Table of Contents

[What is software testing? 3](#_Toc192929699)

[What is the objective of software testing 3](#_Toc192929700)

[Verification techniques besides testing 3](#_Toc192929701)

[Classification of Testing Techniques: 3](#_Toc192929702)

[Statistical Testing 3](#_Toc192929703)

[Defect Testing 3](#_Toc192929704)

[Functional Testing (Black Box Testing) 3](#_Toc192929705)

[Structure Testing (White Box Testing) 3](#_Toc192929706)

[Unit Testing 3](#_Toc192929707)

[Sub System Testing (Integration testing) 3](#_Toc192929708)

[System Testing 3](#_Toc192929709)

[Acceptance Testing (Alpha, Beta testing) 4](#_Toc192929710)

[What is functional testing 4](#_Toc192929711)

[Writing Test cases from functional specs 4](#_Toc192929712)

[Main parts of functional test cases are the following 4](#_Toc192929713)

[Software Testing - How to Log A Defect (Bug) 5](#_Toc192929714)

[Importance of Severity VS Priority: 6](#_Toc192929715)

[Automation 7](#_Toc192929716)

[Guidelines for Automation 7](#_Toc192929717)

[Scope 7](#_Toc192929718)

[Preparation Timeframe 7](#_Toc192929719)

[Return on Investment 7](#_Toc192929720)

[The Degree of Change 7](#_Toc192929721)

[Test Integrity 7](#_Toc192929722)

[Test Independence 7](#_Toc192929723)

[Debugging 7](#_Toc192929724)

[Record & Playback 7](#_Toc192929725)

[Maintenance of Scripts 8](#_Toc192929726)

[Benefits of Automation 8](#_Toc192929727)

[Reducing Testing Time 8](#_Toc192929728)

[Reducing Testing Costs 8](#_Toc192929729)

[Replicating Testing Across Different Platforms 8](#_Toc192929730)

[Repeatability and Control 8](#_Toc192929731)

[Greater Application Coverage 8](#_Toc192929732)

[Results Reporting 8](#_Toc192929733)

[Business Case for Automation 9](#_Toc192929734)

[Testing Tools 10](#_Toc192929735)

[Test Planning and Management 10](#_Toc192929736)

[Testing Product Integration 10](#_Toc192929737)

[Internet/Intranet Testing 10](#_Toc192929738)

[Ease of Use 10](#_Toc192929739)

[GUI and Client/Server Testing 10](#_Toc192929740)

[Load and Performance Testing 10](#_Toc192929741)

[Sample Tools 11](#_Toc192929742)

[Glossary: 11](#_Toc192929743)

# What is software testing?

Process of determing a computer program with pre-determined inputs and comparing the actual outputs with the expected outputs.

# What is the objective of software testing?

* Phase I: Show software works
* Phase II: Show software doesn’t work
* Phase III : Test for risk reduction
* Phase IV: A state of Product/Software

# Verification techniques besides testing

* Testing depends on having the system to “execute” (Dynamic)
* Inspection methods
* Static analysis methods

# Classification of Testing Techniques:

* **Statistical Testing**

To evaluate a system or component. Used to demonstrate a system's fitness for use, to predict the reliability of a system in an operational environment, to efficiently allocate testing resources, to predict the amount of testing required after a system change, to qualify components for reuse, and to identify when enough testing has been accomplished. Used to test a program’s performance and reliability.

* **Defect Testing**

Used to find areas of non conformance to the system specification.

* **Functional Testing (Black Box Testing)**

Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

Synonym: black-box testing

* **Structure Testing (White Box Testing)**

Test cases are generated based on internal structure of the software Views program as a white box.

* **Unit Testing**

Testing of individual component usually requires of using test drivers. Unit testing is a development procedure where programmers create tests as they develop software. The tests are simple short tests that test functionality of a particular unit or module of their code, such as a class or function.

* **Sub System Testing (Integration testing)**

Testing of interfaces between integrated components.

* **System Testing**

Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. Validating functional as well as non-functional requirement.

* **Acceptance Testing (Alpha, Beta testing)**

Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

# What is functional testing?

Functional testing, or black box testing, is a quality assurance process used to verify that an application’s end-user functionality (i.e., the ability to log in, complete a transaction, etc.) works accurately, reliably, predictably and securely. Functional testing can involve either manual or automated methods. Either way, it entails running a series of tests that emulate the interaction between the user and the application in order to verify whether or not the application does what it was designed to do.

# Writing Test cases from functional specs:

A test case is input-action-output along with expected results. So you basically take the functionality that is listed in the spec you have and write out how to test that functionality. So if you've got a form on a Web page your input would be the inputs to the form, the action would probably be clicking a button or something to submit the contents of the form, and the output would be whatever happens after you submit the input. The expected results are what SHOULD happen but the output may be something totally different. So you have to record that difference. The functional spec will tell you what SHOULD happen when certain functionality occurs. That is what you can use to check against. So just write a test case to simulate the actions you would take to act out some piece of functionality listed in the spec you have.

## Main parts of functional test cases are the following:

* Specified Input(s)
* Expected Output(s) for the task(s) that the function accomplishes

Functional Tests cases may also state:

* Initial state considerations.
* Special set-up considerations.
* Expected end-of-test state.

Functional test cases should be written to verify that the function does what it is supposed to do. They may also be written to verify that the function is does not do what it is not supposed to do.

Functional test cases should, in addition to testing typical inputs, focus on boundary conditions. For example, if the function is to accept as input a number between 0 and 99, test cases should be developed for inputs of -1 and 100.

If the function spec includes Data Flow Diagrams (or their OOA equivalents), the determining Input-Activity-Output is relatively straight forward. However, most functional specs are not that rigorous. Often either input is given - but not output, or output is given - but not input, or inputs and outputs are stated but not the activity. This can be a major difficulty for the tester.  
  
  
Step1: Get a Condition from the FS.  
E.g. Take a Login Screen with conditions

1. User Name should be unique

2. No Blank Space & Special Characters

3. Password shouldn’t be greater than 5 Characters

Step 2: Now we will take a single condition for building a test case

E.g. Password shouldn’t be greater than 5 characters.  
(Step 2 shows a single condition now we can write a test case for it)

Prerequisite: Login Screen should be displayed   
Navigation Path/Description: Go to the "User Name" field and enter a valid user name, go to password field and enter a password with character more than five   
Data's used:

User Name : ABC

Password : XXXX

Expected Result: The system should pop-up an error message stating that "Password Cannot be greater than 5 Characters"

This how we could get a condition from a FS and write test cases .Here we can have more than one condition for test case and vice versa is also possible.

The Architecture of Test Case goes like this

1. Equivalence Class   
2. Boundary Value Analysis  
3. Cause Effect Graph  
4. Error Guessing

While writing test cases we must be very care full that this is going to be the last fragmented part ,no more fragmentation from the test case should be possible.

# Software Testing - How to Log A Defect (Bug)

A brief introduction to how a bug/defect/error is reported during software testing.

As we already have discussed importance of Software Testing in any software development project (Just to summarize: Software testing helps in improving quality of software and deliver a cost effective solution which meet customer requirements), it becomes necessary to log a defect in a proper way, track the defect, and keep a log of defects for future reference etc.   
  
As a tester tests an application and if he/she finds any defect, the life cycle of the defect starts and it becomes very important to communicate the defect to the developers in order to get it fixed, keep track of current status of the defect, find out if any such defect (similar defect) was ever found in last attempts of testing etc. For this purpose, previously manually created documents were used, which were circulated to everyone associated with the software project (developers and testers), now a days many Bug Reporting Tools are available, which help in tracking and managing bugs in an effective way.   
  
**Reporting a bug:**   
It’s a good practice to take screen shots of execution of every step during software testing. If any test case fails during execution, it needs to be failed in the bug-reporting tool and a bug has to be reported/logged for the same. The tester can choose to first report a bug and then fail the test case in the bug-reporting tool or fail a test case and report a bug. In any case, the Bug ID that is generated for the reported bug should be attached to the test case that is failed.   
  
At the time of reporting a bug, all the mandatory fields from the contents of bug (such as Project, Summary, Description, Status, Detected By, Assigned To, Date Detected, Test Lead, Detected in Version, Closed in Version, Expected Date of Closure, Actual Date of Closure, Severity, Priority and Bug ID etc.) are filled and detailed description of the bug is given along with the expected and actual results. The screen-shots taken at the time of execution of test case are attached to the bug for reference by the developer.   
  
After reporting a bug, a unique Bug ID is generated by the bug-reporting tool, which is then associated with the failed test case. This Bug ID helps in associating the bug with the failed test case.   
  
After the bug is reported, it is assigned a status of ‘New’, which goes on changing as the bug fixing process progresses.   
  
If more than one tester are testing the software application, it becomes a possibility that some other tester might already have reported a bug for the same defect found in the application. In such situation, it becomes very important for the tester to find out if any bug has been reported for similar type of defect. If yes, then the test case has to be blocked with the previously raised bug (in this case, the test case has to be executed once the bug is fixed). And if there is no such bug reported previously, the tester can report a new bug and fail the test case for the newly raised bug.   
  
If no bug-reporting tool is used, then in that case, the test case is written in a tabular manner in a file with four columns containing Test Step No, Test Step Description, Expected Result and Actual Result. The expected and actual results are written for each step and the test case is failed for the step at which the test case fails.   
  
This file containing test case and the screen shots taken are sent to the developers for reference. As the tracking process is not automated, it becomes important keep updated information of the bug that was raised till the time it is closed.   
  
(**Please Note:** The above given procedure of reporting a bug is general and not based on any particular project. Most of the times, the bug reporting procedures, values used for the various fields used at the time of reporting a bug and bug tracking system etc. may change as par the software testing project and company requirements.)

**Importance of Severity VS Priority:**

Severity is important according to Testing Team perceptive while raising a defect. Where as Priority comes first while executing the Bug because based on the priority only we (testers) will decide which to execute first and not.

So both are important in different directions.

Severity will be given by Testers and Priority will be given by Business Analyst or by Delivery Managers.

It depends on the condition for example there is release time and there are two bugs the first one is if you inter a 100 word password then the system will crash and the second one is the GUI related bug in which your company name spelling is not correct. And you can solve only one bug this time due to less time then

For password related bug I will assign: high severity, low priority

For GUI related bug I will assign: high priority, low severity.

# Automation

**Guidelines for Automation**

**Scope:**

It is not practical to try to automate everything, nor is there the time available generally. Pick very carefully the functions/areas of the application that are to be automated.

**Preparation Timeframe:**

The preparation time for automated test scripts has to be taken into account. In general, the preparation time for automated scripts can be up to 2/3 times longer than for manual testing. An automated testing tool does not replace manual testing, nor does it replace the test engineer. Initially, the test effort will increase, but when automation is done correctly it will decrease on subsequent releases.

**Return on Investment:**

Because the preparation time for test automation is so long, the benefit of the test automation only begins to occur after the tests have been run.

**The Degree of Change:**

The best use of test automation is for regression testing, whereby you use automated tests to ensure that pre-existing functions (e.g. functions from version 1.0 - i.e. not new functions in this release) are unaffected by any changes introduced in version 1.1.

**Test Integrity:**

How do you know (measure) whether a test passed or failed? Just because the tool returns a ‘pass’ does not necessarily mean that the test itself passed.

**Test Independence:**

Test independence must be built in so that a failure in the first test case won't cause a domino effect and either prevents, or cause to fail, the rest of the test scripts in that test suite. However, in practice this is very difficult to achieve.

**Debugging:**

Time must be allowed for this, and to prove the integrity of the tests themselves.

**Record & Playback:**

DO NOT RELY on record & playback as the SOLE means to generate a script.

**Maintenance of Scripts:**

There is a high maintenance overhead for automated test scripts. They have to be continuously kept up to date.

## Benefits of Automation:

**Reducing Testing Time:**

Since testing is a repetitive task, Automation of testing processes allows machines to complete the tedious, repetitive work while humans can perform other tasks. An automated test executes the next operation in the test hierarchy at machine speed, allowing tests to be completed many times faster than the fastest individual.

**Reducing Testing Costs:**

The cost of performing manual testing is prohibitive when compared to automated methods. But when load / stress testing needs to be done, automated testing is the only solution as simulation for *n* number of users can be done easily with a single computer compared to manual testing where *n* number of computers have to be arranged with individual testers.

**Replicating Testing Across Different Platforms:**

Automation allows the testing organization to perform consistent and repeatable tests. When applications need to be deployed across different hardware or software platforms, standard or benchmark tests can be created and repeated on target platforms to ensure that new platforms operate consistently.

**Repeatability and Control:**

By using automated techniques, the tester has a very high degree of control over which types of tests are being performed, and how the tests will be executed. Using automated tests enforces consistent procedures that allow developers to evaluate the effect of various application modifications as well as the effect of various user actions.

**Greater Application Coverage:**

The productivity gains delivered by automated testing allow and encourage organizations to test more often and more completely. Greater application test coverage also reduces the risk of exposing users to malfunctioning or non-compliant software.

**Results Reporting:**

Full-featured automated testing systems also produce convenient test reporting and analysis. These reports provide a standardized measure of test status and results, thus allowing more accurate interpretation of testing outcomes. Manual methods require the user to self-document test procedures and test results.

**Business Case for Automation**

There are three key benefits to automation:

1. Expand your test coverage.
2. Save time and resources.
3. Retain knowledge.

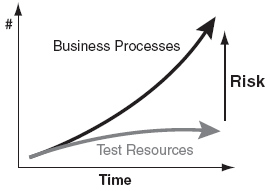
Expanding your test coverage is one of the most valuable benefits of automation because it translates into higher quality and thus less costs associated with downtime, errors, and rework. Over the life of your SAP deployment you will likely experience an increase in the number of business processes it supports, either through the implementation of additional modules or integration with other systems.

As a result, each successive implementation or modification affects a greater number of business processes, which increases the risk and opportunity for failure. Even a 10 percent increase in total functionality still requires testing of 100 percent of the process inventory due to the risk of unexpected impact. The tightly integrated nature of SAP increases this risk.

As Exhibit 5.1 shows, a manual test process cannot keep pace with this expanding workload because time and resources available for testing are either fixed or even declining. In this exhibit, the lighter arrow indicates the processes that need to be tested and the dark arrow indicates the number of test resources. This combination of increasing processes that need to be tested with a static number of testers leads to increased risk and potential cost of failure.

Under the scenario represented in Exhibit 5.1, automation is the only practical answer. It enables one to capture tests as repeatable assets that can be executed for each successive release or deployment, so that the inventory of tests can keep pace with the inventory of business processes at risk.

This repeatability saves time and resources as well. Instead of requiring repetitive manual effort to reverify processes each time changes are introduced, tests can be automatically executed in an unattended mode. This allows your resources to focus on adding new



tests to support new functionality instead of constantly repeating existing tests.

Ironically, when test time is short, testers will often sacrifice regression testing in favor of testing new features. The irony is that the greatest risk to the user is in the existing features, not the new ones. If a business process that the enterprise depends on stops working— or worse, starts doing the wrong thing—then you could halt operations. The loss of a new feature may be inconvenient or even embarrassing, but it is unlikely to be devastating.

This benefit will be lost if the automated tests are not designed to be maintainable as the application changes. If they either have to be rewritten or require significant modifications to be reused, you will keep starting over instead of building on prior efforts. Therefore, it is essential to adopt an approach to test library design that supports maintainability over the life of the application.

Finally, the process of automating your test cases introduces discipline and formality to testing, which results in the capture of application knowledge in the form of test assets. You cannot automate what is not defined. By defining your business processes and rules as test cases, you are converting the experience of subject matter experts (SMEs) and business analysts (BAs) into an asset that can be preserved and reused over the long term, protecting you from the inevitable loss of expertise due to turnover.

# Testing Tools

## Test Planning and Management:

A robust testing tool should have the capability to manage the testing process, provide organization for testing components, and create meaningful end-user and management reports. It should also allow users to include non-automated testing procedures within automated test plans and test results.

## Testing Product Integration:

Testing tools should provide tightly integrated modules that support test component reusability. Test components built for performing functional tests should also support other types of testing including regression and load/stress testing. All products within the testing product environment should be based upon a common, easy-to-understand language. User training and experience gained in performing one testing task should be transferable to other testing tasks.

## Internet/Intranet Testing:

A good tool will have the ability to support testing within the scope of a web browser. The tests created for testing Internet or intranet-based applications should be portable across browsers, and should automatically adjust for different load times and performance levels.

## Ease of Use:

Testing tools should be engineered to be usable by non-programmers and application end-users. Even if programmers are responsible for testing, the testing tool itself should have a short learning curve.

## GUI and Client/Server Testing:

A robust testing tool should support testing with a variety of user interfaces and create simple-to manage, easy-to-modify tests. Test component reusability should be a cornerstone of the product architecture.

## Load and Performance Testing:

The selected testing solution should allow users to perform meaningful load and performance tests to accurately measure system performance. It should also provide test results in an easy-to-understand reporting format.

## Sample Tools

|  |  |  |
| --- | --- | --- |
| **Functionality** | **Description** | **Representative Tools** |
| Functional Testing | Record and Playback tools with scripting support aid in automating the functional testing of online applications | Win Runner, Rational Robot, Silk Test and QA Run. Tools like CA-Verify can be used in the m/f environment |
| Test Management | Management the test effort | Test Director |
| Test Coverage | Reports from the tool provide data on coverage per unit like Function, Program, and Application | Rational Pure Coverage |
| File Comparators | Verify regression test results (by comparison of results from original and changed applications). | Comparex (from Sterling Software) |
| Load Testing | Performance and scalability testing | Load Runner, Performance Studio, Silk Performer and QA Load |
| Run Time Error Checking | Detect hard to find run-time errors, memory leaks, etc. | Rational Purify |
| Debugging Tools | Simplify isolation and fixing of errors | Xpediter, ViaSoft (Mainframe applications), VisualAge debuggers. |
| Test Bed Generator | Tools aid in preparing test data by analyzing program flows and conditional statements | CA-Datamacs |

# Glossary:

**Error** - The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically corrects value or condition.

**Fault** - An incorrect step, process, or data definition in a computer program.

**Debug** - To detect, locate, and correct faults in a computer program.

**Failure** - The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.

**Testing** - The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.

**Static analysis** - The process of evaluating a system or component based on its form, structure, content, or documentation.

**Dynamic analysis** - The process of evaluating a system or component based on its behavior during execution.

**Correctness** - The degree to which a system or component is free from faults in its specification, design, and implementation. The degree to which software, documentation, or other items meet specified requirements. The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.

**Verification** - The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Formal proof of program correctness.

**Validation** - The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

**Test cases** – the collection of inputs, expected results, environment and procedural requirement for a single test.

**Test suite/test pool** – a collection of test cases necessary to “adequately” test a product.

**Test plan** – a document describing the scope, approach, resources and scheduled testing activity.

**Failure** – the manifested inability of a system or component to perform a required function within

specified limits (external incorrect behavior, symptoms)