SOFTWARE ENGINEERING

UNIT-1
MCA20205--------SYLLABUS

INTRODUCTION TO SOFTWARE ENGINEERING: The evolving role of software, changing nature of software, software myths.

A GENERIC VIEW OF PROCESS: Software engineering- a layered technology, a process framework, the capability maturity model integration (CMMI), process patterns, process assessment, personal and team process models.

PROCESS MODELS: The waterfall model, incremental process models, evolutionary process models, the unified process.

INTRODUCTION TO SOFTWARE ENGINEERING

❖ INTRODUCTION

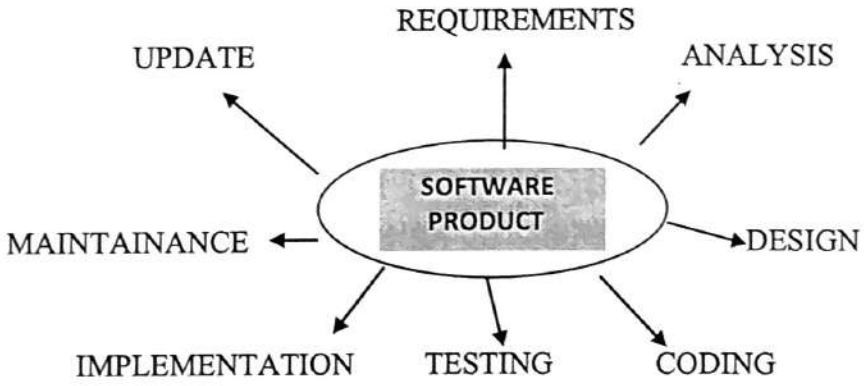
Let us first understand what software engineering stands for the term is made of two words. They are SOFTWARE and ENGINEERING.

- ***SOFTWARE:***

Software is considered to the collection of executable programming code associated with libraries and engineering documentations.

- ***ENGINEERING:***

It is all about developing products using well defined scientific principles and methods and procedures.



1. Evolving role of software:

"Software engineering is a systematic approach towards the analysis, design, development, testing and maintenance of software". The main objective of software engineering is to develop low cost, high quality software product with in a scheduled time.

❖ Evolving of software:

Software Engineering was introduced as a disciplined to provide a well-defined approach for developing the software using high level languages like

"FORTRAN", "COBOL" etc. High level languages were invented in late 1950's for better communication with the computer and for developing useful software.

In olden days projects development considered as a tightly coupled procedures. Generally, project is a collection of "modules" or individual building blocks or components. In tightly coupled development modules are interdependent. To overcome this problem, In Object oriented languages were invented loosely coupled procedure is considered. Here modules are independent.

Large software projects were extremely difficult to design without a software engineering discipline, as it requires very large team of software developers. The development of large software projects was similar to the development of other engineering systems such as Factories, oil Refineries and Ship Management system.

The main purpose of the software engineering is how to manage the different phases of the development of software project.

According to "FRED BROOKS". There are two types of difficulties in developing the software namely,

- ACCIDENTAL
- ESSENTIAL.

Accidental difficulties are those difficulties which occur due to the problems that arise by using tools and technologies.

Essential difficulties are those difficulties which occur due to lack of proper software engineering, such as improper designing and improper methods used for developing software projects.

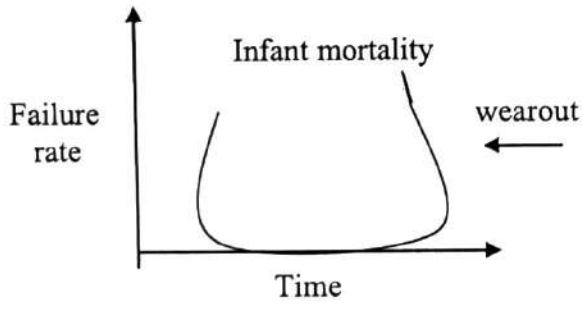
2. SOFTWARE AND ITS CHARACTERISTICS:

Software is defined as computer program which on execution provides the desired functionality. The characteristics are distinguishes to hardware and software. The characteristics of software are:

- The first characteristic that distinguishes from hardware is that software is developed or engineered, but not manufactured.

  Like Hardware the quality of software is achieved through a good design. However, the software is not characterized by manufacturing the errors. If errors are occurred during software development. It can be easily corrected.
- The second characteristic of software is does not wear out, with time as the hardware does.

- **Hardware curve:**

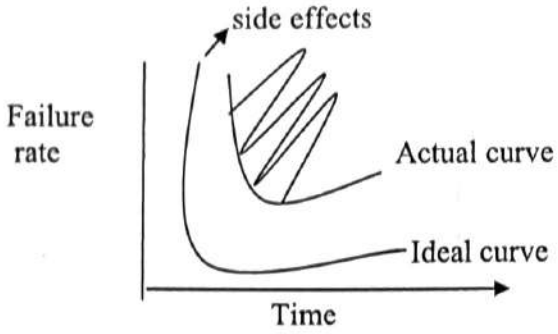


The above figure shows the failure rate is high during the early phase of development, Because of manufacturing defects. After the defects being corrected the failure rate drops very low to a constant level. As the time passes, it again starts raising due to heat, dust and environmental melodies.

❖ **Software Curve:**

Software is not subjected to wear as that of as the hardware. It should follow the ideal curve as shown in below figure. But in actual curve the process is initially different during its life span the software has to undergo changes for maintain purpose.

❖ *Software Curve*



Gradually the level of minimum failure rate for the software begins to use and software gets "deteriorated".

❖ The third characteristic of software, it can be customized according to the user requirements. A software component is designed and implemented in such a way that can be reused in another programs.

3. The Nature of software:

The nature of software medium has many consequences for systems engineering (SE) of software-intensive systems. Four properties of software, taken together, differentiate it from other kinds of engineering artifacts. These four properties are:

1. Complexity
2. Conformity
3. Changeability
4. Invisibility

- **Complexity:** the complexity of software arises from the large number of unique interacting parts in a software system. The parts are unique because they are encapsulated as functions, subroutines or objects, and invoked as needed rather than being replicated.

  Complexity can hide the defects that may not be discovered easily, thus requiring significant additional and unplanned rework.

- **Conformity:** software must conform to exacting specifications in the representation of each of its parts, in the interfaces to other internal parts, and in the connections to the environment in which it operates.
- **Changeability:** software is the most malleable (easily changed) element in a software intensive system; it is the most frequently changed element.
- **Invisibility:** software is said to be invisible because it has no physical properties, while the effects of executing software on a digital computer are observable, software itself cannot be seen, tasted, smelled, touched, or heard.

4. Software Myths:

There are some misbelieves in the software industry about the software and process of building software. For any software developer it is must to know such beliefs and reality about them.

Myths --> misleading attitudes of people --> serious problems in software production

The Myths are classified into 3 categories. They are
1. Management Myths
2. Customer Myths
3. Practitioner Myths

1. **Management Myths:-**In the management, the managers are responsible for the development of software. Some of myths of managers are

(i)Myth:- The management thinks "We already have books with full of standards and procedures for developing software?
Reality:-Even though we have all standards and procedures with us for helping the developer to develop a software. But it is not possible for software professionals to develop a desired product with required quality.

(ii)Myth:-Add more people or programmers to meet deadline of the project.

Reality:-Adding more people in order to catch the schedule will cause the reverse effect on the software project that is software project will get delayed. Because, we have to spend more time on educating people or informing them about the project.

2. Customer Myths:- [A customer who requests computer software may be a person at the next desk, *an outside company* that has requested software under contract, , an in-house group ,a marketing or sales group.]Customer myths --lead to false expectations (by customers) and ultimately, dissatisfaction with the developers.

(i)Myth:- We can start writing the program by using general problem statements only. Later we can add more required functionalities in the program.
Reality:-It is not possible each time to have comprehensive problem statement. We have to start with general problem statements; however by proper communication with

customer the software professionals can gather useful information. The most important thing is that the problem statement should be unambiguous to begin with.

(ii)Myth:- "Project requirements continually change, but change can be easily accommodated because software is flexible."

Reality: It's true that software requirement change, but the impact of change varies with the time at which it is introduced. When requirement changes are requested early(before design), cost impact is relatively small. However, as time passes, cost impact grows rapidly .Similarly the additional resources and more design modifications may be demanded by the software.

3.Practitioner's Myths:-[The practitioner may be Planning group- System analysts, system architects, Development group- Software engineers ,Verification group-Test engineers, quality assurance group, Support group- Customer supporters, technical supports, Marketing/sales- Marketing people and product sales .]

(i)Myth:- "Once we write the program and get it to work, our job is done."
Reality:-Even though we obtain that the program is running major part of work is after delivering it to the customer.

(ii)Myth:- The success of project depends only on the Quality of the product.
Reality:- quality is not only the factor , that makes the project successful , instead of documentation, SCM also plays a prominent role in the successful project.

(iii)Myth:-There is no need of documenting the software project; it unnecessarily slows down the development process.
Reality:-Documenting the software project helps in establishing ease in use of software. It helps in creating better quality. Hence documentation is not wastage of time but it is a must for any software project.

A GENERIC VIEW OF PROCESS

5. Software engineering layered technology:

The software engineering is a systematic approach to the development, maintenance and retirement of the software. Software Engineering must rest on an organizational commitment to **quality**, which leads to continued improvements in the software engineering process. The Software Engineering can be viewed as a layered technology. The main layers are

1. Process layer 2. Methods layer 3.Tools layer

Figure 1 Software engineering layers

1. **Process layer:-** The foundation for software engineering is the process layer. A Process (set of activities) defines a framework for a set of *key process areas* that must be established for effective delivery of software. The Key process areas form the basis for management control of software projects. These include tasks such as:
   - Determining Deliverables
   - Establishing milestones
   - Software Quality Assurance
   - Software Configuration (change) Management

2. **Methods layer :-** It provides technical knowledge for developing software. This layer includes a wide array of tasks such as:
   - Requirements Analysis
   - Design
   - Program Construction
   - Testing
   - Maintenance

3. **Tools Layer:-** It provides computerized or semi-computerized support for the process and method layer. When tools are integrated, the information created by one tool can be used by another tool.. This multi-usage is commonly referred to as computer-aided software engineering or CASE. CASE combines software, hardware and software engineering database. CASE helps in application development including analysis, design, code generation, debugging and testing etc.

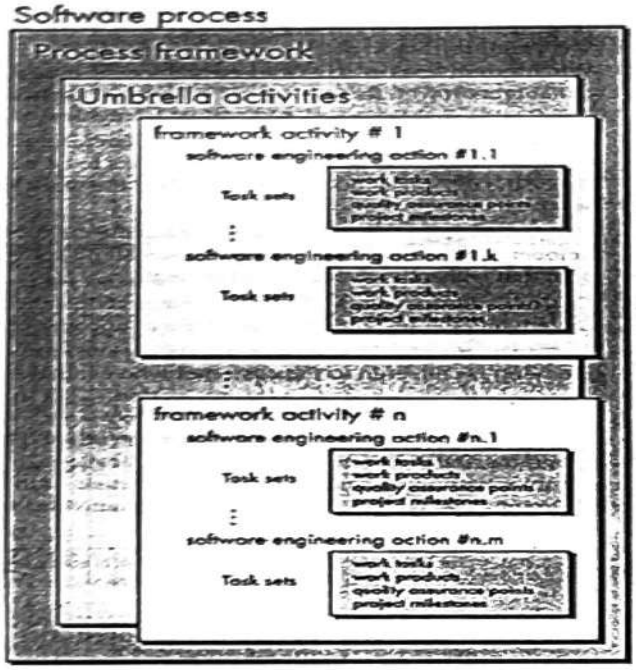
4. **Quality:**

   This layer defines the quality of the project .The output or work product developed after each phase of the software process is validated to ensure the quality. Finally the product achieves the international quality standards for their development.

6. Process framework:

A process framework establishes the foundation for a complete software process by identifying a small number of **framework activities** that are applicable to all software projects. In addition to framework activities, the process also contains a set of **umbrella**

activities that are applicable across the entire software process. Each framework activity is populated by a set of software engineering actions. Each action is populated with individual work tasks that accomplish some part of the work implied by the action. The software process framework is shown as:



The following generic process framework is applicable to the majority of software projects:

- **Communication.** This framework activity involves heavy communication and Collaboration with the customer (and other stakeholders) and contains Requirements gathering and other related activities.
- **Planning.** This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- **Modeling.** This activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.
- **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
- **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems.

7. CMMI (CAPABILITY MATURITY MODEL INTEGRATION):

The Software Engineering Institute (SEI) has developed a process meta model based on a set of system and software engineering capabilities(that should be present as organizations reach different levels of process capability) and maturity. To achieve these capabilities, **the SEI develops a process model that conforms to the CMMI guidelines. The CMMI represents a process Meta model in 2 different ways. They are**

1) Continuous model
2) Staged model

1) Continuous model:

The continuous CMMI meta model describes a process in 2 dimensions(process area and capability level).Each process area (for ex: project planning or requirements management)is formally accessed against specific goals and practices and is rated according to the following capability levels:

Level 0: Incomplete. The process area (e.g., requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.

Level 1: Performed. All of the specific goals of the process area (as defined by the CMMI) have been satisfied. Work tasks required to produce defined work products are being conducted.

Level 2: Managed. All level 1criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy, stakeholders are actively involved in the process area as required; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description".

Level 3: Defined. All level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets" .

Level 4: Quantitatively managed. All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process" .

Level 5: Optimized. All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative (statistical) means to meet changing customer needs and to continually improve the efficacy of the process area under consideration" .

2) Staged Model: The staged CMMI model defines the same process areas, goals, and practices as the continuous model. The primary difference is that the staged model defines five maturity levels, rather than five capability levels. To achieve a maturity level, the specific goals and practices associated with a set of process areas must be achieved. The process areas required to achieve a maturity level is shown as:

Maturity Level	Focus	Process Areas
Optimizing	Continuous process improvement	Organizational Innovation and Deployment Causal Analysis and Resolution
Quantitatively managed	Quantitative management	Organizational process performance Quantitative Project Management
Defined	Process standardization	Requirements Development Technical Solution Verification Validation Project Management Integrated Supplier Management Risk Management Decision Analysis and Resolution
Managed	Basic *project* management	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance
Performed		

8. Process Patterns:

The software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and /or related behaviors required to develop computer software. That means the **process patterns can be used to define the characteristics of a process.**

A template is used to define a pattern. Typical examples: Customer communication (a process activity) Analysis (an action) Requirements gathering (a process task) Reviewing a work product (a process task) Design model (a work product) .

Types of process patterns: Ambler suggested 3 types of patterns for a process. They are

1. Task patterns
2. Stage patterns
3. Phase patterns

1.Task patterns: The task patterns define a software engineering action or work task that is part of the process and relevant to successful software engineering practice For example Requirements gathering is a task pattern.

2. **Stage patterns:** It defines a framework activity for the process. The framework activity consists of multiple work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage (framework activity).For example Communication is a stage pattern. This pattern would incorporate the task pattern requirements gathering and others.

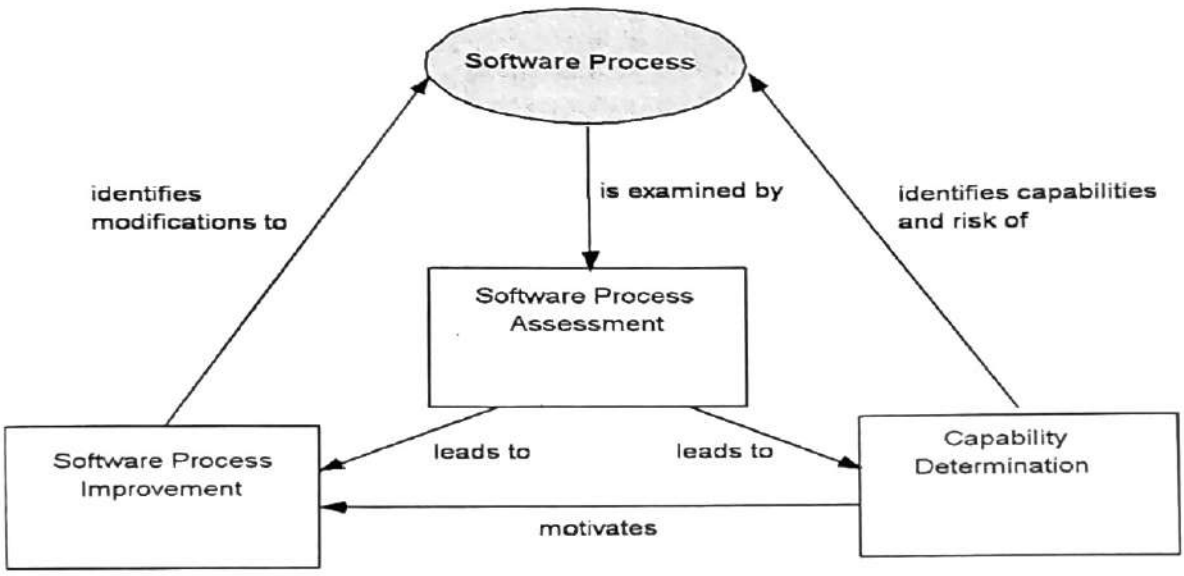
3. **Phase patterns:** The phase pattern defines the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature. For example Spiral Model or Prototyping Model is a phase pattern.

9. PROCESS ASSESSMENT AND IMPROVEMENT

The process itself should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering .A number of different approaches are proposed to the software process assessment. They are

- **SCAMPI(Standard CMMI Assessment Method for Process Improvement):**It provides Five-step process assessment model containing initiating, diagnosing, establishing, acting, learning.This method uses SEI CMMI as the basis for assessment.
- **CBA IPI(CMM Based Appraisal for Internal Process Improvement):**It Uses SEI CMM and provides a diagnostic technique for assessing the relative maturity of a software organization.
- **SPICE (ISO/IEC15504) Standard**: This standard defines a set of requirements for software process assessment. The intent of this standard is to assisst organizations in developing an objective evaluation of the efficacy of any defined S/W process.
- **ISO 9001:2000 for Software:** It is a generic standard that is applied to any organization that wants to improve the overall quality of the products, systems or services. Therefore ,this standard is directly applicable to S/W organizations and companies.

The relationship between the software process and the methods applied for assessment and improvement is shown as :



PROCESS MODELS

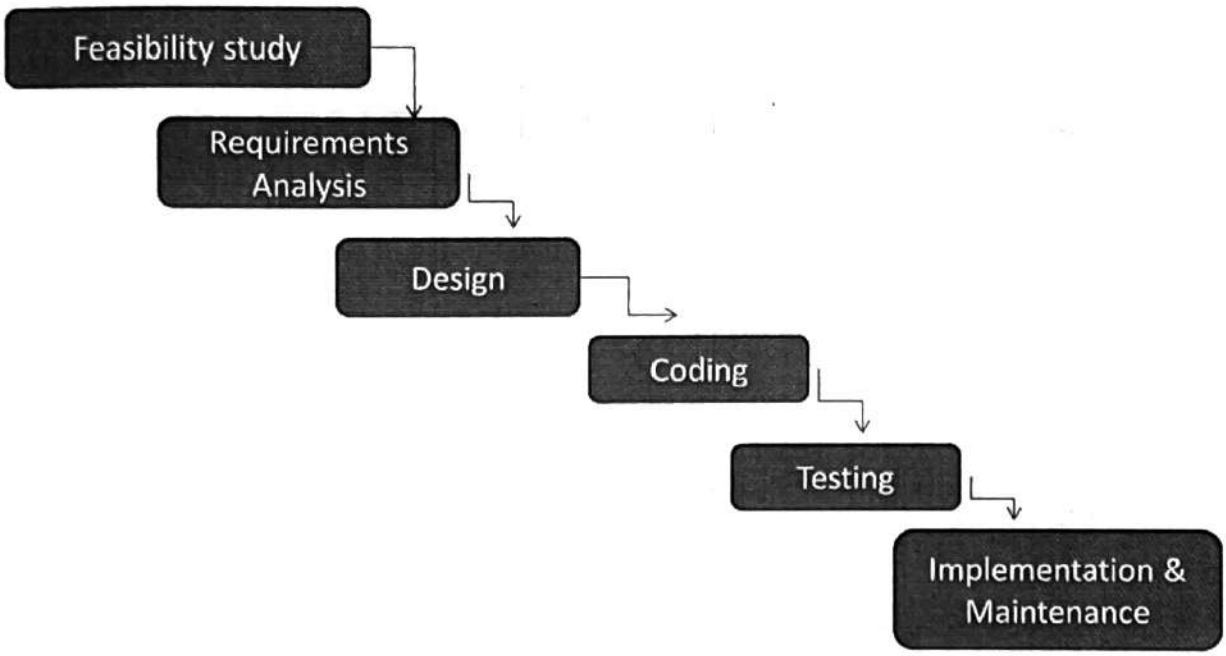
10. THE WATERFALL MODEL:

Process model is an activity in the SDLC. The process model represents the descriptive and diagrammatic model of software product with a series of different phases. A number of process models are available for Software Engineering. These process models are not perfect, but they provide an useful road map of software engineering work.

❖ **Prescriptive process model:**

This model defines a set of framework activities that establishes the work flow for the SDLC. It is also known as LINEAR SEQUENTIAL MODEL or PRESCRIPTIVE MODEL or WATERFALL MODEL or CLASSICAL LIFE CYCLE MODEL. It is the oldest paradigm for developing the software. The framework activities are included in waterfall model are:

- Feasibility study
- Requirements analysis
- Design
- Coding
- Testing
- Implementation and maintenance



❖ **Advantages of waterfall model:**

- This model is easy to understand and use
- It is well suited for small projects

- In this model work flows in linear fashion.
- In this model phases do not overlap.

❖ **Disadvantages of waterfall model:**

- Customer interaction in the first phase only.
- This model is not suitable to make the changes.
- It is not suitable for big and object oriented projects.

❖ **When to use waterfall model:**

- There are no ambiguous requirements.
- Technology is understood
- The project is too short.

11. INCREMENTAL PROCESS MODEL:

Process model is an activity in the SDLC. The process model represents the descriptive and diagrammatic model of software product with a series of different phases. A number of process models are available for Software Engineering. These process models are not perfect, but they provide an useful road map of software engineering work.

❖ **Incremental Model:**

The incremental process model delivers the software in small usable pieces, those are called as INCREMENTS. This model provides more functionality for the customer of each increment is delivered. The incremental model applies the linear sequences in incremental steps. The framework activities are: communication, planning, modelling, construction, and Deployment.

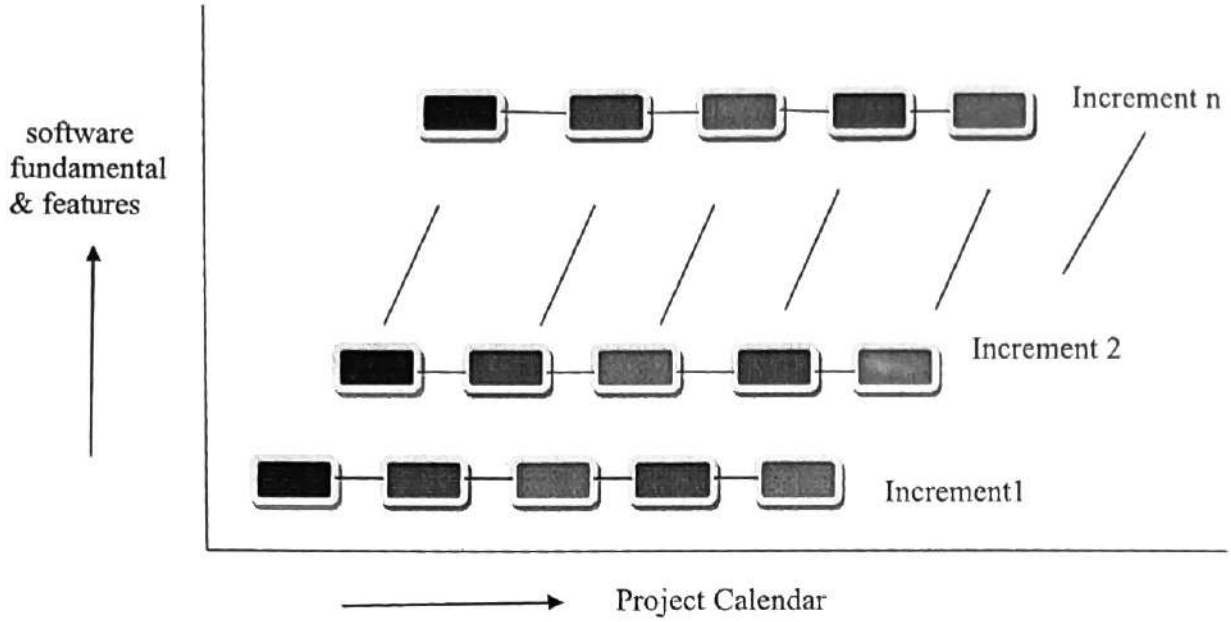
There are two types of incremental models. They are:

- Incremental model
- RAD (Rapid Application Development) model.

1. Incremental Model:

In Incremental model each linear sequence produces deliverable increment software. The first incremental product is called "core product". The core product is used by the customer , as a result of use, a plan is developed for the next increment. This plan specifies modification of a core product to be better in order to meet the user requirements and the delivery of additional features and functionalities. This process is repeated until the complete product is delivered.

The graphical representation of incremental model is as follows.



❖ ***Advantages of incremental model:***

- This model is more flexible.
- This model is useful, when staffing is too short, for a full scale development.
- Generates working software quickly.

❖ ***Disadvantages of incremental model:***

- The cost is higher than waterfall model.
- Needs good planning and designing.
- There are some high risk features and goals.

❖ ***When to use this model:***

- A new technology is being used.
- There is need to get a product in the market early.

2. RAPID APPLICATION DEVELOPMENT MODEL (RAD) :

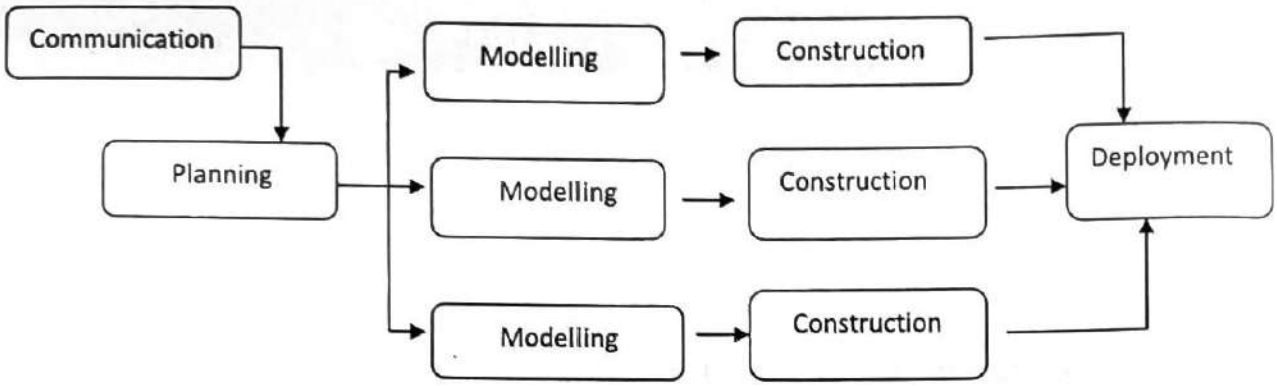
Process model is an activity in the SDLC. The process model represents the descriptive and diagrammatic model of software product with a series of different phases. A number of process models are available for Software Engineering. These process models are not perfect, but they provide an useful road map of software engineering work.

❖ **RAD model:**

RAD is an incremental software process model. It is also known as SHORTEST DEVELOPMENT LIFE CYCLE MODEL. (SDLCM).

This model is an high speed adaptation of the linear process model, in which a rapid development is achieved by using component based construction. The RAD contains five generic framework activities, those are, communication, planning, modelling, construction and Deployment

The diagrammatic representation is as follows:



- **Communication**: In this phase to understand the problem and customer characteristic that the software must accommodate.
- **Planning**: In this phase the system analysis can be done by multiple teams work in parallel on different system functions.
- **Modelling**: This phase concentrates on designing .It contains three activities.

  They are:
  - **Business Modelling:** Here, the information is modelled on away that is answered what information is generated, who generates the information.
  - **Data Modelling:** Here the information flow is refined in to set of objects.
  - **Process Modelling:** Here, the processing descriptions are created for adding, deleting, modifying and updating the data object.
- **Construction:** The RAD construction emphasis the re-use of the existing program components i.e., coding and testing.
- **Deployment:** This phase includes the delivery, support and feedback.

Advantages of RAD model:

- This model reduces development time.
- When the customer needs the product very urgent.
- More staff the work can be distributed.
- This model increases the quality.

Disadvantages of RAD model:

- Staff should be experts.
- RAD model is not useful for smaller projects.
- This model requires more money and resources to implement the product. Because it requires highly skilled designers.

When to use RAD model:

- RAD should be used when there is an need to create a system, that can be modularized in 2-3 months of a time.
- The project which needs automated modelling and code generating tools.

12. Evolutionary process models:

Process model is an activity in the SDLC. The process model represents the descriptive and diagrammatic model of software product with a series of different phases. A number of process models are available for Software Engineering. These process models are not perfect, but they provide a useful road map of software engineering work.

The evolutionary model accommodates the changes in the designing of the software and also provides a scope for the enhancement of the software.

There are two types of evolutionary process models. They are:

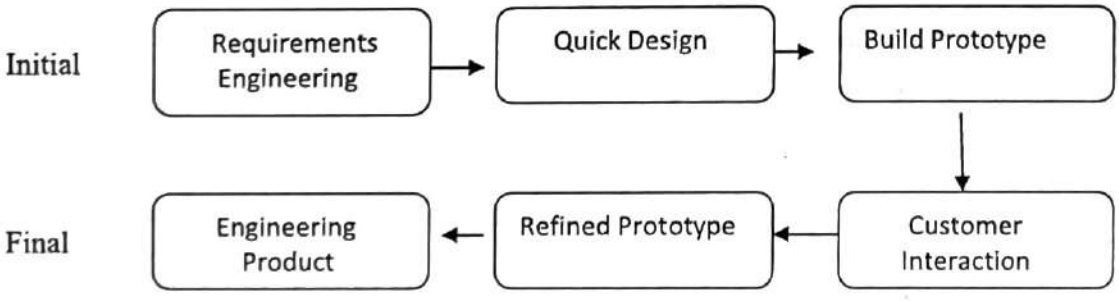
- Prototype process model.
- Spiral process model.

A. PROTOTYPE PROCESS MODEL:

Prototyping model is a working model of software with limited functions. This model is an attractive approach for complex and large systems development.

This model consists of various framework activities. Those are:

- Requirements gathering.
- Quick design
- Build prototype.
- Customer evolution.
- Refined prototype.
- Engineering product.



- **Requirements Engineering**: The model begins with requirements gathering, where the requirements of the system are defined in detailed.

- **Quick design**: When the requirements are known, quick design for the system are created.
- **Build Prototype:** The first prototype of the required system is created from quick design. Here a prototype means a sample product of the whole product.
- **Customer Interaction:** Here the user thoroughly evaluate the prototype and recognise the strength and weakness of the product that means, what is added are, what is to be removed, comments and suggestions are collected from the user and provided to the developer.
- **Refined prototype:** Once the users evaluate the prototype, it is refined (iterative) according to the user requirements.
- **Engineering product:** After evaluate the final prototype, start the development of the product, which can be delivered to the user as a final product.

❖ ***Advantages of prototype model:***

1. This model is excellent for designing the good human computer interfaces.
2. Users are actively participated in the development
3. This model reduces the risks.
4. Feedback is used to refine the prototype.

❖ ***Disadvantages of prototype model:***

1. The effort invested in prototyping model may be too much, if it is not monitored properly.
2. It gives less importance to the software quality and maintainability, because it gives importance to the workable prototype of the system.

❖ ***When to use prototype model:***

- When requirements are not well understood and difficult to determine.
- Typical online systems , web interfaces have a very high amount of interaction with end users are best suited for prototyping model.

B. SPIRAL PROCESS MODEL:

Process model is an activity in the SDLC. The process model represents the descriptive and diagrammatic model of software product with a series of different phases. A number of process models are available for Software Engineering. These process models are not perfect, but they provide a useful road map of software engineering work.

The evolutionary model accommodates the changes in the designing of the software and also provides a scope for the enhancement of the software.

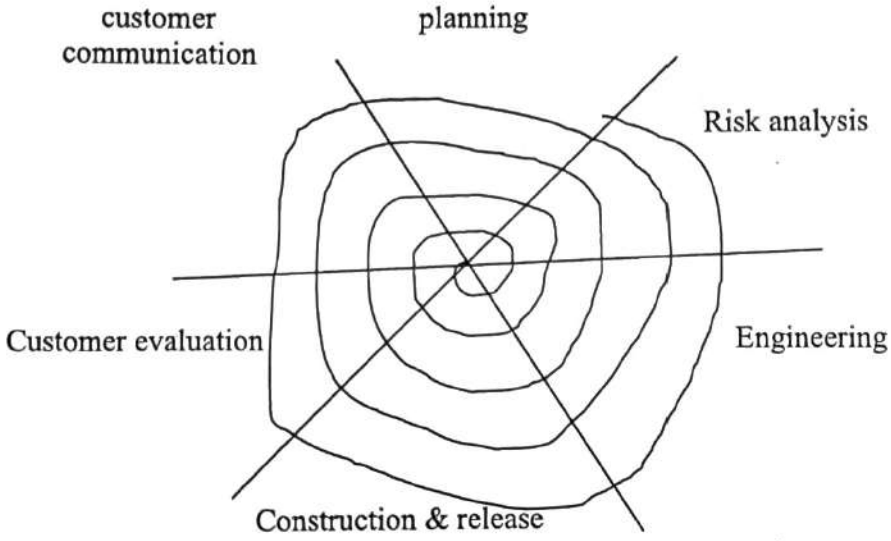
❖ **Spiral process Model:**

This model was invented by "Dr. Barry Boehm", in 1980, which follows an evolutionary approach. This model combines the systematic approach of waterfall model and iterative nature of prototyping model.

This model consists of six framework activities .They are:

- ❖ Customer communication.
- ❖ Planning.
- ❖ Risk analysis.
- ❖ Engineering.

- Construction and release.
- Customer evolution.



1. **Customer communication:** This phase includes the effective communication between the client and the software developers, to understand the user requirements.
2. **Planning:** This phase includes the required resources, deadlines, project schedule, and other project related information can be defined.
3. **Risk analysis:** This phase identification and rectification of technical and management risk are to be done.
4. **Engineering:** This phase, the tasks which are required to design one or more representations of software can be done.
5. **Construction and release:** This phase, the tasks which are required to construct the coding and testing, and also provide user support can be done.
6. **Customer evaluation:** This phase, to evaluate the software to the client and take feedback, regarding the quality and functionalities of the software.

Advantages of spiral model:

- This model is suited for Object oriented projects.
- This model reduces the risk management.
- This model is good for large and machine critical projects.

Disadvantages of spiral model:

- This model is costly model.
- It is not suitable for low risk projects.

When to use spiral model:

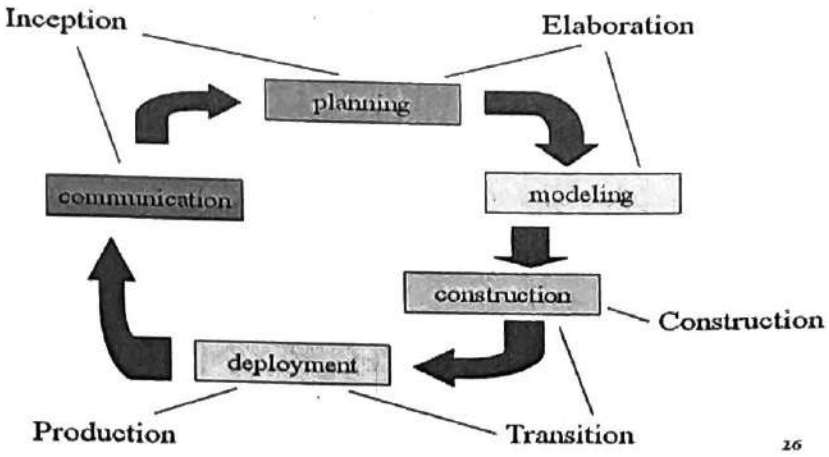
- Requirements are complex.
- Significant changes are expected.
- When cost and risk evolution is important.

13. Unified Process:

In 1997, Ivar Jacobson, James Rumbaugh and Grady Booch developed the **Unified Process Model**. It provides a framework for object-oriented software engineering using UML(Unified Modelling Language).

The Unified process is a "Use-case driven, architecture process, iterative and incremental" software process designed as a framework for UML methods and tools. It is an incremental model, in which 5 phases are defined. They are

1. Inception phase
2. Elaboration phase
3. Construction phase
4. Transition phase
5. Production phase



1.Inception Phase: It combines both customer communication and planning activities of the generic process. In this phase business requirements for the software are identified. A rough architecture for the system is proposed.

A plan is created for an incremental, iterative development. Fundamental business requirements are described through preliminary use cases. Here a use case describes a sequence of actions that are performed by a user.

2. Elaboration Phase: It combines both the planning and modeling activities of the generic process. It refines and expands the preliminary use cases. It expands the architectural representation to include five views

- Use-case model
- Analysis model
- Design model
- Implementation model
- Deployment model

It often results an executable **architectural baseline** .

3. Construction phase:- It combines the construction activity of the generic process.This phase uses the architectural model from the elaboration phase as input. It develops or acquires the software components that make each use-case operational for end-users. Analysis and design models from the previous phase are completed to reflect the final version of the software increment. Use cases are used to derive a set of acceptance tests that are executed prior to the next phase.

4. Transition phase:- It combines (or) encompasses the last part of the construction activity and the first part of the deployment activity of the generic process. Software is given to end users for beta testing and user feedback reports on defects and necessary changes. The software team create necessary support documentation or information (ex: user manuals, trouble-shooting guides, installation procedures).At the conclusion of this phase, the software increment becomes a usable software release.

5. Production phase:- It encompasses the last part of the deployment activity of the generic process. During this phase, On-going use of the software is monitored, support for the operating environment (infrastructure) is provided and defect reports and requests for changes are submitted and evaluated .

Unified Process Work Products:-

- Work products are produced in each of the first four phases of the unified process.
- In this process, we will concentrate on the analysis model and the design model work products.
- Analysis model includes
  - –Scenario-based model, class-based model, and behavioural model
- Design model includes
  - –Component-level design, interface design, architectural design, and data/class design

UNIT-11

MCA20205-------SYLLABUS

SOFTWARE REQUIREMENTS: Functional and non-functional requirements, user requirements, system requirements, interface specification, the software requirements document.
REQUIREMENTS ENGINEERING PROCESS: Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management.
SYSTEM MODELS: Context models, behavioral models, data models, object models, structured methods.

SOFTWARE REQUIREMENTS

1. Functional and non-functional requirements:

A requirement specifies a function that a system or component must be able to perform.

❖ **Functional Requirements:**

The official definition for a functional requirement specifies what the system should do. Functional requirements specify specific behavior or functions.
For example: "Display the heart rate, blood pressure and temperature of a patient connected to the patient monitor.

"The functional requirements depend on the type of s/w being developed, the expected users of the s/w & the general approach taken by the organization when writing requirements. Functional requirements for a s/w system may be expressed in different ways. For example functional requirements for a university library system used by students and faculty to order books and documents from other libraries.

1.The user shall be able to search either all of the initial set of databases or select a subset from it.
2.The system shall provide appropriate viewers for the user to read documents in the document store.
3.Every order shall be allocated a unique identifier (order_id) which the user shall be able to copy to the accounts permanent storage area.

❖ **Non-Functional requirements:**

The official definition for a non-functional requirement specifies how the system should behave. Non-functional requirements specify all the remaining requirements not covered by the functional requirements. They specify criteria that judge (or) decide the operation of a system, rather than specific behaviors. for example: "Display of the patient's vital signs must respond to a change in the patient's status within 2 seconds."

There are three different types of non-functional requirements: They are: Product, Organizational and External requirements.

- **Product requirements:** The product requirements specify product behavioour. Examples include **performance requirements** defines how fast the system must execute and how much memory it requires. **Reliability requirements** that set out the acceptable failure rate, portability and usability requirements.
- **Organizational requirements:** These requirements are derived from policies and procedures in the customer's and developer's organization. Examples include process standards that must be used, **implementation requirements** such as the programming language or design method used, **and delivery requirements** that specify when the product and its documentation are to be delivered.
- **External requirements:** These requirements are derived from factors external to the system and its development process. These requirements may include **interoperability requirements** that define how the system interacts with systems in other organizations, **ethical requirements** placed on a system to ensure that it will be acceptable to its users and the general public & **legislative requirements** that must be followed to ensure that the system operates within the law.

2. User requirements:

Requirements are descriptions of the services that a software system must provide and the constraints under which it must operate .Requirements can range from high-level abstract statements of services or system constraints to detailed mathematical functional specifications.

❖ **User requirements define:**

- Statements in natural language plus diagrams of the services that the systems provide and its operational constraints.
- Written for customers.

The software must provide a means of representing and accessing external files created by other tools.

1. The user should be provided with facilities to define the type of external files.
2. Each external file type may have an associated tool which may be applied to the file.
3. Each external file type may be represented as a specific icon on the user's display.
4. Facilities should be provided for the icon representing an external file to be defined by the user.
5. When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of external file to the file represented by the selected icon.

❖ **User requirement**

- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge
- User requirements are defined using natural language, tables and diagrams.

Problems with natural language

➢ **Ambiguity** – Readers and writers may not interpret words in the same way

- **Over-flexibility** – The same thing may be expressed in a number of different ways
- **Requirements amalgamation & confusion** – Several different requirements may be expressed together; functional and non-functional requirements may be mixed together .
- **Lack of modularization** – NL structures are inadequate to structure system requirements

Example

1. The requirements for a CASE tool for editing software design models include the requirement for a grid to be displayed in the design window .
2. "To assist in the positioning of entities on a diagram, the user may turn on a grid in either centimeters or inches, via an option on the control panel"
3. This statement mixes up three different kinds of requirement

- A conceptual functional requirement stating that the editing system should provide a grid; it presents a rationale for this
- A non-functional requirement giving information about the grid units
- A non-functional user interface requirement defining how the grid is switched on and off by the user.

3. System requirements

System requirements are all of the **requirements** at the *system level* that describe the functions which the system as a whole should fulfill to satisfy the **stakeholder needs and requirements**, and are expressed in an appropriate combination of textual statements, views, and non-functional requirements; the latter expressing the levels of safety, security, reliability, etc., that will be necessary.

A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor .Written for developers

System requirements play major roles in systems engineering, as they:

- Form the basis of system **architecture** and **design** activities.
- Form the basis of system **integration** and **verification** activities.
- Act as reference for **validation** and stakeholder acceptance.
- Provide a means of communication between the various technical staff that interact throughout the project.

❖ Classification of System Requirements

Several classifications of system requirements are possible, depending on the requirements definition methods and/or the architecture and design methods being applied.

Types of System Requirement	Description
Functional Requirements	Describe qualitatively the system functions or tasks to be performed in operation.
Performance Requirements	Define quantitatively the extent, or how well and under what conditions a function or task is to be performed (e.g. rates, velocities). These are quantitative requirements of system performance and are verifiable individually. Note that there may be more than one performance requirement associated with a single function, functional requirement, or task.
Usability Requirements	Define the quality of system use (e.g. measurable effectiveness, efficiency, and satisfaction criteria).
Interface Requirements	Define how the system is required to interact or to exchange material, energy, or information with external systems (external interface), or how system elements within the system, including human elements, interact with each other (internal interface). Interface requirements include physical connections (physical interfaces) with external systems or internal system elements supporting interactions or exchanges.
Operational Requirements	Define the operational conditions or properties that are required for the system to operate or exist. This type of requirement includes: human factors, ergonomics, availability, maintainability, reliability, and security.
Environmental Conditions	Define the environmental conditions to be encountered by the system in its different operational modes. This should address the natural environment (e.g. wind, rain, temperature, fauna, salt, dust, radiation, etc.), induced and/or self-induced environmental effects (e.g. motion, shock, noise, electromagnetism, thermal, etc.), and threats to societal environment (e.g. legal, political, economic, social, business, etc.).
Logistical Requirements	Define the logistical conditions needed by the continuous utilization of the system. These requirements include sustainment (provision of facilities, level support, support personnel, spare parts, training, technical documentation, etc.), packaging, handling, shipping, transportation.

4. Interface specification:

A **user interface specification (UI specification)** is a document that captures the details of the software user interface into a written document. The specification covers all possible actions that an end user may perform and all visual, auditory and other interaction elements.

❖ **Purpose:**

The UI specification is the main source of implementation information for how the software should work Beyond implementation, a UI specification should consider usability, localization, and demo limits.

- A UI spec may also be incorporated by those within the organization responsible for marketing, graphic design, and software testing. As future designers might continue or build on top of existing work.
- A UI specification should consider forward compatibility constraints in order to assist the implementation team.
- The UI specification can be regarded as the document that bridges the gap between the product management functions and implementation. One of the main purposes of a UI specification is to process the product requirements into a more detailed format. The level of detail and document type varies depending the needs and design practices of the organizations. The small scale prototypes might require only modest documentation with high-level details.

In general, the goal of requirement specifications are to describe what a product is capable of, whereas the UI specification details how these requirements are implemented in practice.

❖ **Process:**

Before UI specification is created, a lot of work is done already for defining the application and desired functionality. Usually there are requirements for the software which are basis for the use case creation and use case prioritizing. UI specification is only as good as the process by which it has been created, so lets consider the steps in the process:

➢ **Use case definition:**

Use cases are then used as basis for drafting the UI concept (which can contain for example main views of the software, some textual explanations about the views and logical flows), these are short stories that explain how the end user starts and completes a specific task, but not about how to implement it.

The purpose of writing use cases is to enhance the UI designer understanding of the features that the product must have and of the actions that take place when the user interacts with the product.

➢ **Design draft creation:**

The UI design draft is done on the basis of the use case analysis. The purpose of the UI design draft is to show the design proposed, and to explain how the user interface enables the user to complete the main use cases, without going into details.

It should be as visual as possible and all the material created must be in such a format that it can be used in the final UI specification. (This is good time to conduct usability testing or expert evaluations and make changes.)

➢ **Writing the user interface specification**:

The UI specification is then written to describe the UI concept. The UI specification can be seen as an extension of the design draft that provides a complete description that contains all details, exceptions, error cases, notifications, and so forth. The amount of detail provided depends on the needs and characteristics of the development organization (scope of the product, culture of the organization, and development methodology used, among others). Usually, the UI concept and specifications are reviewed by the stakeholders to ensure that all necessary details are in place.

5. Software requirements document:

Software requirement specification (also called as a software requirement specification)is a kind of document which is created by a software analyst after the requirements collected from the various sources - the requirement received by the customer written in ordinary language. It is the job of the analyst to write the requirement in technical language so that they can be understood and beneficial by the development team.

The models used at this stage include ER diagrams, data flow diagrams (DFDs), function decomposition diagrams (FDDs), data dictionaries, etc.

- **Data Flow Diagrams:** Data Flow Diagrams (DFDs) are used widely for modeling the requirements. DFD shows the flow of data through a system. The system may be a company, an organization, a set of procedures, a computer hardware system, a software system, or any combination of the preceding. The DFD is also known as a data flow graph or bubble chart.
- **Data Dictionaries:** Data Dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirements stage, the data dictionary should at least define customer data items, to ensure that the customer and developers use the same definition and terminologies.
- **Entity-Relationship Diagrams:** Another tool for requirement specification is the entity-relationship diagram, often called an "***E-R diagram***." It is a detailed logical representation of the data for the organization and uses three main constructs i.e. data entities, relationships, and their associated attributes.

A software requirements specification describes the essential behaviour of a software product from a user's point of view. The purpose of the SRS is to:

- **Establish the basis for agreement between the customers and the suppliers on what the software product is to do.**
- **Provide a basis for developing the software design**
- **Reduce the development effort.**
- **Provide a basis for estimating costs and schedules**

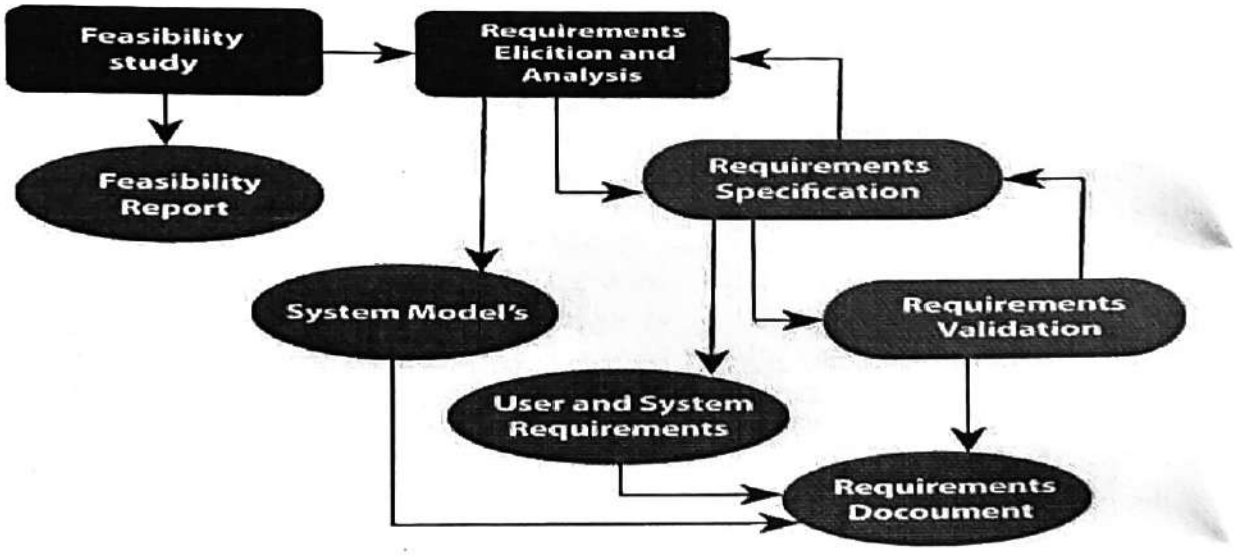
- **Provide a baseline for validation and verification.**
- **Facilitate transfer.** The SRS makes it easier to transfer the software product to new users or new machines. Customers thus find it easier to transfer the software to other parts of their organisation and suppliers find it easier to transfer it to new customers
- **Serve as a basis for enhancement.** Because the SRS discusses the product but not the project that developed it, the SRS serves as a basis for later enhancement of the finished product. The SRS may need to be altered, but it does provide a foundation for continued product evaluation.

6 .REQUIREMENTS ENGINEERING PROCESS :

Requirements engineering (RE) refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system.

Requirement Engineering Process. **It is a four -step process, which includes -**

1. Feasibility Study
2. Requirement Elicitation and Analysis
3. Software Requirement Validation
4. Software Requirement Management



Requirement Engineering Process

1. Feasibility Study:

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

❖ **Types of Feasibility:**

1. **Technical Feasibility** - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.
2. **Operational Feasibility** - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.
3. **Economic Feasibility** - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

2. Requirement Elicitation and Analysis:

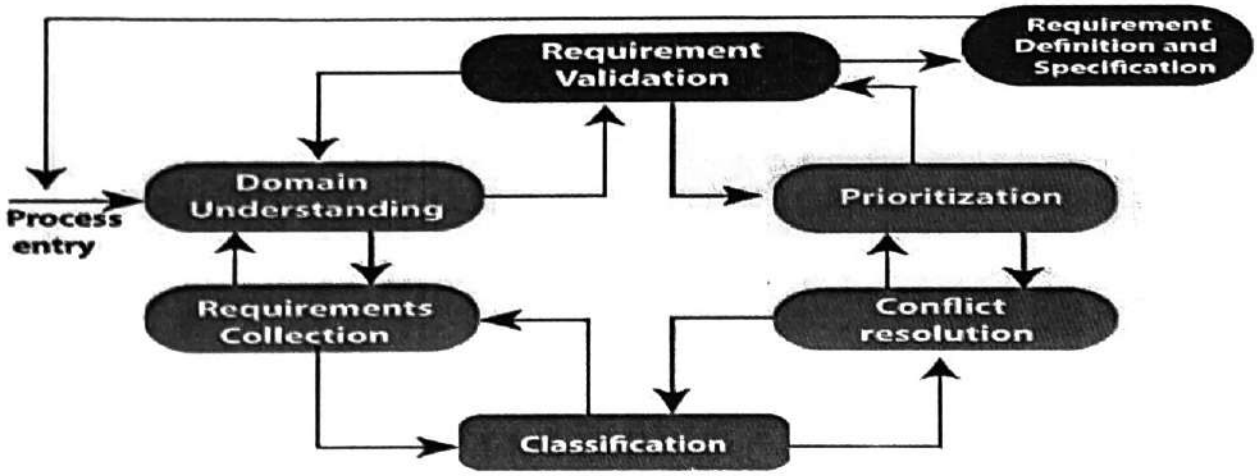
This is also known as the **gathering of requirements**. Here, requirements are identified with the help of customers and existing systems processes, if available.

Analysis of requirements starts with requirement elicitation. The requirements are analyzed to identify inconsistencies, defects, omission, etc. We describe requirements in terms of relationships and also resolve conflicts if any.

❖ **Problems of Elicitation and Analysis**

- Getting all, and only, the right people involved.
- Stakeholders often don't know what they want
- Stakeholders express requirements in their terms.
- Stakeholders may have conflicting requirements.
- Requirement change during the analysis process.
- Organizational and political factors may influence system requirements.

Elicitation and Analysis Process



3. Software Requirement Validation:

After requirement specifications developed, the requirements discussed in this document are validated. The user might demand illegal, impossible solution or experts may misinterpret the needs. Requirements can be the check against the following conditions -

- If they can practically implement
- If they are correct and as per the functionality and specially of software
- If there are any ambiguities
- If they are full
- If they can describe

❖ **Requirements Validation Techniques**

- **Requirements reviews/inspections:** systematic manual analysis of the requirements.
- **Prototyping:** Using an executable model of the system to check requirements.
- **Test-case generation:** Developing tests for requirements to check testability.
- **Automated consistency analysis:** checking for the consistency of structured requirements descriptions.

4. Software Requirement Management:

Requirement management is the process of managing changing requirements during the requirements engineering process and system development.

New requirements emerge during the process as business needs a change, and a better understanding of the system is developed.

- ❖ The priority of requirements from different viewpoints changes during development process.
- ❖ The business and technical environment of the system changes during the development.

7. SYSTEM MODELS

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. It is about representing a system using some kind of graphical notation, which is now almost always based on notations in the **Unified Modeling Language (UML)**. Models help the analyst to understand the functionality of the system; they are used to communicate with customers.

System Models can explain the system from **different perspectives**:

- An **external** perspective, where you model the context or environment of the system.
- An **interaction** perspective, where you model the interactions between a system and its environment, or between the components of a system.

- A **structural** perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A **behavioral** perspective, where you model the dynamic behavior of the system and how it responds to events.

Five types of UML diagrams that are the most useful for system modeling:

- **Activity** diagrams, which show the activities involved in a process or in data processing.
- **Use case** diagrams, which show the interactions between a system and its environment.
- **Sequence** diagrams, which show interactions between actors and the system and between system components.
- **Class** diagrams, which show the object classes in the system and the associations between these classes.
- **State** diagrams, which show how the system reacts to internal and external events.

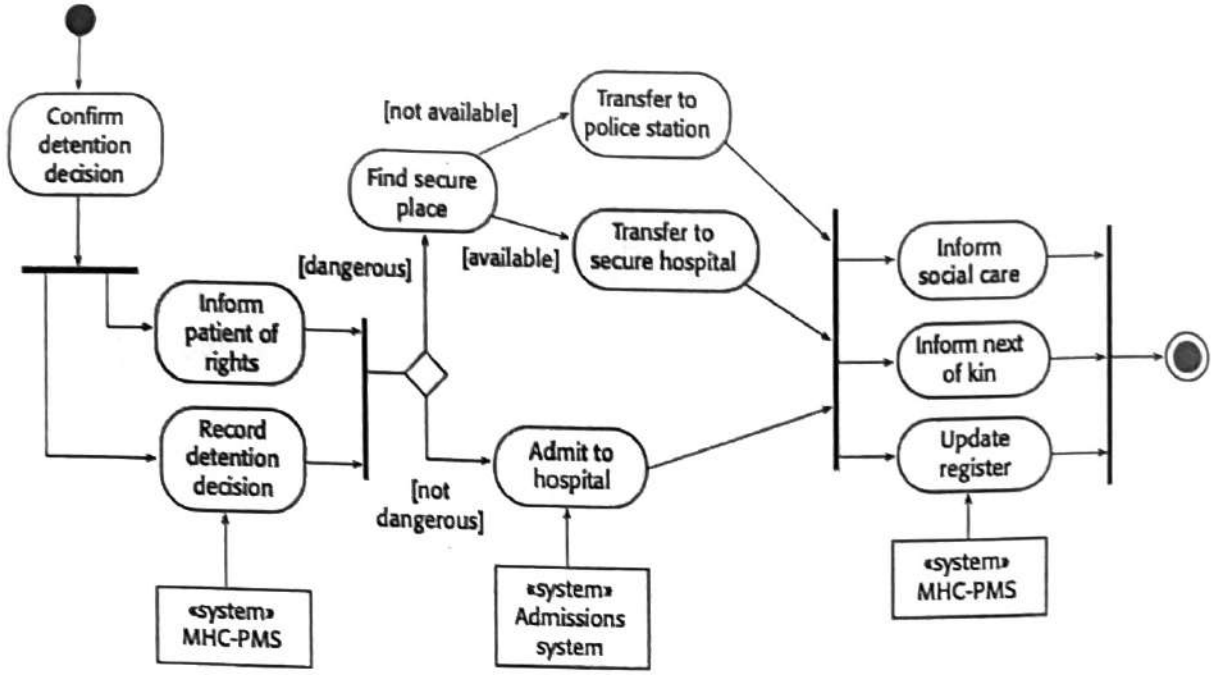
Models of both new and existing system are used during **requirements engineering**. Models of the **existing systems** help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system. Models of the **new system** are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.

❖ Context models

Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries. Social and organizational concerns may affect the decision on where to position system boundaries. Architectural models show the system and its relationship with other systems.

System boundaries are established to define what is inside and what is outside the system. They show other systems that are used or depend on the system being developed. The position of the system boundary has a profound effect on the system requirements. Defining a system boundary is a political judgment since there may be pressures to develop system boundaries that increase/decrease the influence or workload of different parts of an organization.

Context models simply show the other systems in the environment, not how the system being developed is used in that environment.The example below shows a UML **activity diagram** describing the process of involuntary detention and the role of MHC-PMS (mental healthcare patient management system) in it.

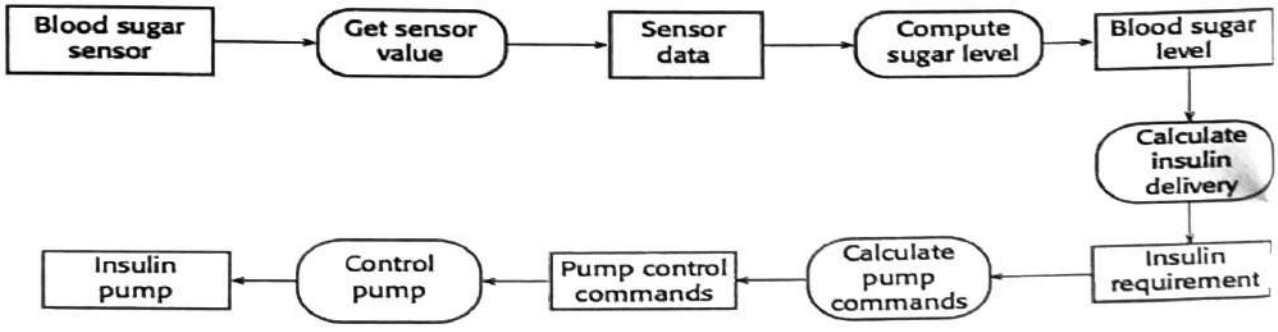


❖ Behavioral models

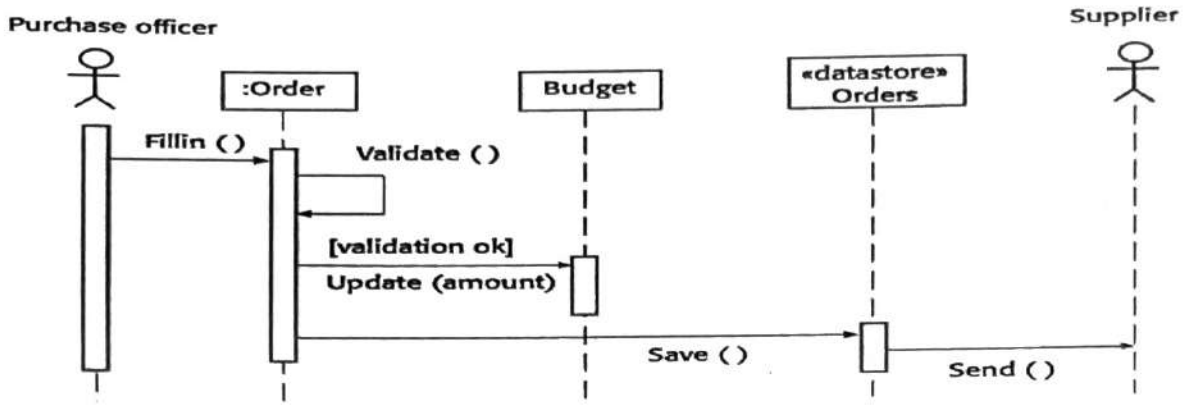
Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. Two types of stimuli:

- Some **data** arrives that has to be processed by the system.
- Some **event** happens that triggers system processing. Events may have associated data, although this is not always the case.

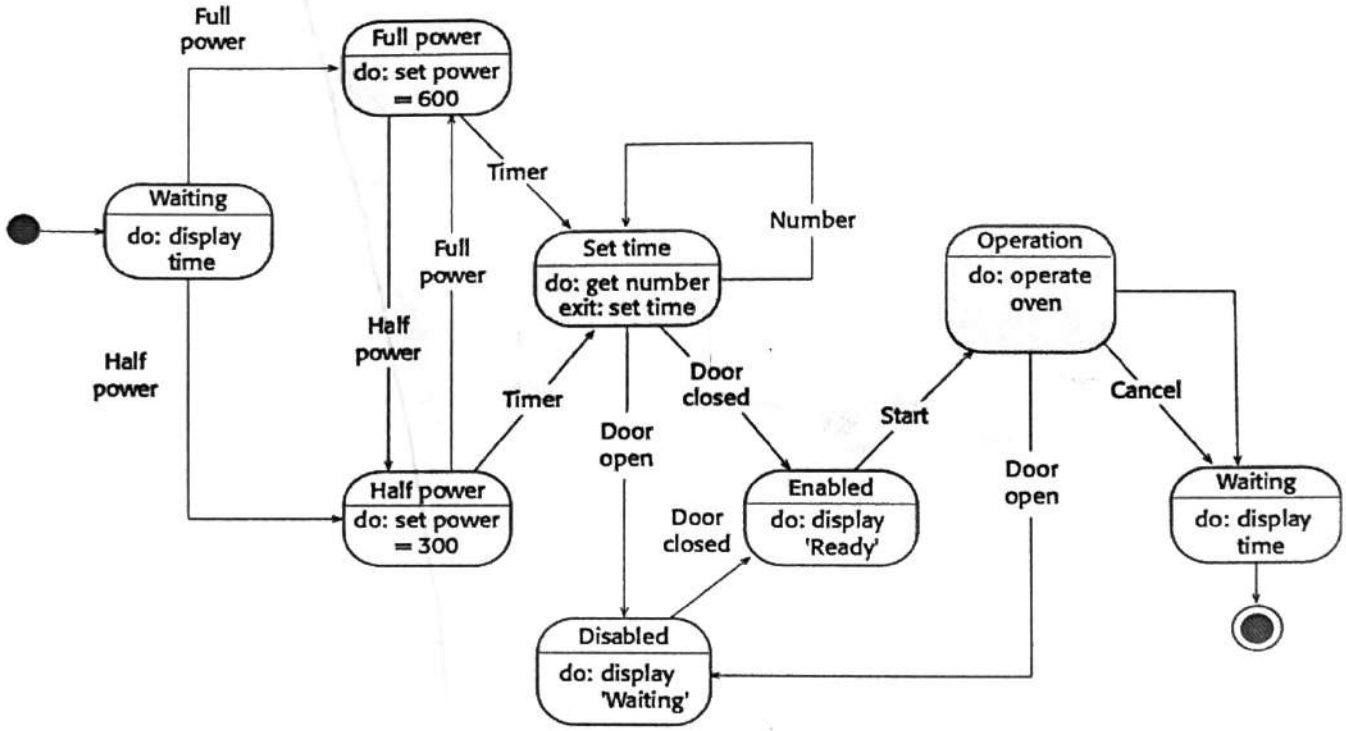
Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing. **Data-driven models** show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. Data-driven models can be created using UML **activity diagrams**:



Data-driven models can also be created using UML **sequence diagrams**:



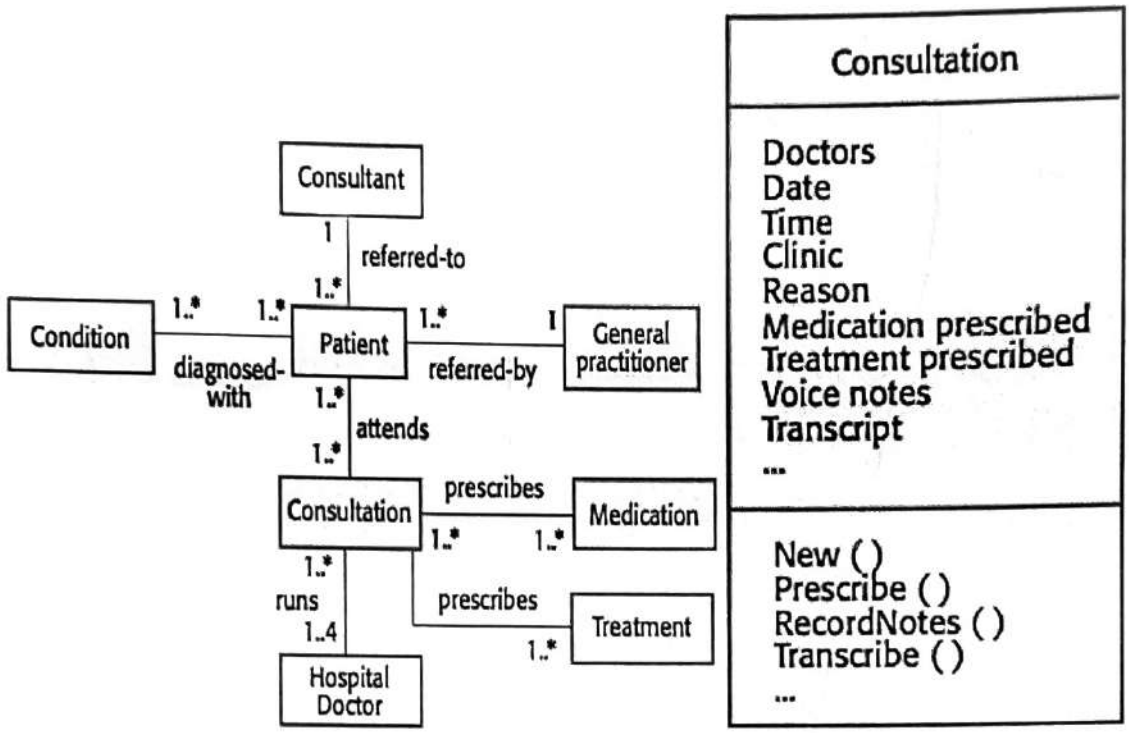
Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone. **Event-driven models** shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. Event-driven models can be created using UML **state diagrams**:



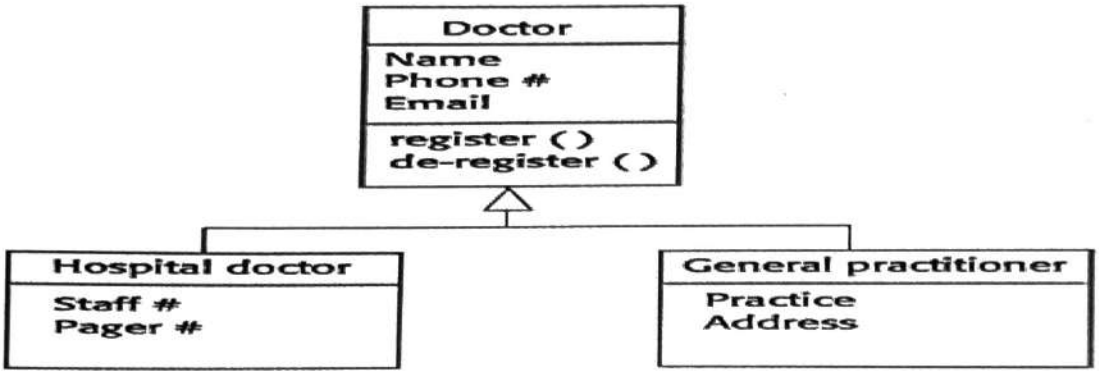
❖ Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be **static** models, which show the structure of the system design, or **dynamic** models, which show the organization of the system when it is executing. You create structural models of a system when you are discussing and designing the system architecture.

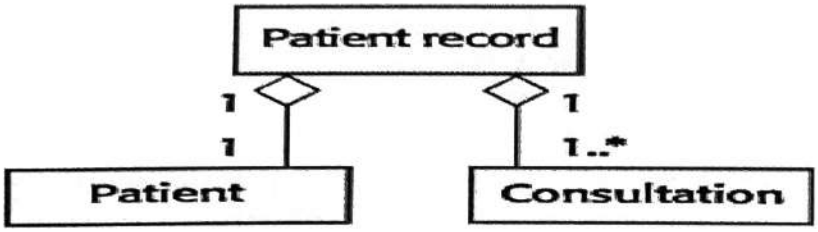
UML **class diagrams** are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. An object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is some relationship between these classes. When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.



Generalization is an everyday technique that we use to manage complexity. In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. In object-oriented languages, such as Java, generalization is implemented using the class **inheritance** mechanisms built into the language. In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.



An **aggregation** model shows how classes that are collections are composed of other classes. Aggregation models are similar to the part-of relationship in semantic data models.



❖ Data models:

A **data model** is an abstract model that organizes elements of data and standardizes how they relate to one another and to the properties of real-world entities. For instance, a data model may specify that the data element representing a car be composed of a number of other elements which, in turn, represent the color and size of the car and define its owner.

The term **data model** can refer to two distinct but closely related concepts. Sometimes it refers to an abstract formalization of the objects and relationships found in a particular application domain: for example the customers, products, and orders found in a manufacturing organization. At other times it refers to the set of concepts used in defining such formalizations: for example concepts such as entities, attributes, relations, or tables. So the "data model" of a banking application may be defined using the entity-relationship "data model". This article uses the term in both senses.

A data model explicitly determines the structure of data. Data models are typically specified by a data specialist, data librarian, or a digital humanities scholar in a data modeling notation. These notations are often represented in graphical form.[7]

A data model can sometimes be referred to as a data structure, especially in the context of programming languages. Data models are often complemented by function models, especially in the context of enterprise models.

➢ The role of data models:

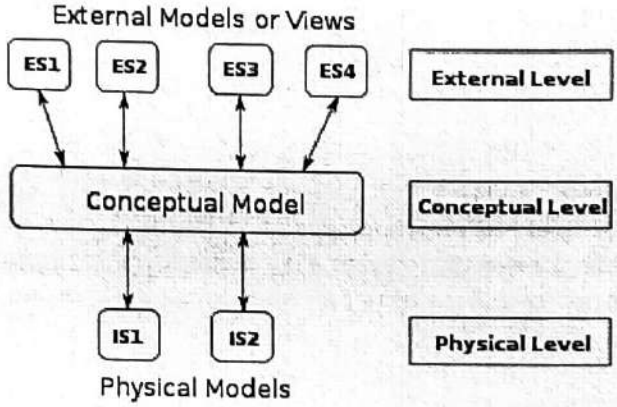
The main aim of data models is to support the development of information systems by providing the definition and format of data

"Business rules, specific to how things are done in a particular place, are often fixed in the structure of a data model. This means that small changes in the way business is conducted lead to large changes in computer systems and interfaces".

- "Entity types are often not identified, or incorrectly identified. This can lead to replication of data, data structure, and functionality, together with the attendant costs of that duplication in development and maintenance".
- "Data models for different systems are arbitrarily different. The result of this is that complex interfaces are required between systems that share data. These interfaces can account for between 25-70% of the cost of current systems".
- "Data cannot be shared electronically with customers and suppliers, because the structure and meaning of data has not been standardized. For example, engineering design data and drawings for process plant are still sometimes exchanged on paper".

➢ Three perspectives:

A data model *instance* may be one of three kinds :



1. **Conceptual data model:** describes the semantics of a domain, being the scope of the model. For example, it may be a model of the interest area of an organization or industry. This consists of entity classes, representing kinds of things of significance in the domain, and relationship assertions about associations between pairs of entity classes
2. **Logical data model:** describes the semantics, as represented by a particular data manipulation technology. This consists of descriptions of tables and columns, object oriented classes, and XML tags, among other things.
3. **Physical data model**: describes the physical means by which data are stored. This is concerned with partitions, CPUs, tablespaces, and the like.

❖ Object Models:

An object model is a logical interface, software or system that is modeled through the use of object-oriented techniques. It enables the creation of an architectural software or system model prior to development or programming.

An object model is part of the object-oriented programming (OOP) lifecycle. An object model helps describe or define a software/system in terms of objects and classes. It defines the interfaces or interactions between different models, inheritance, encapsulation and other object-oriented interfaces and features.

Object models types:

- **Document Object Model (DOM)**: A set of objects that provides a modeled representation of dynamic HTML and XHTML-based Web pages
- **Component Object Model (COM)**: A proprietary Microsoft software architecture used to create software components

❖ Basics of the object Model:

The object Model sees an information system as a set of objects and classes. The reader will get an in-depth understanding of the object model and its elements. We will also look into object-oriented programming (OOP), object-oriented design (OOD), and object-oriented analysis (OOA).

Object Model refers to a visual representation of software or systems' objects, attributes, actions, and relationships. The basic factors of an object model are classes and objects.

❖ *Object:*

An object is a physical component in the object-oriented domain. An object may be tangible such as a person, car, or intangible such as a project.

❖ *Class:*

A class is a representation of objects. It represents a group of objects that have similar properties and exhibit an expected behavior.

Below is an example of a class and a few objects.

Class (Car Brand) = Objects (Toyota, Subaru, Hyundai, Audi, Volkswagen)

An object model uses various diagrams to show how objects behave and perform real-world tasks. The diagrams used include use-case diagram and sequence diagram.

The object model describes objects in object-oriented programming, object-oriented analysis, and object-oriented design.

❖ Elements of the Object model:

Here are the significant features of the object model.

- ***Abstraction***

Abstraction reduces complexity. It comes from the recognizing similarities between objects. Abstraction takes place when the system stress details those that are important to the user. It focuses mostly on the outside view of the object. Data is abstracted when protected by a set of methods, and only those methods can access data in the object.

- ***Encapsulation***

Encapsulation is achieved through information hiding or data hiding to used to reduce complexity and increase reusability. The user cannot see the inside of an object or a class, but the object can be accessed by calling the object's methods.

Encapsulation and Abstraction are complementary concepts. In Abstraction, the system focuses on object behavior and functionality. Encapsulation focuses on implementation that gives rise to action.

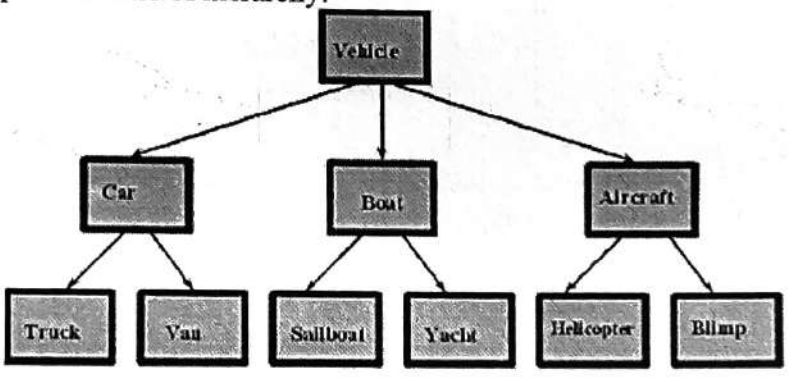
- ***Hierarchy***

The hierarchy shows the order in which objects in a system are put together. It also explains the relationship between different parts of a system. Different properties and functions form a class of the hierarchy.

A hierarchy class is composed of a base class (parent class) and derived classes (subclass). A derived class inherits the properties of a parent class.

Through hierarchy, a class can be composed of inter-related sub-classes, that can have their sub-classes until the smallest level of components is reached.

Here is an example of a class of hierarchy:



- ***Modularity***

Modularity refers to dividing a program into components or modules to reduce the problem's complexity. Modularity takes place on broad and powerful applications with multiple classes.

The modules help to manage complexity. Modularity focuses on implementation. Making modularity and Encapsulation related.

Modularity can be viewed as a way of matching encapsulated abstraction into basic components. This takes place after the partitioning of a system into modules.

UNIT-3

MCA20205-------SYLLABUS

DESIGN ENGINEERING: Design process and design quality, design concepts, the design model.
CREATING AN ARCHITECTURAL DESIGN: Software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams.
TESTING STRATEGIES: A strategic approach to software testing, test strategies for conventional software, black-box and white-box testing, validation testing, system testing, the art of debugging.

DESIGN ENGINEERING

1. Design Process and Design quality

❖ SOFTWARE DESIGN QUALITY GUIDELINES &ATTRIBUTES

S/W DESIGN QUALITY GUIDELINES: -

After analysing and specifying all the requirement the process of software design begins. The goal of software design is to produce high quality software. To develop high quality software, the designer should follow, some of the guidelines. They are

- A design should exhibit an architecture that has been created using recognizable architectural styles/patterns.
- A design should be composed of components that exhibit good design characteristics.
- A design should be implemented in an evolutionary fashion.
- A design should be modular.
- A design should contain distinct representation of data, architecture, interfaces and components.
- A design should lead to data structure that are appropriate for the classes to be implemented.
- A design should lead to components that exhibit independent functional characteristics etc…

❖ DESIGN QUALITY ATTRIBUTES :-

In order to maintain high quality in S/W development, we can use some attributes called Design quality attributes.

These attributes are also called as "FURPS" they are

1. Functionality
2. Usability
3. Reliability
4. Performance
5. Supportability (or) Maintainability

➢ **FUNCTIONALITY**:-The functionality attributes can be checked by assigning a set of features of capabilities

- **USABILITY:** - This attribute can be accessed by knowing the usefulness of various requirements (or) components in the system.
- **RELIABILITY:** - It is a measure of frequency and rate of failure of various components of the system.
- **PERFORMANCE:** - It is a measure that represents the response of the system. Measuring performance means "measuring the processing speed, memory usage, response time & efficiency".
- **SUPPORTABILITY:** - It is also called as maintainability. It is the ability to make enhancement and changes in the software to correct errors, to satisfy all the needs of end users. The maintainability of a given program can be calculated based on the values like Configuration, Compatibility, Testability, Extensibility, the simplicity of installation of program etc..

2. DESIGN CONCEPTS

A set of fundamental S/W design concepts has evolved over the history of S/W engineering . These concepts may be varied over the year and time. The S/W design concept provides a framework for implementing the right S/W.

The various fundamental design concepts are

- **Abstraction: -** when we consider a modular solution to any problem, many levels of abstraction can be defined.
  - **High level** :- In which a solution is defined in broad terms
  - **Low level** :- In which a more detailed description of the solution is provided While moving through different levels of abstraction, procedural & data abstraction are created.

A procedural abstraction refers to a sequence of instructions that have aspecific & limited function

Ex: - Open for a door is the example of a procedural abstraction.

Open implies a long sequence of procedural steps like walk to the door, reach out& group knob, turn knob and pull door, step away from moving door...

A data abstraction is a named collection of data that describes a data object. In the context of procedural abstraction "open" we can define a data abstraction called "door". Like any data object, the data abstraction for door would consist a set of attributesthat describes, the door.

Ex:- door type, swing direction, opening mechanism ...

- **Refinement:-** Stepwise refinement is a top - down design strategy.
  - Refinement is actually a process of elaboration. The architecture of a program is developed by successively refining levels of procedural detail.
  - The process of program refinement is analogous to the process of refinement &partitioning that is used during requirement analysis.
  - Abstraction & Refinement are complementary concepts.
  - Abstraction enables a designer to specify procedure & data. Suppress low-level details. Refinement helps the designer to elaborate low - level details.

- **Modularity: -** Software architecture & design patterns use modularity. The S/W is divided into separately named & addressable components, sometimes called "Modules" that are integrated to satisfy the problem requirements.

Modularity is the single attribute of S/W that allows a program to be intellectually manageable. We modularize a design , so that development can be easily planned , S/W increments can be defined & delivered , changes can be more easily accommodated, tested & debugging can be conduction more efficiently.

- **Architecture** :- Architecture means" representation of overall structure of an integration system (or) S/W ". In architecture various components interact and the data of the structure is used by various components. These components are called System elements

In architectural design various system models can be used. They are..

- **Structural Models** :- In which, the overall architecture of the system can be represented as an organized collection of program components.
- **Framework Models** :- These models shows the architectural framework & corresponding applicability.
- **Dynamic models** :- Theses models shows the reflection of changes on the system due to external events. That means these models address the behavioral aspects of the program architecture.
- **Process Models** :-These sequence of processes and their functioning is represented in this model.
- **Functional Models** :- The functional hierarchy of a system is represented in this model.

- **Information Hiding: -** The information hiding suggests modules can be "characterized by design decisions that hide from all others".

That means modules should be specified & designed, so that information contained with in a module is in accessible to other modules that have no need for such information. The use of information hiding as a design criterion for modular system provides the greatest benefits when modifications are required during testing & S/W maintenance. Because most data & procedure are hidden from other parts of the S/W.

- **Functional independence: -** functional independence is a key to good design & design is the key to S/W quality. It is achieved by modularity, abstraction & information hiding. The functional independence is accessed using two factors. They are
  - cohesion
  - coupling

Cohesion :- It is a natural extension of the information hiding concept. It is a measure of the relative functional strength of a module.
Coupling :- It is a indication of inter connection among modules in a S/W structure.

- **Refactoring** :- It is necessary for simplifying the design without changing the function & behavior.

❖ **THE BENEFITS OF REFACTORING ARE:**

- The redundancy can be achieved.
- Inefficient algorithms can be eliminated (or) can be replaced by efficient one.
- Poorly constructed (or)inaccurate data structures can be removed/replaced
- Other design failures can be rectified.

3. DESIGN MODEL:

The design model can be viewed in two different dimensions.

- **Process Dimension:** Indicates the evolution of the design model as design tasks Are executed as part of the software Process.
- **Abstraction Dimension**: Represents the level of details as each element of the analysismodel is transformed into a design equivalent and then refined iteratively.

➢ **DESIGN MODEL ELEMENTS:**

- **Data Design Elements:** Data Design creates a model of data that is represented at a high level of abstraction.
- **Architectural Design Elements:** Architectural Design gives us an overall view of the software. Architectural model is derived from three sources 1) Information about the application domain for the software to be built. 2) Specific analysis model elements such as DFDs, CRC. 3)The availability of Architectural Patterns.
- **Component Level Design Elements:** This level fully describes the internal details of each software component. It defines data structures for all local data objects and operations.
- **Deployment Level Design Elements:** This level indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

CREATING AN ARCHITECTURAL DESIGN

4. Software architecture:

❖ **ARCHITECTURE & ITS IMPORTANCE:-**

Software architecture provide a holistic view of the system to be built. It represents the structure & organization of S/W components, their properties and the connections between them.

For the following reasons the S/W architecture is important.

- Architectural design represents the structure of data and program components that are required to build a computer-based system.
- It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.
- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a reflective impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together"

5. Data Design :

This section describes data design at both the architectural and component levels. At the architecture level, data design is the process of creating a model of the information represented at a high level of abstraction (using the customer's view of data).

- **Data Design at the Architectural Level**

➢ The challenge is extract useful information from the data environment, particularly when the information desired is cross-functional.

➢ To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information.

➢ However, the existence of multiple databases, their different structures, and the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment.

➢ An alternative solution, called a data warehouse, adds on additional layer to the data architecture.

➢ A data warehouse is a separate data environment that is not directly integrated with day-to-day applications that encompasses all data used by a business.

- **Data Design at the Component Level**

At the component level, data design focuses on specific data structures required to realize the data objects to be manipulated by a component. Refine data objects and develop a set of data abstractions. Implement data object attributes as one or more data structures .Review data structures to ensure that appropriate relationships have been established

Set of principles for data specification:

1. The systematic analysis principles applied to function and behavior should also be applied to data.
2. All data structures and the operations to be performed on each should be identified.
3. A data dictionary should be established and used to define both data and program design.
4. Low level data design decisions should be deferred until late in the design process.
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types

6. Architectural styles:

An architectural style means the overall structure of the software. The architectural styles can be represented as by architecture views such as components, connectors, constraints and Semantic models.

For example a well describing house builder uses common phrases like L-shaped wall, U-Shaped dining hall etc. Here the builder uses different architectural styles as descriptive mechanism because to differentiate the house from other houses.

The elements of the architectural styles are:

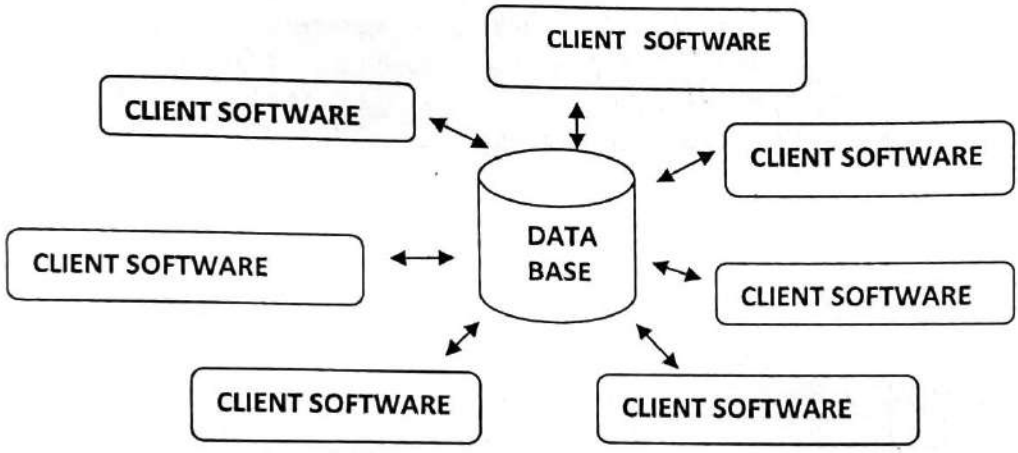
- **Component**: They must perform a function.
- **Connectors:** It enables communication between the components.
- **Constraints:** It defines how the components integrate in to the system.
- **Semantic models:** These elements focus on the designer to understand the properties and values f the system.

The various architectural styles are :

- Data Centered Architectural styles.
- Data Flow (or) Flow-Oriented Architectural syles.
- Call and Return Architectural styles.
- Object Oriented Architectural styles.
- Layered Architectural styles.

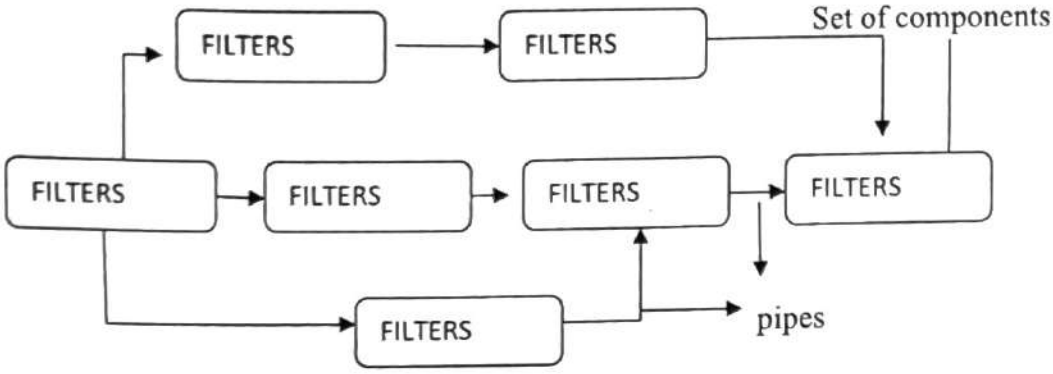
❖ **Data Centered Architectural styles:**

Here a central data store is available, which is accessed by the other client component. The client software access the data independent of any changes to actions of other client software. The data centered architecture is shown below.



❖ **Data Flow (or) Flow-Oriented Architectural styles:**

In this architecture the set of components are called as "FILTERS" and they are connected with pipes. So it is called as "PIPE FILTER ARCHITECTURE". This architecture is applied when input data has to pass through a series of computational components into output data. The pipe filter architecture is shown below:

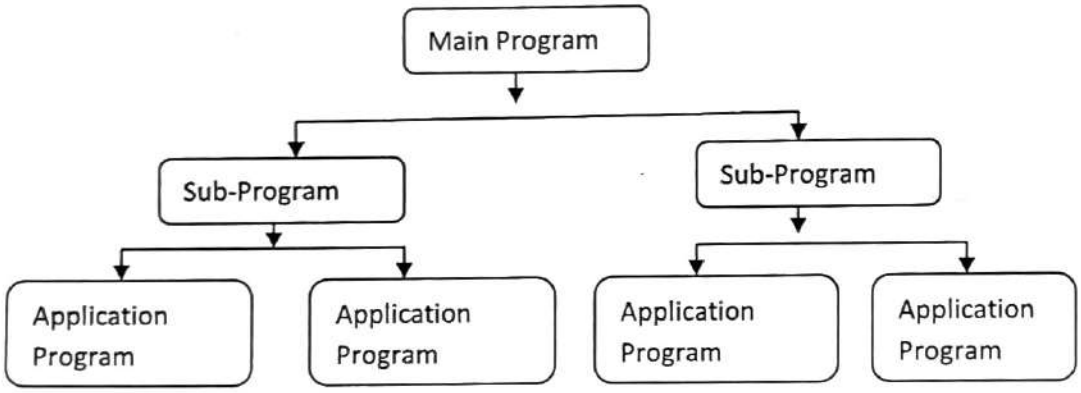


❖ **Call & return architecture:**

This architecture style enables a software designer to achieve a program structure, which can be easily modified. It defines:

- Main program architecture.
- Remote Procedure Call (RPC) architecture.

  - **Main Program Architecture:** It decomposes the functions into control hierarchy. The main program decomposes in to a number of subprograms. The architecture is shown below.

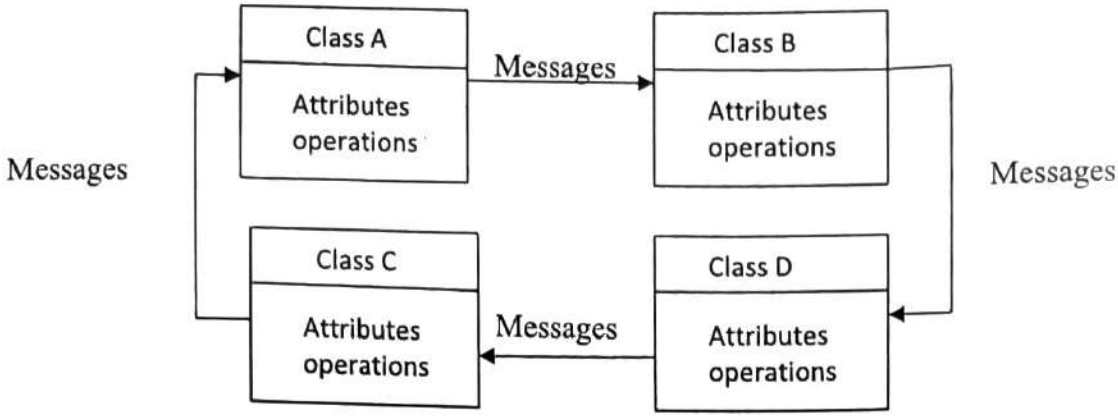


  - **Remote Procedure Call Architecture:**

The programs of main program architecture are distributed across multiple computers on networks.

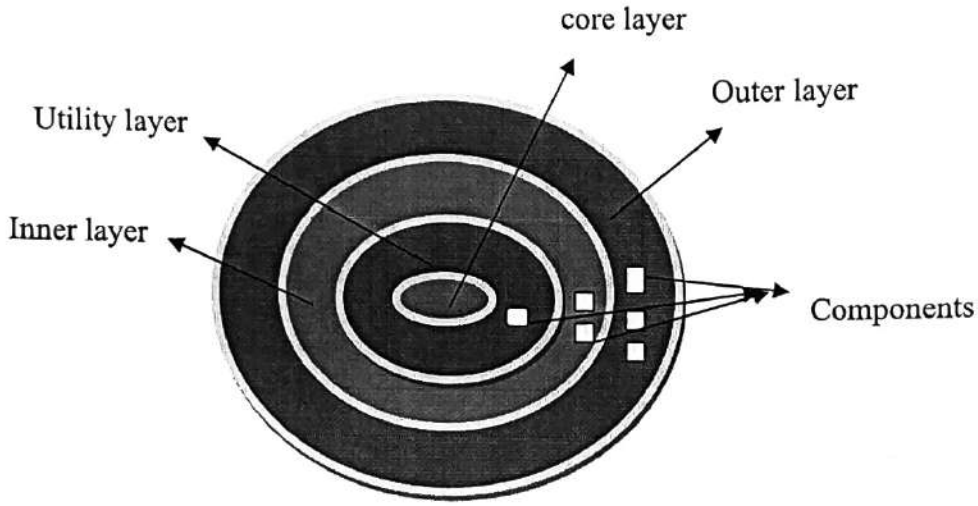
❖ **Object Oriented Architectural styles.**

This architecture contains the classes and their components that encapsulate data and operations through message passing. Here classes can be organized in to a hierarchy by using inheritance. The architecture is shown below.



❖ **Layered Architecture Styles:**

This architecture contains number of different layers. Each layer performs each task. At first the outer layer focus on the components service, user interface operations exist. Inner layer concentrates the components perform operating system exist. Finally the intermediate layer provides utility services. The architecture is shown below.



7. Architectural Patterns:

Architectural patterns define the specific approach for handling some behavioural characteristic of the system.

The Architectural patterns are:

- ❖ Concurrency
- ❖ Persistence
- ❖ Distribution

1. Concurrency: Concurrency consist of different ways in which an application can handles the multiple tasks in a manner that simultaneously parallelism.

For example, one approach is to use operating system process management pattern that provide built-in operating system features that allows the components to execute concurrently. Another approach is , the task scheduler at the application level.

2. Persistence: Persistent data are stored in a database; the data may be read or modified by other process at a later time. Here the data persist, if it serves past the execution of the process that create it.

3. Distribution: The distribution, pattern addresses the manner in which systems or components are communicated with one another in distributed environment.

Thus pattern concentrates on:

- The way in which the entities connect to one another.
- The way in which the nature of communication that occurs.

8. conceptual model of UML

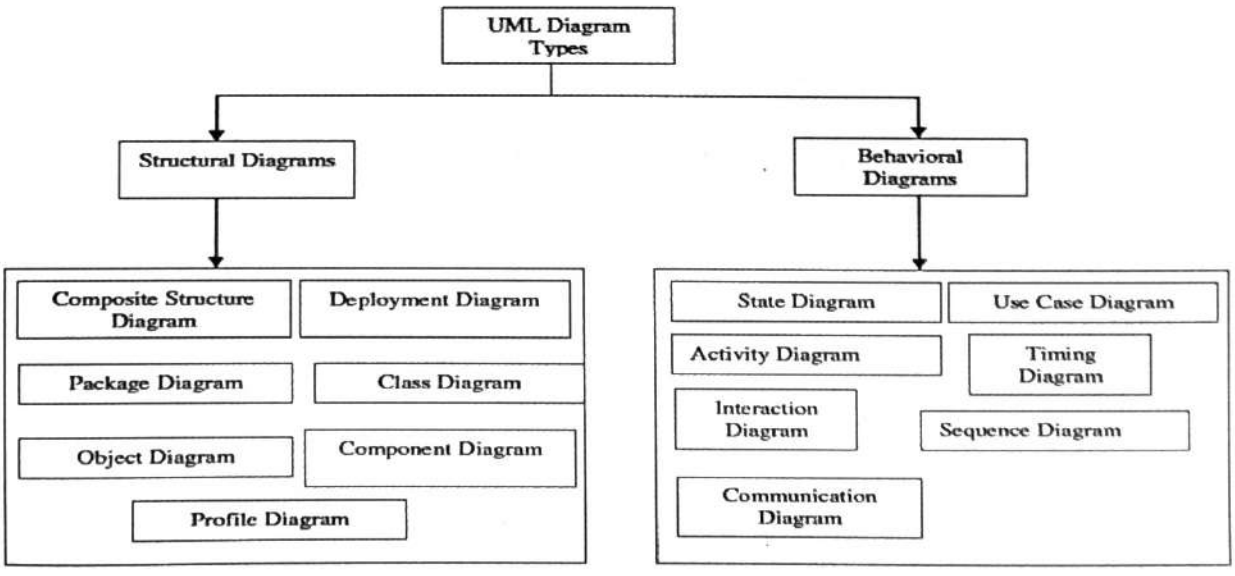
1. INTRODUCTION:

- UML stands for "Unified Modeling Language"
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design .
- Use graphical notation to communicate more clearly than natural language (imprecise) and code (too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

2. Basic structural modeling:

The UML has numerous types of diagrams, each providing a certain view of the system. One must understand both the structure and the function of the objects involved. One must understand the individual behavior of objects, and the dynamic behavior of the system as a whole.

UML diagrams can be classified into two groups: Structure diagrams and Behavior diagrams.



❖ **Structure Diagrams :**

- These diagrams show the static structure of elements of the system. They may depict such things as the architectural organization of the system, the physical elements of the system, its runtime configuration, and domain-specific elements of your business.
- Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

❖ **Behavior Diagrams :**

- Behavior diagrams emphasize on what must happen in the system being modeled.
- Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.
- Behavior diagrams express the dynamic behavioral semantics of a problem or its implementation through the following additional diagrams.

3. Component Diagrams:

A component represents a reusable piece of software that provides some meaningful aggregate functionality. Each class in the system must live in a single component or at the top level of the system. A component may also contain other components.

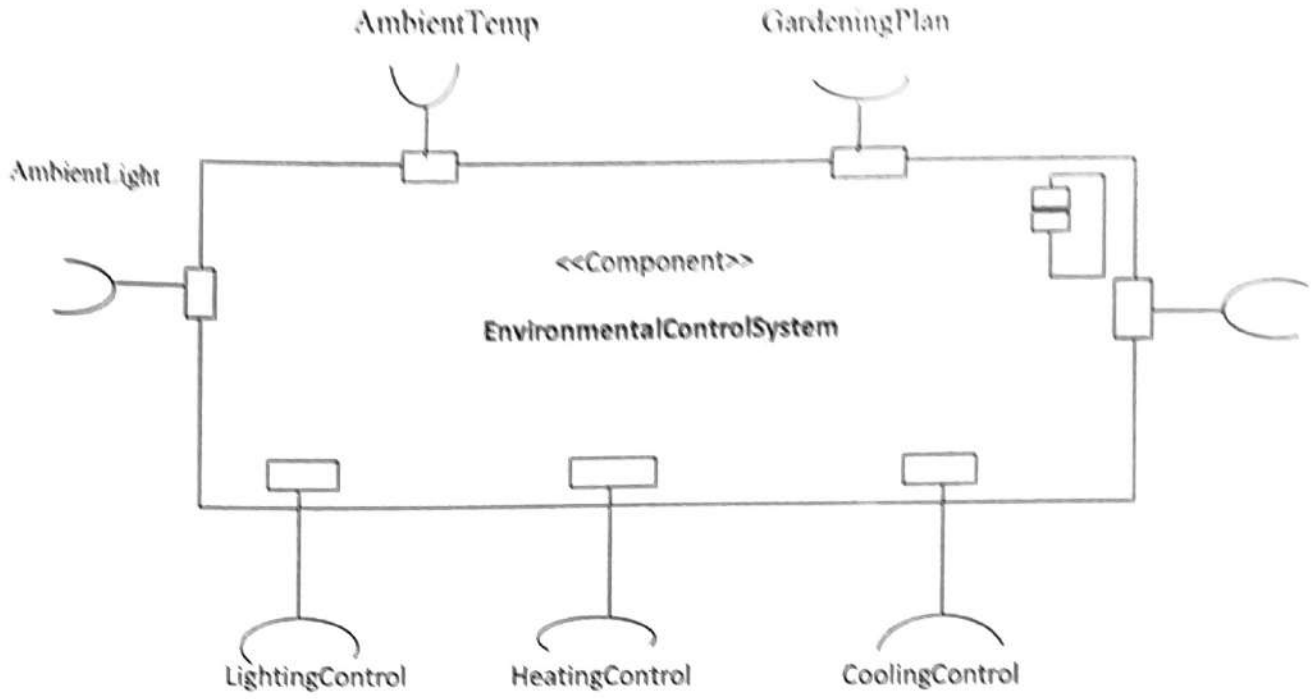
A component collaborating with other components through well-defined interfaces to provide a system's functionality, may itself be comprised of components that collaborate to provide its own functionality.

The essentials of a component diagram are component Notation, component interfaces and component realizations.

➢ Essential: The component notation:

Components are a type of structured classifier; its detailed assembly can be shown with a composite structure using parts, ports, and connectors. In below component its name, EnvironmentalControlSystem , is included within the classifier rectangle in bold lettering, using specific naming convention defined by the development team. The keyword label <<component>> and the component icon shown in the upper right-hand corner of the classifier rectangle.

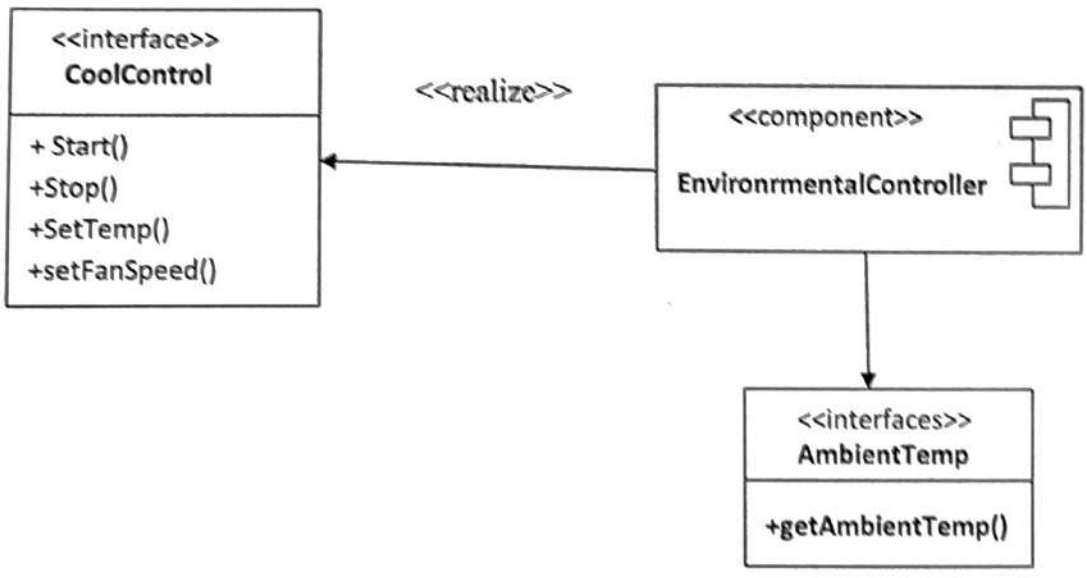
On the boundary of the classifier rectangle, we have seven ports, which are denoted by small squares. These seven ports are unnamed but should be named, in the format of port name : port type. The port type is optional when naming a port.



➢ **Essential: Component interfaces:**

The interfaces specify the functionality that the component will provide to its environment. In above figure, Lightingcontrol is an example of a provided interface. AmbientTemp is one of the required interfaces.

In below figure clearly explain the interfaces in component diagram. In below figure focuses on only two of the seven interfaces of EnvironmentalController: Coolcontrol and AmbientTemp. Realizes the CoolControl interface; The functionality of interface coolcontrol is starting, stopping, setting the temperature , and setting the fan speed for any component using the interface , as shown by the contained operations.

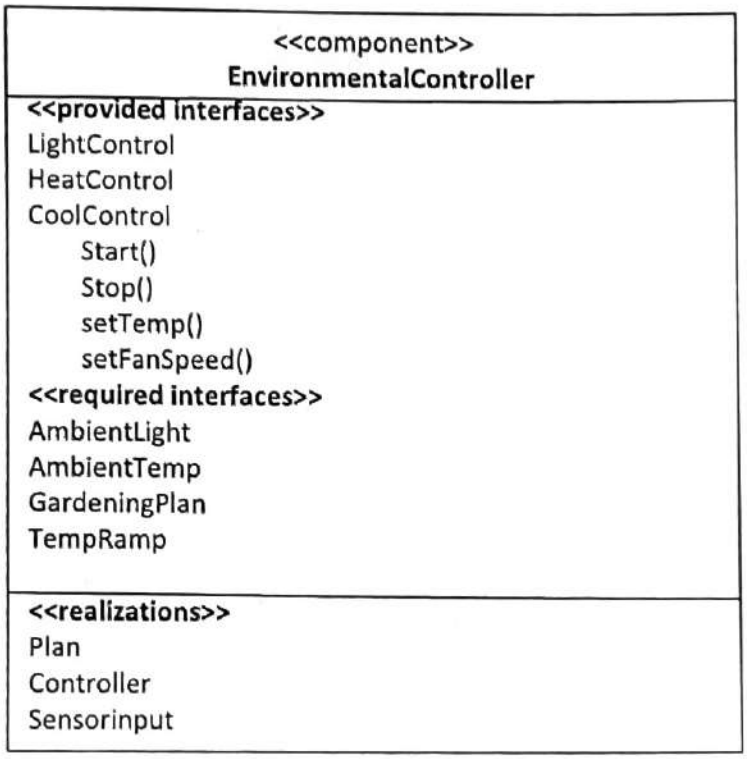


➢ **Essentials: component Realizations:**

The component realization provide all of the functionality advertised by its provided interfaces. In below figure specifies that the EnvironmentalController component is realized by the classes plan,controller, and sensorInput. These three classes provide all of the functionality advertised by its provided interfaces. But ,In doing so, they require the functionality specified by its required interfaces.

This realization relationship between the EnvironmentalController component is realized by the classes plan, Controller and SensorInput classes shown in below figure.

Here, we see a realization dependency from each of the classes to EnvironmentalController.



4. Use Case Diagrams:

Use case diagrams are used to describe a set of actions (use cases) that some system or systems should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.

❖ **Purpose:**

1. Used to gather requirements of a system
2. Used to get an outside view of a system
3. Identify external and internal factors influencing the system
4. Show the interaction among the requirements through actors.

❖ **Uses:**

1. Requirement analysis and high level design

2. Model the context of a system

The essential elements of use-case diagram are: Actors, Use cases and the use case diagram.

➢ Essentials: Actors:

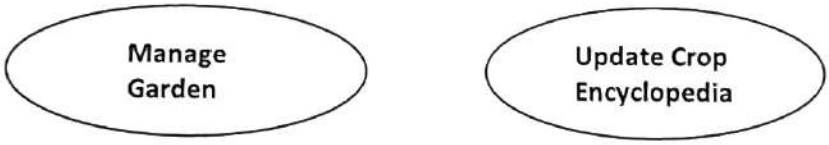
Actors are entities that interface with the system. They can be people or other systems. Actors, which are external to the system they are using depicted as stylized stick figures In below figure shows two actors for the Hydroponics Gardening system.

One way to think of actors is to consider the roles the actors play. In the real world, people (and systems) may serve in many different roles; for example, a person can be a salesperson, a manager, a father, an artist etc.

➢ Essentials: Use Cases:

Use cases represent *what* the actors want your system to do for them. The below figure shows some use cases, shown as Ovals, for the Hydroponics Gardening system.

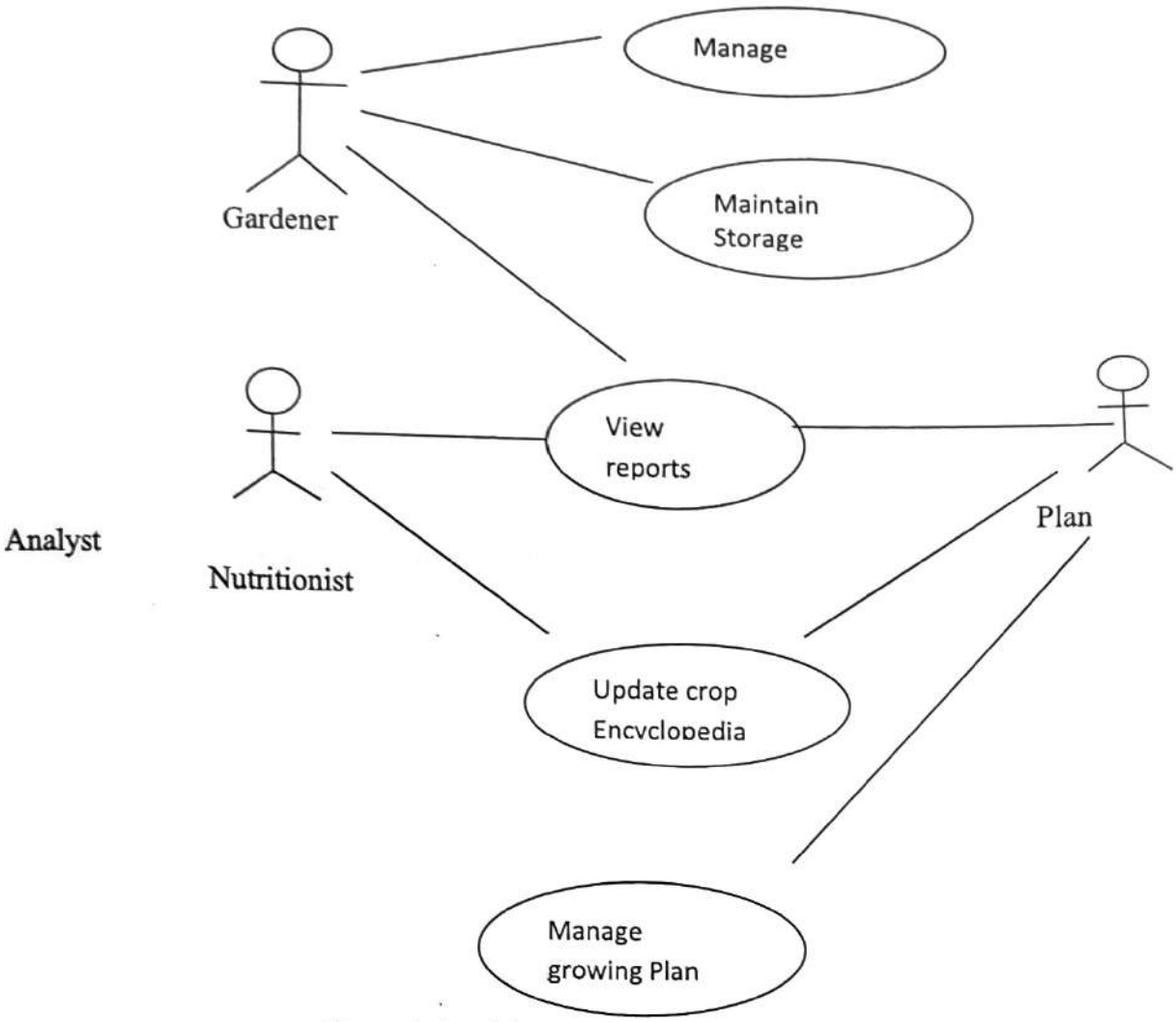
A use case must be a *complete* flow of activity, from the *actor's point of view*, that provides *value* to the actor.



A use case is a specific way of using the system by using some part of the functionality….A use case is thus a special sequence of related transactions performed by an actor and the system.

➢ Essentials: The use case Diagram:

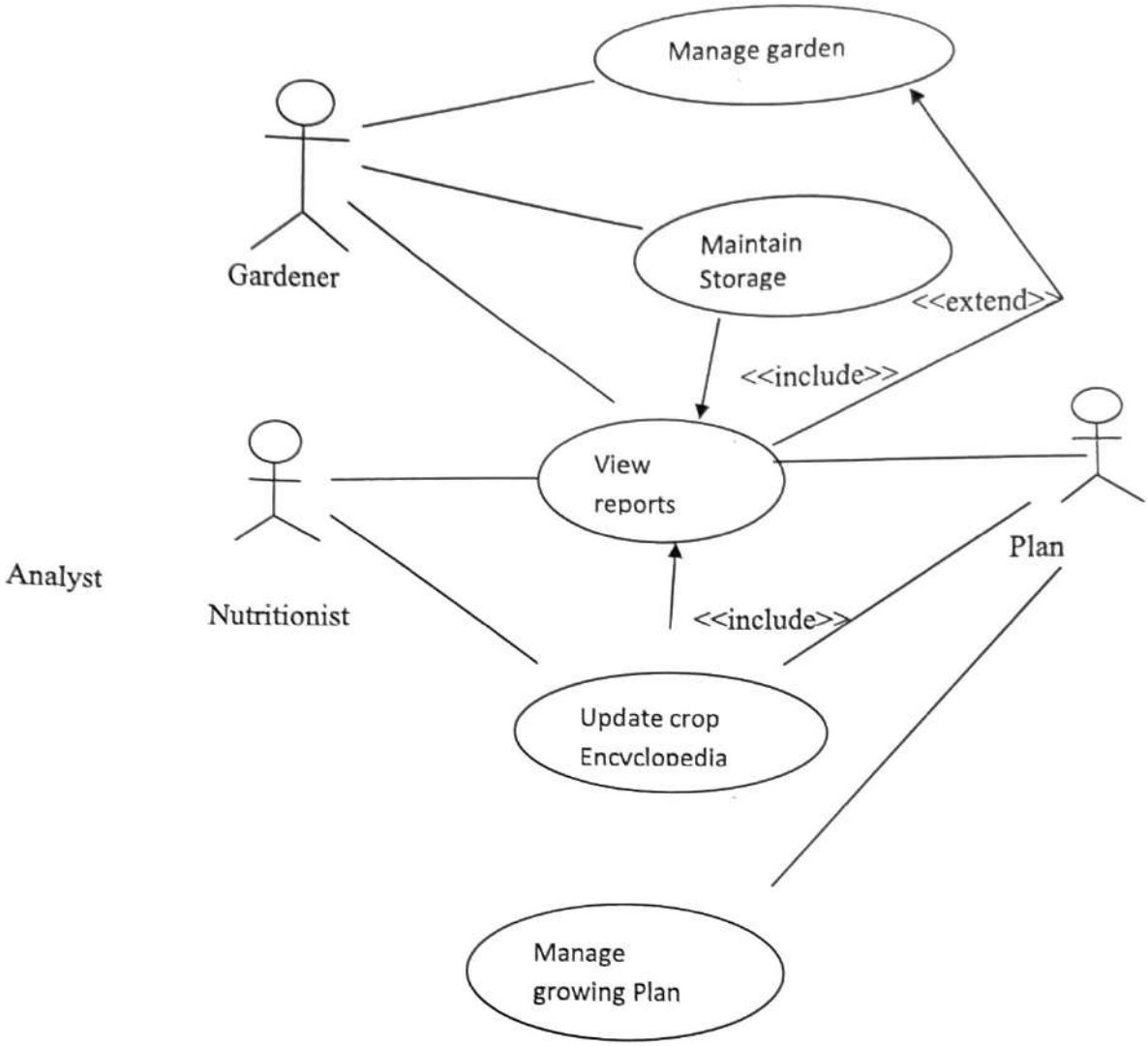
To show which actors use which use cases, you can create a use case diagram by connecting them via basic associations, shown by lines. The associations in the use case diagram indicate which actors initiate which use cases. Here we see that only the Gardener actor can maintain the storage tanks, but all the actors may review reports.



Two relationships used primarily for organizing use case models are both powerful and commonly misused: the <<include>> and <<extend>> relationships. These relationships are used between use cases.

Where an included use case is executed, it is indicated in the use case specification as an inclusion point. The inclusion point specifies where, in the flow of the including use case, the included use case is to be executed.

Where an extending use case is executed, it is indicated in the use case specification as an extension point. The extension point specifies where, in the flow of the including use case, the extending use case is to be executed.



A use case diagram showing an <<include>> and <<extend>> relationships

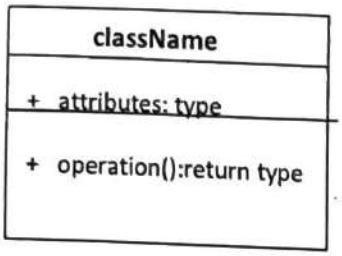
5. Class Diagrams:

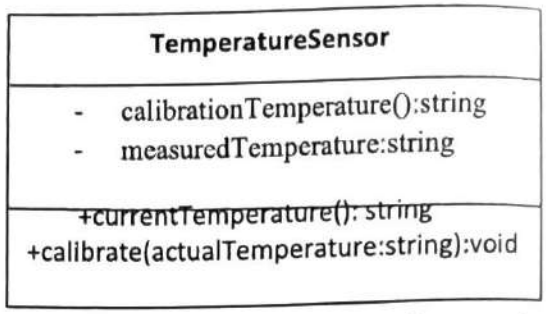
A class diagram is used to show the existence of classes and their relationships in the logical view of a system. A single class diagram represents a view of the class structure of a system. During analysis, we use class diagrams to indicate the common roles and responsibilities of the entities that provide the system's behaviour.

Two essentials elements of a class diagram are class notation and their basic relationships.

- **Essentials: The class Notation:**

The below figure shows the icon used to represent a class in a class diagram and an example from our hydroponics Gardening System. The class icon consists of three compartments. With the first occupied by the class name , the second by the attributes, and the third by the operations.





A name is required for each class and must be unique to its enclosing namespace .By convention, the name begins in capital letters, and the space between multiple words is omitted .The first letter of the attribute and operation names is lowercase, with subsequent words starting in uppercase, and spaces are omitted just as in the class name.

- Attribute specification format:

Visibility attributeName: Type[multiplicity]=DefaultValue{property string}

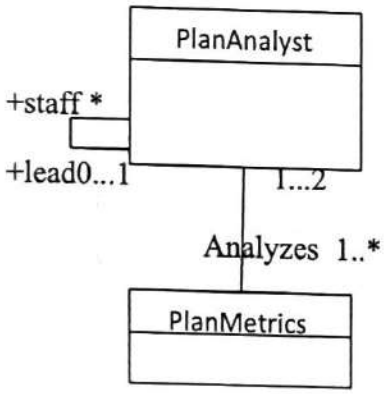
- Operation specification format:

Visibility operationName(parameterName:Type):ReturnType {property string}

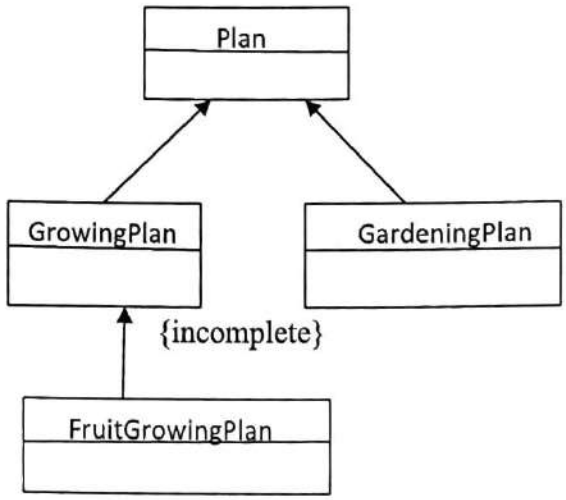
➢ **Essentials: Class Relationships:**

The essential connections among classes include association, generalization, aggregation, and composition. Each such relationship may include a textual label that documents the name of the relationship. The below diagram shows the class relationship icons.

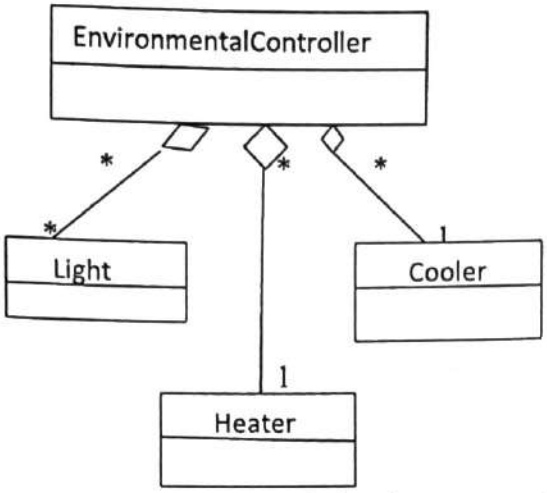
ASSOCIATION



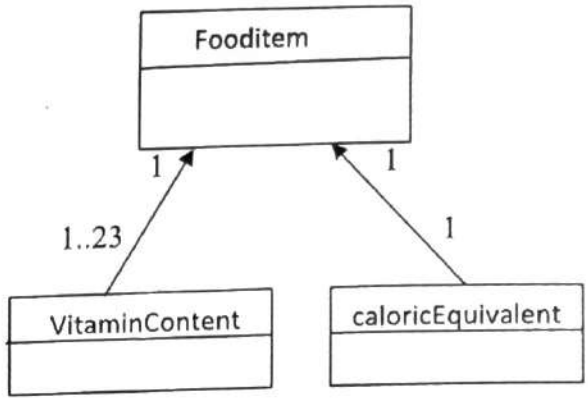
GENERALIZATION



AGGREGATION



COMPOSITION



Association icon connects two classes. Associations may be further adorned with their multiplicity, using the syntax in the following examples.

- 1 Exactly one
- * Unlimited number(zero or more)
- 0..* Zero or more
- 1..* One or more
- 0..1 Zero or one
- 3..7 specified range

The generalization denotes a " is a" relationship. The arrowhead points to the superclass, and the opposite end of the association designates the sub class. Aggregation as manifested in the "part of" relationship, is a constrained form of the more general association relationship.

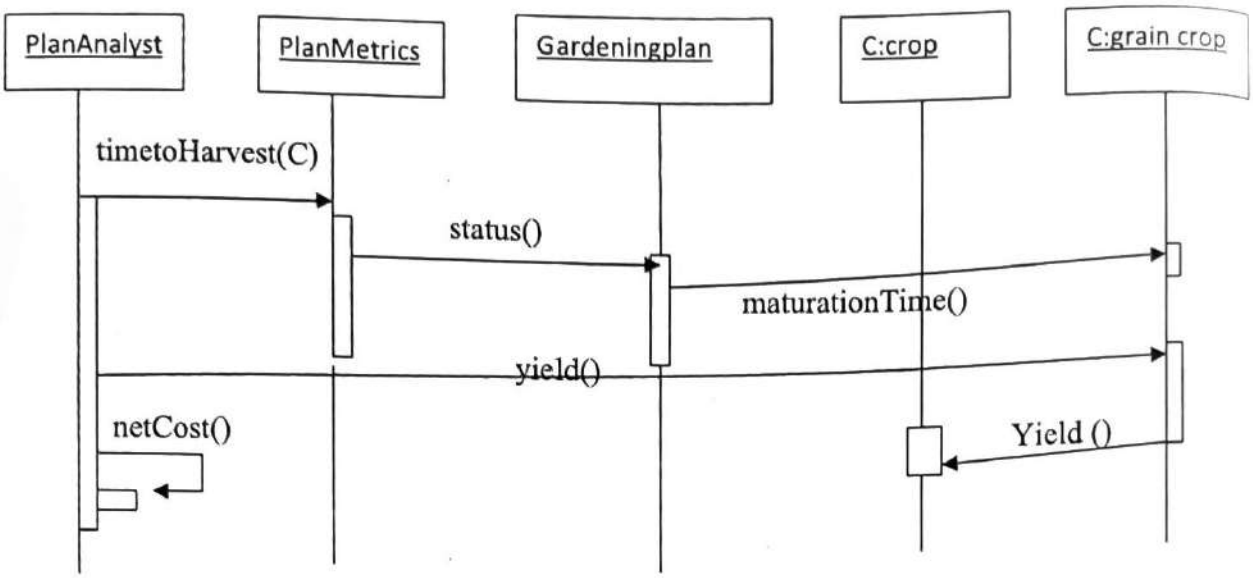
6. Sequence diagram:

A sequence diagram is used to trace the execution of a scenario in the same context as a communication diagram. A sequence diagram is simply way to represent a communication diagram.

The essential elements of sequence diagrams are objects and Interactions, lifelines and Messages.

➢ **Essentials: Objects and Interactions:**

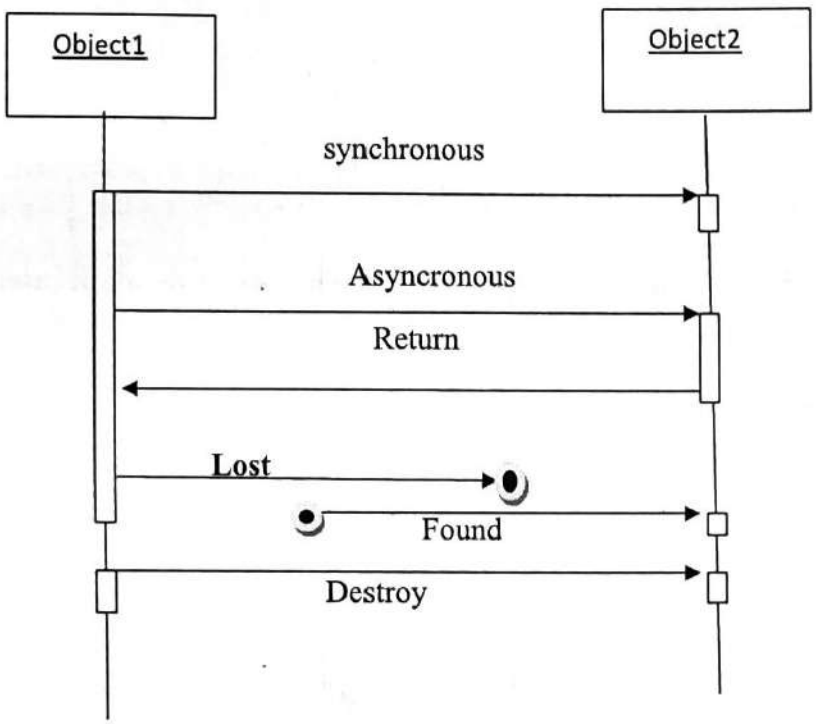
The advantage of using a sequence diagram is that it is easier to read the passing of messages in relative order. Early sequence diagrams tend to focus on events as opposed to operations because events help to define the boundaries of a system under development.



➢ **Essentials: Lifelines and Messages:**

In sequence diagram , the entities which are same as for object diagrams are written horizontally across the top of the diagram. A dashed vertical line, called the "lifeline", is drawn below each object. These indicate the existence of the object.

Messages, which may denote events or the invocation of operation are shown horizontally. The endpoints of the message icons connect with the vertical lines that connect with the entities at the top of the diagram. Messages are drawn from the sender to the receiver. Ordering is indicated by vertical position, with the first message shown at the top of the diagram, and the last message shown at the bottom. The notation used for messages indicates the type of message being used as in below figure.



X

Asynchronous message(typically an operation call) is shown as a solid with a filled arrowhead. An asynchronous message has a solid line with an open arrowhead. A return message uses a dashed line with an open arrowhead. A lost message appears as a synchronous message that terminates at an end point(a black dot). A found message appears as a synchronous message that originates at an end point symbol.

7. Collaboration diagrams:

The collaboration diagram is used to show the relationship between the objects in a system. Both the sequence and the collaboration diagrams represent the same information but differently. Instead of showing the flow of messages, it depicts the architecture of the object residing in the system as it is based on object-oriented programming.

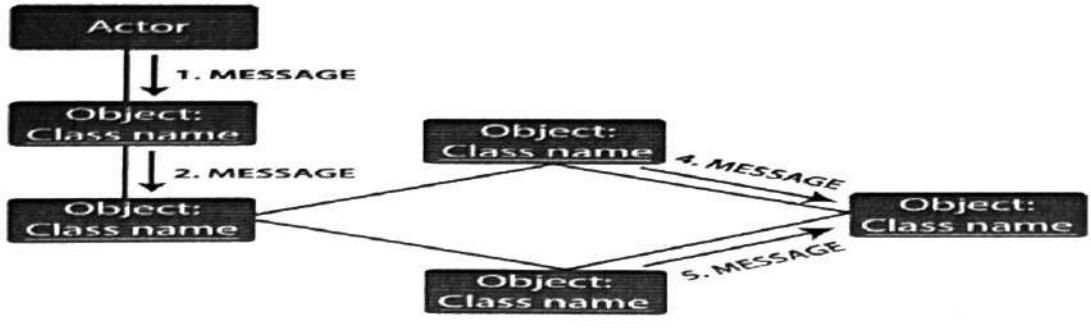
An object consists of several features. Multiple objects present in the system are connected to each other. The collaboration diagram, which is also known as a communication diagram, is used to portray the object's architecture in the system.

➢ Notations of a Collaboration Diagram

Following are the components of a component diagram that are enlisted below:

1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
   In the collaboration diagram, objects are utilized in the following ways:
   - The object is represented by specifying their name and class.
   - It is not mandatory for every class to appear.
   - A class may constitute more than one object.
   - In the collaboration diagram, firstly, the object is created, and then its class is specified.
   - To differentiate one object from another object, it is necessary to name them.
2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.

Components of a collaboration diagram



➢ **Steps for creating a Collaboration Diagram**

1. Determine the behaviour for which the realization and implementation are specified.
2. Discover the structural elements that are class roles, objects, and subsystems for performing the functionality of collaboration.
   - Choose the context of an interaction: system, subsystem, use case, and operation.

➢ **Benefits of a Collaboration Diagram**

1. The collaboration diagram is also known as Communication Diagram.
2. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails..
3. The special case of a collaboration diagram is the object diagram.
4. It focuses on the elements and not the message flow, like sequence diagrams.
5. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.

Testing strategies

8. A Strategic approach to software testing:

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing-a set of steps into which we can place specific test case design techniques and testing methods-should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent testgroup.
- Testing and debugging are different activities, but debugging must be accommodated in any testingstrategy.

1. Verification and Validation:

Software testing is one element of a broader topic that is often referred to as *verification and validation* (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this anther way.

- Verification: Are we building the product right?
- Validation: Are we building the right product?

2. Organizing for software testing:

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who built the software are now asked to test the software.

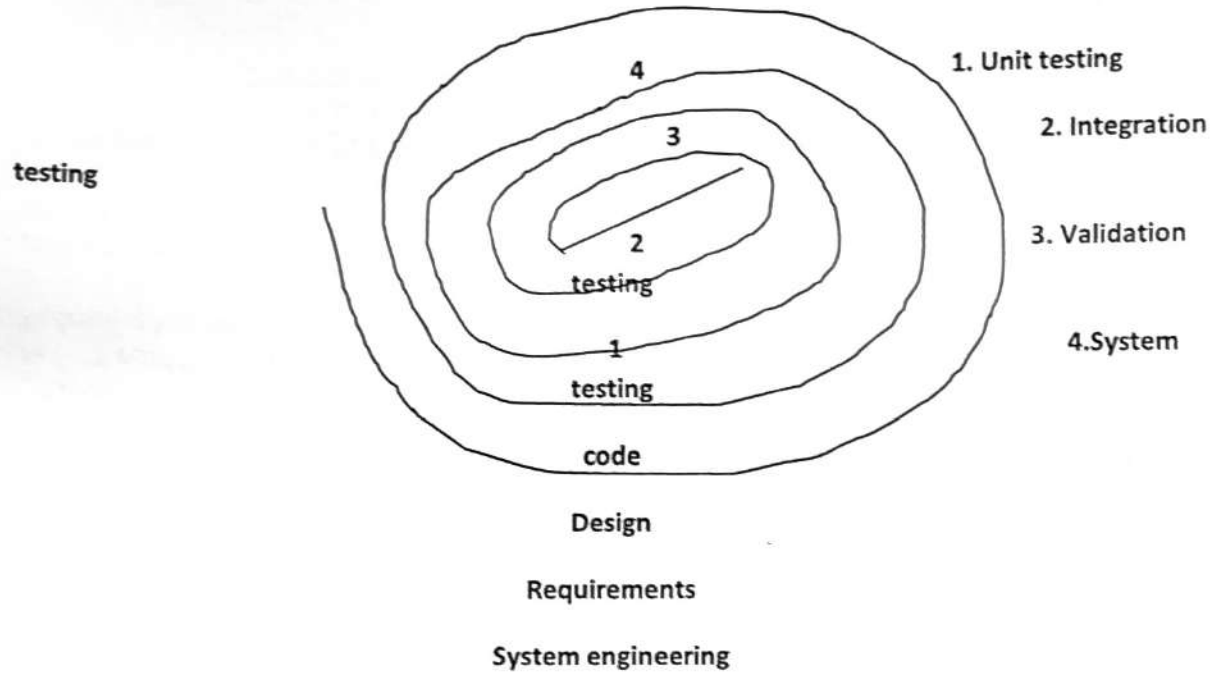
The software developer is responsible for testing the individual units of the program, ensuring that each performs the function or exhibits the behavior for which it was designed. In many cases the developer also conducts integration testing. Only after the software architecture is complete does an

independent test group (ITG) become involved.

The role of ITG is to remove the inherent problems associated with letting the builder test the thing that has been built. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

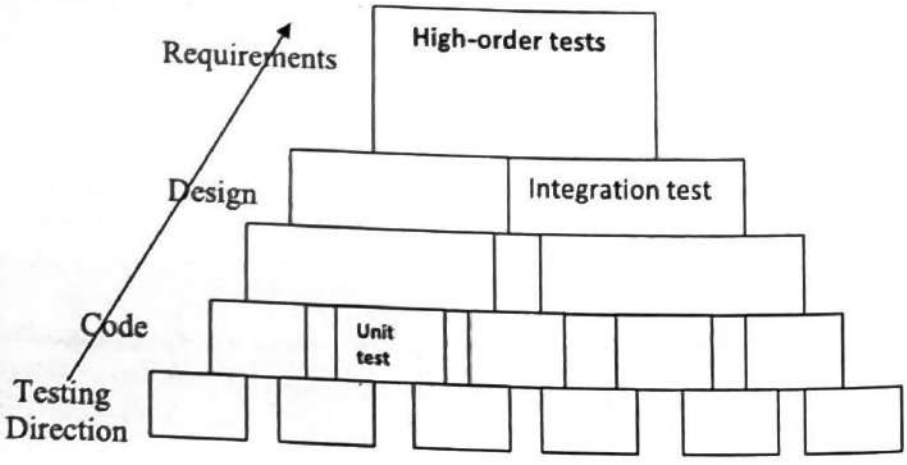
3. A software testing strategy for conventional software architectures:

The software process may be viewed as the spiral illustrated in below figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.



A Strategy for software testing may also be viewed in the context of the spiral in above figure. **Unit testing** begins at the vortex and concentrates on the each unit of the software a implemented in source code. Testing progresses by moving outward along the spiral to **integration testing**, where the focus is on design and the construction of the software architecture.

Take another turn outward on the spiral, we encounter **validation testing,** where requirements established as a part of software requirements analysis are validated against the software has been constructed. Finally, we arrive at **System testing**, where the software and other system elements are tested as a whole.



4. A software testing strategy for object-oriented architectures:

The testing of object-oriented systems presents a different set of challenges for the software engineer. The completeness and consistency of object-oriented representations must be assessed as they are built.

Unit testing loses some of its meaning, and integration strategies change significally. In summary, both testing strategies and testing tactics must account for the unique characteristics of object oriented software.

The overall approach for object oriented software is identical in philosophy to the one applied for conventional architectures, but differs in approach. We begin with 'testing in the small' and work outward toward 'testing in the large'.

As classes are integrated into an object-oriented architecture, a series of regression tests are run to uncover errors due to communication and collaboration between classes and side effects caused by the addition of new classes.

5. Criteria for completion of testing:

A classic question arises every time software testing is discussed. When are we done testing?- how do we know that we have tested enough? Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

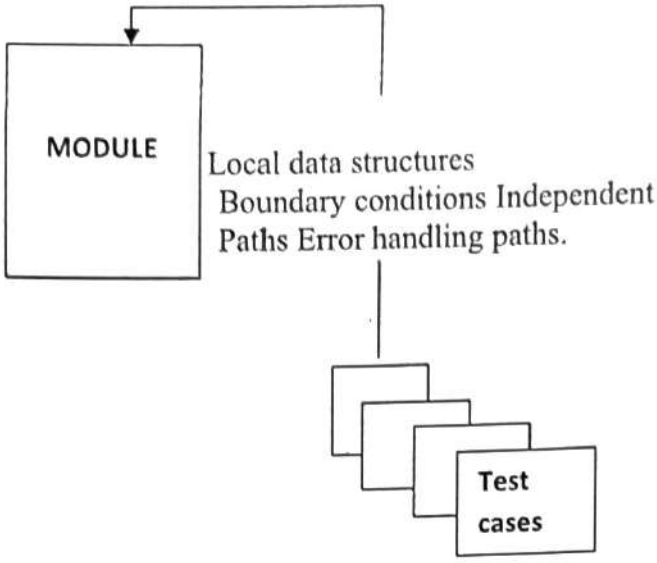
One response to the question is : You are never done testing; the burden simply shifts from you to your customer. Every time the customer/user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities.

Another response is : you are done testing when you run out of time or you run out of money.

9. Test strategies for conventional software:

There are many strategies that can be used to test software. Each of these classes of tests is described in the sections that follow.

➢ **Unit Testing:**



- Unit is the smallest part of a software system which is testable it may include code files, classes and methods which can be tested individual for correctness.
- Unit is a process of validating such small building block of a complex system, much before testing an integrated large module or the system as a whole.
- Driver and/or stub software must be developed for each unit test a driver is nothing more than a "main program" that accepts test case data, passes such data to the component, and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested.
- A stub or "dummy subprogram" uses the subordinate module's interface.

➢ **Integration Testing:**

- Integration is defined as a set of integration among component.
- Testing the interactions between the module and interactions with other system externally is called Integration Testing.
- Testing of integrated modules to verify combined functionality after integration.
- Modules are typically code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.
- Types of integration testing are:
  1. Top-down integration
  2. Bottom-up integration
  3. Regression testing
  4. Smoke testing

➢ **Validation Testing:**

- The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.
- Validation testing provides final assurance that software meets all informational, functional, behavioral, and performance requirements.
- The **alpha test** is conducted at the developer's site by a representative group of end users. The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
- Alpha tests are conducted in a controlled environment.
- The **beta test** is conducted at one or more end-user sites.
- Unlike alpha testing, the developer generally is not present.
- Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer.

➢ **System Testing**

- In system testing the software and other system elements are tested as a whole.
- System testing verifies that all elements mesh properly and that overall system function/performance is achieved.
- **Recovery testing** is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), re initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.
- **Security testing** attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. .
- **Stress testing** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- A variation of stress testing is a technique called sensitivity testing.
- **Performance testing** is designed to test the run-time performance of software within the context of an integrated system.
- **Deployment testing,** sometimes called configuration testing, exercises the software in each environment in which it is to operate.

➢ **Acceptance Testing :**

- Acceptance Testing is a level of the software testing where a system is tested for acceptability.
- The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
- It is a formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.
- Acceptance Testing is performed after System Testing and before making the system available for actual use.

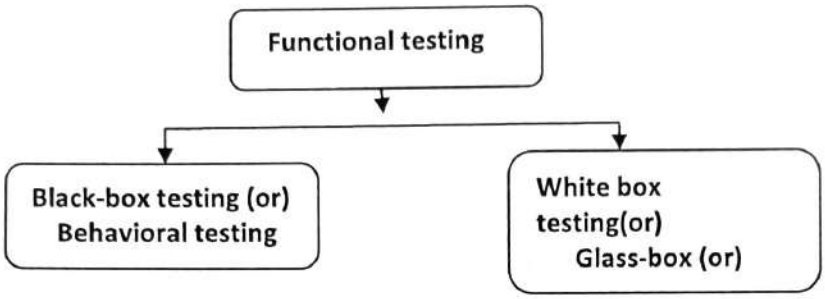
10. FUNCTIONAL TESTING:

"Testing is a set of activities, whose objective is to find out the errors, that were made unknowingly while designing and constructing the software."

Functional Testing:

It is a testing technique that is used to test the features and functionalities of the software. There are two types of functional testing techniques . They are:

- Black Box (or) Behavioural testing.
- White Box (or) Glass Box (or) Structural testing.



- **Black Box (or) Behavioural testing:**

"This testing ignores the internal mechanism of the system. It focus only on the output generated in response to selected inputs and execution condition."

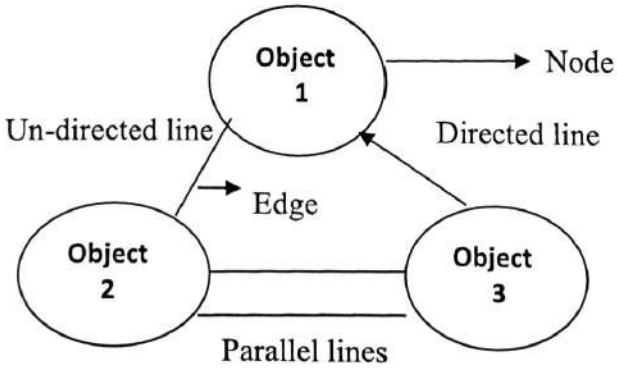
There are different testing techniques are included in black-box testing. They are:

- Graph based testing method.
- Equivalence partitioning testing method.
- Boundary value analysis (BVA) testing method.
- Orthogonal array testing method.

1. Graph based testing method:

Graph s a collection of nodes and edges. In this testing method the nodes are represented the objects and the edges represents the relationship between the objects.

In this testing method, the first step is to understand the object and the next step is to verify all the objects and finally expected the relationship between the objects.



2. **Equivalence portioning testing method:**

This testing method divides the input domain of a program into classes of data from which test case can be derived. For example.,

- If an input requires a specific value then three equivalence partitions are defined, if (i==10) then the three classes are(i<10, i==10,i>10).
- If an input condition requires a range then the three equivalence partitions are defined, suppose marks[0,100] then Minimum, within a range, maximum.

3. **Boundary value analysis testing method:**

In this testing method, that compliments the equivalence partitioning and it leads to the selection of test cases at the edges of the class.

In an input specify the range [a , b] then the test cases are min-1,min,min+1,normal values,max-1,max,max+1.

Ex: i={5,10,12,20}, then the test cases are { 4,5,6,10,12,19,20,21}.

4. **Orthogonal array testing method:**

The testing can be applied to the problems in which the input domain is relatively small, but too large to accommodate this testing. It is also called as Exhaustive testing method.

For example a system has two input items x and y, each of these input items has two discrete values associated with it. They are x^2 i.e., 2^2=4 possible test cases.

❖ ***White Box (or) Glass Box (or) Structural testing:***

"This testing technique mainly focuses on the internal mechanism of the system."

It derives the following test cases:

- Independent paths with in a module, have been executed atleast once.
- Execute all logical decisions on their true and false values.
- Execute all loops at their boundaries..

They are two types of white box testing techniques. They are:

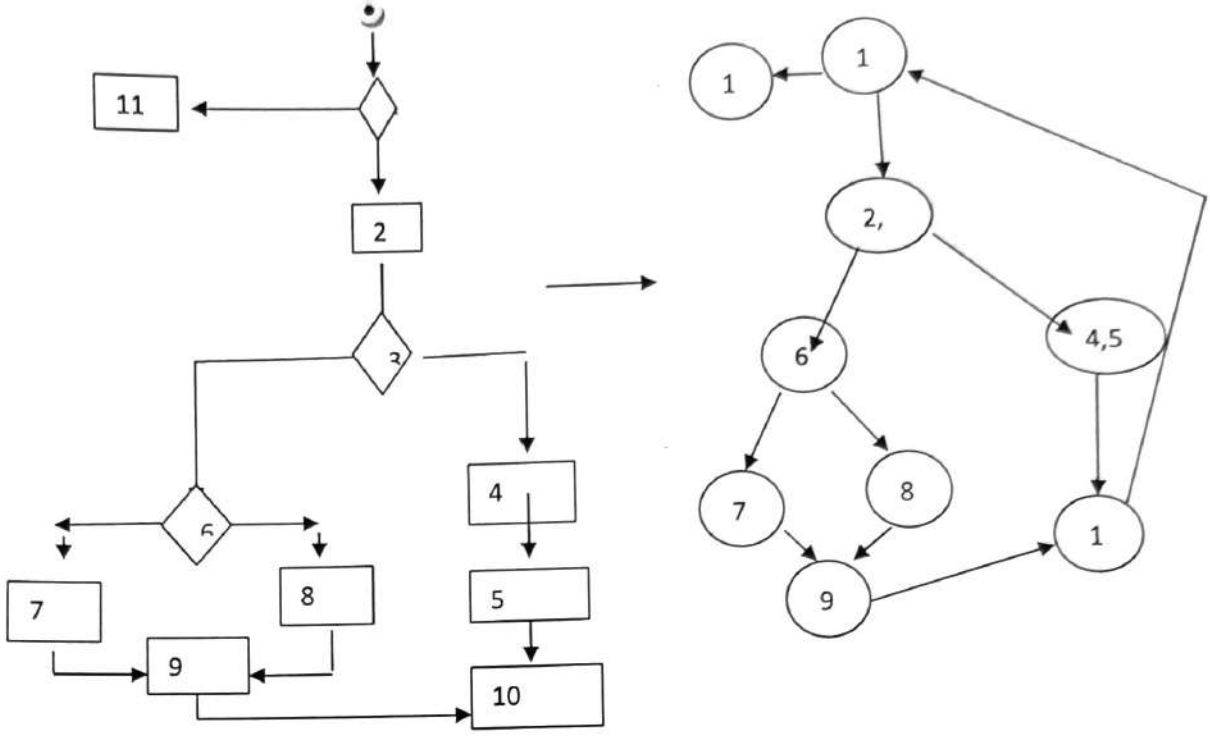
1. Basis path testing technique.
2. Control structures testing technique.

- **Basis path testing method:**

It is proposed by TOM M.C Cabe. It defines the basis set of execution paths. It includes 1. Flow graph notation 2. Cyclomatic complexity.

- **Flow graph notation:** Here the system module flow chart can be converted into flow graph.

Path1:1-11
Path2:1-2-3-6-7-9-10-1-11
Path3: 1-23-6-7-8-9-10-1-11
Path4: 1-2-3-6-8-9-10-1-11



Therefore , number of independent paths are:::::4

- **Cyclomatic complexity:**
  - How many independent paths have to be find out is obtained from the cyclomatic complexity. It is denoted by V(G).

    V (G) = E-N+2

    Here, E= number of edges

    N= number of nodes.

    In above graph, V (G)=11-9+2=13-9=4

    Number of independent paths= 4.
- In other words , cyclomatic complexity is represented as V (G)=P+1

Here P=Predicate node, it means a node that contains condition.

In above graph , predicate nodes are: 3

V (G)= 3+1=4

Number of independent paths=4

- **Control structure testing:**

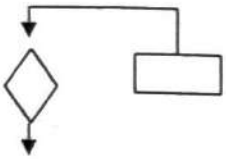
By using this testing, to improve the quality of white box testing. It includes:

- Condition testing.
- Data flow testing.
- Loop testing.

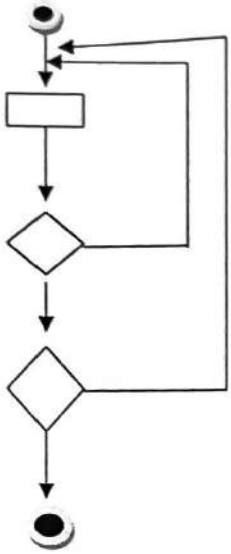
1. **Condition testing**: This testing checks the logical conditions of the program module. The logical conditions are <, <=, >, >=,==.!= etc.

2. **Data flow testing:** This testing follows the path of a program according to locations of definitions and use of variables in the program.
3. **Loop testing:** This testing mainly focus on the validity of the loop structure. It includes, Simple loop testing, Nested loop testing and Concatenated loop testing.

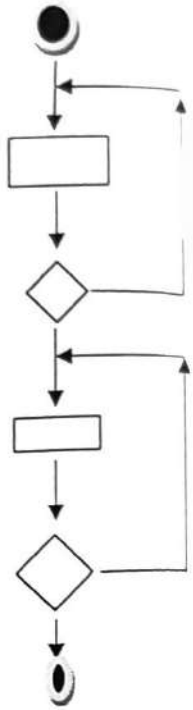
Simple loop



nested loop



concatenated loop



11. System testing :

System testing is a series of different tests, whose purpose is to verify the system elements have been properly integrated and performing allocating the functions. In this testing first the software components are individually tested and finally they are integrated and tested.

The following types of system tests are performed. They are:

1. Recovery testing
2. Security testing
3. Stress testing.
4. Performance testing.

- **Recovery testing:**
  This testing focuses, the software to fail in a variety of ways and verifies the recovery is properly performed or not. If recovery is automatic, re-initialization and restart are evaluated for correctness.

- **Security testing:**
  This testing mainly focuses, the protection mechanism and it protects the system from improper functions.

Security testing checks the system for invulnerability from frontal attacks as well as rear attacks. This testing always protects the sensitive information from hackers, employees or an individual attempt to enter the system for information.

❖ **Stress testing:**

This testing checks the system performance in abnormal situation. For example software is smoothly running, when an average hundred transactions are done. Suddenly the transaction rises up to thousands or lakhs. In those situations, some period of time the system may Hault that is called stress point. The stress testing is to check the stress point is accepted or not.

❖ **Performance testing:**

Thos testing is designed to test the runtime performance of the system and also this testing test the both the hardware and software instrumentation.

For example, an application is tested to use proper resources that are processing speed, response time, memory usage etc.

12. Validation testing:

Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer.

❖ **Validation Test Criteria**

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human-engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible condi-tions exist:

(I) The function or performance characteristics conform to specification and are accepted or

(2) a deviation from specification is uncovered and a *deficiency list* is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

❖ **Configuration Review**

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been

properly developed, are cataloged, and have the necessary detail to bolster the sup-port phase of the software life cycle. The configuration review, sometimes called an *audit*

❖ **Alpha and Beta Testing**

The ***alpha test*** is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The ***beta test*** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be con-trolled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.

As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire cus-tomer base.

13. ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error.

1. The Debugging Process:

Debugging is not testing but always occurs as a consequence of testing.
The debugging process will always have one of two outcomes:
(1) the cause will be found and corrected, or
(2) the cause will not be found. In the latter case, the per-son performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

2. CHARACTERSTICS OF BUGS:

I. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.
2. The symptom may disappear (temporarily) when another error is corrected. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies)
3..The symptom may be caused by human error that is not easily traced.
4. The symptom may be a result of timing problems, rather than processing problems.

5. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which inputordering is indeterminate).
6. The symptom may be intermittent. This is particularly common in embedded systems that couplehardware and software inextricably.
7. The symptom may be due to causes that are distributed across a number of tasks running on differentprocessors

Debugging Approaches

(1) Brute force, (2) Backtracking, and (3) Cause elimination.

- The ***brute force*** category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information thatis produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!
- ***Backtracking*** is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential back-ward paths may become unmanageably large.
- The third approach to ***debugging-cause elimination-is*** manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolatepotential causes.

--------All the best---------

UNIT-1V
MCA20205-------SYLLABUS

PRODUCT METRICS: Software quality, metrics for analysis model, metrics for design model, metrics for source code, metrics for testing, metrics for maintenance.
RISK MANAGEMENT: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM, RMMM plan.
QUALITY MANAGEMENT: Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000 quality standards.

RISK MANAGEMENT

1. SOFTWARE RISKS:

"Risk is the probability of suffering a loss. Risk provides an opportunity to develop the project better". The main difference between the risk and the problem is, the problem is some event which has already occurred but risk is something that is unpredictable, without a reason.

There are different types of software risks are mentioned below:

- Project risk
- Business risk
- Technical risk
- Known risk
- Unpredictable risk

➢ **Project risk:**
- ✓ These risks are identified by personal problems .
- ✓ These are identified by project planning.

➢ **Business risk:**
- ✓ These are identified the losing support of senior management.
- ✓ These are identified at the losing of budgetary commitment.
- ✓ Building a product, that has no longer period fit in to business strategy.
- ✓ They threaten the feasibility of the software to be built
- ✓ If they become real, they threaten the project or the product

➢ **Technical risk:**

These are identified at the improper designing , verification, maintenance and monitoring of the product.

➢ **Known risk:**

The known risk can occur and cover the careful evolution of project plan and these are identified by the developers.

➢ **Unpredictable Risk:**

These are identified after the project delivery and these are extremely difficult to be identified in advanced stages.

2. Re-active and Pro-active risk strategies:

"Risk is the probability of suffering a loss. Risk provides an opportunity to develop the project better". The main difference between the risk and the problem is, the problem is some event which has already occurred but risk is something that is unpredictable, without a reason.

❖ ***Steps for risk management:***

- Identify the risk, recognise, what can go wrong.
- Analyse the risk to estimate the probability that it will be occur impact.
- Rank the risk, by probability and impact.
- Develop a plan to manage the risk having high probability and high impact.

➢ ***Reactive risks:***

1. It depends on past accidental analysis and responses.
2. It attempts to reduce the tendency of the same or similar accidents which happened in past.
3. It includes creative thinking, problem solving ability of humans in its approach . It makes less flexible changes and challenges.
4. It is also known as "Response based risk management approach".

➢ ***Pro-active risks:***

1. It depends on present and future analysis and responses
2. It attempts to reduce the tendency of the any accident happening in present and future.
3. It includes creative thinking, problem solving ability of humans in its approach. It makes more flexible changes and challenges.
4. It is also known as "Adaptive, closed loop, feedback control risk management approach".

3. Risk Management Process:

"Risk is the probability of suffering a loss. Risk provides an opportunity to develop the project better."

➢ ***Risk identification:***

- The project manager identifies the software risks. He takes the first step toward for avoiding the software risk.
- The method for risk identification is creating risk item check list.
- The method focus on customer characteristics, business impact, technology etc.

➢ ***Risk Table:***

The project manager identifies a table and creates a table known as Risk table. It contains five coloumns. The risk table explains number of risks, risk categories, risk probabilities and their impacts.

Risk	category	probability	Impact	RMMM

In above table,

1. The first coloumn contains all types of risks.
2. Each risk is categorized in the second coloumn of a table.
3. The probability of occurrence of each risk is entered in the third coloumn of a table. Here the high probability risks drops to top of the table, and the low probability risks are drops to bottom of the table.
4. The impact of each risk is mentioned in the fourth coloumn of a table. The impact depends on the performance, cost, support and schedule.
5. The last coloumn of a table contains RMMM (Risk Mitigation Monitoring Management) plan provides and identifies the risks which are already solved.

4. ***Risk Identification***
   - One method for identifying risks is to create a risk item checklist.
   - The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:
   - **Product size**—risks associated with the overall size of the software to be built or modified.
   - **Business impact**—risks associated with constraints imposed by management or the marketplace.
   - **Stakeholder characteristics**—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
   - **Process definition**—risks associated with the degree to which the software process has been defined and is followed by the development organization.
   - **Development environment**—risks associated with the availability and quality of the tools to be used to build the product.
   - **Technology to be built**—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
   - **Staff size and experience**—risks associated with the overall technical and project experience of the software engineers who will do the work.

5. **Risk Estimation / Risk Projection**
   - ❖ ***Risk Projection Steps***

- Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
- Explain the consequences of the risk
- Estimate the impact of the risk on the project and product
- Note the overall accuracy of the risk projection so that there will be no misunderstandings.

❖ ***Risk Mitigation, Monitoring, and Management***

- Risk mitigation (proactive planning for risk avoidance)
- Risk monitoring (assessing whether predicted risks occur or not, ensuring risk aversion steps are being properly applied, collect information for future risk analysis, attempt to determine which risks caused which problems)
- Risk management and contingency planning (actions to be taken in the event that mitigation steps have failed and the risk has become a live problem)
- The goal of the risk mitigation, monitoring and management plan is to identify as many potential risks as possible.
- When all risks have been identified, they will then be evaluated to determine their probability of occurrence,
- Plans will then be made to avoid each risk, to track each risk to determine if it is more or less likely to occur, and to plan for those risks should they occur.
- It is the organization's responsibility to perform risk mitigation, monitoring, and management in order to produce a quality product.
- The quicker the risks can be identified and avoided, the smaller the chances of having to face that particular risk's consequence.
- The fewer consequences suffered as a result of good RMMM plan, the better the product, and the smoother the development process.

❖ ***Risk Mitigation***

- 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, and 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.

❖ ***Risk Monitoring***

- 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, inter-personal 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.

- The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

❖ ***Risk Management***

- Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality.
- 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."

QUALITY MANAGEMENT

1. Software Quality Assurance (SQA):

Software quality is defined as "Conformance to explicitly stated functional and performance requirements, documents and development standards, and implicit characteristic that are expected of all professionally developed software."

Software requirements are the foundation from which quality is measured. Generally there are two types of requirements. They are:

- Implicit requirements.
- Explicit requirements.

These are also called as direct and indirect requirements. Generally SQA group includes software engineers, product managers, customers, sales representatives and individuals, who serve with in a SQA group.

The goal of SQA is to remove errors in the software. The problems are referred to various names such as bugs, errors and defects. An error is the quality problem, it found before the software is released to the client. A defect is also the quality problem, found after the software has been released to end user.

SQA activities:

- ❖ Prepare an SQA plan for a project.
- ❖ Participate in the development of the projects software process description.
- ❖ Conducting REVIEWS, it means inspections and Demo's and verifies the engineering activities with the defined software process.
- ❖ Conducting Auditions (official inspection), it means official inspection and verifies he software work products.
- ❖ Records and any complaints and reports to senior management.

User Satisfaction= Complete product+ Quality+ Deliver the product within budget and Schedule.

2. Software Reviews (Formal Technical Reviews) :

A formal technical review (FTR) is a software quality control activity performed by software engineers (and others).

- **The objectives of an FTR are:**

(1) To uncover errors in function, logic, or implementation for any representation of the software.
(2) To verify that the software under review meets its requirements.
(3) To ensure that the software has been represented according to predefined standards.
(4) To achieve software that is developed in a uniform manner.
(5) To make projects more manageable.

➢ ***Review Reporting and Record Keeping***

- During the FTR, a reviewer (the recorder) actively records all issues that have been raised.
- These are summarized at the end of the review meeting, and a review issues list is produced. In addition, a formal technical review summary report is completed.

➢ ***Review Guidelines***

- Guidelines for conducting formal technical reviews must be established in advance, distributed to all reviewers, agreed upon, and then followed.
- A review that is un-controlled can often be worse than no review at all. Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and denial:
- Speak problem areas, but don't attempt to solve every problem noted.
- Take written notes.
- Limit the number of participants and insist upon advance preparation.
- Develop a checklist for each product that is likely to be reviewed.
- Allocate resources and schedule time for FTRs
- Conduct meaningful training for all reviewers.
- Review your early reviews.

➢ ***Sample-Driven Reviews***

- In an ideal setting, every software engineering work product would undergo a formal technical review.
- In the real word of software projects, resources are limited and time is short.
- As a consequence, reviews are often skipped, even though their value as a quality control mechanism is recognized.

3. Software Reliability :

Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

➢ ***Measures of Reliability***

- A simple measure of reliability is meantime-between-failure (**MTBF**):

  MTBF = MTTF + MTTR

- Where the acronyms MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively. Many researchers argue that MTBF is a far more useful measure than other quality-related software metrics. An end user is concerned with failures, not with the total defect count.
- Because each defect contained within a program does not have the same failure rate, the total defect count provides little indication of the reliability of a system.
- An alternative measure of reliability is failures-in-time (FIT) a statistical measure of how many failures a component will have over one billion hours of operation.

➢ **Software Safety**

- Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- A modeling and analysis process is conducted as part of software safety.
- Although software reliability and software safety are closely related to one another, it is important to understand the subtle difference between them.
- Software reliability uses statistical analysis to determine the likelihood that a software failure will occur.
- Software safety examines the ways in which failures result in conditions that can lead to an accident

4. Statistical software quality assurance:

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

- Information about software defects is collected and categorized.
- An attempt is made to trace each defect to its underlying cause (ex: non-conformance to specifications, design error, poor communication with the customer).
- Using the Pareto Principle(80 percent of the defects can be traced to 20 percent ofvall possible causes).

➢ **Six Sigma for software engineering:**

- Six sigma is " generic quantitative approach to improvement that applies to any process"
- "Six Sigma is a disciplined, data-driven approach and methodology for eliminating in any process from manufacturing to transactional and from product to service"
- To achieve six sigma a process must not produce more than 3.4 defects per million opportunities.

- Six sigma have two methodologies

(1) DMAIC (Define, Measure, Analyze, Improve, Control)

- Define: Define the problem or process to improve upon related to the customer and goals
- Measure: How can you measure this process in a systematic way?
- Analyze: Analyze the process or problem and identify the way in which it can be improved. What are the root causes of problems within the process?
- Improve: Once you know the causes of the problems, present solutions for them and implement them
- Control: Utilize Statistical Process Control to continuously measure your results and ensure you are improving

  Several Software Packages available to assist in measuring yield, defects per million opportunities, etc.

(2) DMADV: (Define, Measure, Analyze, Design, Verify)

- Define, Measure and analyze are similar to above method.
- Design: Avoid root causes of defects and meet the customer requirements.
- Verify: To verify the process, compare the process with the standard plan and find differences.

5. The quality standards ISO 9000 and 9001:

ISO 9001

- In order to bring quality in product & service, many organizations are adopting Quality Assurance System.
- ISO standards are issued by the International Organization for Standardization (ISO) in Switzerland.
- Proper documentation is an important part of an ISO 9001 Quality Management System.
- ISO 9001 is the quality assurance standard that applies to software engineering. It includes, requirements that must be present for an effective quality assurance system.
- ISO 9001 standard is applicable to all engineering discipline.
- The requirements delineated by ISO 9001:2000 address topics such as

➢ Management responsibility
➢ quality system
➢ contract review design control document
➢ data control
➢ product identification Traceability
➢ process control inspection
➢ Testing
➢ preventive action
➢ control of quality records internal quality
➢ Audits
➢ Training Servicing
➢ Statistical techniques.

- In order for a software organization to become registered to ISO 9001:2000, it must establish policies and procedures to address each of the requirements just noted (and others) and then be able to demonstrate that these policies and procedures are being followed.

PRODUCT METRICS

Introduction:

Although product metrics for computer software are often not absolute, they provide us with a systematic way to assess quality based on a set of clearly defined rules. They also provide the software engineer with on-the-spot, rather than after-the-fact insight.

1. Software Quality:

Software quality is a complex mix of factors that will vary across different applications and the customers who request them.

- Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
- Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not allowed, lack of quality will almost surely result.

❖ **McCall's quality factors:**

The factors that affect software quality can be categorized in two broad groups.

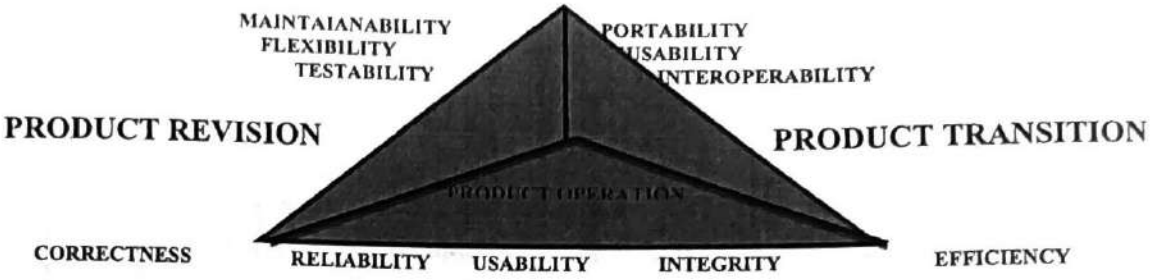
1. Factors that can be directly measured.
2. Factors that can be measured only indirectly.

Quality Factors:

- **Correctness:** satisfies its specification and fulfils the customer's mission objectives.
- **Reliability:** satisfies the program can be expected to perform its intended functions with required precision.
- **Efficiency**: The amount of computing resources and code required by a program to perform its function.
- **Integrity**: The extent to which access to software or data by unauthorized persons can be controlled.
- **Usability**: The effort required to learn, operate, prepare input for, and interpret output of a program.
- **Maintainability**: The effort required to locate and fix an error in a program.
- **Flexibility**: The effort required to modify an operational program.
- **Testability**: The effort required to test a program to ensure that it performs its intended function.

- **Portability**: The effort required to transfer the program from one hardware and/or software system environment to another.

- **Reusability**: The extent to which a program can be reused in other applications.



❖ **ISO 9126 quality factors:**

The ISO 9126 standard was developed in an attempt to identify quality attributes for computer software. The standard identifies six key quality attributes.

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

2. Metrics for the Analysis Model:

These metrics examine the analysis model with the intent of predicting the "size" of the resultant system. Size is sometimes an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort.

❖ **Function- Based Metrics:**

The function point metric (FP) , first proposed by Albrecht, can be used effectively as a means for measuring the functionality delivered by a system.. using this data, the FP can be used to:

- Estimate the cost or effort required to design , coding, and test the software.
- Predict the number of errors that will be encountered during testing.
- Forecast the number of components and / or the number of projected source lines in the implemented system.

The information domain values are defined in the following manner:

- **Number of external inputs:** The inputs are often used to update the internal logical files, and also provide the distinct application-oriented data.
- **Number of external outputs:** The external output is derived within the application, and provides information to the user.

- **Number of external inquires:** An external enquiry is defines as an online input that results in the generation of some immediate software response in the form of an online output.
- **Number of internal logical files:** Each internal logical file is a logical grouping of data that resides within the application boundary and is maintained vai external inputs.
- **Number of external interface files:** Each external file is a logical grouping of data that resides external to the application, but provides data that may be of use to the application.

Once these data have been collected, the below table is completed and a complexity value is associated with each count.

To compute function points (FP), the following relationship is used .:

FP= count total * [0.65+0.01*€ (F_i)]

Where count total is the sum of all FP entries obtained the below table.

Weighting factor

Information domain value	Count		Simple	Average	Complex		
External inputs	3	x	3	4	6	=	9
External outputs	2	X	4	5	7	=	8
External inquiries	2	X	3	4	6	=	6
Internal logical files	1	X	7	10	15	=	7
External interface files	4	X	5	7	10	=	20
Count total	→						50

The count total shown in above table must be adjusted using equation

FP=count total * [0.65+0.01*€ (F_i)]

Where count total is the sum of all FP entries obtained from above table and F_i (i=1 to 14) are value adjustment factors. For the purpose of this example, we assume that *€ (F_i) is 46. Therefore,

FP=50*[0.65+(0.01*46)]=56

Based on the projected FP value derived from the analysis model, the project team can estimate the overall implemented size of the *safehome* user interaction function.

3. Metrics for the Design model:

It is inconceivable that the design of a new aircraft, a new computer chip, or a new office building would be conducted without defining design features, determining

metrics for various aspects of design quality, and using them to guide the manner in which the design evolves.

❖ **Architectural Design Metrics:**

These are focus on the characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components with in the architecture.

The scientists Card and Glass define three software design complexity measures: structural complexity, data complexity and system complexity.

- For hierarchical architectures **structural complexity** of a module I is defined in the following manner:

$$S(i) = f^2_{out}(i)$$

Where $f_{out}(i)$ is the fan out of module. Fan-out is defined as the number of modules immediately subordinate to the module I, that is the number of modules that are directly invoked by module.

- **Data complexity** provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = V(i) / [f_{out}(i) + 1]$$

Where v(i) is the number of input and output variables that are passed to and from module i.

- Finally , **System complexity** is defined as the sum of structural and data complexity specified as

$$C(i) = S(i) + D(i).$$

As each of these complexity values increases, the overall architecture complexity of the system also increases.

❖ **Metrics for object-oriented design:**

In a detailed treatment of software metrics for OO systems, Whitmire describes nine distinct and measurable characteristics of an OO design.

- **Size:** Size is defined in terms of four views: population, volume, length and functionality. **Population** is measured by taking a static count of OO entities such as classes or operations. **Volume** measures are identical to population measures but are collected dynamically- at a given instant of time. **Length** is a measure of a chain of interconnected design elements. **Functionality** metrics provide an indirect indication of the value delivered to the customer by an OO application.
- **Complexity:** Complexity defines the structural characteristics by examing how classes of an OO design are interrelated to one another.
- **Coupling:** The physical connections between elements of the OO design (e.g., the number of collaboration between classes).
- **Sufficiency:** It defines "the degree to which an abstraction possesses the features required of it.

- **Completeness:** Completeness considers multiple points of view, asking the question: what properties are required to fully represent the problem domain object?.
- **Cohesion:** It defines the manner, that has all operations working together to achieve single, well- defined purpose.
- **Primitiveness:** Primitiveness is the degree to which an operation is atomic-that is, the operation cannot be constructed out of a sequence of other operations contained with in a class.
- **Similarity**: The degree to which two or more classes are similar in terms of their structure, function, behaviour, or purpose is indicated by this measure.
- **Volatility:** Volatility of an OO design component measures the likelihood that a change will occur.

4. Metrics for source code:

Halstead assigned quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated once again is complete. The measures are:

n_1 = the number of distinct operators that appear in a program
n_2 = the number of distinct operands that appear in a program.
N_1 = the total number of operator occurrences.
N_2 = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume, the program level, the language level, and other features such as development effort, development time, and even the projected number of faults in the software.

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2).$$

It should be noted that V will vary with programming language and represents the volume of information required to specify a program.

5. Metrics for Testing:

Although much has been written on software metrics for testing, the majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

❖ Halstead Metrics Applied to testing:

Testing effort can also be estimated using metrics derived from Halstead measures. Using the definitions for program volume, V, and program level. PL, Halstead effort,e, can be computed as.

$$PL = 1/ [(n_1/2)*(N_2/n_2)]$$
$$e = V/PL.$$

The percentage of overall testing effort to be allocated to a module K can be estimated using the following relationship:

Percentage of testing effort (k) = $e(k)/\sum e(i)$

Where e(k) is computed for module k using equations and the summation in the denominator of equation is the sum of halstead effort across all modules of the system.

❖ **Metrics for Object-Oriented Testing:**

The OO design metrics provide an indication of design quality. They also provide a general indication of the amount of testing effort required to exercise an OO system.

- **Lack of cohesion in methods(LCOM):** The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.
- **Percent Public and Protected (PAP):** This metric indicates the percentage of class attributes that are public or protected. Tests must be designed to ensure that such side effects are uncovered.
- **Public access to data members (PAD):** This metric indicates the number of classes that can access class attributes. Tests must be designed to ensure that such side effects are uncovered.
- **Number of root classes (NOR):** This metric is a count of the distinct class hierarchies that are described in the design model. Tests suites for each root class and the corresponding class hierarchy must be developed.

6. **Metrics for Maintenance:**

All of the software metrics are used to development of new software and the maintenance activities have been proposed. IEEE suggests a software maturity index (SMI) that provides an indication of the stability of a software product. The following information is determined:

M_r = the number of modules in the current release.

F_c= the number of modules in the current release that have been changed.

F_a= the number of modules in the current release that have been added.

F_d= the number of modules from the preceding release that were deleted in the current release.

The software maturity index is computed in the following manner:

$SMI=[M_r - (F_a + F_c + F_d)]/ M_r$.

SMI may also be used as a metric for planning software maintenance activities.