ST - Module 3

Integration, System and Acceptance Testing

Integration Testing

- It is defined as a type of testing where **software modules** are **integrated logically** and **tested as a group**.
- A typical software project consists of multiple software modules, coded by different programmers.
- The **purpose** of this level of testing is to **expose defects** in the **interaction between** these **software modules** when they are integrated.
- Integration Testing focuses on **checking data communication** amongst these modules.

Example of Integration Testing

- Database scripts, application main code and GUI components were made by 3 different programmers.
- We need test all these 3 components as 1 system. For this we combine them into 1 system and then test the interfaces between the modules, like, interaction between database and GUI.

Example of Integration Test Case

Sample Integration TCs for the following scenario: Application has 3 modules say 'Login Page', 'Mailbox' and 'Delete emails' and each of them is integrated logically:

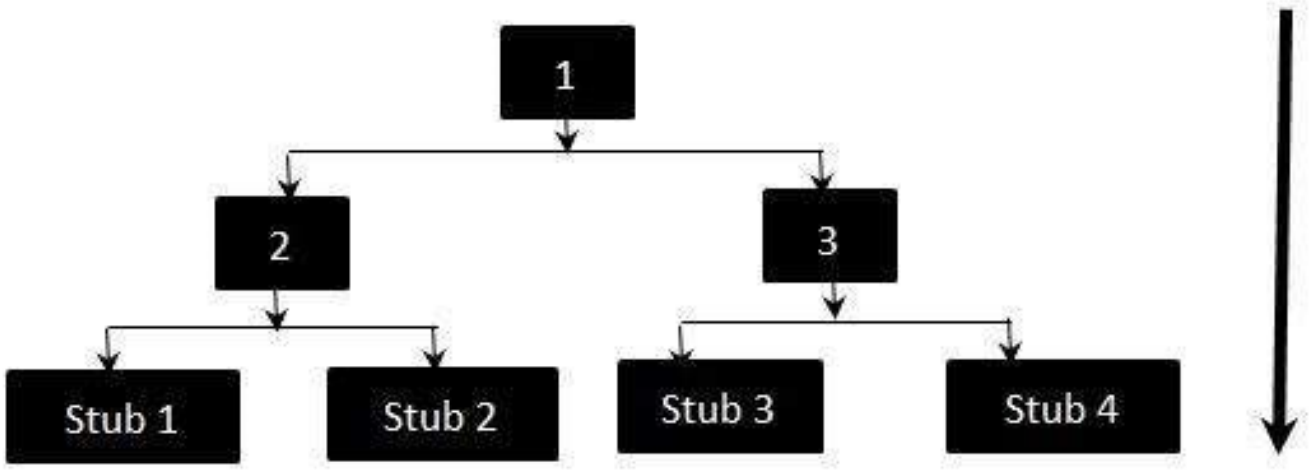
Test Case ID	Test Case Objective	Test Case Description	Expected Result
1	Check the interface link between the Login and Mailbox module	Enter login credentials and click on the Login button	To be directed to the Mail Box
2	Check the interface link between the Mailbox and Delete Mails Module	From Mailbox select the email and click a delete button	Selected email should appear in the Deleted/Trash folder

Integration Testing Techniques

1. Top-down integration
2. Bottom-up integration
3. Bi-directional integration
4. System integration

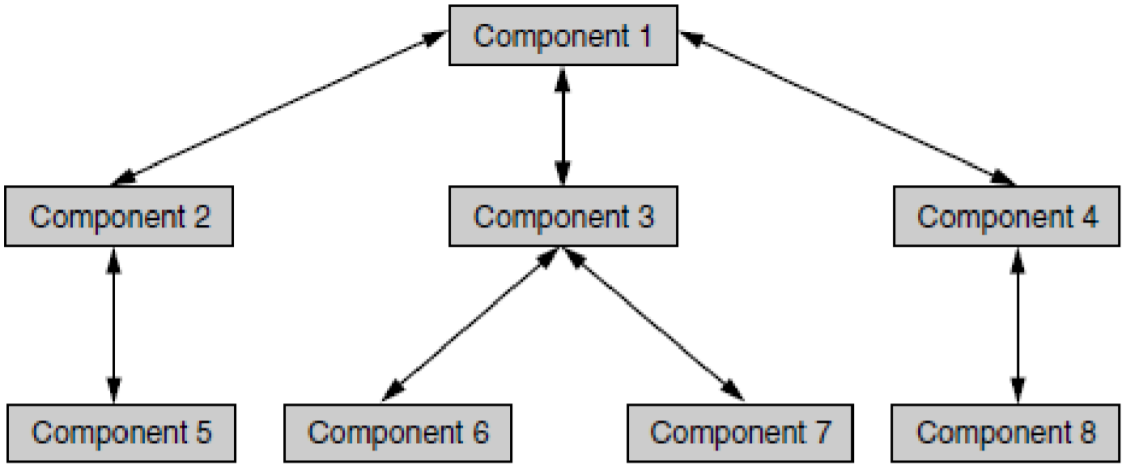
Top-down integration

- Top-Down Integration Testing is a method in which integration testing takes place from **top to bottom** following the **control flow of software system**.
- The **higher-level modules are tested first** and then lower-level modules are tested and integrated in order to check the software functionality.
- **Stubs** are used during Top-down integration testing, in order to **simulate** the **behavior of the lower-level modules** that are not yet integrated.
- Stubs are the modules that act as **temporary replacement** for a called module and give the same output as that of the actual product.
- Stubs are used for testing if some modules are not ready.



- **Advantages**:
  - Fault Localization (detecting fault in software) is easier.
  - Possibility to obtain an early prototype.
  - Critical modules are tested on priority
  - Major design flaws could be found and fixed first.
- **Disadvantages:**
  - Needs many Stubs.
  - Modules at a lower level are tested inadequately.

- Example of Top-down integration:



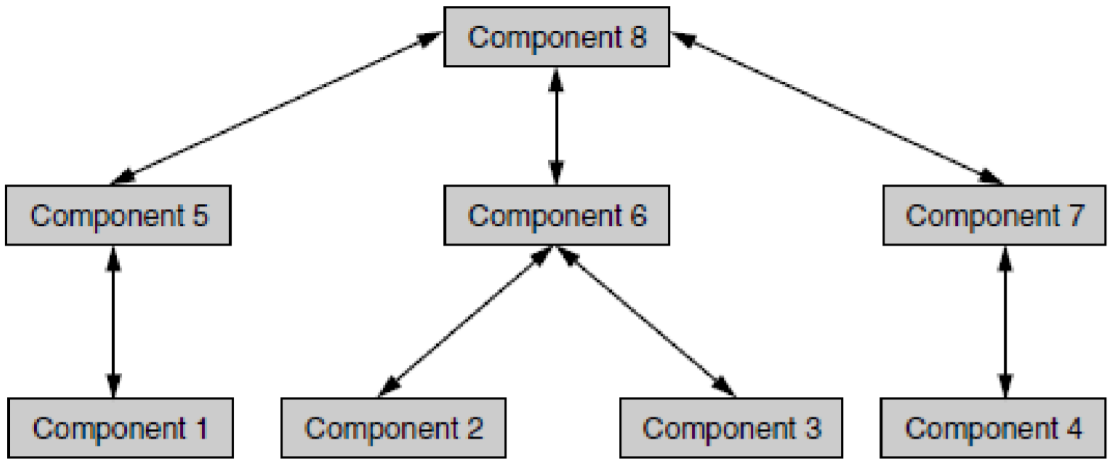
- The order in which the interfaces are to be tested is shown in the following table:

Step	Interfaces Tested (BFS)
1	1-2
2	1-3
3	1-4
4	1-2-5
5	1-3-6
6	1-3-6-(3-7)
7	(1-2-5)-(1-3-6-(3-7))
8	1-4-8
9	(1-2-5)-(1-3-6-(3-7))-(1-4-8)

Bottom-up Integration Testing

- Bottom-up Integration Testing is a strategy in which the **lower-level modules are tested first**.
- These **tested modules** are then further used to **facilitate the testing of higher-level modules**.
- The process continues until all modules at top level are tested.
- Once the lower-level modules are tested and integrated, then the next level of modules are formed.
- **Advantages:**
  - Fault localization is easier.
  - No time is wasted waiting for all modules to be developed unlike Big-bang approach (all modules are combined at once and make a complicated system. This unity of different modules is then tested as an entity.)

- **Disadvantages:**
  - Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
  - An early prototype (early sample, model, or release of a product built to test a concept or process) is not possible.
- Example of Bottom-up integration; Note: Double arrows denote both the logical flow of components and integration approach.
- Logic flow is from top to bottom, and integration path is from bottom to top.
- The navigation in bottom-up integration starts from component 1 covering all sub-systems, till component 8 is reached.



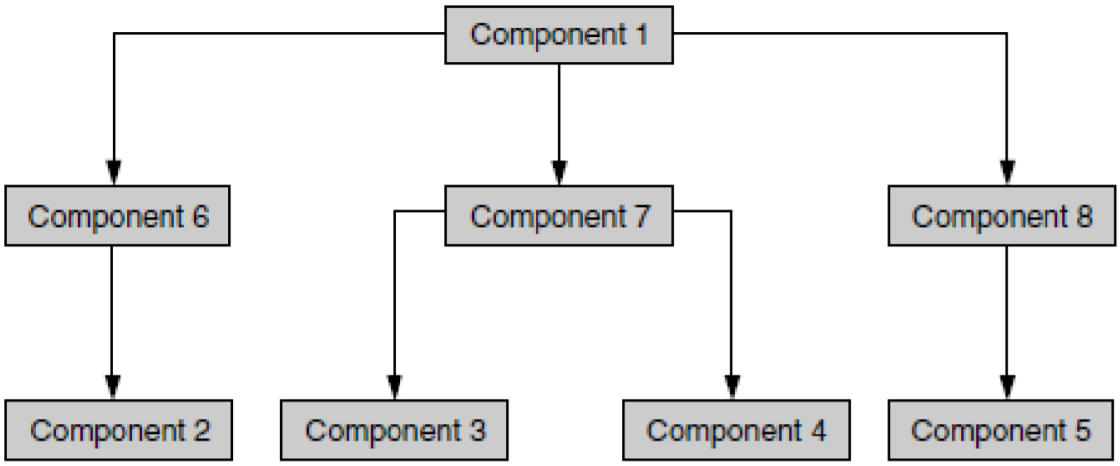
- The order in which the interfaces are to be tested is shown in the following table:

Step	Interfaces Tested
1	1-5
2	2-6, 3-6
3	2-6-(3-6)
4	4-7
5	1-5-8
6	2-6-(3-6)-8
7	4-7-8
8	(1-5-8)-(2-6-(3-6)-8)-(4-7-8)

- The number of steps in the bottom-up approach can be optimized into four steps, by combining steps 2 and 3 and by combining steps 5–8 as shown above.

Bi-Directional integration

- It's a **combination of top-down and bottom-up integration** approach used together to derive integration steps.



- The individual components 1, 2, 3, 4, and 5 are tested separately and bi-directional integration is performed initially with the use of stubs and drivers.
- Drivers are used to provide upstream connectivity while stubs provide downstream connectivity.
- A driver is a function which redirects the requests to some other component.
- Stubs simulate the behavior of a missing component.
- After the functionality of these integrated components are tested, the drivers and stubs are discarded.
- Once components 6, 7, and 8 are available, the testers then focus only on those components, as these are the components which need focus and are new.
- This approach is also called "sandwich integration."
- The order in which the interfaces are to be tested is shown in the following table:

Step	Interfaces Tested
1	6-2
2	7-3-4
3	8-5
4	(1-6-2)-(1-7-3-4)-(1-8-5)

- Steps 1–3 use a bottom-up integration approach and step 4 uses a top-down integration approach.
- An area where this approach comes in handy is when migrating from a two-tier to a three-tier environment. In the product development phase when a transition happens from two-tier architecture to three-tier architecture, the middle tier

(components 6–8) gets created as a set of new components from the code taken from bottom-level applications and top-level services.

System Integration

- System Integration is the process of **integrating all the physical and virtual components** of a system.
- The physical components consist of the various machine systems, computer hardware, inventory, etc.
- The virtual components consist of data stored in databases, software and applications.
- The process of integrating all these components, so that act like a single system, is the main focus of system integration.
- **Advantages**:
    - System integration using the big bang approach is **well suited** in a product development **scenario** where the **majority of components** are already available and **stable** and very few components get added or modified.
    - In this case, instead of testing component interfaces one by one, it makes sense to integrate all the components at one go and test once.
    - This **saves effort and time** for the multi-step component integrations.
- **Disadvantages**:
    - When a **failure/defect is encountered** during system integration, it is **difficult** to **locate the problem**, to find out in which interface the defect exists.
    - The testers may need to focus on specific interfaces and test them again.
    - Finding and correcting the root cause of the defect may be a difficult issue.
    - When integration testing happens in the end, the pressure from the approaching release date is very high. This **pressure on the engineers** may cause them to **compromise on the quality** of the product.
    - A certain component may take an excessive amount of time to be ready. This prevents testing other interfaces and wastes time.

Choosing Integration Method

- The following table gives some broad level guidelines on selecting the integration method. As mentioned in the above discussions, the integration method depends not only on the process, development model, but also on various other aspects.

Step	Factors	Suggested Integration method
1	Clear requirements and design	Top-down
2	Changing requirements, design and architecture	Bottom-up
3	Changing architecture, stable design	Bi-directional
4	Limited changes to existing architecture having less impact	System/Big Bang
5	Combination of above	Any one after careful analysis

Scenario Testing

- Scenario testing is defined as a **set of realistic user activities** that are **used for evaluating the product**.
- It is also defined as the **testing involving customer scenarios**.
- There are two methods to evolve scenarios:
    - System scenarios
    - Use-case scenarios/role-based scenario

System Scenarios

System scenario is a method whereby the **set of activities** used for scenario testing **covers several components in the system**.

The following approaches can be used to develop system scenarios:

1. **Story line:**
    - Develop a story line that combines various activities of the product that may be executed by an end user.
    - A user enters his or her office, logs into the system, checks mail, responds to some mails, compiles some programs, performs unit testing and so on.
    - All these typical activities carried out in the course of normal work, when coined together become a scenario.
2. **Life cycle/state transition:**
    - Consider an object, derive the different transitions/modifications that happen to the object, and derive scenarios to cover them.
    - Ex: in a bank account, you can start with opening an account with a certain amount of money, make a deposit, perform a withdrawal, calculate interest etc.

- All these activities are applied to the “money” object, and the different transformations, applied to the “money” object becomes different scenarios.

3. **Deployment/implementation stories from customer**: Develop a scenario from a known customer deployment/implementation details and create a set of activities by various users in that implementation.

Coverage of activities by scenario testing

End-user activity	Frequency	Priority	Applicable environments	No. of times covered
1. Login to application	High	High	W2000, W2003, XP	10
2. Create an object	High	Medium	W2000,XP	7
3. Modify parameters	Medium	Medium	W2000,XP	5
4. List object parameters	Low	Medium	W2000,XP, W2003	3
5. Compose email	Medium	Medium	W2000,XP	6
6. Attach files	Low	Low	W2000,XP	2
7. Send composed mail	High	High	W2000,XP	10

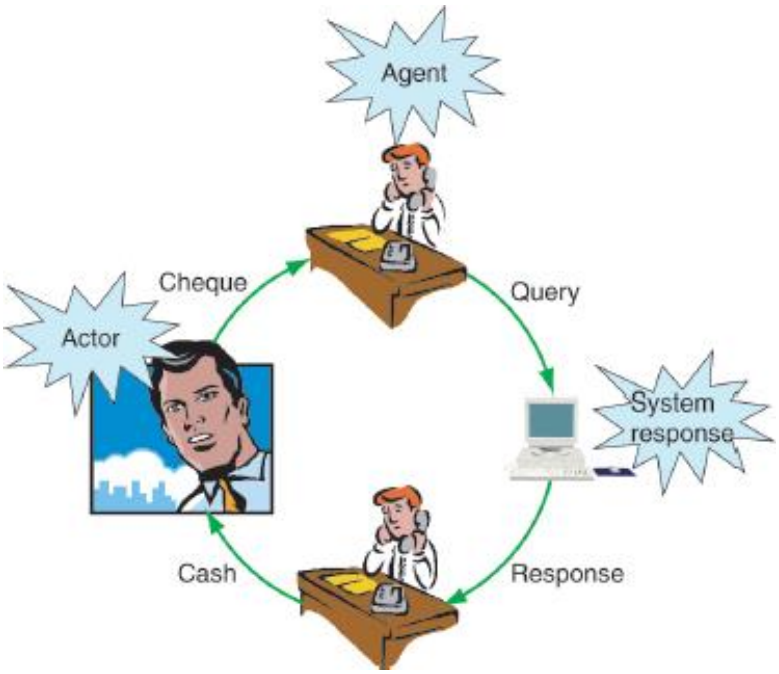
- Scenario testing is not meant to cover different permutations and combinations of features and usage in a product. However, by using a simple technique, some comfort feeling can be generated on the coverage of activities by scenario testing.
- The above table explains the concept with an example.
- From the table, it is clear that important activities have been very well covered by set of scenarios in the system scenario test.
- This kind of table also helps us to ensure that all activities are covered according to their frequency of usage in customer place and according to the relative priority assigned based on customer usage.

Use Case Scenarios

- A use case scenario is a **stepwise procedure** on how a **user intends to use a system**, with **different user roles** and associated parameters.
- A use case scenario can include **stories**, **pictures**, and **deployment details**.
- A use case can **involve several roles or class of users** who typically perform different activities based on the role.
- There are some activities that are common across roles and there are some activities that are very specific and can be performed only by the use belonging to a particular role.

- Use case scenarios term the **users with different roles as actors**.
- **What the product should do for a particular activity** is called as **system behavior**.
- Users with a specific role to **interact between the actors and the system** are called **agents**.

Example 1 of Use case scenario (withdrawing cash from a bank)



- A customer fills up a check and gives it to a clerk in the bank. The clerk verifies the balance from the computer and gives the required cash to the customer.
- The customer is the actor, the clerk is the agent, and the response given by the computer, which gives the balance in the account, is called the system response.
- This way of **describing different roles** in test cases **helps in testing the product without getting into the details** of the product.
- In the above example, the actor (i.e., the customer) need not know what the official is doing and what command he is using to interact with the computer. The actor is only concerned about getting the cash.
- The agent (i.e., the clerk) is not concerned about the logic of how the computer works. He or she is only interested in knowing from the computer whether he or she can give the requested cash or not.
- However, the system behavior (computer logic) needs to be tested before applying the sequence of agent activities and actor activities.
- In this example, the activities performed by the actor and the agent can be tested by testers who do not have much knowledge of the product. Testers who have in-depth knowledge of the product can perform the system behavior part of testing.

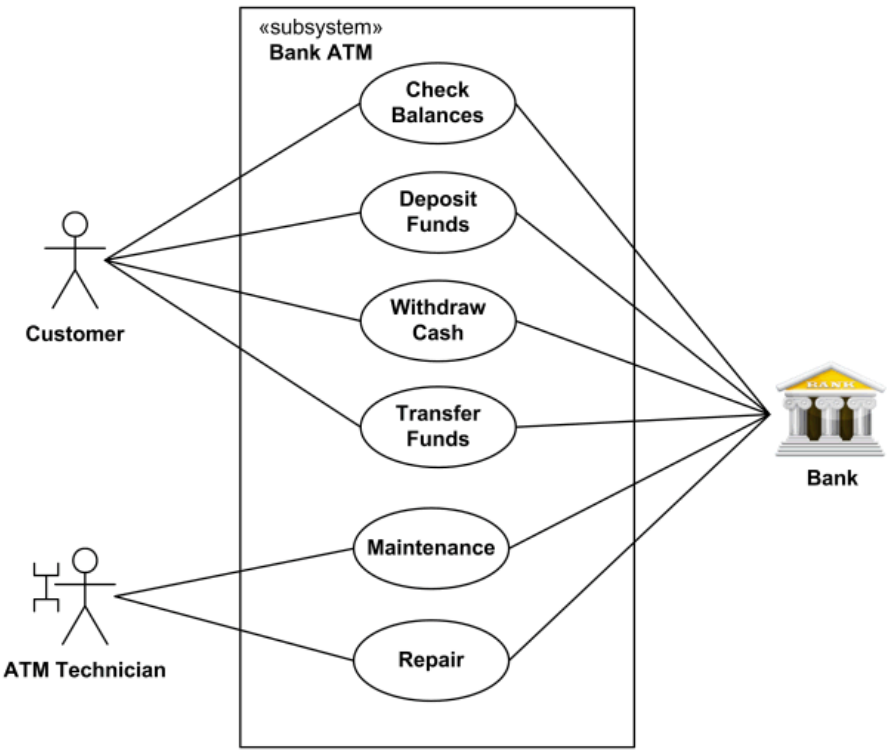
Example 2 of Use case scenario (withdrawing cash from an ATM)

- The **agent** part of the use cases is **not needed in all cases**.
- In a **completely automated system** involving the **customer and the system**, use cases can be written **without considering** the **agent** portion.
- Let us extend the earlier example of cash withdrawal using an ATM.

Actor	System response
User wants to withdraw cash and inserts the card in the ATM machine	Request for password or PIN
User fills in the password or PIN	Validate the password or PIN. Give a list containing types of accounts
User selects an account type	Ask the user for amount to withdraw
User fills in the amount of cash required	Check availability of funds Update account balance Prepare receipt Dispense cash
Retrieve cash from ATM	Print receipt

Example 3 of Use case scenario (ATM)

- *Customer* (actor) uses bank ATM to *Check Balances* of his/her bank accounts, *Deposit Funds*, *Withdraw Cash* and/or *Transfer Funds* (use cases).
- *ATM Technician* provides *Maintenance* and *Repairs*.
- All these use cases also involve *Bank* actor whether it is related to customer transactions or to the ATM servicing.



An example of use case diagram for Bank ATM subsystem - top level use cases.

Defect Bash

- Defect bash is an **ad hoc testing** where **people performing different roles** in an organization **test the product together** at the **same time**.
- This is very **popular among application development companies**, where the **product** can be **used by people** who **perform different roles**.
- The testing by all the participants during defect bash is **not based on written TCs**. What is to be tested is left to an individual's decision and creativity.
- They can also try some operations which are beyond the product specifications.
- Defect bash **brings together plenty of good practices** that are popular in testing industry. They are as follows:
  1. Enabling people to **cross boundaries** and **test beyond assigned areas**
  2. Bringing different **people performing different roles together** in the organization for testing
  3. **Letting everyone** in the organization **use the product before delivery**
  4. Bringing **fresh pairs of eyes** to **uncover new defects**
  5. Bringing in **people having different levels of product understanding** to **test** the product together **randomly**
  6. Enabling people to say "**system works**" as well as to "**break the system**"
- Even though it is said that defect bash is an ad hoc testing, **not all activities of defect bash are unplanned**.

All the **activities** in the **defect bash** are **planned activities**, except for what to be tested. It involves several steps:

1. **Choosing the Frequency and Duration of Defect Bash:**
   - **Frequent defect bashes** will **incur low return on investment**, and too **few defect bashes may not** meet the objective of **finding all defects**
   - Duration is also an important factor. **Optimizing the small duration** is a big saving as a large number of people are involved.
2. **Selecting the Right Product Build**
   - Since defect bash involves a large number of people, **effort** and **planning** a **good quality build** is needed for defect bash
   - An **intermediate build** where the **code** functionality is **evolving** or an **untested** build will make the purpose and outcome of a **defect bash ineffective**

3. **Communicating the Objective of Defect Bash**
   - The **purpose** and **objective** should be **clear**.
   - Once they are **told in advance**, the **members** of defect bash team will be in a **better** position to **contribute** towards stated objective.
4. **Setting up and Monitoring the Lab for defect bash**
   - During the defect bash, the **product parameters** and **system resources** (CPU, RAM, disk) need to be **monitored** for defects and also **corrected** so that **users** can continue to use the system for the **complete duration** of defect bash
   - There are **2 types of defects** that will emerge during the defect bash
     - The defects that are in the product, as **reported by** the **users** are called ***functional defects***
     - Defects that **unearthed while monitoring the system resources**, such as memory leak, long turnaround time and so on are called ***non-functional defects***
5. **Taking actions and fixing issues**
   - Take necessary **corrective action** after the defect bash
   - It is **difficult to solve** all the problems if they are taken **one by one** and fixed
   - The **defects** need to be **classified** into issues **at a higher level**, so that a similar outcome can be avoided in future defect bashes
6. **Optimizing the effort involved in defect bash**
   - Conduct "**micro level**" **defect bashes before** conducting one on a **large scale**
   - Since a defect bash is an **integration testing phase activity**, it can be **experimented by integration test team before** they **open** it up **for others**

Functional versus Non-functional Testing

Functional Testing	Non-functional Testing
Test the functionality of the software	Test the *non-functional aspects* or **readiness** of the the software including **performance**, **usability**, **reliability**
It has to be **done first**, i.e., before Non-Functional Testing.	It will be **done second**, i.e., after Functional Testing completes.

Functional Testing	Non-functional Testing
It is also called as **Behavioural Testing** and focuses on the **underlying application features**.	Focuses on the **performance** of the application
It can be **done manually**, though **test cases** can be **automated once** application is **stable**	It's **hard to do it manually**. Requires **3rd party applications** to **measure and test** application **performance**.
Types of Functional Testing includes ▪ Unit Testing ▪ Smoke Testing ▪ Integration Testing ▪ Regression Testing ▪ System Testing ▪ User Acceptance Testing	Types of Non-Functional Testing includes ▪ Volume testing. ▪ Load Testing ▪ Stress Testing ▪ Recovery Testing ▪ Scalability Testing ▪ Security Testing
Test data can be **prepared using** the **business** or **functional requirements** of the application.	**Test data** can be **prepared using** the **performance requirements** of the application
Testing tools used for functional testing includes **QTP**, **Selenium**, Ranorex, Teierilk Test Studio, **Sahi**	**Testing tools** used for non-functional testing includes **JMeter**, **LoadRunner**, WebLDAD, **NeoLoad**, **LoadComplete**
Example Test Case - Test whether the **user is able to login** to the application	**Example** Test Case - **Time required** to **load** the **home page**.

System Functional Testing approaches

1. Design/architecture verification
2. Business vertical testing
3. Deployment testing
4. Beta testing
5. Certification, standards, and testing for compliance

1. Design/Architecture Verification

- In this method of functional testing, the **System-level test cases** are developed and **checked against** the **design** and **architecture** to check whether they are actual product-level (aka system-level) test cases.

- This technique helps in **validating the product features** that are **written based** on **customer scenarios** and **verifying** them **using product implementation**.
- To know **whether a Test Case belongs to System Functional testing**, we have the following guidelines:
    - Is this focusing on code logic, data structures, and unit of the product?
      → If yes → **shift to Unit Testing**
    - Is this specified in the functional specification of any component?
      → If yes → **shift to Component Testing**
    - Is this specified in design & architecture specification for integration testing?
      → If yes → **shift to Integration Testing**
    - Is it focusing on product implementation but not visible to customers?
      → If yes → **shift to Unit/Component/Integration Testing**
    - Is it the right mix of customer usage and product implementation?
      → If yes → **System Testing**

---

2. Business Vertical Testing

- General purpose products like **workflow automation systems** can be used by different businesses and services.
- Using and testing the product for **different business verticals** such as **insurance**, **banking**, **asset management**, and so on, and **verifying the business operations** and usage, is called "business vertical (BV) testing."
- For this type of testing, the **procedure** in the **product** is **altered** to **suit the process** in the business.
- It is important that the **product understands** the **business processes** and **includes customization** as a feature **so** that **different BVs can use the product**.
- With the help of the ***customization*** feature, a **general workflow** of a system is **altered** to **suit specific BVs**.
- **Terminology**:
    - To explain this concept let us take the example of e-mail.
    - An **email** sent in the **insurance context** may be called a **claim** whereas when an **email** is sent in a **loan-processing system**, it is called a **loan application**.
    - An **email** sent to a **blood bank service cannot take the same priority** as an **internal email** sent to an employee by another employee.

    - Hence the **terminology feature** of the product **should call** the **email appropriately** as a claim or a transaction and also **associate the profile** and **properties** in a way a particular business vertical works.
- **Syndication**:
    - *Dictionary meaning*: the transfer of something for control or management by a group of individuals or organizations.
    - **Not all the work needed for BVs** are **done by product development organizations**.
    - **Solution integrators**, **service providers pay** a **license fee** to a product company and **sell the products** and **solutions using their name** and **image**.
    - In this case the product name, company name, technology names, and **copyrights** may **belong** to the **product organizations** and the **solution integrators/service providers change** the **names** in the **product**.
    - A product should **provide features for those syndications** in the product and they are as tested part of BV testing.
- BV testing can be done in two ways
    - **Simulation**: In simulation of a vertical test, the **customer** or the **tester assumes requirements** and the **business flow** is **tested**.
    - **Replication**: In replication, customer data and process are obtained and the product is completely customized, tested, and the customized product as it was tested is released to the customer.

---

3. **<u>Deployment Testing</u>**

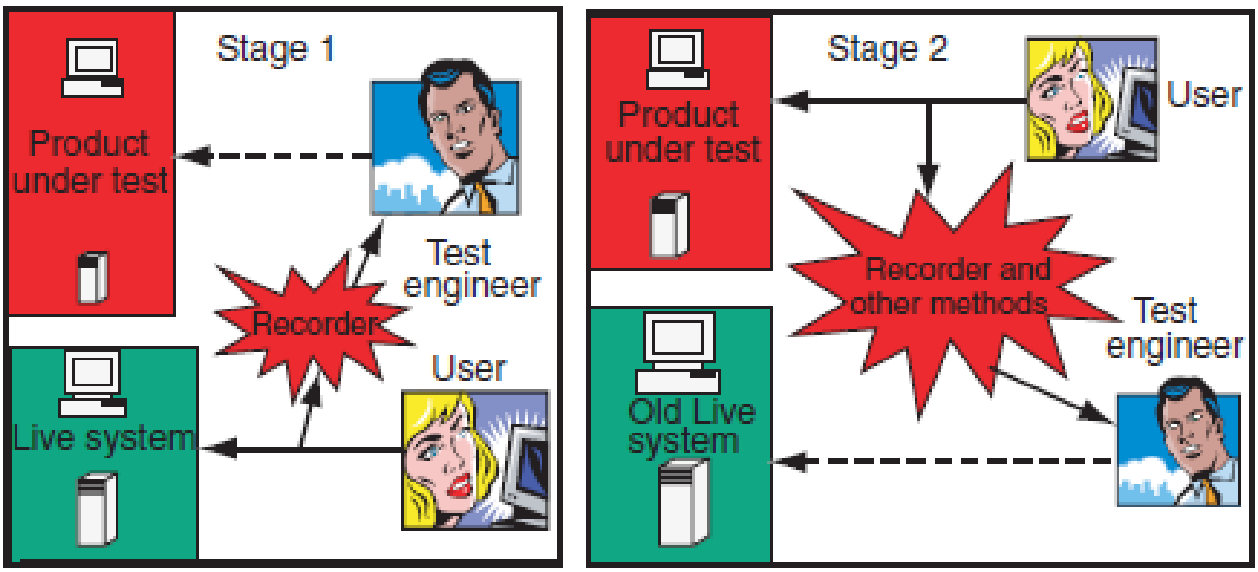
- System testing is the final phase before product delivery. By this time its prospective **customers** and **configuration used by them** would be known.
- In some cases, the **products** would have been **committed for sale**.
- Hence, **system testing** is the **right time** to **test** the product **for those customers** who are **waiting** for it.
- ***Offsite deployment***
    - This type of deployment testing **happens in a product company**.
    - It **ensures** that **customer deployment requirements** are **met**.
- ***Onsite deployment***
    - This type of **deployment testing after** the **release** of the product.
    - It **utilizes** the **resources** and **setup available** in **customers' locations**.

- This is a **combined effort by** the **product development organization** and the organization trying to use the product (**customer**).

Stages of Onsite Deployment Testing

Stage 1

- Actual **data from the live system** is **taken** and similar machines and configurations are mirrored, and the **operations from the users** are **rerun** on the **mirrored deployment machine**.
- This **gives an idea whether** the **enhanced** or similar **product** can **perform** the **existing functionality without affecting** the **user**.
- Some deployments use **intelligent recorders** to **record the transactions** that **happen on a live system** and **commit these operations** on a **mirrored system** and then **compare the results against the live system**. The objective of the recorder is to help in keeping the mirrored and live system identical with respect to business transactions.
- Figure: In Stage 1, it can be seen that the recorder intercepts the user and live system to record all transactions. All the recorded transactions from the live system are then played back on the product under test under the supervision of the test engineer (shown by dotted lines).



Stage 2

- After a successful first stage, the **mirrored system** is **made a live system** that runs the new product.
- This stage helps to **avoid any major failures since** some of the **failures** can be **noticed only after** an **extended period of time**.

- In this stage, the **old system** is **preserved** to **enable going back** if any major failures are observed at this stage.
- **If no failures** are observed in this stage of deployment for an extended period (for example, one month), then the **onsite deployment is considered successful** and the **old live system** is **replaced by the new system**.
- Figure: The test engineer records all transactions using a recorder and other methods and plays back on the old live system (shown again by dotted lines).

---

4. **<u>Beta Testing</u>**

- A product rejected by the customer after delivery means a huge loss to the organization. There are m**any reasons for a product not meeting the customer requirements**. They are as follows:
    1. Product **may not be meeting implicit customer requirements** like **ease of use**
    2. Customers' **business requirements keep changing constantly** and a **failure to reflect these changes** in the product makes the latter **obsolete**
    3. Using **ambiguous customer requirements** and not resolving them with the customer results in rejection of the product.
    4. The **understanding** of the **requirements** may be **correct but** their **implementation** could be **wrong**.
    5. **Lack of usability** and **documentation** makes it difficult for the customer to use the product
- To **reduce the risk of rejection**, which is the objective of system testing, **periodic feedback is obtained** on the product.
- One of the mechanisms used is **sending the product that is under test** to the **customers** and **receiving the feedback**. This is called ***beta testing***.
- This testing is performed by the customer and helped by the product organization.
- There are various activities that are planned and executed according to a specific schedule. This is called a beta program.
- Some of the activities involved in the beta program are as follows:
    1. **Collecting the list of customers** and their **beta testing requirements** along with their expectations on the product
    2. Working out a **beta program schedule** and informing the customers. Not all the customers in the list need to agree to the **start date and end date** of the beta program. The end date of a beta program should be **reasonably before**

the product release date so that the beta testing defects can be fixed before the release.

3. Sending **documentation** for reading in advance and **training** the customer on product usage.
4. **Prepare sets of entry/exit criteria** for beta testing.
5. **Sending the beta product** to the customer and enable them to carry out their own testing.
6. **Collecting the feedback periodically** from the customers and prioritizing the defects for fixing
7. **Responding to customers' feedback** with **product fixes** or **documentation changes** and closing the communication loop with the customers periodically
8. **Analyzing** and concluding **whether the beta program met the exit criteria**.
9. **Communicate the progress** and action items **to customers** and formally **closing the beta program**.
10. **Incorporating** the appropriate **changes** in the **product**.

Alpha Testing vs Beta Testing

Alpha Testing	Beta Testing
Goals: • Evaluate **software quality** • Ensuring **every functionality works** • **Check readiness for beta testing**	Goals: • Evaluate **users' satisfaction** • **Receive** valuable **feedback** • Ensure **release readiness**
Performed by **company employees**	Performed by **customers/users**
Performed in **lab/testing environment**	Performed in **real-time environment**
Uses both **BBT and WBT**	Uses only **BBT**
Performed **after System Testing**	Performed **after Alpha Testing**
Requires **knowledge** about the **design and architecture** of the system	Requires **knowledge** about the **functionality of the system**
Does not check the system's **reliability** and **security**	It **checks** the system's **reliability** and **security**

5. Certification, Standards and Testing for Compliance

Certification

- A **product needs to be certified** with the **popular hardware**, **operating system**, **database**, and other infrastructure pieces. This is called certification testing.
- The **sales** of a product **depend** on **whether** it was **certified with** the **popular systems** or not.
- This is one type of testing where there is **equal interest** from the **product development organization, the customer, and certification agencies** to certify the product.
- The **certification agencies produce automated test suites**, the **product** development **organization runs those test suites** and **corrects the problems** to ensure that tests are successful.
- The **results** are then **sent to** the **certification agencies** and they give the **certification for the product**. The test suite may be rerun by the certification agencies to verify the results.

Standards

- There are **many standards for each technology area** and the **product may need to conform to those standards**.
- So, **companies select** the **standards to be implemented** at the **beginning** of the product cycle.
- This also **helps** the **customer not** to **worry** too much about the **product's future compatibility** with other products.
- Eg: for example, **IPv6 in networking** and **4G in mobile technology**
- Testing the product to ensure that **these standards** are **properly implemented** is called *testing for standards.*
- Once the product is tested for a set of standards, they are **published** in the **release documentation for** the **information** of the **customers** so that they know what standards, are implemented in the product.

Compliance

- Testing the product for **contractual, legal, and statutory compliance** is one of the critical activities of the system testing team. The following are some examples of compliance testing.

- **Compliance to FDA:** This act by the **Food and Drug Administration** requires that **adequate testing** be done for products such as **cosmetics**, **drugs**, and **medical sciences**. This also requires that all the **test reports** along with complete **documentation** of **test cases**, **execution information** for **each test cycle** along with supervisory approvals be **preserved for checking adequacy of tests** by the FDA.
- **508 Accessibility Guidelines:** This accessibility set of guidelines requires the product to meet some **requirements for its physically challenged users**. These guidelines insist that the product should be as accessible to physically challenged people as it is to people without those disabilities.
- **SOX (Sarbanes–Oxley's Act):** This act requires that **products and services be audited** to **prevent financial fraud** in the organization. The **software** is required to **go through all transactions** and **list out the suspected faulty transactions** for analysis. The testing for this act **helps the top executives** by keeping them aware of **financial transactions** and their **validity**.
- **OFAC and Patriot Act:** This act requires the **transactions** of the **banking applications** be **audited** for **misuse of funds** for terrorism.

Non-Functional Testing

Scalability Testing

- The objective of scalability testing is to find out the **maximum capability** of the **product parameters**.
- As the exercise **involves finding the maximum**, the **resources required** for this kind of testing are **normally very high**.
- For example, one of the scalability test cases could be finding out **how many client machines can simultaneously log in to the server** to perform some operations. In Internet space, some of the services can get up to **a million access to the server**. Hence, **trying to simulate** that kind of real-life scalability parameter is **very difficult but** at the same time **very important**.
- At the beginning of the scalability exercise, the **maximum capability** of the system **may not be known**. Hence a high-end configuration is selected and the **scalability parameter is increased step by step** to reach the maximum capability

- **If** the **customer's requirements** are **more than what the system can provide**, the **system is reworked** to meet the requirements **before scalability testing can begin**
- Contrary to other types of testing, **scalability testing does not end when** the **customer requirements are met**. The testing **continues till the maximum capability** of a scalable parameter is **found** out for a particular configuration. Having a highly scalable system that **considers the future requirements** of the customer helps a product to have a **long lifetime**.
- **Failures during scalability test** include the **system not responding**, or the **system crashing**, and so on. But whether the **failure is acceptable or not** has to be **decided** on the **basis of business goals** and **objectives**.
- **For example**, a product not able to respond to 100 concurrent users while its objective is to serve at least 200 users simultaneously is considered a failure. When a product expected to withstand only 100 users, fails, when its load is increased to 200, then it is a passed test case and an acceptable situation.
- Scalability tests are performed on different configurations to check the product's behavior. For each configuration, data are collected and analyzed.
- **Scalability Testing Attributes**:
  - Response Time
  - Screen transition
  - Throughput
  - Time (Session time, reboot time, printing time, task execution time)
  - Performance measurement with a number of users
  - Request/Hits/Transactions per seconds
  - Performance measurement under load
  - Network Usage
  - CPU / Memory Usage
  - Web Server (request and response per seconds)
  - Performance measurement with a number of users

Reliability Testing

- Reliability testing is done to **evaluate the product's ability to perform** its **required functions** under stated conditions **for a specified period of time** OR **for a large number of iterations**.
- Examples of reliability include **querying a database continuously for 48 hours** and **performing login operations 10,000 times**. The result after each iteration should remain the same i.e., it should be reliable.

- Producing a reliable product requires **sound techniques, good discipline, robust processes, strong management, and involves a complete range of activities** for every role or function in a product organization.
- This product reliability is achieved by focusing on the following activities:
    1. **Defined engineering processes**:
        - Software reliability can be **achieved** by following **clearly defined processes**.
        - The team is mandated to **understand requirements for reliability**, right **from the beginning** and **focuses** on **creating** a **reliable design upfront**.
        - **All the activities** (such as design, coding, testing, documentation) are **planned**, taking into consideration the reliability requirements of the software.
    2. **Review of products at each stage**:
        - **At the end of each stage** of the product development life cycle, **the products** produced are **reviewed**.
        - This ensures **early detection of error** and their fixes as soon as they are introduced.
    3. **Change management procedures**:
        - Many **errors percolate to the product** due to **improper impact analysis of changes made to the product** (adding new features/fixing defects).
        - **Changes** received **late** during the product development life cycle can prove harmful.
        - There **may not be adequate time for regression testing** and hence the product is likely to have errors due to changes.
        - Hence, having a **clearly defined change management procedure** is necessary to deliver reliable software.
    4. **Review of testing coverage**:
        - **Allocating time for the different phases and types of testing** can help in catching errors as and when the product is being developed, rather than after the product is developed.
        - All the testing activities are reviewed for adequacy of time allotted, test cases, and effort spent for each type of testing.

5. **Ongoing monitoring of the product**:
    - **Once** the **product** has been **delivered**, it is **analyzed proactively** for any possibly missed errors.
    - In this case, the process as well as the product is fixed for missed defects.
    - This prevents the same type of defects from reappearing.

Stress Testing

- Stress testing is done to **evaluate a system beyond the limits** of **specified requirements or resources**, to **ensure that system does not break**.
- Stress testing is done to **find out if the product's behavior degrades under extreme conditions** and when it is **denied the necessary resources**.
- Stress testing helps in understanding **how the system can behave under extreme** (insufficient memory, inadequate hardware) and **realistic situations**.
- System resources upon being exhausted may cause such situations.
- **Extreme situations** such as a **resource not being available** can also be **simulated**.
- The product is **over-loaded deliberately** to **simulate the resource crunch** and to find out its behavior.
- It is **expected to gracefully degrade** on increasing the load, but the **system is not expected to crash** at any point of time during stress testing.
- Unlike **reliability testing** which is **performed by keeping a constant load condition** till the test case is completed (the load is increased only in the next iteration of the test case), in **stress testing**, the **load is generally increased through various means** such as **increasing the number of clients, users, and transactions** till and beyond the resources are completely utilized.
- When the load keeps on increasing, the **product reaches a stress point** when some of the **transactions start failing** due to resources not being available.
- To continue the stress testing, the **load is slightly reduced below this stress point** to **check whether the product recovers** and whether the **failure rate decreases** appropriately.
- This **exercise of increasing/decreasing the load** is **performed 2/3 times** to check for consistency in behavior and expectations.

- Sometimes, the **product may not recover immediately when the load** is **decreased**.
- There are **several reasons for this**:
    - Some **transactions may be in the wait queue**, **delaying the recovery**.
    - Some **rejected transactions many need to be purged**, delaying the recovery.
    - Due to failures, some **clean-up operations may be needed by the product**, delaying the recovery.
    - Certain **data structures may have got corrupted** and may permanently prevent recovery from stress point.

Acceptance Testing

- Acceptance testing is a **phase after system testing** that is normally **done by** the **customers or representatives of the customer**.
- The **customer defines a set of test cases** that will be executed **to qualify and accept** the product.
- Acceptance test cases are **normally small in number** and are **not written with the intention of finding defects**.
- These test cases are **executed by the customers themselves** to **quickly judge** the quality of the product **before deciding to buy the product**.
- **If acceptance tests are performed by the product organization** alone, acceptance tests are executed to **verify whether** the **product meets** the **acceptance criteria defined** during the **requirements** definition **phase**.
- Acceptance test cases are **black box type of test cases**.
- They are directed at verifying one or more acceptance criteria.
- Acceptance test cases failing in a customer site may cause the product to be rejected and may mean financial loss or may mean rework of product involving effort and time.

Acceptance Criteria

1. **Product Acceptance**
   - **Existing test cases** are looked at and **certain categories of test cases** can be **grouped to form acceptance criteria**.
   - For example**, all performance test cases** should pass meeting the response time criteria.
   - **Testing for adherence** to any **specific legal or contractual terms** is included in the acceptance criteria. Testing for compliance to specific laws like **Sarbanes–Oxley** can be part of the acceptance criteria.
2. **Procedure Acceptance**
   - Acceptance criteria can be **defined based on the procedures followed for delivery**.
   - An example of procedure acceptance could be **documentation** and **release media**. Some examples of acceptance criteria of this nature are as follows:
     - **User, admin and troubleshooting documentation** should be part of the release.
     - Along with **binary code**, the **source code** of the product with **build scripts** to be delivered in a CD.
     - A **minimum of 20 employees are trained** on the product usage prior to deployment
3. **Service Level Agreements (SLAs)**
   - Service level agreements are ge**nerally part of a contract signed by the customer and product organization**.
   - The important contract items are taken and verified as part of acceptance testing. For example:
     - All **major defects** that come up during **first three months** of deployment need to be **fixed free of cost**
     - **Downtime** of the implemented system **should be less than 0.1**%
     - All **major defects** are to be **fixed within 48 hours of reporting**.

Selecting Test Cases for Acceptance Testing

- The test cases for acceptance testing are **selected from the existing set of test cases** from different phases of testing.
- The following are the **guidelines to decide which test cases can be included** for acceptance testing:
    1. **End-to-end functionality verification**:
        - Test cases that include the **end-to-end functionality of the product** are taken up for acceptance testing.
        - This ensures that **all the business transactions are tested as a whole** and those transactions are completed successfully.
        - **Real-life test scenarios are tested** when the product is tested end-to-end.
    2. **Domain tests**:
        - Since acceptance tests focus on **business scenarios**, the product domain tests are included.
        - **Test cases that reflect business domain knowledge** are included.
    3. **User scenario tests**: Acceptance tests reflect the **real-life user scenario verification**. As a result, test cases that portray them are included.
    4. **Basic sanity tests**:
        - Tests that **verify the basic existing behavior** of the product are included.
        - These tests **ensure that the system performs the basic operations** that it was intended to do.
        - Such tests may gain more attention when a product undergoes changes or modifications.
        - It verifies that the **existing behavior is retained without any breaks**.
    5. **New functionality**: When the product undergoes modifications or changes, the acceptance test cases **focus on verifying the new features**.
    6. **A few non-functional tests**: Some non-functional (**scalability, reliability etc**) tests are included and executed as part of acceptance testing **to double-check** that the non-functional aspects of the product **meet the expectations**.
    7. **Tests pertaining to Legal Obligations and SLAs**: Tests that are written to check if the **product complies with certain legal obligations** and **SLAs** are included in the acceptance test criteria.

8. **Acceptance test data**: Test cases that **make use of customer real-life data** are included for acceptance testing.

Executing Acceptance Tests

- Acceptance testing is done by the customer or by the representative of the customer to check whether the product is ready for use in the real-life environment.
- **Sometimes** the **customers themselves do the acceptance tests**.
- In such cases, the job of the **product organization** is to **assist the customers** in acceptance testing and **resolve the issues** that come out of it.
- **If** the acceptance testing is **done by the product organization**, **forming the acceptance test team** becomes an **important activity**.
- The acceptance test team **may get the help of team members** who **developed/tested the software** to obtain the **required product knowledge**.
- There could also be **in-house training material** that could serve the purpose.
- The **role of the testing team members** during and prior to acceptance test is crucial since they may **constantly interact with the acceptance team members**.
- Test team members help the acceptance members to
    - **get the required test data**,
    - **select and identify test cases**, and
    - **analyze the acceptance test results**.
- During test execution, the **acceptance test team reports its progress regularly**. The **defect reports** are **generated** on a **periodically**.
- **Defects reported** during acceptance tests could be of **different priorities**. Test teams help acceptance test team report defects.
