SOFTWARE TESTING STRATEGIES

A STRATEGIC APPROACH TO SOFTWARE TESTING

A number of software testing strategies have been proposed in the literature. All provide you with a template for testing and all have the following generic characteristics:

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as **verification** and **validation** (V&V). ***Verification*** refers to the set of tasks that ensure that software correctly implements a specific function. ***Validation*** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Boehm states this another way:

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

Verification and validation includes a wide array of **SQA** activities: **technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing**.

Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software.

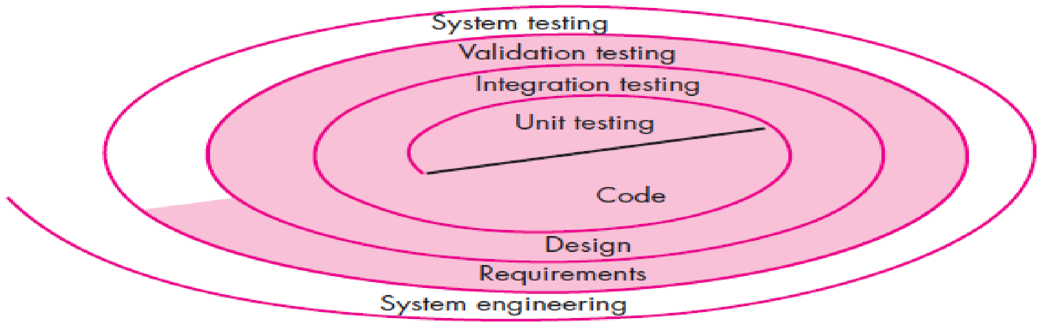
The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.

The role of an ***independent test group*** **(ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in following figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counter clockwise) along streamlines that decrease the level of abstraction on each turn.

Fig : Testing Strategy



A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code. Testing progresses by moving outward along the spiral to ***integration testing***, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter ***validation testing***, where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at ***system testing***, where the software and other system elements are tested as a whole.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of **four** steps that are implemented sequentially. The steps are shown in following figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete software package. ***Integration testing*** addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

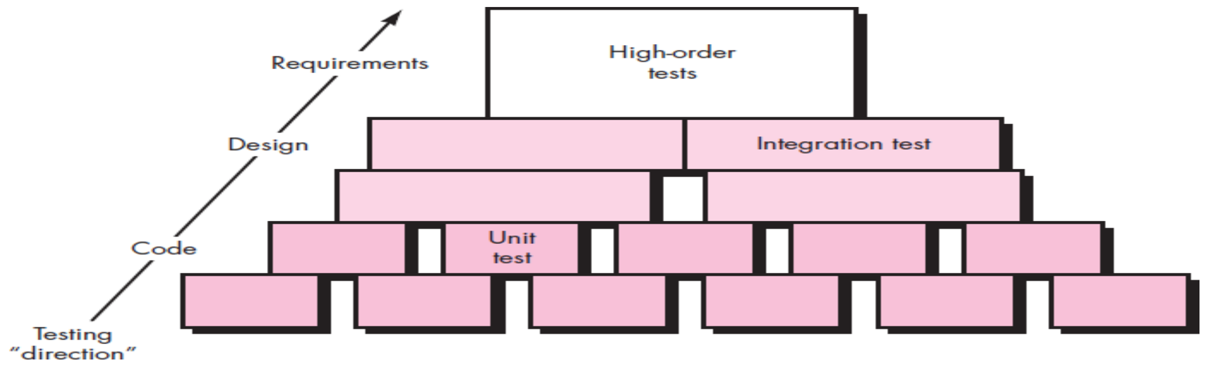


Fig : Software testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). ***System testing*** verifies that all elements mesh properly and that overall system function/performance is achieved.

Criteria for Completion of Testing

"When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

One response to the question is: "You're never done testing; the burden simply shifts from you (the software engineer) to the end user." Every time the user executes a computer program, the program is being tested.

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The ***clean room software engineering*** approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

. By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"

<u>STRATEGIC ISSUES</u>

Tom Gilb argues that a software testing strategy will succeed when software testers:

- ***Specify product requirements in a quantifiable manner long before testing commences***. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability.. These should be specified in a way that is measurable so that testing results are unambiguous.

- ***State testing objectives explicitly***. The specific objectives of testing should be stated in measurable terms.
- ***Understand the users of the software and develop a profile for each user category***. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
- ***Develop a testing plan that emphasizes "rapid cycle testing."*** Gilb recommends that a software team "learn to test in rapid cycles The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
- ***Build "robust" software that is designed to test itself***. Software should be designed in a manner that uses anti bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
- ***Use effective technical reviews as a filter prior to testing***. Technical reviews can be as effective as testing in uncovering errors.
- ***Conduct technical reviews to assess the test strategy and test cases themselves***. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- ***Develop a continuous improvement approach for the testing process***. The test strategy should be measured. The metrics

collected during testing should be used as part of a statistical process control approach for software testing.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

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

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Unit-test considerations. Unit tests are illustrated schematically in following figure. The module **interface** is tested to ensure that information properly flows into and out of the program unit under test. **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. All **independent paths** through the control structure are exercised to ensure that all statements in a module have been executed at least once. **Boundary conditions** are tested to ensure that the module operates

properly at boundaries established to limit or restrict processing.

And finally, all **error-handling paths** are tested.

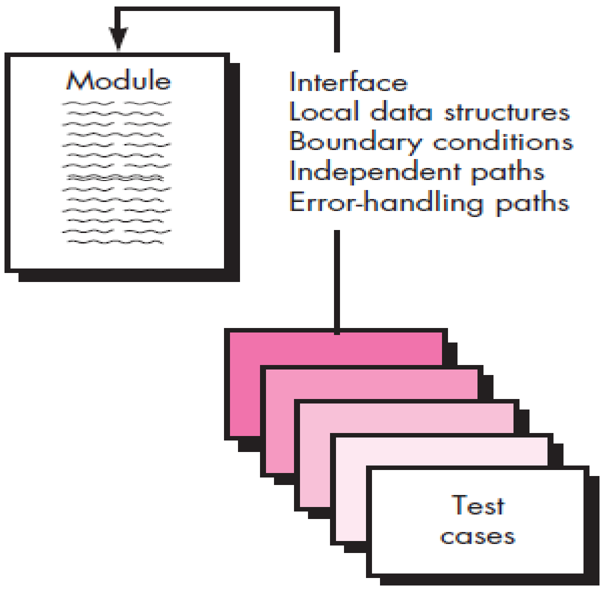


Fig : Unit Test

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *i* passes is invoked, when the maximum or minimum allowable value is encountered.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach ***antibugging***.

Among the potential errors that should be tested when error handling is evaluated are: (1) error description is unintelligible, (2) error noted does not correspond to error encountered, (3) error condition causes system intervention prior to error handling, (4) exception-condition processing is incorrect, or (5) error description does not provide enough information to assist in the location of the cause of the error.

Unit-test procedures. Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated.

The unit test environment is illustrated in following figure.. In most applications a ***driver*** is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. ***Stubs*** serve to replace modules that are subordinate (invoked by) the component to be tested.

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

Integration Testing

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

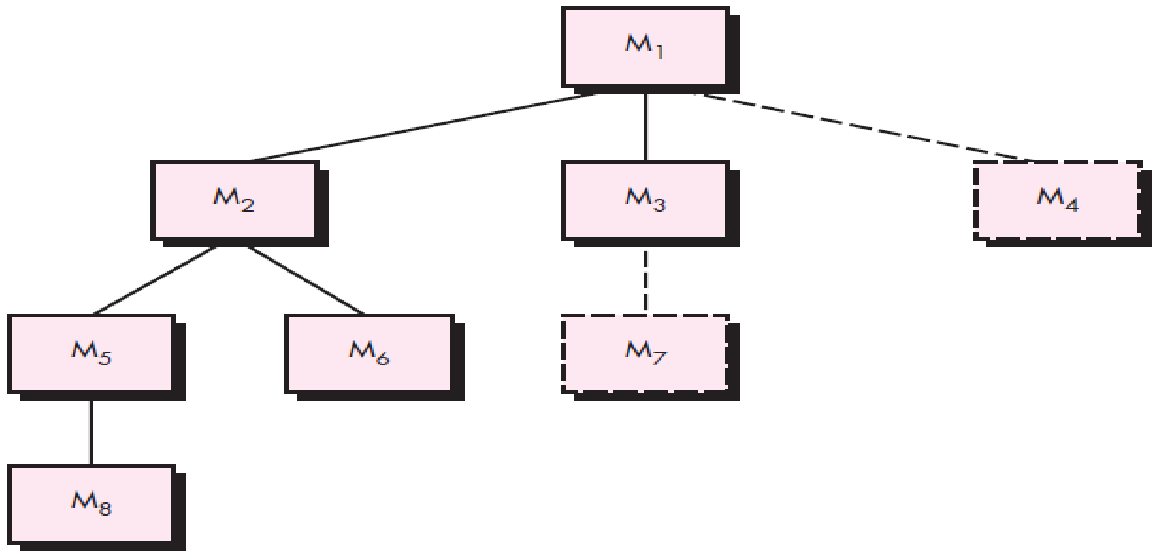
There is often a tendency to attempt non incremental integration; that is, to construct the program using a "**big bang**" approach. All components are combined in advance. The entire program is tested as a **whole**. If a set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. There are **two** different incremental integration strategies :

Top-down integration. *Top-down integration testing* is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a **depth-first or breadth-first** manner. Referring to the following figure, ***depth-first integration*** integrates all components on a major control path of the program structure. For example, selectingthe left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and right-hand control paths are built. ***Breadth-first integration*** incorporates all

components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

Fig : Top-down integration



The integration process is performed in a series of **five** steps:

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

Bottom-up integration. *Bottom-up integration testing,* as its name implies, begins construction and testing with ***atomic modules*** (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated. A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

Integration follows the pattern illustrated in following figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M*a*. Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3 for cluster 3 is removed prior to integration with module M*b*. Both M*a* and M*b* will ultimately be integrated with component M*c*, and so forth.

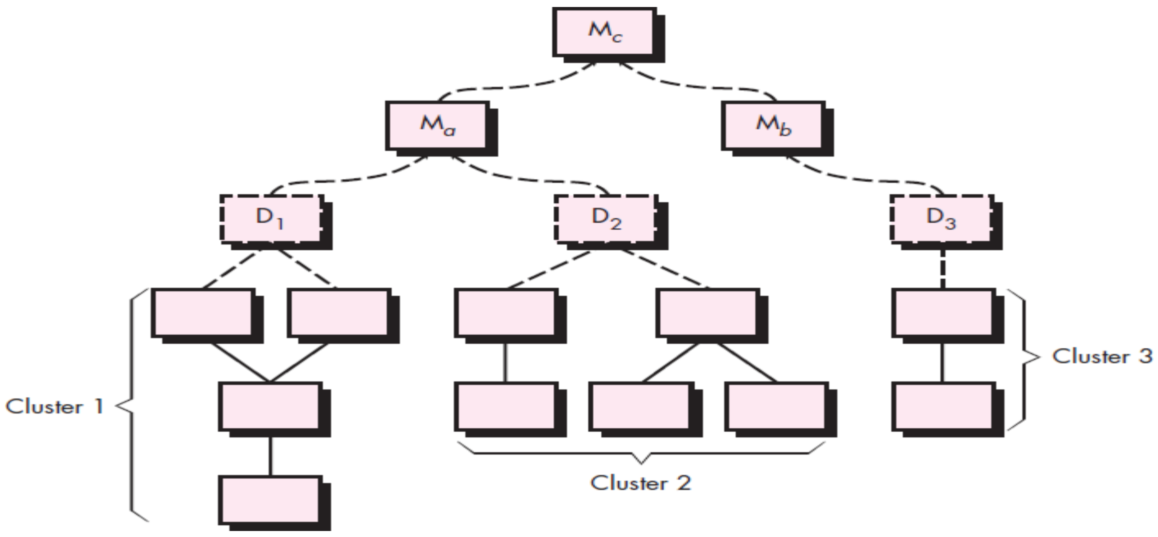


Fig : Bottom-up integration

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

Regression testing. R*egression testing* is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression testing helps to ensure that changes do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by reexecuting a subset of all test cases or using automated **capture/playback** tools.

Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison. The ***regression test suite*** (the subset of tests to be executed) contains **three** different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large.

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

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "showstopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

McConnell describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of **benefits** when it is applied on complex, time critical software projects:

- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ***The quality of the end product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- ***Error diagnosis and correction are simplified***. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess***. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

Software testing methodologies are the various strategies or approaches used to test an application to ensure it behaves and looks as expected. These encompass everything from front to back-end testing, including unit and system testing. This article is designed to highlight the myriad of testing techniques used by quality assurance professionals.

Functional vs. Non-functional Testing

The goal of utilizing numerous testing methodologies in your development process is to make sure your software can successfully operate in multiple environments and across different platforms. These can typically be broken down between functional and non-functional testing. Functional testing involves testing the application against the business requirements. It incorporates all test types designed to guarantee each part of a piece of software behaves as expected by using uses cases provided by the design team or business analyst. These testing methods are usually conducted in order and include:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Non-functional testing methods incorporate all test types focused on the operational aspects of a piece of software. These include:

- Performance testing
- Security testing
- Usability testing
- Compatibility testing

The key to releasing high quality software that can be easily adopted by your end users is to build a robust testing framework that implements both functional and non-functional software testing methodologies.

Unit Testing

Unit testing is the first level of testing and is often performed by the developers themselves. It is the process of ensuring individual components of a piece of software at the code level are functional and work as they were designed to. Developers in a test-driven environment will typically write and run the tests prior to the software or feature being passed over to the test team. Unit testing can be conducted manually, but automating the process will speed up delivery cycles and expand test coverage. Unit testing will also make debugging easier because finding issues earlier means they take less time to fix than if they were discovered later in the testing process. TestLeft is a tool that allows advanced testers and developers to shift left with the fastest test automation tool embedded in any IDE.

Integration Testing

After each unit is thoroughly tested, it is integrated with other units to create modules or components that are designed to perform specific tasks or activities. These are then tested as group through integration testing to ensure whole segments of an application behave as expected (i.e, the interactions between units are seamless). These tests are often framed by user scenarios, such as logging into an application or opening files. Integrated tests can be conducted by either developers or independent testers and are usually comprised of a combination of automated functional and manual tests.

System Testing

System testing is a black box testing method used to evaluate the completed and integrated system, as a whole, to ensure it meets specified requirements. The functionality of the software is tested from end-to-end and is typically conducted by a separate testing team than the development team before the product is pushed into production.

Acceptance Testing

Acceptance testing is the last phase of functional testing and is used to assess whether or not the final piece of software is ready for delivery. It involves ensuring that the product is in compliance with all of the original business criteria and that it meets the end user's needs. This requires the product be tested both internally and externally, meaning you'll need to get it into the hands of your end users for beta testing along with those of your QA team. Beta testing is key to getting real feedback from potential customers and can address any final usability concerns.

Performance Testing

Performance testing is a non-functional testing technique used to determine how an application will behave under various conditions. The goal is to test its responsiveness and stability in real user situations. Performance testing can be broken down into four types:

- **Load testing** is the process of putting increasing amounts of simulated demand on your software, application, or website to verify whether or not it can handle what it's designed to handle.
- **Stress testing** takes this a step further and is used to gauge how your software will respond at or beyond its peak load. The goal of stress testing is to overload the application on purpose until it breaks by applying both realistic and unrealistic load scenarios. With stress testing, you'll be able to find the failure point of your piece of software.
- **Endurance testing,** also known as soak testing, is used to analyze the behavior of an application under a specific amount of simulated load over longer amounts of time. The goal is to understand how your system will behave under sustained use, making it a longer process than load or stress testing (which are designed to end after a few hours). A critical piece of endurance testing is that it helps uncover memory leaks.

- **Spike testing** is a type of load test used to determine how your software will respond to substantially larger bursts of concurrent user or system activity over varying amounts of time. Ideally, this will help you understand what will happen when the load is suddenly and drastically increased.

Security Testing

With the rise of cloud-based testing platforms and cyber attacks, there is a growing concern and need for the security of data being used and stored in software. Security testing is a non-functional software testing technique used to determine if the information and data in a system is protected. The goal is to purposefully find loopholes and security risks in the system that could result in unauthorized access to or the loss of information by probing the application for weaknesses. There are multiple types of this testing method, each of which aimed at verifying six basic principles of security:

1. Integrity
2. Confidentiality
3. Authentication
4. Authorization
5. Availability
6. Non-repudiation

Usability Testing

Usability testing is a testing method that measures an application's ease-of-use from the end-user perspective and is often performed during the system or acceptance testing stages. The goal is to determine whether or not the visible design and aesthetics of an application meet the intended workflow for various processes, such as logging into an application. Usability testing is a great way for teams to review separate functions, or the system as a whole, is intuitive to use.

Compatibility Testing

Compatibility testing is used to gauge how an application or piece of software will work in different environments. It is used to check that your product is compatible with multiple operating systems, platforms, browsers, or resolution configurations. The goal is to ensure that your software's functionality is consistently supported across any environment you expect your end users to be using.

Testing With TestComplete

TestComplete is our robust automated GUI testing tool that excels in compatibility and integration testing. It helps QA teams create and run tests across desktop, mobile, and web applications – enabling testing professionals to speed up delivery cycles and improve software quality. Testcomplete comes with built-in support for various test environments, integrations to performance testing tools, as well as support for developer friendly SCMs, allowing you to seamlessness integrate it into your development process. Using TestComplete will enable you to build a robust testing framework that utilizes the broad spectrum of available software testing methodologies.

Debugging in Software Testing

On successful culmination of software testing, debugging is performed. Debugging is defined as a process of analyzing and removing the error. It is considered necessary in most of the newly developed software or hardware and in commercial products/ personal application programs. For complex products, debugging is done at all the levels of the testing.

Debugging is considered to be a complex and time-consuming process since it attempts to remove errors at all the levels of testing. To perform debugging, debugger (debugging tool) is used to reproduce the conditions in which failure occurred, examine the program state, and locate the cause. With the help of debugger, programmers trace the program execution step by step (evaluating the value of variables) and halt the execution wherever required to reset the program variables. Note that some programming language packages include a debugger for checking the code for errors while it is being written.

Some guidelines that are followed while performing debugging are discussed here.

- Debugging is the process of solving a problem. Hence, individuals involved in debugging should understand all the causes of an error before starting with debugging.
- No experimentation should be done while performing debugging. The experimental changes instead of removing errors often increase the problem by adding new errors in it.
- When there is an error in one segment of a program, there is a high possibility that some other errors also exist in the program. Hence, if an error is found in one segment of a program, rest of the program should be properly examined for errors.
- It should be ensured that the new code added in a program to fix errors is correct and is not introducing any new error in it. Thus, to verify the correctness of a new code and to ensure that no new errors are introduced, regression testing should be performed.

We'll be covering the following topics in this tutorial:

- The Debugging Process
- Debugging Strategies

The Debugging Process

During debugging, errors are encountered that range from less damaging (like input of an incorrect function) to catastrophic (like system failure, which lead to economic or physical damage). Note that with the increase in number of errors, the amount of effort to find their causes also increases.

Once errors are identified in a software system, to debug the problem, a number of steps are followed, which are listed below.

- **Defect confirmation/identification:** A problem is identified in a system and a defect report is created. A software engineer maintains and analyzes this error report and finds solutions to the following questions.
  - Does a .defect exist in the system?
  - Can the defect be reproduced?
  - What is the expected/desired behavior of the system?
  - What is the actual behavior?
- **Defect analysis:** If the defect is genuine, the next step is to understand the root cause of the problem. Generally, engineers debug by starting a debugging tool (debugger) and they try to understand the root cause of the problem by following a step-by-step execution of the program.
- **Defect resolution:** Once the root cause of a problem is identified, the error can be resolved by making an appropriate change to the system by fixing the problem.

When the debugging process ends, the software is retested to ensure that no errors are left undetected. Moreover, it checks that no new errors are introduced in the software while making some changes to it during the debugging process.

Debugging Strategies

As debugging is a difficult and time-consuming task, it is essential to develop a proper debugging strategy. This strategy helps in performing the process of debugging easily and efficiently. The commonly-used debugging strategies are debugging by brute force, induction strategy, deduction strategy, backtracking strategy, and debugging by testing.

Brute force method of debugging is the most commonly used but least efficient method. It is generally used when all other available methods fail. Here, debugging is done by taking memory (or storage) dumps. Actually, the program is loaded with the output statements that produce a large amount of information including the intermediate values. Analyzing this information may help to identify the errors cause. However, using a memory dump for finding errors requires analyzing huge amount of information or irrelevant data leading to waste of time and effort.

This strategy is a 'disciplined thought process' where errors can be debugged by moving outwards from the particulars to the whole. This strategy assumes that once the symptoms of the errors are identified, and the relationships between them are established, the errors can be easily detected by just looking at the symptoms and the relationships. To perform induction strategy, a number of steps are followed, which are listed below.

1. Locating relevant data: All the information about a program is collected to identify the functions, which are executed correctly and incorrectly.

2. Organizing data: The collected data is organized according to importance. The data can consist of possible symptoms of errors, their location in the program, the time at which the symptoms occur during the execution of the program and the effect of these symptoms on the program.

3. Devising hypothesis: The relationships among the symptoms are studied and a hypothesis is devised that provides the hints about the possible causes of errors.

4. Proving hypothesis: In the final step, the hypothesis needs to be proved. It is done by comparing the data in the hypothesis with the original data to ensure that the hypothesis explains the existence of hints completely. In case, the hypothesis is unable to explain the existence of hints, it is either incomplete or contains multiple errors in it.

Deduction Strategy

In this strategy, first step is to identify all the possible causes and then using the data each cause is analyzed and eliminated if it is found invalid. Note that as in induction strategy, deduction strategy is also based on some assumptions. To use this strategy following steps are followed.

1. Identifying the possible causes or hypotheses: A list of all the possible causes of errors is formed. Using this list, the available data can be easily structured and analyzed.

2. Eliminating possible causes using the data: The list is examined to recognize the most probable cause of errors and the rest of the causes are deleted.

3. Refining the hypothesis: By analyzing the possible causes one by one and looking for contradiction leads to elimination of invalid causes. This results in a refined hypothesis containing few specific possible causes.

4. Proving the hypothesis: This step is similar to the fourth step in induction method.

Backtracking Strategy

This method is effectively used for locating errors in small programs. According to this strategy, when an error has occurred, one needs to start tracing the program backward one step at a time evaluating the values of all variables until the cause of error is found. This strategy is useful but in a large program with many thousands lines of code, the number of backward paths increases and becomes unmanageably large.

Debugging by Testing

This debugging method can be used in conjunction with debugging by induction and debugging by deduction methods. Additional test cases are designed that help in obtaining information to devise and prove a hypothesis in induction method and to eliminate the invalid causes and refine the hypothesis in deduction method. Note that the test cases used in debugging are different from the test cases used in testing process. Here, the test cases are specifically designed to explore the internal program state.

System Documentation (Manuals)

The formal description of a mechanical system or a technical process is known as its documentation. Documentation takes the form of technical and user manuals that accompany various technological objects, materials, and processes. Electronic hardware, computers, chemicals, automobiles all are accompanied by descriptive documentation in the form of manuals.

Two kinds of documentation are required when products are sold: technical documentation and user documentation.

- Technical documentation is a physical description of a system, device, material, or process. This technical description is used by expert users and designers as guidelines to maintain and modify various elements of the system. Good examples of technical documentation are the wiring diagrams that accompany electrical hardware, the computer code that accompanies many programmed instruments, and the detailed pharmaceutical descriptions that accompany various medicines. These descriptions are all intended for experts, who must make informed decisions about the installation, capabilities, modifications, and applications of the technology in question.

- User documentation includes the product guidelines addressed to the general user who needs to know basic requirements for getting the best use out of the technology. User documentation includes the manuals for product use, assembly, maintenance, operations, and repair.

Manuals should be prepared with the object of making the information quickly available to the expert or general reader. Keep the audience foremost in mind. In manuals, ease of finding and reading information is a priority. Hence manuals should contain these components:

Easy-to-use locating elements, such as tables of contents, indexes, and page headers

Useful big-picture and close-up diagrams that make it easy for the reader to become familiar with the technology

Effective warnings against personal harm and cautions against harm to the equipment

Page designs that lay the material out effectively, with effective labeling, chunking, and white spaces

Clear manual arrangements into sections and chapters organized around important tasks

Simple, economical style, pared down to just what is needed

Consistent, well-designed terminology that keeps the reader focused on the task Effective packages, with binders and covers designed for the working space in which the manual will be used.

Documentation Review

Documentation consists of the organization's business documents used to support security and accounting events. The strength of documentation is that it is prevalent and available at a low cost. Documents can be internal or externally generated. Internal documents provide less reliable evidence than external ones, particularly if the client's internal control is suspect. Documents that are external and have been prepared by qualified individuals such as attorneys or insurance brokers provide additional reliability. The use of documentation in support of a client's transactions is called vouching.

Documentation review criteria include three areas of focus:

1.*Review* is used for the "generalized" level of rigor, that is, a high-level examination looking for required content and for any obvious errors, omissions, or inconsistencies.

2.*Study* is used for the "focused" level of rigor, that is, an examination that includes the intent of "review" and adds a more in-depth examination for greater evidence to support a determination of whether the document has the required content and is free of errors, omissions, and inconsistencies.

3.*Analyze* is used for the "detailed" level of rigor, that is, an examination that includes the intent of both "review" and "study," adding a thorough and detailed analysis for significant grounds for confidence in the determination of whether required content is present and the document is correct, complete, and consistent.

The Software Training Basics

There is no doubt that software training has become a trend. But what makes software training so popular? Be it just a demo or a simple PowerPoint presentation, there is always some form of training needed in order to transfer knowledge to your learners.

In simple words, software training is here to stay as long as organizations are driven by the need to save time and cost by digitizing the tasks for themselves and all their stakeholders ranging from customers to vendors to business partners. Therefore, recent years have seen a boom in software training that use various technologies.

So how does software training develop? What is the science that determines its stages? Where and when does it need intervention in terms of training and knowledge transfer?

These are thoughts that must be pondered upon...

The Systems Development Life Cycle (SDLC)

Behind the software that you use, be it for banking or for paying bills, there is a software development story that is as fascinating as it is complex.

Universally, software development organizations follow a systemic development model called the Systems Development Life Cycle (SDLC). The aim of the SDLC life cycle is to create a high quality training system keeping in mind client requirements and technological possibilities.

From agile to iterative to sequential, SDLC models follow different sequences of development based on the requirements of the client and the project.

This methodology helps to plan and control the development of an information system, dividing the entire process into distinct stages.

The 5 Stages of Software Development Life Cycle

Therefore, a typical software development lifecycle goes through the following standard stages:

1. **Requirement Analysis**
   In this phase the requirements for the project are defined by the Business Analysts. Therefore, the team identifies why the application needs to be built, who will be the system users, what are the gaps, how the new application can bridge the gaps, infrastructure requirements etc. At the end of this phase the functional specification documents, gap analysis document etc. are created.
2. **Design and Development**
   During this stage database modeling, software structuring, interface design, prototype development etc. takes place. At the end of this phase design documents such as the General Design Document (GDD)/Detailed Design Document (DDD) and also the codes are developed.
3. **Quality Assurance and Testing**
   This is the stage for installing build, system testing, bug fixing, User Acceptance Testing (UAT) and for preparing testing reports. At the

end of this phase a stable application with minimized errors is ready for deployment.

4. **Implementation and Deployment**
   During this stage trainings are conducted and the software application is deployed at pilot sites initially followed by complete roll out. At the end of this phase a fully functional, stable application is ready for use.

5. **Maintenance and Support**
   During this period knowledge transfer, change requests, if any, impact analysis and all pending documentation is completed. At the end of this phase, the vendor team hands over all the elements of the project including codes, documents and software licenses to the client.

Mapping Training and Systems Development Life Cycle Functions

Now let's find out what creates the need for trainings during software development? Here are some answers:

When the system is tested by external testers, such as client representatives, the testers will need to understand the features and functionalities of the application. Therefore training may be needed before testing activities, especially User Acceptance Testing (UAT). Sometimes clients opt for a walkthrough and avoid going in for a full fledged training for testers.

If a vendor is developing the system, it will finally be handed over to the client, who will be in charge of maintaining it. This implies the development team from the client side will need training/knowledge transfer. Trainings for the developers/technical staff are classified as technical trainings

The database of the system has to be administered by the Database Administrator (DBA) group of the client, after the vendor leaves. This implies that this group will need DBA trainings.

The system will eventually be used by a set of people from the list of client stakeholders. This implies that the users of the system will need training.

The client may want to carry forward the training in other location/users. This implies that the trainers from the client side will require training. Such trainings are classified as ‘Train-the-Trainer’ (TTT) trainings.

Therefore, software applications will need the following types of trainings at the corresponding stages to be successfully implemented:

- Pre-UAT trainings
- User trainings
- Technical training/knowledge transfer
- DBA training/knowledge transfer

The ADDIE Life Cycle

We have already seen the phases in the SDLC. Training also has phases that are as distinct as the SDLC.

Trainings typically follow the ADDIE model of development. The ADDIE model consists of five distinct phases:

- Analysis
- Design
- Development

- Implementation
- Evaluation

The following sections briefly explain the five phases:

1. Analysis Phase

During Analysis phase the Instructional Designer (ID) finds answers to the following questions:

1. What is the requirement for the training?
2. Who is the audience?
3. What are the audience characteristics?
4. What are their current skills, knowledge and abilities?
5. What are the desired skills, knowledge, and abilities that need to be achieved through training?
6. What is the training gap, if any?
7. What are the preferred modes of training delivery?
8. What are the timelines for the Project?
9. What is the prerequisite knowledge that the learners need to have for the training?

In this phase, all the information related to the training, is gathered. A Needs Analysis Report (NAR) is prepared with all the information that has been gathered.

2. Design Phase

After the ID has gathered all the necessary information regarding the training that needs to be developed, the next stage is to plan for all the activities. Therefore, in the Design phase:

1. Templates are created
2. Style Guide is prepared
3. Prototypes are prepared, if required
4. Learning strategy is developed
5. All documentation such as General Design Document (GDD) and Use Cases are analyzed
6. Learning objectives are derived
7. Course duration is estimated
8. Course curriculum is developed

At the end of this phase, all the planning and preparations for the proposed trainings are completed and the stage is set for development.

3. Development Phase

This is the stage for the development of the actual training material. All the logistics regarding training delivery are also finalized at this stage. Therefore, at the end of the Development phase, the stage is set for the actual training delivery.

Implementation Phase

This is the stage for the actual delivery of the classroom Trainings and online Trainings.

Evaluation Phase

This is the final phase of the Training. During this phase the following types of evaluation are performed:

1. Trainers evaluate the learners informally through ‘check the understanding’ questions or other interactive activities.

2. Learners evaluate the effectiveness of the training through feedback forms or other identified methods.
3. Lessons learned are documented.

The ADDIE model ensures that all training activities are well organized and chunked into distinct phases that prelude or follow each other in a logical sequence.

This approach ensures that all the activities are planned well ahead of time and all stakeholders are aware of what to expect and when.

Mapping Training and Software Development Life Cycle Phases

So how do the phases of ADDIE align with SDLC?

The Training phases follow the sequence portrayed in the ADDIE model. These activities are in turn aligned sequentially with the phases in the SDLC. Let's now see how this close amalgamation happens.

Training Analysis and Design - Software Development Life Cycle Design Phase

Typically, training activities begin during, or, at the end of the Design phase of SDLC. The ID performs a detailed audience and task analysis through interaction with the proposed users of the system, key client representatives for training, the Business Analyst and technical team.

The ID typically analyzes the following inputs:

- Request for Proposal (RFP)
- Master Agreement document

- Functional Specification Document (FSD)
- General Design Document (GDD)/Detailed Design Document (DDD)
- Htmls and Wireframes of the application

Based on these inputs the ID typically develops the following deliverables:

- Needs Analysis Report stating the findings of the Analysis phase
- Training Management plan that clearly states the list of deliverables and the schedule of development and delivery of these deliverables corresponding to the SDLC
- Curriculum map linking the user tasks and training needs
- Curriculum Design providing the course structure including the modules, topics, duration and other requirements for the training.

Training Development - Software Development Life Cycle Development and Testing Phase

All development activities for a software application only begin after the sign off of a Training Management Plan. Development of Training materials occur towards the end of the Development phase of the SDLC and continues till UAT, after which minor updates are made to the materials based on the changes in the application.

Delivery of pre-UAT trainings may also occur during this phase.

Some clients also show a preference for the Train-the-Trainer (TTT) trainings after UAT, so that their trainers can coordinate and co-facilitate training delivery to the end-users.

Training Implementation and Evaluation - Software Development Life Cycle Deployment and Maintenance Phase

Delivery of user trainings occurs prior to Go-live and complete deployment. Sometimes Web-based Trainings (WBTs) are delivered after deployment for reference purposes.

For classroom trainings, a training environment is created in the servers and the users learn all about the system through practice and walkthrough.

Feedback forms and online methods are provided for the learners to share their feedback for trainings. To evaluate how much the learners have progressed through the trainings, classroom practice tests and online tests are used. Online tests can be tracked through the Learning Management System (LMS) of the client, if it is available.

All technical and DBA trainings and knowledge transfer activities occur during the Maintenance and Support phase of SDLC.

Mapping Training Materials to Training Categories

Taking into consideration the varied needs and complexities of software trainings, it is important to have a variety of training materials.

Here’s a list of the categories of trainings and the corresponding materials.

- **For user trainings**
  WBTS and Classroom training materials such as Facilitator slides, Quick Reference Guide, Participant Workbook, Instructor Guide etc.
- **For technical trainings**
  System Operations Guide, Configuration Guide, User Manuals etc.

- **For DBA trainings**
  Admin Guide and select sections of the user training materials and User Guide
- **For TTT trainings**
  Relevant sections of all the materials based on the audience group

Industry Trends and 'Rapid' Learning

While software development is the order of the day, trainings for software applications have gathered momentum.

The trend shows that software application trainings are more in volume for any organization in the training and development space.

In a bid to meet the training needs, many clients who require software applications to be developed, request for trainings using rapid development tools.

These tools have gained popularity for their quick and effective method of rendering screenshots of the application into a quick online training.

However, very little time is invested in analysis, design or other key aspects of such trainings. The usual 'rapid' approach is to capture the screenshots and record audio to take the learners through the application using a WBT.

For 'rapid' classroom trainings a user manual is often substituted for structured Participant and Instructor Guides and other training materials. A PowerPoint presentation is used to support the user manual and the trainings are conducted through a rapid 'walkthrough'.

While this is a ‘quick fix’ solution, research has shown that many of these trainings have not resulted in effective on-the-job transition of knowledge. After such trainings, the users have found it difficult to use a new system and execute their tasks.

A structured training consisting of the following can prevent such ineffective and rushed learning:

- A precursor WBT with specific modules to prepare the learners for classroom trainings
- A detailed classroom training with all the training materials such as Participant Workbook, Instructor Guide, Facilitator Slides and Quick Reference Guides, with clearly defined learning modules, walkthrough, practice and tests
- A follow up WBT with all the content for the training for reference and brush up

Saving time and money through ‘rapid’ trainings can prove costly in the long run, when system users and maintenance staff grapple with problems while using the system. This could even impact the business through critical errors or non-usage of the system.

Therefore, it is for trainers to suggest the best practices to the client when they opt for a software application training.

Conclusion

As we can see, training and SDLC are by no means exclusive of each other. They complement each other and make each other effective. For a software application to be deployed and used effectively, a well planned and structured training holds the key to easy adaptation by the audience.

Post Implementation Review

A Post Implementation review is conducted after completing the project. Its activities aim to evaluate whether project objectives were met, how effectively the project was run, lessons for the future, and the actions required to maximise the benefits from the project outputs.

1. **Conduct a post implementation review**

   Post Implementation Review activities include:

   - Consulting with the business owner area/s and key stakeholders to assess how well the project outputs achieved the intended business outcomes and the expected benefits and identifying if more needs to be done to maximise the realisation of the benefits from the project outputs.
   - Referring to the Project Plan, Benefit Management Plan and Cost Benefit Analysis spreadsheet to assess the final cost of the project, anticipated cost savings and level of attainment of the expected benefit.
   - Identifying all the ongoing and support activities required beyond the life of the project to fully realise the business benefits. These activities may include introducing new policies or procedures; training; or updates to websites and other information resources.

   The Project Manager will use this information to complete the Post Implementation Review document and provide recommendations of ongoing actions required to fully realise the benefits, deliver operational requirements such as training, and enable guidance for future projects based on lessons learned.

2. **Update the lessons learned**
   Review the Lessons Learned Log and update with any additional relevant lessons.
3. **Ongoing benefits realisation**
   The Project Sponsor and/or Project Board/Steering Committee needs to review the Post Implementation Review document and where appropriate, assign the recommendations to key stakeholders and the business owner to ensure the new system/service/facilities is fully adopted and benefits are realised.

Software Maintenance

Software maintenance is a part of the Software Development Life Cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real world changes, the software require alteration wherever possible.

Software Maintenance is an inclusive activity that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization.

Need for Maintenance

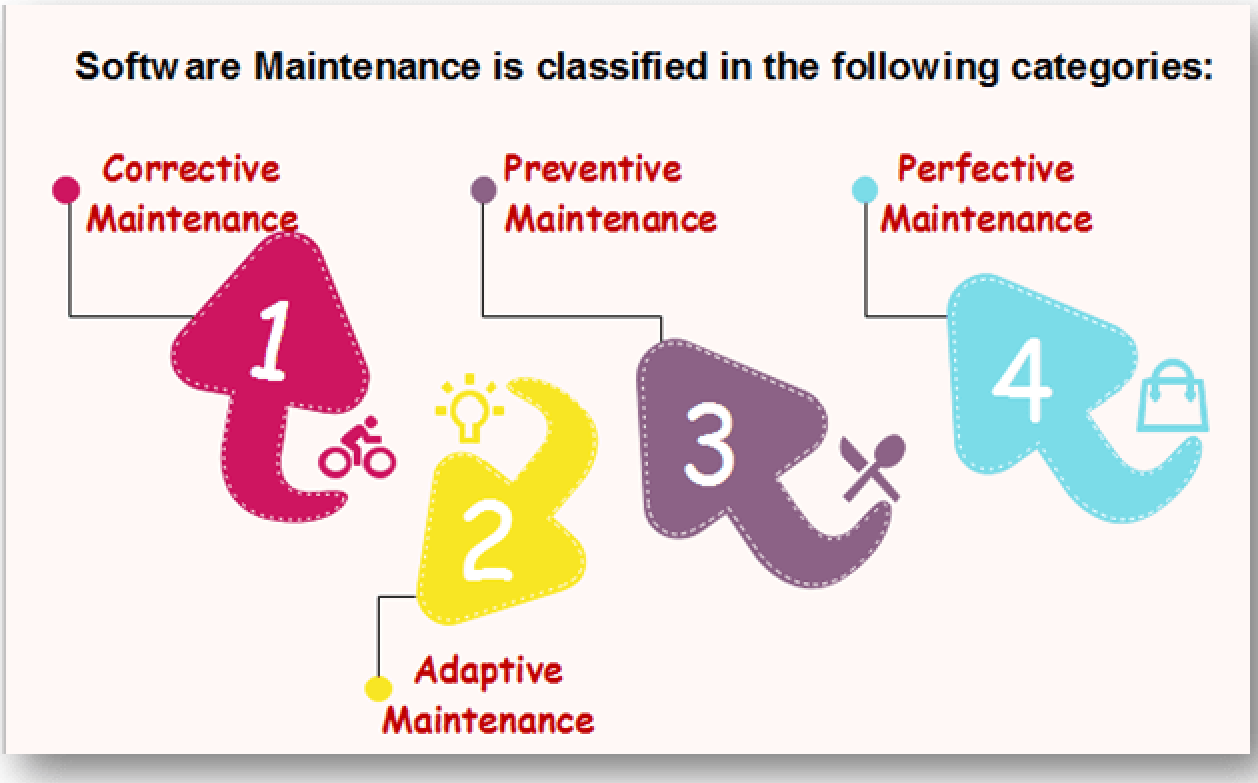
Software Maintenance is needed for:-

- Correct errors
- Change in user requirement with time
- Changing hardware/software requirements

- To improve system efficiency
- To optimize the code to run faster
- To modify the components
- To reduce any unwanted side effects.

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Types of Software Maintenance



<u>1. Corrective Maintenance</u>

Corrective maintenance aims to correct any remaining errors regardless of where they may cause specifications, design, coding, testing, and documentation, etc.

<u>2. Adaptive Maintenance</u>

It contains modifying the software to match changes in the ever-changing environment.

<u>3. Preventive Maintenance</u>

It is the process by which we prevent our system from being obsolete. It involves the concept of reengineering & reverse engineering in which an old system with old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

<u>4. Perfective Maintenance</u>

It defines improving processing efficiency or performance or restricting the software to enhance changeability. This may contain enhancement of existing system functionality, improvement in computational efficiency, etc.

Automated Testing tools

Test Automation is the best way to increase the effectiveness, test coverage, and execution speed in software testing. Automated software testing is important due to the following reasons:

- Manual Testing of all workflows, all fields, all negative scenarios is time and money consuming
- It is difficult to test for multilingual sites manually
- Test Automation in software testing does not require Human intervention. You can run automated test unattended (overnight)
- Test Automation increases the speed of test execution
- Automation helps increase Test Coverage
- Manual Testing can become boring and hence error-prone.

Which Test Cases to Automate?

Test cases to be automated can be selected using the following criterion to increase the automation ROI

- High Risk - Business Critical test cases
- Test cases that are repeatedly executed
- Test Cases that are very tedious or difficult to perform manually
- Test Cases which are time-consuming

The following category of test cases are not suitable for automation:

- Test Cases that are newly designed and not executed manually at least once
- Test Cases for which the requirements are frequently changing
- Test cases which are executed on an ad-hoc basis.

Automated Testing Process:

Following steps are followed in an Automation Process

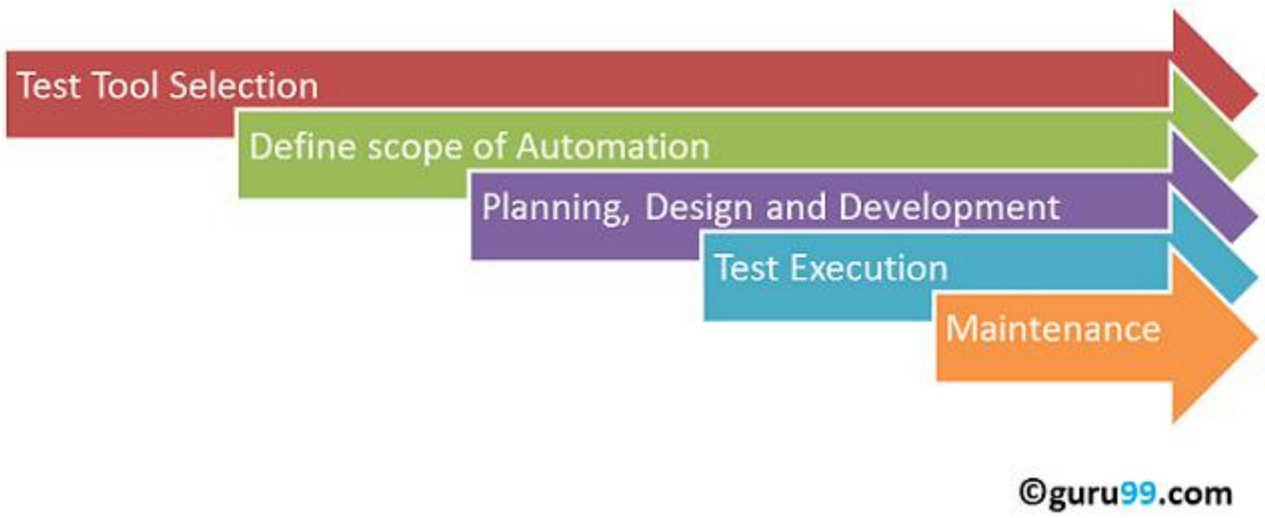
Step 1) Test Tool Selection

Step 2) Define scope of Automation

Step 3) Planning, Design and Development

Step 4) Test Execution

Step 5) Maintenance



Test Automation Process

Test tool selection

Test Tool selection largely depends on the technology the Application Under Test is built on. For instance, QTP does not support Informatica. So QTP cannot be used for testing Informatica applications. **It's a good idea to conduct a Proof of Concept of Tool on AUT.**

Define the scope of Automation

The scope of automation is the area of your Application Under Test which will be automated. Following points help determine scope:

- The features that are important for the business
- Scenarios which have **a large amount of data**
- **Common functionalities** across applications
- Technical feasibility
- The extent to which business components are reused
- **The complexity** of test cases
- Ability to use the same test cases for cross-browser testing

Planning, Design, and Development

During this phase, you create an Automation strategy & plan, which contains the following details-

- Automation tools selected
- Framework design and its features
- In-Scope and Out-of-scope items of automation
- Automation testbed preparation
- Schedule and Timeline of scripting and execution
- Deliverables of Automation Testing

Test Execution

Automation Scripts are executed during this phase. The scripts need input test data before there are set to run. Once executed they provide detailed test reports.

Execution can be performed using the automation tool directly or through the Test Management tool which will invoke the automation tool.

Example: Quality center is the Test Management tool which in turn it will invoke QTP for execution of automation scripts. Scripts can be executed in a single machine or a group of machines. The execution can be done during the night, to save time.

Test Automation Maintenance Approach

Test Automation Maintenance Approach is an automation testing phase carried out to test whether the new functionalities added to the software are working fine or not. Maintenance in automation testing is executed when new automation scripts are added and need to be reviewed and maintained in order to improve the effectiveness of automation scripts with each successive release cycle.

Framework for Automation

A framework is set of automation guidelines which help in

- Maintaining consistency of Testing
- Improves test structuring
- Minimum usage of code
- Less Maintenance of code
- Improve re-usability
- Non Technical testers can be involved in code
- The training period of using the tool can be reduced
- Involves Data wherever appropriate

There are four types of frameworks used in automation software testing:

1. Data Driven Automation Framework
2. Keyword Driven Automation Framework
3. Modular Automation Framework
4. Hybrid Automation Framework

Automation Tool Best Practices

To get maximum ROI of automation, observe the following

- The scope of Automation needs to be determined in detail before the start of the project. This sets expectations from Automation right.
- Select the right automation tool: A tool must not be selected based on its popularity, but it's fit to the automation requirements.
- Choose an appropriate framework
- Scripting Standards- Standards have to be followed while writing the scripts for Automation. Some of them are-
    - Create uniform scripts, comments, and indentation of the code
    - Adequate Exception handling - How error is handled on system failure or unexpected behavior of the application.
    - User-defined messages should be coded or standardized for Error Logging for testers to understand.
- Measure metrics- Success of automation cannot be determined by comparing the manual effort with the automation effort but by also capturing the following metrics.
    - Percent of defects found
    - The time required for automation testing for each and every release cycle
    - Minimal Time is taken for release
    - Customer Satisfaction Index
    - Productivity improvement

The above guidelines if observed can greatly help in making your automation successful.