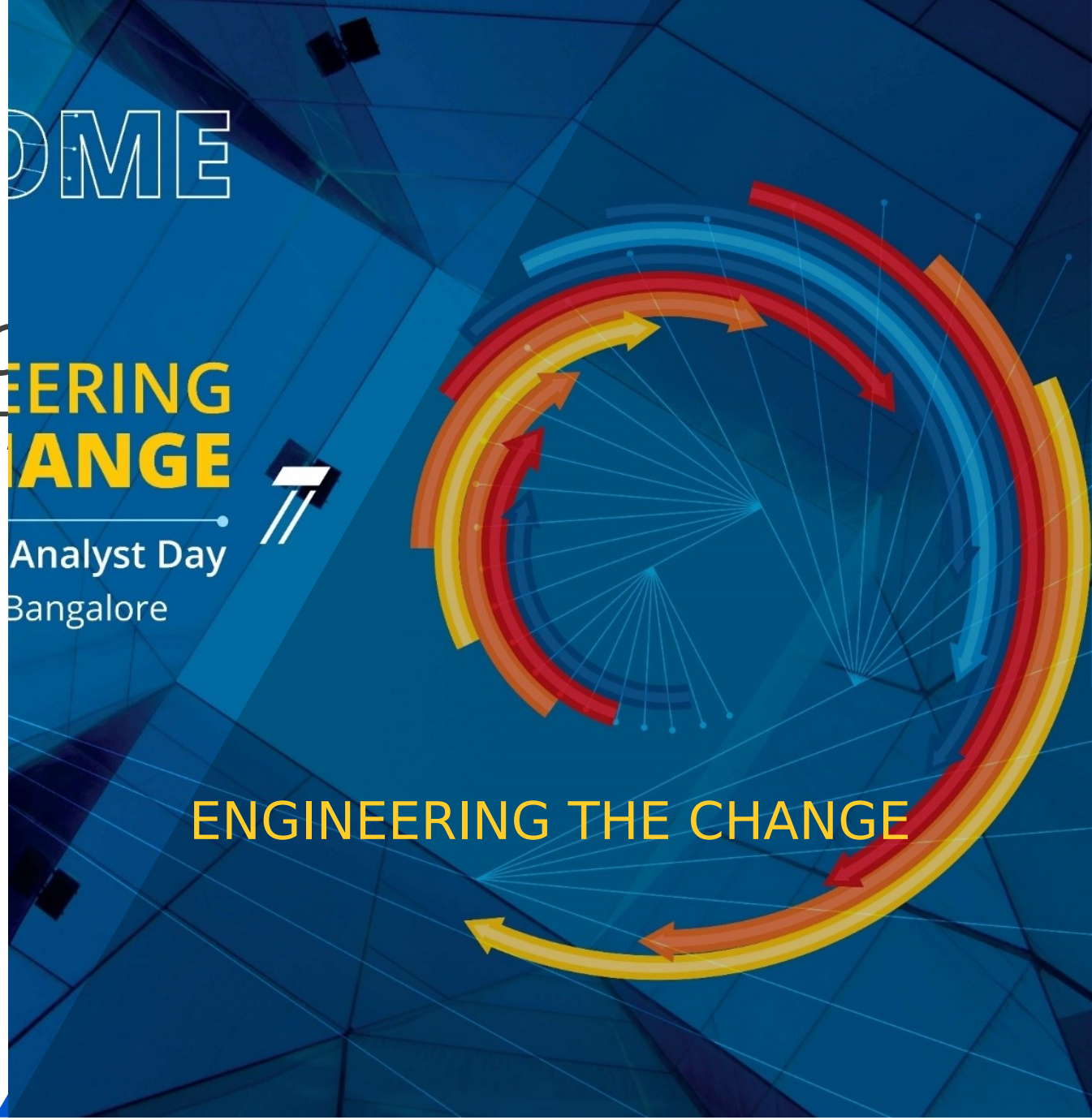
Manual Testing / TDLC
Fundamentals Of Test

L&T Technology Services

Date



Contents

FUNDAMENTALS OF TESTING

TESTING THROUGHOUT THE SOFTWARE LIFE CYCLE

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

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

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.

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

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

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

THANK YOU