**SOFTWARE ENGINEERING LABORATORY**

**COURSE CODE: BCSE2355**

Lab Manual

*for*

BACHELOR OF

Engineering & Technology

**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING**

**GALGOTIAS UNIVERSITY, GREATER NOIDA**

**UTTAR PRADESH**

**SUBMITTED BY:**

**ABHINAV KUMAR CHOUDHARY**

**(21SCSE1011615)**

**SEC-6**

**SUBMITTED TO:**

**EXPERIMENT LIST**

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Title of Lab Experiments** |  |
| 1. | Introduction to software development lifecycle model |  |
| 2. | Understanding an SRS . |  |
| 3. | Make an ER diagram for Library management system. |  |
| 4. | To prepare DATA FLOW DIAGRAM for any project. |  |
| 5. | Introduction to software testing and quality assurance. |  |
| 6 | Make a Use case diagram for railway reservation system |  |
| 7 | WAP in C/C++ to find the area of a circle, Triangle, Square and Rectangle. |  |
| 8 | Draw the component diagram. |  |
| 9. | Perform software development and software testing in e commerce. |  |
| 10 | Draw class diagram. Identify them as weak and strong class. |  |

# EXPERIMENT NO. 1

**Aim**: Introduction to software development lifecycle model.

#### Requirements:

**Hardware Requirements:**

* PC with 300 megahertz or higher processor clock speed recommended; 233 MHz minimum required.
* 128 megabytes (MB) of RAM or higher recommended (64 MB minimum supported)
* 1.5 gigabytes (GB) of available hard disk space
* CD ROM or DVD Drive
* Keyboard and Mouse(compatible pointing device).

**Software Requirements:** Rational Rose, Windows XP, **Theory:**

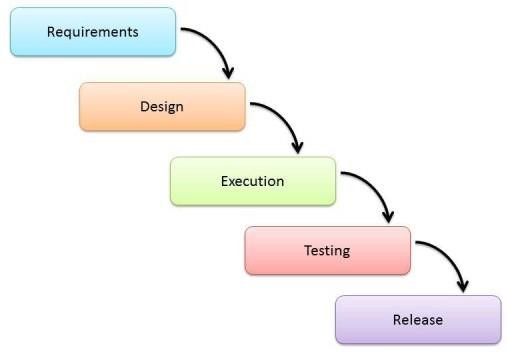
Software development life cycle (**SDLC**) is a series of phases that provide a common understanding of the software building process. How the software will be realized and developed from the business understanding and requirements elicitation phase to convert these business ideas and requirements into functions and features until its usage and operation to achieve the business needs. A good software engineer should have enough knowledge on how to choose the SDLC model based on the project context and the business requirements.



*Types of Software developing life cycles (SDLC)*

* + Waterfall Model
  + V-Shaped Model
  + Evolutionary Prototyping Model
  + Spiral Method (SDM)
  + [Iterative and Incre](http://melsatar.blog/2018/02/16/the-waterfall-model-a-different-perspective/)mental Method
  + [Agile developmen](https://melsatar.blog/2018/08/27/the-validation-and-verification-model-the-v-model/)t **Waterfall** [**Model**](http://en.wikipedia.org/wiki/Spiral_model) ***Description***

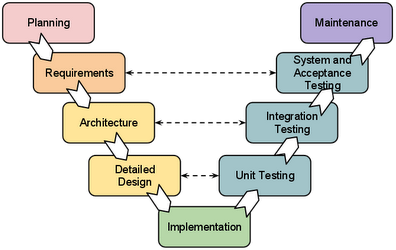
The Waterfall Model is a linear sequential flow. In which progress is seen as flowing steadily downwards (like a waterfall) through the phases of software implementation. This means that any phase in the development process begins only if the previous phase is complete. The waterfall approach does not define the process to go back to the previous phase to handle changes in requirement. The waterfall approach is the earliest approach and most widely kno[wn that was used f](http://melsatar.blog/2018/02/16/the-waterfall-model-a-different-perspective/)or software development.



For projects which not focus on changing the requirements, for example, projects initiated from a request for proposals (RFPs), the customer has very clear documented requirements.

#### SHAPED MODELS

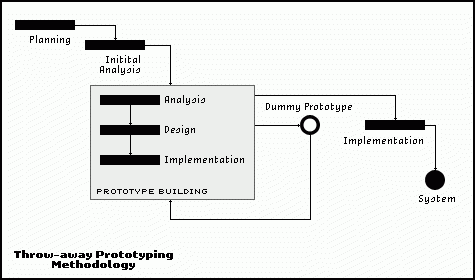
It is an extension of the waterfall model, Instead of moving down in a linear way, the process steps are bent upwards after the implementation and coding phase, to form the typical V shape. The major difference between the V-shaped model and the waterfall model is the early test planning in the V-shaped model.



Software requirements clearly defined and known. Software development technologies and tools are well-known. Simple and easy to use Each phase has specific deliverables. Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle. Works well for where requirements are easily understood. Verification and validation of the product in the early stages of product development.

#### Prototyping Model

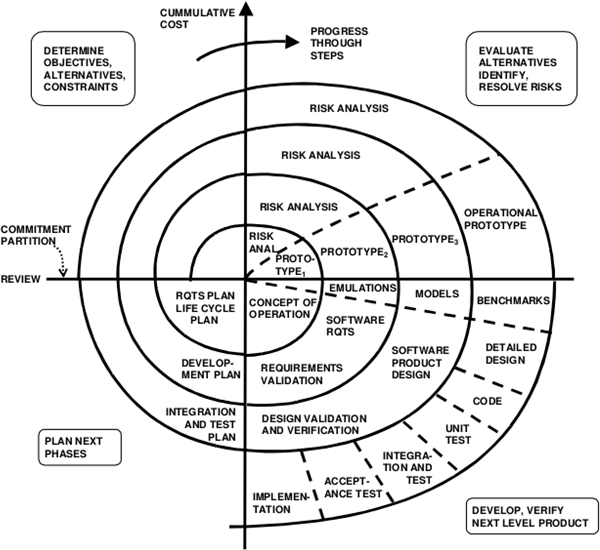
It refers to the activity of creating prototypes of software applications, for example, incomplete versions of the software program being developed. It is an activity that can occur in software development and It used to visualize some components of the software to limit the gap of misunderstanding the customer requirements by the development team. This also will reduce the iterations that may occur in the waterfall approach and are hard to be implemented due to the inflexibility of the waterfall approach. So, when the final prototype is developed, the requirement is considered to be frozen.



#### Spiral Model (SDM)

##### Description

It is combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. This model of development combines the features of the prototyping model and the waterfall model. The spiral model is favored for large, expensive, and complicated projects. This model uses many of the same phases as the waterfall model, in essentially the same order, separated by planning, risk assessment, and the building of pro



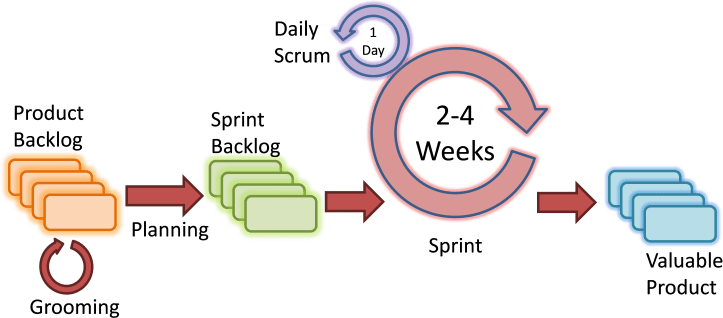
It is used in large applications and systems which built-in small phases or segments.

##### INCREMENTAL AND ITERATIVE MODEL

It is developed to overcome the weaknesses of the waterfall model. It starts with initial planning and ends with deployment with the cyclic interactions in between. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental), allowing software developers to take advantage of what was learned during the development of earlier parts or versions of the system. It can consist of mini waterfalls or mini V-Shaped models.

#### incremental-sdlcAgile Model

It is based on iterative and incremental development, where requirements and solutions evolve through collaboration between cross-functional teams.



**EXPERIMENT NO. 2**

**Aim**: Understanding an SRS.

#### Requirements:

**Hardware Requirements:**

* PC with 300 megahertz or higher processor clock speed recommended; 233 MHz minimum required.
* 128 megabytes (MB) of RAM or higher recommended (64 MB minimum supported)
* 1.5 gigabytes (GB) of available hard disk space
* CD ROM or DVD Drive
* Keyboard and Mouse(compatible pointing device).

**Software Requirements:** Rational Rose, Windows XP, **Theory:**

An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies *at a particular point in time* (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

The SRS document itself states in precise and explicit language those functions and capabilities a software system (i.e., a software application, an eCommerce Web site, and so on) must provide, as well as states any required constraints by which the system must abide. The SRS also functions as a blueprint for completing a project with as little cost growth as possible. The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.

It's important to note that an SRS contains functional and nonfunctional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be.

A well-designed, well-written SRS accomplishes four major goals:

* It provides feedback to the customer. An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems. Therefore, the SRS should be written in natural language (versus a formal language, explained later in this article), in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables, and so on.
* It decomposes the problem into component parts. The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion.
* It serves as an input to the design specification. As mentioned previously, the SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.
* It serves as a product validation check. The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. This information-gathering stage can include onsite visits, questionnaires, surveys, interviews, and perhaps a return- on-investment (ROI) analysis or needs analysis of the customer or client's current business environment. The actual specification, then, is written after the requirements have been gathered and analyzed.

## SRS should address the following

The basic issues that the SRS shall address are the following:

* 1. ***Functionality****.* What is the software supposed to do?
  2. ***External interfaces****.* How does the software interact with people, the system’s hardware, other hardware, and other software?
  3. ***Performance****.* What is the speed, availability, response time, recovery time of various software functions, etc.?
  4. ***Attributes****.* What are the portability, correctness, maintainability, security, etc. considerations?
  5. ***Design constraints imposed on an implementation****.* Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?

## Chracteristics of a good SRS

An SRS should be

1. Correct
2. Unambiguous
3. Complete
4. Consistent
5. Ranked for importance and/or stability
6. Verifiable
7. Modifiable
8. Traceable

**Correct** - This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.

**Unambiguous -** An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them.

**Complete -** A simple judge of this is that is should be all that is needed by the software designers to create the software.

**Consistent -** The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.

**Ranked for Importance -** Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful provide this information in the SRS.

**Verifiable -** Don't put in requirements like - "It should provide the user a fast response." Another of my favorites is

- "The system should never crash." Instead, provide a quantitative requirement like: "Every key stroke should provide a user response within 100 milliseconds."

**Modifiable -** Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.

**Traceable -** Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

## A sample of basic SRS Outline

#### Introduction

* 1. Purpose
  2. Document conventions
  3. Intended audience
  4. Additional information
  5. Contact information/SRS team members
  6. References

#### Overall Description

* 1. Product perspective
  2. Product functions
  3. User classes and characteristics
  4. Operating environment
  5. User environment
  6. Design/implementation constraints
  7. Assumptions and dependencies

#### External Interface Requirements

* 1. User interfaces
  2. Hardware interfaces
  3. Software interfaces
  4. Communication protocols and interfaces

#### System Features

* 1. System feature A
     1. Description and priority
     2. Action/result
     3. Functional requirements
  2. System feature B

#### Other Nonfunctional Requirements

* 1. Performance requirements
  2. Safety requirements
  3. Security requirements
  4. Software quality attributes
  5. Project documentation
  6. User documentation

#### Other Requirements

Appendix A: Terminology/Glossary/Definitions list Appendix B: To be determined

**Conclusion:** The SRS was made successfully by following the steps described above.

Experiment no.3

|  |  |
| --- | --- |
| Title | Make a Use case diagram for railway reservation system/ ATM |
| Objective | It will provide students graphical overview of the functionality provided by the system. |
| Prerequis ite | Knowledge of UML |
| Theory | Use case diagram is a platform that can provide a common understanding for the end-users, developers and the domain experts. It is used to capture the basic functionality i.e. use cases, and the users of those available functionality, i.e. actors, from a given problem statement.  In this experiment, we will learn how use cases and actors can be captured and how different use cases are related in a system.  ***Use case diagrams***  Use case diagrams belong to the category of behavioral diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform one or more actors, and dependencies among them.  Actor  An actor can be defined as an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems. For example, consider the case where a customer *withdraws cash* from an ATM. Here, customer is a human actor.  Actors can be classified as below: |

|  |  |
| --- | --- |
|  | * **Primary actor**: They are principal users of the system, who fulfill their goal by availing some service from the system. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here. * **Supporting actor**: They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.   In a use case diagram primary actors are usually drawn on the top left side of the diagram.  Use Case  A use case is simply functionality provided by a system.  Continuing with the example of the ATM, *withdraw cash* is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases include, *check balance*, *change PIN*, and so on.  Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.  Subject  Subject is simply the system under consideration. Use cases apply to a subject. For example, an ATM is a subject, having multiple use cases, and multiple actors interact with it. However, one should be careful of external systems interacting with the subject as actors.  ***Graphical Representation***  An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as below:  Use case diagram |

|  |  |
| --- | --- |
|  | ***Association between Actors and Use Cases***  A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.  An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Associations among the actors are usually not shown. However, one can depict the class hierarchy among actors.  ***Use Case Relationships***  Three types of relationships exist among use cases:   * Include relationship * Extend relationship * Use case generalization   Include Relationship  Include relationships are used to depict common behavior that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.  *Example*  For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a *login* use case, which is included by *compose mail*, *reply*, and *forward email* use cases. The relationship is shown in figure - 02.  Include relationship  Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.  ***Identifying Actors*** |

|  |  |
| --- | --- |
|  | Given a problem statement, the actors could be identified by asking the following questions :   * Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor) * Who keeps the system working? (This will help to identify a list of potential users) * What other software / hardware does the system interact with? * Any interface (interaction) between the concerned system and any other system?   ***Identifying Use cases***  Once the primary and secondary actors have been identified, we have to find out their goals i.e. what the functionality they can obtain from the system is. Any use case name should start with a verb like, "Check balance".  ***Guidelines for drawing Use Case diagrams***  Following general guidelines could be kept in mind while trying to draw a use case diagram:   * Determine the system boundary * Ensure that individual actors have well-defined purpose * Use cases identified should let some meaningful work done by the actors * Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection * Use include relationship to encapsulate common behavior among use cases , if any |
| Sample Output | Use case diagrams for the chosen project is captured using above mentioned basics |

|  |  |
| --- | --- |
| **Experiment No:3** | |
| Title | **Modeling E-R diagram for the chosen project.** |
| Objective | Students will be able to create a logical design of database for real world objects |
| Prerequis ite | Knowledge of UML |
| Theory | ***Introduction***  Developing databases is a very important task to develop a system. Before going to form exact database tables and establishing relationships between them, we conceptually or logically can model our database using ER diagrams.  In this experiment we will learn how to find the entities, its attributes and how the relationships between the entities can be established for a system.  ***Entity Relationship Model***  Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.  For example, student and school -- they are two entities. Students study in school. So, these two entities are associated with a relationship "Studies in".  As another example, consider a system where some job runs every night, which updates the database. Here, job and database could be two entities. They are associated with the relationship "Updates".  ***Entity Set and Relationship Set***  An entity set is a collection of all similar entities. For example, "Student" is an entity set that abstracts all students. Ram, John are specific entities belonging to this set. Similarly, a "Relationship" set is a set of similar relationships.  ***Attributes of Entity***  Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.  For example, any student has got a name, age, an address. At any given time a student can study only at one school. In the school he would have a roll number, and of course a grade in which he studies. These data are the attributes of the entity set Student. |

|  |  |
| --- | --- |
|  | ***Keys***  One or more attribute(s) of an entity set can be used to define the following keys:   * **Super key:** One or more attributes, which when taken together, helps to uniquely identify an entity in an entity set. For example, a school can have any number of students. However, if we know grade and roll number, then we can uniquely identify a student in that school. * **Candidate key:** It is a minimal subset of a super key. In other words, a super key might contain extraneous attributes, which do not help in identifying an object uniquely. When such attributes are removed, the key formed so is called a candidate key. * **Primary key:** A database might have more than one candidate key. Any candidate key chosen for a particular implementation of the database is called a primary key. * **Prime attribute:** Any attribute taking part in a super key   ***Weak Entity***  An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by it's attributes. In other words, it doesn't have a super key.  For example, consider a company that allows employees to have travel allowance for their immediate family. So, here we have two entity sets: employee and family, related by "Can claim for". However, family doesn't have a super key. Existence of a family is entirely dependent on the concerned employee. So, it is meaningful only with reference to employee.  ***Entity Generalization and Specialization***  Once we have identified the entity sets, we might find some similarities among them. For example, multiple people interact with a banking system. Most of them are customers, and rest employees or other service providers. Here, customers, employees are persons, but with certain specializations. Or in other way, person is the generalized form of customer and employee entity sets.  ER model uses the "ISA" hierarchy to depict specialization (and thus, generalization).  ***Mapping Cardinalities***  One of the main tasks of ER modeling is to associate different entity sets. Let's consider two entity sets E1 and E2 associated by a relationship set R. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:   * **One to one:** An entity in E1 is related to at most a single entity in E2, and vice versa * **One to many:** An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1. * **Many to one:** Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1. |

|  |  |  |
| --- | --- | --- |
| * **Many to many:**   including zero, a  ***ER Diagram***  From a given problem s among different entity s entity-relationship (ER)  ***Graphical Notatio***  **Term** | Any number of entities could be related to any nd vice versa.  tatement we identify the possible entity sets, the ets. Once we have these information, we repres diagram.  ***ns for ER Diagram***  **Notation** | number of entities in E2,  ir attributes, and relationships ent them pictorially, called an  **Remarks** |
| Entity set | Entity | Name of the set is writt |
| Attribute | Attribute | Name of the attribute ellipse |
| Entity with attributes | Entity with attributes | Roll is the primary k underline |
| Weak entity set | Weak entity |  |
| Relationship set | Relationship | Name of the relationsh diamond |
| Related enity sets |  |  |
| Relationship cardinality |  | A person can own zer two persons can own t |
| Relationship with weak entity set  ***Importance of ER***  Figure - 01 shows the d  Given a problem state represent them using an required constraints. Fi | Weak entity relationship  ***modeling***  ifferent steps involved in implementation of a (relational) database.  ment, the first step is to identify the entities, attributes and relationships. We ER diagram. Using this ER diagram, table structures are created, along with nally, these tables are normalized in order to remove redundancy and maintain | |

Database design steps

|  |  |
| --- | --- |
|  | data integrity. Thus, to have data stored efficiently, the ER diagram is to be drawn as much detailed and accurate as possible. |
| Sample Output | download |
| Post Lab Assignm ent (If Any) | Dram E-R diagrams for other projects |

## EXPERIMENT NO. 4

**AIM:** To prepare DATA FLOW DIAGRAM for any project.

**REQUIREMENTS:**

#### Hardware Interfaces

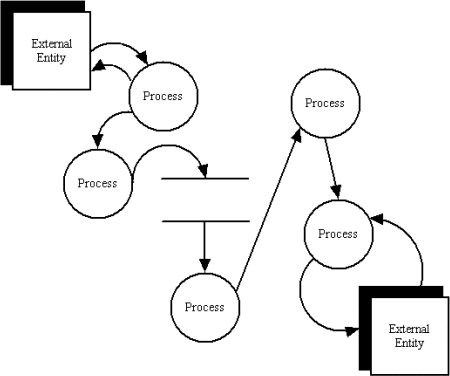
* Pentium(R) 4 CPU 2.26 GHz, 128 MB RAM
* Screen resolution of at least 800 x 600 required for proper and complete viewing of screens. Higher resolution would not be a problem.
* CD ROM Driver

#### Software Interfaces

* Any window-based operating system (Windows 95/98/2000/XP/NT)
* WordPad or Microsoft Word

***THEORY***

Data flow diagrams illustrate how data is processed by a system in terms of inputs and outputs.



#### Data Flow Diagram Notations

You can use two different types of notations on your data flow diagrams: *Yourdon & Coad* or *Gane & Sarson*.

#### Yourdon and Coad Process NotationsProcess Notations

*Yourdon and Coad Process Notations*



#### Process

*Gane and Sarson Process Notation*

A process transforms incoming data flow into outgoing data flow.

#### Datastore Notations

Datastore Notations

**Yourdon and Coad Datastore Notations**

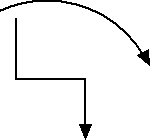
DataStoreNumber DataStore

#### DataStore

**Gane and Sarson Datastore Notations**

Datastores are repositories of data in the system. They are sometimes also referred to as files.

#### Dataflow Notations



**Dataflow**

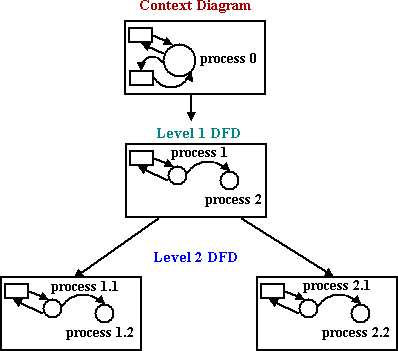
Dataflows are pipelines through which packets of information flow. Label the arrows with the name of the data that moves through it.

HOW TO DRAW DATA FLOW DIAGRAMS (cont'd)

#### Data Flow Diagram Layers

Draw data flow diagrams in several nested layers. A single process node on a high level diagram can be expanded

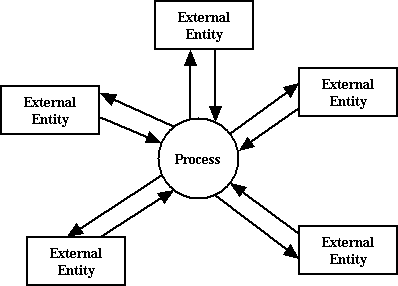
to show a more detailed data flow diagram. Draw the context diagram first, followed by various layers of data flow diagrams.



*The nesting of data flow layers*

#### Context Diagrams

A context diagram is a top level (also known as Level 0) data flow diagram. It only contains one process node (process 0) that generalizes the function of the entire system in relationship to external entities.



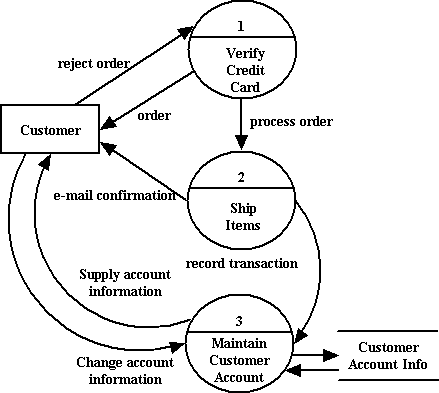
#### External Entity NotationsExternal Entity Notations

**External Entity**

External entities are objects outside the system, with which the system communicates. External entities are sources and destinations of the system's inputs and outputs.

#### DFD levels

The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes until you reach pseudocode.



#### An example first-level data flow diagram

**Conclusion:** The dataflow diagram was made successfully by following the steps described above.

## Experiment 5

### AIM: Draw class diagram. Also identify weak and strong class.

Theory:

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

*Vital components of a Class Diagram*

The class diagram is made up of three sections:

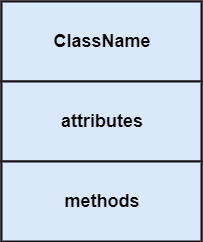
* **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:

1. Capitalize the initial letter of the class name.
2. Place the class name in the center of the upper section.
3. A class name must be written in bold format.
4. The name of the abstract class should be written in italics format.

* **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:

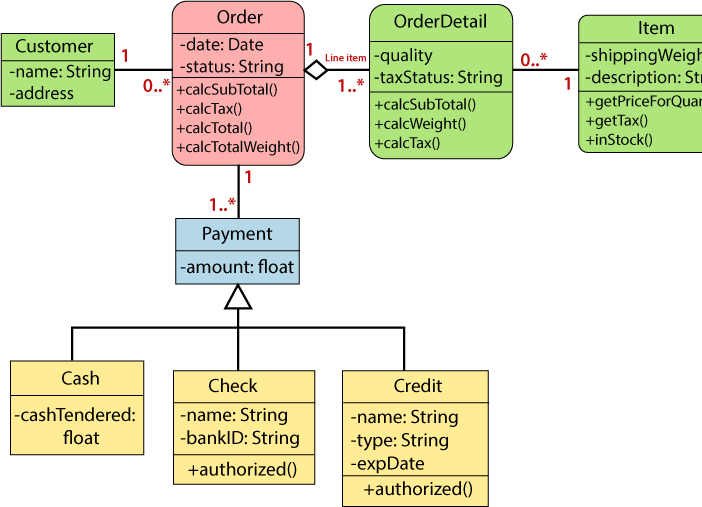
1. The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).
2. The accessibility of an attribute class is illustrated by the visibility factors.
3. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.

* **Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.



A class diagram describing the sales order system is given below. Steps to draw a class diagram

1. To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
2. The objects and their relationships should be acknowledged in advance.
3. The attributes and methods (responsibilities) of each class must be known.
4. A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
5. Notes can be used as and when required by the developer to describe the aspects of a diagram.
6. The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.



## Experiment 6

### AIM: Draw component diagram.

Theory: to create a Component Diagram:

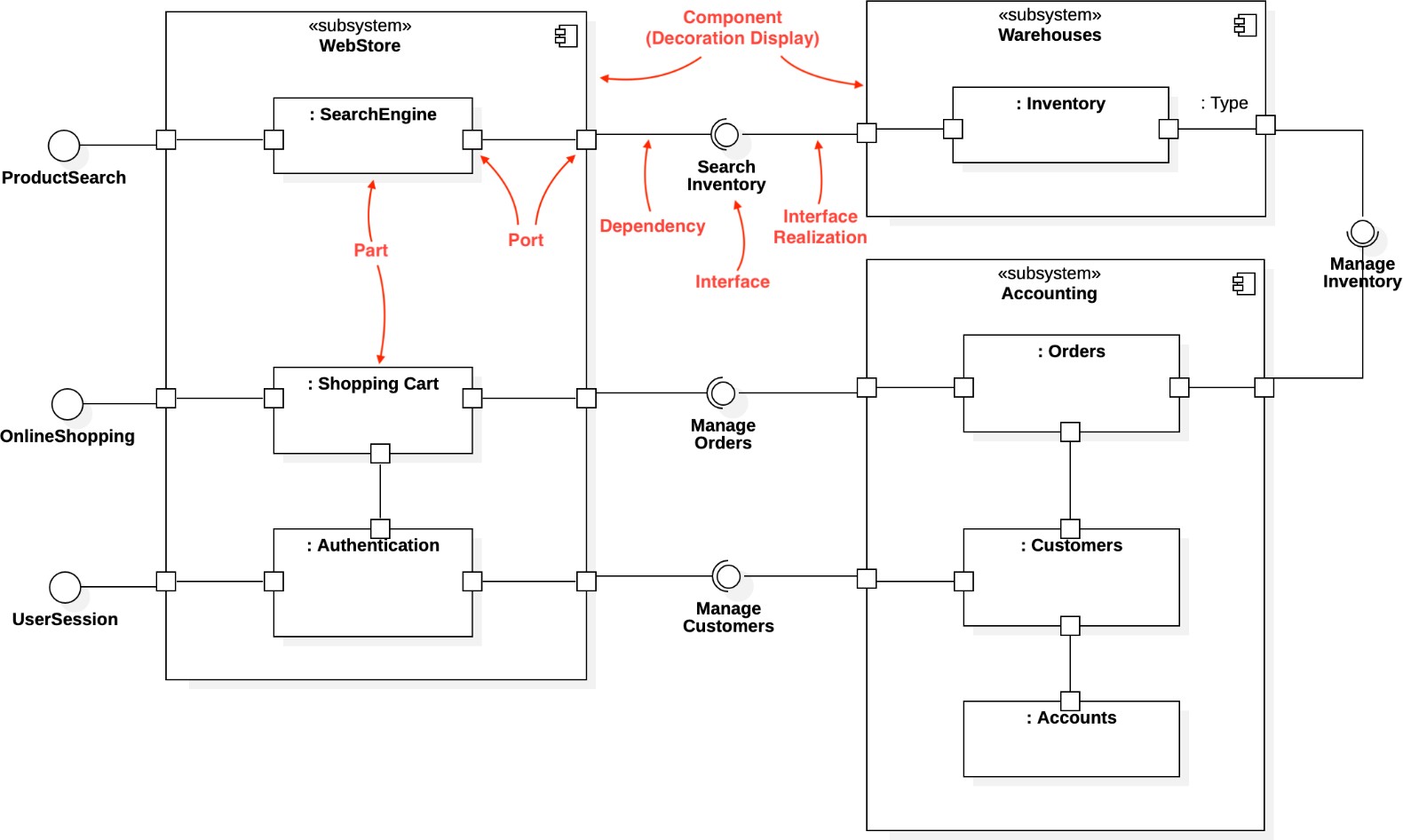
1. Select first an element where a new Component Diagram to be contained as a child.
2. Select **Model | Add Diagram | Component Diagram** in Menu Bar or select **Add Diagram | Component Diagram** in Context Menu.

To create a Component:

1. Select **Component** in **Toolbox**.
2. Drag on the diagram as the size of Component.

To create a Component (model element only) by Menu:

1. Select an Element where a new Component to be contained.
2. Select **Model | Add | Component** in Menu Bar or **Add | Component** in Context Menu



Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

component diagrams are used to describe the physical artifacts of a system. This artifact includes files, executables, libraries, etc

The purpose of this diagram is different. Component diagrams are used during the implementation phase of an application. However, it is prepared well in advance to visualize the implementation details.

Initially, the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.

This diagram is very important as without it the application cannot be implemented efficiently. A well-prepared component diagram is also important for other aspects such as application performance, maintenance, etc.

## Experiment 7

### AIM: Introduction to software testing and quality assurance.

➔Software

Software is a set of instructions and its documentation that tells a computer what to do or how to perform a task. Software includes all different software programs on a computer, such as applications and the operating system.

Examples : Adobe Photoshop, MacOS, Google Chrome, etc.

➔Program

A computer program is a list of instructions that tell a computer what to do. Everything a computer does is done by using a computer program. A computer program is written in a programming language. Programs stored in the memory of a computer enable the computer to perform tasks in sequence or even intermittently.

Example:

#### import random print(‘Random Dice Roll: ’) N = random.randint(1, 6) print(N)

➔Software Engineering

Software engineering is an engineering branch associated with the development of software products using well-defined scientific principles, methods and procedures Objectives of Software Engineering :

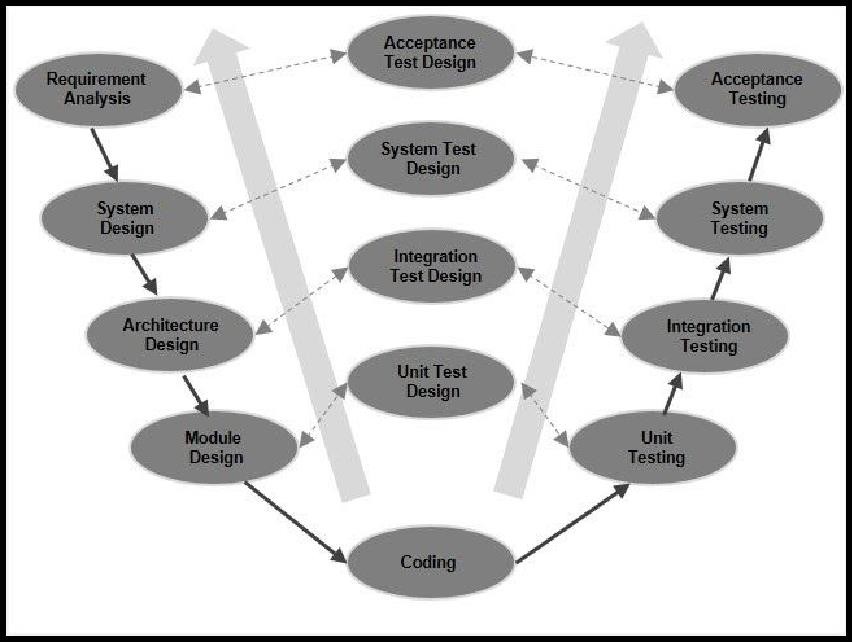
* Maintainability
* Correctness
* Reusability
* Testability
* Reliability
* Portability

➔V-Model

The V-model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as the Verification and Validation model.

The V-Model is an extension of the waterfall model and is based on the association of a testing phase for each corresponding development stage. This means that for every single phase in the development cycle, there is a directly associated testing phase. This

is a highly-disciplined model and the next phase starts only after completion of the previous phase.



➔Software Testing

Software testing is the process of verifying a system with the purpose of identifying any errors, gaps or missing requirements versus the actual requirement. Software testing is broadly categorised into two types - functional testing and non-functional testing.

The process of software testing aims not only at finding faults in the existing software but also finding measures to improve the software in terms of efficiency, accuracy and usability.

➔Software Testing Process

We can divide the activities within the fundamental test process into the following basic steps:

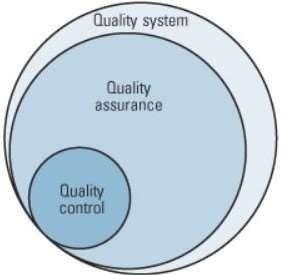
* **Planning and Control**
  + Test planning involves producing a document that describes an overall approach and test objectives. It involves reviewing the test basis, identifying the test conditions based on analysis of test items, writing test cases and Designing the test environment.
* **Analysis and Design**
  + To review the test basis. The test basis is the information on which test cases are based, such as requirements, design specifications, product risk analysis, architecture and interfaces
  + To identify test conditions
  + To design the tests
* **Implementation and Execution**
  + Test execution involves actually running the specified test on a computer system either manually or by using an automated test tool. It is a Fundamental Test Process in which actual work is done.
* **Evaluating exit criteria and Reporting**
  + To assess if more test are needed or if the exit criteria specified should be changed
  + To write a test summary report for stakeholders
* **Test closure**
  + To check which planned deliverables are actually delivered and to ensure that all incident reports have been resolved
  + To finalize and archive testware such as scripts, test environments, etc. for later reuse
  + To handover the testware to the maintenance organization. They will give support to the software

➔Quality Assurance

Quality Assurance is defined as an activity to ensure that an organization is providing the best possible product or service to customers. QA focuses on improving the processes to deliver Quality Products to the customer. An organization has to ensure that processes are efficient and effective as per the quality standards defined for software products.

➔Quality Control

It is a Software Engineering process used to ensure quality in a product or a service. It does not deal with the processes used to create a product; rather it examines the quality of the "end products" and the final outcome. The main aim of Quality control is to check whether the products meet the specifications and requirements of the customer. If an issue or problem is identified, it needs to be fixed before delivery to the customer. QC also evaluates people on their quality level skill sets and imparts training and certifications.



➔Why should we test?

The testing is important since it discovers defects/bugs before the delivery to the client, which guarantees the quality of the software. It makes the software more reliable and easy to use. Thoroughly tested software ensures reliable and high- performance software operation. Without proper testing, we could potentially release software which could malfunction and cause serious injuries.

If software testing is not done, it can lead to major software failures like Year 2000 problem, Patriot Missile Error, etc.

➔Who should do the testing?

It depends on the process and the associated stakeholders of the project(s). In the IT industry, large companies have a team with responsibilities to evaluate the developed software in context of the given requirements.

Professionals who test software include QA Analysts, Test Engineers, Test Analysts, Software Testers, SQA, Quality Assurance, Performance Testers, Usability Testers.

➔What should we test?

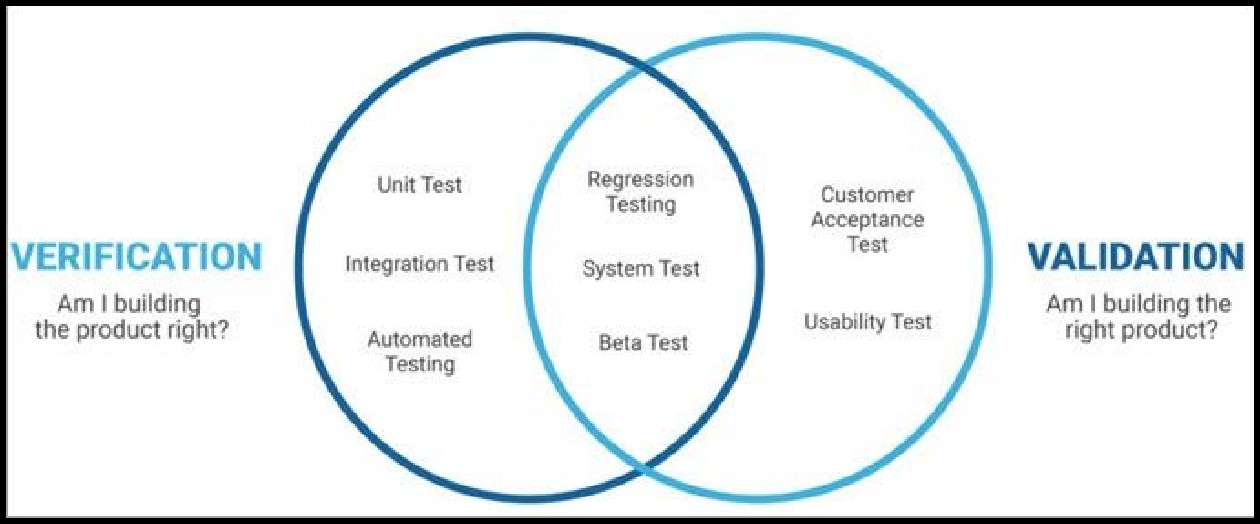
We would want that worse conditions should occur in the very beginning of the project only than in the later phases.

Test the common case of everything you can. This will tell you when that code breaks after you make some change. Test the edge cases of a few unusually complex code that you think will probably have errors. Whenever you find a bug, write a test case to cover it before fixing it. Add edge-case tests to less critical code whenever someone has time to kill.

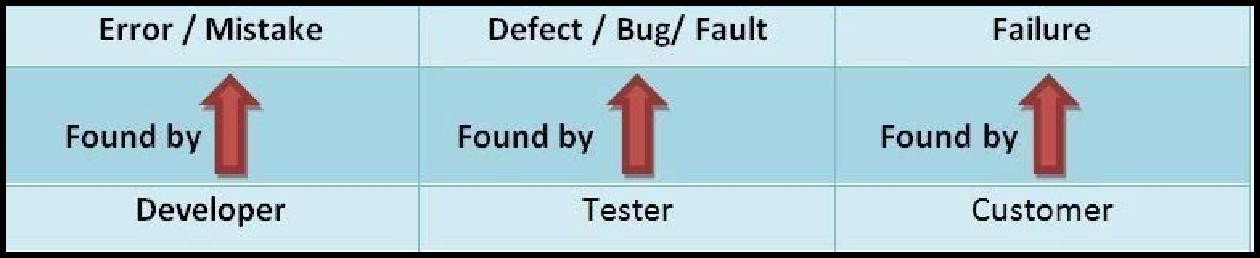
➔Verification and Validation

**Verification** involves the evaluation of artifacts of software development to ensure that the product being developed will comply with its requirements. Verification involves the evaluation of artifacts of software development to ensure that the product being developed will comply with its requirements. It involves activities like document review, test cases review, walk-throughs, inspection etc.

**Validation** involves validation of a developed software product to check if it conforms to the specified business requirements. It involves dynamic testing of a software product by running it. It involves activities like functional testing, automation testing etc.



➔Error, Bug, Fault and Failure

* **Error**
  + Any incorrect human action that produces a problem in the system is called error. A person can make an error (mistake), which can lead to introduction of a defect (fault or bug).
  + Error may occur because of time pressure, human fallibility, Inexperienced or insufficient people on project, miscommunication, technology constraint, complexity of project.
* **Bug**
  + Occurs due to coding error.
  + These are fatal errors that could block a functionality,
  + results in a crash, or cause performance bottlenecks
* **Fault**
  + Fault can be caused just because of error or can be defined as the representation of errors.
  + Problems like an invalid step, lack of resources or inappropriate data definition could cause a fault in a program.
* **Failure**
  + Whenever fault executes failure occur or the deviation identified by the end user while execution is called as failure. When user expectation is not met it is a failure. If a defect in a code is executed that can lead to failure, but not necessarily in all circumstances.

➔Deliverables and Milestones

A **milestone** is a term related to a schedule. Similar to milestones on a road, project milestones tell you where you are in the project schedule. These control points help identify that a number of tasks or key deliverables have been completed allowing you to move on to the next phase of your project.

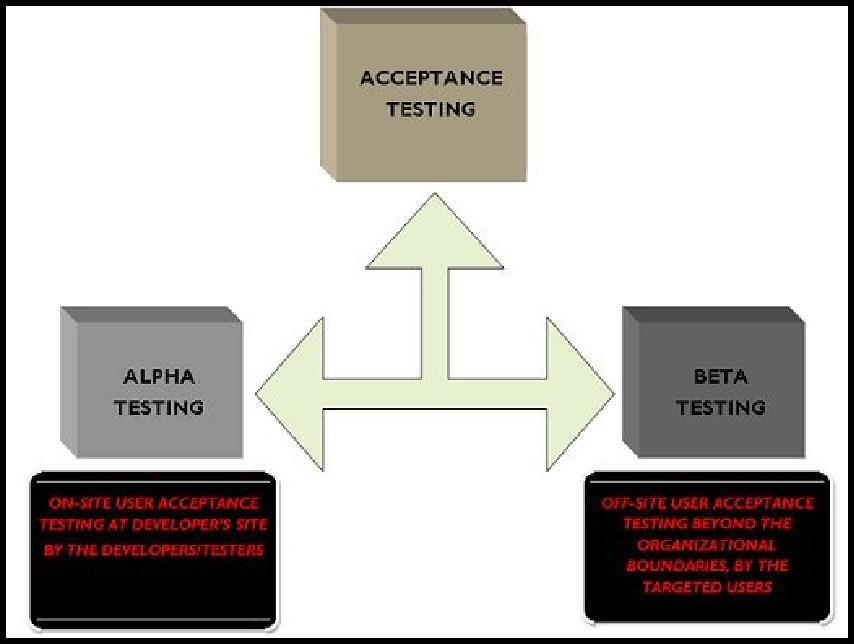
A **deliverable** is just another word for a product that is planned to be supplied to certain project stakeholders for their perusal. It may be an intermediate product for comments, or a final product.

➔Alpha, Beta and Acceptance Testing

* **Acceptance testing** is performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to gauge whether the application meets the intended specifications and satisfies the client’s requirements.
* **Alpha Testing** is a type of software testing performed to identify bugs before

releasing the product to real users or to the public. Unit testing, integration testing and system testing when combined are known as alpha testing.

* **Beta Testing** is performed by real users of the software application in a real

environment. In beta testing a sample of the intended audience tests the application. Beta testing is also known as pre-release testing.

➔Quality and Reliability

**Quality** is the degree to which something is fit for purpose or how well something performs its functions. For example, a high speed train that is fast, energy efficient, safe, comfortable and easy to operate might be considered high quality.

**Reliability** is defined as the probability that a product, system, or service will perform its intended function adequately for a specified period of time, or will operate in a defined environment without failure.. For example, a high speed train that is durable for 20 years and remains safe in high winds and earthquakes.

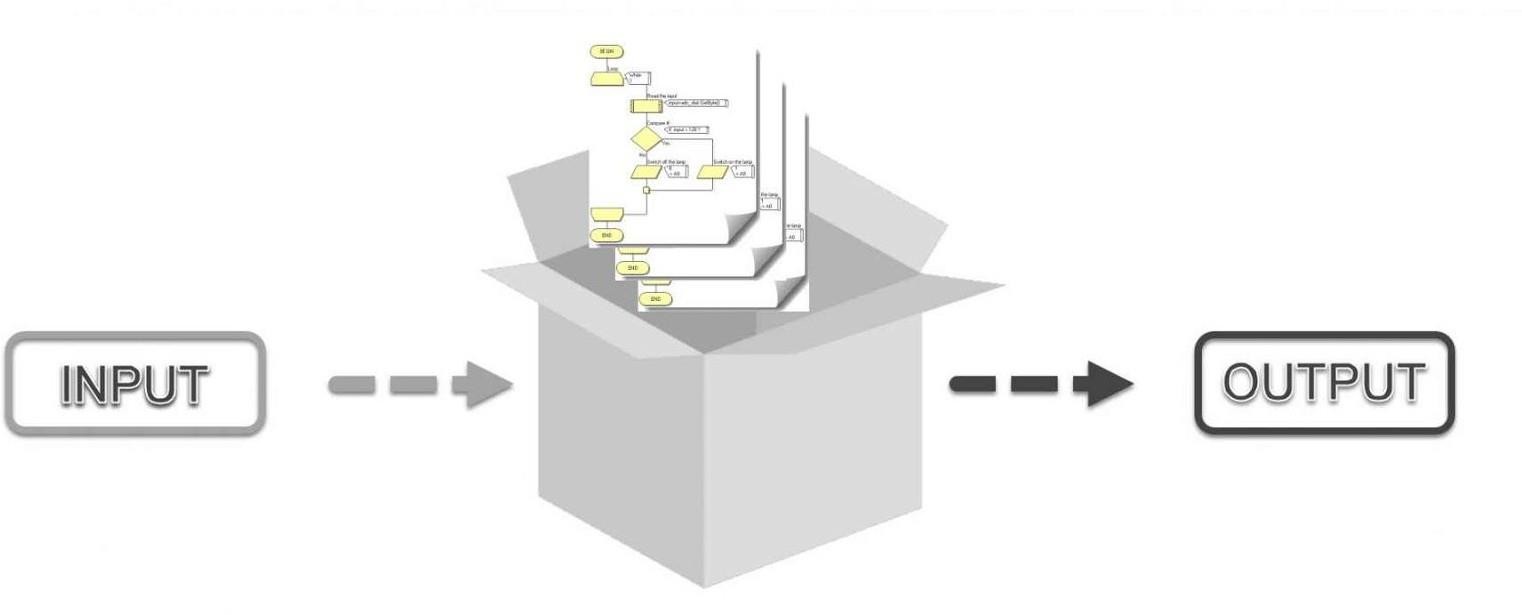
➔Static and Dynamic Testing

* **Static Testing** is a type of testing in which the code is not executed. It can be done manually or by a set of tools. This type of testing checks the code, requirement documents and design documents and puts review comments on the work document.
* **Dynamic testing** is done when the code is in operation mode. Dynamic testing is performed in a runtime environment. When the code being executed is input with a value, the result or the output of the code is checked and compared with the expected output.

➔White box and Black box testing

**White box** testing is when we test the internal structure of a software module: the code itself. White-box testing is also called glass testing or open-box testing. It is usually performed by the team members who know the code, usually developer. Since the developers have an in-depth understanding of the project code, they are capable of making the changes in the source code easily and in a small time. Techniques included in white-box testing :

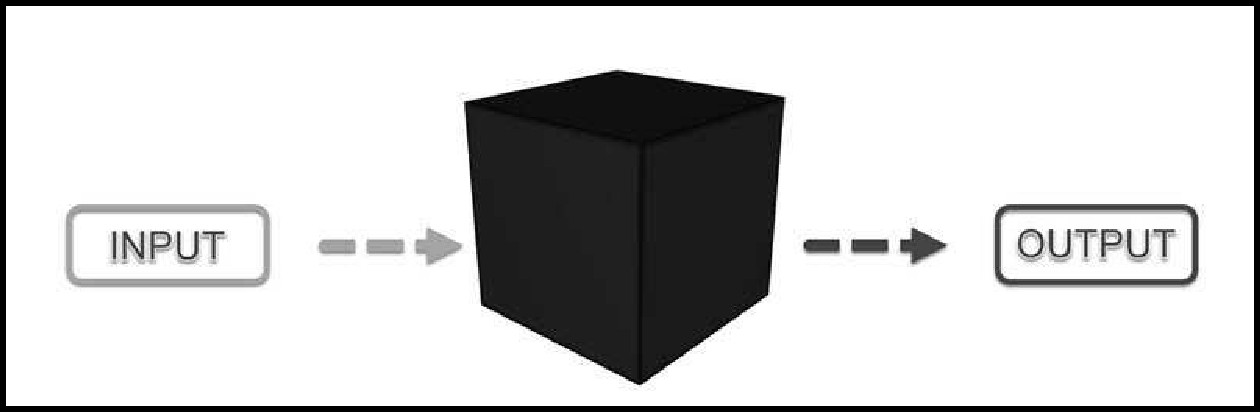
* Control Flow Testing
* Data Flow testing
* Statement coverage
* Path testing



**Black box** testing is a type of testing in which the tester only focuses on the inputs and the expected outputs, without knowing how the application works internally and how

these inputs are processed. Tester treats the Application Under Test (AUT) as a black box.

Techniques included in black-box testing :

* Regression Testing
* Functional Testing

➔Why is 100% testing not possible?

100% testing is often not possible due to the following reasons:

* The domain of possible inputs of a program is too large to be completely used in testing a system. There are both valid inputs and invalid inputs.
* The program may have a large number of states. There may be timing constraints on the inputs, that is, an input may be valid at a certain time and invalid at other times. An input value which is valid but is not properly timed is called an inopportune input.
* The input domain of a system can be very large to be completely used in testing

a program.

* The design issues may be too complex to completely test. The design may have included implicit design decisions and assumptions. For example, a programmer may use a global variable or a static variable to control program execution.
* It may not be possible to create all possible execution environments of the system. This becomes more significant when the behavior of the software system depends on the real, outside world, such as weather, temperature, altitude, pressure, and so on.

➔Limitations of testing

Limitations to software testing depend upon many factors:

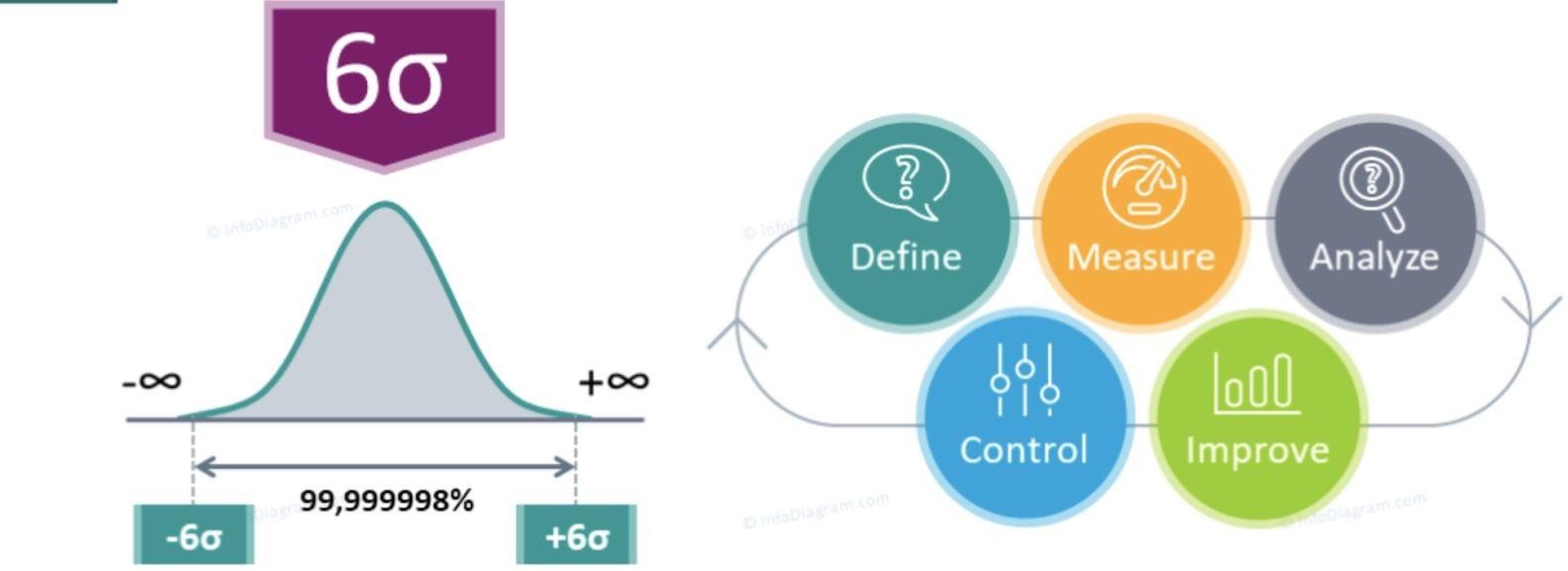
* **Money** - Software testing requires a considerable amount of money in order to upkeep a product.
* **Time constraint** - Testing is a huge part of the software development process and requires a considerable amount of time to assure the quality of a product.
* **Number of Resources** - This is related to time constraint. More testers and human resources are needed to meet a deadline.
* **Dedicated Staging Environment** - It is a tough job to maintain a stage environment or QA environment as close as possible to the Production environment.
* **How much to automate?** - You cannot automate 100% business process. So a

thorough evaluation is needed on that part.

* **False Positives and False Negatives** - This is in respect to automation testing. False positive is a case where in spite of a bug the automation script yields a positive result. A false negative is vice-versa.

➔Six Sigma Concept

Six Sigma is a disciplined, statistical-based, data-driven approach and continuous improvement methodology for eliminating defects in a product, process or service. Sigma represents the population standard deviation, which is a measure of the variation in a data set collected about the process. If a defect is defined by specification limits separating good from bad outcomes of a process, then a six sigma process has a process mean (average) that is six standard deviations from the nearest specification limit.



## Experiment 8

Aim:

A. WAP in C/C++ to find the area of a circle, Triangle, Square and Rectangle and perform equivalence class testing.

Algorithm:

1. Display Menu with choices for Circle, Triangle, Square, Rectangle
2. Take user input
3. If choice=1, take radius as input and calculate area of circle
4. Else if choice=2, take height and base of triangle as input and calculate area of triangle.
5. Else if choice=3, take side of square as input and calculate area of square.
6. Else if choice=4, take height and width of rectangle as input and calculate area of rectangle.
7. Else if choice=5, STOP.
8. Else, display wrong choice

Code:

#include<iostream> using namespace std;

void menu()

{

cout << endl << "1. Area of circle" << endl; cout << "2. Area of triangle" << endl; cout << "3. Area of square" << endl; cout << "4. area of rectangle" << endl; cout << "5. Exit" << endl;

}

void result(int choice)

{

switch (choice) { case 1: {

int r;

cout << "Enter radius" <<endl; cin >> r;

if ((r<1)||(r>100)) {

cout << "Invalid range" << endl; break;

}

cout << "Area of circle is: " << 3.14\*r\*r << endl; break;

}

case 2: {

int h, b;

cout << "Enter base and height" << endl; cin >> b >> h;

if (((h<1) || (h>100)) || ((b<1) || (b>100))) { cout << "Invalid range" << endl;

break;

}

cout << "Area of triangle is " << 0.5\*b\*h << endl; break;

}

case 3:{

int s;

cout << "Enter side:" << endl; cin >> s;

if ((s<1) || (s>100)) {

cout << "Invalid range"<<endl; break;

}

cout << "Area of square " << s\*s << endl; break;

}

case 4: {

int h,w;

cout << "Enter width and height" << endl; cin >> w >> h;

if (((h<1) || (h>100)) || ((w<1) || (w>100))) { cout << "Invalid range" << endl;

break;

}

cout << "Area of rectangle is " << w\*h << endl; break;

}

case 5: break; default: {

cout << "Wrong input" <<endl;

}

}

}

int main()

{

int choice; do {

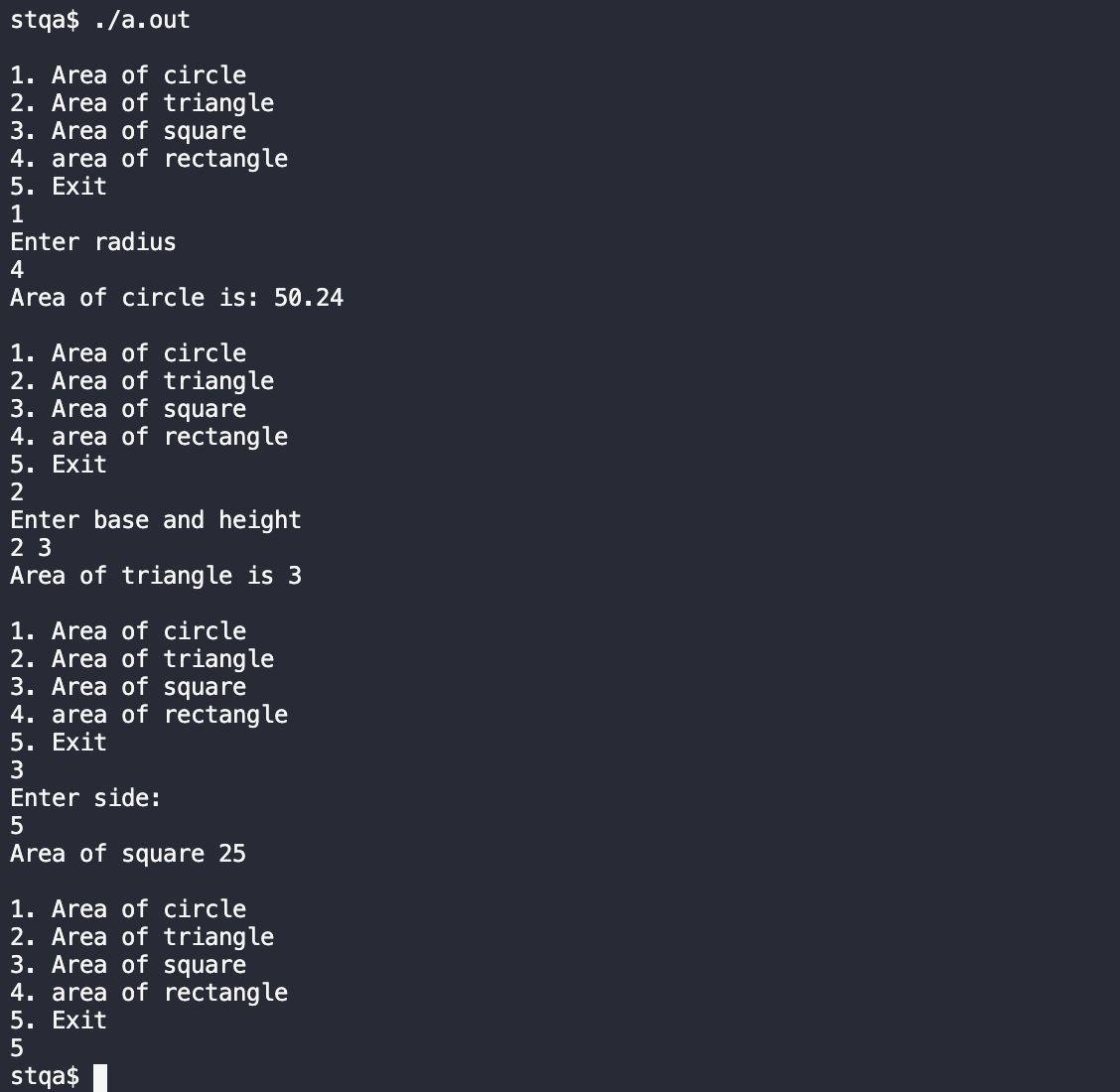
menu();

cin >> choice; result(choice);

} while (choice != 5);

return 0;

}

Output

Equivalence Class Testing:

➔Case 1: Circle

* Input Domain

|  |  |
| --- | --- |
| Input Class | Domain |
| I1 | r: r<=0 |
| I2 | r: r>100 |
| I3 | r: 1 <= r <= 100 |

* Output Domain

|  |  |
| --- | --- |
| Output Class | Domain |
| O1 | Circle if 1<=r<=100 |
| O2 | Not a circle if r<0 |

* Test Cases

|  |  |
| --- | --- |
| r | Expected Output |
| 0 | Invalid input |
| 5 | 50.24 |
| 101 | Invalid input |

➔Case 2: Triangle

* Input Domain

|  |  |
| --- | --- |
| Input Class | Domain |
| I1 | h: h<=0 |
| I2 | h: h>100 |
| I3 | h: 1<= h <= 100 |

|  |  |
| --- | --- |
| I4 | b: b<=0 |
| I5 | b: b > 100 |
| I6 | b: 1<=b<=100 |

* Output Domain

|  |  |
| --- | --- |
| Output Class | Domain |
| O1 | Triangle if h,b >0 |
| O2 | Not a triangle otherwise |

* Test Cases

|  |  |  |
| --- | --- | --- |
| H | B | Expected Output |
| 0 | 8 | Invalid Input |
| 2 | 3 | 3 |
| 101 | 8 | Invalid input |
| 2 | 0 | Invalid input |
| 3 | 2 | 3 |
| 3 | 101 | Invalid input |

➔Case 3: Square

* Input Domain

|  |  |
| --- | --- |
| Input Class | Domain |
| I1 | s: s<=0 |
| I2 | s: s>100 |
| I3 | s: 1 <= s <= 100 |

* Output Domain

|  |  |
| --- | --- |
| Output Class | Domain |
| O1 | Square if 1<=r<=100 |
| O2 | Not a square if r<0 |

* Test Cases

|  |  |
| --- | --- |
| s | Expected Output |
| 0 | Invalid input |
| 5 | 25 |
| 101 | Invalid input |

➔Case 4: Rectangle

* Input Domain

|  |  |
| --- | --- |
| Input Class | Domain |
| I1 | h: h<=0 |
| I2 | h: h>100 |
| I3 | h: 1<= h <= 100 |
| I4 | b: b<=0 |
| I5 | b: b > 100 |
| I6 | b: 1<=b<=100 |

* Output Domain

|  |  |
| --- | --- |
| Output Class | Domain |
| O1 | Rectangle if h,b >0 |
| O2 | Not a rectangle otherwise |

* Test Cases

|  |  |  |
| --- | --- | --- |
| h | b | Expected Output |
| 0 | 8 | Invalid Input |
| 2 | 3 | 6 |
| 101 | 8 | Invalid input |
| 2 | 0 | Invalid input |
| 3 | 2 | 6 |
| 3 | 101 | Invalid input |

# EXPERIMENT NO. 9

Perform software development and software testing in e commerce.

Today, in the new economy, while the possibilities for software are seemingly limitless, so is the growing demand. The problem is that speed and quality have typically been opposing forces in software development, and they still are. In the past, businesses could sacrifice software quality to respect the deadlines, or compromise on software features to meet time-to-market deadlines. In the new Internet economy, it have no choice: the software developers must produce higher quality software. For this reason in the paper, we described the key points of software development process in the new economy, trends and major characteristics, the evolution of this process. In particular, the ecommerce system development methodologies which resolve the problems of e-business applications.

Technology still drives the new economy, the e-economy. The difference is, the stakeholders in the new economy have remembered how important profitability is. The Internet - with the software and communications technologies that drive it - is changing everything. But, software is everywhere, from the cars we drive to military equipment (today, 80% of a fighter jet's capability is performed by software, compared to 6% in 1960); to the cell phones that are allowing us more freedom and mobility. As software is becoming more present in more places, so too is the Internet impacting every business in this e-business world. become For businesses the world over, the Internet continues to drive more change and more opportunity. Today, in the new economy, to be competitive, every business must embrace the technologies of its customers, its supply chain, and its partners. Software is increasingly used as the basis of this connection. It has made so many surveys in this direction and the conclusions are: - Software will continue to become increasingly complex. - The new economy requires software that can be continuously updated. - There are unlimited possibilities for software's growth. Grady Booch has often said, "We cannot reduce complexity in how we build software. The best we can do is manage it." Today, the software developers are pressured to deliver more complex applications that require more mastery of more computing concepts and techniques than ever before. In building for the Web, the trend is toward even more concurrent, more distributed, and more connected applications. In the past, most new software systems were developed by an internal IT organization, and the predominant value of software was in reducing the cost of doing business. Today, the new demands are 2 competition, revenue and ever-faster cycles of development and upgrade. Indeed, the Internet has revolutionized the ability to both include the user during development and deploy software products transparently to a broad user base. This need to support continuously evolving systems has driven today's software development processes away from the traditional sequence of waterfall development activities toward a highly iterative model. In the past, many software domains drew a distinct line between development and maintenance, but future software projects (legacy system upgrades, new applications, or some combination of the two) will not differentiate much between development and maintenance. Iterative development and the Internet are also driving software engineering toward a more homogeneous software management approach. This includes process frameworks, advanced requirements and design notations, and Web-based architectural pa

*EXPERIMENT-10*

#### Aim:- a) Study of Any Test Management Tool (QA Complete).

**Introduction:** Take the guess work out of the software delivery lifecycle. Provide your QA and development teams with the power to collaborate, track project progress, and report on requirements, test cases, and defects.

QA Complete allows you to take a strategic approach to testing by prioritizing key test functions, accounting for risk, planning for coverage, and controlling test execution. Employing effective test case management helps you ensure you’re running the right tests, and thus avoid releasing an application that is not customer-ready.

*Key Features:*

#### Test Case Management:

The ability to organize, plan, and analyze your testing efforts across the lifecycle is critical to your success or failure whether you use manual or automated test cases today. As projects cope with fewer development resources, higher quality expectations, and shorter development timelines, any serious development effort needs better test case management. QA Complete delivers.

* Manage manual test cases and link them back to the original requirements, thereby ensuring a requirement has been met. Evaluate the test run history of those automated tests right from QA Complete.
* Employ re-usable manual test libraries to quickly create new test scenarios.
* Graphically report automated test runs with plug-ins for many leading automated testing tools, including Test Complete and HP Quick Test Pro (QTP).
* Organize your test library any way you like: by component, functional area, release, or Agile sprint.
* Add, print, edit, or email test cases with a single click.

*Test Automation Tool Integrations:*

QA Complete supports many automated testing tools, including Automated QA Test Complete and HP Quick Test Pro. Integration with test automation tools allows you review the run history of any automated test on any machine, so if you have a test lab with multiple machines running automated tests, you can compare machine run history. Since you can co-ordinate both manual and automated tests, you have better test information to make release decisions.

By integrating automated testing into QA Complete, you can:

* Launch the tests from within your automated tool and automatically report the run information to QA Complete for analysis of runs over time.
* Trend results using graphical dashboards and schedule tests to run unattended.

*Bi-Directional Traceability:*

The goal of traceability is to ensure “adequate” test coverage for each software requirement. It is important to maintain traceability both forwards and backwards, from requirement to test case

and from test case to requirement. This ensures that design specifications are appropriately verified

The software development process of ecommerce systems IT professionals, web designers, engineers, IT managers or executives, everyone must understand how to apply software engineering concepts for ecommerce systems, for better integrate their software with the needs of their business. IT professionals need to establish a methodology that resolve all the problems of e-business applications. Those software developers who understand the concepts of analysis, architecture and design, will have the success in developing ecommerce systems. It is important to understand the ecommerce systems that a new and unique form of software development which must not be build with the same methodologies necessary to build any information system. The development of successful ecommerce systems depends on powerful analysis, design and implementation. The software developers must be able to understand the customer needs, provide user interface requirements, establish security, network architecture, ensure integration with legacy systems. Standard methodologies tend to depend on the existence of homogeneous conditions in business, but this fact didn’t exist for ecommerce systems. Because it doesn’t speak about the information system analysis or design without the approach of software life cycle; for ecommerce systems, IT specialists must use a life cycle witch integrates creative design, advertising, marketing concepts (distinctive characteristics of the Web) and software engineering requirements. This life cycle must combine the traditional software life cycle with the spiral life cycle. The spiral approach develops each component of the total project independent, as such should have its own life cycle, each component can be middleware component, objects or reusable applications. The traditional approach is used for analysis, design or implementation. In this way, it takes the best from both cycles.

Testing is crucial in ECommerce?

Testing is crucial to e-commerce because e-commerce sites are both business-critical and highly visible to their users; any failure can be immediately expensive in terms of lost revenue and even more expensive in the longer term if disaffected users seek alternative sites. Yet the time pressures in the e-commerce world militate against the thorough testing usually associated with business criticality, so a new approach is needed to enable testing to be integrated into the development process and to ensure that testing does not present a significant time burden.

**Experiment no 10**

DRAW Activity diagram.

Theory

Activity diagram is another important diagram in UML to describe the dynamic aspects of the system.

Activity diagram is basically a flowchart to represent the flow from one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched, or concurrent. Activity diagrams deal with all type of flow control by using different elements such as fork, join, etc

*How to Draw an Activity Diagram?*

Activity diagrams are mainly used as a flowchart that consists of activities performed by the system. Activity diagrams are not exactly flowcharts as they have some additional capabilities. These additional capabilities include branching, parallel flow, swimlane, etc.

Before drawing an activity diagram, we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities, we need to understand how they are associated with constraints and conditions.

Before drawing an activity diagram, we should identify the following elements −

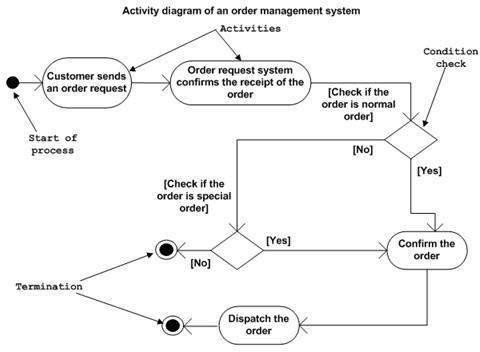
* + Activities
  + Association
  + Conditions
  + Constraints

Once the above-mentioned parameters are identified, we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

Following is an example of an activity diagram for order management system. In the diagram, four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and is mainly used by the business users

Following diagram is drawn with the four main activities −

* + Send order by the customer
  + Receipt of the order
  + Confirm the order
  + Dispatch the order



After receiving the order request, condition checks are performed to check if it is normal or special order. After the type of order is identified, dispatch activity is performed and that is marked as the termination of the process.