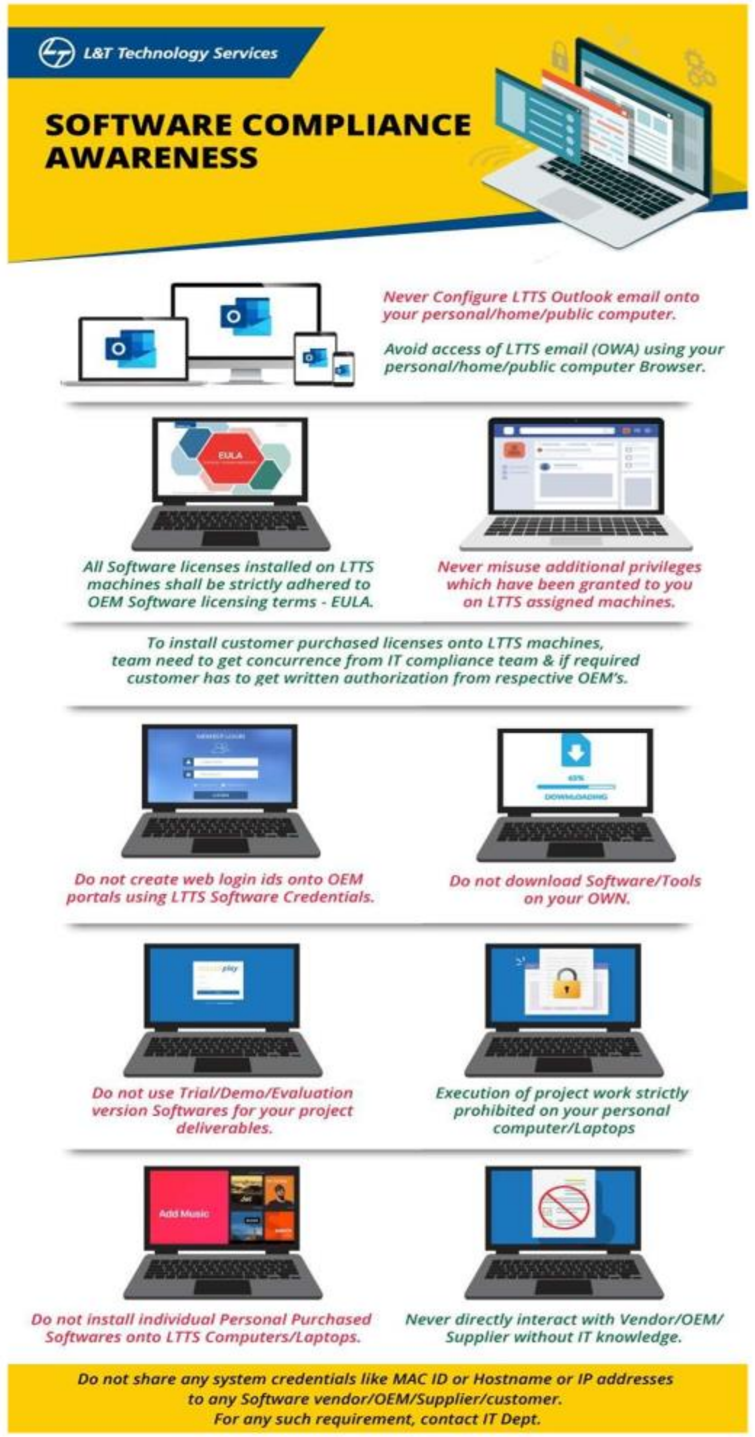
Manual Testing / TDLC: Fundamentals Of Testing

L&T Technology Services

Date



L&T Technology Services

SOFTWARE COMPLIANCE AWARENESS

Never Configure LTTS Outlook email onto your personal/home/public computer.

Avoid access of LTTS email (OWA) using your personal/home/public computer Browser.

All Software licenses installed on LTTS machines shall be strictly adhered to OEM Software licensing terms - EULA.

Never misuse additional privileges which have been granted to you on LTTS assigned machines.

To install customer purchased licenses onto LTTS machines, team need to get concurrence from IT compliance team & if required customer has to get written authorization from respective OEM's.

Do not create web login ids onto OEM portals using LTTS Software Credentials.

Do not download Software/Tools on your OWN.

Do not use Trial/Demo/Evaluation version Softwares for your project deliverables.

Execution of project work strictly prohibited on your personal computer/Laptops

Do not install individual Personal Purchased Softwares onto LTTS Computers/Laptops.

Never directly interact with Vendor/OEM/ Supplier without IT knowledge.

Do not share any system credentials like MAC ID or Hostname or IP addresses to any Software vendor/OEM/Supplier/customer. For any such requirement, contact IT Dept.

Contents

What is testing?

Software systems are an integral part of life, from business applications (e.g., banking) to consumer products (e.g. cars).

Software that does not work correctly can lead to many problems, including loss of money, time, or business reputation, and even injury or death.

Software testing is a way to assess the quality of the software and to reduce the risk of software failure in operation.

A common misperception of testing is that it only consists of running tests, i.e., executing the software and checking the results.

The test process also includes activities such as test planning, analyzing, designing, and implementing tests, reporting test progress and results, and evaluating the quality of a test object.

What is testing? - Static & Dynamic Testing

Some testing does involve the execution of the component or system being tested; such testing is called dynamic testing.

Other testing does not involve the execution of the component or system being tested; such testing is called static testing.

Testing also includes reviewing work products such as requirements, user stories, and source code.

What is testing? - Validation

Another common misperception of testing is that it focuses entirely on verification of requirements, user stories, or other specifications.

While testing does involve checking whether the system meets specified requirements, it also involves validation, which is checking whether the system will meet user and other stakeholder needs in its operational environment(s).

What is testing? – Objectives of Testing

- To prevent defects by evaluate work products such as requirements, user stories, design, and code
- To verify whether all specified requirements have been fulfilled
- To check whether the test object is complete and validate if it works as the users and other stakeholders expect
- To build confidence in the level of quality of the test object
- To find defects and failures thus reduce the level of risk of inadequate software quality
- To provide sufficient information to stakeholders to allow them to make informed decisions, especially regarding the level of quality of the test object
- To comply with contractual, legal, or regulatory requirements or standards, and/or to verify the test object’s compliance with such requirements or standards

What is testing? – Different Objectives in Testing Stages

During component testing, one objective may be to find as many failures as possible so that the underlying defects are identified and fixed early. Another objective may be to increase code coverage of the component tests.

During acceptance testing, one objective may be to confirm that the system works as expected and satisfies requirements.

Another objective of this testing may be to give information to stakeholders about the risk of releasing the system at a given time.

What is testing? – Difference between testing and debugging

Executing tests can show failures that are caused by defects in the software.

Debugging is the development activity that finds, analyzes, and fixes such defects.

Subsequent confirmation testing checks whether the fixes resolved the defects.

In some cases, testers are responsible for the initial test and the final confirmation test, while developers do the debugging, associated component and component integration testing (continues integration).

However, in Agile development and in some other software development lifecycles, testers may be involved in debugging and component testing.

Why is testing necessary?

Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation.

When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems.

Software testing may also be required to meet contractual or legal requirements or industry-specific standards.

Why is testing necessary? - Testing's Contributions to Success

Throughout the history of computing, it is quite common for software and systems to be delivered into operation and, due to the presence of defects, to subsequently cause failures or otherwise not meet the stakeholders' needs. However, using appropriate test techniques can reduce the frequency of such problematic deliveries, when those techniques are applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the software development lifecycle.

Having testers involved in requirements reviews or user story refinement could detect defects in these work products. The identification and removal of requirements defects reduces the risk of incorrect or un-testable features being developed.

Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. This increased understanding can reduce the risk of fundamental design defects and enable tests to be identified at an early stage.

Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. This increased understanding can reduce the risk of defects within the code and the tests.

Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed, and support the process of removing the defects that caused the failures (i.e., debugging). This increases the likelihood that the software meets stakeholder needs and satisfies requirements.

Why is testing necessary? - Quality Assurance and Testing

While people often use the phrase quality assurance (or just QA) to refer to testing, quality assurance and testing are not the same, but they are related. A larger concept, quality management, ties them together.

Quality management includes all activities that direct and control an organization with regard to quality. Among other activities, quality management includes both quality assurance and quality control.

Quality assurance is typically focused on adherence to proper processes, in order to provide confidence that the appropriate levels of quality will be achieved. When processes are carried out properly, the work products created by those processes are generally of higher quality, which contributes to defect prevention. In addition, the use of root cause analysis to detect and remove the causes of defects, along with the proper application of the findings of retrospective meetings to improve processes, are important for effective quality assurance.

Quality control involves various activities, including test activities, that support the achievement of appropriate levels of quality. Test activities are part of the overall software development or maintenance process. Since quality assurance is concerned with the proper execution of the entire process, quality assurance supports proper testing.

Why is testing necessary? - Errors, Defects, and Failures

A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product.

An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product. For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.

If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances. For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

Errors may occur for many reasons, such as:
- Time pressure
- Human fallibility
- Inexperienced or insufficiently skilled project participants
- Miscommunication between project participants, including miscommunication about requirements and design
- Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used
- Misunderstandings about intra-system and inter-system interfaces, especially when such intrasystem and inter-system interactions are large in number
- New, unfamiliar technologies

Why is testing necessary? - Defects, Root Causes and Effects

The root causes of defects are the earliest actions or conditions that contributed to creating the defects. Defects can be analyzed to identify their root causes, so as to reduce the occurrence of similar defects in the future. By focusing on the most significant root causes, root cause analysis can lead to process improvements that prevent a significant number of future defects from being introduced.

For example, suppose incorrect interest payments, due to a single line of incorrect code, result in customer complaints. The defective code was written for a user story which was ambiguous, due to the product owner's misunderstanding of how to calculate interest. If a large percentage of defects exist in interest calculations, and these defects have their root cause in similar misunderstandings, the product owners could be trained in the topic of interest calculations to reduce such defects in the future.

In this example,

- The customer complaints are effects.
- The incorrect interest payments are failures. The improper calculation in the code is a defect, and it resulted from the original defect, the ambiguity in the user story.
- The root cause of the original defect was a lack of knowledge on the part of the product owner, which resulted in the product owner making an error while writing the user story.

Testing principles

Testing shows the presence of defects, not their absence

- Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, testing is not a proof of correctness.

Exhaustive testing is impossible

- Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Rather than attempting to test exhaustively, risk analysis, test techniques, and priorities should be used to focus test efforts.

Early testing saves time and money

- To find defects early, both static and dynamic test activities should be started as early as possible in the software development lifecycle. Early testing is sometimes referred to as shift left. Testing early in the software development lifecycle helps reduce or eliminate costly changes.

Defects cluster together

- A small number of modules usually contains most of the defects discovered during pre-release testing, or is responsible for most of the operational failures. Predicted defect clusters, and the actual observed defect clusters in test or operation, are an important input into a risk analysis used to focus the test effort.

Beware of the pesticide paradox

- If the same tests are repeated over and over again, eventually these tests no longer find any new defects. To detect new defects, existing tests and test data may need changing, and new tests may need to be written (Tests are no longer effective at finding defects, just as pesticides are no longer effective at killing insects after a while). In some cases, such as automated regression testing, the pesticide paradox has a beneficial outcome, which is the relatively low number of regression defects.

Testing is context dependent

- Testing is done differently in different contexts. For example, safety-critical industrial control software is tested differently from an e-commerce mobile app. As another example, testing in an Agile project is done differently than testing in a sequential software development lifecycle project.

Absence-of-errors is a fallacy

- Some organizations expect that testers can run all possible tests and find all possible defects, but principles 2 and 1, respectively, tell us that this is impossible. Further, it is a fallacy (i.e., a mistaken belief) to expect that just finding and fixing a large number of defects will ensure the success of a system. For example, thoroughly testing all specified requirements and fixing all defects found could still produce a system that is difficult to use, that does not fulfill the users’ needs and expectations, or that is inferior compared to other competing systems.

Fundamental Test Process

There is no one universal software test process, but there are common sets of test activities without which testing will be less likely to achieve its established objectives. These sets of test activities are a test process. The proper, specific software test process in any given situation depends on many factors. Which test activities are involved in this test process, how these activities are implemented, and when these activities occur may be discussed in an organization's test strategy.

Contextual factors that influence the test process for an organization, include, but are not limited to:

- Software development lifecycle model and project methodologies being used
- Test levels and test types being considered
- Product and project risks
- Business domain
- Operational constraints, including but not limited to:
    - ✓ Budgets and resources
    - ✓ Timescales
    - ✓ Complexity
    - ✓ Contractual and regulatory requirements
- Organizational policies and practices
- Required internal and external standards

Fundamental Test Process - Test Activities and Tasks

Test planning

- Test planning involves activities that define the objectives of testing and the approach for meeting test objectives within constraints imposed by the context (e.g., specifying suitable test techniques and tasks, and formulating a test schedule for meeting a deadline). Test plans may be revisited based on feedback from monitoring and control activities.

Test monitoring and control

- Test monitoring involves the on-going comparison of actual progress against planned progress using any test monitoring metrics defined in the test plan. Test control involves taking actions necessary to meet the objectives of the test plan (which may be updated over time). Test monitoring and control are supported by the evaluation of exit criteria, which are referred to as the definition of done in some software development lifecycle models (see ISTQB-CTFL-AT). For example, the evaluation of exit criteria for test execution as part of a given test level may include:
  - Checking test results and logs against specified coverage criteria
  - Assessing the level of component or system quality based on test results and logs
  - Determining if more tests are needed (e.g., if tests originally intended to achieve a certain level of product risk coverage failed to do so, requiring additional tests to be written and executed)
  - Test progress against the plan is communicated to stakeholders in test progress reports, including
  - deviations from the plan and information to support any decision to stop testing.

Test analysis

- During test analysis, the test basis is analyzed to identify testable features and define associated test conditions. In other words, test analysis determines “what to test” in terms of measurable coverage criteria. Test analysis includes the following major activities:
  - Analyzing the test basis appropriate to the test level being considered
  - Evaluating the test basis and test items to identify defects of various types
  - Identifying features and sets of features to be tested
  - Defining and prioritizing test conditions for each feature based on analysis of the test basis, and considering functional, non-functional, and structural characteristics, other business and technical factors, and levels of risks
  - Capturing bi-directional traceability between each element of the test basis and the associated test conditions

Fundamental Test Process - Test Activities and Tasks (continued)

Test design

- During test design, the test conditions are elaborated into high-level test cases, sets of high-level test cases, and other testware. So, test analysis answers the question "what to test?" while test design answers the question "how to test?". Test design includes the following major activities:
  - Designing and prioritizing test cases and sets of test cases
  - Identifying necessary test data to support test conditions and test cases
  - Designing the test environment and identifying any required infrastructure and tools
  - Capturing bi-directional traceability between the test basis, test conditions, and test cases

Test implementation

- During test implementation, the testware necessary for test execution is created and/or completed, including sequencing the test cases into test procedures. So, test design answers the question "how to test?" while test implementation answers the question "do we now have everything in place to run the tests?". Test implementation includes the following major activities:
  - Developing and prioritizing test procedures, and, potentially, creating automated test scripts
  - Creating test suites from the test procedures and (if any) automated test scripts
  - Arranging the test suites within a test execution schedule in a way that results in efficient test execution
  - Building the test environment (including, potentially, test harnesses, service virtualization, simulators, and other infrastructure items) and verifying that everything needed has been set up correctly
  - Preparing test data and ensuring it is properly loaded in the test environment
  - Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test suites

Test execution

- During test execution, test suites are run in accordance with the test execution schedule. Test execution includes the following major activities:
  - Recording the IDs and versions of the test item(s) or test object, test tool(s), and testware
  - Executing tests either manually or by using test execution tools
  - Comparing actual results with expected results
  - Analyzing anomalies to establish their likely causes (e.g., failures may occur due to defects in the code, but false positives also may occur
  - Reporting defects based on the failures observed (see section 5.6)
  - Logging the outcome of test execution (e.g., pass, fail, blocked)
  - Repeating test activities either as a result of action taken for an anomaly, or as part of the planned testing (e.g., execution of a corrected test, confirmation testing, and/or regression testing)
  - Verifying and updating bi-directional traceability between the test basis, test conditions, test cases, test procedures, and test results.

Fundamental Test Process - Test Activities and Tasks (continued)

Test completion

- Test completion activities collect data from completed test activities to consolidate experience, testware, and any other relevant information. Test completion activities occur at project milestones such as when a software system is released, a test project is completed (or cancelled), an Agile project iteration is finished, a test level is completed, or a maintenance release has been completed. Test completion includes the following major activities:
- Checking whether all defect reports are closed, entering change requests or product backlog items for any defects that remain unresolved at the end of test execution
- Creating a test summary report to be communicated to stakeholders
- Finalizing and archiving the test environment, the test data, the test infrastructure, and other testware for later reuse
- Handing over the testware to the maintenance teams, other project teams, and/or other stakeholders who could benefit from its use
- Analyzing lessons learned from the completed test activities to determine changes needed for future iterations, releases, and projects
- Using the information gathered to improve test process maturity

The Psychology of Testing

Human Psychology and Testing

Identifying defects during a static test such as a requirement review or user story refinement session, or identifying failures during dynamic test execution, may be perceived as criticism of the product and of its author. An element of human psychology called confirmation bias can make it difficult to accept information that disagrees with currently held beliefs.
For example, since developers expect their code to be correct, they have a confirmation bias that makes it difficult to accept that the code is incorrect.

In addition to confirmation bias, other cognitive biases may make it difficult for people to understand or accept information produced by testing. Further, it is a common human trait to blame the bearer of bad news, and information produced by testing often contains bad news. As a result of these psychological factors, some people may perceive testing as a destructive activity, even though it contributes greatly to project progress and product quality .To try to reduce these perceptions, information about defects and failures should be communicated in a constructive way. This way, tensions between the testers and the analysts, product owners, designers, and developers can be reduced. This applies during both static and dynamic testing.

Testers and test managers need to have good interpersonal skills to be able to communicate effectively about defects, failures, test results, test progress, and risks, and to build positive relationships with colleagues. Ways to communicate well include the following examples:

- Start with collaboration rather than battles. Remind everyone of the common goal of better quality systems.
- Emphasize the benefits of testing. For example, for the authors, defect information can help them improve their work products and their skills. For the organization, defects found and fixed during testing will save time and money and reduce overall risk to product quality.
- Communicate test results and other findings in a neutral, fact-focused way without criticizing the person who created the defective item. Write objective and factual defect reports and review findings.
- Try to understand how the other person feels and the reasons they may react negatively to the information.
- Confirm that the other person has understood what has been said and vice versa.

The Psychology of Testing (continued)

Tester’s and Developer’s Mindsets

• Developers and testers often think differently. The primary objective of development is to design and build a product.

• As discussed earlier, the objectives of testing include verifying and validating the product, finding defects prior to release, and so forth. These are different sets of objectives which require different mindsets. Bringing these mindsets together helps to achieve a higher level of product quality.

• A mindset reflects an individual’s assumptions and preferred methods for decision making and problem-solving.

• A tester’s mindset should include curiosity, professional pessimism, a critical eye, attention to detail, and a motivation for good and positive communications and relationships. A tester’s mindset tends to grow and mature as the tester gains experience.

• A developer’s mindset may include some of the elements of a tester’s mindset, but successful developers are often more interested in designing and building solutions than in contemplating what might be wrong with those solutions. In addition, confirmation bias makes it difficult to become aware of errors committed by themselves.

• With the right mindset, developers are able to test their own code. Different software development lifecycle models often have different ways of organizing the testers and test activities. Having some of the test activities done by independent testers increases defect detection effectiveness, which is particularly important for large, complex, or safety-critical systems.

• Independent testers bring a perspective which is different than that of the work product authors (i.e., business analysts, product owners, designers, and developers), since they have different cognitive biases from the authors.

Software development models

A software development lifecycle model describes the types of activity performed at each stage in a software development project, and how the activities relate to one another logically and chronologically. There are a number of different software development lifecycle models, each of which requires different approaches to testing.

Software Development and Software Testing

It is an important part of a tester's role to be familiar with the common software development lifecycle models so that appropriate test activities can take place. In any software development lifecycle model, there are several characteristics of good testing:

- For every development activity, there is a corresponding test activity
- Each test level has test objectives specific to that level
- Test analysis and design for a given test level begin during the corresponding development activity
- Testers participate in discussions to define and refine requirements and design, and are involved in reviewing work products (e.g., requirements, design, user stories, etc.) as soon as drafts are available.

No matter which software development lifecycle model is chosen, test activities should start in the early stages of the lifecycle, adhering to the testing principle of early testing. The categories of common software development lifecycle models as follows:

Sequential development models - A sequential development model describes the software development process as a linear, sequential flow of activities. This means that any phase in the development process should begin when the previous phase is complete. In theory, there is no overlap of phases, but in practice, it is beneficial to have early feedback from the following phase.

Software development models – Waterfall, V model, Agile & DevOps

Waterfall Model - In the Waterfall model, the development activities (e.g., requirements analysis, design, coding, testing) are completed one after another. In this model, test activities only occur after all other development activities have been completed.

V-model - It integrates the test process throughout the development process, implementing the principle of early testing. Further, the V-model includes test levels associated with each corresponding development phase, which further supports early testing. In this model, the execution of tests associated with each test level proceeds sequentially, but in some cases overlapping occurs.

Sequential development models deliver software that contains the complete set of features, but typically require months or years for delivery to stakeholders and users.

Iterative and incremental development models - Incremental development involves establishing requirements, designing, building, and testing a system in pieces, which means that the software's features grow incrementally. The size of these feature increments varies, with some methods having larger pieces and some smaller pieces. The feature increments can be as small as a single change to a user interface screen or new query option.
Iterative development occurs when groups of features are specified, designed, built, and tested together in a series of cycles, often of a fixed duration. Iterations may involve changes to features developed in earlier iterations, along with changes in project scope. Each iteration delivers working software which is a growing subset of the overall set of features until the final software is delivered or development is stopped.
Examples include:
• **Agile/Scrum**: Each iteration tends to be relatively short (e.g., hours, days, or a few weeks), and the feature increments are correspondingly small, such as a few enhancements and/or two or three new features.

Software development models – Waterfall, V model, Agile & DevOps (contd.)

DevOps/CI/CD – Some teams use continuous delivery or continuous deployment, both of which involve significant automation of multiple test levels as part of their delivery pipelines. Many development efforts using these methods also include the concept of self-organizing teams, which can change the way testing work is organized as well as the relationship between testers and developers.

These methods form a growing system, which may be released to end-users on a feature-by-feature basis, on an iteration-by-iteration basis, or in a more traditional major-release fashion. Regardless of whether the software increments are released to end-users, regression testing is increasingly important as the system grows.
In contrast to sequential models, iterative and incremental models may deliver usable software in weeks or even days, but may only deliver the complete set of requirements product over a period of months or even years.

Reasons why software development models must be adapted to the context of project and product characteristics can be:

- Difference in product risks of systems (complex or simple project)
- Many business units can be part of a project or program (combination of sequential and agile development)
- Short time to deliver a product to the market (merge of test levels and/or integration of test types in test levels)

Test levels

Test levels are groups of test activities that are organized and managed together. Each test level is an instance of the test process, consisting of the activities described in section 1.4, performed in relation to software at a given level of development, from individual units or components to complete systems or, where applicable, systems of systems. Test levels are related to other activities within the software development lifecycle.

- Component testing
- Integration testing
- System testing
- Acceptance testing

Test levels are characterized by the following attributes:

- Specific objectives
- Test basis, referenced to derive test cases
- Test object (i.e., what is being tested)
- Typical defects and failures
- Specific approaches and responsibilities

For every test level, a suitable test environment is required. In acceptance testing, for example, a production-like test environment is ideal, while in component testing the developers typically use their own development environment.

Test levels - Component Testing

Objectives of component testing

Component testing (also known as unit or module testing) focuses on components that are separately testable. Objectives of component testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviours of the component are as designed and specified
- Building confidence in the component's quality
- Finding defects in the component
- Preventing defects from escaping to higher test levels

Test basis

Examples of work products that can be used as a test basis for component testing include:

- Detailed design
- Code
- Data model
- Component specifications

Test objects

Typical test objects for component testing include:

- Components, units or modules
- Code and data structures
- Classes
- Database modules

Typical defects and failures

Examples of typical defects and failures for component testing include:

- Incorrect functionality (e.g., not as described in design specifications)
- Data flow problems
- Incorrect code and logic

Test levels - Component Testing (contd.)

Specific approaches and responsibilities
Component testing is usually performed by the developer who wrote the code, but it at least requires access to the code being tested. Developers may alternate component development with finding and fixing defects. Developers will often write and execute tests after having written the code for a component. However, in Agile development especially, writing automated component test cases may precede writing application code.

Test levels - Integration Testing

Objectives of integration testing

Integration testing focuses on interactions between components or systems. Objectives of integration testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviours of the interfaces are as designed and specified
- Building confidence in the quality of the interfaces
- Finding defects (which may be in the interfaces themselves or within the components or systems)
- Preventing defects from escaping to higher test levels

Test basis

Examples of work products that can be used as a test basis for integration testing include:

- Software and system design
- Sequence diagrams
- Interface and communication protocol specifications
- Use cases
- Architecture at component or system level
- Workflows
- External interface definitions

Test objects

Typical test objects for integration testing include:

- Subsystems
- Databases
- Infrastructure
- Interfaces
- APIs
- Microservices

Test levels - Integration Testing (contd.)

Typical defects and failures

Examples of typical defects and failures for component integration testing include:

- Incorrect data, missing data, or incorrect data encoding
- Incorrect sequencing or timing of interface calls
- Interface mismatch
- Failures in communication between components
- Unhandled or improperly handled communication failures between components
- Incorrect assumptions about the meaning, units, or boundaries of the data being passed between components

Examples of typical defects and failures for system integration testing include:

- Inconsistent message structures between systems
- Incorrect data, missing data, or incorrect data encoding
- Interface mismatch
- Failures in communication between systems
- Unhandled or improperly handled communication failures between systems

Specific approaches and responsibilities

Component integration tests and system integration tests should concentrate on the integration itself. For example, if integrating module A with module B, tests should focus on the communication between the modules, not the functionality of the individual modules, as that should have been covered during component testing. If integrating system X with system Y, tests should focus on the communication between the systems, not the functionality of the individual systems, as that should have been covered during system testing. Functional, non-functional, and structural test types are applicable.

Component integration testing is often the responsibility of developers. System integration testing is generally the responsibility of testers. Ideally, testers performing system integration testing should understand the system architecture, and should have influenced integration planning.

Test levels - System Testing

Objectives of system testing

System testing focuses on the behaviour and capabilities of a whole system or product, often considering the end-to-end tasks the system can perform and the non-functional behaviours it exhibits while performing those tasks. Objectives of system testing include:

- Reducing risk
- Verifying whether the functional and non-functional behaviours of the system are as designed and specified
- Validating that the system is complete and will work as expected
- Building confidence in the quality of the system as a whole
- Finding defects
- Preventing defects from escaping to higher test levels or production

Test basis

Examples of work products that can be used as a test basis for system testing include:

System and software requirement specifications (functional and non-functional)

- Risk analysis reports
- Use cases
- Epics and user stories
- Models of system behavior
- State diagrams
- System and user manuals

Test objects

Typical test objects for system testing include:

- Applications
- Hardware/software systems
- Operating systems
- System under test (SUT)
- System configuration and configuration data

Test levels – System Testing (contd.)

Typical defects and failures

Examples of typical defects and failures for system testing include:

- Incorrect calculations
- Incorrect or unexpected system functional or non-functional behaviour
- Incorrect control and/or data flows within the system
- Failure to properly and completely carry out end-to-end functional tasks
- Failure of the system to work properly in the system environment(s)
- Failure of the system to work as described in system and user manuals

Specific approaches and responsibilities

System testing should focus on the overall, end-to-end behaviour of the system as a whole, both functional and non-functional. System testing should use the most appropriate techniques (see chapter 4) for the aspect(s) of the system to be tested. For example, a decision table may be created to verify whether functional behaviour is as described in business rules.

System testing is typically carried out by independent testers who rely heavily on specifications. Defects in specifications (e.g., missing user stories, incorrectly stated business requirements, etc.) can lead to a lack of understanding of, or disagreements about, expected system behaviour. Such situations can cause false positives and false negatives, which waste time and reduce defect detection effectiveness, respectively. Early involvement of testers in user story refinement or static testing activities, such as reviews, helps to reduce the incidence of such situations.

Test levels - Acceptance Testing

Objectives of system testing

Acceptance testing, like system testing, typically focuses on the behaviour and capabilities of a whole system or product. Objectives of acceptance testing include:

- Establishing confidence in the quality of the system as a whole
- Validating that the system is complete and will work as expected
- Verifying that functional and non-functional behaviours of the system are as specified

Acceptance testing may produce information to assess the system's readiness for deployment and use by the customer (end-user). Defects may be found during acceptance testing, but finding defects is often not an objective, and finding a significant number of defects during acceptance testing may in some cases be considered a major project risk. Common forms of acceptance testing include the following:

- User acceptance testing
- Operational acceptance testing
- Contractual and regulatory acceptance testing
- Alpha and beta testing.

Test basis

Examples of work products that can be used as a test basis for any form of acceptance testing include:

- Business processes
- User or business requirements
- Regulations, legal contracts and standards
- Use cases and/or user stories
- System requirements
- System or user documentation
- Installation procedures
- Risk analysis reports

Test levels – Acceptance Testing (contd.)

Typical test objects

Typical test objects for any form of acceptance testing include:

- System under test
- System configuration and configuration data
- Business processes for a fully integrated system
- Recovery systems and hot sites (for business continuity and disaster recovery testing)
- Operational and maintenance processes
- Forms
- Reports
- Existing and converted production data

Typical defects and failures

Examples of typical defects for any form of acceptance testing include:

- System workflows do not meet business or user requirements
- Business rules are not implemented correctly
- System does not satisfy contractual or regulatory requirements
- Non-functional failures such as security vulnerabilities, inadequate performance efficiency under high loads, or improper operation on a supported platform

Specific approaches and responsibilities

Acceptance testing is often the responsibility of the customers, business users, product owners, or operators of a system, and other stakeholders may be involved as well.

Test Types

A test type is a group of test activities aimed at testing specific characteristics of a software system, or a
part of a system, based on specific test objectives. Such objectives may include:

• Evaluating functional quality characteristics, such as completeness, correctness, and appropriateness

• Evaluating non-functional quality characteristics, such as reliability, performance efficiency, security, compatibility, and usability

• Evaluating whether the structure or architecture of the component or system is correct, complete, and as specified

• Evaluating the effects of changes, such as confirming that defects have been fixed (confirmation testing) and looking for unintended changes in behaviour resulting from software or environment changes (regression testing)

Functional Testing

Functional testing of a system involves tests that evaluate functions that the system should perform. Functional requirements may be described in work products such as business requirements specifications, epics, user stories, use cases, or functional specifications, or they may be undocumented.
The functions are "what" the system should do.

Functional tests should be performed at all test levels (e.g., tests for components may be based on a component specification), though the focus is different at each level

.

Functional testing considers the behaviour of the software, so black-box techniques may be used to derive test conditions and test cases for the functionality of the component or system.

The thoroughness of functional testing can be measured through functional coverage. Functional coverage is the extent to which some functionality has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and functional requirements, the percentage of these requirements which are addressed by testing can be calculated, potentially identifying coverage gaps.

Test Types (contd.)

Non-functional Testing
Non-functional testing of a system evaluates characteristics of systems and software such as usability, performance efficiency or security. Non-functional testing is the testing of “how well” the system behaves. Contrary to common misperceptions, non-functional testing can and often should be performed at all test levels, and done as early as possible. The late discovery of non-functional defects can be extremely dangerous to the success of a project.

Black-box techniques may be used to derive test conditions and test cases for non-functional testing. For example, boundary value analysis can be used to define the stress conditions for performance tests.

The thoroughness of non-functional testing can be measured through non-functional coverage. Non-functional coverage is the extent to which some type of non-functional element has been exercised by tests, and is expressed as a percentage of the type(s) of element being covered. For example, using traceability between tests and supported devices for a mobile application, the percentage of devices which are addressed by compatibility testing can be calculated, potentially identifying coverage gaps.

Non-functional test design and execution may involve special skills or knowledge, such as knowledge of the inherent weaknesses of a design or technology (e.g., security vulnerabilities associated with particular programming languages) or the particular user base (e.g., the personas of users of healthcare facility management systems).

White-box Testing
White-box testing derives tests based on the system’s internal structure or implementation. Internal structure may include code, architecture, work flows, and/or data flows within the system. The thoroughness of white-box testing can be measured through structural coverage. Structural coverage is the extent to which some type of structural element has been exercised by tests, and is expressed as a percentage of the type of element being covered.
White-box test design and execution may involve special skills or knowledge, such as the way the code is built, how data is stored (e.g., to evaluate possible database queries), and how to use coverage tools and to correctly interpret their results.

Test Types (contd.)

Change-related Testing

When changes are made to a system, either to correct a defect or because of new or changing functionality, testing should be done to confirm that the changes have corrected the defect or implemented the functionality correctly, and have not caused any unforeseen adverse consequences.

• Confirmation testing: After a defect is fixed, the software may be tested with all test cases that failed due to the defect, which should be re-executed on the new software version. The software may also be tested with new tests to cover changes needed to fix the defect. At the very least, the steps to reproduce the failure(s) caused by the defect must be re-executed on the new software version. The purpose of a confirmation test is to confirm whether the original defect has been successfully fixed.

• Regression testing: It is possible that a change made in one part of the code, whether a fix or another type of change, may accidentally affect the behaviour of other parts of the code, whether within the same component, in other components of the same system, or even in other systems. Changes may include changes to the environment, such as a new version of an operating system or database management system. Such unintended side-effects are called regressions. Regression testing involves running tests to detect such unintended side-effects.

Confirmation testing and regression testing are performed at all test levels.

Maintenance testing

Once deployed to production environments, software and systems need to be maintained. Changes of various sorts are almost inevitable in delivered software and systems, either to fix defects discovered in operational use, to add new functionality, or to delete or alter already-delivered functionality. Maintenance is also needed to preserve or improve non-functional quality characteristics of the component or system over its lifetime, especially performance efficiency, compatibility, reliability, security, , and portability.

When any changes are made as part of maintenance, maintenance testing should be performed, both to evaluate the success with which the changes were made and to check for possible side-effects (e.g., regressions) in parts of the system that remain unchanged (which is usually most of the system). Maintenance can involve planned releases and unplanned releases (hot fixes).

A maintenance release may require maintenance testing at multiple test levels, using various test types, based on its scope. The scope of maintenance testing depends on:

- The degree of risk of the change, for example, the degree to which the changed area of software communicates with other components or systems
- The size of the existing system
- The size of the change

Triggers for Maintenance

There are several reasons why software maintenance, and thus maintenance testing, takes place, both for planned and unplanned changes. We can classify the triggers for maintenance as follows:

- Modification, such as planned enhancements (e.g., release-based), corrective and emergency changes, changes of the operational environment (such as planned operating system or database upgrades), upgrades of COTS software, and patches for defects and vulnerabilities
- Migration, such as from one platform to another, which can require operational tests of the new environment as well as of the changed software, or tests of data conversion when data from another application will be migrated into the system being maintained
  - ✓ Retirement, such as when an application reaches the end of its life. When an application or system is retired, this can require testing of data migration or archiving if long data-retention periods are required.
  - ✓ Testing restore/retrieve procedures after archiving for long retention periods may also be needed.
  - ✓ Regression testing may be needed to ensure that any functionality that remains in service still works.

Maintenance testing (contd.)

Impact Analysis for Maintenance

Impact analysis evaluates the changes that were made for a maintenance release to identify the intended consequences as well as expected and possible side effects of a change, and to identify the areas in the system that will be affected by the change. Impact analysis can also help to identify the impact of a change on existing tests. The side effects and affected areas in the system need to be tested for regressions, possibly after updating any existing tests affected by the change. Impact analysis may be done before a change is made, to help decide if the change should be made, based on the potential consequences in other areas of the system.

Impact analysis can be difficult if:

- Specifications (e.g., business requirements, user stories, architecture) are out of date or missing
- Test cases are not documented or are out of date
- Bi-directional traceability between tests and the test basis has not been maintained
- Tool support is weak or non-existent
- The people involved do not have domain and/or system knowledge
- Insufficient attention has been paid to the software's maintainability during development

Manual Testing / TDLC

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 for spelling, grammar, and readability).

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)

Moreover, most types of maintainability defects can only be found by static testing (e.g., improper modularization, poor reusability of components, code that is difficult to analyze and modify without introducing new defects).

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.)

<u>Applying Review Techniques</u>

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, performance tester, etc.). The same principles apply as in perspective-based reading because the roles are similar.

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.
- To identify anomalies or defects in the code.

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 may contain information regarding the loading of data and executables into the test system, directions regarding sign in procedures, instructions regarding the handling of test results, and anything else required to successfully conduct the test.

Test Case Writing (contd.)

<u>Types of Test Cases</u>

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:

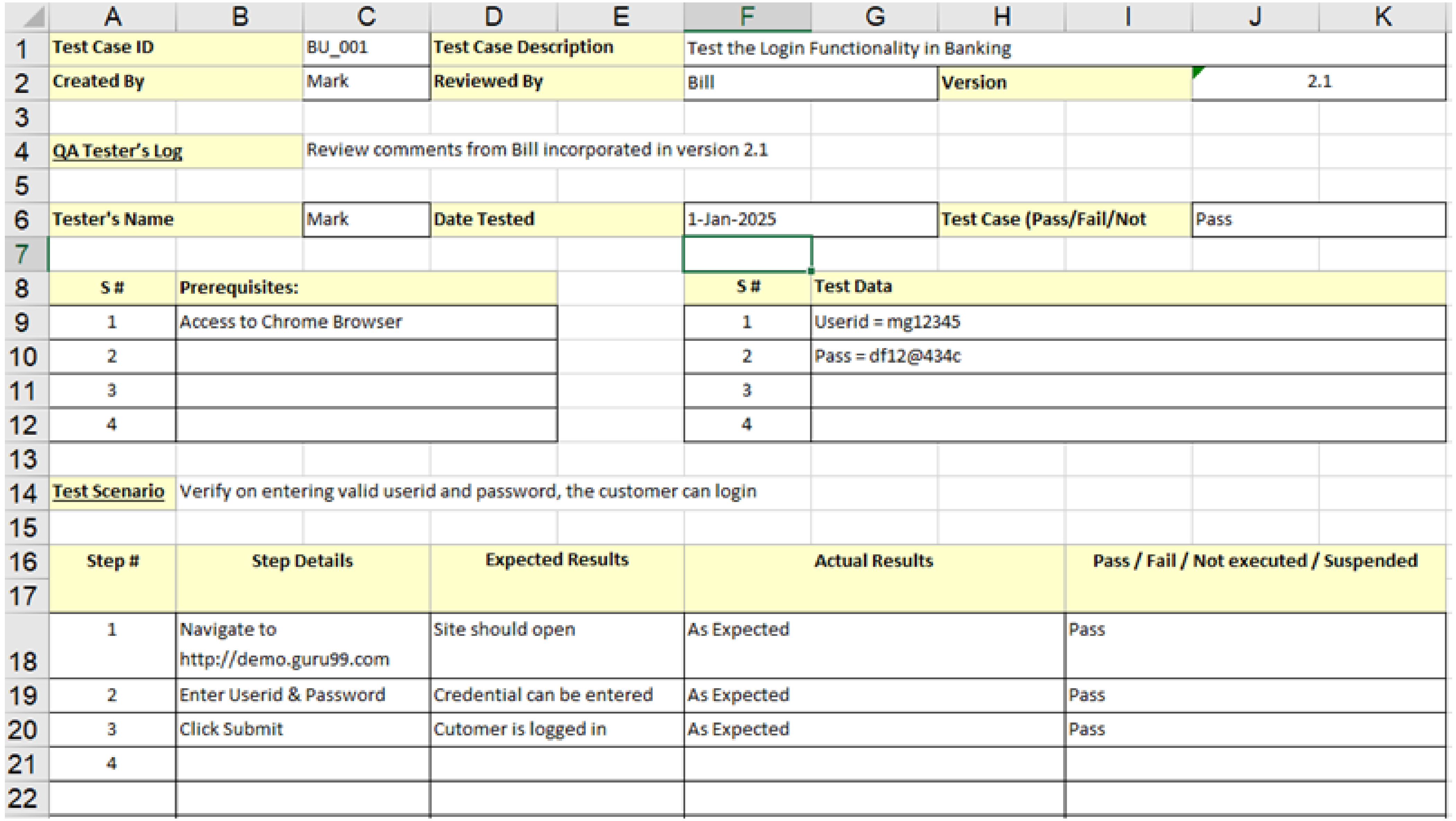
Sr. No.	**Functional Test Cases**	**Type- Negative/ Positive Test Case**
1	Verify if a user will be able to login with a valid username and valid password.	Positive
2	Verify if a user cannot login with a valid username and an invalid password.	Negative
3	Verify the login page for both, when the field 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.)

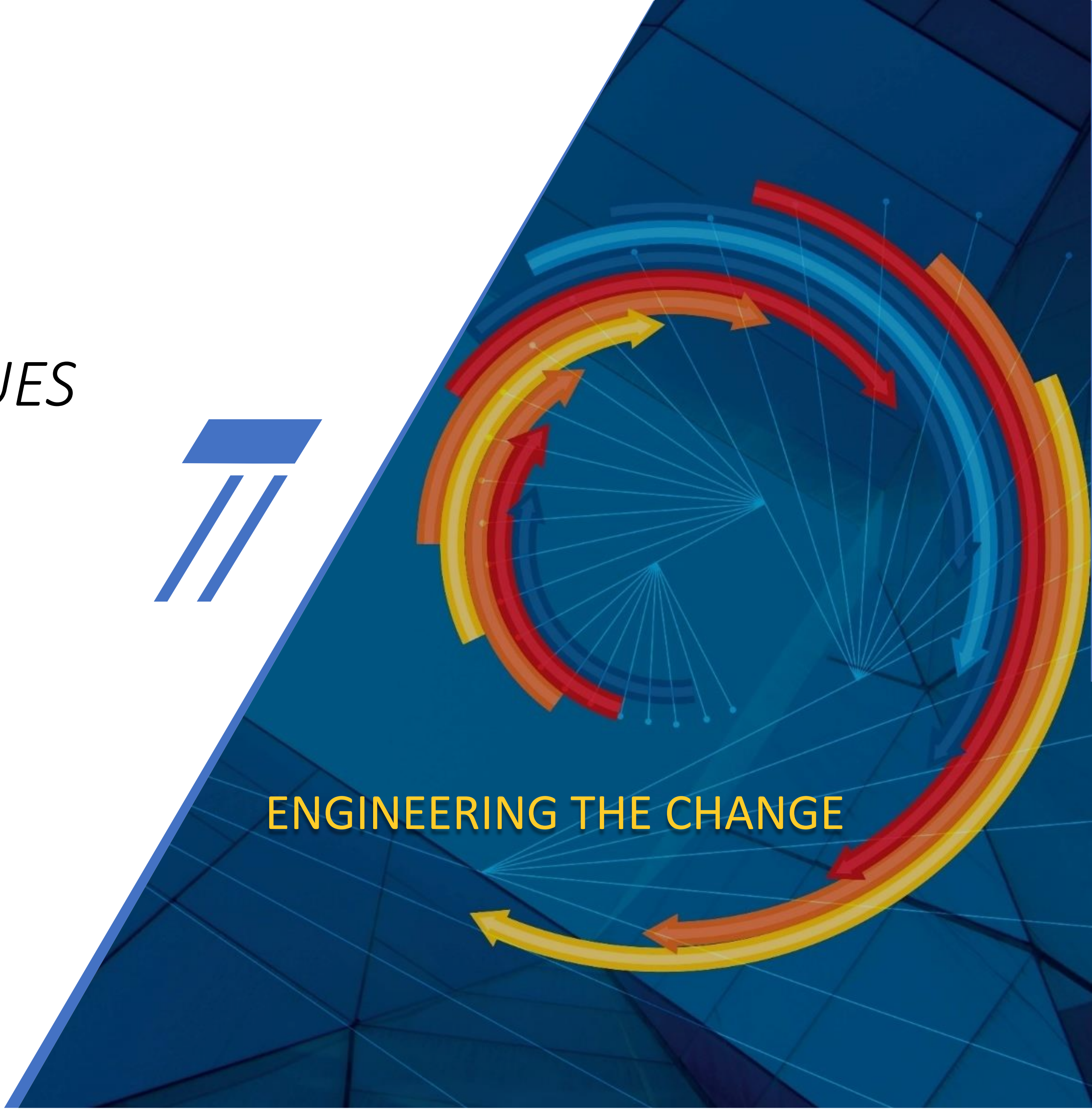


	A	B	C	D	E	F	G	H	I	J	K
1	**Test Case ID**		BU_001	**Test Case Description**		Test the Login Functionality in Banking					
2	**Created By**		Mark	**Reviewed By**		Bill		**Version**		2.1	
3											
4	**QA Tester's Log**		Review comments from Bill incorporated in version 2.1								
5											
6	**Tester's 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) and the technique applied to the test basis

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 is measured as the number of boundary values tested, divided by the total number of identified boundary test values, normally expressed as a percentage.

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.
Achieving 100% decision coverage guarantees 100% statement coverage (but not vice versa).

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.

Contents

TEST MANAGEMENT

TEST MANAGEMENT - Test organization

Independent Testing

Testing tasks may be done by people in a specific testing role, or by people in another role (e.g., customers). A certain degree of independence often makes the tester more effective at finding defects due to differences between the author's and the tester's cognitive biases. Independence is not, however, a replacement for familiarity, and developers can efficiently find many defects in their own code.

Degrees of independence in testing include the following (from low level of independence to high level):

- No independent testers; the only form of testing available is developers testing their own code
- Independent developers or testers within the development teams or the project team; this could be developers testing their colleagues' products
- Independent test team or group within the organization, reporting to project management or executive management
- Independent testers from the business organization or user community, or with specializations in specific test types such as usability, security, performance, regulatory/compliance, or portability
- Independent testers external to the organization, either working on-site (in-house) or off-site (outsourcing)

For most types of projects, it is usually best to have multiple test levels, with some of these levels handled by independent testers. Developers should participate in testing, especially at the lower levels, so as to exercise control over the quality of their own work.

TEST MANAGEMENT - Test organization (contd.)

The way in which independence of testing is implemented varies depending on the software development lifecycle model. For example, in Agile development, testers may be part of a development team. In some organizations using Agile methods, these testers may be considered part of a larger independent test team as well. In addition, in such organizations, product owners may perform acceptance testing to validate user stories at the end of each iteration.

Potential benefits of test independence include:

- Independent testers are likely to recognize different kinds of failures compared to developers because of their different backgrounds, technical perspectives, and biases
- An independent tester can verify, challenge, or disprove assumptions made by stakeholders during specification and implementation of the system
- Independent testers of a vendor can report in an upright and objective manner about the system under test without (political) pressure of the company that hired them

Potential drawbacks of test independence include:

- Isolation from the development team, may lead to a lack of collaboration, delays in providing feedback to the development team, or an adversarial relationship with the development team
- Developers may lose a sense of responsibility for quality
- Independent testers may be seen as a bottleneck
- Independent testers may lack some important information (e.g., about the test object)

TEST MANAGEMENT - Test organization (contd.)

Tasks of a Test Manager and Tester

The activities and tasks performed by these two roles depend on the project and product context, the skills of the people in the roles, and the organization. The test manager is tasked with overall responsibility for the test process and successful leadership of the test activities. The test management role might be performed by a professional test manager, or by a project manager, a development manager, or a quality assurance manager. In larger projects or organizations, several test teams may report to a test manager, test coach, or test coordinator, each team being headed by a test leader or lead tester.

Typical test manager tasks may include:

- Develop or review a test policy and test strategy for the organization
- Plan the test activities by considering the context, and understanding the test objectives and risks. This may include selecting test approaches, estimating test time, effort and cost, acquiring resources, defining test levels and test cycles, and planning defect management
- Write and update the test plan(s)
- Coordinate the test plan(s) with project managers, product owners, and others
- Share testing perspectives with other project activities, such as integration planning
- Initiate the analysis, design, implementation, and execution of tests, monitor test progress and results, and check the status of exit criteria (or definition of done) and facilitate test completion activities
- Prepare and deliver test progress reports and test summary reports based on the information gathered

TEST MANAGEMENT - Test organization (contd.)

• Adapt planning based on test results and progress (sometimes documented in test progress reports, and/or in test summary reports for other testing already completed on the project) and take any actions necessary for test control
• Support setting up the defect management system and adequate configuration management of testware
• Introduce suitable metrics for measuring test progress and evaluating the quality of the testing and the product
• Support the selection and implementation of tools to support the test process, including recommending the budget for tool selection (and possibly purchase and/or support), allocating time and effort for pilot projects, and providing continuing support in the use of the tool(s)
• Decide about the implementation of test environment(s)
• Promote and advocate the testers, the test team, and the test profession within the organization
• Develop the skills and careers of testers (e.g., through training plans, performance evaluations,coaching, etc.)

The way in which the test manager role is carried out varies depending on the software development lifecycle. For example, in Agile development, some of the tasks mentioned above are handled by the Agile team, especially those tasks concerned with the day-to-day testing done within the team, often by a tester working within the team. Some of the tasks that span multiple teams or the entire organization, or that have to do with personnel management, may be done by test managers outside of the development team, who are sometimes called test coaches.

TEST MANAGEMENT - Test organization (contd.)

Typical tester tasks may include:
- Review and contribute to test plans
- Analyze, review, and assess requirements, user stories and acceptance criteria, specifications, and models for testability (i.e., the test basis)
- Identify and document test conditions, and capture traceability between test cases, test conditions, and the test basis
- Design, set up, and verify test environment(s), often coordinating with system administration and network management
- Design and implement test cases and test procedures
- Prepare and acquire test data
- Create the detailed test execution schedule
- Execute tests, evaluate the results, and document deviations from expected results
- Use appropriate tools to facilitate the test process
- Automate tests as needed (may be supported by a developer or a test automation expert)
- Evaluate non-functional characteristics such as performance efficiency, reliability, usability, security, compatibility, and portability
- Review tests developed by others

TEST MANAGEMENT - Test organization (contd.)

People who work on test analysis, test design, specific test types, or test automation may be specialists in these roles. Depending on the risks related to the product and the project, and the software development lifecycle model selected, different people may take over the role of tester at different test levels. For example, at the component testing level and the component integration testing level, the role of a tester is often done by developers. At the acceptance test level, the role of a tester is often done by business analysts, subject matter experts, and users. At the system test level and the system integration test level, the role of a tester is often done by an independent test team. At the operational acceptance test level, the role of a tester is often done by operations and/or systems administration staff.

TEST MANAGEMENT - Test plans, estimates, and strategies

Purpose and Content of a Test Plan

A test plan outlines test activities for development and maintenance projects. Planning is influenced by the test policy and test strategy of the organization, the development lifecycles and methods being used, the scope of testing, objectives, risks, constraints, criticality, testability, and the availability of resources.

As the project and test planning progress, more information becomes available and more detail can be included in the test plan. Test planning is a continuous activity and is performed throughout the product‘s lifecycle. (Note that the product’s lifecycle may extend beyond a project's scope to include the maintenance phase.) Feedback from test activities should be used to recognize changing risks so that planning can be adjusted. Planning may be documented in a master test plan and in separate test plans for test levels, such as system testing and acceptance testing, or for separate test types, such as usability testing and performance testing. Test planning activities may include the following and some of these may be documented in a test plan:

- Determining the scope, objectives, and risks of testing
- Defining the overall approach of testing
- Integrating and coordinating the test activities into the software lifecycle activities
- Making decisions about what to test, the people and other resources required to perform the various test activities, and how test activities will be carried out
- Scheduling of test analysis, design, implementation, execution, and evaluation activities, either on particular dates (e.g., in sequential development) or in the context of each iteration (e.g., in iterative development)
- Selecting metrics for test monitoring and control
- Budgeting for the test activities
- Determining the level of detail and structure for test documentation (e.g., by providing templates or example documents)

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

Test Strategy and Test Approach

A test strategy provides a generalized description of the test process, usually at the product or organizational level. Common types of test strategies include:

- **Analytical**: This type of test strategy is based on an analysis of some factor (e.g., requirement or risk). Risk-based testing is an example of an analytical approach, where tests are designed and prioritized based on the level of risk.
- **Model-Based**: In this type of test strategy, tests are designed based on some model of some required aspect of the product, such as a function, a business process, an internal structure, or a non-functional characteristic (e.g., reliability). Examples of such models include business process models, state models, and reliability growth models.
- **Methodical**: This type of test strategy relies on making systematic use of some predefined set of tests or test conditions, such as a taxonomy of common or likely types of failures, a list of important quality characteristics, or company-wide look-and-feel standards for mobile apps or web pages.
- **Process-compliant (or standard-compliant)**: This type of test strategy involves analyzing, designing, and implementing tests based on external rules and standards, such as those specified by industry-specific standards, by process documentation, by the rigorous identification and use of the test basis, or by any process or standard imposed on or by the organization.
- **Directed (or consultative)**: This type of test strategy is driven primarily by the advice, guidance, or instructions of stakeholders, business domain experts, or technology experts, who may be outside the test team or outside the organization itself.
- **Regression-averse**: This type of test strategy is motivated by a desire to avoid regression of existing capabilities. This test strategy includes reuse of existing testware (especially test cases and test data), extensive automation of regression tests, and standard test suites.
- **Reactive:** In this type of test strategy, testing is reactive to the component or system being tested, and the events occurring during test execution, rather than being pre-planned (as the preceding strategies are). Tests are designed and implemented, and may immediately be executed in response to knowledge gained from prior test results. Exploratory testing is a common technique employed in reactive strategies.

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

An appropriate test strategy is often created by combining several of these types of test strategies. For example, risk-based testing (an analytical strategy) can be combined with exploratory testing (a reactive strategy); they complement each other and may achieve more effective testing when used together.

While the test strategy provides a generalized description of the test process, the test approach tailors the test strategy for a particular project or release. The test approach is the starting point for selecting the test techniques, test levels, and test types, and for defining the entry criteria and exit criteria (or definition of ready and definition of done, respectively). The tailoring of the strategy is based on decisions made in relation to the complexity and goals of the project, the type of product being developed, and product risk analysis. The selected approach depends on the context and may consider factors such as risks, safety, available resources and skills, technology, the nature of the system, test objectives, and regulations.

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

Entry Criteria and Exit Criteria (Definition of Ready and Definition of Done)
In order to exercise effective control over the quality of the software, and of the testing, it is advisable to have criteria which define when a given test activity should start and when the activity is complete. Entry criteria (more typically called definition of ready in Agile development) define the preconditions for undertaking a given test activity. If entry criteria are not met, it is likely that the activity will prove more difficult, more time-consuming, more costly, and more risky. Exit criteria (more typically called definition of done in Agile development) define what conditions must be achieved in order to declare a test level or a set of tests completed. Entry and exit criteria should be defined for each test level and test type, and will differ based on the test objectives.

Typical entry criteria include:
- Availability of testable requirements, user stories, and/or models (e.g., when following a model-based testing strategy)
- Availability of test items that have met the exit criteria for any prior test levels
- Availability of test environment
- Availability of necessary test tools
- Availability of test data and other necessary resources

Typical exit criteria include:
- Planned tests have been executed
- A defined level of coverage (e.g., of requirements, user stories, acceptance criteria, risks, code) has been achieved
- The number of unresolved defects is within an agreed limit
- The number of estimated remaining defects is sufficiently low
- The evaluated levels of reliability, performance efficiency, usability, security, and other relevant quality characteristics are sufficient

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

Even without exit criteria being satisfied, it is also common for test activities to be curtailed due to the budget being expended, the scheduled time being completed, and/or pressure to bring the product to market. It can be acceptable to end testing under such circumstances, if the project stakeholders and business owners have reviewed and accepted the risk to go live without further testing.

Test Execution Schedule

Once the various test cases and test procedures are produced (with some test procedures potentially automated) and assembled into test suites, the test suites can be arranged in a test execution schedule that defines the order in which they are to be run. The test execution schedule should take into account such factors as prioritization, dependencies, confirmation tests, regression tests, and the most efficient sequence for executing the tests.

Ideally, test cases would be ordered to run based on their priority levels, usually by executing the test cases with the highest priority first. However, this practice may not work if the test cases have dependencies or the features being tested have dependencies. If a test case with a higher priority is dependent on a test case with a lower priority, the lower priority test case must be executed first.

Similarly, if there are dependencies across test cases, they must be ordered appropriately regardless of their relative priorities. Confirmation and regression tests must be prioritized as well, based on the importance of rapid feedback on changes, but here again dependencies may apply. In some cases, various sequences of tests are possible, with differing levels of efficiency associated with those sequences. In such cases, trade-offs between efficiency of test execution versus adherence to prioritization must be made.

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

Factors Influencing the Test Effort

Test effort estimation involves predicting the amount of test-related work that will be needed in order to meet the objectives of the testing for a particular project, release, or iteration. Factors influencing the test effort may include characteristics of the product, characteristics of the development process, characteristics of the people, and the test results, as shown below.

Product characteristics

- The risks associated with the product
- The quality of the test basis
- The size of the product
- The complexity of the product domain
- The requirements for quality characteristics (e.g., security, reliability)
- The required level of detail for test documentation
- Requirements for legal and regulatory compliance

Development process characteristics

- The stability and maturity of the organization
- The development model in use
- The test approach
- The tools used
- The test process
- Time pressure

People characteristics

- The skills and experience of the people involved, especially with similar projects and products (e.g., domain knowledge)
- Team cohesion and leadership

Test results

- The number and severity of defects found
- The amount of rework required

TEST MANAGEMENT - Test plans, estimates, and strategies (contd.)

Test Estimation Techniques

There are a number of estimation techniques used to determine the effort required for adequate testing. Two of the most commonly used techniques are:

- The metrics-based technique: estimating the test effort based on metrics of former similar projects, or based on typical values
- The expert-based technique: estimating the test effort based on the experience of the owners of the testing tasks or by experts

For example, in Agile development, burn-down charts are examples of the metrics-based approach as effort remaining is being captured and reported, and is then used to feed into the team's velocity to determine the amount of work the team can do in the next iteration; whereas planning poker is an example of the expert-based approach, as team members are estimating the effort to deliver a feature based on their experience.

Within sequential projects, defect removal models are examples of the metrics-based approach, where volumes of defects and time to remove them are captured and reported, which then provides a basis for estimating future projects of a similar nature; whereas the Wideband Delphi estimation technique is an example of the expert-based approach in which a group of experts provides estimates based on their experience.

Test progress monitoring, control and reporting

The purpose of test monitoring is to gather information and provide feedback and visibility about test activities. Information to be monitored may be collected manually or automatically and should be used to assess test progress and to measure whether the test exit criteria, or the testing tasks associated with an Agile project's definition of done, are satisfied, such as meeting the targets for coverage of product risks, requirements, or acceptance criteria.

Test control describes any guiding or corrective actions taken as a result of information and metrics gathered and (possibly) reported. Actions may cover any test activity and may affect any other software lifecycle activity.
Examples of test control actions include:

- Re-prioritizing tests when an identified risk occurs (e.g., software delivered late)
- Changing the test schedule due to availability or unavailability of a test environment or other resources
- Re-evaluating whether a test item meets an entry or exit criterion due to rework

Metrics Used in Testing
Metrics can be collected during and at the end of test activities in order to assess:

- Progress against the planned schedule and budget
- Current quality of the test object
- Adequacy of the test approach
- Effectiveness of the test activities with respect to the objectives

Test progress monitoring, control and reporting (contd.)

Common test metrics include:

- Percentage of planned work done in test case preparation (or percentage of planned test cases implemented)
- Percentage of planned work done in test environment preparation
- Test case execution (e.g., number of test cases run/not run, test cases passed/failed, and/or test conditions passed/failed)
- Defect information (e.g., defect density, defects found and fixed, failure rate, and confirmation test results)
- Test coverage of requirements, user stories, acceptance criteria, risks, or code
- Task completion, resource allocation and usage, and effort
- Cost of testing, including the cost compared to the benefit of finding the next defect or the cost compared to the benefit of running the next test

Purposes, Contents, and Audiences for Test Reports

The purpose of test reporting is to summarize and communicate test activity information, both during and at the end of a test activity (e.g., a test level). The test report prepared during a test activity may be referred to as a test progress report, while a test report prepared at the end of a test activity may be referred to as a test summary report.

During test monitoring and control, the test manager regularly issues test progress reports for stakeholders. In addition to content common to test progress reports and test summary reports, typical test progress reports may also include:

- The status of the test activities and progress against the test plan
- Factors impeding progress
- Testing planned for the next reporting period
- The quality of the test object

Test progress monitoring, control and reporting (contd.)

When exit criteria are reached, the test manager issues the test summary report. This report provides a summary of the testing performed, based on the latest test progress report and any other relevant information.
Typical test summary reports may include:

- Summary of testing performed
- Information on what occurred during a test period
- Deviations from plan, including deviations in schedule, duration, or effort of test activities
- Status of testing and product quality with respect to the exit criteria or definition of done
- Factors that have blocked or continue to block progress
- Metrics of defects, test cases, test coverage, activity progress, and resource consumption
- Residual risks
- Reusable test work products produced

The contents of a test report will vary depending on the project, the organizational requirements, and the software development lifecycle. For example, a complex project with many stakeholders or a regulated project may require more detailed and rigorous reporting than a quick software update. As another example, in Agile development, test progress reporting may be incorporated into task boards, defect summaries, and burn-down charts, which may be discussed during a daily stand-up meeting.

In addition to tailoring test reports based on the context of the project, test reports should be tailored based on the report's audience. The type and amount of information that should be included for a technical audience or a test team may be different from what would be included in an executive summary report. In the first case, detailed information on defect types and trends may be important. In the latter case, a high-level report (e.g., a status summary of defects by priority, budget, schedule, and test conditions passed/failed/not tested) may be more appropriate.

Configuration management

The purpose of configuration management is to establish and maintain the integrity of the component or system, the testware, and their relationships to one another through the project and product lifecycle. To properly support testing, configuration management may involve ensuring the following:

• All test items are uniquely identified, version controlled, tracked for changes, and related to each other

• All items of test-ware are uniquely identified, version controlled, tracked for changes, related to each other and related to versions of the test item(s) so that traceability can be maintained throughout the test process

• All identified documents and software items are referenced unambiguously in test documentation

During test planning, configuration management procedures and infrastructure (tools) should be identified and implemented.

Risk Management

Definition of Risk

Risk involves the possibility of an event in the future which has negative consequences. The level of risk is determined by the likelihood of the event and the impact (the harm) from that event.

Product and Project Risks

Product risk involves the possibility that a work product (e.g., a specification, component, system, or test) may fail to satisfy the legitimate needs of its users and/or stakeholders. When the product risks are associated with specific quality characteristics of a product (e.g., functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability), product risks are also called quality risks. Examples of product risks include:

- Software might not perform its intended functions according to the specification
- Software might not perform its intended functions according to user, customer, and/or stakeholder needs
- A system architecture may not adequately support some non-functional requirement(s)
- A particular computation may be performed incorrectly in some circumstances
- A loop control structure may be coded incorrectly
- Response-times may be inadequate for a high-performance transaction processing system
- User experience (UX) feedback might not meet product expectations

Project risk involves situations that, should they occur, may have a negative effect on a project's ability to achieve its objectives. Examples of project risks include:

Project issues:

o Delays may occur in delivery, task completion, or satisfaction of exit criteria or definition of done

o Inaccurate estimates, reallocation of funds to higher priority projects, or general cost-cutting across the organization may result in inadequate funding

o Late changes may result in substantial re-work

Risk Management (contd.)

Organizational issues:

o Skills, training, and staff may not be sufficient

o Personnel issues may cause conflict and problems

o Users, business staff, or subject matter experts may not be available due to conflicting business priorities

Political issues:

o Testers may not communicate their needs and/or the test results adequately

o Developers and/or testers may fail to follow up on information found in testing and reviews (e.g., not improving development and testing practices)

o There may be an improper attitude toward, or expectations of, testing (e.g., not appreciating the value of finding defects during testing)

Technical issues:

o Requirements may not be defined well enough

o The requirements may not be met, given existing constraints

o The test environment may not be ready on time

o Data conversion, migration planning, and their tool support may be late

o Weaknesses in the development process may impact the consistency or quality of project work products such as design, code, configuration, test data, and test cases

o Poor defect management and similar problems may result in accumulated defects and other technical debt

Supplier issues:

o A third party may fail to deliver a necessary product or service, or go bankrupt

o Contractual issues may cause problems to the project

Project risks may affect both development activities and test activities. In some cases, project managers are responsible for handling all project risks, but it is not unusual for test managers to have responsibility for test-related project risks.

Risk Management (contd.)

Risk-based Testing and Product Quality

Risk is used to focus the effort required during testing. It is used to decide where and when to start testing and to identify areas that need more attention. Testing is used to reduce the probability of an adverse event occurring, or to reduce the impact of an adverse event. Testing is used as a risk mitigation activity, to provide information about identified risks, as well as providing information on residual (unresolved) risks.

A risk-based approach to testing provides proactive opportunities to reduce the levels of product risk. It involves product risk analysis, which includes the identification of product risks and the assessment of each risk's likelihood and impact. The resulting product risk information is used to guide test planning, the specification, preparation and execution of test cases, and test monitoring and control. Analyzing product risks early contributes to the success of a project.

In a risk-based approach, the results of product risk analysis are used to:

• Determine the test techniques to be employed
• Determine the particular levels and types of testing to be performed (e.g., security testing, accessibility testing)
• Determine the extent of testing to be carried out
• Prioritize testing in an attempt to find the critical defects as early as possible
• Determine whether any activities in addition to testing could be employed to reduce risk (e.g., providing training to inexperienced designers)

Risk-based testing draws on the collective knowledge and insight of the project stakeholders to carry out product risk analysis. To ensure that the likelihood of a product failure is minimized, risk management activities provide a disciplined approach to:

• Analyze (and re-evaluate on a regular basis) what can go wrong (risks)
• Determine which risks are important to deal with
• Implement actions to mitigate those risks
• Make contingency plans to deal with the risks should they become actual events

In addition, testing may identify new risks, help to determine what risks should be mitigated, and lower uncertainty about risks.

Defect Management

Since one of the objectives of testing is to find defects, defects found during testing should be logged. The way in which defects are logged may vary, depending on the context of the component or system being tested, the test level, and the software development lifecycle model. Any defects identified should be investigated and should be tracked from discovery and classification to their resolution (e.g., correction of the defects and successful confirmation testing of the solution, deferral to a subsequent release,
acceptance as a permanent product limitation, etc.). In order to manage all defects to resolution, an organization should establish a defect management process which includes a workflow and rules for classification. This process must be agreed with all those participating in defect management, including architects, designers, developers, testers, and product owners. In some organizations, defect logging and tracking may be very informal.

During the defect management process, some of the reports may turn out to describe false positives, not actual failures due to defects. For example, a test may fail when a network connection is broken or times out. This behavior does not result from a defect in the test object, but is an anomaly that needs to be investigated. Testers should attempt to minimize the number of false positives reported as defects.

Defects may be reported during coding, static analysis, reviews, or during dynamic testing, or use of a software product. Defects may be reported for issues in code or working systems, or in any type of documentation including requirements, user stories and acceptance criteria, development documents, test documents, user manuals, or installation guides. In order to have an effective and efficient defect management process, organizations may define standards for the attributes, classification, and workflow
of defects.

Defect Management (contd.)

Typical defect reports have the following objectives:

- Provide developers and other parties with information about any adverse event that occurred, to enable them to identify specific effects, to isolate the problem with a minimal reproducing test, and to correct the potential defect(s), as needed or to otherwise resolve the problem
- Provide test managers a means of tracking the quality of the work product and the impact on the testing (e.g., if a lot of defects are reported, the testers will have spent a lot of time reporting them instead of running tests, and there will be more confirmation testing needed)
- Provide ideas for development and test process improvement

A defect report filed during dynamic testing typically includes:

- An identifier
- A title and a short summary of the defect being reported
- Date of the defect report, issuing organization, and author
- Identification of the test item (configuration item being tested) and environment
- The development lifecycle phase(s) in which the defect was observed
- A description of the defect to enable reproduction and resolution, including logs, database dumps, screenshots, or recordings (if found during test execution)
- Expected and actual results
- Scope or degree of impact (severity) of the defect on the interests of stakeholder(s)
- Urgency/priority to fix
- State of the defect report (e.g., open, deferred, duplicate, waiting to be fixed, awaiting confirmation testing, re-opened, closed)
- Conclusions, recommendations and approvals

Defect Management (contd.)

• Global issues, such as other areas that may be affected by a change resulting from the defect
• Change history, such as the sequence of actions taken by project team members with respect to the defect to isolate, repair, and confirm it as fixed
• References, including the test case that revealed the problem

Some of these details may be automatically included and/or managed when using defect management tools, e.g., automatic assignment of an identifier, assignment and update of the defect report state during the workflow, etc. Defects found during static testing, particularly reviews, will normally be documented in a different way, e.g., in review meeting notes.

Contents

TESTING ON AGILE DEVELOPMENT

What is Agile Methodology

A tester on an Agile project will work differently than one working on a traditional project. Testers must understand the values and principles that underpin Agile projects, and how testers are an integral part of a whole-team approach together with developers and business representatives. The members in an Agile project communicate with each other early and frequently, which helps with removing defects early and developing a quality product.

Agile Software Development and the Agile Manifesto
The Agile Manifesto contains four statements of values:
•Individuals and interactions over processes and tools
•Working software over comprehensive documentation
•Customer collaboration over contract negotiation
•Responding to change over following a plan

The Agile Manifesto argues that although the concepts on the right have value, those on the left have greater value.

Individuals and Interactions
Agile development is very people-centred. Teams of people build software, and it is through continuous communication and interaction, rather than a reliance on tools or processes, that teams can work most effectively.
Working Software
From a customer perspective, working software is much more useful and valuable than overly detailed documentation and it provides an opportunity to give the development team rapid feedback. In addition, because working software, albeit with reduced functionality, is available much earlier in the development lifecycle, Agile development can confer significant time-to-market advantage. Agile development is, therefore, especially useful in rapidly changing business environments where the problems and/or solutions are unclear or where the business wishes to innovate in new problem domains.

What is Agile Methodology (contd.)

Customer Collaboration

Customers often find great difficulty in specifying the system that they require. Collaborating directly with the customer improves the likelihood of understanding exactly what the customer requires. While having contracts with customers may be important, working in regular and close collaboration with them is likely to bring more success to the project.

Responding to Change

Change is inevitable in software projects. The environment in which the business operates, legislation, competitor activity, technology advances, and other factors can have major influences on the project and its objectives. These factors must be accommodated by the development process. As such, having flexibility in work practices to embrace change is more important than simply adhering rigidly to a plan.

Principles

The core Agile Manifesto values are captured in twelve principles:

•Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

•Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

•Deliver working software frequently, at intervals of between a few weeks to a few months, with a preference to the shorter timescale.

•Business people and developers must work together daily throughout the project.

•Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. Working software is the primary measure of progress.

•Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

•Continuous attention to technical excellence and good design enhances agility.

•Simplicity—the art of maximizing the amount of work not done—is essential.

•The best architectures, requirements, and designs emerge from self-organizing teams.

•At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Agile VS Waterfall Method

What is Waterfall methodology?
Waterfall Model methodology which is also known as Liner Sequential Life Cycle Model. Waterfall Model followed in the sequential order, and so project development team only moves to next phase of development or testing if the previous step completed successfully.

What is the Agile methodology?
Agile methodology is a practice that helps continuous iteration of development and testing in the software development process. In this model, development and testing activities are concurrent, unlike the Waterfall model. This process allows more communication between customers, developers, managers, and testers.

Advantages of Waterfall Model:
- It is one the easiest model to manage. Because of its nature, each phase has specific deliverables and a review process.
- It works well for smaller size projects where requirements are easily understandable.
- Faster delivery of the project
- Process and results are well documented.
- Easily adaptable method for shifting teams
- This project management methodology is beneficial to manage dependencies.

Advantages of the Agile Model:
- It is focused client process. So, it makes sure that the client is continuously involved during every stage.
- Agile teams are extremely motivated and self-organized so it likely to provide a better result from the development projects.
- Agile software development method assures that quality of the development is maintained
- The process is completely based on the incremental progress. Therefore, the client and team know exactly what is complete and what is not. This reduces risk in the development process.

Agile VS Waterfall Method (contd.)

Limitations of Waterfall Model:
It is not an ideal model for a large size project
If the requirement is not clear at the beginning, it is a less effective method.
Very difficult to move back to makes changes in the previous phases.
The testing process starts once development is over. Hence, it has high chances of bugs to be found later in development where they are expensive to fix.

Limitations of Agile Model
It is not useful method for small development projects.
It requires an expert to take important decisions in the meeting.
Cost of implementing an agile method is little more compared to other development methodologies.
The project can easily go off track if the project manager is not clear what outcome he/she wants.

Agile	Waterfall
It separates the project development lifecycle into sprints.	Software development process is divided into distinct phases.
It follows an incremental approach	Waterfall methodology is a sequential design process.
Agile methodology is known for its flexibility.	Waterfall is a structured software development methodology so most times it can be quite rigid.
Agile can be considered as a collection of many different projects.	Software development will be completed as one single project.
Agile is quite a flexible method which allows changes to be made in the project development requirements even if the initial planning has been completed.	There is no scope of changing the requirements once the project development starts.

Agile VS Waterfall Method (contd.)

Agile methodology, follow an iterative development approach because of this planning, development, prototyping and other software development phases may appear more than once.	All the project development phases like designing, development, testing, etc. are completed once in the Waterfall model.
Test plan is reviewed after each sprint	The test plan is rarely discussed during the test phase.
Agile development is a process in which the requirements are expected to change and evolve.	The method is ideal for projects which have definite requirements and changes not at all expected.
In Agile methodology, testing is performed concurrently with software development.	In this methodology, the "Testing" phase comes after the "Build" phase
Agile introduces a product mindset where the software product satisfies needs of its end customers and changes itself as per the customer's demands.	This model shows a project mindset and places its focus completely on accomplishing the project.
Agile methdology works exceptionally well with Time & Materials or non-fixed funding. It may increase stress in fixed-price scenarios.	Reduces risk in the firm fixed price contracts by getting risk agreement at the beginning of the process.
Prefers small but dedicated teams with a high degree of coordination and synchronization.	Team coordination/synchronization is very limited.
Products owner with team prepares requirements just about every day during a project.	Business analyst prepares requirements before the beginning of the project.
Test team can take part in the requirements change without problems.	It is difficult for the test to initiate any change in requirements.
Description of project details can be altered anytime during the SDLC process.	Detail description needs to implement waterfall software development approach.
The Agile Team members are interchangeable, as a result, they work faster. There is also no need for project managers because the projects are managed by the entire team	In the waterfall method, the process is always straightforward so, project manager plays an essential role during every stage of SDLC.

Agile VS Waterfall Method (contd.)

KEY DIFFERENCE

•Waterfall is a Liner Sequential Life Cycle Model whereas Agile is a continuous iteration of development and testing in the software development process.

•Agile methodology is known for its flexibility whereas Waterfall is a structured software development methodology.

•Agile follows an incremental approach whereas the Waterfall methodology is a sequential design process.

•Agile performs testing concurrently with software development whereas in Waterfall methodology testing comes after the “Build” phase.

•Agile allows changes in project development requirement whereas Waterfall has no scope of changing the requirements once the project development starts.

Overview of Agile Testing/TDD/BDD/Automation Approach

There are certain testing practices that can be followed in every development project (agile or not) to produce quality products. These include writing tests in advance to express proper behavior, focusing on early defect prevention, detection, and removal, and ensuring that the right test types are run at the right time and as part of the right test level. Agile practitioners aim to introduce these practices early. Testers in Agile projects play a key role in guiding the use of these testing practices throughout the lifecycle.

Test-Driven Development

Test-driven development(TDD)is used to develop code guided by automated test cases. The process for test-driven development is:

•Add a test that captures the programmer's concept of the desired functioning of a small piece of code
•Run the test, which should fail since the code doesn't exist
•Write the code and run the test in a tight loop until the test passes
•Refactor the code after the test is passed, re-running the test to ensure it continues to pass against the re-factored code
•Repeat this process for the next small piece of code, running the previous tests as well as the added tests

Benefits of Test-Driven Development:

•Helps reduce the amount of time required for rework
•Helps explore bugs or errors very quickly
•Helps get faster feedback
•Encourages the development of cleaner and better designs
•Enhances the productivity of the programmer
•Allows any team member to start working on the code in the absence of a specific team member. This encourages knowledge sharing and collaboration
•Gives the programmer confidence to change the large architecture of an application easily
•Results in the creation of extensive code that is flexible and easy to maintain

Overview of Agile Testing/TDD/BDD/Automation Approach (contd.)

Behavioural-Driven Development

Business-Driven Development (BDD) is a testing approach derived from the Test-Driven Development (TDD) methodology. In BDD, tests are mainly based on systems behaviour. This approach defines various ways to develop a feature based on its behaviour. In most cases, the ***Given-When-Then*** approach is used for writing test cases. Let's take an example for better understanding:

- **Given** the user has entered valid login credentials
- **When** a user clicks on the login button
- **Then** display the successful validation message

Key benefits of Behavioural-Driven Development approach:

- Helps reach a wider audience by the usage of non-technical language
- Focuses on how the system should behave from the customer's and the developer's perspective
- BDD Is a cost-effective technique
- Reduces efforts needed to verify any post-deployment defects

How does BDD help in SDLC?

Debugging the errors in the latter stages of the development life cycle often proves to be very expensive. In the majority of cases, ambiguity in understanding the requirements is the root cause behind this. One needs to ensure that all the development efforts remain aligned towards fulfilling pre-determined requirements. BDD allows developers to do the above by :

- Allowing the requirements to be defined in a standard approach using simple English
- Providing several ways to illustrate real-world scenarios for understanding requirements
- Providing a platform that enables the tech and non-tech teams to collaborate and understand the requirements

Scrum

Scrum is an agile framework for developing, delivering, and sustaining complex products, with an initial emphasis on software development, although it has been used in other fields including research, sales, marketing and advanced technologies. It is designed for teams of ten or fewer members, who break their work into goals that can be completed within time-boxed iterations, called *sprints*, no longer than one month and most commonly two weeks. The Scrum Team track progress in 15-minute time-boxed daily meetings, called daily scrums. At the end of the sprint, the team holds sprint review, to demonstrate the work done, and sprint retrospective to continuously improve.
Scrum relies on cross-functional teams to deliver products and services in ***short cycles***, enabling:

- Fast feedback
- Continuous improvement
- Rapid adaptation to change
- Accelerated delivery

Understanding the Scrum Flow
At its heart, Scrum works by breaking large products and services into small pieces that can be completed (and **potentially released**) by a cross-functional team in a short timeframe. Scrum teams **inspect** each batch of functionality as it is completed and then adapt what will be created next based on **learning** and **feedback**, **minimizing risk and reducing waste**. This cycle repeats until the full product or service is delivered—one that meets customer needs because the business has the opportunity to adjust the fit at the end of each timeframe.

Benefits of Using Scrum
Because Scrum teams are continuously creating only the highest priority chunks of functionality, the business is assured maximum return on investment. Scrum helps businesses:

- Innovate faster
- Move from idea to delivery more quickly
- Drive higher customer satisfaction
- Increase employee morale

Role of Tester in Agile

The role of a tester in an Agile team includes activities that generate and provide feedback not only on test status, test progress, and product quality, but also on process quality. In addition to the activities described else where in this syllabus, these activities include:

- Understanding, implementing, and updating the test strategy
- Measuring and reporting test coverage across all applicable coverage dimensions
- Ensuring proper use of testing tools
- Configuring, using, and managing test environments and test data
- Reporting defects and working with the team to resolve them
- Coaching other team members in relevant aspects of testing
- Ensuring the appropriate testing tasks are scheduled during release and iteration planning
- Actively collaborating with developers and business stakeholders to clarify requirements, especially in terms of testability, consistency, and completeness
- Participating proactively in team retrospectives, suggesting and implementing improvements.

Within an Agile team, each team member is responsible for product quality and plays a role in performing test-related tasks. Agile organizations may encounter some test-related organizational risks:

- Testers work so closely to developers that they lose the appropriate tester mindset
- Testers become tolerant of or silent about inefficient, ineffective, or low-quality practices within the team
- Testers cannot keep pace with the incoming changes in time-constrained iterations

Agile Software Development Approaches

Extreme Programming

Extreme Programming (XP),originally introduced by Kent Beck, is an Agile approach to software development described by certain values, principles, and development practices. XP embraces five values to guide development: communication, simplicity, feedback, courage, and respect. XP describes a set of principles as additional guidelines: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

Kanban

Kanban is a management approach that is sometimes used in Agile projects. The general objective is to visualize and optimize the flow of work within a value-added chain. Kanban utilizes three instruments:

• **Kanban Board**: The value chain to be managed is visualized by a Kanban board. Each column shows a station, which is a set of related activities, e.g., development or testing. The items to be produced or tasks to be processed are symbolized by tickets moving from left to right across the board through the stations.

• **Work-in-Progress Limit**: The amount of parallel active tasks is strictly limited. This is controlled by the maximum number of tickets allowed for a station and/or globally for the board. Whenever a station has free capacity, the worker pulls a ticket from the predecessor station.

• **Lead Time**: Kanban is used to optimize the continuous flow of tasks by minimizing the (average) lead time for the complete value stream. Kanban features some similarities to Scrum. In both frameworks, visualizing the active tasks (e.g., on a public whiteboard) provides transparency of content and progress of tasks. Tasks not yet scheduled are waiting in a backlog and moved onto the Kanban board as soon as there is new space (production capacity) available. Iterations or sprints are optional in Kanban. The Kanban process allows releasing its deliverables item by item, rather than as part of a release. Timeboxing as a synchronizing mechanism, therefore, is optional, unlike in Scrum, which synchronizes all tasks within a sprint.

Contents

AUTOMATION TESTING

What is Test Automation

Software Test automation makes use of specialized tools to control the execution of tests and compares the actual results against the expected result. Usually, regression tests, which are repetitive actions, are automated.
Testing Tools not only helps us to perform regression tests but also helps us to automate data set up generation, product installation, GUI interaction, defect logging, etc. Automation tools are used for both Functional and Non-Functional testing.

Test automation is expected to help run many test cases consistently and repeatedly on different versions of the SUT and/or environments. But test automation is more than a mechanism for running a test suite without human interaction. It involves a process of designing the test-ware, including:

- Software
- Documentation
- Test cases
- Test environments
- Test data

Why Test Automation

Objectives of test automation include:

- Improving test efficiency
- Providing wider function coverage
- Reducing the total test cost
- Performing tests that manual testers cannot
- Shortening the test execution period
- Increasing the test frequency/reducing the time required for test cycles

Benefits & Challenges

Advantages of test automation include:

- More tests can be run per build
- The possibility to create tests that cannot be done manually (real-time, remote, parallel tests)
- Tests can be more complex
- Tests run faster
- Tests are less subject to operator error
- More effective and efficient use of testing resources
- Quicker feedback regarding software quality
- Improved system reliability(e.g. repeatability, consistency)
- Improved consistency of tests

Disadvantages of test automation include:

- Additional costs are involved
- Initial investment to setup TAS
- Requires additional technologies
- Team needs to have development and automation skills
- On-going TAS maintenance requirement
- Can distract from testing objectives, e.g. focusing on automating tests cases at the expense of executing tests
- Tests can become more complex
- Additional errors may be introduced by automation

Limitations of test automation include:

- Not all manual tests can be automated
- The automation can only check machine-interpretable results
- The automation can only check actual results that can be verified by an automated test
- Not a replacement for exploratory testing

Manual Testing Vs Test Automation

Software testing is a huge domain, but it can be broadly categorized into two areas: manual testing and automated testing.

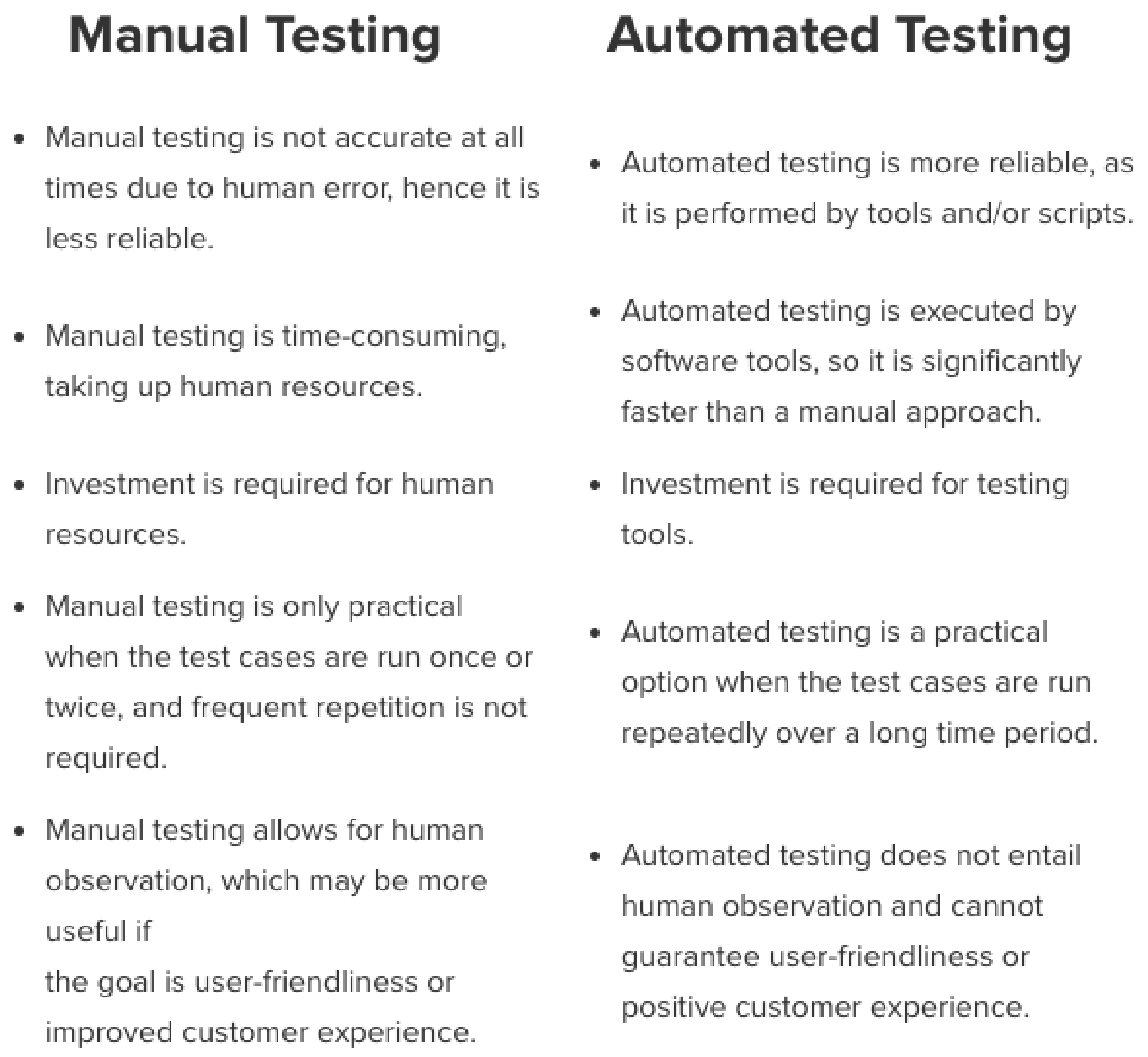
Both manual and automated testing offer benefits and disadvantages. It’s worth knowing the difference, and when to use one or the other for best results.

In manual testing (as the name suggests), test cases are executed manually (by a human, that is) without any support from tools or scripts. But with automated testing, test cases are executed with the assistance of tools, scripts, and software.

Testing is an integral part of any successful software project. The type of testing (manual or automated) depends on various factors, including project requirements, budget, timeline, expertise, and suitability. Three vital factors of any project are of course time, cost, and quality - the goal of any successful project is to reduce the cost and time required to complete it successfully while maintaining quality output. When it comes to testing, one type may accomplish this goal better than the other.

Manual vs. Automated Testing: the Pros and Cons
Manual testing and automated testing cover two vast areas. Within each category, specific testing methods are available, such as black box testing, white box testing, integration testing, system testing, performance testing, and load testing. Some of these methods are better suited to manual testing, and some are best performed through automation. Here’s a brief comparison of each type, along with some pros and cons:

Manual Testing Vs Test Automation (contd.)



Manual Testing	Automated Testing
• Manual testing is not accurate at all times due to human error, hence it is less reliable.	• Automated testing is more reliable, as it is performed by tools and/or scripts.
• Manual testing is time-consuming, taking up human resources.	• Automated testing is executed by software tools, so it is significantly faster than a manual approach.
• Investment is required for human resources.	• Investment is required for testing tools.
• Manual testing is only practical when the test cases are run once or twice, and frequent repetition is not required.	• Automated testing is a practical option when the test cases are run repeatedly over a long time period.
• Manual testing allows for human observation, which may be more useful if the goal is user-friendliness or improved customer experience.	• Automated testing does not entail human observation and cannot guarantee user-friendliness or positive customer experience.

When to automate

Manual testing is best suited to the following areas/scenarios:
Exploratory Testing: This type of testing requires the tester's knowledge, experience, analytical/logical skills, creativity, and intuition. The test is characterized here by poorly written specification documentation, and/or a short time for execution. We need the human skills to execute the testing process in this scenario.
Usability Testing: This is an area in which you need to measure how user-friendly, efficient, or convenient the software or product is for the end users. Here, human observation is the most important factor, so a manual approach is preferable.
Ad-hoc Testing: In this scenario, there is no specific approach. It is a totally unplanned method of testing where the understanding and insight of the tester is the only important factor.

Automated testing is the preferred option in the following areas/scenarios:
Regression Testing: Here, automated testing is suitable because of frequent code changes and the ability to run the regressions in a timely manner.
Load Testing: Automated testing is also the best way to complete the testing efficiently when it comes to load testing.
Repeated Execution: Testing which requires the repeated execution of a task is best automated.
Performance Testing: Similarly, testing which requires the simulation of thousands of concurrent users requires automation.

What to automate

Automation's benefits can be applied to just about any department within an enterprise; from HR to Accounts, Dev to Ops - even the mailroom. However, certain processes are more suited to automation than others. There are telltale signs to watch out for which indicate a process is primed for automation.

Medium and High Volume

Workflows vary dramatically in size. They can consist of simple processes which are composed of few steps, to processes requiring dozens, if not hundreds of items.

When we think about workflows with minimal steps or items, we should ask 'does it make business sense to automate this process?' Conversely, processes with medium and high volume items are clearly business-pivotal processes, primed for automation.

Manual Completion Requires Three or More Users

Generally, if a repeatable task involves three or more people, the likelihood is that it would be more efficient if it was automated. There is less chance of a communication breakdown, making it more secure and more accurate. Furthermore, by automating such routines, you'll free up the man hours of at least three individuals.

The Process Relies Upon Time-Sensitive Activities

Automation creates a log of when items pass through the workflow; recording when issues occur, when they are resolved and when they are actioned. By analyzing these logs, it's easy to spot where bottlenecks occur in your process. New workflow rules can be attributed to overcome these, thus ensuring deadlines are consistently met.

A Process Impacts Upon Various Systems

If a workflow item requires the aid of, or touches upon, various other systems and tools in the tech-stack, then guess what? Yep, it's ready for automation. Rules can applied which grant and restrict permissions to agents, enabling the timely completion of the process as well as ensuring security protocols are adhered to.

Transparent Processes Require Automation

Remaining compliant and providing full audit trails is crucial. If something goes wrong with a manual process, it's nearly impossible to provide a full accurate trail of exactly what happened, meaning the failure can't be properly identified. With automation, you have no such worries. A dedicated automation solution provides governable audit trails as standard, supporting compliance needs.

A typical Test Automation Process

Determine the scope
Any process starts with the definition. Therefore, before implementing test automation, you should **determine the automation scope**. When starting tests development, a QA engineer should first define the order according to the tests' priority rate.

Prepare to automate
The following points need to be considered while preparation for automation:
- Automated tests should cover the most stable part of the functionality and the one that is tested for about 3-4 times per week.
- As a rule, smoke tests (or other regression tests) are chosen for that very purpose.

Select tools for automation
As soon as the scope is defined, a QA engineer should **select test automation tools**. The tested interfaces define the package of applied tools. Different types of interfaces presuppose different tools' range.
There is no any one-size-fits-all solution. The choice of a test automation tool will depend on the technology the software is built on. For example, QTP does not support Informatica. That means the tool cannot be used for the software. We prefer the most reliable and proven solutions: Selenium WebDriver, Coded UI, Ranorex, TestStack.White, Appium, Xamarin, and many more.

Develop the framework
The framework is the basis for further automated tests' development. It provides an opportunity to optimize test development efforts by re-using the code. You can utilize any of the ready-to-use frameworks presented on the market, like the Robot framework for Selenium.
The usefulness of frameworks is hard to underestimate. Thanks to frameworks, it's possible to maintain consistency of testing and improve re-usability. You can also count on minimizing code usage and improving the structure of tests, which is a perfect scenario for long-term projects.

A typical Test Automation Process (contd.)

Configure the environment
All the tests run in **the environment**, which is to be **well-configured**. Upon this step, you should create and support the environment to successfully run automated tests and store the results.
Test automation will require **test data**, which means you are to **prepare** the set of files and test accounts beforehand. Otherwise, you tackle risks that may damage the process and provide you with irrelevant test results.

Start to automate
Eventually, when all the preliminary preparations are done, testers can get down to **automated test development**. A regular process of providing new automated tests includes the following points:
• Selection of the manual test case according to the stated priorities
• Code writing for the automated test
• Adding the automated test to the debug test execution
• Adding the automated test to the test execution for newly created builds.
As well as analysis, the **tests need support**, which presupposes the process of updating automated tests along with the updates of the interface and/or business logic.

When the tests are launched, they should be **monitored**. You cannot let them go along without tracking the process. While monitoring the automated test, remember to take into consideration the following aspects:
• Automated test coverage, cost per test
• Useful vs. irrelevant results after test execution
• Cost per test
• The scope of support in comparison with the number of executed tests
• Economic effect (ROI, return on investment).

Test Automation Tools

Here, is a simple process to determine the best tool for your project needs

- Identify the tests that need to be automated
- Research and analyze the automation tools that meet your automation needs and budget
- Based on the requirements, shortlist two most suitable tools
- Do a pilot for two best tools and select the better one
- Discuss the chosen automation tools with other stakeholders, explain the choice, and get their approval
- Proceed to test automation

There is no best tool in the market. You need to find the best tool for your test automation project goals.

Advance Automation Topics – AI, Bots, CI/CD/Devops

AI

The terms *Artificial Intelligence* and *Automation* are often used interchangeably. They're are associated with software or physical robots and other machines that allow us to operate more efficiently and effectively — whether it's a mechanical construct piecing together a car or sending a follow-up email the day after your customer hasn't completed his order.

However, the are pretty big differences between complexity level of both systems. Automation is basically making a hardware or software that is capable of doing things automatically — without human intervention.

Artificial Intelligence, however, is a science and engineering of making intelligent machines (according to John McCarthy, person who coined this term). AI is all about trying to make machines or software mimic, and eventually supersede human behaviour and intelligence.

Bots or Robotic Process Automation (RPA)

The basic difference between automation and robotics can be seen in their definitions:

Automation — Automation means using computer software, machines or other technology to carry out a task which would otherwise be done by a human worker. There are many types of automation, ranging from the fully mechanical to the fully virtual, and from the very simple to the mind-blowingly complex.

Robotics — Robotics is a branch of engineering which incorporates multiple disciplines to design, build, program and use robotic machines.

There are obviously crossovers between the two. Robots are used to automate some physical tasks, such as in manufacturing. However, many types of automation have nothing to do with physical robots. Also, many branches of robotics have nothing to do with automation.

Advance Automation Topics – AI, Bots, CI/CD/Devops (contd.)

Continuous integration (CI) is a software engineering practice where members of a team integrate their work with increasing frequency. In keeping with CI practice, teams strive to integrate at least daily and even hourly, approaching integration that occurs "continuous-ly."
Historically, integration has been a costly engineering activity. So, to avoid thrash, CI emphasizes automation tools that drive build and test, ultimately focusing on achieving a software-defined life cycle. When CI is successful, build and integration effort drops, and teams can detect integration errors as quickly as practical.

Continuous delivery (CD) is to packaging and deployment what CI is to build and test. Teams practicing CD can build, configure, and package software and orchestrate its deployment in such a way that it can be released to production in a software-defined manner (low cost, high automation) at any time.
High-functioning CI/CD practices directly facilitate agile development because software change reaches production more frequently. As a result, customers have more opportunities to experience and provide feedback on change.

DevOps focuses on limitations of culture and roles as agile development does process. The intention of DevOps is to avoid the negative impact that overspecialization and stovepiping roles in an organization have on preventing rapid or even effective response to production issues. DevOps organizations break down the barriers between Operations and Engineering by cross-training each team in the other's skills. This approach improves everyone's ability to appreciate and participate in each other's tasks and leads to more high-quality collaboration and more frequent communication.

THANK YOU

L&T Technology Services