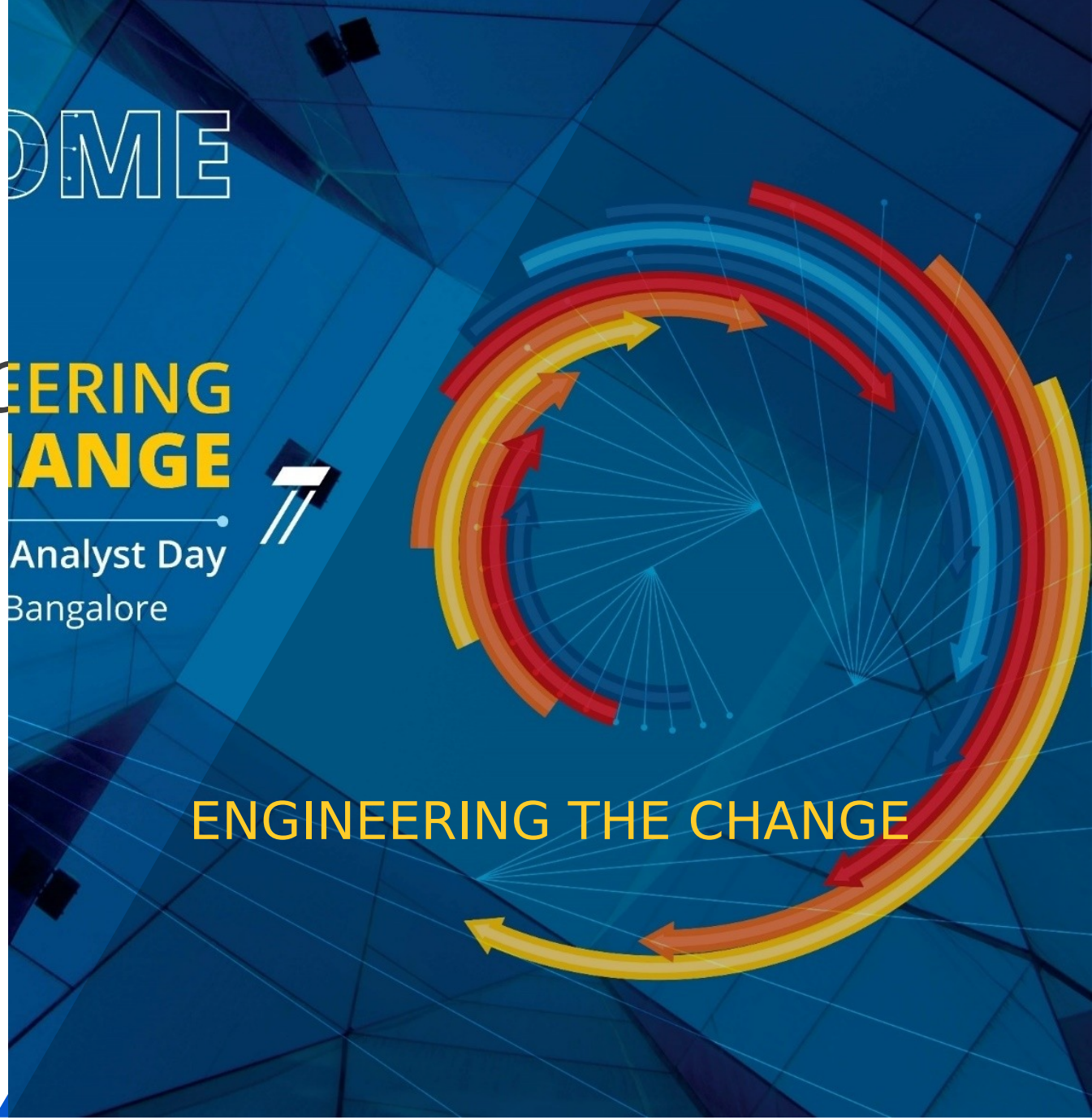
Manual Testing / TDL

L&T Technology Services



Date

Contents

STATIC TECHNIQUES

TEST DESIGN TECHNIQUES

Static Testing

Almost any work product can be examined using static testing (reviews and/or static analysis), for example:

- Specifications, including business requirements, functional requirements, and security requirements
- Epics, user stories, and acceptance criteria
- Architecture and design specifications
- Code
- Testware, including test plans, test cases, test procedures, and automated test scripts
- User guides
- Web pages
- Contracts, project plans, schedules, and budget planning
- Configuration set up and infrastructure set up
- Models, such as activity diagrams, which may be used for Model-Based testing

Reviews can be applied to any work product that the participants know how to read and understand.
Static analysis can be applied efficiently to any work product with a formal structure (typically code or
models) for which an appropriate static analysis tool exists. Static analysis can even be applied with tools
that evaluate work products written in natural language such as requirements (e.g., checking

Static Testing (contd.)

Benefits of Static Testing

Static testing techniques provide a variety of benefits.

- When applied early in the software development lifecycle, static testing enables the early detection of defects before dynamic testing is performed (e.g., in requirements or design specifications reviews, backlog refinement, etc.).
- Defects found early are often much cheaper to remove than defects found later in the lifecycle, especially compared to defects found after the software is deployed and in active use.
- Using static testing techniques to find defects and then fixing those defects promptly is almost always much cheaper for the organization than using dynamic testing to find defects in the test object and then fixing them, especially when considering the additional costs associated with updating other work products and performing confirmation and regression testing.

Additional benefits of static testing may include:

- Detecting and correcting defects more efficiently, and prior to dynamic test execution
- Identifying defects which are not easily found by dynamic testing
- Preventing defects in design or coding by uncovering inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies in requirements
- Increasing development productivity (e.g., due to improved design, more maintainable code)
- Reducing development cost and time
- Reducing testing cost and time
- Reducing total cost of quality over the software's lifetime, due to fewer failures later in the lifecycle or after delivery into operation
- Improving communication between team members in the course of participating in reviews

Static Testing (contd.)

Differences between Static and Dynamic Testing

Static testing and dynamic testing can have the same objectives, such as providing an assessment of the quality of the work products and identifying defects as early as possible. Static and dynamic testing complement each other by finding different types of defects.

One main distinction is that static testing finds defects in work products directly rather than identifying failures caused by defects when the software is run. A defect can reside in a work product for a very long time without causing a failure. The path where the defect lies may be rarely exercised or hard to reach, so it will not be easy to construct and execute a dynamic test that encounters it. Static testing may be able to find the defect with much less effort.

Another distinction is that static testing can be used to improve the consistency and internal quality of work products, while dynamic testing typically focuses on externally visible behaviours. Compared with dynamic testing, typical defects that are easier and cheaper to find and fix through static testing include:

- Requirement defects (e.g., inconsistencies, ambiguities, contradictions, omissions, inaccuracies, and redundancies)
- Design defects (e.g., inefficient algorithms or database structures, high coupling, low cohesion)
- Coding defects (e.g., variables with undefined values, variables that are declared but never used, unreachable code, duplicate code)
- Deviations from standards (e.g., lack of adherence to coding standards)
- Incorrect interface specifications (e.g., different units of measurement used by the calling system than by the called system)
- Security vulnerabilities (e.g., susceptibility to buffer overflows)
- Gaps or inaccuracies in test basis traceability or coverage (e.g., missing tests for an acceptance criterion)

Review Process

Reviews vary from informal to formal. Informal reviews are characterized by not following a defined process and not having formal documented output. Formal reviews are characterized by team participation, documented results of the review, and documented procedures for conducting the review. The formality of a review process is related to factors such as the software development lifecycle model, the maturity of the development process, the complexity of the work product to be reviewed, any legal or regulatory requirements, and/or the need for an audit trail.

The focus of a review depends on the agreed objectives of the review (e.g., finding defects, gaining understanding, educating participants such as testers and new team members, or discussing and deciding by consensus).

Work Product Review Process

The review process comprises the following main activities:

Planning

- Defining the scope, which includes the purpose of the review, what documents or parts of documents to review, and the quality characteristics to be evaluated
- Estimating effort and timeframe
- Identifying review characteristics such as the review type with roles, activities, and checklists
- Selecting the people to participate in the review and allocating roles
- Defining the entry and exit criteria for more formal review types (e.g., inspections)
- Checking that entry criteria are met (for more formal review types)

Review Process (contd.)

Initiate review
• Distributing the work product (physically or by electronic means) and other material, such as issue log forms, checklists, and related work products
• Explaining the scope, objectives, process, roles, and work products to the participants
• Answering any questions that participants may have about the review

Individual review (i.e., individual preparation)
• Reviewing all or part of the work product
• Noting potential defects, recommendations, and questions

Issue communication and analysis
• Communicating identified potential defects (e.g., in a review meeting)
• Analyzing potential defects, assigning ownership and status to them
• Evaluating and documenting quality characteristics
• Evaluating the review findings against the exit criteria to make a review decision (reject; major changes needed; accept, possibly with minor changes)

Fixing and reporting
• Creating defect reports for those findings that require changes to a work product
• Fixing defects found (typically done by the author) in the work product reviewed
• Communicating defects to the appropriate person or team (when found in a work product related to the work product reviewed)
• Recording updated status of defects (in formal reviews), potentially including the agreement of the comment originator
• Gathering metrics (for more formal review types)
• Checking that exit criteria are met (for more formal review types)
• Accepting the work product when the exit criteria are reached

Review Process (contd.)

Roles and responsibilities in a formal review

A typical formal review will include the roles below:

Author

- Creates the work product under review
- Fixes defects in the work product under review (if necessary)

Management

- Is responsible for review planning
- Decides on the execution of reviews
- Assigns staff, budget, and time
- Monitors ongoing cost-effectiveness
- Executes control decisions in the event of inadequate outcomes

Facilitator (often called moderator)

- Ensures effective running of review meetings (when held)
- Mediates, if necessary, between the various points of view
- Is often the person upon whom the success of the review depends

Review leader

- Takes overall responsibility for the review
- Decides who will be involved and organizes when and where it will take place

Review Process (contd.)

Reviewers

• May be subject matter experts, persons working on the project, stakeholders with an interest in the work product, and/or individuals with specific technical or business backgrounds

• Identify potential defects in the work product under review

• May represent different perspectives (e.g., tester, developer, user, operator, business analyst, usability expert, etc.)

Scribe (or recorder)

Collates potential defects found during the individual review activity

Records new potential defects, open points, and decisions from the review meeting (when held)

In some review types, one person may play more than one role, and the actions associated with each role may also vary based on review type. In addition, with the advent of tools to support the review process, especially the logging of defects, open points, and decisions, there is often no need for a scribe.

Review Process (contd.)

Review Types

Although reviews can be used for various purposes, one of the main objectives is to uncover defects. All review types can aid in defect detection, and the selected review type should be based on the needs of the project, available resources, product type and risks, business domain, and company culture, among other selection criteria.

A single work product may be the subject of more than one type of review. If more than one type of review is used, the order may vary. For example, an informal review may be carried out before a technical review, to ensure the work product is ready for a technical review.

The types of reviews described below can be done as peer reviews, i.e., done by colleagues qualified to do the same work. The types of defects found in a review vary, depending especially on the work product being reviewed. Reviews can be classified according to various attributes. The following lists the four most common types of reviews and their associated attributes:.

Informal review (e.g., buddy check, pairing, pair review)

- Main purpose: detecting potential defects
- Possible additional purposes: generating new ideas or solutions, quickly solving minor problems
- Not based on a formal (documented) process
- May not involve a review meeting
- May be performed by a colleague of the author (buddy check) or by more people
- Results may be documented
- Varies in usefulness depending on the reviewers
- Use of checklists is optional
- Very commonly used in Agile development

Review Process (contd.)

Walkthrough

• Main purposes: find defects, improve the software product, consider alternative implementations, evaluate conformance to standards and specifications
• Possible additional purposes: exchanging ideas about techniques or style variations, training of participants, achieving consensus
• Individual preparation before the review meeting is optional
• Review meeting is typically led by the author of the work product
• Scribe is mandatory
• Use of checklists is optional
• May take the form of scenarios, dry runs, or simulations
• Potential defect logs and review reports are produced
• May vary in practice from quite informal to very formal

Technical review

• Main purposes: gaining consensus, detecting potential defects
• Possible further purposes: evaluating quality and building confidence in the work product, generating new ideas, motivating and enabling authors to improve future work products, considering alternative implementations
• Reviewers should be technical peers of the author, and technical experts in the same or other disciplines
• Individual preparation before the review meeting is required
• Review meeting is optional, ideally led by a trained facilitator (typically not the author)
• Scribe is mandatory, ideally not the author
• Use of checklists is optional
• Potential defect logs and review reports are produced

Review Process (contd.)

Inspection

• Main purposes: detecting potential defects, evaluating quality and building confidence in the work product, preventing future similar defects through author learning and root cause analysis
• Possible further purposes: motivating and enabling authors to improve future work products and the software development process, achieving consensus
• Follows a defined process with formal documented outputs, based on rules and checklists
• Uses clearly defined roles which are mandatory, and may include a dedicated reader (who reads the work product aloud often paraphrase, i.e. describes it in own words, during the review meeting)
• Individual preparation before the review meeting is required
• Reviewers are either peers of the author or experts in other disciplines that are relevant to the work product
• Specified entry and exit criteria are used
• Scribe is mandatory
• Review meeting is led by a trained facilitator (not the author)
• Author cannot act as the review leader, reader, or scribe
• Potential defect logs and review report are produced
• Metrics are collected and used to improve the entire software development process, including the inspection process

Review Process (contd.)

Applying Review Techniques

There are a number of review techniques that can be applied during the individual review (i.e., individual preparation) activity to uncover defects. These techniques can be used across the review types described above. The effectiveness of the techniques may differ depending on the type of review used. Examples of different individual review techniques for various review types are listed below.

Ad hoc

In an ad hoc review, reviewers are provided with little or no guidance on how this task should be performed. Reviewers often read the work product sequentially, identifying and documenting issues as they encounter them. Ad hoc reviewing is a commonly used technique needing little preparation. This technique is highly dependent on reviewer skills and may lead to many duplicate issues being reported by different reviewers.

Checklist-based

A checklist-based review is a systematic technique, whereby the reviewers detect issues based on checklists that are distributed at review initiation (e.g., by the facilitator). A review checklist consists of a set of questions based on potential defects, which may be derived from experience. Checklists should be specific to the type of work product under review and should be maintained regularly to cover issue types missed in previous reviews. The main advantage of the checklist-based technique is a systematic coverage of typical defect types. Care should be taken not to simply follow the checklist in individual reviewing, but also to look for defects outside the checklist.

Review Process (contd.)

Scenarios and dry runs
In a scenario-based review, reviewers are provided with structured guidelines on how to read through the work product. A scenario-based review supports reviewers in performing “dry runs” on the work product based on expected usage of the work product (if the work product is documented in a suitable format such as use cases). These scenarios provide reviewers with better guidelines on how to identify specific defect types than simple checklist entries. As with checklist-based reviews, in order not to miss other defect types (e.g., missing features), reviewers should not be constrained to the documented scenarios.

Perspective-based
In perspective-based reading, similar to a role-based review, reviewers take on different stakeholder viewpoints in individual reviewing. Typical stakeholder viewpoints include end user, marketing, designer, tester, or operations. Using different stakeholder viewpoints leads to more depth in individual reviewing with less duplication of issues across reviewers. In addition, perspective-based reading also requires the reviewers to attempt to use the work product under review to generate the product they would derive from it. For example, a tester would attempt to generate draft acceptance tests if performing a perspective-based reading on a requirements specification to see if all the necessary information was included. Further, in perspective-based reading, checklists are expected to be used.
Empirical studies have shown perspective-based reading to be the most effective general technique for reviewing requirements and technical work products. A key success factor is including and weighing different stakeholder viewpoints appropriately, based on risks.

Role-based
A role-based review is a technique in which the reviewers evaluate the work product from the perspective of individual stakeholder roles. Typical roles include specific end user types (experienced, inexperienced, senior, child, etc.), and specific roles in the organization (user administrator, system administrator,

Review Process (contd.)

Success Factors for Reviews

In order to have a successful review, the appropriate type of review and the techniques used must be considered. In addition, there are a number of other factors that will affect the outcome of the review.

Organizational success factors for reviews include:

- Each review has clear objectives, defined during review planning, and used as measurable exit criteria
- Review types are applied which are suitable to achieve the objectives and are appropriate to the type and level of software work products and participants
- Any review techniques used, such as checklist-based or role-based reviewing, are suitable for effective defect identification in the work product to be reviewed
- Any checklists used address the main risks and are up to date
- Large documents are written and reviewed in small chunks, so that quality control is exercised by providing authors early and frequent feedback on defects
- Participants have adequate time to prepare
- Reviews are scheduled with adequate notice
- Management supports the review process (e.g., by incorporating adequate time for review activities in project schedules)
- Reviews are integrated in the company's quality and/or test policies.

People-related success factors for reviews include:

- The right people are involved to meet the review objectives, for example, people with different skill sets or perspectives, who may use the document as a work input
- Testers are seen as valued reviewers who contribute to the review and learn about the work product, which enables them to prepare more effective tests, and to prepare those tests earlier
- Participants dedicate adequate time and attention to detail

Review Process (contd.)

- Reviews are conducted on small chunks, so that reviewers do not lose concentration during individual review and/or the review meeting (when held)
- Defects found are acknowledged, appreciated, and handled objectively
- The meeting is well-managed, so that participants consider it a valuable use of their time
- The review is conducted in an atmosphere of trust; the outcome will not be used for the evaluation of the participants
- Participants avoid body language and behaviours that might indicate boredom, exasperation, or hostility to other participants
- Adequate training is provided, especially for more formal review types such as inspections
- A culture of learning and process improvement is promoted

Static Analysis by Tools

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

• **Static analysis tools** are generally used by developers as part of the development and component testing process. The key aspect is that the code (or other artefact) is not executed or run but the tool itself is executed, and the source code we are interested in is the input data to the tool.
• These tools are **mostly used by developers.**
• Static analysis tools are an extension of compiler technology – in fact some compilers do offer static analysis features. It is worth checking what is available from existing compilers or development environments before looking at purchasing a more sophisticated static analysis tool.
• Other than software code, static analysis can also be carried out on things like, static analysis of requirements or static analysis of websites (for example, to assess for proper use of accessibility tags or the following of HTML standards).
• Static analysis tools for code can help the developers to understand the structure of the code, and can also be used to enforce coding standards.

Features or characteristics of static analysis tools are:
• To calculate metrics such as cyclomatic complexity or nesting levels (which can help to identify where more testing may be needed due to increased risk).
• To enforce coding standards.
• To analyze structures and dependencies.
• Help in code understanding.

Test Case Writing

A **test case** is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not.

Test cases are often referred to as test scripts, particularly when written. Written test cases are usually collected into test suites. The following testing items have close correlation with test cases:
Test Script is a detailed description of the test steps or transaction(s) to be performed to validate the system or application under test. The test script must contain the actual entries to be executed as well as the expected results.

Test Step is the lowest level of a test script that performs a specific operation such as clicking a button or entering data in a text box. Test steps are created for both manual and automated tests. Each test step must be followed by a description of the 'expected result' of the test step, after it has been executed.

Test Set is a collection of test scripts that are grouped for a specific purpose (i.e. business process, function/ feature).

Test Scenario is a high-level description of a business process or system functionality that will be tested. Detailed information such as input data, expected results, parameters, etc. will not be included in the test scenario, but will rather be located in the test script.
Example: Check Login Functionality.

Test Data Set is a specific set of values for variables in the communication space of a module, which are used in a test.

Test Procedures define the activities necessary to execute a test script or set of scripts. Test procedures

Test Case Writing (contd.)

Types of Test Cases

Requirement and Design based test cases
1.Identify the basic cases that indicate program functionality.
2.Create a minimal set of tests to cover all inputs & outputs.
3.Breakdown complex cases into single cases.
4.Remove unnecessary or duplicate cases.
5.Review systematically and thoroughly.
6.Design based test cases supplement requirements based test cases. Etc.

Code Based test cases
1.Identify test cases with that every statement in a code exercised at least once.
2.Every decision exercised over all outcomes. Etc.

Extreme test cases
1.Looks for exceptional conditions, extremes, boundaries, and abnormalities.

Extracted and Randomized test cases
1.Extracted cases involved extracting samples of real data for the testing process.
2.Randomized cases involved using tools to generate potential data for the testing process.

Test Case Writing (contd.)

Positive Test Cases and Negative Test Cases:

Positive Testing = (Not showing error when not supposed to) + (Showing error when supposed to) So if either of the situations in parentheses happens you have a positive test in terms of its result - not what the test was hoping to find. The application did what it was supposed to do. Here user tends to put all positive values according to requirements.

Negative Testing = (Showing error when not supposed to) + (Not showing error when supposed to)(Usually these situations crop up during boundary testing or cause-effect testing.) Here if either of the situations in parentheses happens you have a negative test in terms of its result - again, not what the test was hoping to find. The application did what it was not supposed to do. User tends to put negative values, which may crash the application.

For example in Registration Form, for Name field, user should be allowed to enter only alphabets. Here for Positive Testing, tester will enter only alphabets and application should run properly and should accept only alphabets. For Negative Testing, in the same case user tries to enter numbers, special characters and if the case is executed successfully, negative testing is successful.

Test Case Writing (contd.)

Test Case Development Process:
- Identify all potential Test Cases needed to fully test the business and technical requirements
- Document Test Procedures
- Document Test Data requirements
- Prioritize test cases
- Identify Test Automation Candidates
- Automate designated test cases

Test Case Design Methods:
1.Boundary Value Analysis
2.Equivalence Partitioning
3.Error Guessing

Characteristics of good test cases:
- Simple and specific. Any one in the test team should be able to execute the test cases without the author help.
- Clear, concise, and complete
- No assumptions in test case description, steps, expected result etc.
- Non-redundant
- Reasonable probability of catching an error
- Medium complexity
- Repeatable
- Test cases that have written with the help of test case design methods
- Test cases that are easily identifiable with their names.
- Always list expected results

Test Case Writing (contd.)

Following is the possible list of functional and non-functional test cases for a login page:

Functional Test Cases:

Sr. No.	Functional Test Cases	Type- Negative/ Positive Test Case
1	Verify if a user will be able to login with a valid username and valid password.	Positive
2	Verify if a user cannot login with a valid username and an invalid password.	Negative
3	Verify the login page for both, when the field is blank and Submit button is clicked.	Negative
4	Verify the ‘Forgot Password’ functionality.	Positive
5	Verify the messages for invalid login.	Positive
6	Verify the ‘Remember Me’ functionality.	Positive
7	Verify if the data in password field is either visible as asterisk or bullet signs.	Positive
8	Verify if a user is able to login with a new password only after he/she has changed the password.	Positive
9	Verify if the login page allows to log in simultaneously with different credentials in a different browser.	Positive
10	Verify if the ‘Enter’ key of the keyboard is working correctly on the login page.	Positive
	Other Test Cases	
11	Verify the time taken to log in with a valid username and password.	Performance & Positive Testing
12	Verify if the font, text color, and color coding of the Login page is as per the standard.	UI Testing & Positive Testing
13	Verify if there is a ‘Cancel’ button available to erase the entered text.	Usability Testing
14	Verify the login page and all its controls in different browsers	Browser Compatibility & Positive Testing.

Test Case Writing (contd.)

Non-functional Security Test Cases:

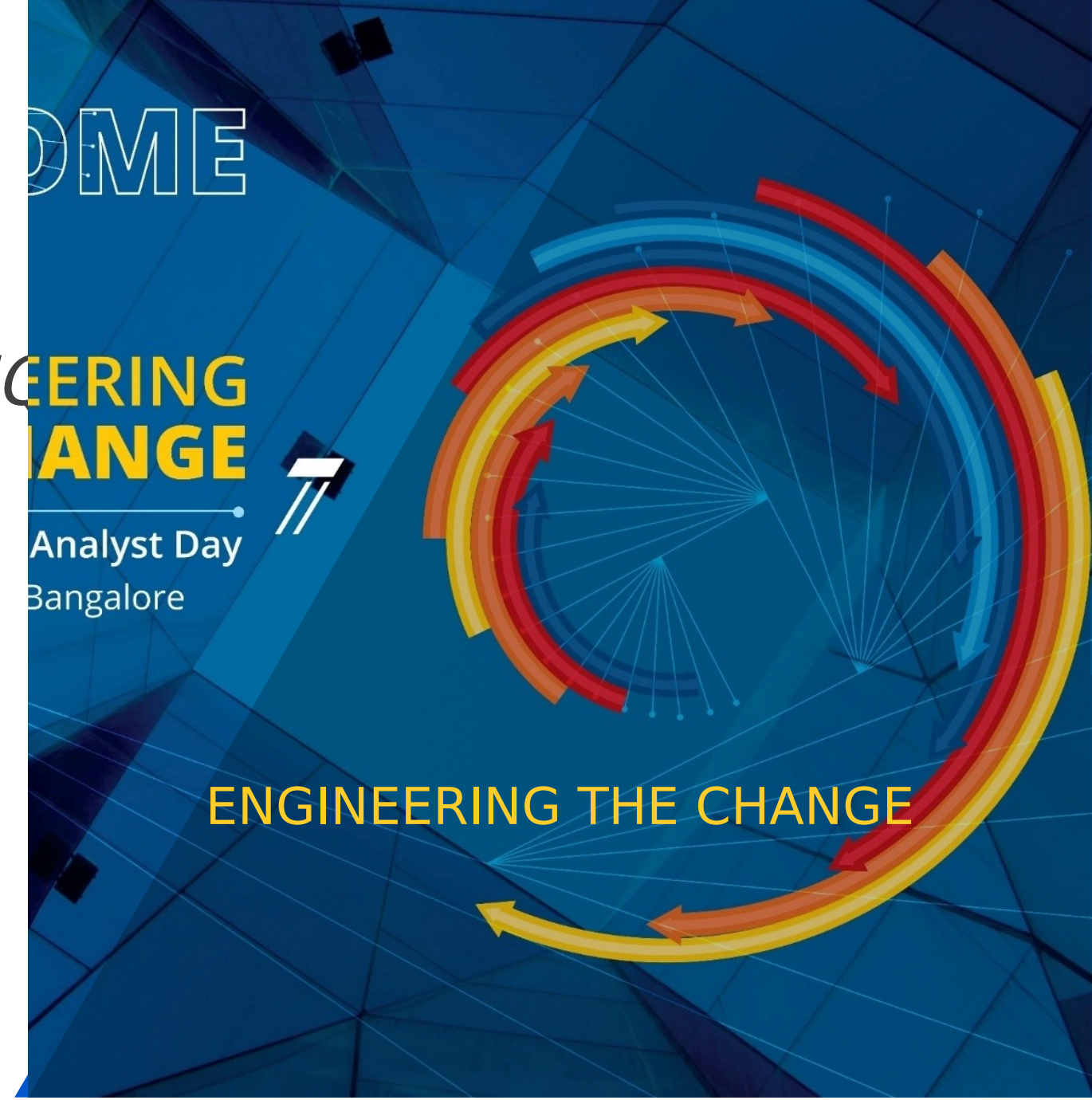
Sr. No.	Security test cases	Type- Negative/ Positive Test Case
1	Verify if a user cannot enter the characters more than the specified range in each field (Username and Password).	Negative
2	Verify if a user cannot enter the characters more than the specified range in each field (Username and Password).	Positive
3	Verify the login page by pressing ‘Back button’ of the browser. It should not allow you to enter into the system once you log out.	Negative
4	Verify the timeout functionality of the login session.	Positive
5	Verify if a user should not be allowed to log in with different credentials from the same browser at the same time.	Negative
6	Verify if a user should be able to login with the same credentials in different browsers at the same time.	Positive
7	Verify the Login page against SQL injection attack.	Negative
8	Verify the implementation of SSL certificate.	Positive

Test Case Writing (contd.)

	A	B	C	D	E	F	G	H	I	J	K
1	**Test Case ID**		BU_001	**Test Case Description**		Test the Login Functionality in Banking					
2	**Created By**		Mark	**Reviewed By**		Bill		**Version**		2.1	
3											
4	**QA Tester's Log**		Review comments from Bill incorporated in version 2.1								
5											
6	**Tester's Name**		Mark	**Date Tested**		1-Jan-2025		**Test Case (Pass/Fail/Not**		Pass	
7											
8	**S #**	**Prerequisites:**				**S #**	**Test Data**				
9	1	Access to Chrome Browser				1	Userid = mg12345				
10	2					2	Pass = df12@434c				
11	3					3					
12	4					4					
13											
14	**Test Scenario**	Verify on entering valid userid and password, the customer can login									
15											
16	**Step #**	**Step Details**		**Expected Results**		**Actual Results**			**Pass / Fail / Not executed / Suspended**		
17											
18	1	Navigate to http://demo.guru99.com		Site should open		As Expected			Pass		
19	2	Enter Userid & Password		Credential can be entered		As Expected			Pass		
20	3	Click Submit		Cutomer is logged in		As Expected			Pass		
21	4										
22											

TEST DESIGN TECHNIQUES

L&T Technology Services



Date

Identifying test conditions and designing test cases

The purpose of a test technique, including those discussed in this section, is to help in identifying test conditions, test cases, and test data.
The choice of which test techniques to use depends on a number of factors, including:

- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels and types
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- Software development lifecycle model
- The types of defects expected in the component or system

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels. When creating test cases, testers generally use a combination of test techniques to achieve the best results from the test effort.

The use of test techniques in the test analysis, test design, and test implementation activities can range from very informal (little to no documentation) to very formal. The appropriate level of formality depends on the context of testing, including the maturity of test and development processes, time constraints, safety or regulatory requirements, the knowledge and skills of the people involved, and the software development lifecycle model being followed.

Categories of test design techniques

The test techniques are classified as black-box, white-box, or experience-based.
• Black-box test techniques (also called behavioural or behaviour-based techniques) are based on an analysis of the appropriate test basis (e.g., formal requirements documents, specifications, use cases, user stories, or business processes). These techniques are applicable to both functional and non-functional testing. Black-box test techniques concentrate on the inputs and outputs of the test object without reference to its internal structure.
• White-box test techniques (also called structural or structure-based techniques) are based on an analysis of the architecture, detailed design, internal structure, or the code of the test object. Unlike black-box test techniques, white-box test techniques concentrate on the structure and processing within the test object.
• Experience-based test techniques leverage the experience of developers, testers and users to design, implement, and execute tests. These techniques are often combined with black-box and white-box test techniques.

Common characteristics of black-box test techniques include the following:
• Test conditions, test cases, and test data are derived from a test basis that may include software requirements, specifications, use cases, and user stories
• Test cases may be used to detect gaps between the requirements and the implementation of the requirements, as well as deviations from the requirements
• Coverage is measured based on the items tested in the test basis and the technique applied to the test basis

Common characteristics of white-box test techniques include::
• Test conditions, test cases, and test data are derived from a test basis that may include code, software architecture, detailed design, or any other source of information regarding the structure of the software
• Coverage is measured based on the items tested within a selected structure (e.g., the code or interfaces)

Categories of test design techniques (contd.)

Common characteristics of experience-based test techniques include::
Test conditions, test cases, and test data are derived from a test basis that may include knowledge and experience of testers, developers, users and other stakeholders

This knowledge and experience includes expected use of the software, its environment, likely defects, and the distribution of those defects

Specification-based or black-box techniques

Equivalence Partitioning

Equivalence partitioning divides data into partitions (also known as equivalence classes) in such a way that all the members of a given partition are expected to be processed in the same way. There are equivalence partitions for both valid and invalid values.

- Valid values are values that should be accepted by the component or system. An equivalence partition containing valid values is called a “valid equivalence partition.”
- Invalid values are values that should be rejected by the component or system. An equivalence partition containing invalid values is called an “invalid equivalence partition.”
- Partitions can be identified for any data element related to the test object, including inputs, outputs, internal values, time-related values (e.g., before or after an event) and for interface parameters (e.g., integrated components being tested during integration testing).
- Any partition may be divided into sub partitions if required.
- Each value must belong to one and only one equivalence partition.
- When invalid equivalence partitions are used in test cases, they should be tested individually, i.e., not combined with other invalid equivalence partitions, to ensure that failures are not masked. Failures can be masked when several failures occur at the same time but only one is visible, causing the other failures to be undetected.

To achieve 100% coverage with this technique, test cases must cover all identified partitions (including invalid partitions) by using a minimum of one value from each partition. Coverage is measured as the number of equivalence partitions tested by at least one value, divided by the total number of identified equivalence partitions, normally expressed as a percentage. Equivalence partitioning is applicable at all test levels.

Specification-based or black-box techniques (contd.)

Boundary Value Analysis

Boundary value analysis (BVA) is an extension of equivalence partitioning, but can only be used when the partition is ordered, consisting of numeric or sequential data. The minimum and maximum values (or first and last values) of a partition are its boundary values . For example, suppose an input field accepts a single integer value as an input, using a keypad to limit inputs so that non-integer inputs are impossible. The valid range is from 1 to 5, inclusive. So, there are three equivalence partitions: invalid (too low); valid; invalid (too high). For the valid equivalence partition, the boundary values are 1 and 5. For the invalid (too high) partition, the boundary value is 6. For the invalid (too low) partition, there is only one boundary value, 0, because this is a partition with only one member.

In the example above, we identify two boundary values per boundary. The boundary between invalid (too low) and valid gives the test values 0 and 1. The boundary between valid and invalid (too high) gives the test values 5 and 6. Some variations of this technique identify three boundary values per boundary: the values before, at, and just over the boundary. In the previous example, using three-point boundary values, the lower boundary test values are 0, 1, and 2, and the upper boundary test values are 4, 5, and 6.

Behaviour at the boundaries of equivalence partitions is more likely to be incorrect than behaviour within the partitions. It is important to remember that both specified and implemented boundaries may be displaced to positions above or below their intended positions, may be omitted altogether, or may be supplemented with unwanted additional boundaries. Boundary value analysis and testing will reveal almost all such defects by forcing the software to show behaviours from a partition other than the one to which the boundary value should belong.

Boundary value analysis can be applied at all test levels. This technique is generally used to test requirements that call for a range of numbers (including dates and times). Boundary coverage for a partition

Specification-based or black-box techniques (contd.)

Decision Table Testing

Decision tables are a good way to record complex business rules that a system must implement. When creating decision tables, the tester identifies conditions (often inputs) and the resulting actions (often outputs) of the system. These form the rows of the table, usually with the conditions at the top and the actions at the bottom. Each column corresponds to a decision rule that defines a unique combination of conditions which results in the execution of the actions associated with that rule. The values of the conditions and actions are usually shown as Boolean values (true or false) or discrete values (e.g., red, green, blue), but can also be numbers or ranges of numbers. These different types of conditions and actions might be found together in the same table.

The common notation in decision tables is as follows:
For conditions:
• Y means the condition is true (may also be shown as T or 1)
• N means the condition is false (may also be shown as F or 0)
• — means the value of the condition doesn't matter (may also be shown as N/A)
For actions:
• X means the action should occur (may also be shown as Y or T or 1)
• Blank means the action should not occur (may also be shown as – or N or F or 0)

A full decision table has enough columns (test cases) to cover every combination of conditions. By deleting columns that do not affect the outcome, the number of test cases can decrease considerably. For example by removing impossible combinations of conditions.

Specification-based or black-box techniques (contd.)

State Transition Testing

Components or systems may respond differently to an event depending on current conditions or previous history (e.g., the events that have occurred since the system was initialized). The previous history can be summarized using the concept of states. A state transition diagram shows the possible software states, as well as how the software enters, exits, and transitions between states. A transition is initiated by an event (e.g., user input of a value into a field). The event results in a transition. The same event can result in two or more different transitions from the same state. The state change may result in the software taking an action (e.g., outputting a calculation or error message).

A state transition table shows all valid transitions and potentially invalid transitions between states, as well as the events, and resulting actions for valid transitions. State transition diagrams normally show only the valid transitions and exclude the invalid transitions.

Tests can be designed to cover a typical sequence of states, to exercise all states, to exercise every transition, to exercise specific sequences of transitions, or to test invalid transitions.

State transition testing is used for menu-based applications and is widely used within the embedded software industry. The technique is also suitable for modelling a business scenario having specific states or for testing screen navigation. The concept of a state is abstract – it may represent a few lines of code or an entire business process.

Coverage is commonly measured as the number of identified states or transitions tested, divided by the total number of identified states or transitions in the test object, normally expressed as a percentage.

Specification-based or black-box techniques (contd.)

Use Case Testing

Tests can be derived from use cases, which are a specific way of designing interactions with software items. They incorporate requirements for the software functions. Use cases are associated with actors (human users, external hardware, or other components or systems) and subjects (the component or system to which the use case is applied).

Each use case specifies some behaviour that a subject can perform in collaboration with one or more actors. A use case can be described by interactions and activities, as well as preconditions, post-conditions and natural language where appropriate. Interactions between the actors and the subject may result in changes to the state of the subject. Interactions may be represented
graphically by work flows, activity diagrams, or business process models.

A use case can include possible variations of its basic behaviour, including exceptional behaviour and error handling (system response and recovery from programming, application and communication errors, e.g., resulting in an error message). Tests are designed to exercise the defined behaviours (basic, exceptional or alternative, and error handling). Coverage can be measured by the number of use case behaviours tested divided by the total number of use case behaviors, normally expressed as a percentage.

Structure-based or white-box techniques

White-box testing is based on the internal structure of the test object. White-box test techniques can be used at all test levels, but the two code-related techniques discussed in this section are most commonly used at the component test level.

Statement Testing and Coverage
Statement testing exercises the potential executable statements in the code. Coverage is measured as the number of statements executed by the tests divided by the total number of executable statements in the test object, normally expressed as a percentage.

Decision Testing and Coverage
Decision testing exercises the decisions in the code and tests the code that is executed based on the decision outcomes. To do this, the test cases follow the control flows that occur from a decision point (e.g., for an IF statement, one for the true outcome and one for the false outcome; for a CASE statement, test cases would be required for all the possible outcomes, including the default outcome).
Coverage is measured as the number of decision outcomes executed by the tests divided by the total number of decision outcomes in the test object, normally expressed as a percentage.

The Value of Statement and Decision Testing
When 100% statement coverage is achieved, it ensures that all executable statements in the code have been tested at least once, but it does not ensure that all decision logic has been tested. Of the two white-box techniques discussed in this syllabus, statement testing may provide less coverage than decision testing.
When 100% decision coverage is achieved, it executes all decision outcomes, which includes testing the true outcome and also the false outcome, even when there is no explicit false statement (e.g., in the case of an IF statement without an else in the code). Statement coverage helps to find defects in code that was not exercised by other tests. Decision coverage helps to find defects in code where other tests have not taken both true and false outcomes.

Experience-based techniques

When applying experience-based test techniques, the test cases are derived from the tester's skill and intuition, and their experience with similar applications and technologies. These techniques can be helpful in identifying tests that were not easily identified by other more systematic techniques. Depending on the tester's approach and experience, these techniques may achieve widely varying degrees of coverage and effectiveness. Coverage can be difficult to assess and may not be measurable with these techniques. Commonly used experience-based techniques are discussed in the following sections.

Error Guessing

Error guessing is a technique used to anticipate the occurrence of errors, defects, and failures, based on the tester's knowledge, including:

- How the application has worked in the past
- What kind of errors tend to be made
- Failures that have occurred in other applications

A methodical approach to the error guessing technique is to create a list of possible errors, defects, and failures, and design tests that will expose those failures and the defects that caused them. These error, defect, failure lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.

Experience-based techniques (contd.)

Exploratory Testing

In exploratory testing, informal (not pre-defined) tests are designed, executed, logged, and evaluated dynamically during test execution. The test results are used to learn more about the component or system, and to create tests for the areas that may need more testing.
Exploratory testing is sometimes conducted using session-based testing to structure the activity. In session-based testing, exploratory testing is conducted within a defined time-box, and the tester uses a test charter containing test objectives to guide the testing. The tester may use test session sheets to document the steps followed and the discoveries made.
Exploratory testing is most useful when there are few or inadequate specifications or significant time pressure on testing. Exploratory testing is also useful to complement other more formal testing techniques.
Exploratory testing is strongly associated with reactive test strategies. Exploratory testing can incorporate the use of other black-box, white-box, and experience-based techniques.

Checklist-based Testing

In checklist-based testing, testers design, implement, and execute tests to cover test conditions found in a checklist. As part of analysis, testers create a new checklist or expand an existing checklist, but testers may also use an existing checklist without modification. Such checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails.
Checklists can be created to support various test types, including functional and non-functional testing. In the absence of detailed test cases, checklist-based testing can provide guidelines and a degree of consistency. As these are high-level lists, some variability in the actual testing is likely to occur, resulting in potentially greater coverage but less repeatability.

Choosing a test technique

The choice of test technique to use depends upon: the type of the system and its criticality (formal or informal), test objectives, documentation available, knowledge of testers, regulatory and industry standards which have specific requirements to respect.

Additionally, customer and contractual requirements could also require specific test techniques to use according to their acceptance criteria. Risk levels (functional, non functional, structural or SLA) have also to be considered for choosing the right technique to use (systematic or non-systematic).

Test technique is also dependent on time and cost available for test preparation (choosing white box test technique or exploratory testing) and should be efficient with regards to development life.

Test techniques during integration testing may vary on systems or users interactions (BV, decision coverage, user cases). But generally, reviews are performed on all test levels, static analysis on code and component level, white box testing on component and system testing (testing data flows) and black-box technique is carried out in user and acceptance testing.

THANK YOU