Software Engineering

UNIT I

Software Engineering:

Software Engineering (SE) is Systematic, Disciplined approach for the Development, Maintenance and Retirement of the software.

CHARACTERISTICS OF SOFTWARE

There are three Characteristics of software they are

1. Software is developed or engineered; it is not manufactured in the classical sense.
2. Software doesn't wear out.
3. Software is custom built or component built rather than been assembled.

1. Software is developed or engineered; it is not manufactured in the classical sense.

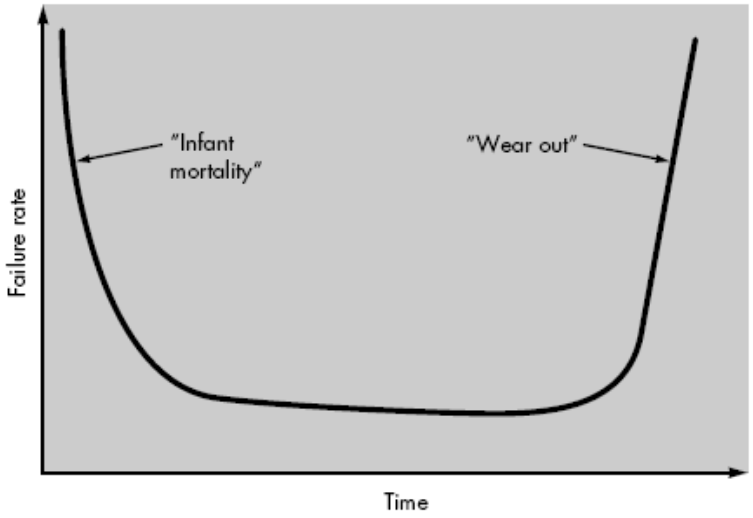
Hardware is designed and manufactured but Software is designed and developed or engineered in a systematic manner.

Both attain quality through good design. Hardware Quality differs from one hardware to hardware but software quality does not differ.

Both hardware and software needs many people for doing different works.

2. Software doesn't wear out.

Only hardware begins to wear out. When the hardware is manufactured the failure rate is more at the beginning after some times the failure rate decreases. After some more time failure rate start to increase because hardware components suffer from dust, vibration, temperature extreme and environment maladies. The below figure shows the failure curve for hardware.

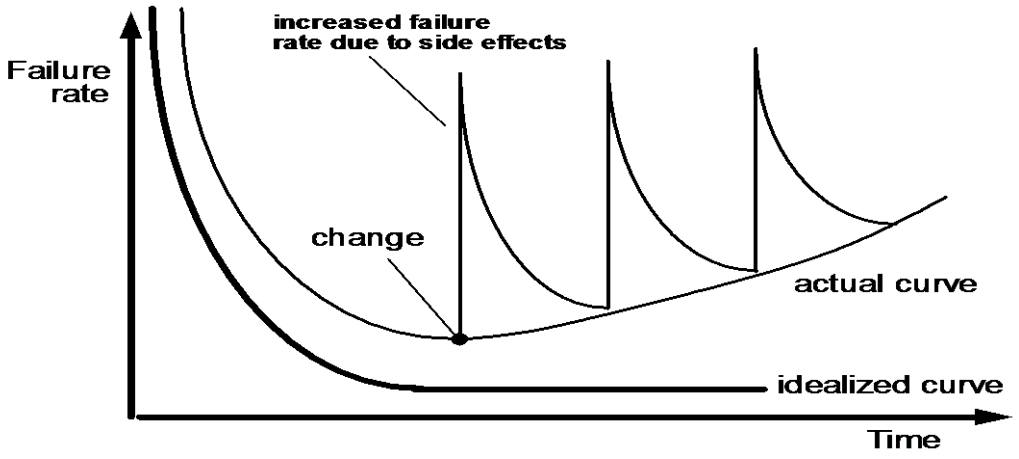


Software at the beginning will have more failure rate. But after sometime the failure rate decreases and it will sustain (idealized curve).

During software life cycle it will undergo changes. As the changes are made it will introduce some errors so the failure rate curve go to spike and before curve return to original state another change is requested causing the curve to spike again.

Software will suffer only from undiscovered errors. These errors should be corrected, without introducing new errors. As the day progresses the software will deteriorate and won't wear out.

When a hardware component wears out, it is replaced by spare parts. There are no software spare parts. Every software failure shows an error in the design.



3. **Software is custom built or component built rather than been assembled.**

Hardware is assembled from the existing components i.e. hardware is designed using printed circuit board. Only some of the hardware parts or mechanical parts are designed newly other hardware components are reused.

Software is custom built according to each customer need. Only some software is component based and it can be reused for different purpose.

In hardware world, component reuse is a natural part of the engineering process. In the software world it has only begun to be achieved on a broad scale.

Software Myths

Software Myths- false beliefs or idea about the software.

There are three software myths.

- Management myths
- Customer myths
- Practitioner myths

Management myths

Managers or CEO of a company will be under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Myth1: A book full of standards and procedures provide the developers to develop the software.

Reality: Though they are available, they are not properly applied and quality cannot be maintained.

Myth2: Add more people if project gets delayed.

Reality: "Adding more people to a late project will make it later" because people who are working must spend time to educate the newcomers.

Myth3: If we decide to outsource the software project to a third party then we can relax.

Reality: If you are not able to manage and control the software internally, then how we will manage the outsourced project.

Customer myths.

A customer who requests the software is called as customer. He may be a person at the next desk, technical groups, the marketing/ sales department, or an outside company that has requested software under contract.

Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myths1: A general objectives from the customer is enough to begin a software project. Later we can fill the details.

Reality: Ambiguous statement of objective causes disaster. Clear and unambiguous requirements are required.

Myth2: Project requirements change continually and change is easy to accommodate.

Reality: True, but should be requested early before design or code has been started because cost impact will be more in later stages.

Practitioner's Myths.

Practitioners are the person who develops the project.

Myth1: Once a program is written, the software engineer's work is finished.

Reality: The actual effort is spend after it is delivered to the customer for the first time.

Myth2: Until I get the program "running" I have no way of assessing its quality.

Reality: Formal technical reviews can be conducted form the initial stage of the project to access the quality.

Myth3: The only deliverable work product for a successful project is the working program.

Reality: Documentation play a very important role since it is the foundation and guideline.

Myth4: Software Engineering will make us create unnecessary documentation.

Reality: Software Engineering is not about creating documents, it is about creating quality software.

Software Process Models

A software process model is an abstract representation of a process. The process model is chosen based on nature of software project.

Prescriptive Models

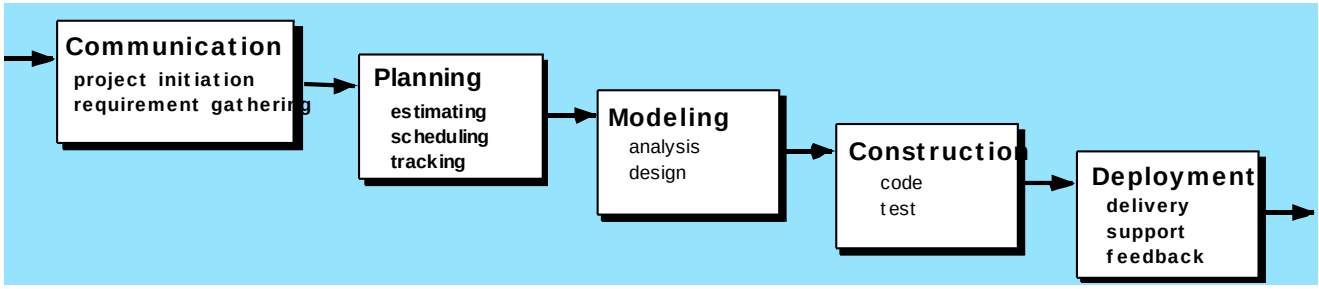
Prescriptive process models is an orderly or systematic approach to software engineering.

Agile Models

Agile process model is a non-systematic approach to software engineering.

The Waterfall Model

***Linear Sequential Model ("waterfall model")*:** A model which is known very well as linear sequential model or system life cycle model or classic life cycle model, emphasizes



a systematic and sequential approach.

Each phase is explicitly completed with specific baselines or milestone and all preceding phases are completed before it moves to the next phase.

Assumptions for waterfall model

- All requirements must be defined clearly at the beginning of the project.
- Output of one software engineering phase acts as an input for next phase.

Phases of waterfall model

The waterfall model follows 5 different phases or steps

- **Communication**
- **Planning**
- **Modeling**
- **Construction**

- **Deployment**

- **Communication** – Before the project is started all the requirements should be gathered or collected from the user through proper communication. Communication can be done in many ways such as
    - Feedback
    - Questionnaires
    - Open dialogue
    - E-mail
    - Chat etc..
- **Planning-** After all the requirements are collected the project should be properly planned. The planning is done on the following basis
    - Estimation(Cost)
    - Schedule (Time, No of persons)
    - Tracking (checking whether everything works fine)
- **Modeling-** After the communication and planning step the project should be properly modeled. The project can be designed using different techniques
    - Tree structure diagram
    - Data flow diagram
    - UML diagrams.
- **Construction** – This is the step where the programmer starts to write the program. This involves Coding and testing. Testing is done by giving some sample inputs and checks the output.
- **Deployment** – The software delivered to the customer's site ,supported documentation is given and feedback is obtained.

Advantage:

It can be appropriate for small, well-understood projects.

Disadvantages:

- Real projects rarely follow a linear sequential flow
- It is often difficult for the customer to state requirements unambiguously

- Customer must have patience to see the working version of the whole product.
- A major blunder, undetected until the end will become disastrous.
- The development team have to wait for the others to finish i.e. team will be blocked.

The Incremental Process Models

Incremental process models are used when the user wants to develop a project with initial requirements and later if he wants to expand the project in many iterations and also if staffing is unavailable.

Types

1. The incremental model
2. The RAD model

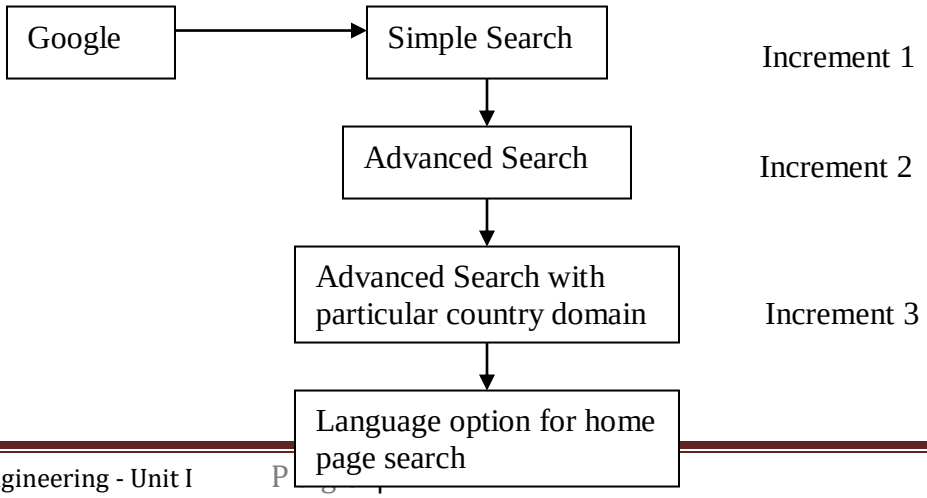
1) The Incremental Model

The incremental model combines elements of the waterfall model applied in iterative fashion. This model delivers a series of releases called as increments. Each increment provides progressively more functionality.

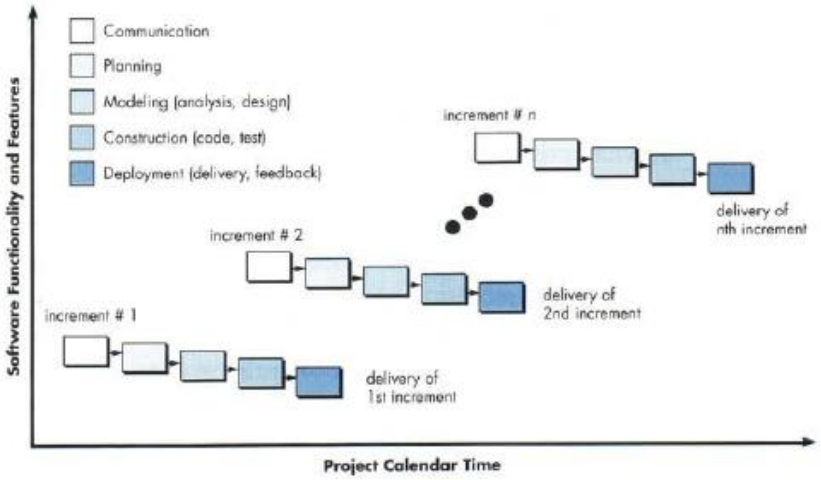
When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered.

The core product is used by the customer. As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated until the complete product is produced.

Example



Increment 4



Used:

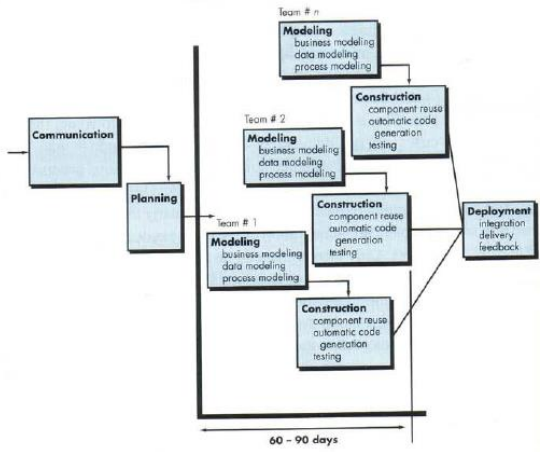
- Staffing is unavailable initially.
- Increments can be planned to manage technical risks (unavailability of the hardware).

2) The RAD Model

Rapid application development (RAD) is useful when the project needs to be completed very quickly. It emphasizes an extremely short development cycle. The RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days).

Multiple software teams work in parallel on different system functions.

In this model communication and planning is done and then the project is modularized and given to different team. The modeling and construction phase has been iterated and finally it has been combined for deployment.



Drawbacks

- For large projects, RAD requires sufficient human resources to create the right number of RAD teams otherwise it will fail.
- If the developer and customer are not in a rapid fire RAD projects will fail.
- If a system cannot be properly modularized then RAD will be problematic.
- RAD is not appropriate when technical risks are high (when a new application makes heavy use of new technology).

Evolutionary Models

The Evolutionary models are iterative. It is followed when the user and developer has no idea about

- Input
- Process
- Output
- Algorithm.

Certain Situation for choosing Evolutionary Models:

- Tight market deadlines
- Business pressure
- Not possible to complete the complete version of project
- Extend the project.

There are three evolutionary process models

1. Prototype Model
2. The Spiral Model

3. The Concurrent Development model

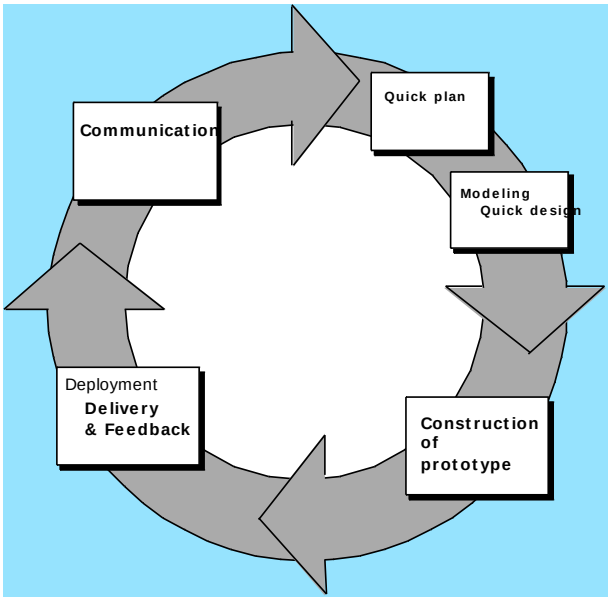
Prototype Model

Prototype means sample model. It is used for demonstration purpose. When the user and the developer does not have a clear knowledge about the input, process, output, algorithm and operating system prototype can be developed.

First the developer /practitioner /programmer develop the project with the given requirement. This project is now called as prototype and it is given to the user.

The user then tells the modification and additional things that is needed to the developer.

Now the developer starts to develop the exact project and complete it and give it to the customer.



- **Communication** – The requirements should be gathered or collected from the user through quick communication.
- **Planning**- After the requirements are collected the cost and schedule is planned very quickly.
    - Estimation(Cost)
    - Schedule (Time, No of persons)
    - Tracking (checking whether everything works fine)
- **Modeling**- A quick model is developed and the design is completed.

- **Construction of prototype** – A sample model is developed with the given requirements.
- **Deployment** – The prototype is deployed and then evaluated by the customer/user.

Iteration occurs until the prototype satisfies the need of the customer.

Advantages:

- Focuses on customer need
- Quick design leads to a prototype .Prototype is evaluated by the customer.

Disadvantages:

- Software quality is not considered in this model.
- The developer may use inappropriate OS or algorithm in the initial stage to develop the prototype and later this will be used to develop the project.

The Spiral Model

The spiral model, originally proposed by Boehm, is an evolutionary software process model. It couples iterative nature of **prototyping** with the systematic aspect of the **waterfall model.**

" 1 spiral might be a paper model
next a prototype
then beta….etc."

Using this model the software is developed in a series of evolutionary releases.

During early stage the release might be a paper model or prototype. In later iteration the complete versions of the project is produced.

This can be adopted to apply throughout the life cycle of an application from concept development to maintenance.

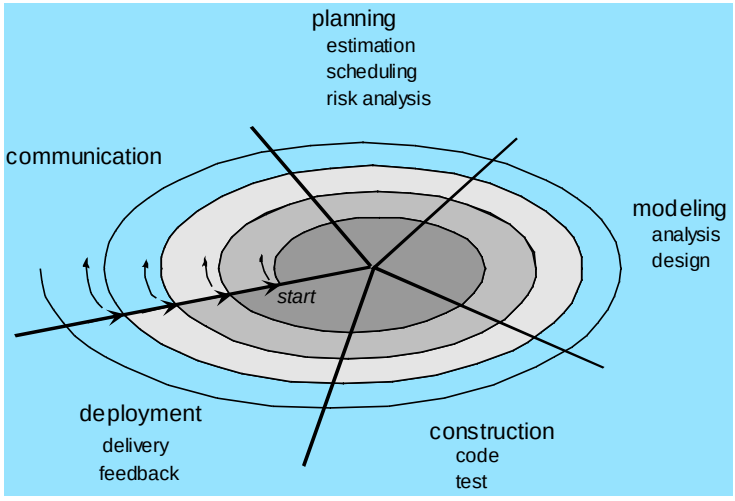
In this model it is divided into set of frame work activities

Communication – Before the project is started all the requirements should be gathered or collected from the user through proper communication. Communication can be done in many ways such as

- Feedback
- Questionnaires
- Open dialogue

- E-mail
- Chat etc..

- **Planning**- After all the requirements are collected the project should be properly planned. The planning is done on the following basis
    - Estimation(Cost)
    - Schedule (Time, No of persons)
    - Tracking (checking whether everything works fine)
- **Modeling**- After the communication and planning step the project should be properly modeled. The project can be designed using different techniques
    - Tree structure diagram
    - Data flow diagram
    - UML diagrams.
- **Construction** – This is the step where the programmer starts to write the program. This involves Coding and testing. Testing is done by giving some sample inputs and checks the output.
- **Deployment** – The software delivered to the customer's site ,supported documentation is given and feedback is obtained.



It starts from the inner circuit and rotates in clockwise. The first circuit around the spiral might represent a **"Concept Development Project"** and continues until the concept is developed.

The next circuit proceeds with **"New Product Development project**" and continues through no of iteration around the spiral.

The next spiral might be used to represent **"Product enhancement project".**

The next spiral might be used to represent **"Product Maintenance Project".**

Advantages

- It is a Realistic approach for large-scale software.
- Developer/customer better understand risk at each stage.
- Reusability of the software

Disadvantages

- It is only suitable for large sized projects
- Fixed budget projects will fail, because at the end of each circuit the project cost is increased
- If the risk are not identified it will be problematic.

Concurrent Development Model:

The concurrent development model is also called as concurrent engineering. The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states.

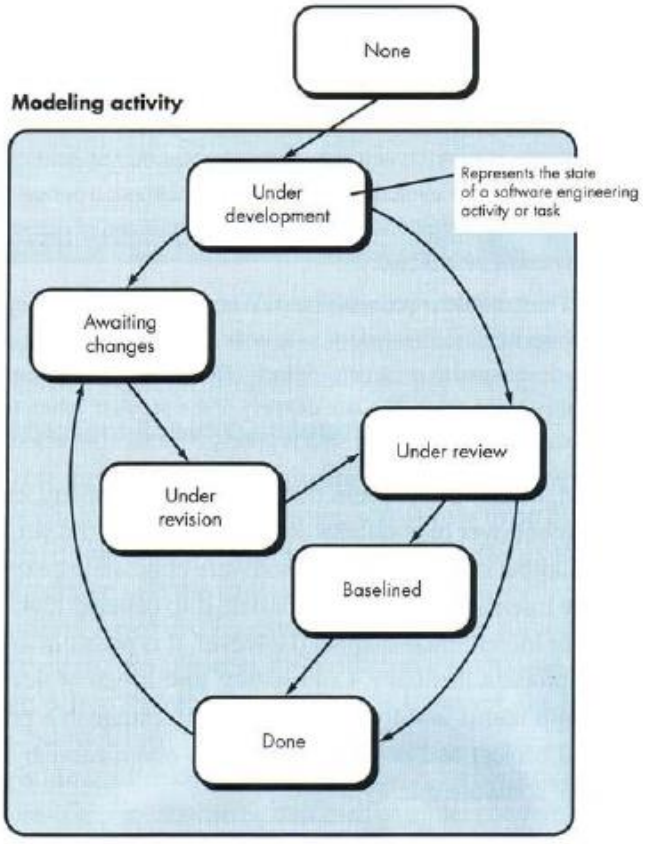
This model is appropriate for system engineering projects where different engineering teams are involved.

This model conducts all steps in waterfall model parallel.

Early in a project customer communication activity has completed its first iteration and exists in the done state. The modeling activity which existed in the none state while initial customer communication was completed now makes a transition into the under development state. If, however, the customer again wants to make some changes in requirements the modeling activity moves from the under development state into the awaiting changes state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities.

In reality, the concurrent process model is applicable to all types of software development.



Advantages:

Provides an accurate picture of project state.

UNIT II

REQUIREMENT ENGINEERING

Requirement engineering the helps the software engineer to better understand the problem.

- It deals with the business impact of the software
- What the customer wants.
- How the end-user react with the software.

The requirement engineering is done with software engineer, Project stakeholders (managers, customer, and end-user) all participate in the requirement engineering.

Requirement Engineering task:

Requirement engineering is a bridge for design and construction. It determines whether the system to build is really going to satisfy the needs of the customer.

The requirement engineering process is accomplished through the execution seven distinct functions.

- Inception
- Elicitation
- Elaboration
- Negotiation
- Specification
- Validation
- Requirement management

Inception

Inception process initiate the requirement engineering task by asking the number of context free question to the stakeholder. This is to establish the basic understanding of the problem and people who want the solution. The primary communication between the customer and the developer is done.

Elicitation

Elicitation phase ask the customer, users and others about what will be the objective of the system.

Why elicitation is difficult:

Due to the below reason the elicitation becomes difficult

- **Problem of scope** The boundary of the system is ill defined.
- **Problem of understanding** the user or customer are not completely sure about the problem domain
- **Problem of Volatility** the requirement change overtime

To overcome these difficulties the requirement engineering must be done in an organized manner.

Elaboration

In elaboration phase the information captured during the previous elicitation phase is expanded and refined.

All the requirements given by the customer may not be useful it is the duty of the analyst to design validity and reliability of the requirement.

Step in Elaboration

- Refining and expanding the requirement.
- Developing an analysis model that define the information, functional and behavior domain of the problem.

Negotiation

Different customers or users may propose conflicting requirement which may create deadlock. Negotiation is necessary to make things to move smooth." **There should be no winner and no loser in an effective negotiation".**

Steps in Negotiation

- The development team and stake-holder must reconcile the conflicts through negotiation by sitting together and see the most important requirement to discuss the conflict.
- Based on rough estimate effective conflict can be resolved.
- The conflict resolving process must make each stake-holder and developer to satisfy.

Specification

It is the final work product produced by the requirement engineering. It serves as the foundation for subsequent software engineering activities.

Specification means different things to different people. A specification document may be a written document, graph model or a mathematical model. There is no predefined template because it may restrict the flow of the system to be built. While writing the specification it is necessary to remain flexible.

Validation

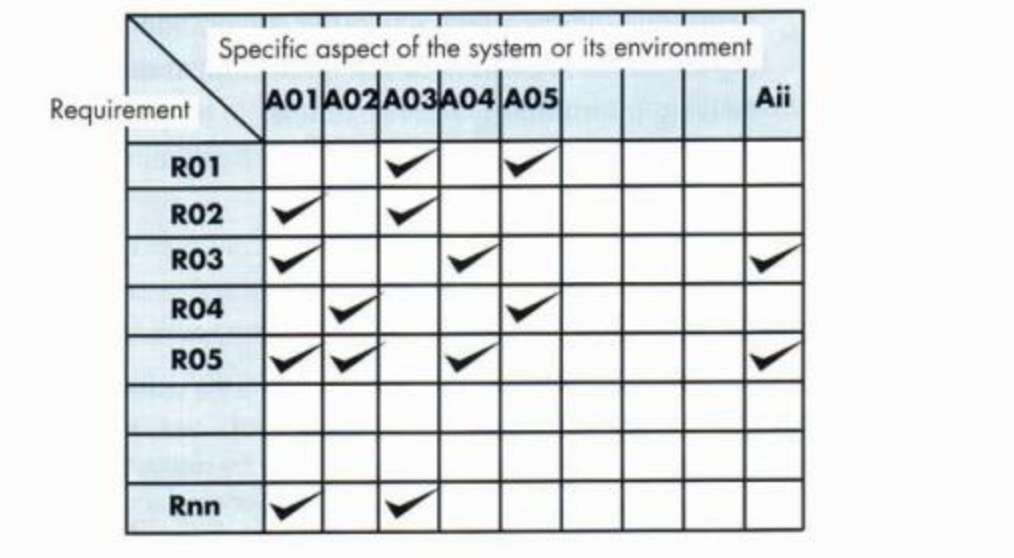
Validation is to asses the Quality and reliability of the specification created. It checks whether all the software requirement

state are unambiguous, inconsistent, omission, errors and detects and correct it. The primary requirement for validation mechanism is the formal technical review (FTR)

Note: validation is performed to asses the quality of the specification.

Requirement Management

Requirement management is to identify, control and track the changing requirement. The changing requirement is done with the help of traceability table. The traceability table is created after identifying the requirement.



Initiating the requirement Engineering Process

- Identifying the stake-holder
- Recognizing the multiple view point
- Working towards collaboration
- Asking the first question

Identifying the stake-holder

The stake-holder is some one who benefits in a direct or indirect association with the software being built.

For example: product manager, business manager, marketing people, internal and external customer, end-user, consultancy, software engineers and support and maintenance engineers.

Recognizing the multiple view point

Each stake-holder may have different opinions about the system to be built. Each stake-holder accepts the different functionality

For example: Business manager - will need a quality system

Finance manager – may need a cost effective system

End user - may want easy to learn and use

Support Engineers - may want a maintainability of the a software

Working towards collaboration

The customers with the other stake-holder should collaborate among themselves built as successful system. The job of the requirement engineer is to identify in which all the stake-holder agree and in which some of the stake-holder have conflicts with each other.

The final decision of the conflict will be decided by the business manager or a senior technologist.

Asking the first questions

The first set of context free questions focuses on the customers and other stake-holder, overall goal and benefits.

For example: Who is behind the request for this work?

Who will use the solution?

The final set of questions focuses on the effectiveness of the communication activities.

For example: Are you the right person to answer else provide the additional information.

All my questions relevant to the problem you have.

This question will help to "break the ice "and initiate the communication for the successful elicitation.

Eliciting requirements

Elicitation phase ask the customers, the end - users and other stake-holder about the **objective of the system.**

This phase has the following activities

- Collaborative requirements gathering
- Quality function deployment (QFD)
- Users scenarios
- Elicitations work product

Collaborative requirement gathering

The stake holder and the developers work together to identify the problem. Some of the basic guidelines for the conducting the collaborative requirement gather.

- Meetings are conducted and attended by both software engineers and customer along with other interested stake – holders
- Rules for preparation and participation are established.
- An agenda is prepared to cover all the important points.
- "Facilitators"can be a customer, a developer or an outsider controls the meeting.
- A definition mechanism can be in the form of wall stickers, chart room or electronic bulletin board.
- The goal of the meeting is to identify the problem and propose the solution.

Quality function deployment (QFD)

QFD is a technique that translates the customer needs to requirement for the software. QFD defines in a way that maximizes the customer satisfaction.

QFD identifies three types of requirement

- Normal requirement
- Expected requirement
- Exciting requirement

Normal requirement:

The objective defines the requirement engineering is satisfied .for example: normal requirement may be requested types of graphical displays and defines the level of performance.

Expected requirement:

They are fundamental requirement that are not explicitly declared by the customer during the requirement gathering meeting.

Exciting Requirements:

These requirements reflect features that go beyond the customer's expectations and prove to be very satisfactory when it is presented.

User Scenario:

As requirements are gathered the developers and users can create a set of scenario ("Use case" (Gathering the requirements of the system by user point of view) to provide a description of how the system will be used.

Elicitation work products:

The work product produced as a consequence of requirement elicitation will vary depending on the size of the system or product to be built.

For most systems, the work product includes:

- A statement of need and feasibility.
- Scope of the system or product.
- A list of customers, users, and other stakeholders who participated in requirement elicitation.
- Description of the technical environment.
- A set of scenarios.

Building the Analysis Model

Analysis modeling is the first technical representation of a system. It uses a *combination of text and diagrammatic* forms to depict requirements for *data, function, and behavior* in a way that is relatively easy to understand, and more important, straightforward to review for *correctness, completeness and consistency (i.e Quality)*.

Requirement Analysis:

Requirement analysis allows the software engineers (sometimes called an analyst or modeler) to elaborate the basic requirements established during requirement engineering task and provides the software designer with a representation of data, function and behavior that can be translated into architectural , interface and component – level design.

It also provides the developer and customer to access the software quality once software built.

The primary focus of analysis modeling is on 'what' not 'how'.

For example:

- What data objects the systems manipulate?
- What functions must the system perform?
- What behavior does the system exhibit?

The Requirement Analysis deals with,

- Overall objectives and Philosophy
- Analysis Rule of thumbs
- Domain Analysis

Overall Objective and Philosophy:

- Describe what the customer requires.
- Establish a basis for the creation of a software design.

- Devise a set of requirements that can be validated once the software is built.

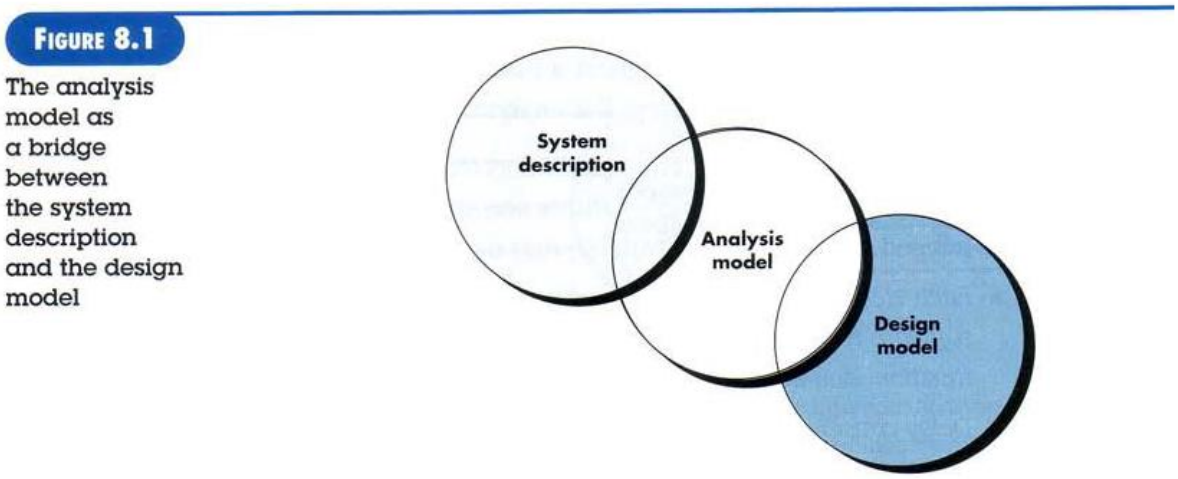


FIGURE 8.1 The analysis model as a bridge between the system description and the design model

Analysis Rules of Thumb

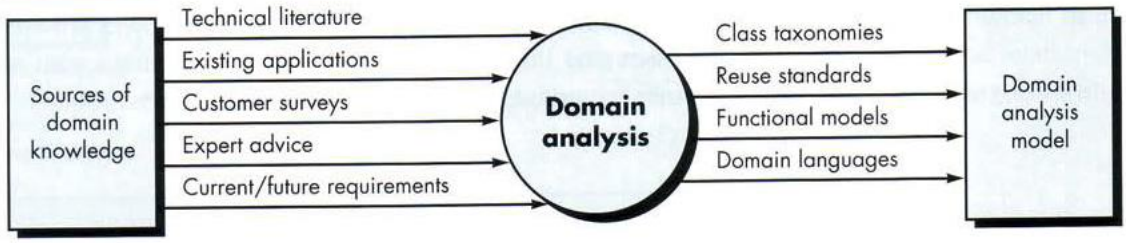
- The model should focus on requirements that are visible within the problem or business domain and be written as a relatively high level of abstraction.
- Each element of the analysis model should understand the requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain the analysis model provides value to all stakeholders.
- Keep the model as simple as possible.

Domain Analysis

Software domain analysis is to identify, analysis, and specify the common requirements from a specific application domain and reuse on multiple project [Object- Oriented Domain Analysis].

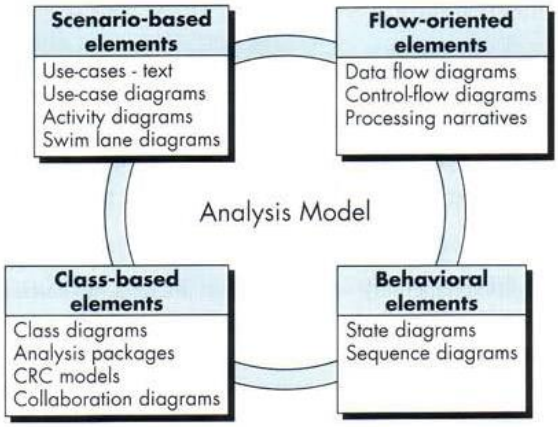
The role of a domain analyst is similar to the role of a toolsmith [The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily same object] In the same way the Domain analyst is to discover and define reusable patterns which can be used by many people. The source of domain knowledge is the input and output, that is obtained. The below figure explains the input and output for domain analysis.

FIGURE 8.2 Input and output for domain analysis



Elements of Analysis Model:

FIGURE 8.3 Elements of the analysis model



Data Modeling Concepts

Analysis modeling often begins with *Data Modeling*.

The software engineering or the analyst defines all data objects that are processed within the system, the relationship between the data objects and other information that is relevant to the relationship.

Data Modeling deals with,

1. Data Objects
2. Data Attributes
3. Relationships
4. Cardinality and Modality

1. Data Objects:

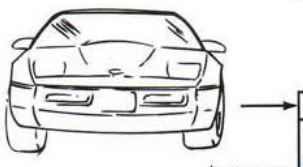
A data object is a representation of any **composite information** (i.e which has different properties or attributes) that must be processed by the software.

Eg: Dimensions (height, width, depth)

- In context with a particular **application a Data object** can be an **external entity** example anything that **produces or consumes information.**
- A thing (Report or a display), an occurrence (telephone call), an event (an alarm), a role (salesperson) , a place(a warehouse).

The data object can be represented as a table as shown in Figure 8.4

FIGURE 8.4
Tabular representation of data objects



Naming attributes
Ties one data object to another, in this case, owner
Identifier
Descriptive attributes
Referential attributes

Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

A **car or a person** can be viewed as a data object which has different data attributes. They are,

- Make
- Model Number
- ID number
- Body type
- Color
- Owner

Instances:

The body of the table represents specific instances of the data object (Eg: Lexus, Chevy etc).

2. Data Attributes :

Data Attributes defines the properties of data object.The heading in the table reflects the data attributes.

It consist of one of the three characteristic,

i. Name an instance of the data object
ii. Describe the instance
iii. Make reference to another instance in another table.

Identifier attribute: It becomes a key when we search for an instance (Eg ID number)

3. Relationship :

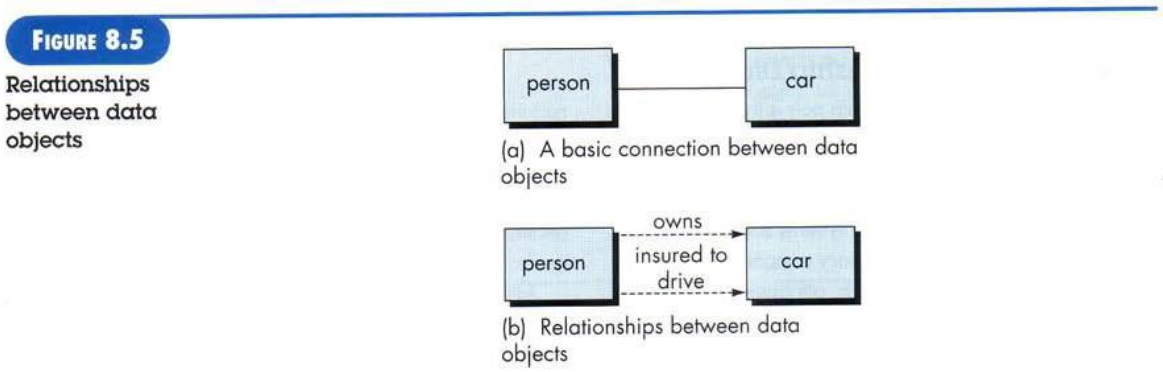
- Relationship tells about relation between two data objects.
- Consider, two data objects person and car
- These objects can be represented using simple notation.

A **connection** is established between person and car because the two data objects are related Figure 8.5 (a).

The relationship is

- a person **owns** a car

- a person is **insured** to drive a car Figure 8.5 (b)



Cardinality & Modality :

Cardinality specifies the type of relationship between two or more data objects. (i.e) how many occurrences of a data objects relates to occurrence of another data objects.

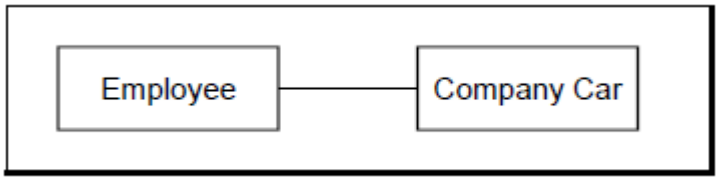
Different cardinality exists between data objects.

Example :

- One to one
- One to Many
- Many to Many

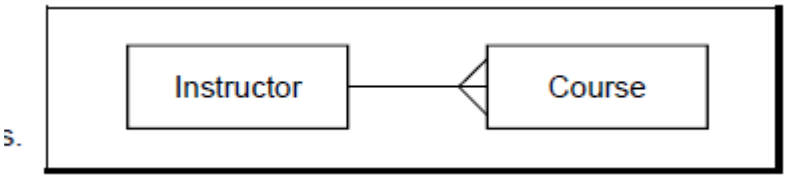
One to one

One object can relate to only one object example the employee owns a company car.



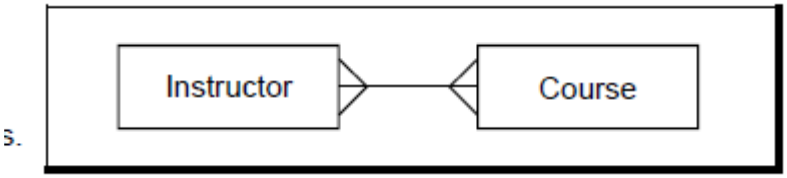
One to Many

One object can relate to only many object example an Instructor handles many course.



Many to Many

Some number of occurrence of an object can relate to some other number of occurrence of another object example Different instructor handles different course.



Modality :

Modality is specifying whether the relationship between two data objects should be compulsorily given or optional.

If Modality=0, then it means relationship is optional.

If Modality=1, then the relationship between two data objects should be given.

Flow oriented Modeling :

Flow Oriented Model consists of input view, process view and output view. This model tells about the flow of data from one entity to another entity.

Every computer based system is an information transformer.

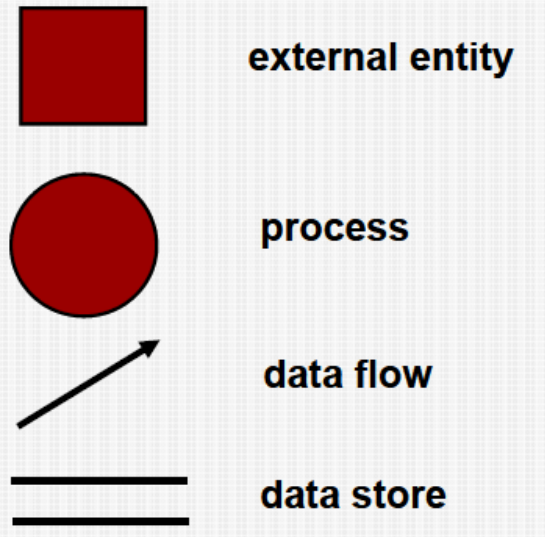
The Flow Oriented Model consist of

- Creating a Data Flow Model
- Creating a Control Flow Model
- The Control specification
- The Process Specification

Creating a Data Flow Model

- Data Flow Diagram
  - Depicts how input is transformed into output as data objects move through a system

Flow modeling Notations:



External Entity:

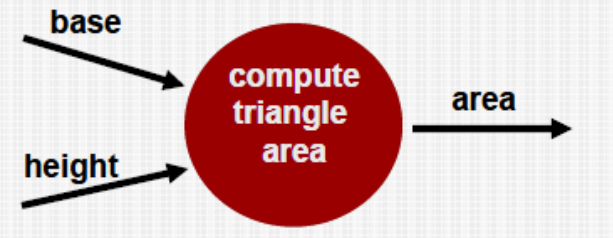
A producer or consumer of the data.Example :a person, a device or sensor etc.

Process:

Changes input into output . Example: compute taxes, determine area, display graph etc.

Data Flow:

Data flows through a system, beginning as input and transformed into output



Data Store:

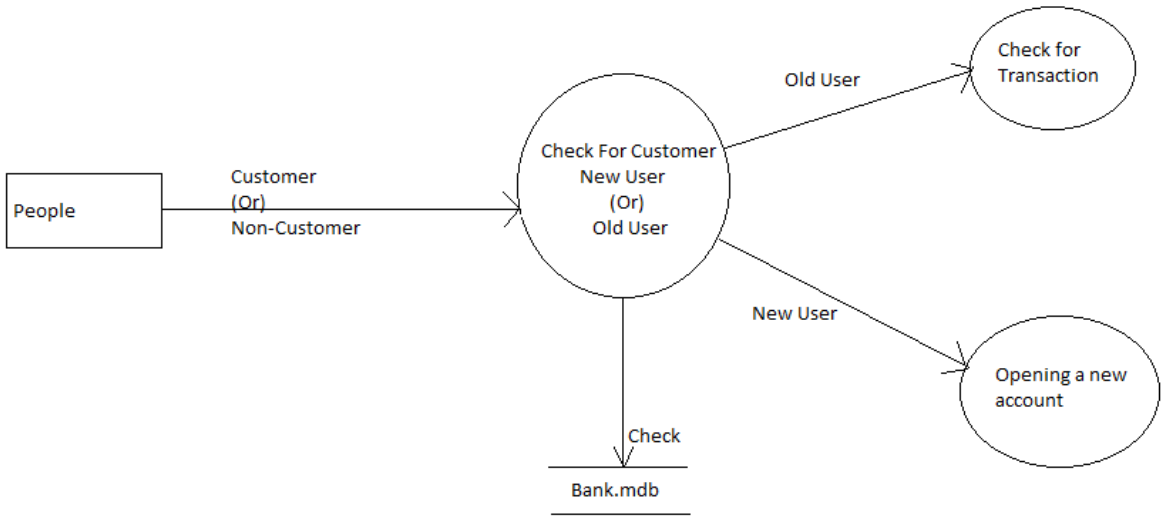
Data is often stored for later use.

Rules for drawing the Data Flow Diagram (DFD):

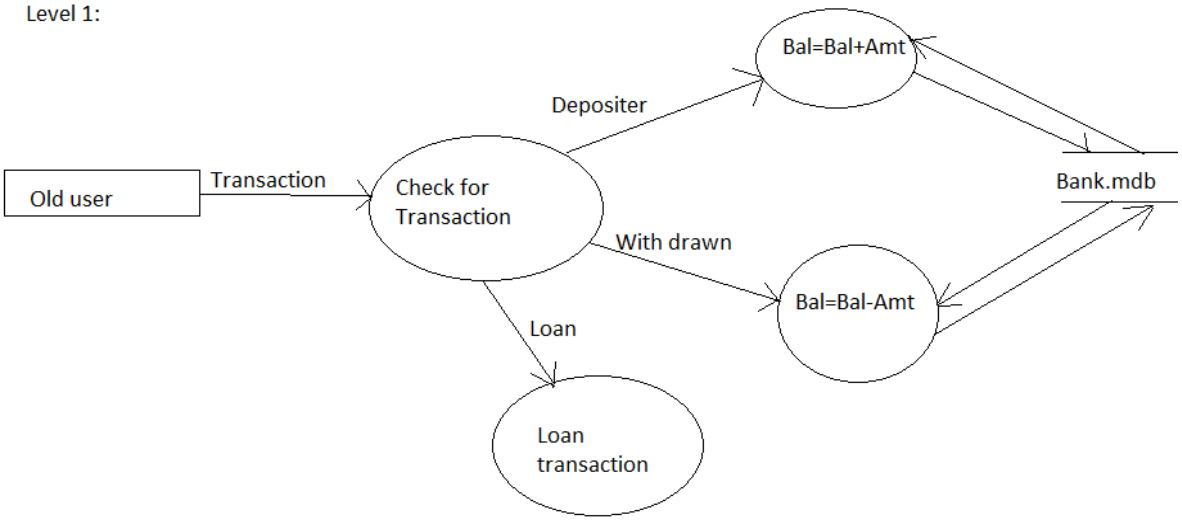
- All arrows and bubbles should be named (caption)
- The flow should be continuous.
- There should not be any unfinished arrow.
- Only the above symbols must be used.
- There are different levels of DFD (i.e) level 0, level 1, up to level n.

- Finally all these levels can be combined to complete DFD.

Level 0:



Level 1:



Level 2:

Example of Level 0 DFD :

Banking process system (Within Online Banking)

i. Input ---- Is the Customer

Old-Customer or New-Customer

ii. Process ---- Opening a new account

Deposit

Withdraw

Loan Facility

iii. Output ---- Cash, Receipt, passbook.

Creating a Control Flow Model:

A large class of software are driven by events rather than data. So it is important to use control flow

Control Data Flow Model :

The control data flow model make use of different control specification **(CSPEC) :**

a) Control specification indicates the process that should be taken place at different events.

b) Basically, events have two different states.

Set [1] [On, Start]

Reset [0] [Off, Stop]

c) Some of the other states are idle state, activate state, etc…..

d) To represent the different state it is necessary to draw the state transition diagram.

e) The following is state transition diagram for how alarm system maintain.

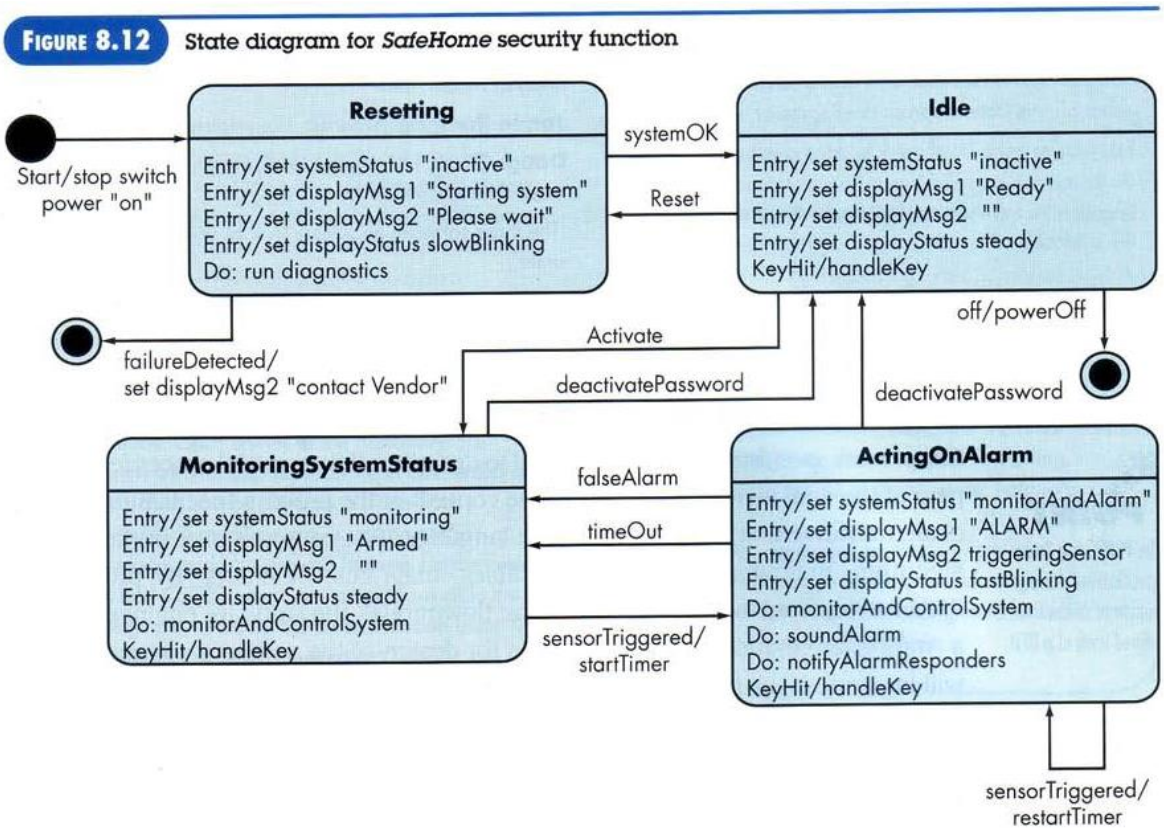


FIGURE 8.12 State diagram for *SafeHome* security function

Process Specification for Flow Model (PSPEC) :

PSPEC is a "mini-spec" for each level of the DFD. Flow model process specification is always available at final level of refinement.

The content of the PSPEC contains can include

- Narrative Text
- Tables
- Charts or diagrams
- Mathematical equations

"Mini-Specs" serve as guide for s/w component which implement the process.

Behavioral Model

Behavioral Model is used to represent the dynamic behavior of the system or product. It indicates how software will respond to external events.

To create the model the following steps should be performed.

i. Evaluate all use-case diagrams to understand the interaction within the system.
ii. Identify events that drive the interaction sequence
iii. Create a sequence for use-case diagram.
iv. Build a state diagram for the system.
v. Review the Behavioral Model to verify accuracy and consistency.

Identifying the events with the Use-case :

Use-case shows the interaction between system and its environment. In a Use-case diagram two things are involved,

i. Actors.
ii. Systems.

i. Actors :

People can be called as Actors.

Example :

No. of persons involved in project or any external events.

ii. Systems :

a) Systems (or) program inside the computer.

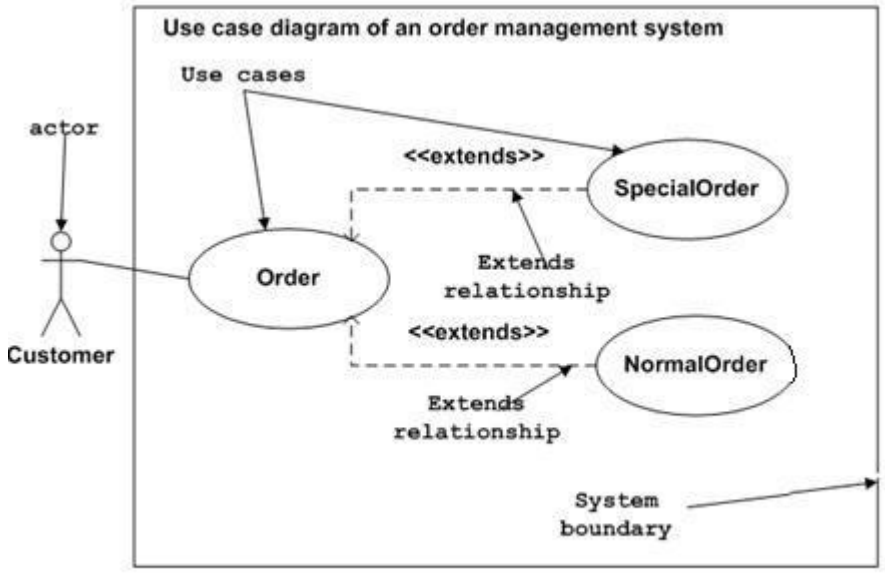


Figure: Sample Use Case diagram

When actors are involved in system different behaviors can be easily studied with the help of state representation.

State Representation:

State diagrams are used to describe the behavior of a system. State diagrams describe all of the possible states of an object as events occur.

There are two types of state :

i. Passive State
ii. Active State

i. Passive State :

Passive state are the one which normally occurs at the beginning and end of the system.

Example : In a game system, the starting position of the actors can be called as a passive states or Start of the bike race or Finishing the bike race

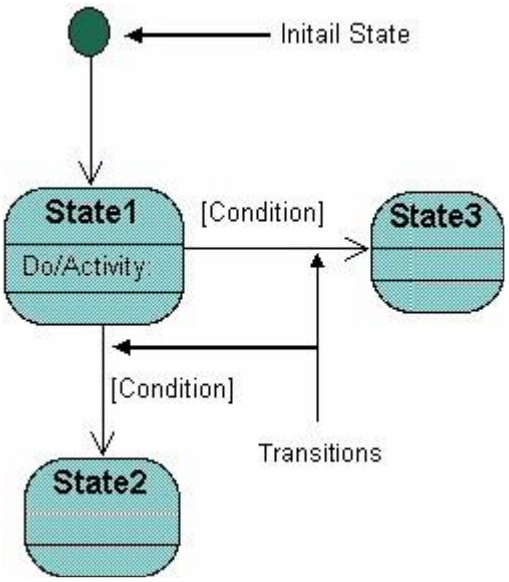
ii. Active State :

When the game proceeds different states occurs. This state keep on changing throughout the Game. It is called as Active State.

Example :

- Jump ---- Active State
- Fall ---- Active State
- Overtake ---- Active State

Various Notations used in State Chart Diagram:



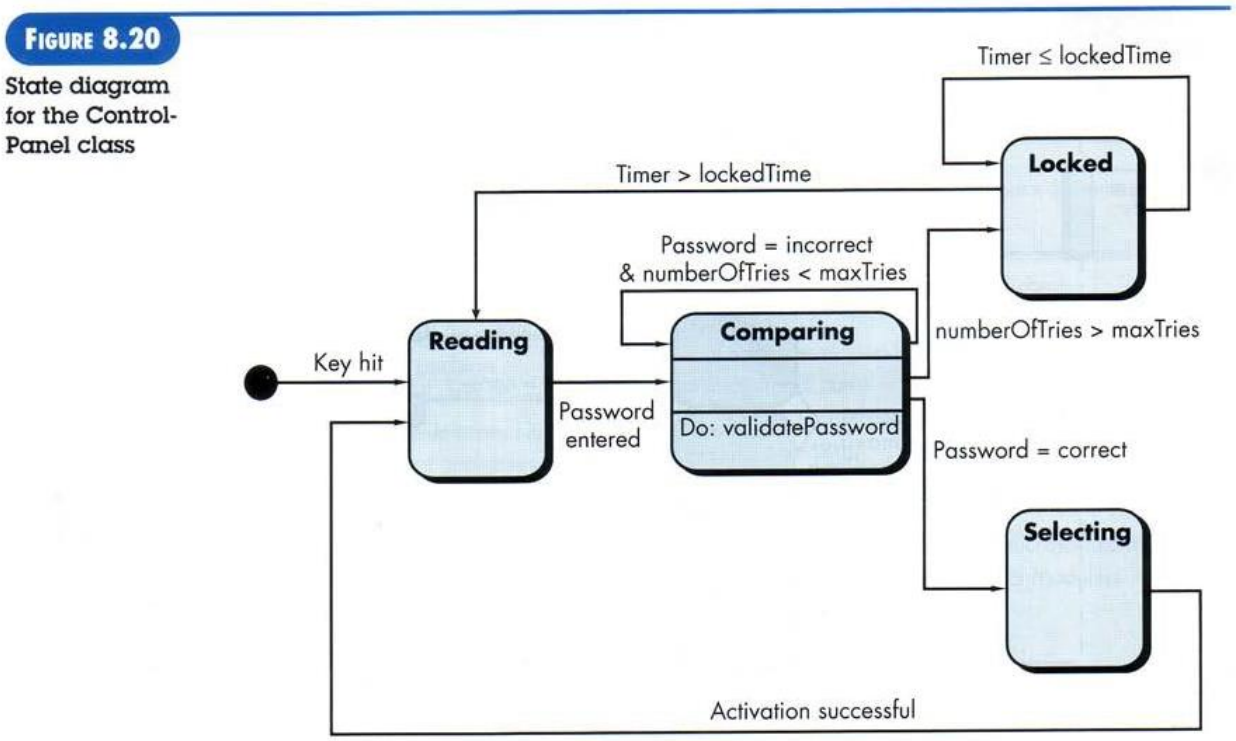


FIGURE 8.20
State diagram for the Control-Panel class

For example, the guard for the transition from the "reading state" to the " comparing state" is shown in figure 8.20.

Class Based Model :

Definition : Class --- implies collection of Member data and Member functions.

Each and every program can be properly organized into different classes. Each class have their own attributes (member Data) and operations (Member function).Classes can be an external entity.

1) **External Entity :**
   Consume information to be used by the computer based system.
   Example : Other systems, devices, people…. Etc.
2) **Things :**
   That are part of the information domain for the problem.
   Example :Reports, letters, signals.
3) **Occurrence (or) Events :**

That occur within the context of system operation.

Example : Property Transfer.

4) **Roles :**

Played by people who interact with the system.

Example : Manager, Engineer, Sales person.

5) **Organizational Units :**

That are relevant to an Application. Example : Team, Group, Division.

6) **Places :**

That establish the context of the problem and the overall function of the system. Example : Manufacturing Floor.

7) **Structures :**

That define a class of objects or related classes of objects.

Example : A student can be called as a class which has own attributes and operations.

A student class will have the following attribute.

i. Register Number
ii. Student Name
iii. Age
iv. Department
v. Mark 1
vi. Mark 2
vii. Mark 3
viii. Total
ix. Average
x. Rank

The values given to the attributes should be meaningful,

Register Number-----A10CADA01

A---- Autonomous

11---- Year

C---- Computer

A---- Application

D---- Day college

A---- Section

01---- Roll number

Total---- m1+m2+m3

Average---- Total/3

Student
Regno Name Age Department Mark1

Age---Negative number should not be specified (>0).

Operations :

A student class can have the following set of operations.

1) Find Total ( )
2) Find Average ( )
3) Find Rank ( )

The class diagram for the student class,

Class	Mark2 Mark3
Attributes	Find Total() Find Average() Find Rank()

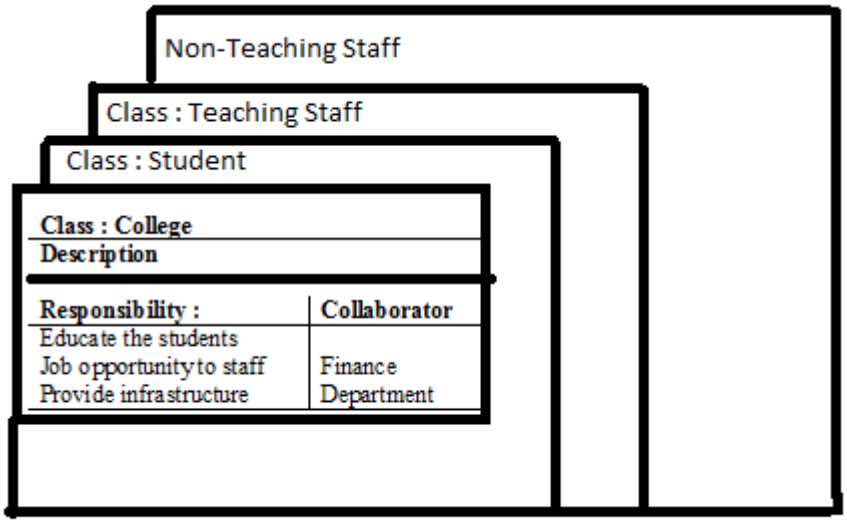
Operations

Class-Responsibility-Collaborator (CRC) Modeling :

CRC is used to identify classes and organize the classes that are relevant to system or product requirement.CRC makes use of actual or virtual index cards. It consists of the ***name, responsibility, and collaborator.***

Name: Class Name

Responsibility: The Responsibility are the attributes and operations that are relevant for the class.



Collaborators: Collaboration implies either a request for information or a

request for some action.

Associations and Dependencies :

Associations :

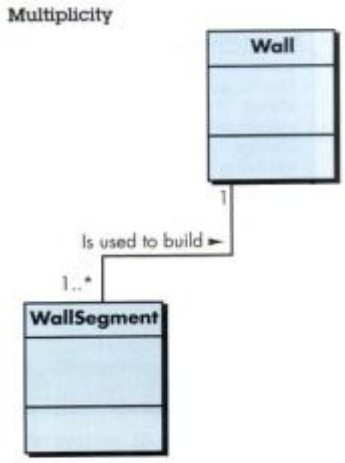
In a client server environment client is always depend on the server.

Example : The client class will not function if the several class is off.

Different types of association dependencies will exist b/w two or more classes.

Some of the associations are,

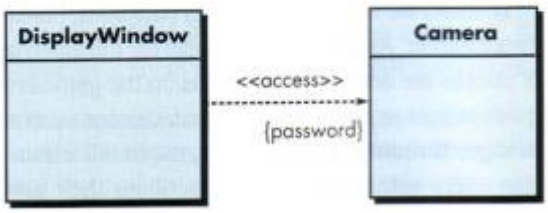
i. One to One (1…1)
ii. One to Many (1…*)
iii. Many to Many (*….*)



Dependencies:

Dependencies are defined by stereo type. Stereo type means "extensibility mechanism". In UML, stereo types are represented in ***double angle brackets.***

Dependencies



Unit IV
TESTING

Testing:

Testing is a set of activities that can be planned in advance and conducted systematically to uncover errors.

General characteristics:

- If FTR is performed effectively it may eliminate many errors before testing
- Testing begins at the **component level** to the **integration level.**
- Testing is conducted by the **developer for small projects** and for **large projects an independent test group.**
- **Testing and debugging are different activities**, but **debugging** must be **accommodated** in any testing strategy.

Verification and Validation:

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 **meets customer requirements.**

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

Test Strategies for Conventional software

A testing strategy falls between two extremes

- Incremental view of testing beginning with the testing of individual program**(Unit Testing)** and moving to **integration testing**

Unit Testing:

Unit testing focuses on the **smallest unit** of software design the **Software component or module.**

Unit Test Considerations:

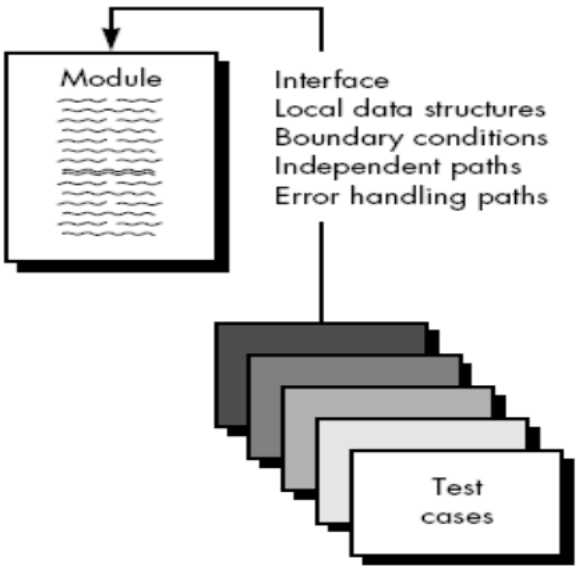
The **module interface** is tested to ensure that **information properly flows in** and **out** of the program unit under test.

The **local data structure** is examined to ensure that **data stored** temporarily **maintains its integrity.**

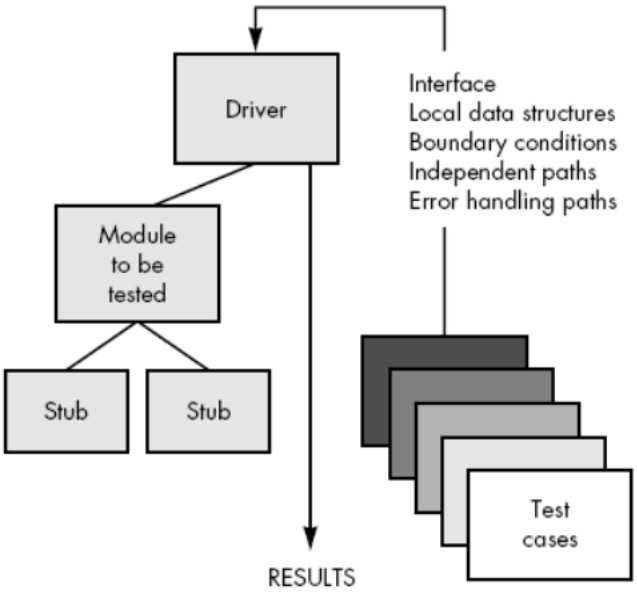
Boundary conditions are tested to ensure that the **module operates properly at boundaries.**

All independent paths (basis paths) through the **control structure** are exercised to ensure that **all statements in a module** have been **executed at least once**.

And finally, all **error handling paths** are tested.



Unit Test Procedure



Unit Test Environment

In Unit testing the Component(module) is not a standalone program so to test the component a **driver** and **stub** concept is introduced.

Driver is nothing more than a **"main program"** that accepts test case data, passes such data to the component (to be tested), and prints relevant results.

Stubs serve to replace modules that are subordinate. A stub or **"dummy subprogram"**.

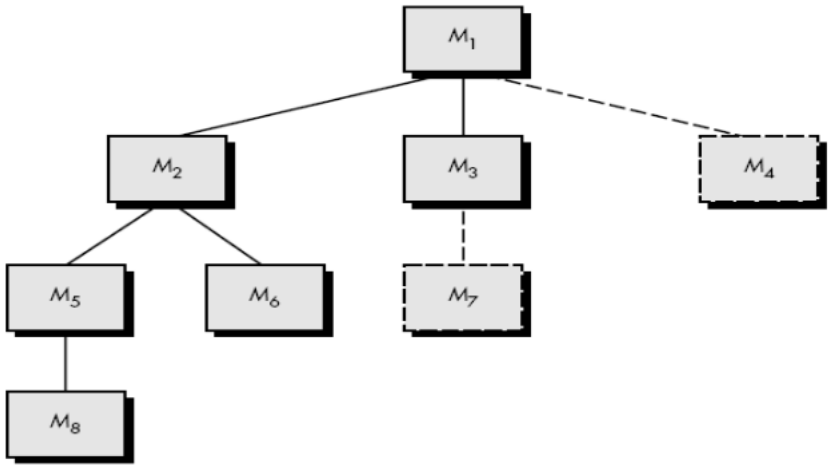
Integration Testing:

Integration testing is a systematic technique conducted to test errors associated with interfacing. There are two approaches in integration testing

1. **Non-incremental testing (Big Bang Approach)**
   The entire program is tested as a whole. So it is difficult to deduct errors.
2. **Incremental Integration:**
   The program **is constructed and tested in a small increments**. So that errors can be isolated easily. There are two types of integration testing
   - **Top-down Integration**
   - **Bottom-up Integration**

Top-down Integration

Top-down integration testing is an incremental approach. **Modules are integrated** by **moving downwards** beginning with the main control module (main program). It follows either a **depth-first or breadth-first manner.**



Depth-first integration would integrate all components on a major control path of the structure. For example, components M1, M2, M5 would be integrated first. Next, M8 or M6 would be integrated and goes on.

Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. Components M2, M3, and M4 would be integrated first. The next level, M5, M6, and so on.

The integration process is performed in a series of steps:

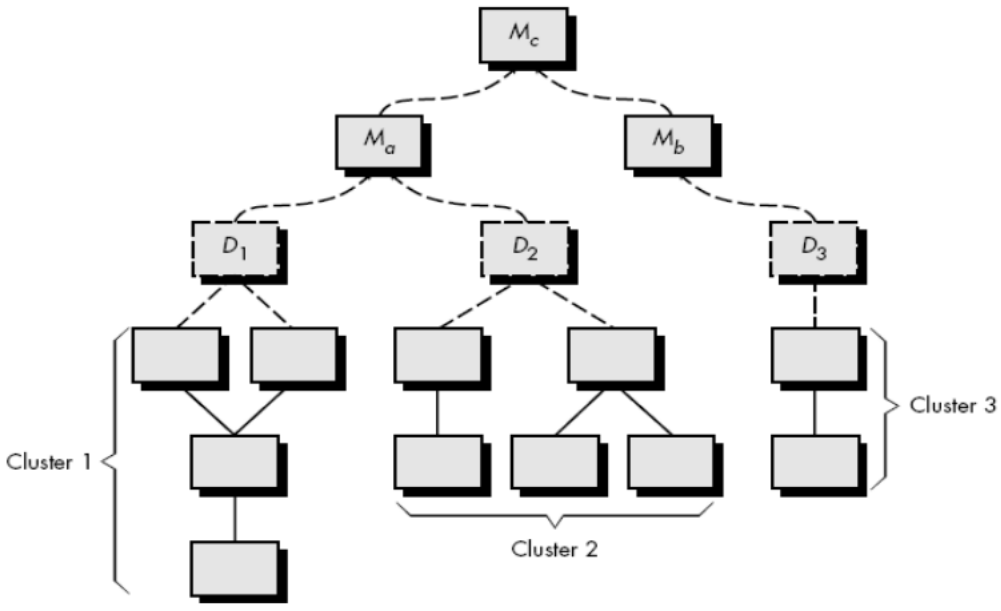
1. Test Main Module and stubs are substituted for all components directly subordinate to the main module.
2. Then integration testing for subsequent module is

tested using either **Depth first integration or Breadth first integration.**

3. Driver is not needed. Only stubs are required in this approach.
4. Regression testing may be conducted to ensure that new errors have not been introduced.

Bottom-up integration testing, as its name implies, **begins construction and testing with *atomic modules*** (i.e., components at the lowest levels in the program structure). A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into **clusters (sometimes called *builds*)** that perform a specific software sub function.
2. A **driver (a control program for testing)** is written to



coordinate test case input

3.The cluster is tested.

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

Integration follows the pattern illustrated in Figure Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

VALIDATION TESTING:

Validation succeeds when software functions satisfies the customer reasonable expected output. The customer Reasonable expectations are defined in the ***Software Requirements Specification***—a document that describes all user-visible attributes of the software. The specification contains a section called ***Validation Criteria***.

Validation Test Criteria :

Software validation is achieved through these possible conditions:

(1) Whether the specification is met or not.

(2) If it is not met we have to find out the deviation from specification and a ***deficiency list*** **is created.**

Configuration Review:

Configuration Review is to ensure that all elements of the software configuration have been properly developed. E.g Harddisk,clock speed,RAM.

Alpha and Beta Testing

The *alpha test* is conducted at the **developer's site by a customer**. The developers record the errors of the user. 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 controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer and the software engineers make modifications and then release the software product.

System Testing:

System Testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based

system. Different testing is involved. They are,

Recovery Testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
This testing will test whether the system is fault tolerant. In this testing mean-time-to-repair (MTTR) is used to find in what time the system will recover.

Security Testing attempts to verify that protection mechanisms built into a system protect it from improper penetration.

- This testing will test authentication and authorization.
- Authentication means it checks for valid username and password.
- Testing the type of rights is called authorization.

Stress Testing:
The system is tested for abnormal input, process and output. For example,

(1) special tests may be designed that generate **ten interrupts per second**
(2) input data rates may be increased to determine how input functions will respond,
(3) test cases that require **maximum memory** or other resources are executed,
(4) test cases that may **cause thrashing in a virtual**

operating system are designed,
(5) test cases that may cause **excessive hunting for disk-resident** data are created.

Performance Testing:[Time and Storage]

This testing will check the run-time performance of the system. It is often coupled with stress testing.

WHITE-BOX AND BLACK BOX TESTING

Testing is the procedure which helps to arrive at a conclusion that the software is functioning properly or not.

To test software two different methods are followed.

- White-Box testing
- Black-Box testing

White Box Testing

White-box testing, sometimes called *glass-box testing. It tests the internals control structure of the software. It can be tested only by the software Engineers,* because the tester needs *to possess knowledge of the internal working of the code*.

White-box test the following factors,

(1) All independent paths are executed at least once,
(2) It checks both the true and false sides.
(3) All loops at their boundaries should be tested. and

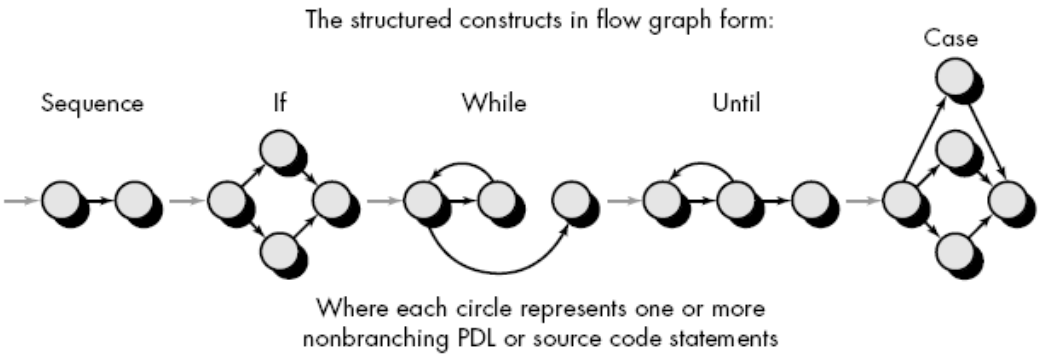
(4) Internal data structures are tested.

BASIS PATH TESTING

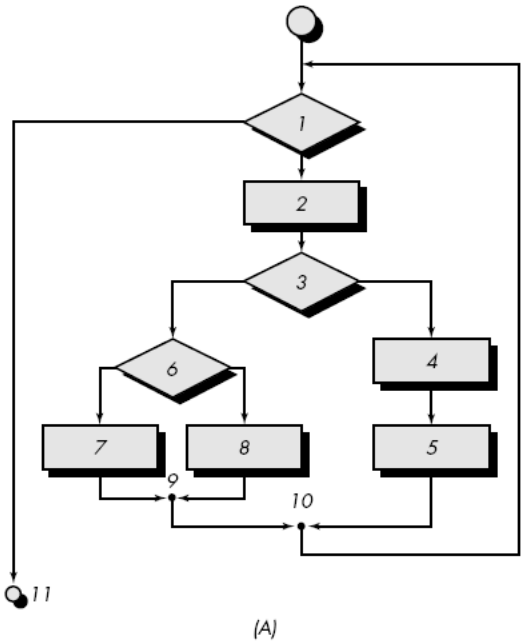
- White Box testing have different methods such method is **Basis path testing.**
- This testing will test all the execution paths that are involved in the program
- To **test the path** many methods are they are
  - ✔ **Flow Graph Notation**
  - ✔ **Independent paths**
  - ✔ **Cyclomatic Complexity**
  - ✔ **Graph matrices method.**

FLOW GRAPH NOTATION

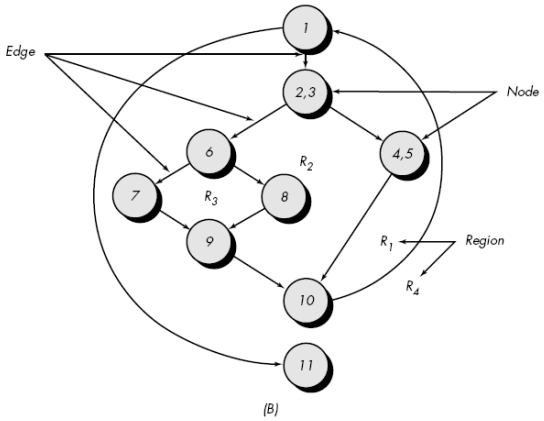
A simple notation for the representation of logical control flow, called a *flow graph* (or *program graph*) is developed. Some of the flow graph notation is listed below.



- The flow graph notation makes use of the nodes and edges.
- The node is represented by a circle and the edges are represented by arrow.
- Each node is a collection of statement or procedures.
- A sequence of process boxes and a decision diamond can map into a single node.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called *regions.* When counting regions, we include the *area outside* the graph as a region.
- Each node that contains a condition is called a *predicate node*.



(A)



(B)

Independent Program paths:

An *independent path* is any path through the program that must have at least one edge that has not been traversed. For example, a set of independent paths for the flow graph illustrated in above Figure is

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11
Note that each **new path introduces a new edge**. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

CYCLOMATIC COMPLEXITY

Cyclomatic complexity is software metric that provides a ***quantitative measure*** of the logical complexity of a program. It is used to ***compute the number of independent paths .***

Complexity is computed in one of three ways:

1. The number of **regions of the flow graph** corresponds to the cyclomatic complexity.

2. Cyclomatic complexity, *V*(*G*), for a flow graph, *G,* is defined as

$V(G) = E - N + 2$ where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, $V(G)$, for a flow graph, G, is also defined as
$V(G) = P + 1$ where P is the number of predicate nodes contained in the flow graph G.

For the above flow graph the cyclomatic complexity are:

1. The flow graph has four regions.
2. $V(G)$ = 11 edges _ 9 nodes + 2 = 4.
3. $V(G)$ = 3 predicate nodes + 1 = 4.

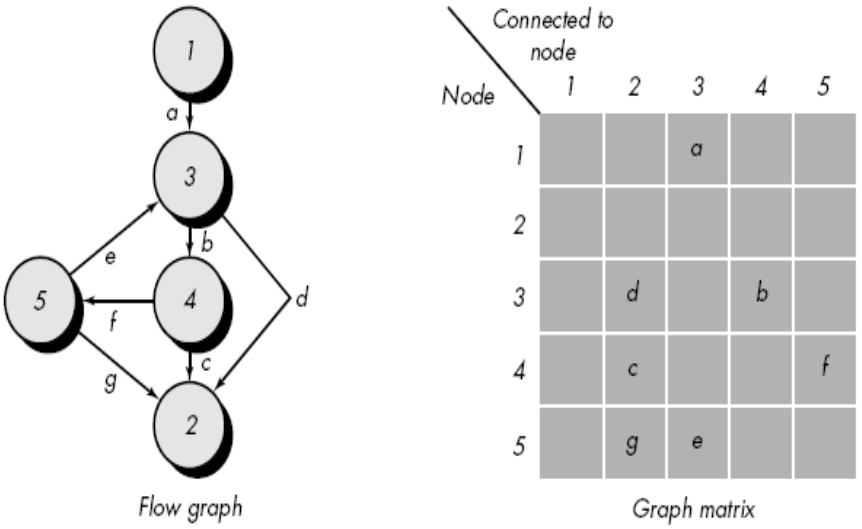
Therefore, the cyclomatic complexity of the above flow graph is 4.

GRAPH MATRICES

- A *graph matrix* is a square matrix whose size (i.e., number of rows and columns)
  is equal to the number of nodes on the flow graph.
  - It is a basic path testing which tells about the path that are involved in a program.
- Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b.*

- To this point, the graph matrix is nothing more than a tabular representation of a
  flow graph.
  However, by adding a *link weight* to each matrix entry, the graph matrix
  can become a powerful tool for evaluating program control structure during testing.
  - ✔ The link weight is 1 (a connection exists) or 0 (a connection does not exist).

FIGURE 17.6
Graph matrix



CONTROL STRUCTURE TESTING

It is another method to test control structure. It is highly effective and improves the quality of white-box testing. There are three types of testing in control structure testing they are

1. Condition testing
2. Data Flow testing
3. Loop Testing.

1. Condition Testing

The condition testing is test case design method that exercise the logical conditions contained in a program module.

A relational expression takes the form

E1 **<relational-operator>** ***E2***

where *E1* and *E2* are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A *compound condition* include OR (|), AND (&) and NOT (¬).

Eg.

If (a+b) ≥ (c+a)

(a+b) E1,
(c+a) (E2) they are arithmetic expression.
≥ is a relational operator.

In this condition the following things are tested.

1. Without E1
2. Without E2
3. With E1 & E2
4. E1 greater and E2 lesser
5. E1 positive and E2 Negative
6. E1 Negative and E2 Positive.
7. Condition is test for true.

8. Condition is test for false.

If a condition is incorrect, then these, **types of errors occurs:**

• Boolean operator error (incorrect/missing/extra Boolean operators).
• Boolean variable error.
• Boolean parenthesis error.
• Relational operator error.
• Arithmetic expression error.

Data Flow Testing:

Dataflow Testing focuses on the points at which ***variables receive values*** and the points at which these ***values are used***. It checks the correctness of the du-paths.

Test case definitions based on four groups of coverage

– All definitions

– All c-uses

– All p-uses

– All du-paths

c-uses (computation uses)

The variable occurrences in the right hand side of an assignment statement, or an output statement.

p-uses (predicate uses)

Occur in the predicate portion of a decision statement such as if-then-else, while-do etc.

du-path

A sub-path from a variable definition to its use.

Syntax:

DEF(*S*) = {*X* | statement *S* contains a **definition** of *X*}
USE(*S*) = {*X* | statement *S* contains a **use** of *X*}

Eg.

```
 main()
{
    int a,b,c;   Line No -1

   a=70;       Line No -2
   b=50;       Line No – 3
   c=76;       Line No – 4

a=a+20;    Line No -5

if(a>b) then printf("%d",a);   Line No -6
}
```

Definition of S

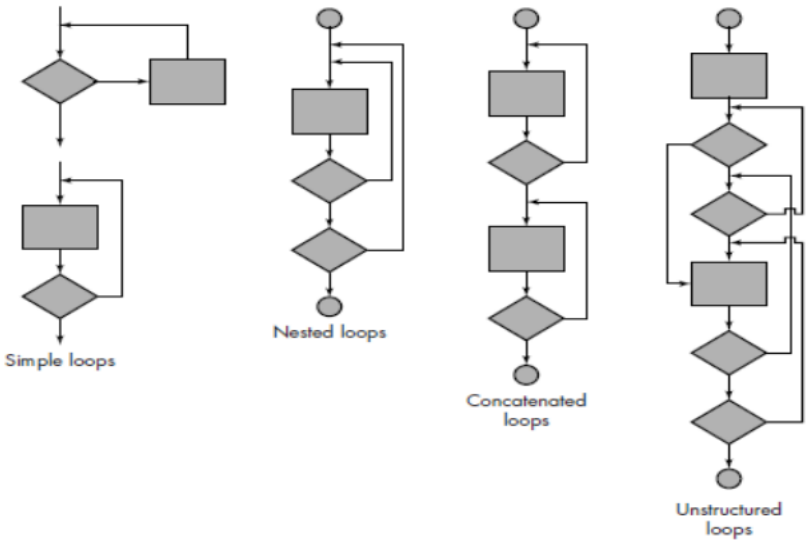
C-Use of S

P-USE of S

Loop Testing

This testing will test the different kinds of loops inside a program some of the loops are

1. Simple loop
2. Nested Loop
3. Concatenated loop
4. Unstructured loop.



Generally the loops are tested for the following reason

1. Skipping the loop fully
2. Execute the loop only once
3. Execute the loop for two times
4. Execute the loop for M time(m<n)
5. Execute the loop for n-1 time, n time, n+1 times.

BLACK-BOX TESTING

The technique of testing without having any knowledge of the interior workings of the application is Black Box testing. The tester does not have access to the source code. This testing can be tested by the end user or a tester. Typically, when performing a black box test, a tester will interact with the system's user interface by *providing inputs* and *examining outputs without knowing how and where the inputs are worked upon.*

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. Black box testing is not an alternative to white box rather it is and complementary approach.

Black-box testing attempts to find errors in the following categories:

(1) Incorrect or missing functions
(2) Interface errors
(3) Errors in data structures or external database access
(4) Behavior or performance errors
(5) Initialization and termination errors.

Graph-Based Testing Methods

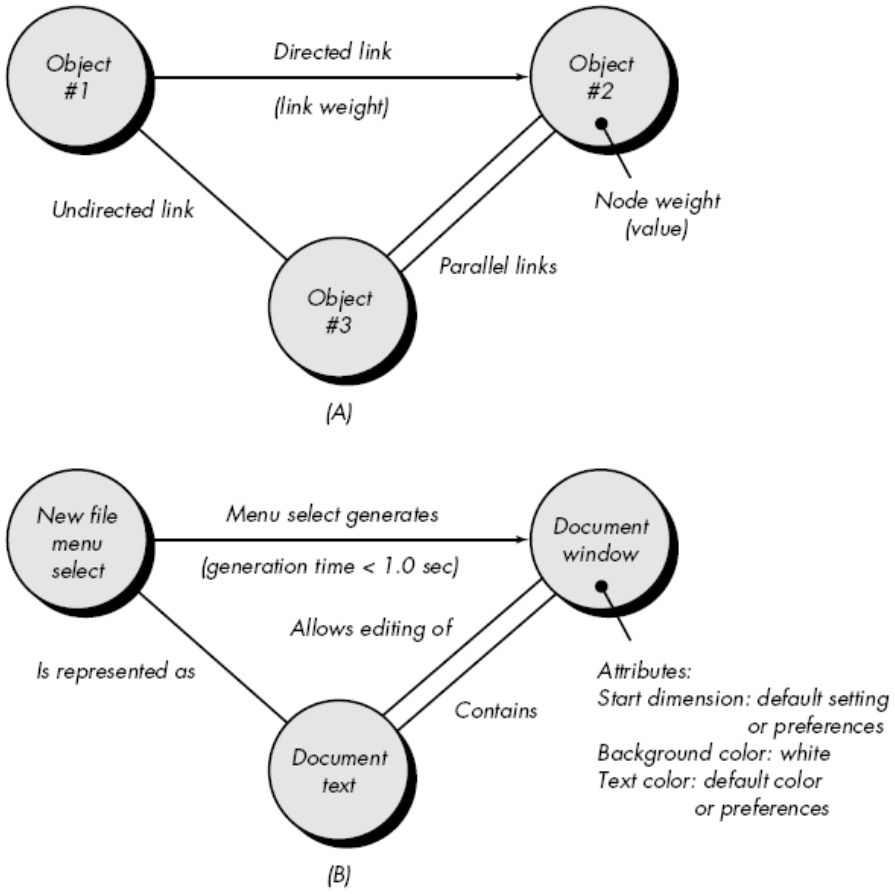
Software testing begins by creating a graph of

important objects and their relationships and tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, the software engineer begins by creating a graph —

- a collection of nodes that represent objects;
- links that represent the relationships between objects;
- node weights that describe the properties of a node (e.g., a specific state behavior);
- Link weights that describe some characteristic of a link.

FIGURE 17.9
(A) Graph notation
(B) Simple example



The symbolic representation of a graph is shown in Figure 17.9A. Nodes are represented as circles connected by links that take a number of different forms.

- A directed link (represented by an arrow) indicates that

a relationship moves in only one direction.

- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions.
- Parallel links are used when a number of different relationships are established between graph nodes.

Following describes a number of behavioral testing methods that can make use of graphs:

- Transaction flow modeling.
- Finite state modeling.
- Data flow modeling.

Equivalence Partitioning

Equivalence partitioning is a black-box testing method that ***divides the input domain of a program into classes of data*** from which test cases can be derived.

Example:

The Below example best describes the equivalence class Partitioning:

Assume that the application accepts an integer in the range 100 to 999

Valid Equivalence Class partition: 100 to 999 inclusive.

Non-valid Equivalence Class partitions: less than 100, more than 999, decimal numbers and alphabets/non-numeric characters

An equivalence class represents a set of valid or invalid states for input conditions. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

Boundary Value Analysis

Boundary value analysis leads to a selection of test cases in the boundaries. Boundary value analysis is a type of black box or specification based testing technique in which

tests are performed using the boundary values.

Example:

An exam has a pass boundary at 40 percent, merit at 75 percent and distinction at 85 percent. The ***Valid Boundary*** values for this scenario will be as follows:

39, 40 - for pass
74, 75 - for merit
84, 85 - for distinction

Boundary values are validated against both the valid boundaries and invalid boundaries.

The ***Invalid Boundary*** Cases for the above example can be given as follows:

0 - for lower limit boundary value
101 - for upper limit boundary value

Procedure:

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

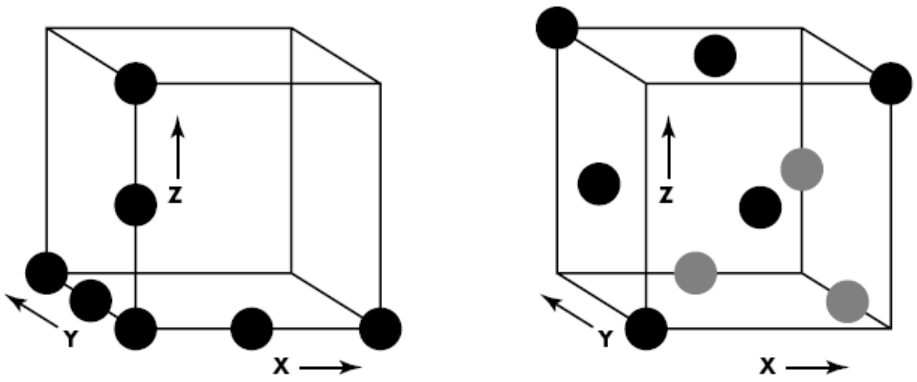
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and

maximum numbers. Values just above and below minimum and maximum are also tested.

Orthogonal Testing

Orthogonal array testing is a ***systematic and statistical*** way of a black box testing technique used when number of inputs to the application under test is small but too complex for an exhaustive testing. The orthogonal array testing method is particularly useful in ***finding errors associated with region faults***—an error category associated with faulty logic within a software component.

FIGURE 17.10
A geometric view of test cases [PHA97]



One input item at a time

L9 orthogonal array

To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions.

If a “one input item at a time” testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1,1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).

An L9 orthogonal array for the fax send function is illustrated in Figure 17.11.

FIGURE 17.11 An L9 orthogonal array

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Advantage

If we have 3 parameters, each can have 3 values then the possible Number of tests using conventional method is 3^3 = 27 While the same using OAT, it boils down to 9 test cases.

UNIT – V

Software Project Management

Software Project Management involves planning, monitoring and controlling the people and the process throughout the software project development.

Different people manage different things in different way. For example the software engineer plan, monitor and control all technical activities, the project manager plan, monitor and control the software team and the senior manager acts as an interface between business and software professionals.

Management Spectrum:

The management spectrum deals about 4 P's . They are

- ✔ People
- ✔ Product
- ✔ Process
- ✔ Project.

PEOPLE:

The Players

The software process is populated by players who can be categorized into one of five constituencies:

1. **Senior managers** who define the business issues that often have significant influence on the project.
2. **Project (technical) managers** who must plan, motivate, organize, and control the practitioners who do software work.
3. **Practitioners** who deliver the technical skills that are necessary to engineer a product or application.
4. **Customers** who specify the requirements for the software to be engineered and other *stakeholders* who have a peripheral interest in the outcome.
5. **End-users** who interact with the software once it is released for production use.

Team Leaders

Model of leadership:

Motivation: The ability to encourage technical people to produce to their best ability.

Organization: The ability to mold existing processes that will enable the initial concept to be translated into a final product.

Ideas or innovation: The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

Problem solving: An effective software project manager can diagnose the Successful project leaders apply a problem solving management style.

Managerial identity: A good project manager must take charge of the project. He/She must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement: To optimize the productivity of a project team, a manager must reward initiative and accomplishment and demonstrate through his own actions that controlled risk taking will not be punished.

Influence and team building: An effective project manager must be able to "read" people; she must be able to understand verbal and nonverbal signals and react to the needs of the people sending these signals. The manager must remain under control in high-stress situations.

The Software Team

The "best" team structure depends on the management style of the organization, the number of people who will populate the team and their skill levels, and the overall problem difficulty

Best Software team is formed on the management style of the organization. The

following factors to be considered when framing a software team.

- ✔ The difficulty of the problem to be solved.

- ✔ The size of the program
- ✔ The life time of the team
- ✔ Quality and reliability of the system to be built
- ✔ Based on delivery date
- ✔ Based on Communication skills.

There are four “organizational paradigms” for software engineering team.

1. A ***Closed Paradigm*** structure team is formed when producing software that is quite similar to past efforts and less likely to be innovative.
2. A ***Random Paradigm*** structure team is formed when innovation and technological breakthrough is required they struggle working in an “orderly performance”.
3. A ***Open Paradigm*** structure team is formed when we need a collaboration i.e it combines the structure nature of closed and innovative idea of random paradigm. It is suited for complex problem.
4. A ***Synchronous paradigm*** team is formed based on the natural compartmentalization of the problem and organizes the team members to work on piece of the problem with little active communication.

Coordination and Communication Issues

There are many reasons the software project get into trouble. One of the main reasons is communication among team members.

In order to have a effective communication between the team members there are two approaches.

1. Formal Communication
2. Informal Communication

Formal Communication

It is accomplished through writing, structured meeting etc.

Informal Communication

It is more personal. Members of the team share ideas on an ad hoc basis. They ask for the help when problem arise and interact with one another on a daily basis.

5.1.2 THE PRODUCT

The product and the problem it is intended to solve at the very beginning of the project. At a minimum, the scope of the product must be established and bounded.

Software Scope

The first software project management activity is the determination of *software scope.* Scope is defined by answering the following questions:

Context: How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?

Information objectives: What customer-visible data objects are produced as output from the software? What data objects are required for input?

Function and performance: What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?

Software scope must be unambiguous and understandable at the management and technical levels.

Problem Decomposition

Problem decomposition, sometimes called *partitioning* or *problem elaboration,* activity that sits at the core of software requirements analysis

Decomposition is applied in two major areas:

(1) the functionality that must be delivered

(2) The process that will be used to deliver it.

Divide and conquer method is applied for complex problem. The complex problem is partitioned into smaller problem.

Problem decomposition is done to develop a reasonable plan inorder to calculate the cost and schedule.

5.1.3 THE PROCESS

The project manager must select which process model that is most appropriate for

1. The customer who have requested the product and the people who will do the work.
2. The characteristics of the product itself
3. The project environment which the software team works.

Melding the Product and the Process

Project Planning begins with melding of the product and the process

The organization has adopted the following set of framework activities

1. **Communication** – Before the project is started all the requirements should be gathered or collected from the user through proper communication. Communication can be done in many ways such as
    - Feedback
    - Questionnaires
    - Open dialogue
    - E-mail
    - Chat etc..
2. **Planning**- After all the requirements are collected the project should be properly planned. The planning is done on the following basis
    - Estimation(Cost)
    - Schedule (Time, No of persons)
    - Tracking (checking whether everything works fine)
3. **Modeling**- After the communication and planning step the project should be properly modeled. The project can be designed using different techniques
    - Tree structure diagram
    - Data flow diagram
    - UML diagrams.
4. **Construction** – This is the step where the programmer starts to write the program. This involves Coding and testing. Testing is done by

giving some sample inputs and checks the output.

5. **Deployment** – The software delivered to the customer's site ,supported documentation is given and feedback is obtained.

COMMON PROCESS FRAMEWORK ACTIVITIES	communication			planning			modeling			construction			deploy-ment		
Software Engineering Tasks															
Product Functions															
Text Input															
Editing and Formatting															
Automatic copy edit															
Page layout capability															

Figure 2: Melding the Problem and the Process

The process framework establishes a skeleton for project planning. The job of the project manager is to estimate the resource requirement, start and end date and final work product.

Process Decomposition

Many Types of software engineering models are available, such as

- The linear sequential model
- The prototyping model
- The RAD model
- The incremental model
- The spiral model
- The concurrent development model

Once the process model is chosen the process framework (communication, planning, modeling, construction and deployment) is adapted

For example, a **small relatively simple project** might require the following work tasks for the *customer communication* activity:

1. Develop list of clarification issues.
2. Meet with customer to address clarification issues.

3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required.

Now, if we consider **a more complex project**, which has a broader scope and more significant business impact. Such a project might require the following work tasks for the customer communication activity:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with the customer.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, function, and behavioral features of the software.
7. Review each mini-spec for correctness, consistency, and lack of ambiguity.

5.1.4 THE PROJECT

In order to manage a successful software project, we must understand what can go wrong (so that problems can be avoided) and how to do it right.

Signs of Project failure

1. Software people don't understand their customer's needs
2. The product scope is poorly defined
3. Changes are managed poorly
4. The chosen technology changes.
5. Sponsorship is lost.
6. Business needs change.

Five-part commonsense approach to software projects:

1. Start on the right foot.

This is accomplished by working hard to understand the problem for the beginning of the project. It is reinforced by building right team and giving the team autonomy and technology.

2. Maintain momentum.

To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum

3. Track progress.

For a software project, progress is tracked as work products are produced and approved as part of a quality assurance activity by conducting FTR.

4. Make smart decisions.

The decisions of the project manager and the software team should be to "keep it simple."

5. Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual get feedback from team members and customers, and record findings in written form.

5.2 FORMAL TECHNICAL REVIEWS

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

The objectives of the 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.

Each FTR is conducted as a meeting and will be successful only if it is properly planned, controlled, and attended.

5.2.1 The Review Meeting

Every review meeting should abide by the following constraints:

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.

- The review meeting is attended by the review leader, all reviewers and the producer.
- One of the reviewers takes on the role of the recorder.
- The FTR begins with an introduction of the agenda and a brief introduction by the producer.
- The producer then proceeds to "walk through"

At the end of the review, all attendees of the FTR must decide whether to

- Accept the product without further modification.
- Reject the product due to severe errors.
- Accept the product provisionally(minor errors have been encountered and must be corrected).

5.2.2 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.
- A formal technical review summary report is completed.

A review summary report answers three questions:

1. What was reviewed?
2. Who reviewed it?
3. What were the findings and conclusions?

The review summary report is a single page form (with possible attachments).

The review issues list serves two purposes:

- To identify problem areas within the product
- To serve as an action item checklist that guides the producer as corrections are made.

5.2.3 Review Guidelines

Guidelines for conducting formal technical reviews must be established in

advance, distributed to all reviewers, agreed upon, and then followed.

Guidelines for formal technical reviews:

1. ***Review the product, not the producer.***
   - ✔ The FTR should leave all participants with a warm feeling of accomplishment.
   - ✔ Conducted improperly errors should be pointed out gently; the tone of the meeting should be loose and constructive.
   - ✔ The review leader should immediately halt a review which goes out of control.
2. ***Set an agenda and maintain it.***

   According to the Agenda given already the FTR must be kept on track and on schedule.
3. ***Limit debate and rebuttal.***

   When an issue is raised by a reviewer, the issue should be recorded for further discussion off-line without spending time in debate.
4. ***Enunciate problem areas, but don't attempt to solve every problem noted.***

   A review is not a problem–solving session. Problem-solving should be postponed until after review meeting.
5. ***Take written notes.***

   It is sometimes a good idea for the recorder to make notes on a wall board, so that wording and priorities can be assessed by other reviewers as information is recorded.
6. ***Limit the number of participants and insist upon advance preparation.***
   - ✔ Keep the number of people involved to the necessary minimum.
   - ✔ All review team members must prepare in advance.
7. ***Develop a checklist for each product that is likely to be reviewed.***
   - ✔ A checklist helps the review leader to structure the FTR meeting
   - ✔ Helps each reviewer to focus on important issues.
8. ***Allocate resources and schedule time for FTRs.***
   - ✔ For reviews to be effective, they should be scheduled as a task during the software engineering process.
   - ✔ Time should be scheduled for the certain modifications that will occur as the result of an FTR.

9. *Conduct meaningful training for all reviewers.*
   - ✔ To be effective all review participants should receive some formal training.
   - ✔ The training should stress both process-related issues and the human psychological side of reviews.
10. *Review your early reviews.*
   - ✔ Debriefing can be beneficial in uncovering problems with the review process itself.
   - ✔ The very first product to be reviewed should be the review guidelines themselves.
