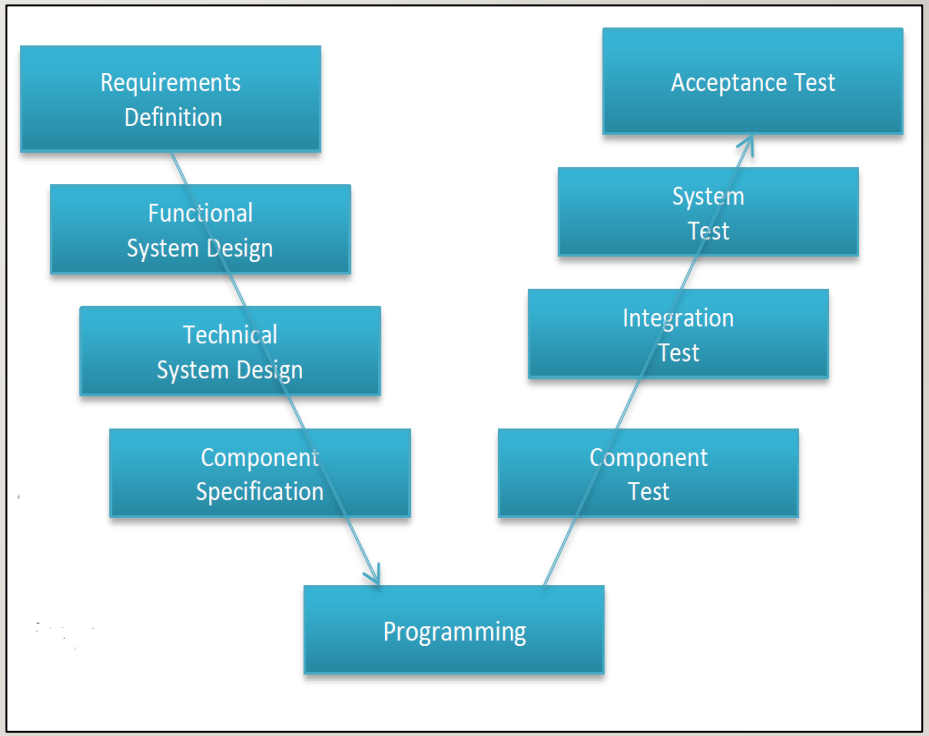
TESTING THROUGHOUT THE SOFTWARE LIFE CYCLE (I)

COMPILED BY –

NAZMUS SAKIB AKASH

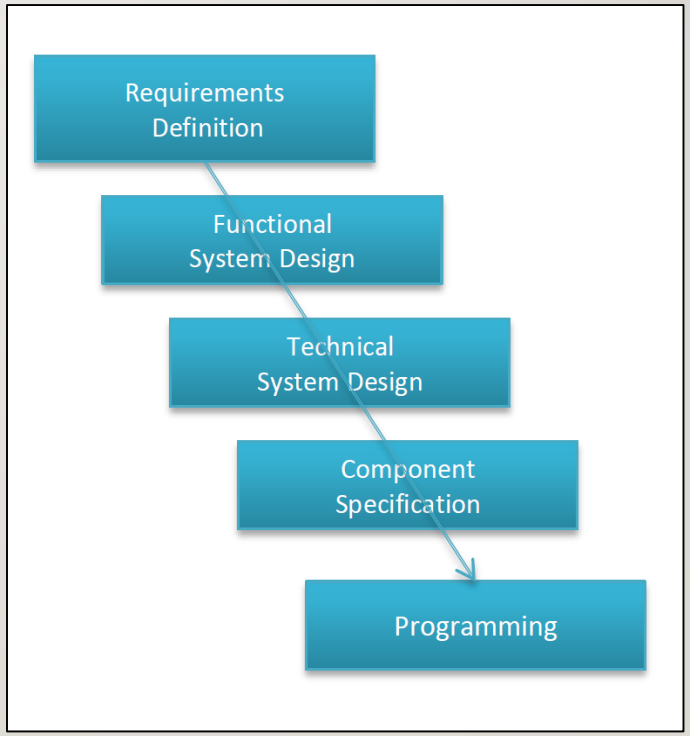
2 SOFTWARE DEVELOPMENT MODELS: **TESTING ALONG THE GENERAL V-MODEL**

- Testing along the general V-Model
  - The general V-Model is most commonly used software development model.
- Development and Test are two equal branches
  - each development level has a corresponding test level.
- Test (right hand side) are designed in parallel with software development (left hand side)
- Testing activities take place through the complete software life cycle



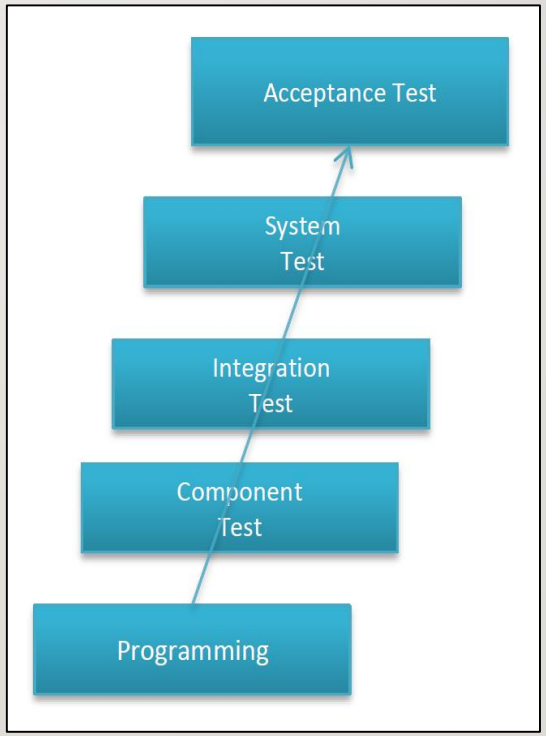
3 TESTING ALONG THE GENERAL V-MODEL **SOFTWARE DEVELOPMENT BRACE**

- Software development brace
  - Requirements Definition
    - specification documents
  - Functional System Design
    - design functional program flow
  - Technical System Design
    - design architecture / interfaces
  - Component Specification
    - structure of component
  - Programming
    - create executable code



4 TESTING ALONG THE GENERAL V-MODEL **SOFTWARE TEST BRACE**

- Software test brace
  - Acceptance test
    - formal test customer requirements
  - System test
    - integration system, specifications
  - Component's functionality
    - component's functionality



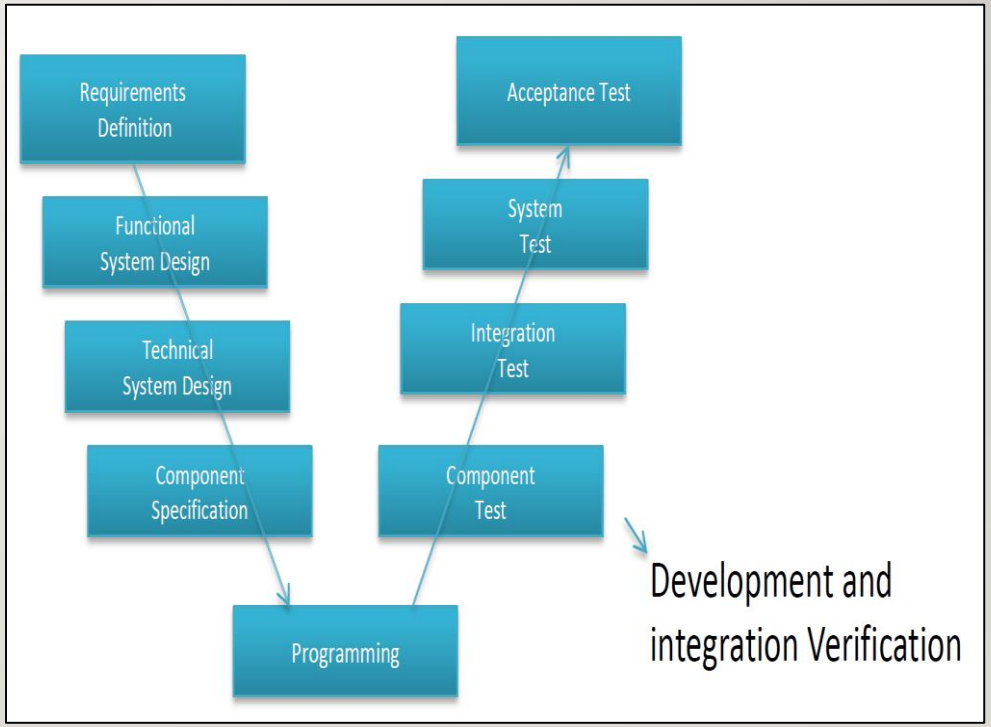
5 VERIFICATION VS VALIDATION

- **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?

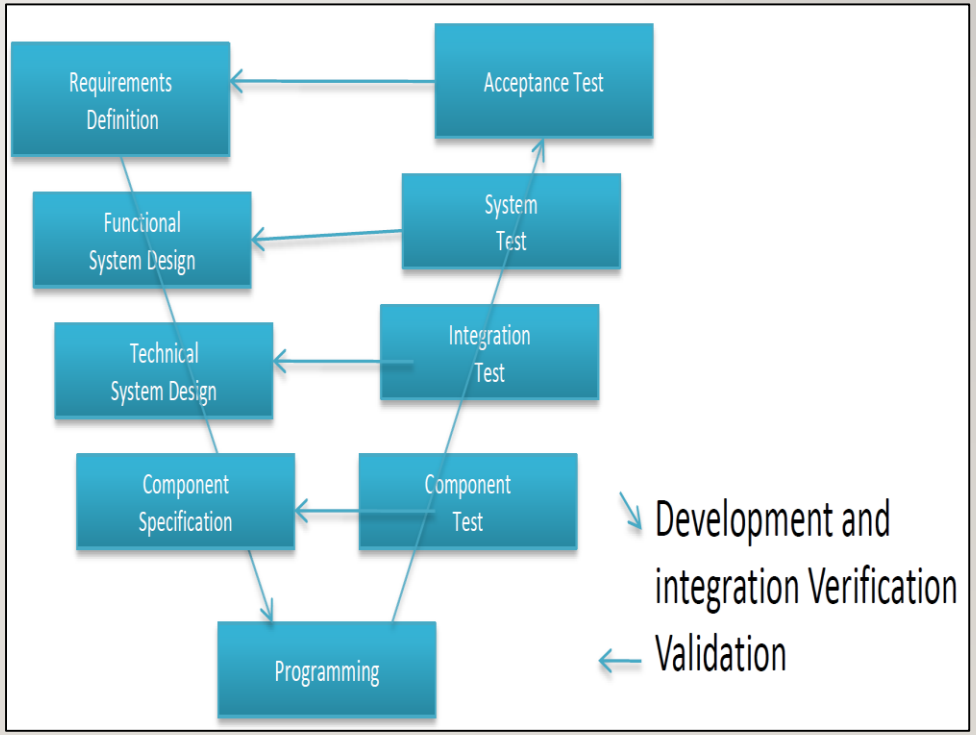
6 VERIFICATION WITHIN THE GENERAL V-MODEL

- Each development level is verified against the contents of the level above it
  - to verify: to give proof of evidence, to substantiate
- To verify means to check whether the requirements and definitions of the previous level were implemented correctly



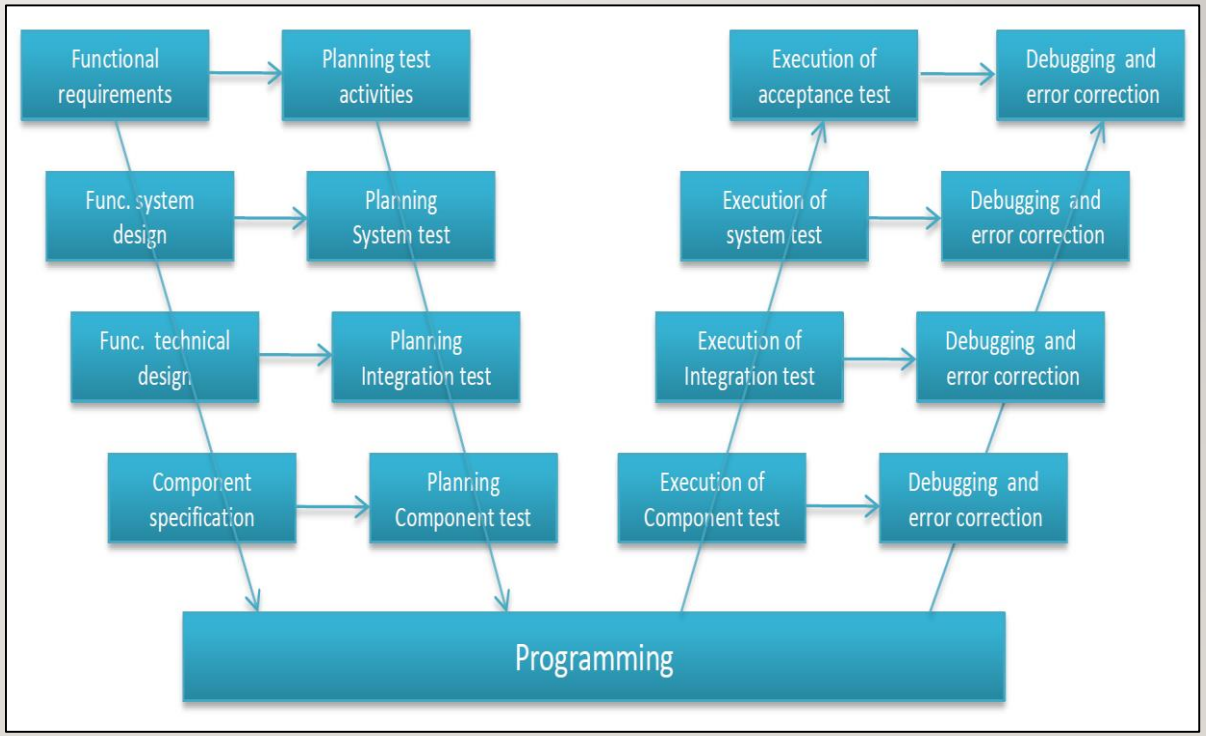
7 VALIDATION WITHIN THE GENERAL V-MODEL

- Validation refers to the correctness of each development level
  - to validate refers to the correctness of each development level
    - to validate: to give proof of having value
- To validate means to check the appropriateness of the results of one development level



8 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



ITERATION MODELS: TYPES OF ITERATION MODELS

- Iteration software development
  - The activities: requirement definition, design, development, testing are divided into small steps and run continuously.
  - In order to redirect the progress if necessary, a consent must be reached with the customer after each iteration.
- **Iteration models are for example**
  - **Prototyping**: Building quickly usable representation of the system, followed by successive modification until the system is ready.
  - **Rapid Application Development** (RAD): The user interface is implemented using out of-the box functionality faking the functionality which will be later developed.
  - **Rational Unified Process** (RUP): Object oriented model and a product of the company Rational/IBM. It mainly provides the modeling language UML and support for the Unified Process.
  - **Extreme Programming** (XP): Development and testing take place without formalized requirements specification.

ITERATION MODELS: CHARACTERISTICS

- Characteristics of iteration model:
  - Each iteration contributes with an **additional characteristic** of the system under development
  - Each iteration can be tested separately
  - **Regression tests** and **test automation** are of high relevance
  - at each iteration, verification(relation to preceding level)

  And validation (correctness of the product within the current level) can be performed separately.

ITERATION MODELS:TEST DRIVEN DEVELOPMENT (TDD)

- Based on: test cycles
  - Prepare **test cycles**
  - **Automated** testing using test tools
- Development accounting to test cases
  - Prepare early versions of the **component** for **testing**
  - **Automatic execution** of tests
  - **Correct defects** on further version
  - **Repeat** test suite until no errors are found

First the **tests** are designed, **then** the software is **programmed**

12 PRINCIPLES OF ALL MODELS

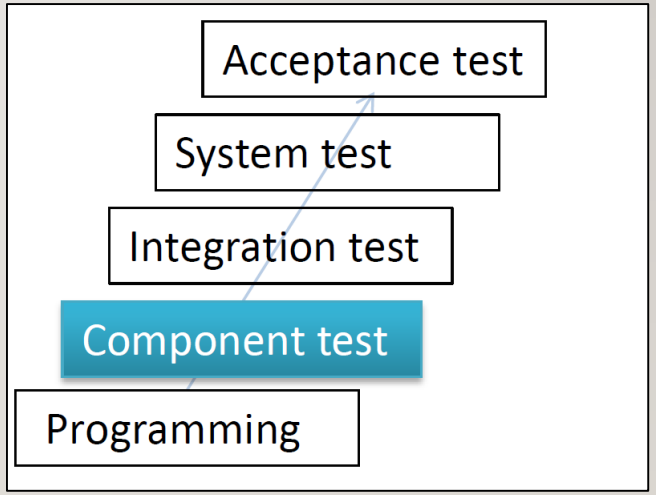
- Each development **activity** must be **tested**
  - No piece of software may be left untested, whenever it was developed "in one procedure" or iteratively
- Each test level should be tested specifically
  - each **test level** has its own test objectives
  - the test performed at each level reflect these objectives
- **Testing begins** long **before test execution**
  - as soon as development begins, the preparation of the corresponding tests cab start
  - this is also the case for document reviews starting with concepts, specification and overall design

13 SUMMARY

- Software development models are used for **software development** including test activities.
- The best known model is the **V-model**, which describes development levels and test levels as two related branches.
- The most relevant **iterative modes** are RUP, XP and SCRUM.
- **Test activities** are recommended at all development levels.

TEST LEVELS: DEFINITIONS

- **Component Testing**
  - 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 java or C++)
  - **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

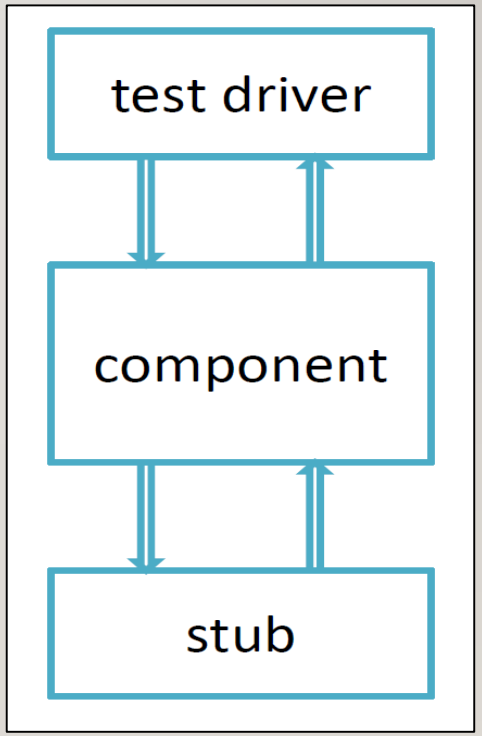
- Only **single components** are tested
  - components may consists of several smaller units
  - test objects often cannot be tested stand alone
- **Every** component is tested **on its own**
  - finding failures caused by internal defects
  - cross effects between components are not within the scope 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

17 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

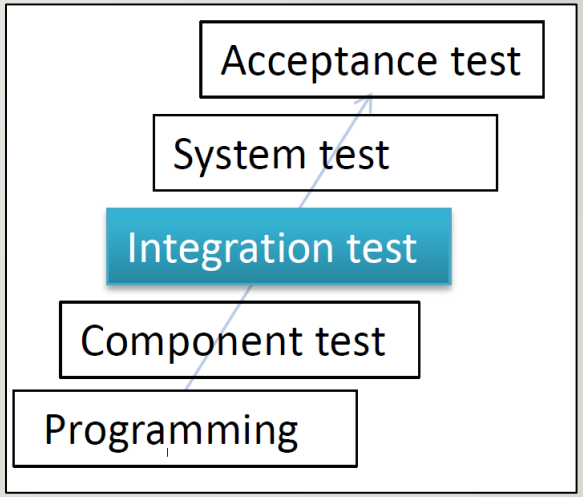
- The program code is available to the tester
  - in case "tested = **developer**":
    - testing take place with a strong **development focus**
  - knowledge about **functionality**, **component structure** and **variable** may be applied to **design test case**
  - often functional testing will apply
    - 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 TESTING (ALSO: INTERFACE TESTING)

- Examine the **interaction** of software elements (components) after system integration.
- Integration is the activity of combining individual software components into a **larger subsystems**.
- **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 (I)

- Integration tests assume that the components have **already been tested**.
- 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 tests 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 (II)

- **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 sub-system) are required
  - to allow for or to produce input and output of the (sub-) system
  - to log data
- Test drivers of the components tests may be re-used here

INTEGRATION TESTING: SCOPE (III)

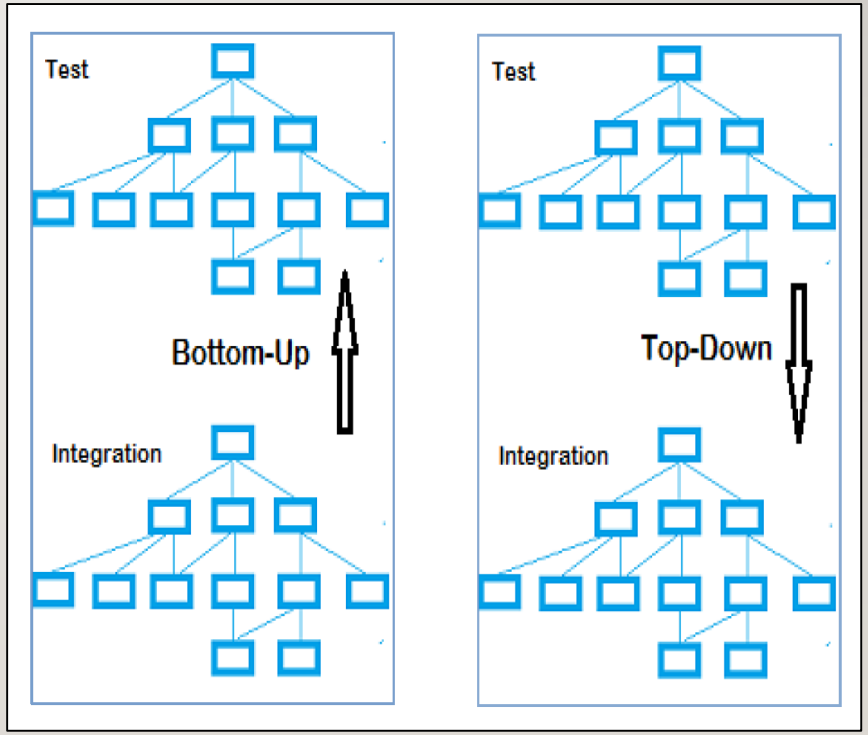
- **Monitoring tools** logging data and controlling test van support testing activities
- **Stubs** replace missing components
  - data or functionality of a component that have not yet been integrated will be replaced by programmed stubs
  - stubs take over the elementary of the missing components

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

25 INTEGRATION TESTING: STRATEGIES (I)

- 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 (II)

- 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

- Testing the integrated software system to prove compliance with the specified requirements
  - Software quality is looked at form the user's point of view
- System tests refer to (as per ISO 9126):
  - **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
  - Induced defects could damage the real life environment
  - Software under development is constantly changing. Most tests will not be reproducible

SYSTEM TEST: FUNCTIONAL REQUIREMENTS (I)

- 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 (II)

- **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 cares 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 / inspiriting 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

34 ACCEPTANCE TESTING

- Acceptance testing is a formal test performed in order to verify compliance of the system with user requirements. The goal is to establish confidence in the system, so it can be accepted by the customer (see: IEEE 610).
- It is often the first test where the customer is involved (it is advisable, to involve the customer already during software development, e.g. for prototyping purposes)
- The customers involvements may very depending on the type of program (individual software or COTS software)
  - **Individual** software is often tested directly by the customer
  - **COTS software may be tested by a selected group of customers.**
    - *[COTS= commercial off the shelf]*

ACCEPTANCE TESTING: CONTRACTUAL ACCEPTANCE

- Does the software fulfill all the **contractual** requirements?
- With formal acceptance legal milestone are reached: begin of **warranty, payment** milestones, **maintenance** agreements, etc.
- **Verifiable acceptance criteria** should be defined when the contract is agreed, this serves is an insurance for both parties
- Governmental, legal, industrial and other regulations have to be taken into account for acceptance testing (e.g. safety regulation FMVSS 208: Federal Motor Vehicle Safety Standard)
- Often **customer**s elect **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

- Requirements that software is fit for use in a productive environment
  - Integration of software into the customer IT-infrastructure (Backup-/Restore-Systems, restart, install and de-install-ability, disaster recovery, etc.)
  - User management, interfacing to file and directory structures in use
  - Compatibility with other systems (other computers, data base servers, etc.)
- Operational acceptance testing is often done by the customer's system administrator

ACCEPTANCE TESTING: ALPHA-AND BETA (OR FILED) TESTING

- A stable preliminary version of the software is needed
- Mostly done for market software (also called COTS* software)
- Customers use the software to process their daily business process at the suppliers location **(beta testing)**
- Feed back is given on problem found, usability, etc.
- Advantages of alpha and beta tests
- Reduce the cost of acceptance testing
- Use different user environments ,
- Involve a high number of users

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).