UT-2 Solution

Q1. **a. Explain following design concept: Abstraction, Modularity.**

1. **Abstraction**
   Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity**
   Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components, then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.

b. What is design? Explain design principles.

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur.

- ✓ The Open-Closed Principle (OCP). "A module [component] should be open for extension but closed for modification" [Mar00]. This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design.
- ✓ The Liskov Substitution Principle (LSP). "Subclasses should be substitutable for their base classes" [Mar00]. This design principle, originally proposed by Barbara Liskov [Lis88], suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.
- ✓ Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions".
- ✓ The Interface Segregation Principle (ISP). "Many client-specific interfaces are better than one general purpose interface" [Mar00]. There are many instances in which multiple client components use the operations provided by a server class.

- ✓ The Release Reuse Equivalency Principle (REP). "The granule of reuse is the granule of release" [Mar00]. When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.
- ✓ The Common Closure Principle (CCP). "Classes that change together belong together." [Mar00]. Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address the same functional or behavioral area.
- ✓ The Common Reuse Principle (CRP). "Classes that aren't reused together should not be grouped together" [Mar00]. When one or more classes within a package changes, the release number of the package changes.

c. Explain coupling with example.

As the amount of communication and collaboration increases (i.e., as the degree of "connectedness" between classes increases), the complexity of the system also increases. And as complexity increases, the difficulty of implementing, testing, and maintaining software grows. Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.

Following is coupling categories:

- ✓ **Content coupling.** Occurs when one component "surreptitiously modifies data that is internal to another component" [Let01]. This violates information hiding—a basic design concept.
- ✓ **Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
- ✓ **Control coupling.** Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.
- ✓ **Stamp coupling.** Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.
- ✓ **Data coupling.** Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.
- ✓ **Routine call coupling.** Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

- ✓ **Type use coupling.** Occurs when component A uses a data type defined in component B (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type" [Let01]). If the type definition changes, every component that uses the definition must also change.
- ✓ **Inclusion or import coupling.** Occurs when component A imports or includes a package or the content of component B.
- ✓ **External coupling.** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

d) Explain the types of cohesion with example.

- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself. Below cohesion type is listed in order of the level of the cohesion:
  - ✓ **Functional.** Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.
  - ✓ **Layer**. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
  - ✓ **Communicational.** All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it.
- Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain. You should strive to achieve these levels of cohesion whenever possible.

Q2. a. Explain risk management process. Prepare RMMM plan for the identified risk "team members will leave the project in between the schedule".

- All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.
- If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for risk mitigation.

For example, assume that high staff turnover is noted as a project risk r1. Based on past history and management intuition, the likelihood of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x1 is projected as critical. That is, high turnover will have a critical

impact on project cost and schedule. To mitigate this risk, you would develop a strategy for reducing turnover.

- Among the possible steps to be taken are:
  - ✓ Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
  - ✓ Mitigate those causes that are under your control before the project starts.
  - ✓ Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
  - ✓ Organize project teams so that information about each development activity is widely dispersed.
  - ✓ Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
  - ✓ Conduct peer reviews of all work (so that more than one person is "up to speed").
  - ✓ Assign a backup staff member for every critical technologist
- As the project proceeds, risk-monitoring activities commence.
  - ✓ The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely.
  - ✓ In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.
  - ✓ In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps.
  - ✓ For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner.
- **Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality.** Continuing the example, the project is well under way and a number of people announce that they will be leaving.
  - ✓ If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team.
  - ✓ In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed."
  - ✓ Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode."

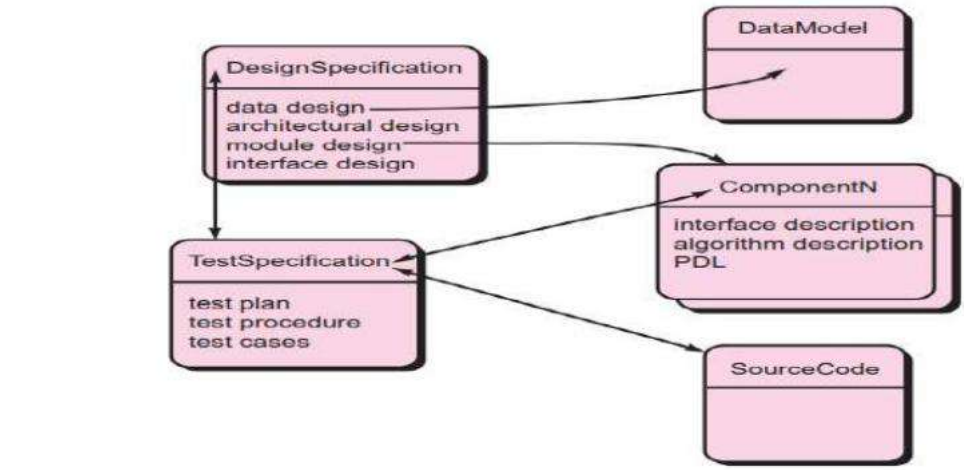
b) **List any five configuration items produced during SDLC. What is the need for SCM in Software Engineering?**

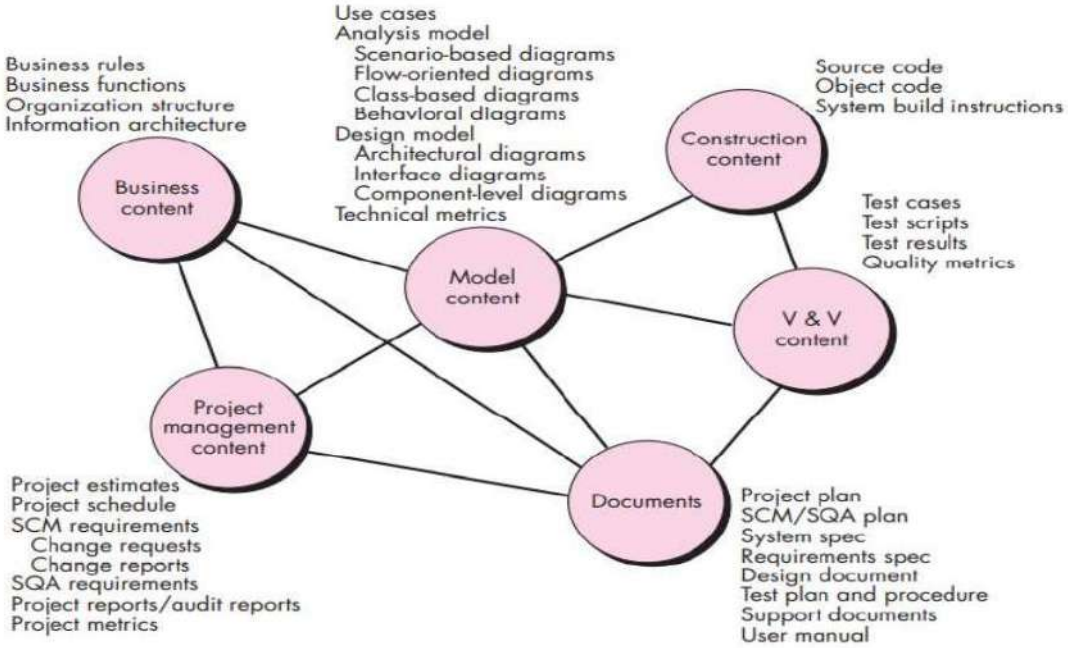
Software configuration item

• software configuration item is information that is created as part of the software engineering process
• More realistically, an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, or a named program component).
• In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, browsers, and other automated tools are "frozen" as part of the software configuration.
• SCIs are organized to form configuration objects that may be catalogued in the project database with a single name.
• A configuration object has a name, and attributes, and is "connected" to other objects by relationships.

SCM repository

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner.
- It provides the obvious functions of a modern database management system by ensuring data integrity, sharing, and integration.
- In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products. To achieve these capabilities, the repository is defined in terms of a meta-model.
- The meta-model determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs





c. What is the importance of change control? Explain it with configuration items and baselines.

- The reality of change control in a modern software engineering context has been summed up beautifully by James Bach [Bac98]: Change control is vital. But the forces that make it

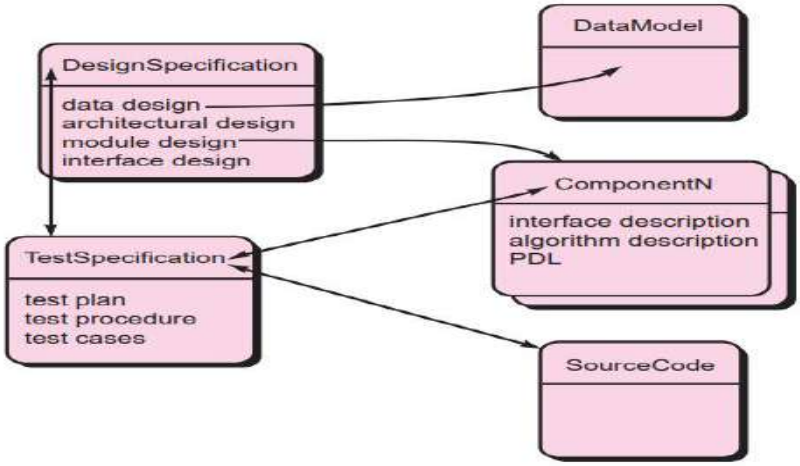
necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

- For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change
- A change request is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change.
- The results of the evaluation are presented as a change report, which is used by a change control authority (CCA)—a person or group that makes a final decision on the status and priority of the change. An engineering change order (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit
- The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made.
- As an alternative the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied.
- The object(s) is (are) then "checked in" to the database and appropriate version control mechanisms are used to create the next version of the software.
- These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control.
- Access control governs which software engineers have the authority to access and modify a particular configuration object. Synchronization control helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

Software configuration item

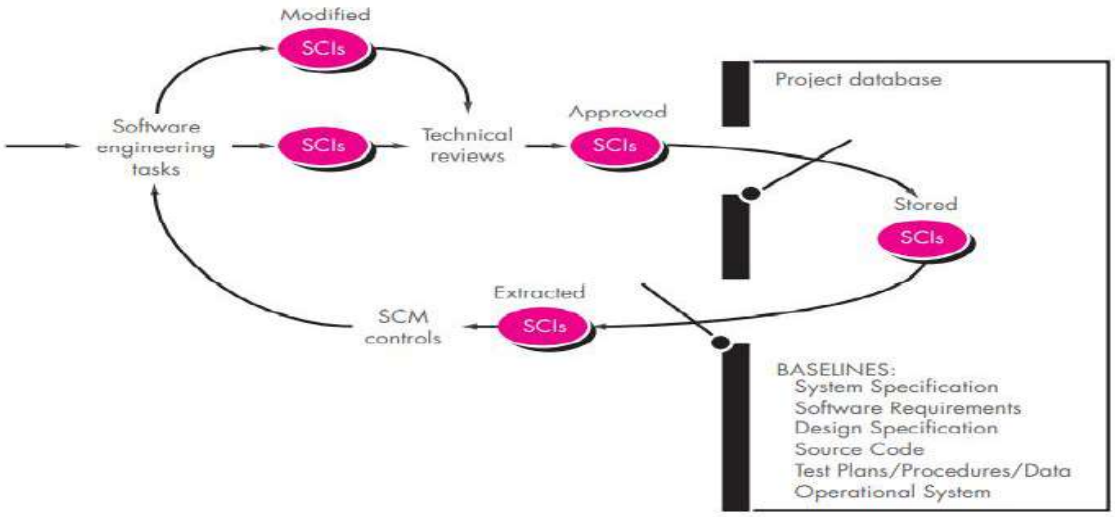
- software configuration item is information that is created as part of the software engineering process
- More realistically, **an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, or a named program component).**
- In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, **specific versions of editors, compilers, browsers, and other automated tools are "frozen"** as part of the software configuration.
- SCIs are organized to form **configuration objects** that may be cataloged in the project database with a single name.

- A **configuration object has a name, attributes, and is "connected" to other objects by relationships.**



5.5.1 Baselines

- ✓ A baseline is a software configuration management concept that helps you to control change without seriously impeding justifiable change.
- ✓ The IEEE defines a baseline as: **A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.**
- ✓ Before a software configuration item becomes a baseline, changes may be made quickly and informally.
- ✓ However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.



Q3. a. Describe test strategies for Conventional Software.

It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

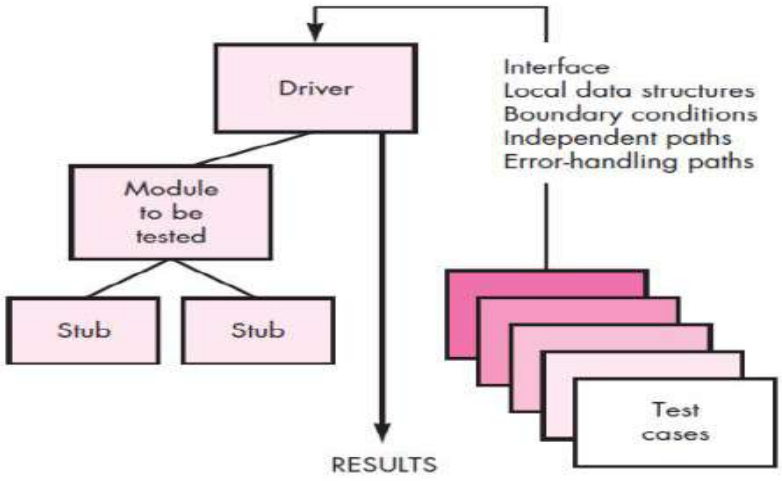
Unit-test considerations: -

- The module interface is tested to ensure proper information flows (into and out).
- Local data structures are examined to ensure temporary data store during execution.
- All independent paths are exercised to ensure that all statements in a module have been executed at least once.
- Boundary conditions are tested to ensure that the module operates properly at boundaries. Software often fails at its boundaries.
- All error-handling paths are tested.

Unit-test procedures: -

- Unit testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results. Because a component is not a stand-alone program, driver and/or stub software must often be developed for each unit test.
- Driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.

- Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.



Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

Top-down integration testing.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

Bottom-up integration

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub function.

2. A driver (a control program for testing) is written to coordinate test case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined moving upward in the program structure.

Regression testing

1. Each time a new module is added as part of integration testing, the software changes.
2. New data flow paths are established, new I/O may occur, and new control logic is invoked.
3. These changes may cause problems with functions that previously worked flawlessly.
4. Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Smoke testing: -

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Benefits:

- Integration risk is minimized.
- The quality of the end product is improved.
- Error diagnosis and correction are simplified.
- Progress is easier to assess.

b. What is testing? Explain the different levels of testing.

Software testing can be stated as the process of verifying and validating whether a software or application is bug-free, meets the technical requirements as guided by its design and development, and meets the user requirements effectively and efficiently by handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy, and usability. It mainly aims at measuring the specification, functionality, and performance of a software program or application.

Testing levels are the procedure for finding the missing areas and avoiding overlapping and repetition between the development life cycle stages. We have already seen the various phases such as Requirement collection, designing, coding testing, deployment, and maintenance of SDLC (Software Development Life Cycle).

In software testing, we have four different levels of testing, which are as discussed below:

Unit testing is the first level of software testing, which is used to test if software modules are satisfying the given requirement or not.

- The first level of testing involves analysing each unit or an individual component of the software application.
- Unit testing is also the first level of functional testing. The primary purpose of executing unit testing is to validate unit components with their performance.
- A unit component is an individual function or regulation of the application, or we can say that it is the smallest testable part of the software. The reason of performing the unit testing is to test the correctness of inaccessible code.
- Unit testing will help the test engineer and developers in order to understand the base of code that makes them able to change defect causing code quickly. The developers implement the unit.

Integration Testing

- The second level of software testing is the integration testing. The integration testing process comes after unit testing.
- It is mainly used to test the data flow from one module or component to other modules.
- In integration testing, the test engineer tests the units or separate components or modules of the software in a group.
- The primary purpose of executing the integration testing is to identify the defects at the interaction between integrated components or units.

- When each component or module works separately, we need to check the data flow between the dependent modules, and this process is known as integration testing.
- We only go for the integration testing when the functional testing has been completed successfully on each application module.
- In simple words, we can say that integration testing aims to evaluate the accuracy of communication among all the modules.

System Testing

- The third level of software testing is system testing, which is used to test the software's functional and non-functional requirements.
- It is end-to-end testing where the testing environment is parallel to the production environment. In the third level of software testing, we will test the application as a whole system.
- To check the end-to-end flow of an application or the software as a user is known as System testing.
- In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

Acceptance Testing

The last and fourth level of software testing is acceptance testing, which is used to evaluate whether a specification or the requirements are met as per its delivery.

- The software has passed through three testing levels (Unit Testing, Integration Testing, System Testing). Some minor errors can still be identified when the end-user uses the system in the actual scenario.
- In simple words, we can say that Acceptance testing is the squeezing of all the testing processes that are previously done.
- The acceptance testing is also known as User acceptance testing (UAT) and is done by the customer before accepting the final product.
- Usually, UAT is done by the domain expert (customer) for their satisfaction and checks whether the application is working according to given business scenarios and real-time scenarios.