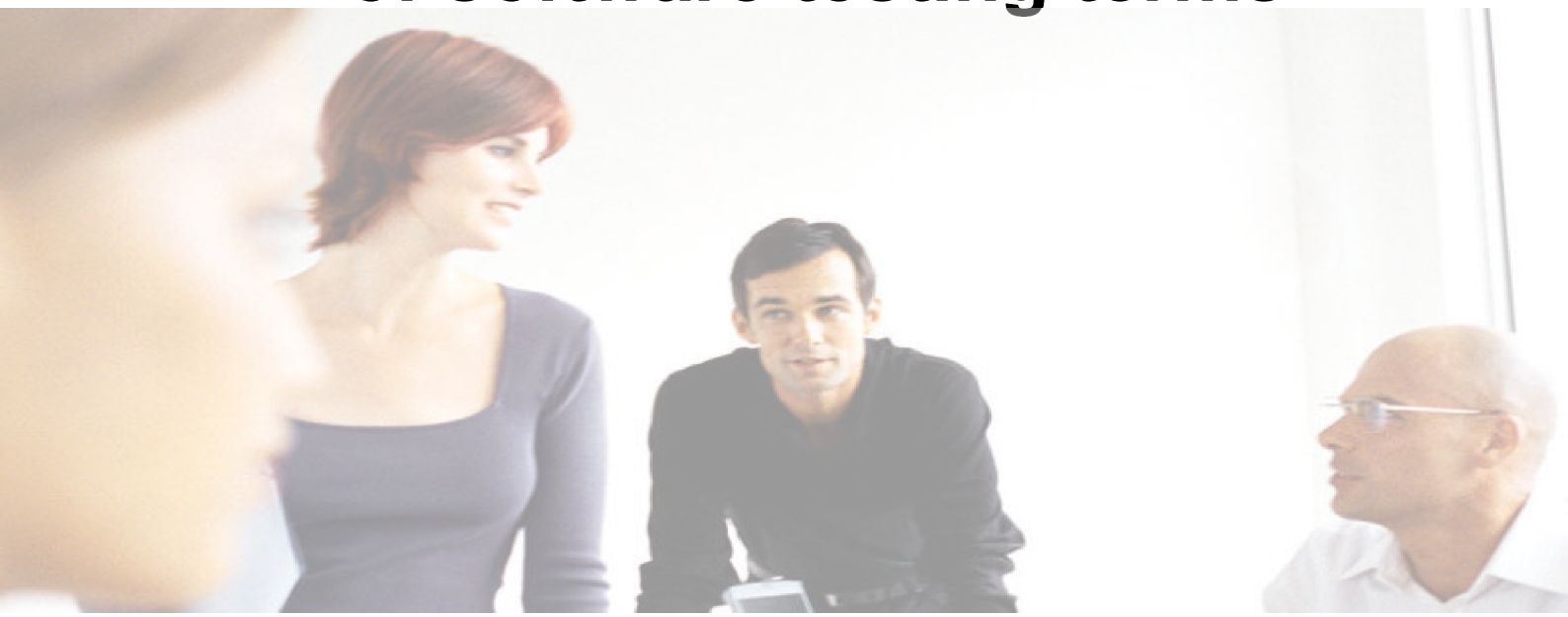
Glossary
of software testing terms



Original Software

This glossary of software testing terms and conditions is a compilation of knowledge, gathered over time, from many different sources. It is provided “as-is” in good faith, without any warranty as to the accuracy or currency of any definition or other information contained herein. If you have any questions or queries about the contents of this glossary, please contact Original Software directly.

Orĭ’gĭnal *a.* Not derivative or dependant, first-hand, not imitative, novel in character or style, inventive, creative, thinking or acting for oneself.

Sŏft’ware *n.* Programs etc. For computer, or other interchangeable material for performing operations.

A

Acceptance Testing: Formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component. Normally performed to validate the software meets a set of agreed acceptance criteria.

Accessibility Testing: Verifying a product is accessible to the people having disabilities (visually impaired, hard of hearing etc.)

Actual Outcome: The actions that are produced when the object is tested under specific conditions.

Ad hoc Testing: Testing carried out in an unstructured and improvised fashion. Performed without clear expected results, ad hoc testing is most often used as a compliment to other types of testing. See also Monkey Testing.

Alpha Testing: Simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Arc Testing: See branch testing.

Agile Testing: Testing practice for projects using agile methodologies, treating development as the customer of testing and emphasizing a test-first design philosophy. In agile development testing is integrated throughout the lifecycle, testing the software throughout its development. See also Test Driven Development.

Application Binary Interface (ABI): Describes the low-level interface between an application program and the operating system, between an application and its libraries, or between component parts of the application. An ABI differs from an application programming interface (API) in that an API defines the interface between source code and libraries, so that the same source code will compile on any system supporting that API, whereas an ABI allows compiled object code to function without changes on any system using a compatible ABI.

Application Development Lifecycle The process flow during the various phases of the application development life cycle.

The Design Phase depicts the design phase up to the point of starting development. Once all of the requirements have been gathered, analyzed, verified, and a design has been produced, we are ready to pass on the programming requirements to the application programmers.

The programmers take the design documents (programming requirements) and then proceed with the iterative process of coding, testing, revising, and testing again, this is the **Development Phase.**

After the programs have been tested by the programmers, they will be part of a series of formal user and system tests. These are used to verify usability and functionality from a user point of view, as well as to verify the functions of the application within a larger framework.

The final phase in the development life cycle is to go to **production** and become a steady state. As a prerequisite to going to production, the development team needs to provide documentation. This usually consists of user training and operational procedures. The user training familiarizes the users with the new application. The operational procedures documentation enables Operations to take over responsibility for running the application on an ongoing basis.

In production, the changes and enhancements are handled by a group (possibly the same programming group) that performs the maintenance. At this point in the life cycle of the application, changes are tightly controlled and must be rigorously tested before being implemented into production

Application Programming Interface (API): Provided by operating systems or libraries in response to support requests for services to be made of it by computer programs

Automated Software Quality (ASQ): The use of software tools, such as automated testing tools, to improve software quality.

Automated Software Testing: The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions, without manual intervention.

Automated Testing Tools: Software tools used by development teams to automate and streamline their testing and quality assurance process.

B

Backus-Naur Form (BNF): A metasyntax used to express context-free grammars: that is, a formal way to describe formal languages.

BNF is widely used as a notation for the grammars of computer programming languages, instruction sets and communication protocols, as well as a notation for representing parts of natural language grammars. Many textbooks for programming language theory and/or semantics document the programming language in BNF.

Basic Block: A sequence of one or more consecutive, executable statements containing no branches.

Basis Path Testing: A white box test case design technique that fulfills the requirements of branch testing & also tests all of the independent paths that could be used to construct any arbitrary path through the computer program.

Basis Test Set: A set of test cases derived from Basis Path Testing.

Baseline: The point at which some deliverable produced during the software engineering process is put under formal change control.

Bebugging: A popular software engineering technique used to measure test coverage. Known bugs are randomly added to a program source code and the programmer is tasked to find them. The percentage of the known bugs not found gives an indication of the real bugs that remain.

Behavior: The combination of input values and preconditions along with the required response for a function of a system. The full specification of a function would normally comprise one or more behaviors.

Benchmark Testing: Benchmark testing is a normal part of the application development life cycle. It is a team effort that involves both application developers and database administrators (DBAs), and should be performed against your application in order to determine current performance and improve it. If the application code has been written as efficiently as possible, additional performance gains might be realized from tuning the database and database manager configuration parameters. You can even tune application parameters to meet the requirements of the application better.

You run different types of benchmark tests to discover specific kinds of information:

- A *transaction per second* benchmark determines the throughput capabilities of the database manager under certain limited laboratory conditions.
- An *application* benchmark tests the same throughput capabilities under conditions that are closer production conditions.

Benchmarking is helpful in understanding how the database manager responds under varying conditions. You can create scenarios that test deadlock handling, utility performance, different methods of loading data, transaction rate characteristics as more users are added, and even the effect on the application of using a new release of the product.

Benchmark Testing Methods: Benchmark tests are based on a repeatable environment so that the same test run under the same conditions will yield results that you can legitimately compare.

You might begin benchmarking by running the test application in a normal environment. As you narrow down a performance problem, you can develop specialized test cases that limit the scope of the function that you are testing. The specialized test cases need not emulate an entire application to obtain valuable information. Start with simple measurements, and increase the complexity only when necessary.

Characteristics of good benchmarks or measurements include:

- Tests are repeatable.
- Each iteration of a test starts in the same system state.
- No other functions or applications are active in the system unless the scenario includes some amount of other activity going on in the system.
- The hardware and software used for benchmarking match your production environment.

For benchmarking, you create a scenario and then applications in this scenario several times, capturing key information during each run. Capturing key information after each run is of primary importance in determining the changes that might improve performance of both the application and the database.

Beta Testing: Comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the company. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Big-Bang Testing: An inappropriate approach to integration testing in which you take the entire integrated system and test it as a unit. Can work well on small systems but is not favorable for larger systems because it may be difficult to pinpoint the exact location of the defect when a failure occurs.

Binary Portability Testing: Testing an executable application for portability across system platforms and environments, usually for conformation to an ABI specification.

Black Box Testing: Testing without knowledge of the internal workings of the item being tested. For example, when black box testing is applied to software engineering, the tester would only know the "legal" inputs and what the expected outputs should be, but not how the program actually arrives at those outputs. It is because of this that black box testing can be considered testing with respect to the specifications, no other knowledge of the program is necessary. For this reason, the tester and the programmer can be independent of one another, avoiding programmer bias toward his own work. For this testing, test groups are often used.

Advantages of Black Box Testing

- More effective on larger units of code than glass box testing.
- Tester needs no knowledge of implementation,
- Tester and programmer are independent of each other.
- Tests are done from a user's point of view.
- Will help to expose any ambiguities or inconsistencies in the specifications.
- Test cases can be designed as soon as the specifications are complete.

Disadvantages of Black Box Testing

- Only a small number of possible inputs can actually be tested, to test every possible input stream would take nearly forever.
- Without clear and concise specifications, test cases are hard to design.
- There may be unnecessary repetition of test inputs if the tester is not informed of test cases the programmer has already tried.
- May leave many program paths untested.
- Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone).
- Most testing related research has been directed toward glass box testing.

Block Matching: Automated matching logic applied to data and transaction driven websites to automatically detect block s of related data. This enables repeating elements to be treated correctly in relation to other elements in the block without the need for special coding. See TestDrive-Gold

Bottom Up Testing: An approach to integration testing where the lowest level components are tested first, then used to facilitate the testing of higher level components.

Boundary Testing: Tests focusing on the boundary or limits of the software being tested.

Boundary Value: An input value or output value which is on the boundary between equivalence classes, or an incremental distance either side of the boundary.

Boundary Value Analysis: In boundary value analysis, test cases are generated using the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values.

Boundary Value Coverage: The percentage of boundary values which have been exercised by a test case suite.

Branch: A conditional transfer of control from any statement to any other statement in a component, or an unconditional transfer of control from any statement to any other statement in the component except the next statement, or when a component has more than one entry point, a transfer of control to an entry point of the component.

Branch Condition Coverage: The percentage of branch condition outcomes in every decision that has been tested.

Branch Condition Combination Coverage: The percentage of combinations of all branch condition outcomes in every decision that has been tested.

Branch Condition Combination Testing: A test case design technique in which test cases are designed to execute combinations of branch condition outcomes.

Branch Condition Testing: A technique in which test cases are designed to execute branch condition outcomes.

Branch Testing: A test case design technique for a component in which test cases are designed to execute branch outcomes.

Breadth Testing: A test suite that exercises the full functionality of a product but does not test features in detail.

Bug: A fault in a program which causes the program to perform in an unintended. See fault.

C

Capture/Playback Tool: A test tool that records test input as it is sent to the software under test. The input cases stored can then be used to reproduce the test at a later time.

Capture/Replay Tool: See capture/playback tool.

CAST: Acronym for computer-aided software testing. Automated software testing in one or more phases of the software life-cycle. See also ASQ.

Cause-Effect Graph: A graphical representation of inputs or stimuli (causes) with their associated outputs (effects), which can be used to design test cases.

Capability Maturity Model for Software (CMM): The CMM is a process model based on software best-practices effective in large-scale, multi-person projects. The CMM has been used to assess the maturity levels of organization areas as diverse as software engineering, system engineering, project management, risk management, system acquisition, information technology (IT) or personnel management, against a scale of five key processes, namely: Initial, Repeatable, Defined, Managed and Optimized.

Capability Maturity Model Integration (CMMI) Capability Maturity Model® Integration (CMMI) is a process improvement approach that provides organizations with the essential elements of effective processes. It can be used to guide process improvement across a project, a division, or an entire organization. CMMI helps integrate traditionally separate organizational functions, set process improvement goals and priorities, provide guidance for quality processes, and provide a point of reference for appraising current processes.

Seen by many as the successor to the CMM, the goal of the CMMI project is to improve the usability of maturity models by integrating many different models into one framework.

Certification: The process of confirming that a system or component complies with its specified requirements and is acceptable for operational use.

Chow's Coverage Metrics: See N-switch coverage.

Code Complete: A phase of development where functionality is implemented in its entirety; bug fixes are all that are left. All functions found in the Functional Specifications have been implemented.

Code Coverage: A measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that looks at the code directly and as such comes under the heading of white box testing.

To measure how well a program has been tested, there are a number of coverage criteria – the main ones being:

- Functional Coverage – has each function in the program been tested?
- Statement Coverage – has each line of the source code been tested?
- Condition Coverage – has each evaluation point (i.e. a true/false decision) been tested?
- Path coverage – has every possible route through a given part of the code been executed?
- Entry/exit Coverage – has every possible call and return of the function been tested?

Code-Based Testing: The principle of structural code based testing is to have each and every statement in the program executed at least once during the test. Based on the premise that one cannot have confidence in a section of code unless it has been exercised by tests, structural code based testing attempts to test all reachable elements in the software under the cost and time constraints. The testing process begins by first identifying areas in the program not being exercised by the current set of test cases, follow by creating additional test cases to increase the coverage.

Code-Free Testing: Next generation software testing technique from Original Software which does not require complicated scripting language to learn. Instead, a simple point and click interface is used to significantly simplify the process of test creation. See TestDrive-Gold

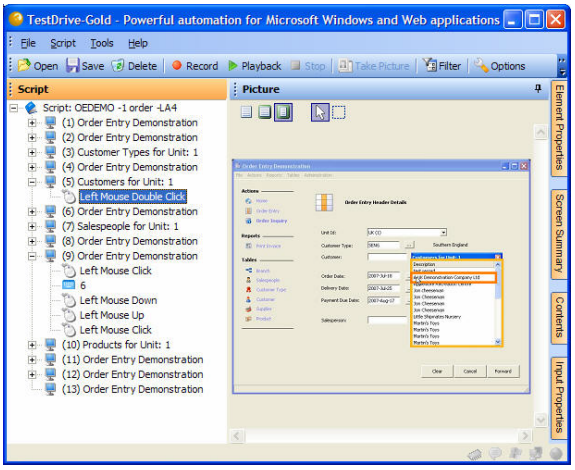


Fig 1. Code-Free testing with TestDrive-Gold

Code Inspection: A formal testing technique where the programmer reviews source code with a group who ask questions analyzing the program logic, analyzing the code with respect to a checklist of historically common programming errors, and analyzing its compliance with coding standards.

Code Walkthrough: A formal testing technique where source code is traced by a group with a small set of test cases, while the state of program variables is manually monitored, to analyze the programmer's logic and assumptions.

Coding: The generation of source code.

Compatibility Testing: The process of testing to understand if software is compatible with other elements of a system with which it should operate, e.g. browsers, Operating Systems, or hardware.

Complete Path Testing: See exhaustive testing.

Component: A minimal software item for which a separate specification is available.

Component Testing: The testing of individual software components.

Component Specification: A description of a component's function in terms of its output values for specified input values under specified preconditions.

Computation Data Use: A data use not in a condition. Also called C-use.

Concurrent Testing: Multi-user testing geared towards determining the effects of accessing the same application code, module or database records. See Load Testing

Condition: A Boolean expression containing no Boolean operators. For instance, A<B is a condition but A and B is not.

Condition Coverage: See branch condition coverage.

Condition Outcome: The evaluation of a condition to TRUE or FALSE.

Conformance Criterion: Some method of judging whether or not the component's action on a particular specified input value conforms to the specification.

Conformance Testing: The process of testing to determine whether a system meets some specified standard. To aid in this, many test procedures and test setups have been developed, either by the standard's maintainers or external organizations, specifically for testing conformance to standards.

Conformance testing is often performed by external organizations; sometimes the standards body itself, to give greater guarantees of compliance. Products tested in such a manner are then advertised as being certified by that external organization as complying with the standard.

Context Driven Testing: The context-driven school of software testing is similar to Agile Testing that advocates continuous and creative evaluation of testing opportunities in light of the potential information revealed and the value of that information to the organization right now.

Control Flow: An abstract representation of all possible sequences of events in a program's execution.

Control Flow Graph: The diagrammatic representation of the possible alternative control flow paths through a component.

Control Flow Path: See path.

Conversion Testing: Testing of programs or procedures used to convert data from existing systems for use in replacement systems.

Correctness: The degree to which software conforms to its specification.

Coverage: The degree, expressed as a percentage, to which a specified coverage item has been tested.

Coverage Item: An entity or property used as a basis for testing.

Cyclomatic Complexity: A software metric (measurement). It was developed by Thomas McCabe and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

D

Data Case: Data relationship model simplified for data extraction and reduction purposes in order to create test data.

Data Definition: An executable statement where a variable is assigned a value.

Data Definition C-use Coverage: The percentage of data definition C-use pairs in a component that are exercised by a test case suite.

Data Definition C-use Pair: A data definition and computation data use, where the data use uses the value defined in the data definition.

Data Definition P-use Coverage: The percentage of data definition P-use pairs in a component that is tested.

Data Definition P-use Pair: A data definition and predicate data use, where the data use uses the value defined in the data definition.

Data Definition-use Coverage: The percentage of data definition-use pairs in a component that are exercised by a test case suite.

Data Definition-use Pair: A data definition and data use, where the data use uses the value defined in the data definition.

Data Definition-use Testing: A test case design technique for a component in which test cases are designed to execute data definition-use pairs.

Data Dictionary: A database that contains definitions of all data items defined during analysis.

Data Driven Testing: A framework where test input and output values are read from data files and are loaded into variables in captured or manually coded scripts. In this framework, variables are used for both input values and output verification values. Navigation through the program, reading of the data files, and logging of test status and information are all coded in the test script.

This is similar to keyword-driven testing in that the test case is contained in the data file and not in the script; the script is just a "driver," or delivery mechanism, for the data. Unlike in table-driven testing, though, the navigation data isn't contained in the table structure. In data-driven testing, only test data is contained in the data files.

Data Flow Diagram: A modeling notation that represents a functional decomposition of a system.

Data Flow Coverage: Test coverage measure based on variable usage within the code. Examples are data definition-use coverage, data definition P-use coverage, data definition C-use coverage, etc.

Data Flow Testing: Data-flow testing looks at the lifecycle of a particular piece of data (i.e. a variable) in an application. By looking for patterns of data usage, risky areas of code can be found and more test cases can be applied.

Data Protection: Technique in which the condition of the underlying database is synchronized with the test scenario so that differences can be attributed to logical changes. This technique also automatically re-sets the database after tests - allowing for a constant data set if a test is re-run. See TestBench

Data Protection Act: UK Legislation surrounding the security, use and access of an individual's information. May impact the use of live data used for testing purposes.

Data Use: An executable statement where the value of a variable is accessed.

Database Testing: The process of testing the functionality, security, and integrity of the database and the data held within.

Functionality of the database is one of the most critical aspects of an application's quality; problems with the database could lead to data loss or security breaches, and may put a company at legal risk depending on the type of data being stored. For more information on database testing see TestBench.

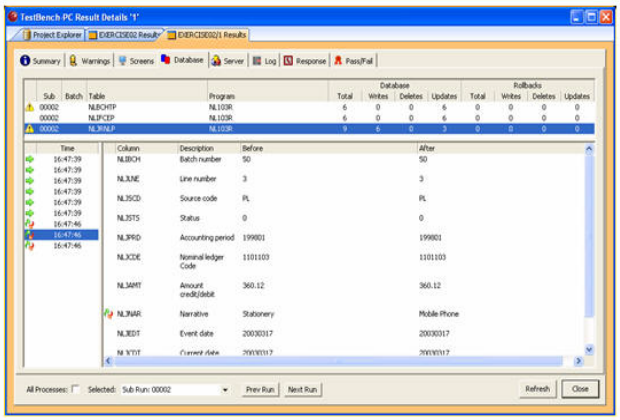


Fig 2. Database testing using TestBench for Oracle

Debugging: A methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.

Decision: A program point at which the control flow has two or more alternative routes.

Decision Condition: A condition held within a decision.

Decision Coverage: The percentage of decision outcomes that have been exercised by a test case suite.

Decision Outcome: The result of a decision.

Defect: Nonconformance to requirements or functional / program specification

Delta Release: A delta, or partial, release is one that includes only those areas within the release unit that have actually changed or are new since the last full or delta release. For example, if the release unit is the program, a delta release contains only those modules that have changed, or are new, since the last full release of the program or the last delta release of certain modules.

Dependency Testing: Examines an application's requirements for pre-existing software, initial states and configuration in order to maintain proper functionality.

Depth Testing: A test that exercises a feature of a product in full detail.

Desk Checking: The testing of software by the manual simulation of its execution.

Design-Based Testing: Designing tests based on objectives derived from the architectural or detail design of the software (e.g., tests that execute specific invocation paths or probe the worst case behavior of algorithms).

Dirty Testing: Testing which demonstrates that the system under test does not work. (Also known as negative testing)

Documentation Testing: Testing concerned with the accuracy of documentation.

Domain: The set from which values are selected.

Domain Expert: A person who has significant knowledge in a specific domain.

Domain Testing: Domain testing is the most frequently described test technique. The basic notion is that you take the huge space of possible tests of an individual variable and subdivide it into subsets that are (in some way) equivalent. Then you test a representative from each subset.

Downtime: Total period that a service or component is not operational.

Dynamic Testing: Testing of the dynamic behavior of code. Dynamic testing involves working with the software, giving input values and checking if the output is as expected.

Dynamic Analysis: the examination of the physical response from the system to variables that are not constant and change with time.

E

Emulator: A device that duplicates (provides an *emulation* of) the functions of one system using a different system, so that the second system behaves like (and appears to be) the first system. This focus on exact reproduction of external behavior is in contrast to simulation, which can concern an abstract model of the system being simulated, often considering internal state.

Endurance Testing: Checks for memory leaks or other problems that may occur with prolonged execution.

End-to-End Testing: Testing a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

Entry Point: The first executable statement within a component.

Equivalence Class: A mathematical concept, an equivalence class is a subset of given set induced by an equivalence relation on that given set. (If the given set is empty, then the equivalence relation is empty, and there are no equivalence classes; otherwise, the equivalence relation and its concomitant equivalence classes are all non-empty.) Elements of an equivalence class are said to be equivalent, under the equivalence relation, to all the other elements of the same equivalence class.

Equivalence Partition: See equivalence class.

Equivalence Partitioning: Leverages the concept of "classes" of input conditions. A "class" of input could be "City Name" where testing one or several city names could be deemed equivalent to testing all city names. In other words each instance of a class in a test covers a large set of other possible tests.

Equivalence Partition Coverage: The percentage of equivalence classes generated for the component, which have been tested.

Equivalence Partition Testing: A test case design technique for a component in which test cases are designed to execute representatives from equivalence classes.

Error: A mistake that produces an incorrect result.

Error Guessing: Error Guessing involves making an itemized list of the errors expected to occur in a particular area of the system and then designing a set of test cases to check for these expected errors. Error guessing is more testing art than testing science but can be very effective given a tester familiar with the history of the system.

Error Seeding: The process of injecting a known number of "dummy" defects into the program and then check how many of them are found by various inspections and testing. If, for example, 60% of them are found, the presumption is that 60% of other defects have been found as well. See Bebugging.

Evaluation Report: A document produced at the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned.

Executable statement: A statement which, when compiled, is translated into object code, which will be executed procedurally when the program is running and may perform an action on program data.

Exercised: A program element is exercised by a test case when the input value causes the execution of that element, such as a statement, branch, or other structural element.

Exhaustive Testing: Testing which covers all combinations of input values and preconditions for an element of the software under test.

Exit Point: The last executable statement within a component.

Expected Outcome: See predicted outcome.

Expert System: A domain specific knowledge base combined with an inference engine that processes knowledge encoded in the knowledge base to respond to a user's request for advice.

Expertise: Specialized domain knowledge, skills, tricks, shortcuts and rules-of-thumb that provide an ability to rapidly and effectively solve problems in the problem domain.

Failure: Non performance or deviation of the software from its expected delivery or service.

Fault: A manifestation of an error in software. Also know as a bug

Feasible Path: A path for which there exists a set of input values and execution conditions which causes it to be executed.

Feature Testing: A method of testing which concentrates on testing one feature at a time.

Firing a Rule: A rule fires when the "if" part (premise) is proven to be true. If the rule incorporates an "else" component, the rule also fires when the "if" part is proven to be false.

Fit For Purpose Testing: Validation carried out to demonstrate that the delivered system can be used to carry out the tasks for which it was designed and acquired.

Forward Chaining: Applying a set of previously determined facts to the rules in a knowledge base to see if any of them will fire.

Full Release: All components of the release unit that are built, tested, distributed and implemented together. See also delta release.

Functional Specification: The document that describes in detail the characteristics of the product with regard to its intended capability.

Functional Decomposition: A technique used during planning, analysis and design; creates a functional hierarchy for the software. Functional Decomposition broadly relates to the process of resolving a functional relationship into its constituent parts in such a way that the original function can be reconstructed (i.e., recomposed) from those parts by function composition. In general, this process of decomposition is undertaken either for the purpose of gaining insight into the identity of the constituent components (which may reflect individual physical processes of interest, for example), or for the purpose of obtaining a compressed representation of the global function, a task which is feasible only when the constituent processes possess a certain level of *modularity* (i.e. independence or non-interaction).

Functional Requirements: Define the internal workings of the software: that is, the calculations, technical details, data manipulation and processing and other specific functionality that show how the use cases are to be satisfied. They are supported by non-functional requirements, which impose constraints on the design or implementation (such as performance requirements, security, quality standards, or design constraints).

Functional Specification: A document that describes in detail the characteristics of the product with regard to its intended features.

Functional Testing: See also Black Box Testing.

- Testing the features and operational behavior of a product to ensure they correspond to its specifications.
- 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.

G

Genetic Algorithms: Search procedures that use the mechanics of natural selection and natural genetics. It uses evolutionary techniques, based on function optimization and artificial intelligence, to develop a solution.

Glass Box Testing: A form of testing in which the tester can examine the design documents and the code as well as analyze and possibly manipulate the internal state of the entity being tested. Glass box testing involves examining the design documents and the code, as well as observing at run time the steps taken by algorithms and their internal data. See structural test case design.

Goal: The solution that the program or project is trying to reach.

Gorilla Testing: An intense round of testing, quite often redirecting all available resources to the activity. The idea here is to test as much of the application in as short a period of time as possible.

Graphical User Interface (GUI): A type of display format that enables the user to choose commands, start programs, and see lists of files and other options by pointing to pictorial representations (icons) and lists of menu items on the screen.

Gray (Grey) Box Testing: A testing technique that uses a combination of black box testing and white box testing. Gray box testing is not black box testing because the tester does know some of the internal workings of the software under test. In gray box testing, the tester applies a limited number of test cases to the internal workings of the software under test. In the remaining part of the gray box testing, one takes a black box approach in applying inputs to the software under test and observing the outputs.

H

Harness: A test environment comprised of stubs and drivers needed to conduct a test.

Heuristics: The informal, judgmental knowledge of an application area that constitutes the "rules of good judgment" in the field. Heuristics also encompass the knowledge of how to solve problems efficiently and effectively, how to plan steps in solving a complex problem, how to improve performance, etc.

High Order Tests: High-order testing checks that the software meets customer requirements and that the software, along with other system elements, meets the functional, behavioral, and performance requirements. It uses black-box techniques and requires an outsider perspective. Therefore, organizations often use an Independent Testing Group (ITG) or the users themselves to perform high-order testing.

High-order testing includes validation testing, system testing (focusing on aspects such as reliability, security, stress, usability, and performance), and acceptance testing (includes alpha and beta testing). The testing strategy specifies the type of high-order testing that the project requires. This depends on the aspects that are important in a particular system from the user perspective.

I

ITIL (IT Infrastructure Library): A consistent and comprehensive documentation of best practice for IT Service Management. ITIL consists of a series of books giving guidance on the provision of quality IT services, and on the accommodation and environmental facilities needed to support IT.

Implementation Testing: See Installation Testing.

Incremental Testing: Partial testing of an incomplete product. The goal of incremental testing is to provide an early feedback to software developers.

Independence: Separation of responsibilities which ensures the accomplishment of objective evaluation.

Independent Test Group (ITG): A group of people whose primary responsibility is to conduct software testing for other companies.

Infeasible path: A path which cannot be exercised by any set of possible input values.

Inference: Forming a conclusion from existing facts.

Inference Engine: Software that provides the reasoning mechanism in an expert system. In a rule based expert system, typically implements forward chaining and backward chaining strategies.

Infrastructure: The organizational artifacts needed to perform testing, consisting of test environments, automated test tools, office environment and procedures.

Inheritance: The ability of a class to pass on characteristics and data to its descendants.

Input: A variable (whether stored within a component or outside it) that is read by the component.

Input Domain: The set of all possible inputs.

Inspection: A group review quality improvement process for written material. It consists of two aspects; product (document itself) improvement and process improvement.

Installability: The ability of a software component or system to be installed on a defined target platform allowing it to be run as required. Installation includes both a new installation and an upgrade.

Installability Testing: Testing whether the software or system installation being tested meets predefined installation requirements.

Installation Guide: Supplied instructions on any suitable media, which guides the installer through the installation process. This may be a manual guide, step-by-step procedure, installation wizard, or any other similar process description.

Installation Testing: Confirms that the application under test recovers from expected or unexpected events without loss of data or functionality. Events can include shortage of disk space, unexpected loss of communication, or power out conditions.

Such testing focuses on what customers will need to do to install and set up the new software successfully and is typically done by the software testing engineer in conjunction with the configuration manager. Implementation testing is usually defined as testing which places a compiled version of code into the testing or pre-production environment, from which it may or may not progress into production. This generally takes place outside of the software development environment to limit code corruption from other future releases which may reside on the development network.

Installation Wizard: Supplied software on any suitable media, which leads the installer through the installation process. It shall normally run the installation process, provide feedback on installation outcomes and prompt for options.

Instrumentation: The insertion of additional code into the program in order to collect information about program behavior during program execution.

Integration: The process of combining components into larger groups or assemblies.

Integration Testing: Testing of combined parts of an application to determine if they function together correctly. Usually performed after unit and functional testing. This type of testing is especially relevant to client/server and distributed systems.

Interface Testing: Integration testing where the interfaces between system components are tested.

Isolation Testing: Component testing of individual components in isolation from surrounding components.

J & K

KBS (Knowledge Based System): A domain specific knowledge base combined with an inference engine that processes knowledge encoded in the knowledge base to respond to a user's request for advice.

Key Performance Indicator: Quantifiable measurements against which specific performance criteria can be set.

Keyword Driven Testing: An approach to test script writing aimed at code based automation tools that separates much of the programming work from the actual test steps. The results is the test steps can be designed earlier and the code base is often easier to read and maintain.

Knowledge Engineering: The process of codifying an expert's knowledge in a form that can be accessed through an expert system.

Known Error: An incident or problem for which the root cause is known and for which a temporary Work-around or a permanent alternative has been identified.

L

LCSAJ: A Linear Code Sequence And Jump, consisting of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

LCSAJ Coverage: The percentage of LCSAJs of a component which are exercised by a test case suite.

LCSAJ Testing: A test case design technique for a component in which test cases are designed to execute LCSAJs.

Logic-Coverage Testing: Sometimes referred to as Path Testing, logic-coverage testing attempts to expose software defects by exercising a unique combination of the program's statements known as a path.

Load Testing: The process of creating demand on a system or device and measuring its response. Load testing generally refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program's services concurrently. As such, this testing is most relevant for multi-user systems, often one built using a client/server model, such as web servers. However, other types of software systems can be load-tested also. For example, a word processor or graphics editor can be forced to read an extremely large document; or a financial package can be forced to generate a report based on several years' worth of data. The most accurate load testing occurs with actual, rather than theoretical, results. See also Concurrent Testing, Performance Testing, Reliability Testing, and Volume Testing.

Localization Testing: This term refers to making software specifically designed for a specific locality. This test is based on the results of globalization testing, which verifies the functional support for that particular culture/locale. Localization testing can be executed only on the localized version of a product.

The test effort during localization testing focuses on:

- Areas affected by localization, such as UI and content
- Culture/locale-specific, language-specific, and region-specific areas

In addition, localization testing should include:

- Basic functionality tests
- Setup and upgrade tests run in the localized environment
- Plan application and hardware compatibility tests according to the product's target region.

Log: A chronological record of relevant details about the execution of tests.

Loop Testing: Loop testing is the testing of a resource or resources multiple times under program control.

M

Maintainability: The ease with which the system/software can be modified to correct faults, modified to meet new requirements, modified to make future maintenance easier, or adapted to a changed environment.

Maintenance Requirements: A specification of the required maintenance needed for the system/software. The released software often needs to be revised and/or upgraded throughout its lifecycle. Therefore it is essential that the software can be easily maintained, and any errors found during re-work and upgrading.

Within traditional software testing techniques, script maintenance is often a problem as it can be very complicated and time consuming to ensure correct maintenance of the software as the scripts these tools use need updating every time the application under test changes. See Code-Free Testing and Self Healing Scripts.

Manual Testing: The oldest type of software testing. Manual testing requires a tester to perform manual test operations on the test software without the help of test automation. Manual testing is a laborious activity that requires the tester to possess a certain set of qualities; to be patient, observant, speculative, creative, innovative, open-minded, resourceful, un-opinionated, and skilful.

As a tester, it is always advisable to use manual white box testing and black-box testing techniques on the test software. Manual testing helps discover and record any software bugs or discrepancies related to the functionality of the product.

Manual testing can be augmented by test automation. It is possible to record and playback manual steps and write automated test script(s) using test automation tools. Although, test automation tools will only help execute test scripts written primarily for executing a particular specification and functionality. Test automation tools lack the ability of decision-making and recording any unscripted discrepancies during program execution. It is recommended that one should perform manual testing of the entire product at least a couple of times before actually deciding to automate the more mundane activities of the product.

Manual testing helps discover defects related to the usability testing and GUI testing area. While performing manual tests the software application can be validated whether it meets the various standards defined for effective and efficient usage and accessibility. For example, the standard location of the OK button on a screen is on the left and of CANCEL button on the right. During manual testing you might discover that on some screen, it is not. This is a new defect related to the usability of the screen. In addition, there could be many cases where the GUI is not displayed correctly and the basic functionality of the program is correct. Such bugs are not detectable using test automation tools.

Repetitive manual testing can be difficult to perform on large software applications or applications having very large dataset coverage. This drawback is compensated for by using manual black-box testing techniques including equivalence partitioning and boundary value analysis. Using which, the vast dataset specifications can be divided and converted into a more manageable and achievable set of test suites.

There is no complete substitute for manual testing. Manual testing is crucial for testing software applications more thoroughly. See TestDrive-Assist.

Metric: A standard of measurement. Software metrics are the statistics describing the structure or content of a program. A metric should be a real objective measurement of something such as number of bugs per lines of code.

Modified Condition/Decision Coverage: The percentage of all branch condition outcomes that independently affect a decision outcome that have been exercised by a test case suite.

Modified Condition/Decision Testing: A test case design technique in which test cases are designed to execute branch condition outcomes that independently affect a decision outcome.

Monkey Testing: Testing a system or an application on the fly, i.e. a unit test with no specific end result in mind.

Multiple Condition Coverage: See Branch Condition Combination Coverage.

Mutation Analysis: A method to determine test case suite thoroughness by measuring the extent to which a test case suite can discriminate the program from slight variants (mutants) of the program. See also Error Seeding.

Mutation Testing: Testing done on the application where bugs are purposely added to it. See Bebugging.

N

N-switch Coverage: The percentage of sequences of N-transitions that have been tested.

N-switch Testing: A form of state transition testing in which test cases are designed to execute all valid sequences of N-transitions.

N-transitions: A sequence of N+ transitions.

N+1 Testing: A variation of regression testing. Testing conducted with multiple cycles in which errors found in test cycle N are resolved and the solution is retested in test cycle N+1. The cycles are typically repeated until the solution reaches a steady state and there are no errors. See also Regression Testing.

Natural Language Processing (NLP): A computer system to analyze, understand and generate natural human languages.

Negative Testing: Testing a system or application using negative data. (For example testing a password field that requires a minimum of 9 characters by entering a password of 6).

Neural Network: A system modeled after the neurons (nerve cells) in a biological nervous system. A neural network is designed as an interconnected system of processing elements, each with a limited number of inputs and outputs. Rather than being programmed, these systems learn to recognize patterns.

Non-functional Requirements Testing: Testing of those requirements that do not relate to functionality. i.e. performance, usability, etc.

Normalization: A technique for designing relational database tables to minimize duplication of information and, in so doing, to safeguard the database against certain types of logical or structural problems, namely data anomalies.

O

Object: A software structure which represents an identifiable item that has a well-defined role in a problem domain.

Object Orientated: An adjective applied to any system or language that supports the use of objects.

Objective: The purpose of the specific test being undertaken.

Operational Testing: Testing performed by the end-user on software in its normal operating environment.

Oracle: A mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test.

Outcome: The result or visible effect of a test.

Output: A variable (whether stored within a component or outside it) that is written to by the component.

Output Domain: The set of all possible outputs.

Output Value: An instance of an output.

P

Page Fault: A program interruption that occurs when a page that is marked 'not in real memory' is referred to by an active page.

Pair Programming: A software development technique that requires two programmers to participate in a combined development effort at one workstation. Each member performs the action the other is not currently doing: for example, while one types in unit tests, the other thinks about the class that will satisfy the test.

The person who is doing the typing is known as the *driver* while the person who is guiding is known as the *navigator*. It is often suggested for the two partners to switch roles at least every half-hour or after a unit test is made. It is also suggested to switch partners at least once a day.

Pair Testing: In much the same way as Pair Programming, two testers work together to find defects. Typically, they share one computer and trade control of it while testing.

Pairwise Testing: A combinatorial software testing method that, for each pair of input parameters to a system (typically, a software algorithm) tests all possible discrete combinations of those parameters. Using carefully chosen test vectors, this can be done much faster than an exhaustive search of all combinations of all parameters, by "parallelizing" the tests of parameter pairs. The number of tests is typically O(*nm*), where *n* and *m* are the number of possibilities for each of the two parameters with the most choices.

The reasoning behind all-pairs testing is this: the simplest bugs in a program are generally triggered by a single input parameter. The next simplest category of bugs consists of those dependent on interactions between pairs of parameters, which can be caught with all-pairs testing. Bugs involving interactions between three or more parameters are progressively less common, whilst at the same time being progressively more expensive to find by exhaustive testing, which has as its limit the exhaustive testing of all possible inputs.

Many testing methods regard all-pairs testing of a system or subsystem as a reasonable cost-benefit compromise between often computationally infeasible higher-order combinatorial testing methods, and less exhaustive methods which fail to exercise all possible pairs of parameters. Because no testing technique can find all bugs, all-pairs testing is typically used together with other quality assurance techniques such as unit testing. See TestDrive-Gold.

Partial Test Automation: The process of automating parts but not all of the software testing process. If, for example, an oracle cannot reasonably be created, or if fully automated tests would be too difficult to maintain, then a software tools engineer can instead create testing tools to help human testers perform their jobs more efficiently. Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

Pass: Software has deemed to have passed a test if the actual results of the test matched the expected results.

Pass/Fail Criteria: Decision rules used to determine whether an item under test has passed or failed a test.

Path: A sequence of executable statements of a component, from an entry point to an exit point.

Path Coverage: The percentage of paths in a component exercised by a test case suite.

Path Sensitizing: Choosing a set of input values to force the execution of a component to take a given path.

Path Testing: Used as either black box or white box testing, the procedure itself is similar to a walk-through. First, a certain path through the program is chosen. Possible inputs and the correct result are written down. Then the program is executed by hand, and its result is compared to the predefined. Possible faults have to be written down at once.

Performance: The degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate.

Performance Testing: A test procedure that covers a broad range of engineering or functional evaluations where a material, product, or system is not specified by detailed material or component specifications: Rather, emphasis is on the final measurable performance characteristics. Also known as Load Testing.

Portability: The ease with which the system/software can be transferred from one hardware or software environment to another.

Portability Requirements: A specification of the required portability for the system/software.

Portability Testing: The process of testing the ease with which a software component can be moved from one environment to another. This is typically measured in terms of the maximum amount of effort permitted. Results are expressed in terms of the time required to move the software and complete data conversion and documentation updates.

Postcondition: Environmental and state conditions that must be fulfilled after the execution of a test or test procedure.

Positive Testing: Testing aimed at showing whether the software works in the way intended. See also Negative Testing.

Precondition: Environmental and state conditions which must be fulfilled before the component can be executed with a particular input value.

Predicate: A logical expression which evaluates to TRUE or FALSE, normally to direct the execution path in code.

Predication: The choice to execute or not to execute a given instruction.

Predicted Outcome: The behavior expected by the specification of an object under specified conditions.

Priority: The level of business importance assigned to an individual item or test.

Predication: The choice to execute or not to execute a given instruction.

Predicted Outcome: The behavior expected by the specification of an object under specified conditions.

Priority: The level of business importance assigned to an individual item or test.

Process: A course of action which turns inputs into outputs or results.

Process Cycle Test: A black box test design technique in which test cases are designed to execute business procedures and processes.

Progressive Testing: Testing of new features after regression testing of previous features.

Project: A planned undertaking for presentation of results at a specified time in the future.

Prototyping: A strategy in system development in which a scaled down system or portion of a system is constructed in a short time, then tested and improved upon over several iterations.

Pseudo-Random: A series which appears to be random but is in fact generated according to some prearranged sequence.

Q

Quality Assurance: The activity of providing evidence needed to establish confidence among all concerned, that quality-related activities are being performed effectively. All those planned or systematic actions necessary to provide adequate confidence that a product or service will satisfy given requirements for quality.

For software development organizations, TMM (Testing Maturity Model) standards are widely used to measure the Quality Assurance. These standards can be divided in to 5 steps, which a software development company can achieve by performing different quality improvement activities within the organization.

Quality Attribute: A feature or characteristic that affects an item's quality.

Quality Audit: A systematic and independent examination to determine whether quality activities and related results comply with planned arrangements and whether these arrangements are implemented effectively and are suitable to achieve objectives.

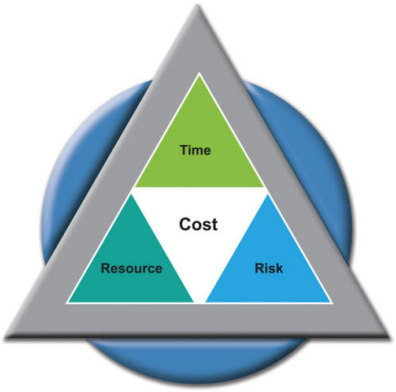
Quality Circle: A group of individuals with related interests that meet at regular intervals to consider problems or other matters related to the quality of outputs of a process and to the correction of problems or to the improvement of quality.

Quality Control: The operational techniques and the activities used to fulfill and verify requirements of quality.

Quality Management: That aspect of the overall management function that determines and implements the quality policy. Direction and control with regard to quality generally includes the establishment of the quality policy and quality objectives, quality planning, quality control, quality assurance and quality improvement.

Quality Conundrum: Resource, risk and appliaction time-to-market are often in conflict as IS teams strive to deliver quality applications within their budgetary constraints. This is the quality conundrum.

Fig. 4 The Quality Conundrum



Quality Policy: The overall intentions and direction of an organization as regards quality as formally expressed by top management.

Quality System: The organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

Query: A question. Often associated with an SQL query of values in a database.

Queuing Time: Incurred when the device, which a program wishes to use, is already busy. The program therefore has to wait in a queue to obtain service from that device.

R

ROI: Return on Investment. A performance measure used to evaluate the efficiency of an investment or to compare the efficiency of a number of different investments. To calculate ROI, the benefit (return) of an investment is divided by the cost of the investment; the result is expressed as a percentage or a ratio.

$$ROI = \frac{(\text{Gain from Investment} - \text{Cost of Investment})}{\text{Cost of Investment}}$$

Ramp Testing: Continuously raising an input signal until the system breaks down.

Random Testing: A black-box testing approach in which software is tested by choosing an arbitrary subset of all possible input values. Random testing helps to avoid the problem of only testing what you know will work.

Re-testing: Testing that runs test cases that failed the last time they were run, in order to verify the success of corrective actions.

Recoverability: The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure.

Recovery Testing: The activity of testing how well the software is able to recover from crashes, hardware failures and other similar problems. Recovery testing is the forced failure of the software in a variety of ways to verify that recovery is properly performed.

Examples of recovery testing:

- While the application running, suddenly restart the computer and after that check the validness of application's data integrity.
- While application receives data from the network, unplug and then in some time plug-in the cable, and analyze the application ability to continue receiving of data from that point, when network connection disappeared.
- To restart the system while the browser will have definite number of sessions and after rebooting check, that it is able to recover all of them.

Recreation Materials: A script or set of results containing the steps required to reproduce a desired outcome.

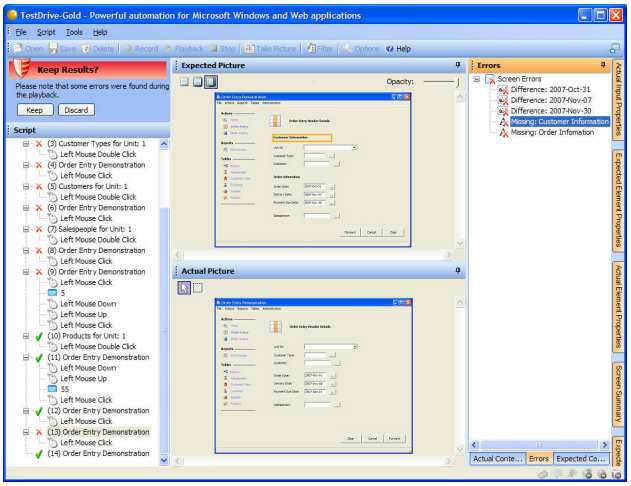
Regression Testing: Any type of software testing which seeks to uncover regression bugs. Regression bugs occur whenever software functionality that previously worked as desired, stops working or no longer works in the same way that was previously planned. Typically regression bugs occur as an unintended consequence of program changes. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged.

Experience has shown that as software is developed, this kind of reemergence of faults is quite common. Sometimes it occurs because a fix gets lost through poor revision control practices (or simple human error in revision control), but often a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Finally, it has often been the case that when some feature is redesigned, the same mistakes will be made in the redesign that were made in the original implementation of the feature.

Therefore, in most software development situations it is considered good practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program. Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools.

Such a 'test suite' contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any regressions. Common strategies are to run such a system after every successful compile (for small projects), every night, or once a week. Those strategies can be automated by an external tool, such as TestDrive-Gold from Original Software.

Fig. 2 A regression test performed in TestDrive-Gold



Relational Operator: Conditions such as “is equal to” or “is less than” that link an attribute name with an attribute value in a rule’s premise to form logical expressions that can be evaluated true or false.

Release Candidate: A pre-release version, which contains the desired functionality of the final version, but which needs to be tested for bugs (which ideally should be removed before the final version is released).

Release Note: A document identifying test items, their configuration, current status and other delivery information delivered by development to testing, and possibly other stakeholders, at the start of a test execution phase.

Reliability: The ability of the system/software to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations.

Reliability Requirements: A specification of the required reliability for the system/software.

Reliability Testing: Testing to determine whether the system/software meets the specified reliability requirements.

Requirement: A capability that must be met or possessed by the system/software (requirements may be functional or non-functional).

Requirements-based Testing: An approach to testing in which test cases are designed based on test objectives and test conditions derived from requirements. For example: tests that exercise specific functions or probe non-functional attributes such as reliability or usability.

Result: The consequence or outcome of a test.

Review: A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users or other interested parties for comment or approval.

Risk: A chance of negative consequences.

Risk Management: Systematic application of procedures and practices to the tasks of identifying, analyzing, prioritizing, and controlling risk.

Robustness: The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions.

Root Cause: An underlying factor that caused a non-conformance and possibly should be permanently eliminated through process improvement.

Rule: A statement of the form: if X then Y else Z. The “if” part is the rule premise, and the “then” part is the consequent. The “else” component of the consequent is optional. The rule fires when the if part is determined to be true or false.

Rule Base: The encoded knowledge for an expert system. In a rule-based expert system, a knowledge base typically incorporates definitions of attributes and rules along with control information.

Safety Testing: The process of testing to determine the safety of a software product.

Sanity Testing: Brief test of major functional elements of a piece of software to determine if it's basically operational. See also Smoke Testing.

Scalability Testing: Performance testing focused on ensuring the application under test gracefully handles increases in work load.

Schedule: A scheme for the execution of test procedures. The test procedures are included in the test execution schedule in the order in which they are to be executed.

Scrambling: Data obfuscation routine to de-identify sensitive data in test data environments to meet the requirements of the Data Protection Act and other legislation. See TestBench.

Scribe: The person who has to record each defect mentioned and any suggestions for improvement during a review meeting, on a logging form. The scribe has to make ensure that the logging form is understandable.

Script: See Test Script.

Security: Preservation of availability, integrity and confidentiality of information:

- Availability is ensuring that authorized users have access to information and associated assets when required.
- Integrity is safeguarding the accuracy and completeness of information and processing methods.
- Confidentiality is ensuring that information is accessible only to those authorized to have access.

Security Requirements: A specification of the required security for the system or software.

Security Testing: Process to determine that an IS (Information System) protects data and maintains functionality as intended.

The six basic concepts that need to be covered by security testing are: confidentiality, integrity, authentication, authorization, availability and non-repudiation.

Confidentiality
A security measure which protects against the disclosure of information to parties other than the intended recipient(s). Often ensured by means of encoding using adefined algorithm and some secret information known only to the originator of the information and the intended recipient(s) (a process known as cryptography) but that is by no means the only way of ensuring confidentiality.

Integrity
A measure intended to allow the receiver to determine that the information which it receives has not been altered in transit or by other than the originator of the information.
Integrity schemes often use some of the same underlying technologies as confidentiality schemes, but they usually involve adding additional information to a communication to form the basis of an algorithmic check rather than the encoding all of the communication.

Authentication
A measure designed to establish the validity of a transmission, message, or originator.
Allows a receiver to have confidence that information it receives originated from a specific known source.

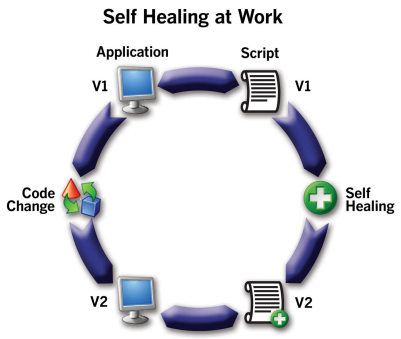
Authorization
The process of determining that a requester is allowed to receive a service or perform an operation.

Availability
Assuring information and communications services will be ready for use when expected.
Information must be kept available to authorized persons when they need it.

Non-repudiation
A measure intended to prevent the later denial that an action happened, or a communication that took place etc. In communication terms this often involves the interchange of authentication information combined with some form of provable time stamp.

Self-Healing Scripts: A next generation technique pioneered by Original Software which enables an existing test to be run over an updated or changed application, and intelligently modernize itself to reflect the changes in the application – all through a point-and-click interface.

Fig 3. Self Healing Scripts.



Simple Subpath: A subpath of the control flow graph in which no program part is executed more than necessary.

Simulation: The representation of selected behavioral characteristics of one physical or abstract system by another system.

Simulator: A device, computer program or system used during software verification, which behaves or operates like a given system when provided with a set of controlled inputs.

Smoke Testing: A preliminary to further testing, which should reveal simple failures severe enough to reject a prospective software release. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire. In the software world, the smoke is metaphorical.

Soak Testing: Involves testing a system with a significant load extended over a significant period of time, to discover how the system behaves under sustained use.

For example, in software testing, a system may behave exactly as expected when tested for 1 hour. However, when it is tested for 3 hours, problems such as memory leaks cause the system to fail or behave randomly.

Soak tests are used primarily to check the reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold. Observations made during the soak test are used to improve the characteristics of the subject under test further.

Software Requirements Specification: A deliverable that describes all data, functional and behavioral requirements, all constraints, and all validation requirements for software.

Software Testing: The process used to measure the quality of developed computer software. Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding errors. Quality is not an absolute; it is value to some person.

With that in mind, testing can never completely establish the correctness of arbitrary computer software; testing furnishes a *criticism* or *comparison* that compares the state and behaviour of the product against a specification. An important point is that *software testing* should be distinguished from the separate discipline of *Software Quality Assurance* (SQA), which encompasses all business process areas, not just testing.

Today, software has grown in complexity and size. The software product developed by a developer is according to the System Requirement Specification. Every software product has a target audience. For example, a video game software has its audience completely different from banking software. Therefore, when an organization invests large sums in making a software product, it must ensure that the software product must be acceptable to the end users or its target audience. This is where Software Testing comes into play. Software testing is not merely finding defects or bugs in the software, it is the completely dedicated discipline of evaluating the quality of the software.

There are many approaches to software testing, but effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following routine procedure. One definition of testing is "the process of questioning a product in order to evaluate it", where the "questions" are operations the tester attempts to execute with the product, and the product answers with its behavior in reaction to the probing of the tester. Although most of the intellectual processes of testing are nearly identical to that of review or inspection, the word testing is also used to connote the dynamic analysis of the product—putting the product through its paces. Sometimes one therefore refers to reviews, walkthroughs or inspections as "static testing", whereas actually running the program with a given set of test cases in a given development stage is often referred to as "dynamic testing", to emphasize the fact that formal review processes form part of the overall testing scope.

Specification: A description, in any suitable form, of requirements.

Specification testing: An approach to testing wherein the testing is restricted to verifying the system/software meets an agreed specification.

Specified input: An input for which the specification predicts an outcome.

State Transition: A transition between two allowable states of a system or component.

State Transition Testing: A test case design technique in which test cases are designed to execute state transitions.

Statement: An entity in a programming language which is typically the smallest indivisible unit of execution.

Statement Coverage: The percentage of executable statements in a component that have been exercised by a test case suite.

Statement Testing: A test case design technique for a component in which test cases are designed to execute statements. Statement Testing is a structural or white box technique, because it is conducted with reference to the code. Statement testing comes under Dynamic Analysis.

In an ideal world every statement of every component would be fully tested. However, in the real world this hardly ever happens. In statement testing every possible statement is tested. Compare this to Branch Testing, where each branch is tested, to check that it can be traversed, whether it encounters a statement or not.

Static Analysis: Analysis of a program carried out without executing the program.

Static Analyzer: A tool that carries out static analysis.

Static Code Analysis: The analysis of computer software that is performed without actually executing programs built from that software. In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding or program comprehension.

Static Testing: A form of software testing where the software isn't actually used. This is in contrast to Dynamic testing. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code or and manually reading of the code or document to find errors. This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used.

From the black box testing point of view, static testing involves review of requirements or specifications. This is done with an eye toward completeness or appropriateness for the task at hand. This is the verification portion of Verification and Validation. Bugs discovered at this stage of development are normally less expensive to fix than later in the development cycle.

Statistical Testing: A test case design technique in which a model is used of the statistical distribution of the input to construct representative test cases.

Storage Testing: Testing that verifies the program under test stores data files in the correct directories and that it reserves sufficient space to prevent unexpected termination resulting from lack of space. This is external storage as opposed to internal storage. See TestBench

Stress Testing: Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. Often this is performance testing using a very high level of simulated load.

Structural Coverage: Coverage measures based on the internal structure of the component.

Structural Test Case Design: Test case selection that is based on an analysis of the internal structure of the component.

Structural Testing: See structural test case design.

Structured Basis Testing: A test case design technique in which test cases are derived from the code logic to achieve % branch coverage.

Structured Walkthrough: See walkthrough.

Stub: A skeletal or special-purpose implementation of a software module, used to develop or test a component that calls or is otherwise dependent on it.

Subgoal: An attribute which becomes a temporary intermediate goal for the inference engine. Subgoal values need to be determined because they are used in the premise of rules that can determine higher level goals.

Subpath: A sequence of executable statements within a component.

Suitability: The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.

Suspension Criteria: The criteria used to (temporarily) stop all or a portion of the testing activities on the test items.

Symbolic Evaluation: See symbolic execution.

Symbolic Execution: A static analysis technique used to analyse if and when errors in the code may occur. It can be used to predict what code statements do to specified inputs and outputs. It is also important for considering path traversal. It struggles when dealing with statements which are not purely mathematical.

Symbolic Processing: Use of symbols, rather than numbers, combined with rules-of-thumb (or heuristics), in order to process information and solve problems.

Syntax Testing: A test case design technique for a component or system in which test case design is based upon the syntax of the input.

System Testing: Testing that attempts to discover defects that are properties of the entire system rather than of its individual components. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

TMM (Testing Maturity Model): A model developed by Dr Ilene Bernstein of the Illinois Institute of Technology, for judging the maturity of the software testing processes of an organization and for identifying the key practices that are required to increase the maturity of these processes.

Such a maturity model provides:

- A place to start
- The benefit of a community's prior experiences
- A common language and a shared vision
- A framework for prioritizing actions
- A way to define what improvement means for your organization.

A maturity model can be used as a benchmark for assessing different organizations for equivalent comparison. The model describes the maturity of the company based upon the project the company is handling and the related clients.

Level 1 - Initial

At *maturity level 1*, processes are usually ad hoc, and the organization usually does not provide a stable environment. Success in these organizations depends on the competence and heroics of the people in the organization, and not on the use of proven processes. In spite of this ad hoc, chaotic environment, maturity level 1 organizations often produce products and services that work; however, they frequently exceed the budget and schedule of their projects.

Maturity level 1 organizations are characterized by a tendency to over commit, abandon processes in the time of crisis, and not be able to repeat their past successes again.

Level 1 software project success depends on having high quality people.

Level 2 - Repeatable [Managed]

At maturity level 2, software development successes are repeatable and the organization may use some basic project management to track costs and schedule.

Process discipline helps ensure that existing practices are retained during times of stress. When these practices are in place, projects are performed and managed according to their documented plans.

Project status and the delivery of services are visible to management at defined points (for example, at major milestones and at the completion of major tasks).

Basic project management processes are established to track cost, schedule, and functionality. The minimum process discipline is in place to repeat earlier successes on projects with similar applications and scope. There is however, still a significant risk of exceeding cost and time estimates.

Level 3 - Defined

The organization's set of standard processes, which are the basis for level 3, are established and improved over time. These standard processes are used to establish consistency across the organization.

The organization's management establishes process objectives for the organization's set of standard processes, and ensures that these objectives are appropriately addressed.

A critical distinction between level 2 and level 3 is the scope of standards, process descriptions, and procedures. At level 2, the standards, process descriptions, and procedures may be quite different in each specific instance of the process (for example, on each particular project). At level 3, the standards, process descriptions, and procedures for a project are tailored from the organization's set of standard processes to suit a particular project or organizational unit.

Effective project management is implemented with the help of good project management software such as TestPlan from Original Software.

Level 4 - Quantitatively Managed

Using precise measurements, management can effectively control the software development effort. In particular, management can identify ways to adjust and adapt the process to particular projects without measurable losses of quality or deviations from specifications. Organizations at this level set quantitative quality goals for both software process and software maintenance. Sub processes are selected that significantly contribute to overall process performance. These selected sub processes are controlled using statistical and other quantitative techniques. A critical distinction between maturity level 3 and maturity level 4 is the predictability of process performance. At maturity level 4, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. At maturity level 3, processes are only qualitatively predictable.

Level 5 - Optimizing

Maturity level 5 focuses on continually improving process performance through both incremental and innovative technological improvements. Quantitative process-improvement objectives for the organization are established, continually revised to reflect changing business objectives, and used as criteria in managing process improvement. The effects of deployed process improvements are measured and evaluated against the quantitative process-improvement objectives. Both the defined processes and the organization's set of standard processes are targets of measurable improvement activities.

Process improvements to address common causes of process variation and measurably improve the

organization's processes are identified, evaluated, and deployed.

Optimizing processes that are nimble, adaptable and innovative depends on the participation of an empowered workforce aligned with the business values and objectives of the organization. The organization's ability to rapidly respond to changes and opportunities is enhanced by finding ways to accelerate and share learning.

A critical distinction between maturity level 4 and maturity level 5 is the type of process variation addressed. At maturity level 4, processes are concerned with addressing special causes of process variation and providing statistical predictability of the results. Though processes may produce predictable results, the results may be insufficient to achieve the established objectives. At maturity level 5, processes are concerned with addressing common causes of process variation and changing the process (that is, shifting the mean of the process performance) to improve process performance (while maintaining statistical probability) to achieve the established quantitative process-improvement objectives.

Technical Review: A peer group discussion activity that focuses on achieving consensus on the technical approach to be taken. A technical review is also known as a peer review.

Test Approach: The implementation of the test strategy for a specific project. It typically includes the decisions made that follow based on the (test) project's goal and the risk assessment carried out, starting points regarding the test process and the test design techniques to be applied.

Test Automation: The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions. Commonly, test automation involves automating a manual process already in place that uses a formalized testing process.

Over the past few years, tools that help programmers quickly create applications with graphical user interfaces have dramatically improved programmer productivity. This has increased the pressure on testers, who are often perceived as bottlenecks to the delivery of software products. Testers are being asked to test more and more code in less and less time. Test automation is one way to do this, as manual testing is time consuming. As and when different versions of software are released, the new features will have to be tested manually time and again. But, now there are tools available that help the testers in the automation of the GUI which reduce the test time as well as the cost; other test automation tools support execution of performance tests.

Many test automation tools provide record and playback features that allow users to record interactively user actions and replay it back any number of times, comparing actual results to those expected. However, reliance on these features poses major reliability and maintainability problems. Most successful automators use a software engineering approach, and as such most serious test automation is undertaken by people with development experience. See Partial Test Automation.

Test Bed: An execution environment configured for testing. May consist of specific hardware, OS, network topology, configuration of the product under test, other application or system software, etc. The Test Plan for a project should enumerate the test beds(s) to be used.

TestBench: A suite of test automation solutions from Original Software that facilitate the management and manipulation of the database and visual layer components. TestBench addresses test verification, disk space, and data confidentiality issues. In addition, control of test data ensures that every test starts with a consistent data state, essential if the data is to be predictable at the end of testing.

Test Case: A commonly used term for a specific test. This is usually the smallest unit of testing. A Test Case will consist of information such as requirements testing, test steps, verification steps, prerequisites, outputs, test environment, etc.

An alternate definition for Test Case is a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Case Design Technique: A method used to derive or select test cases.

Test Case Suite: A collection of one or more test cases for the software under test.

Test Charter: A statement of test objectives, and possibly test ideas. Test charters are amongst other used in exploratory testing.

Test Comparator: A test tool that compares the actual outputs produced by the software under test with the expected outputs for that test case.

Test Comparison: The process of identifying differences between the actual results produced by the component or system under test and the expected results for a test. Test comparison can be performed during test execution (dynamic comparison) or after test execution.

Test Completion Criterion: A criterion for determining when planned testing is complete, defined in terms of a test measurement technique.

Test Data: Data that exists (for example, in a database) before a test is executed, and that affects or is affected by the component or system under test.

Test Data Management: The management of test data during tests to ensure complete data integrity and legitimacy from the beginning to the end of test. See TestBench

Test-Drive Assist: A new concept in testing from Original Software that delivers active support for manual testing by compiling history on recent testing, making it easy to recreate and isolate software defects. With powerful tracking functions that operate in a non intrusive and natural fashion, testers can detect and highlight defects more quickly and effectively resulting in developers correcting these defects fast and efficiently. This means defects can simply be re-created at a touch of a button, whilst the results can eventually be used to build fully automated tests.

Test-Drive: Next Generation automated testing solution from Original Software that allows technicians to define and execute sophisticated tests, without being hindered by complex programming languages. State of the art self updating technology automatically adapts tests to new software releases and upgrades.

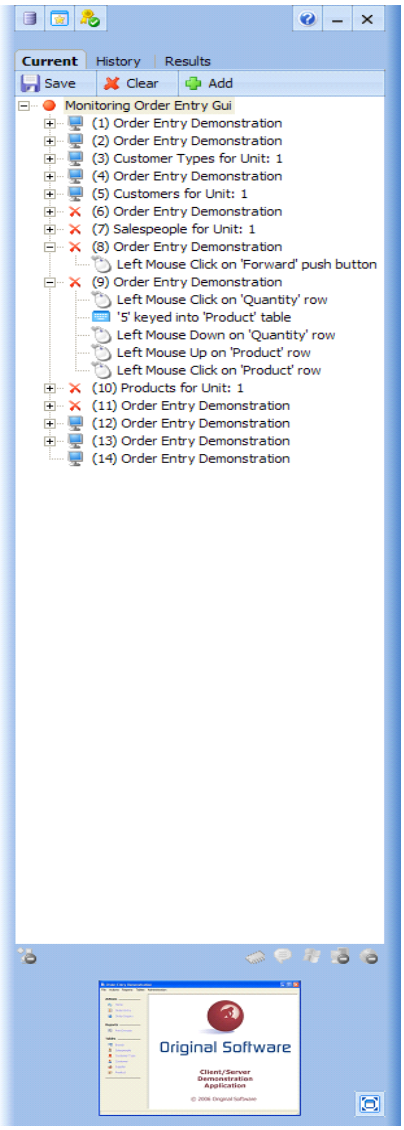
Test Driven Development: Testing methodology associated with Agile Programming in which every chunk of code is covered by unit tests which must all pass all the time, in an effort to eliminate unit-level and regression bugs during development. Practitioners of TDD write a lot of tests, i.e. an equal number of lines of test code to the size of the production code.

Test Driver: A program or test tool used to execute software against a test case suite.

Test Environment: A description of the hardware and software environment in which the tests will be run and any other software with which the software under test interacts when under test including stubs and test drivers.

Test Execution: The process of carrying out a test, whether it be manually or using automated test software.

Fig 3. Main view in TestDrive-Assist showing screens and input. Those screens with a red cross contain errors.



Test Execution Phase: The period of time in the application development life cycle during which the components of a software product are executed, and the software product is evaluated to determine whether or not requirements have been satisfied.

Test Execution Schedule: A scheme for the execution of test procedures. The test procedures are included in the test execution schedule in their context and in the order in which they are to be executed.

Test Execution Technique: The method used to perform the actual test execution, e.g. manual, capture/playback tool, etc.

Test First Design: Test-first design is one of the mandatory practices of Extreme Programming (XP).It requires that programmers do not write any production code until they have first written a unit test.

Test Generator: A program that generates test cases in accordance to a specified strategy.

Test Harness: A program or test tool used to execute a test. Also known as a Test Driver.

Test Infrastructure: The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment and procedures.

Test Level: A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test.

Test Log: A chronological record of relevant details about the execution of tests.

Test Measurement Technique: A method used to measure test coverage items.

Test Object: The component/system/application to be tested.

Test Plan: A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.

TestPlan: A planning solution from Original Software that enables effective planning, team communications, and accurate tracking of every testing activity.

Test Point Analysis: A formula based test estimation method based on function point analysis.

Test Procedure: A document providing detailed instructions for the execution of one or more test cases.

Test Records: For each test, an unambiguous record of the identities and versions of the component under test, the test specification, and actual outcome.

Test Run: Execution of a test on a specific version of the test object.

Test Scenario: Definition of a set of test cases or test scripts and the sequence in which they are to be executed.

Test Script: A test script is the traditional but somewhat outdates way of building test cases. A short program written in a programming language is used to test part of the functionality of a software system. Such scripts underpin the basis of all of the older type automation tools, however these are cumbersome and difficult to use and update, and the scripts themselves often have errors in them. See Code-Free Testing.

Test Specification: A document specifying the test approach for a software feature or combination or features and the inputs, predicted results and execution conditions for the tests.

Test Strategy: A high-level document defining the test levels to be performed and the testing within those levels for a program (one or more projects).

Test Suite: A collection of tests used to validate the behavior of a product. The scope of a Test Suite varies from organization to organization. There may be several Test Suites for a particular product for example. In most cases however a Test Suite is a high level concept, grouping together hundreds or thousands of tests related by what they are intended to test.

Test Target: A set of test completion criteria for the test.

Test Type: A group of test activities aimed at testing a component or system regarding one or more interrelated quality attributes. A test type is focused on a specific test objective, i.e. reliability test, usability test, regression test etc., and may take place on one or more test levels or test phases.

Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

Tester: A person (either a professional tester or a user) who is involved in the testing of a component or system.

Testing:

- The process of exercising software to verify that it satisfies specified requirements and to detect errors.
- 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 item.
- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

Test Tools: Computer programs used in the testing of a system, a component of the system, or its documentation.

Thread Testing: A version of component integration testing where the progressive integration of components follows the implementation of subsets of the requirements, as opposed to the integration of components by levels of a hierarchy.

Top Down Testing: An approach to integration testing where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been tested.

Test Driven Development: A software development technique consisting of short iterations where new test cases covering the desired improvement or new functionality are written first, then the production code necessary to pass the tests is implemented, and finally the software is re factored to accommodate the changes. The availability of tests before actual development ensures rapid feedback after any change. Practitioners emphasize that test-driven development is a method of designing software, not merely a method of testing. (i.e. avoid designing software that is difficult to test).

Total Quality Management: A company commitment to develop a process that achieves high quality product and customer satisfaction.

Traceability: The ability to identify related items in documentation and software, such as requirements with associated tests.

Traceability Matrix: A document showing the relationship between Test Requirements and Test Cases.

Tracked Field: A value captured in one part of an automated test process and retained for use at a later stage.

Timeline: A chronological representation of components in a result set, combining different areas of test analysis, such as visual, database, messages and linked programs. See TestBench.

Total Testing: A complete approach to testing from Original Software describing the desire to optimize all the steps in the testing process (e.g. From unit testing to UAT), as well as the different layers of the application's architecture. (e.g. From the user interface to the database).

U

Understandability: The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.

Unit Testing: A procedure used to validate that individual units of source code are working properly. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a method; which may belong to a base/super class, abstract class or derived/child class.

Ideally, each test case is independent from the others; mock objects and test harnesses can be used to assist testing a module in isolation. Unit testing is typically done by developers and not by end-users.

Unreachable Code: Code that cannot be reached and therefore is impossible to execute.

Unstructured Decisions: This type of decision situation is complex and no standard solutions exist for resolving the situation. Some or all of the structural elements of the decision situation are undefined, ill-defined or unknown.

Usability: The capability of the software to be understood, learned, used and attractive to the user when used under specified conditions.

Usability requirements: A specification of the required usability for the system/software.

Usability Testing: Testing to determine whether the system/software meets the specified usability requirements.

Use Case: The specification of tests that are conducted from the end-user perspective. Use cases tend to focus on operating software as an end-user would conduct their day-to-day activities.

Use Case Testing: A black box test design technique in which test cases are designed to execute user scenarios.

User Acceptance Testing: Black-box testing performed on a system prior to its delivery. In most environments, acceptance testing by the system provider is distinguished from acceptance testing by the customer (the user or client) prior to accepting transfer of ownership. In such environments, acceptance testing performed by the customer is known as beta testing, user acceptance testing (UAT), end user testing, site (acceptance) testing, or acceptance testing.

Unit Testing: Testing of individual software components.

V

V Model: Describes how inspection and testing activities can occur in parallel with other activities.

Validation Testing: Determination of the correctness of the products of software development with respect to the user needs and requirements.

Variable Data: A repository for multiple scenario values which can be used to drive repeatable automated processes through a number of iterations when used in conjunction with an automation solution such as TestDrive.

Verification: The process of evaluating a system or component to determine its completeness.

Version Identifier: A version number; version date, or version date and time stamp.

Volume Testing: Testing which confirms that any values that may become large over time (such as accumulated counts, logs, and data files), can be accommodated by the program and will not cause the program to stop working or degrade its operation in any manner.

W

Walkthrough: A review of requirements, designs or code characterized by the author of the material under review guiding the progression of the review.

Warping: The capability to manipulate dates in data to simulate data of different ages in support of testing date-driven systems. Found in TestBench

Waterline: The lowest level of detail relevant to the Customer.

What If Analysis: The capability of "asking" the software package what the effect will be of changing some of the input data or independent variables.

White Box Testing: (a.k.a. clear box testing, glass box testing or structural testing). Uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs.

While white box testing is applicable at the unit, integration and system levels of the software testing process, it's typically applied to the unit. So while it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements. But you can be sure that all paths through the test object are executed.

Wide Band Delphi: A consensus-based estimation technique for estimating effort. It was developed in the 1940s at the RAND Corporation as a forecasting tool. It has since been adapted across many industries to estimate many kinds of tasks, ranging from statistical data collection results to sales and marketing forecasts. It has proven to be a very effective estimation tool, and it lends itself well to software projects. However, many see great problems with the technique, such as unknown manipulation of a group and silencing of minorites in order to see a preset outcome of a meeting.

Workaround: Method of avoiding an incident or problem, either from a temporary fix or from a technique that means the Customer is not reliant on a particular aspect of a service that is known to have a problem.

Workflow Testing: Scripted end-to-end testing which duplicates specific workflows which are expected to be utilized by the end-user.

X, Y & Z

XML: Extensible Markup Language. XML is a set of rules for designing text formats that let you structure your data. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML avoids common pitfalls in language design: it is extensible, platform-independent, and it supports internationalization and localization.

Bibliography:

The Software Test Management Guide (www.ruleworks.com)
The Software Testing Glossary (www.aptest.com)
The International Software Testing Qualifications Board (www.istqb.org)
The Testing Standards Working Party (www.testingstandards.co.uk)
Wikipedia (wikipedia.org)
IEEE (www.ieee.org)

EMEA OFFICE
Original Software Ltd
Grove House
Chineham Court
Basingstoke
Hampshire
RG24 8AG

Phone: +44 (0) 1256 338666
Fax: +44 (0) 1256 338678

USA OFFICE
Original Software Inc
601 Oakmont Lane
Suite 170
Westmont
IL 60559
USA

Phone: 630-321-0092
Fax: 630-321-0223

solutions@origsoft.com
www.origsoft.com

OSG-GST-10/07

Original Software