of product, OS parameters that impact the performance testing, and their values.
4. Loading pattern.
5. Resource and their configuration (network, hardware, software configurations).
6. The expected results (that is, expected response time, throughput, latency).
7. The product versions/competitive products to be compared with and related information such as their corresponding fields

Automating Performance Test Cases

Automation helps in performance testing because of these reasons:

1. Performance testing is repetitive.
2. Performance test cases cannot be effective without automation and mostly it is almost impossible to do performance testing without automation.
3. The results of performance testing need to be accurate, and manually calculating the response time, throughput, and so on can introduce inaccuracy because of human error.
4. Performance testing takes into account several factors. There are far too many permutations and combination of those factors and it will be difficult to remember all these and use them if the tests are done manually.
5. The analysis of performance results and failures needs to take into account related information such as resource utilization, log

files, trace files, and so on that are collected at regular intervals. It is impossible to do this testing and perform the book-keeping of all related information and analysis manually.

Thus, end-to-end automation is required for performance testing.

Executing Performance Test Cases

The most effort-consuming aspect in execution of performance test cases is usually data collection.

Data for the following points needs to be collected for test case execution:

1. Start and end time of test case execution.
2. Log and trace/audit files of the product and operating system, for future debugging and repeatability purposes.
3. Utilization of resources like CPU, memory, disk, network utilization etc on a periodic basis.
4. Configuration of all environmental factors like hardware, software, and other components.
5. The response time, throughput, latency etc as specified in the test case documentation at regular intervals.

6. What performance a product delivers for different configurations of hardware and network setup, is another aspect that needs to be included during execution.
7. This requirement mandates the need for repeating the tests for different configurations and is referred to as *configuration performance tests*.

Analyzing the Performance Test Results

Before analyzing the test results, some calculations of data and organization of the data are required, which are:

1. Calculating the mean of the performance test result data.
2. Calculating the standard deviation.
3. Removing the noise (noise removal) and re-plotting and recalculating the mean and standard deviation.
4. In terms of caching and other technologies implemented in the product, the data coming from the cache need to be differentiated from the data that gets processed by the product.
5. The majority of the server-client, Internet, and database applications store the data in a local high-speed buffer when a query is made.
6. This enables them to present the data quickly when the same request is made again. This is called *caching*.

7. Differentiating the performance data when the resources are available completely as against when some background activities were going on.

The analysis of performance data is carried out to conclude the following:

1. Whether performance of the product is consistent when tests are executed multiple times.
2. What performance can be expected for what type of configuration resources, both hardware and software.
3. What parameters impact performance and how they can be used to derive better performance.
4. What is the effect of scenarios involving several mix of operations for the performance factors.
5. What is the effect of product technologies such as caching on performance improvements.

6. Up to what load are the performance numbers acceptable.
7. What is the optimum throughput and response time of the product for a set of factors such as load, resources, and parameters.
8. What performance requirements are met and how the performance looks when compared to the previous version.
9. Sometime high-end configuration may not be available for performance testing. Use the current data and charts to extrapolate the expected results.

Performance Tuning

Analyzing performance data helps in narrowing down the list of parameters that really impact the
performance results and improving product performance.

The results of performance tuning are normally published in the form of a guide called the *performance tuning guide* for customers so that they can benefit from the results.

Two steps to get best results from performance tuning:

1. Tuning the product parameters.
2. Tuning the operating system and parameters.

Tuning Product Parameters

Important points to consider while tuning the product parameters:

1. Repeat the performance tests for different values of each parameter that impact performance, by keeping other parameters unchanged. This reveals the effect a parameter has on the performance of the product.
2. Sometimes when a particular parameter value is changed, it needs changes in other parameters, as two parameters can be dependent and share things in common. Repeat the performance tests for a group of parameters and their different values.
3. Repeat the performance tests for default values of all parameters, which are called *factory settings* tests.
4. Repeat the performance tests for low and high values of each parameter and combinations.

Tuning Operating System Parameters

Tuning the OS parameters is another step towards getting better performance. There are various sets of parameters provided by the operating system under different categories, which are:

1. File system related parameters, like number of open files permitted.
2. Disk management parameters, like simultaneous disk reads/writes.
3. Memory management parameters, like virtual memory page size and number of pages.
4. Processor management parameters, like enabling/disabling processors in multiprocessor environment.
5. Network parameters, like setting TCP/IP time out.

Performance Benchmarking

Steps involved in performance benchmarking are the following:

1. Identifying the transactions/scenarios and the test configuration.
2. Comparing the performance of different products.
3. Tuning the parameters of the products being compared fairly to deliver the best performance.
4. Publishing the results of performance benchmarking.

Capacity Planning

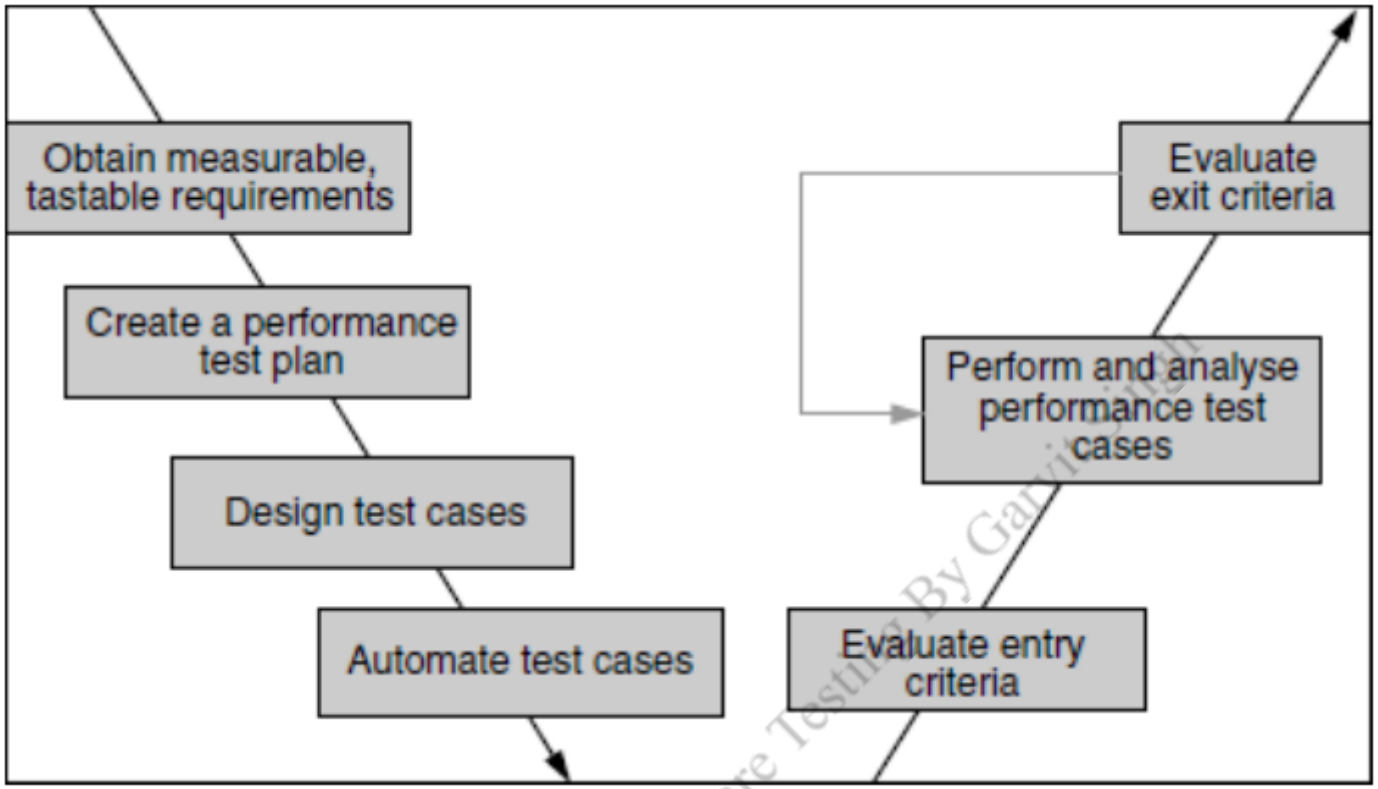
Capacity planning corresponding to short, medium and long-term requirements:

1. Minimum required configuration - Denotes that with anything less than this configuration, the product may not even work. Thus, configurations below the minimum required configuration are usually not supported
2. Typical configuration - Denotes that under that configuration the product will work fine for meeting the performance requirements of the required load pattern and can also handle a slight increase in the load pattern.
3. Special configuration - Denotes that capacity planning was done considering all future requirements.

Role of Load Balancing & High Availability in Capacity Planning

- Load balancing ensures that the multiple machines available are used equally to service the transactions.
- This ensures that by adding more machines, more load can be handled by the product.
- Machine clusters are used to ensure availability.
- In a cluster there are multiple machines with shared data so that in case one machine goes down, the transactions can be handled by another machine in the cluster.

Process For Performance Testing



1. Gather resource requirements.
2. Set up the test lab.
3. Assign all the responsibilities to multiple teams and their members.

4. Set up product traces, audits, internal and external traces, logs etc.
5. Decide the entry and exit criteria.

Ad Hoc Testing

- Testing done without using any formal testing technique and without any formal planning is called ad hoc testing.
- It is done to explore the undiscovered areas in the product by using intuition, previous experience in working with the product, expert knowledge of the platform or technology, and experience of testing a similar product.
- Ad hoc testing does not make use of any of the test case design techniques like equivalence partitioning, boundary value analysis, and so on.
- Ad hoc testing is done as a confidence measure just before the release, to ensure there are no areas that got missed out in testing.
- The test cases are not documented.
- Ad hoc testing can be done in all phases of testing.

Some Drawbacks of ad hoc testing & Possible Resolutions

Drawback	Possible Resolution
Difficult to ensure that the learnings learned in ad hoc testing are used in future.	Document ad hoc tests after test completion.
Large number of defects found in ad hoc testing.	Schedule a meeting to discuss defect impacts. Improve the test cases for planned testing.
Lack of comfort on coverage of ad hoc testing.	When producing test reports combine planned test and ad hoc test. Plan for additional planned test and ad hoc test cycles.
Difficult to track the exact steps.	Write detailed defect reports in a step-by-step manner. Document ad hoc tests after test execution.

Drawback	Possible Resolution
Lack of data for metrics analysis.	Plan the metrics collection for both planned tests and ad hoc tests.

Buddy Testing

- Uses the “buddy system” where two team members are identified as buddies.
- The buddies mutually help each other, with a common goal of identifying defects early and correcting them.
- A developer and a tester usually become buddies.

Pair Testing

- Pair testing is testing done by two testers working simultaneously on the same machine to find defects in the product.
- Two testers pair up to test a product's feature on the same machine.
- The objective of this exercise is to maximize the exchange of ideas between the two testers.
- When one person is executing the tests, the other person takes notes.
- They can swap roles of “tester” and “scribe” during a session. They can mutually decide on the modus operandi.

Exploratory Testing

- Exploratory testing is a technique used to find defects by exploring the product, covering more depth and breadth.
- Exploratory testing tries to do that with specific objectives, tasks, and plans.

There are several ways to perform exploratory testing:

Guesses

- Guesses are used to find the part of the program that is likely to have more errors.
- Previous experience on working with a similar product or software or technology helps in guessing.
- This is because the tester would have already faced situations to test a similar product or software.

- Those tests from guesses are used on the product to check for similar defects.

Architecture Diagrams & Use Cases

- Architecture diagrams depict the interactions and relationships between different components and modules.
- Use cases give an insight of the product's usage from the end user's perspective.
- A use case explains a set of business events, the input required, people involved in those events and the expected output.

Study of Past Defects

- Studying the defects reported in the previous releases helps in understanding of the error prone functionality/modules in a product development environment.

Error Handling

- Error handling is a portion of the code which prints appropriate messages or provides appropriate actions in case of failures.

Discussions

- Exploration may be planned based on the understanding of the system during project discussions or meetings.
- Information can be picked up during these meetings regarding implementation of different requirements for the product.

Questionnaires & Checklists

- Questions like “what, when, how, who and why” can provide leads to explore areas in the product.

Iterative Testing

- Iterative testing aims at testing the product for all requirements, irrespective of the phase they belong to in the spiral model.
- Iterative testing requires repetitive testing.
- Developers create unit test cases to ensure that the program developed goes through complete testing.
- Unit test cases are also generated from black box perspective to more completely test the product.
- After each iteration, unit test cases are added, edited, or deleted to keep up with the revised requirement for the current phase.
- Regression tests may be repeated at least every alternate iteration so that the current functionality is preserved.
- Automation helps in iterative testing.

Agile & Extreme Testing

- Agile and extreme (XP) models take the processes to the extreme to ensure that customer requirements are met in a timely manner.
- Agile and XP methodology emphasizes the involvement of the entire team, and their interactions with each other, to produce a workable software that can satisfy a given set of features.
- Software is delivered as small releases, with features being introduced in increments.
- Extreme programming and testing makes frequent releases and in a controlled way by involving customers.

Activities in XP Work Flow

1. Develop user stories.
2. Prepare acceptance tests.
3. Code.
4. Test.
5. Refactor.
6. Automate.
7. Delivery.

The rules that are followed in extreme programming and testing are as follows:

1. Cross Boundaries - Developers and testers cross boundaries to perform various roles.
2. Make Incremental Changes - Both product and process evolves in an incremental way.

3. Travel Light - Least overhead possible for development and testing.
4. Communicate - More focus on communication.
5. Write Tests Before Code - Unit tests and acceptance tests are written before the coding and testing activities respectively. All unit tests should run 100% all the time. Write code from test cases.
6. Make Frequent Small Releases.
7. Involve Customers All The Time.

Defect Seeding

- Defect seeding is a method of intentionally introducing defects into a product to check the rate of its detection and residual defects.
- Also known as *error seeding* or *bebugging*.
- Usually one group of members in the project injects the defects while another group tests to remove them.
- The purpose of this exercise is while finding the known seeded defects, the unseeded/unearthed defects may also be uncovered.
- Defects that are seeded are similar to real defects. Therefore, they are not very obvious and easy to detect.
- Total latent defects = (Defects seeded / Defects seeded found) * Original defects found. Latent defects are the number of defects which are yet to be found.

Precautions while defect seeding:

1. Care should be taken during the defect seeding process to ensure that all the seeded defects are removed before the release of the product.
2. The code should be written in such a way that the errors introduced can be identified easily. Minimum number of lines should be added to seed defects so that the effort involved in removal becomes reduced.
3. It is necessary to estimate the efforts required to clean up the seeded defects along with the effort for identification. Effort may also be needed to fix the real defects found due to the injection of some defects.

Real life scenarios and suitable ad hoc testing techniques

Scenario	**Most effective ad hoc technique**
Randomly test the product after all planned test cases are done.	Monkey Testing
Capture the programmatic errors early by developers and testers working together.	Buddy Testing
Test the new product/domain/technology.	Exploratory Testing
Leverage on the experience of senior testers and to exploit the ideas of newcomers.	Pair Testing
Deal with changing requirements.	Iterative Testing
Make frequent releases with customer involvement in product development.	Agile/Extreme Testing

A Checklist For Test Planning, Management, Execution and Reporting

Scope Related

- Have you identified the features to be tested?
- Have you identified the features not to be tested?
- Have you justified the reasons for the choice of features not to be tested and ascertained the impact from product management/senior management?
- Have you identified the new features in the release ?
- Have you included in the scope of testing areas which failures can be catastrophed?
- Have you included for testing those area that are defect prove or complex to test ?

Environment Related

- Do you have a software configuration management tool in place?
- Do you have a defect repository in place?
- Do you have a test case data base in place?
- Have you set up institutionalized procedures to update any of these?
- Have you identified the necessary hardware and software to design and run the tests?
- Have you identified the costs and other resource requirements of any test automation tools that may be needed?

Test Case Related

- Have you published naming conventions and other internal standards for designing, writing, and executing test cases?
- Are the test specifications documented adequately according to the above standards?
- Are the test specifications reviewed and approved by appropriate people?
- Are the test specifications baselined into the SCM repository?
- Are the test cases written according to specifications?
- Are the test cases reviewed and approved by appropriate people?
- Are the test cases baselined into the SCM repository?
- Is the traceability matrix updated once the test specifications/test cases are baselined?

Effort Estimation Related

- Have you translated the scope to a size estimate (for example, number of test cases)
- Have you arrived at an estimate of the effort required to design and construct the tests?
- Have you arrived at an effort required for repeated execution of the tests?
- Have the effort estimates been reviewed and approved by appropriate people?

Schedule Related

- Have you put together a schedule that utilizes all the resources available?
- Have you accounted for any parallelism constraints?
- Have you factored in the availability of releases from the development team?
- Have you accounted for any show stopper defects?
- Have you prioritized the tests in the event of any schedule crunch?
- Has the schedule been reviewed and approved by appropriate people?

Risks Related

- Have you identified the possible risks in the testing project?
- Have you quantified the likelihood and impact of these risks?
- Have you identified possible symptoms to catch the risks before they happen?
- Have you identified possible mitigation strategies for the risks?
- Have you taken care not to squeeze in all the testing activities towards then end of the development cycle?
- What mechanisms have you tried to distribute the testing activities throughout the life cycle (for example, doing an early test design like in the V model)?
- Have you prepared for the risk of idle time because of tests being suspended?

Execution Related

- Are you executing the tests as per plan? If there is any deviation, have you updated the plan?
- Did the test execution necessitate changing any test cases design? If so, is the TCDB kept current?
- Have you logged any defects that come up during testing in the defect repository?
- Have you updated the defect repository for any defects that are fixed in the current test cycle?
- Have you kept the traceability matrix current with the changes?

Completion Related

- Have you prepared a test summary report?
- Have you clearly documented the outstanding defects, along with their severity and impact?
- Have you put forth your recommendations for product release?

People Related

- Have you identified the number and skill levels of people required?
- Have you identified the gaps and prepared for training and skill upgradation?

Criteria Definition Related

- Have you defined the entry and exit criteria for the various test phases?
- Have you defined the suspension and resumption criteria for the various tests?

A Template For Test Plan

1. Introduction
    - Scope - What features are to be tested and what features will not be tested; what combinations of environment are to be tested and what not.
2. References
    - Gives references and links to documents such as requirement specifications, design specifications, program specifications, project plan, project estimates, test estimates, process documents, internal standards, external standards, and so on.
3. Test Methodology and Strategy/Approach
4. Test Criteria
    - Entry Criteria
    - Exit Criteria

    - Suspension Criteria
    - Resumption Criteria
5. Assumptions, Dependencies, and Risks
    - Assumptions
    - Dependencies
    - Risks and Risk Management Plans
6. Estimations
    - Size Estimate
    - Effort Estimate
    - Schedule Estimate
7. Test Deliverables and Milestones
8. Responsibilities
9. Resource Requirements
    - Hardware Resources

    - Software Resources
    - People Resources (Number of people, skills, duration, etc.)
    - Other Resources
10. Training Requirements
    - Details of Training Required
    - Possible Attendees
    - Any Constraints
11. Defect Logging and Tracking Process
12. Metrics Plan
13. Product Release Criteria

Software Test Automation

What is Test Automation?

Developing software to test the software is called test automation.

Addresses several problems like:

1. Automation saves time as software can execute test cases faster than human do.
2. Test automation can free the test engineers from mundane tasks and make them focus on more creative tasks.
3. Automated tests can be more reliable.
4. Automation helps in immediate testing.
5. Automation can protect an organization against attrition of test engineers.
6. Test automation opens up opportunities for better utilization of global resources.

7. Certain types of testing cannot be executed without automation.
8. Automation means end-to-end, not test execution alone.

Terms used in Automation

Test data generators

Automation should have scripts that produce test data to maximize coverage of permutations and combinations of inputs and expected output for result comparison. They are called *test data generators*.

Test Suite

When a set of test cases is combined and associated with a set of scenarios, they are called *test suite*. A Test suite is nothing but a set of test cases that are automated and scenarios that are associated with the test cases.

Skills Needed For Automation

First generation - Record & Playback

Skills for test case automation:

- Scripting languages.
- Record-playback tools usage.

Second generation - Data Driven

Skills for test case automation:

- Scripting languages.
- Programming languages.
- Knowledge of data generation techniques.
- Usage of product under test.

Third generation - Action Driven

Skills for test case automation:

- Scripting languages.
- Programming languages.
- Design and architecture of the product under test.
- Usage of the framework.

Skills for framework:

- Programming languages.
- Design & architecture skills for framework creation.
- Generic test requirements for multiple products.

Scope of Automation : What to Automate?

1. Certain types of testing are more suited for automation, which are:
    - Stress, reliability, scalability & performance testing.
    - Regression tests.
    - Functional tests.
2. Automate the areas which are less prone to change in future.
3. Automate the tests which pertain to standards like compliance, legal requirements etc.
4. Management commitment in Automation
    - ROI is to be considered.
    - Test cases that are easy to automate should be automated first.

Design & Architecture For Automation

Components in test automation:

1. External modules.
2. Scenario & Configuration file modules.
3. Test cases & Test framework modules.
4. Tools & Results modules.
5. Report generator & Report/Metrics modules.

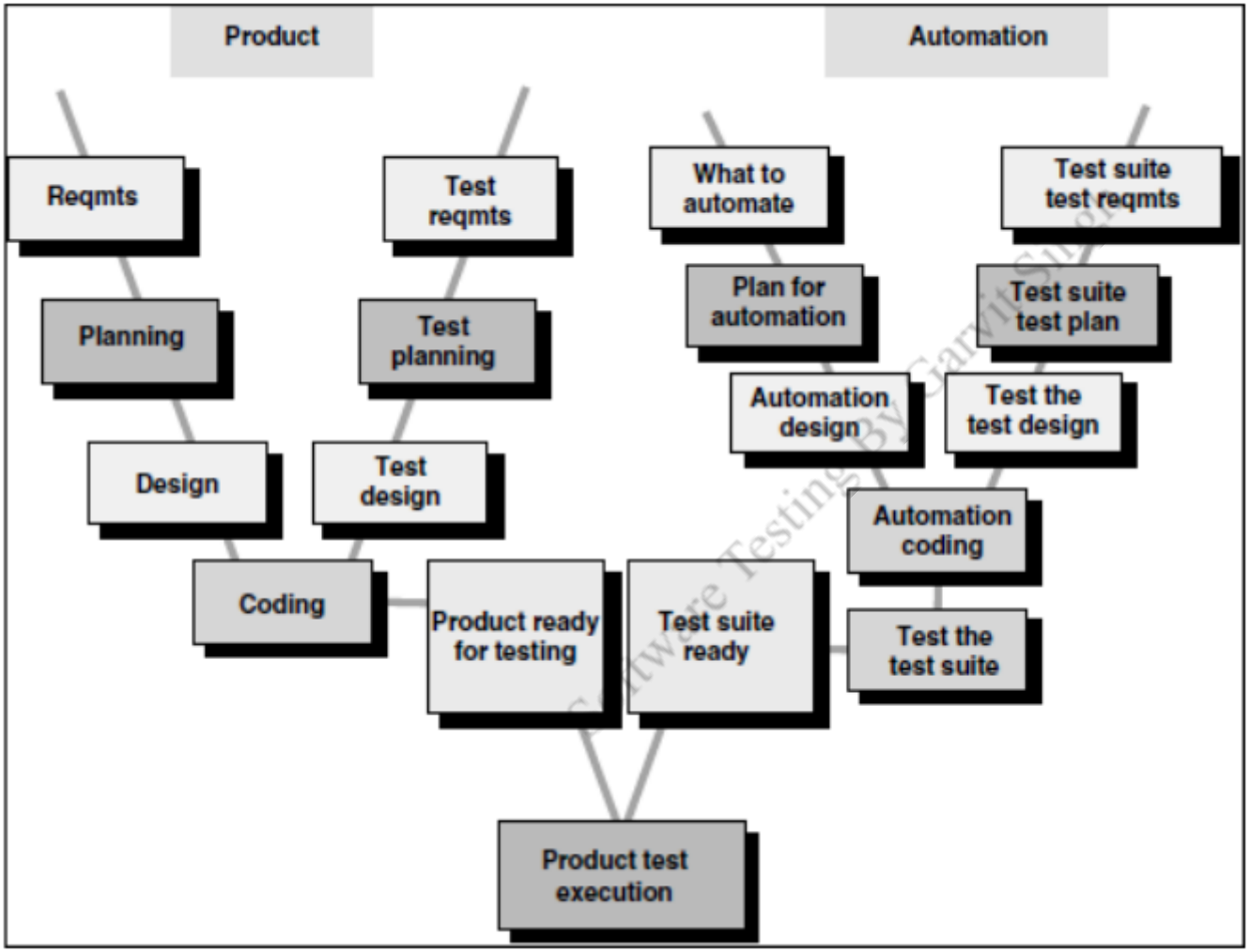
Generic Requirements For Test Tool/Framework

1. No hard coding in the test suite.
2. Test case/suite expandability.
3. Reuse of code for different types of testing, test cases.
4. Automatic setup and cleanup.
5. Independent test cases.
6. Test case dependency.
7. Insulating test cases during execution.
8. Coding standards and directory structure.
9. Selective execution of test cases.
10. Random execution of test cases.
11. Parallel execution of test cases.
12. Looping the test cases.
13. Grouping of test scenarios.

14. Test case execution based on previous results.
15. Remote execution of test cases.
16. Automatic archival of test data.
17. Reporting scheme.
18. Independent of languages.
19. Portability to different platforms.

Process Model For Automation

The W Model for phases involved in automation.



Selecting a Test Tool

Selecting a test tool is an important aspect of test automation for many reasons:

1. Free tools are not well supported and get phased out soon.
2. Developing in-house tools takes time.
3. Test tools sold by vendors are expensive.
4. Test tools require strong training.
5. Test tools generally do not meet all the requirements for automation.
6. Not all test tools run on all platforms.
7. Testing tools have very high entry, maintenance, and exit costs and hence careful selection is required.

Criteria For Selecting Test Tools

1. Meeting Requirements.
2. Technology Expectations.
3. Training skills.
4. Management aspects.

Issues in Selecting a Testing Tool

1. Meeting Requirements
    - Checking wether the tools meet requirements, involves effort and money.
    - Test tools are not fully compatible with products.
    - Test tools are not tested with the same seriousness as products for new requirements.
    - Difficult to isolate problems of product and test suite. Change in product causes test suite to be changed.

2. Technology Expectations
    - Extending the test tool is difficult.
    - Requires instrumented code to be removed for certain tests.
    - Test tools are not cross platform.
3. Training skills
    - Lack of trainers for test tools.
    - Test tools requires people to learn new langauge/scripts.
4. Management aspects.
    - Test tools require system upgrades.
    - Migration to other test tools difficult.
    - Deploying tool requires huge planning and effort.

Steps for Tool Selection & Deployment

1. Identify your test suite requirements among the generic requirements discussed. Add other requirements, if any.
2. Collect the experiences of other organizations which used similar test tools.
3. Keep a checklist of questions to be asked to the vendors on cost/effort/support/anything else.
4. Identify list of tools that meet the above requirements.
5. Give priority for the tool which is available with source code.
6. Evaluate and shortlist one tool or set of tools and train all test developers on the tool.
7. Deploy the tool across test teams after training all potential users of the tool.

Conclusion

- A good automation test suite can help in 24x7 test execution, saving effort and time.
- The quality requirements for the test suite are equal to or more stringent than that of the product.
- At times automation is more complex than product development. Plan to have your best development and test engineers in the automation team.
- Selecting test tools without proper analysis will result in expensive test tools gathering dust on the shelf. This is termed as shelf ware.
- Automation makes life easier for testers for better reproduction of test results, coverage and, of course, reduction in effort as a side product.

Further Exploration

More types of software testing to explore:

1. Accessibility & Usability Testing
2. Internationalization Testing
3. Testing Object Oriented systems.
4. Acceptance Testing