

Chapter-II

Testing Throughout the Software Development Lifecycle

Learning Objectives

The student learns -

- how testing is incorporated into different development approaches.
- the concepts of test-first approaches, as well as DevOps.
- about the different -
 - test levels,
 - test types, and
 - maintenance testing.



2.1 Testing in the Context of a Software Development Lifecycle

2.1.1 Impact of the Software Development Lifecycle on Testing

2.1.2 Software Development Lifecycle and Good Testing Practices

2.1.3 Testing as a Driver for Software Development

2.1.4 DevOps and Testing

2.1.5 Shift-Left Approach

2.1.6 Retrospectives and Process Improvement

2.1 Testing in the Context of a Software Development Lifecycle /1

- A Software Development Lifecycle (SDLC) model is an abstract, high-level representation of the software development process.
- SDLC model defines how different development phases and types of activities performed within this process relate to each other.
- Examples of SDLC models include: **sequential development models** (e.g., waterfall model, V-model), **iterative development models** (e.g., spiral model, prototyping), and **incremental development models** (e.g., Unified Process).

2.1 Testing in the Context of a Software Development Lifecycle /2

- Some activities within software development processes can also be described by more detailed software development methods and Agile practices.
- Examples include: Acceptance Test-Driven Development (ATDD), Behavior-Driven Development (BDD), Domain-Driven Design (DDD), eXtreme Programming (XP), Feature-Driven Development (FDD), Kanban, Lean IT, Scrum, and Test-Driven Development (TDD).

2.1.1. Impact of the Software Development Lifecycle on Testing

Testing must be adapted to the SDLC to succeed. The choice of the SDLC impacts on the:

- Scope and timing of test activities (e.g., test levels and test types)
- Level of detail of test documentation
- Choice of test techniques and test approach
- Extent of test automation
- Role and responsibilities of a tester

2.1.1. Impact of the Software Development Lifecycle on Testing

- In **sequential development models**, in the initial phases testers typically participate in requirement reviews, test analysis, and test design. The executable code is usually created in the later phases, so typically dynamic testing cannot be performed early in the SDLC.
- In some **iterative and incremental development models**, it is assumed that each iteration delivers a working prototype or product increment. This implies that in each iteration both static and dynamic testing may be performed at all test levels. Frequent delivery of increments requires fast feedback and extensive regression testing.
- **Agile software development** assumes that change may occur throughout the project. Therefore, lightweight work product documentation and extensive test automation to make regression testing easier are favored in agile projects. Also, most of the manual testing tends to be done using experience-based test techniques that do not require extensive prior test analysis and design

2.1.2. Software Development Lifecycle and Good Testing Practices

Good testing practices, independent of the chosen SDLC model, include the following:

- For every software development activity, there is a corresponding test activity
- Different test levels have specific and different test objectives
- Test analysis and design for a given test level begins during the corresponding development phase of the SDLC, so that testing can adhere to the principle of early testing
- Testers are involved in reviewing work products as soon as drafts of this documentation are available, so that this earlier testing and defect detection can support the shift-left strategy

2.1.3. Testing as a Driver for Software Development /1

- TDD, ATDD and BDD are similar development approaches, where tests are defined as a means of directing development.
- Each of these approaches implements the principle of early testing and follows a shift-left approach, since the tests are defined before the code is written.
- They support an iterative development model.

These approaches are characterized as follows:

2.1.3. Testing as a Driver for Software Development /2

Test-Driven Development (TDD):

- Directs the coding through test cases (instead of extensive software design) (Beck 2003)
- Tests are written first, then the code is written to satisfy the tests, and then the tests and code are refactored

2.1.3. Testing as a Driver for Software Development /3

Acceptance Test-Driven Development (ATDD):

- Derives tests from **acceptance criteria** as part of the system design process
- Tests are written before the part of the application is developed to satisfy the tests



Acceptance Criteria

Acceptance criteria are specific conditions or requirements that must be met for a project, feature, or task to be considered complete and accepted by stakeholders. They define the expected outcomes and functionality of the deliverables. Acceptance criteria are typically written in a clear and concise manner to ensure that there is a shared understanding between the development team and the stakeholders about what constitutes a successful outcome.

e.g. For a user registration feature:

- The user should be able to create an account with a valid email address and password.
- Upon successful registration, the user should receive a confirmation email.
- The user's information should be stored securely in the database.

2.1.3. Testing as a Driver for Software Development /4

Behavior-Driven Development (BDD):

- Expresses the desired behavior of an application with test cases written in a simple form of natural language, which is easy to understand by stakeholders – usually using the **Given/When/Then** format.
- Test cases are then automatically translated into executable tests

Writing Acceptance Criteria using the Given-When-Then format

Feature: Order Processing

Scenario: Successful Order Placement

Given a customer has selected items for purchase

When the customer proceeds to checkout

And provides valid shipping and billing information

And confirms the order

Then the order should be recorded in the system

And the customer should receive an order confirmation email

And the inventory should be updated to reflect the purchased items

Writing Acceptance Criteria using the Given-When-Then format

Feature: Order Processing

Scenario: Out of Stock Item

Given a customer has selected items for purchase

And one or more of the items are out of stock

When the customer proceeds to checkout

Then the customer should be notified about the out of stock items

And given the option to remove or replace the item

And the order total should be recalculated accordingly

2.1.4. DevOps and Testing

DevOps is a set of practices and principles that aim to improve collaboration and communication between development (including testing) and operations teams throughout the software development lifecycle (SDLC). It involves automating processes, continuous integration and delivery (CI/CD), and using various tools to streamline software development, deployment, and operations.

CI/CT/CD

2.1.4. DevOps and Testing: Relationship between DevOps and Testing

DevOps and testing are closely intertwined and rely on each other to ensure the delivery of high-quality software. Here's how they complement each other:

- **Test Automation:** testing teams automate test cases, test data generation, and test execution. Test automation helps to achieve faster feedback loops, identify defects early, and reduce the overall testing effort.
- **Continuous Testing:** ensures that each software change is thoroughly tested, enabling faster and more reliable delivery of software updates.
- **Collaboration:** Developers, operations personnel, and testers work closely together to ensure that all aspects of software quality, performance, and reliability are addressed
- **Shift-Left Testing:** Testers actively participate in requirements gathering, design discussions, and code reviews. This approach helps identify potential issues or ambiguities in the early stages, leading to better software quality.
- **Monitoring and Feedback:** Testing provides feedback on the quality and performance of the software under different conditions.
- **Continuous Improvement:** Through regular retrospectives, feedback analysis, and metrics, teams can identify areas of improvement in the development, testing, and deployment processes, leading to enhanced software quality and delivery.

2.1.5. Shift-Left Approach

The principle of early testing (see section 1.3) is sometimes referred to as shift-left because it is an approach where testing is performed earlier in the SDLC. Shift-left normally suggests that testing should be done earlier (e.g., not waiting for code to be implemented or for components to be integrated), but it does not mean that testing later in the SDLC should be neglected.



2.1.5. Shift-Left Approach

- The "Shift-Left" approach is a concept commonly used in software development and testing methodologies. It refers to the practice of moving tasks, such as testing and quality assurance, earlier in the software development lifecycle (SDLC). The goal is to identify and address issues as early as possible, thereby reducing the overall cost and effort required to fix them.
- The Shift-Left approach advocates for integrating testing and quality assurance activities earlier in the SDLC, preferably from the initial stages of development. By doing so, potential issues can be identified and resolved promptly, reducing the chances of them propagating to later stages. This approach emphasizes a proactive and iterative approach to quality, aiming to catch defects as early as possible.

2.1.5. Shift-Left Approach


Here are a few key aspects of the Shift-Left approach:

- **Early involvement**
 - involved right from the requirements gathering and design phases. Testers collaborate with developers, business analysts, and other stakeholders to ensure that testability and quality considerations are addressed early on
- **Test automation**
 - Test automation frameworks and tools enable the creation of a robust suite of automated tests that can be executed continuously.
- **Continuous integration and continuous delivery (CI/CD)**
 - integrating testing and quality assurance activities into the CI/CD pipeline, software changes are continuously validated, reducing the chances of introducing defects
- **Collaboration and communication**
 - Early involvement of testers allows them to provide feedback and share their expertise, improving the overall quality of the software.

2.1.6. Retrospectives and Process Improvement

Retrospectives (also known as “post-project meetings”) are often held at the end of a project or an iteration, at a release milestone, or can be held when needed.

In these meetings the participants (testers, developers, architects, product owner, business analysts) discuss:

- 
- What was successful, and should be retained?
 - What was not successful and could be improved?
 - How to incorporate the improvements and retain the successes in the future?

2.1.6. Retrospectives and Process Improvement /2

The results should be recorded and are normally part of the test completion report. Retrospectives are critical for the successful implementation of continuous improvement and it is important that any recommended improvements are followed up.

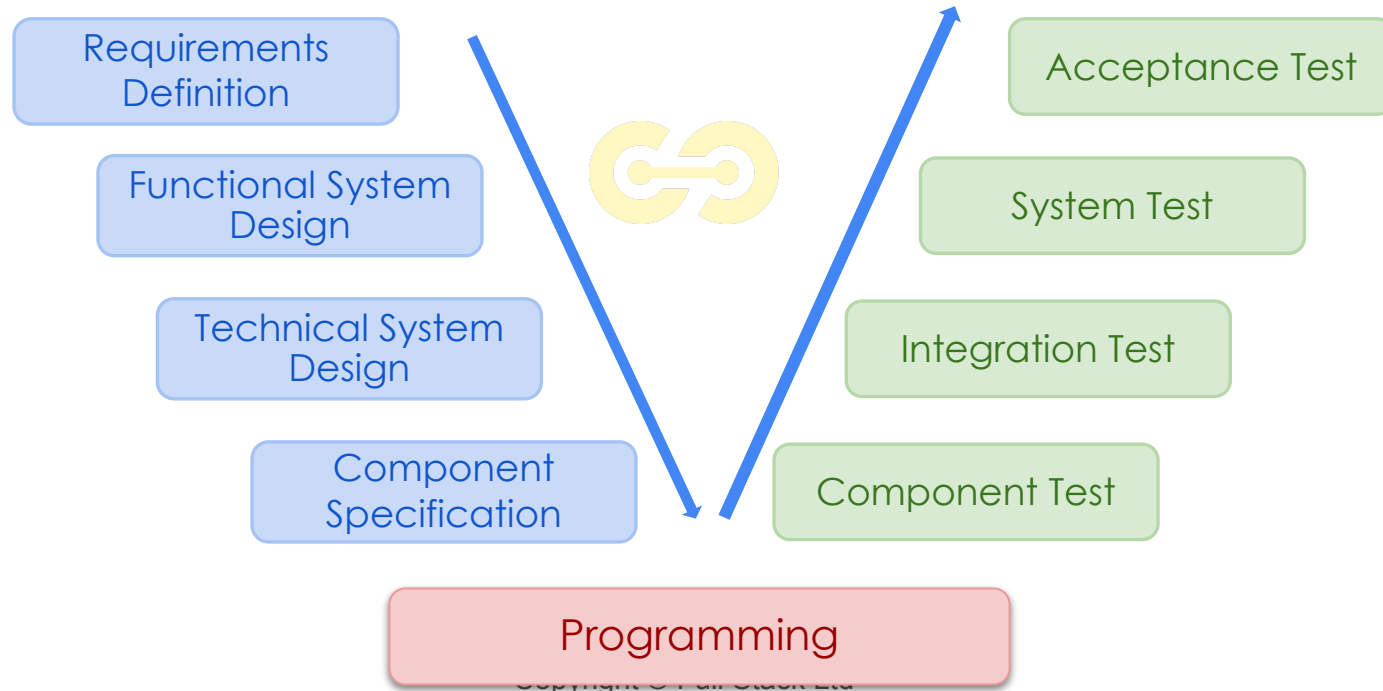


Typical benefits for testing include:

- Increased test effectiveness / efficiency
- Increased quality of testware (e.g., by jointly reviewing the test processes)
- Team bonding and learning (e.g., as a result of the opportunity to raise issues and propose improvement points)
- Improved quality of the test basis
- Better cooperation between development and testing

Testing along the general V-Model

- Development and test are two equal branches
- Each development level has a corresponding test level.
- Testing activities take place through the complete software life cycle



Software development brace

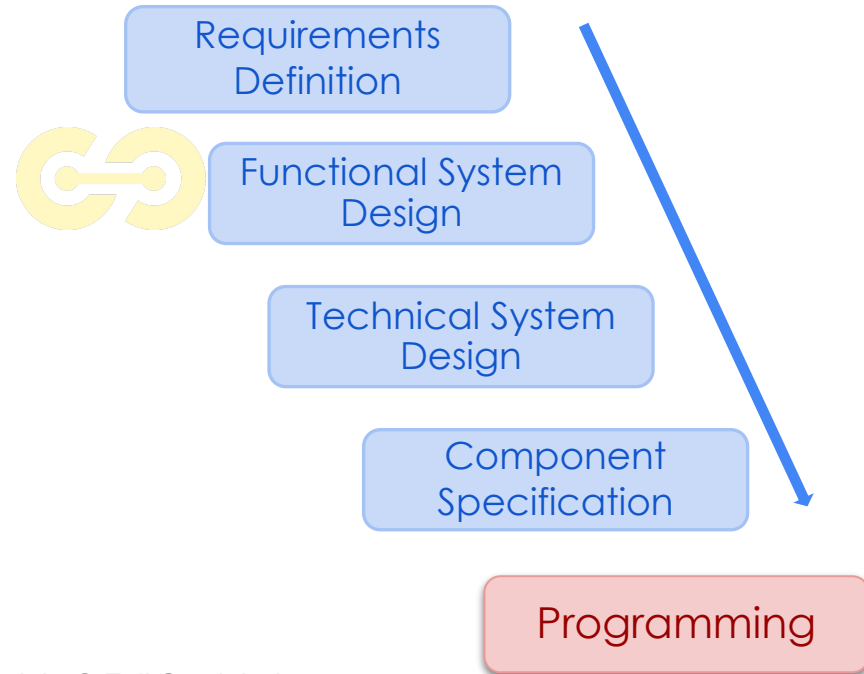
Requirements Definition: SRS

Functional System Design: design functional program flow

Technical System Design: design architecture / interfaces

Component Specification: structure of component

Programming: create executable code



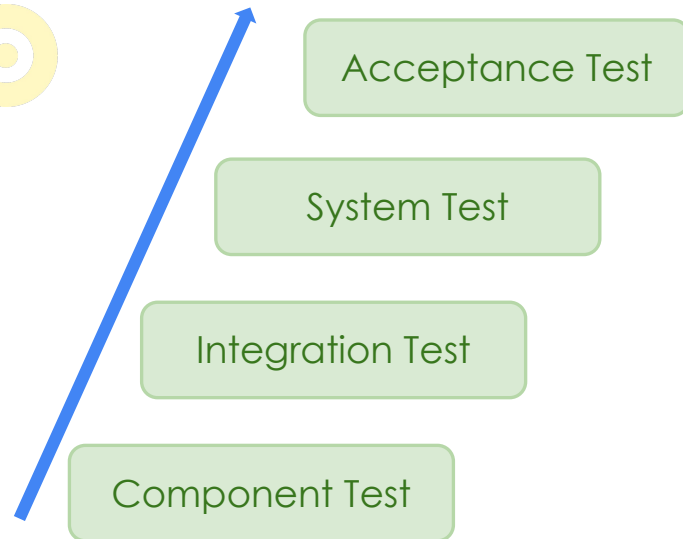
Software test brace

Component Test: Component's internal functionality

Integration Test: Interface between components

System Test: Integrated system, specifications

Acceptance Test: formal test customer requirements



Verification Vs Validation

Verification

- Proof of compliance with the stated requirements (Definition after ISO 9000)
- Main issue: Did we proceed correctly when building the system? Did we add 1 and 1 correctly?

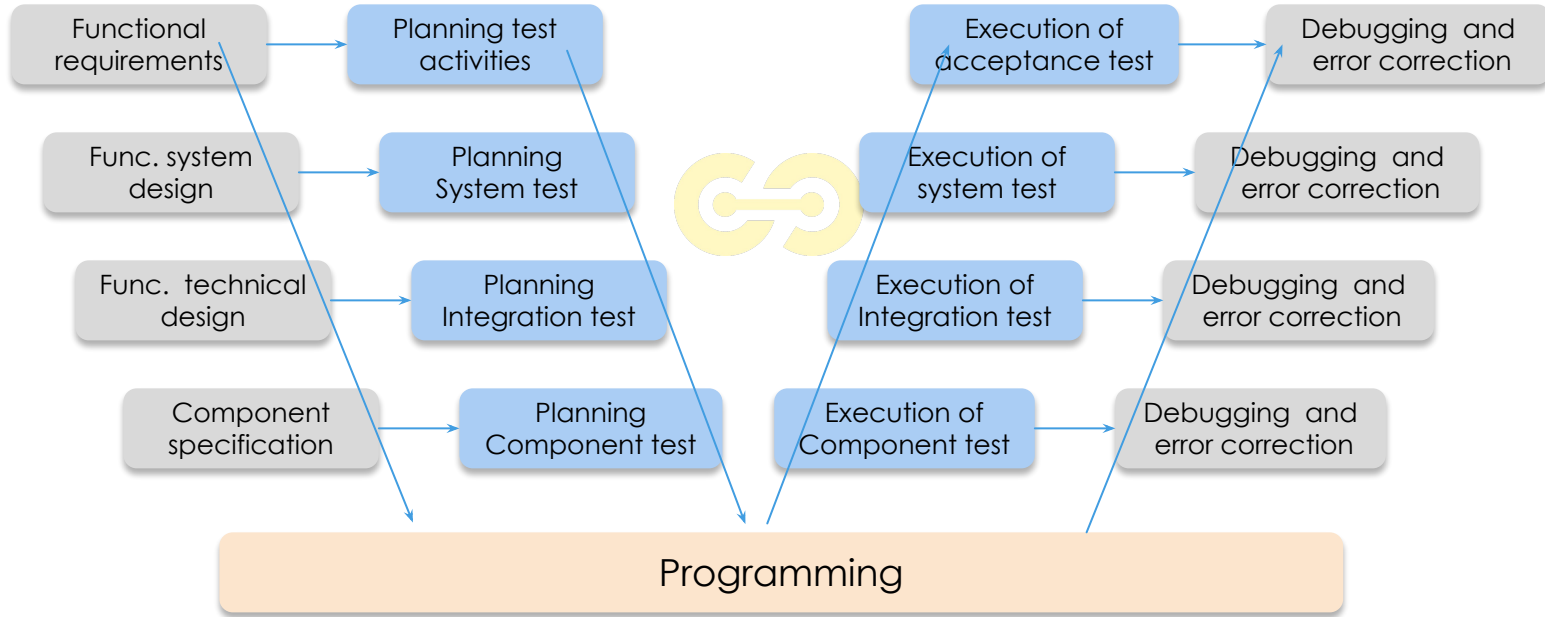


Validation

- Proof of fitness for expected use (Definition after ISO 9000)
- Main issue: Did we build the right software system? was it the matter to add 1 and 1 or should we have subtracted?

Testing within the W-model

- The W-model can be seen as an extension to the general V model.
- The W-model states, that certain quality assurance activities shall be performed in parallel with the development process.



2.2. Test Levels and Test Types

2.2.1. Test Levels

2.2.2. Test Types

2.2.3. Confirmation Testing and Regression Testing



2.2.1. Test Levels

Acceptance Test

System Test

Integration Test

Component Test

Component Test

Component Testing

Definition

Test of each software component after its realization

Because of the naming of components in different programming languages, the component test may be referred to as:

- **module test** (e.g. in C)
- **class test** (e.g. in OOP)
- **unit test** (e.g. in Pascal)



The components are referred to as modules, classes or units.

Because of the possible involvement of developers in the test execution, they are called **developer's test**

Component testing: Scope

- Only **single components** are tested
 - components may consists of several smaller units
 - test objects often cannot be tested stand alone
- **Every** component is tested **on its own**
 - finding failures caused by internal defects
 - cross effects between components are not within the scope of this test
- **Test cases** may be **derived** from
 - component specifications
 - software design
 - data model

Component testing: Functional / non functional testing

-Testing Functionality

- **Every function** must be **tested** with at least one test case
 - are the functions working **correctly**, are all specifications met?
- **Defect** found commonly are:
 - defects in **processing data**, often near **boundary values**
 - **missing functions**
- **Testing robustness** (resistance to invalid input data)
 - Test case representing invalid inputs are called **negative test**
 - A robust system provides an appropriate handling of **wrong inputs**
 - wrong inputs accepted in the system may produce failure in further processing (wrong output, system crash)
- Other **non functional** attributes may be tested
 - e.g. performance and stress testing, reliability

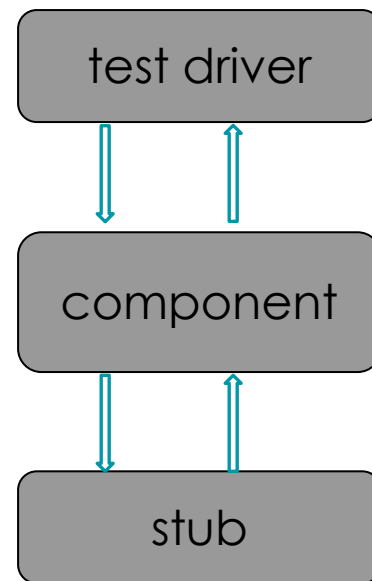
Component testing: Test harness

Test execution of components often requires drivers and stubs

- Drivers handle the interface to the component
 - drivers simulate inputs, record outputs and provide a test harness
 - drivers use programming tools
- Stubs replace or simulate components not yet available or not part of the test object

To program drivers and/or stubs you

- must have programming skills
- must to have the source code available
- may need special tools



Component testing: Methods

- The program code is available to the tester in that case “tester = developer”:
- testing take place with a strong development focus
 - knowledge about functionality, component structure and variable may be applied to design test case
 - often functional testing will apply additionally, the use of debuggers and other development tools
 - e.g. unit test frameworks) will allow to directly access program variables
- Source code knowledge allows to use white box methods for component test

Summary: Component testing

- A component is the smallest system unit specified.
- Module, unit, class and developer's test are used as synonyms.
- Drivers will execute the component functions and adjacent functions that are replaced by stubs.
- Component test may check functional and non functional system properties.

Integration test


Integration testing (also: interface testing)

- Integration is the activity of combining individual software components into a larger subsystems
- Examine the **interaction** of software elements (components) after integration
- **Further** integration of **subsystems** is also part of the system integration process.
- Each component has already been tested for its **internal functionality** (component test).
- Integration test examine the **external functions**.
- May be performed by **developers, testers** both

Integration testing: Scope /1

- Integration tests examine the **interaction** of software components (subsystems) with each other
 - interfaces with the other **components**
 - interfaces among **GUIs / MMIs**
- Integration tests examine the interfaces with the **system environment**.
 - In most cases, the interaction testes is that of the component and **simulated environment** behavior.
 - Under **real conditions**, additional environmental factors may **influence** the components behavior
- **Test case** may be derived from, **interface specifications**, architectural design or data models.

Integration testing: Scope/2

- **A (Sub-) system**, composed of individual components, will be tested.
 - Each component has as interface either external and / or interacting with another component within the (sub-) system.
- **Test drivers** (which provide the process environment of the system or subsystem) are required to 
 - allow for or to produce input and output of the (sub-) system
 - log data
- Test drivers of the components tests may be re-used here.

Integration testing: Scope/3

- ❑ **Monitoring tools** logging data and controlling testing activities
- ❑ **Stubs** replace missing components
 - data or functionality of a component that have not yet been integrated will be replaced by programmed stubs
 - stubs take over the elementary of the missing components

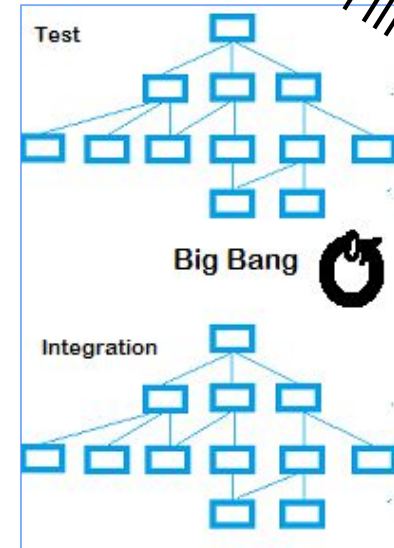
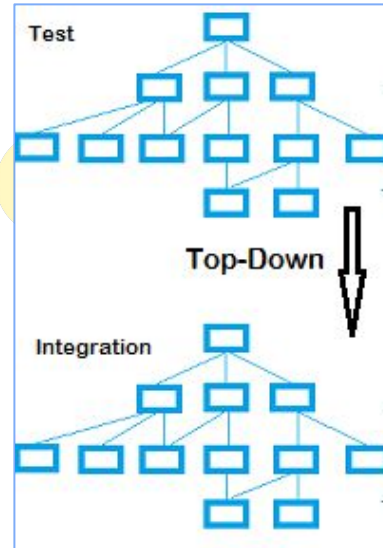
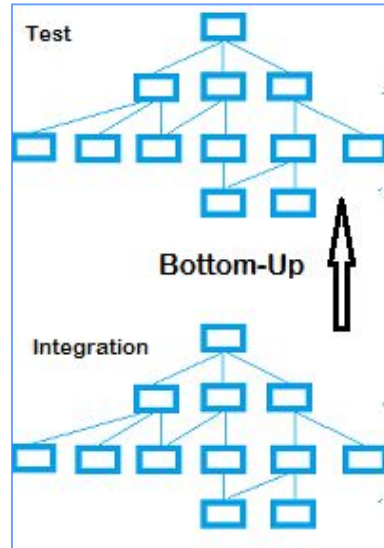
Integration testing: Approach

- Integration tests aim at finding defects in the interface. They check the correct interaction of components
 - among other reason, in order to check performance and security aspects
- Replacing test drivers and stubs with **real components** may produce new defects, such as
 - **Losing data**, wrong handling of data or wrong inputs
 - The components involved interpret the input data in a different manner
 - The **point in time** where data is handed over is not correct: too early, too late, at a wrong frequency

Integration testing: Strategies /1

- There are different strategies for integration testing
 - Common to most strategies is the **incremental** approach (exception for example “Big Bang” strategy)
 - **Bottom-up** and **top-down** are the most commonly used strategies
- Choosing a **strategy** must also consider aspects of the test efficiency
 - The integration strategy determines the amount of test effort needed (e.g. using tools, programming test drivers and stubs etc)
 - Component **completion** determines for all types of integration strategies, at what time frame the component is available. Therefore **development strategy** influences the integration strategy.
- For each individual project, the trade-off between reducing time and reducing test effort must be considered:
 - testing what is ready: more costs for testing but less idle time
 - follow a strict integration test plan: lower cost but more idle time

Integration testing: strategies



Terminology

Integration testing: strategies /5

Ad-hoc integration

- components will be tested, if possible, direct after programming and component test have been completed

Characteristics of ad-hoc integration

- early start of testing activities, possibly allowing for a shorter software development process as a whole
- depending on the type of component completed, stubs as well as test drivers will be needed

Use of ad-hoc integration

- It is a strategy that can be followed at any stage in the project
- It is often used combined with other test strategies.

Summary: Integration testing

- **Integration** means building up groups of component
- **Integration tests** examine component interactions against the specification of interfaces
- Integration takes place either **bottom-up**, **top-down**, or in a **big bang**
- **Integration Sub-systems** (they consists of integrated components) is also a form of integration
- A further integration strategy is **ad-hoc** integration

System test

System test

- System testing is a level of testing that validates the complete and fully integrated software product.
- The purpose of this test is to evaluate the system's compliance with the specified requirements.
- Software quality is looked at from the user's point of view
- System tests refer to:
 - **functional** and **non-functional** requirements (functionality, reliability)
- **Test cases** may be derived from
 - Functional specifications
 - Use cases
 - Business processes
 - Risk assessments

System test: Scope

- Test of the integrated system from the user's point of view
 - Complete and correct implementation of requirements
 - Development in the real system environment with real life data
- The **test environment** should match the **true environment**
 - No test drivers or stubs are needed
 - All external interfaces are tested under true conditions
 - Close representation of the later true environment
- No test in the real life environment
 - Introduced defects could damage the real life environment
 - Software under development is constantly changing. Most tests will not be reproducible

System test: functional requirements /1

- Goal: to prove that the implemented functionality exposes the required characteristics
- Characteristics to be tested include (as per ISO 9126):
 - **Suitability**
 - Are the implemented functions suitable for their expected use
 - **Accuracy**
 - Do the functions produce correct (agreed upon) result?
 - **Interoperability**
 - Does interaction with the system environment show any problem?
 - **Compliance**
 - Does the system comply with applicable norms access or less?

System test: functional requirements /2

- Three approaches for testing functional requirements:
 - Business process based test**
 - each business process service as basis for driving tests
 - the ranking order of the business process can be applied for prioritizing test cases
 - Use case based test :**
 - Test cases are derived from sequences of expected or reasonable use
 - sequence used more frequently receive a higher priority
 - Requirements based test (building blocks)**
 - Test cases are derived from the requirement specification
 - The number of the cases will vary accounting to the type/depth of specification
- Requirements based test

System test: Non-functional requirements

- **Compliance** with non functional requirements is **difficult to achieve**:
 - Their definition is often very vague (e.g. easy to operate, well structured user interface, etc.)
 - They are not stated explicitly. They are an implicit part of system description, still they are expected to be fulfilled
 - **Quantifying** them is **difficult**, often non-objective metrics must be used, e.g. looks pretty, quite safe, easy to learn.
- Example: Testing / inspiring documentation
 - Is documentation of programs in live with the actual system, is it concise, complete and easy to understand?
- Example: Testing maintainability
 - Have all programmers complied with the respective Coding-Standards?
 - Is the system designed in a structured, modular fashion?

Summary: System test

- System testing is performed using functional and non-functional test cases
- Functional system testing confirms that the requirements for a specific intended use have been fulfilled (validation)
- Non functional system testing verifies non functional quality attributes, e.g. usability, efficiency, portability etc.
- Non functional quality attributes are often as implicit part of the requirements, this makes it difficult to validate them

Acceptance test

Objectives of acceptance testing

Acceptance testing, like system testing, typically focuses on the behavior 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 behaviors of the system are as specified

so that it can be accepted by the customer.

- ❖ Acceptance testing may produce information to assess the system's readiness for deployment and use by the customer.
- ❖ 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.
- ❖ Acceptance testing may also satisfy legal or regulatory requirements or standards.

Common forms of acceptance testing include the following:

- Contractual and regulatory acceptance testing
- Operational acceptance testing
- Alpha and beta testing (for COTS)

*COTS= commercial off the shelf

Contractual and regulatory acceptance testing

- **The main objective** is building confidence that contractual or regulatory compliance has been achieved.
- **Contractual acceptance testing**
 - performed against a contract's acceptance criteria for producing custom-developed software.
 - Acceptance criteria should be defined when the parties agree to the contract.
 - often performed by users / independent testers.
 - All the **contractual** requirements are fulfilled?
- **Regulatory acceptance testing**
 - performed against any regulations that must be adhered to, such as government, legal, or safety regulations.
 - often performed by users / independent testers, sometimes with the results being witnessed or audited by regulatory agencies.
- With formal acceptance **legal milestone** are reached: begin of **warranty, payment milestones, maintenance agreements**, etc.
- Often **customer** select **test case** for acceptance testing
 - Possible misinterpretations of the requirements come to light and can be discussed, "The customer knows best"
- Testing is done using the **customer environment**
 - Customer environment may cause new failures

Acceptance testing: operational acceptance testing

- ***The main objective*** of operational acceptance testing is building confidence that the operators or system administrators can keep the system working properly for the users in the operational environment, even under exceptional or difficult conditions.
- The tests focus on operational aspects, and may include:
 - Testing of backup and restore
 - Installing, uninstalling and upgrading
 - Disaster recovery
 - User management
 - Maintenance tasks
 - Data load and migration tasks
 - Checks for security vulnerabilities
 - Performance testing
 - Compatibility with other systems (hardware, OS, database, etc.)
- Operational acceptance testing is often done by the customer's system administrator

Acceptance testing: alpha- and beta (or filed) testing

- Alpha and beta testing are typically used by developers of commercial off-the-shelf (COTS) software who want to get feedback from potential or existing users, customers, and/or operators before the software product is put on the market.
- **Alpha testing** is performed at the developing organization's site, not by the development team, but by potential or existing customers, and/or operators or an independent test team.
- **Beta testing** is performed by potential or existing customers, and/or operators at their own locations.
- Beta testing may come after alpha testing
- **Objective of alpha and beta testing** -
 - building confidence among potential or existing customers, and/or operators that they can use the system under normal, everyday conditions, and in the operational environment(s) to achieve their objectives with minimum difficulty, cost, and risk.
 - may be the detection of defects related to the conditions and environment(s) in which the system will be used, especially when those conditions and environment(s) are difficult to replicate by the development team.

Acceptance testing: Test basis /1

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

Acceptance testing: Test basis /2

In addition, as a test basis for deriving test cases for operational acceptance testing, one or more of the following work products can be used:

- Backup and restore procedures
- Disaster recovery procedures
- Non-functional requirements
- Operations documentation
- Deployment and installation instructions
- Performance targets
- Database packages
- Security standards or regulations



Acceptance testing: 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

Acceptance testing: 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

Acceptance testing: Specific approaches and responsibilities

- Acceptance testing is often thought of as the last test level in a ***sequential development lifecycle***, but it may also occur at other times, for example:
 - A COTS software product may occur when it is installed or integrated
 - A new functional enhancement may occur before system testing
- In ***iterative development***-
 - project teams can employ various forms of acceptance testing during and at the end of each iteration.
 - Alpha tests and beta tests may occur, either at the end of each iteration, after the completion of each iteration, or after a series of iterations.
 - User acceptance tests, operational acceptance tests, regulatory acceptance tests, and contractual acceptance tests also may occur, either at the close of each iteration, after the completion of each iteration, or after a series of iterations.
- 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.

Summary: Acceptance testing

- Acceptance testing is the **customer's** system test
- Acceptance testing is **contractual** activity, the software will then be verified to comply with customers requirements
- **Alpha-** and **beta** tests are tests performed by potential or existing customer either at the developer's site (alpha) or at the customers site (beta).

2.2.2. 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 (**White-Box**)
- Evaluating the **effects of changes**, such as confirming that defects have been fixed (**confirmation testing**) and looking for unintended changes in behavior 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
- Functional testing considers the behavior 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.

Non-Functional Testing

- **Goal: software product characteristics**
 - **How well** does the software perform its functions?
 - The non-functional quality characteristics (reliability, usability, efficiency, maintainability, portability) are often vague, incomplete or missing all together, making testing difficult
- **Area of use**
 - Non-Functional testing may be performed at all test levels
 - Typical non-functional testing:
 - Load testing/ performance testing/ volume testing/ stress testing
 - Testing of safety features
 - Reliability and robustness testing / compatibility testing
 - Usability testing / configuration testing
- 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.

Types of Testing

Non Functional Testing (system test)

- **Load test**

- System under load (minimum load, more user/tractions)

- **Performance test**

- How fast does the system performed a certain function

- **Volume test**

- Processing huge volumes of data / files

- **Stress test**

- Reaction to overload / recovery after return to normal

- **Reliability test**

- Performance while in “continuous operation mode”

- **Test of robustness**

- Reaction to input of wrong or unspecified data
 - Reaction to hardware failures / disaster recovery

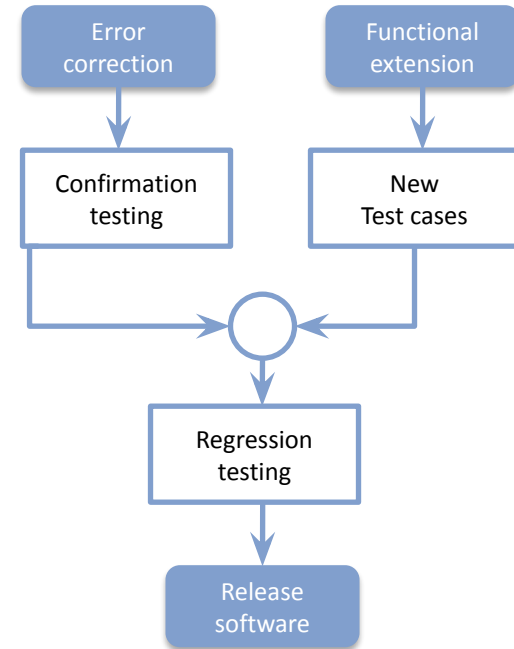
White-box Testing

- **Goal: Coverage**
 - Analyses the structure of the test object (**while box** approach)
 - Testing aims at measuring how well the structure of the test object is covered by the test cases
- **Area of use**
 - Structural testing possible on all test levels, **code coverage** testing using tools mainly done during **component** and integration testing
 - Structural test design is finalized after functional tests have been designed, aiming at producing a high degree of coverage
- **Execution**
 - Will test the internal structure of a test object (e.g. control flow within components, flow through a menu structure)
 - Goal: all identified structural elements should be covered by test cases

Change-related Testing /1

Confirmation /Regression testing

- **Goal: test object after changes**
 - After a test object or its system Environment has been **changed**, results Related to the change have become Invalid: **test** have to be repeated
 - Two main reasons for changing Software
 - Error correction
 - functional extension
- Because of undesired side effects of extended or **new functionality**, it is necessary to also retest adjacent areas!



Change-related Testing /2

Confirmation /Regression testing /2

- **Area of use**
 - Replacing a test of functionality that has already been verified is called a regression test.
 - The scope of the regression test depends on the risk, that the newly implemented functionality (extension or error fix) imposes to the system.
 - Analyzing this risk can be done with an impact analysis
 - Confirmation / Regression testing may be performed at all test levels.
 - Typical test after changes are:
 - Confirmation testing , retest (= Testing after correction of errors)
 - Regression testing (= Testing to uncover newly introduced defects)

Change-related Testing /2

Regression testing should be performed in the following circumstances:

1. **After code changes:** Whenever changes are made to the code, regression testing should be performed to ensure that the changes have not negatively impacted existing functionality.
2. **After bug fixes:** When a bug is fixed, regression testing should be performed to verify that the fix has not introduced any new bugs or broken any existing functionality.
3. **Before a new release:** Before a new release of the software is made, regression testing should be performed to ensure that the changes made in the new release have not impacted existing functionality.
4. **When adding new features:** When new features are added to the software, regression testing should be performed to verify that the new features do not impact existing functionality.
5. **When updating dependencies:** Whenever dependencies are updated, regression testing should be performed to ensure that the updates have not impacted existing functionality.

In summary, regression testing should be performed regularly throughout the software development life cycle, especially after significant changes have been made to the code, to ensure that the software remains stable and high-quality.

Change-related Testing /3

Confirmation /Regression testing /3

- **Execution**

- Basically, execution takes place as in previously executed test iterations
- In most cases, a complete regression test is not feasible, because it is too expensive and takes too much time
- A high degree of modularity in the software allows for more appropriate **reduced** regression tests
- Criteria for the selection of the regression test cases:
 - Test case with high priority
 - Only test standard functionality, skip special cases and variations
 - Only test configuration that is used most often
 - Only test subsystem / selected areas of the test object
- If during early project phases, it becomes obvious that certain tests are suitable for regression testing, test automation should be considered

Summary

- On different **test levels** different **types of tests** are used
- Test types are: **functional**, **non-functional**, **structural** and **change related testing**
- **Functional testing** examines the input / output **behavior** of a test object
- **Non- functional** testing checks **product characteristics**
- **Non- functional testing** include, but is not limited to, **load testing**, stress testing, performance testing, robustness testing
- Common **structural tests** are tests that check data and control flow within the test object, measuring the degree of coverage
- Important test after changes are: **confirmation tests(re-tests)** and **regression tests**

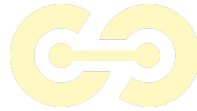
2.3.Maintenance Testing

Testing after Product Acceptance /1

- **Customer** has **approved** the product and sets it into production
 - The initial development cycle, including its related **tests**, has been **completed**
- The **software** itself is at the beginning of its life cycle:
 - it will be used for many years to come, it will be **extended**
 - it most likely still have errors, hence it will be further modified and **corrected**
 - it need to adapt to new conditions and to be integrated into new environments
 - it will one day be retired, put out of operation
- **Any new version of the product, any new update and any other change in the software requires additional testing!**

Testing after Product Acceptance /2

- Software maintenance covers two different fields:
 - **maintenance** as such: correction of error, that already were part of the initial version of the software
 - **software extension**: adaptations as a result of a changed environment or new customer requirements
- **Test scope** of maintenance testing
 - Error correction requires retests
 - Extended functionality requires new test case
 - Migration to another platform requires operational tests
 - In **addition**, intensive **regression testing** is needed



Testing after Product Acceptance /2

- **Scope of testing** is affected by the impact of the change
 - **Impact analysis** is used to determine the affected areas
 - Problems might occur if **documentation** of the old software is **missing** or **incomplete**



- **Software retirement**
 - Test after software retirement may include
 - Data migration test
 - Verifying archiving data and programs
 - parallel testing of old and new systems

Summary

- Ready developed software needs to be **adapted** to new conditions, errors have to be **corrected**
- An **impact analysis** can help to judge the changes related risks
- Maintenance tests make sure , that
 - New function are implemented correctly (**new test cases**)
 - Error have been fixed successfully (**old test cases**)
 - Functionality, that has already been verified, is not affected (**regression test**)
- If software gets **retired**, migration tests or parallel tests may be necessary



Thank You