Software Engineering:

Definition 1:

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Definition 2:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machine

Software Crisis:

Software crisis is characterized by inability to develop the desired Software Project because of such problems:

- **Projects running over-budget.**
- **Projects running over-time.**
- **Software is inefficient.**
- **Software is of low quality.**
- **Software does not meet requirements.**
- **Project is unmanageable/ Code difficult to maintain.**

The roots of the software crisis are complexity, expectations, and change. To avoid software crisis, software engineering principles and process are applied strictly.

Waterfall Model:

The waterfall model, sometimes called the classic life cycle model suggests a systematic, sequential approach to software development. This model is named “Waterfall Model”, because its diagrammatic representation resembles waterfall.

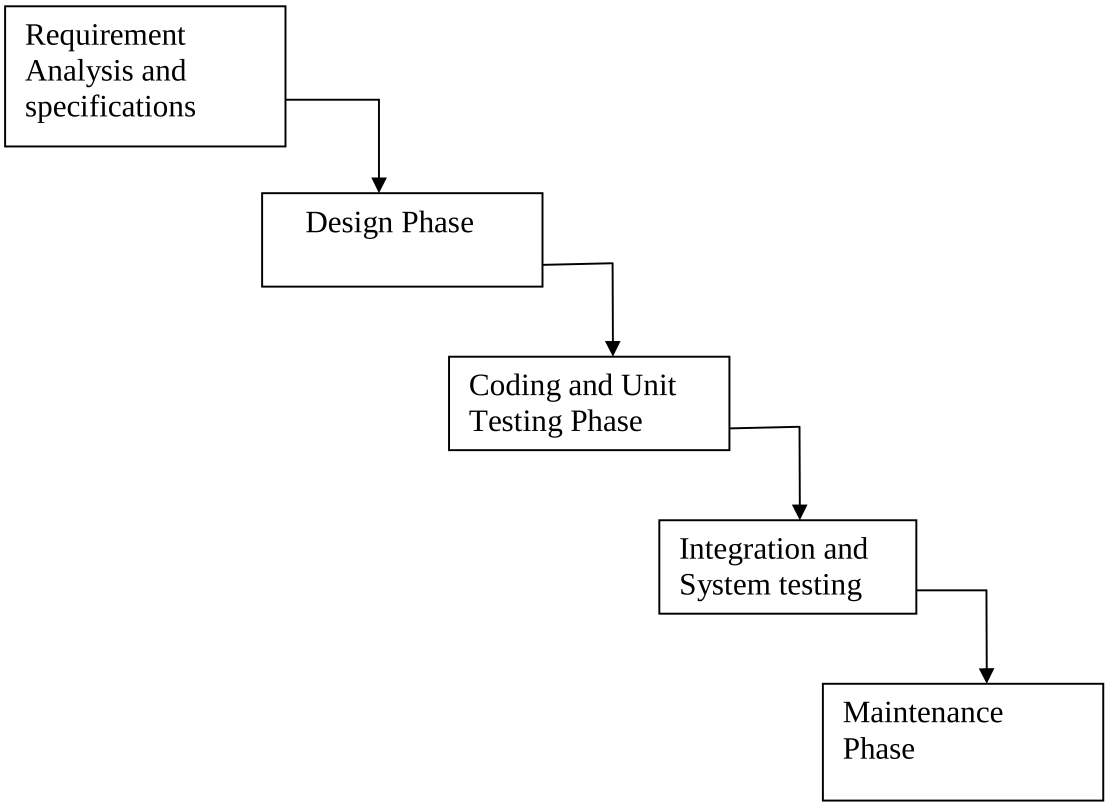
Requirement analysis and specification phase:
The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. This requirement describe the “what” of a system, not the “how”. This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document. This SRS document may act as contract between the developer and the customer.

Design phase:
The goal of this phase is to transform the requirements specifications into a structure that is suitable for implementation in some programming language. Design phase focuses on four distinct attributes of a program: software architecture, data structure, procedural detail, interface representations. This work is documented and known as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the Implementation phase.

Coding and unit testing phase:
The design must be translated to into a machine-readable form. Use of appropriate programming language is important activity in this phase. If SDD is complete, the coding phase proceeds smoothly, because all information needed by the software developers is contained in SDD. After coding individual modules are tested and this testing called unit testing. Testing is an activity in which the error in the program are identified and removed. During testing it is necessary to test the given set of input whether producing correct output or not.

Integration and System Testing:
In this phase, all the modules of the software are integrated and System testing takes place. System testing involves the testing of the entire system, because through this it can be found that the interfaces between the different modules are also functionally correct or not.

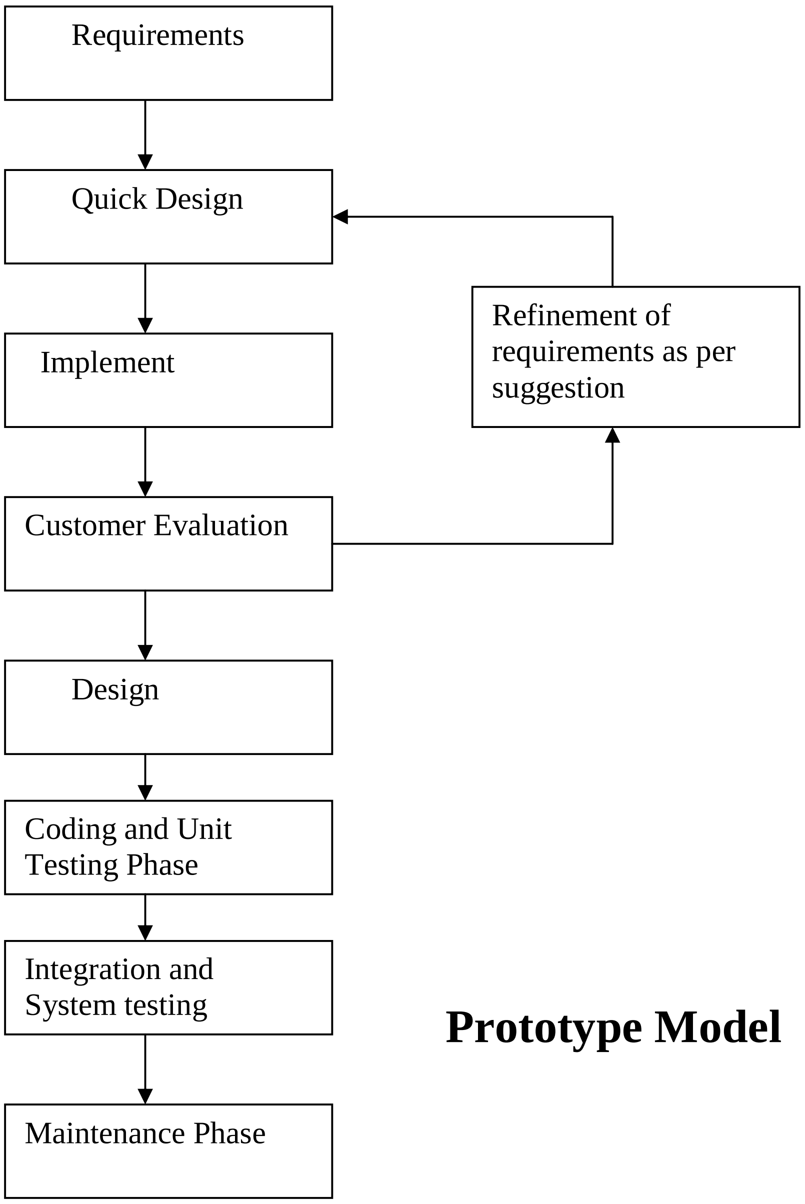
Maintenance Phase:
When software is installed on the customer’s site, then Maintenance phase starts. The phase includes the following activities: error correction, software capabilities enhancement, small modifications in the software with time.


Requirement Analysis and specifications
Design Phase
Coding and Unit Testing Phase
Integration and System testing
Maintenance Phase

Waterfall Model

Prototype model:

A working prototype (has limited functional capabilities, low reliability, and untested performance) is developed as per current available requirements. The developer uses this prototype to refine the requirements and prepare the final specification document. When the prototype is created, it is reviewed by the customer and gives feedback to the developer that helps to remove uncertainties in the requirements of the software, and starts an iteration of refinement in order to further clarify requirements as shown in the diagram. After the finalization of software requirement and specification (SRS) document, the prototype is discarded and actual system is then developed using the Waterfall approach. Thus, it is used as an input to waterfall model and produces maintainable and good quality software.


Requirements
Quick Design
Implement
Refinement of requirements as per suggestion
Customer Evaluation
Design
Coding and Unit Testing Phase
Integration and System testing
Maintenance Phase

Prototype Model

Spiral Model:

The spiral model, originally proposed by Boehm, is an evolutionary software process model that couples the iterative nature of prototype with the controlled and systematic aspects of the linear sequential model. A spiral model is divided into a number of framework activities, also called task regions. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.

Figure depicts a spiral model that contains six task regions.

Customer communication: tasks required to established effective communication between developer and customer.

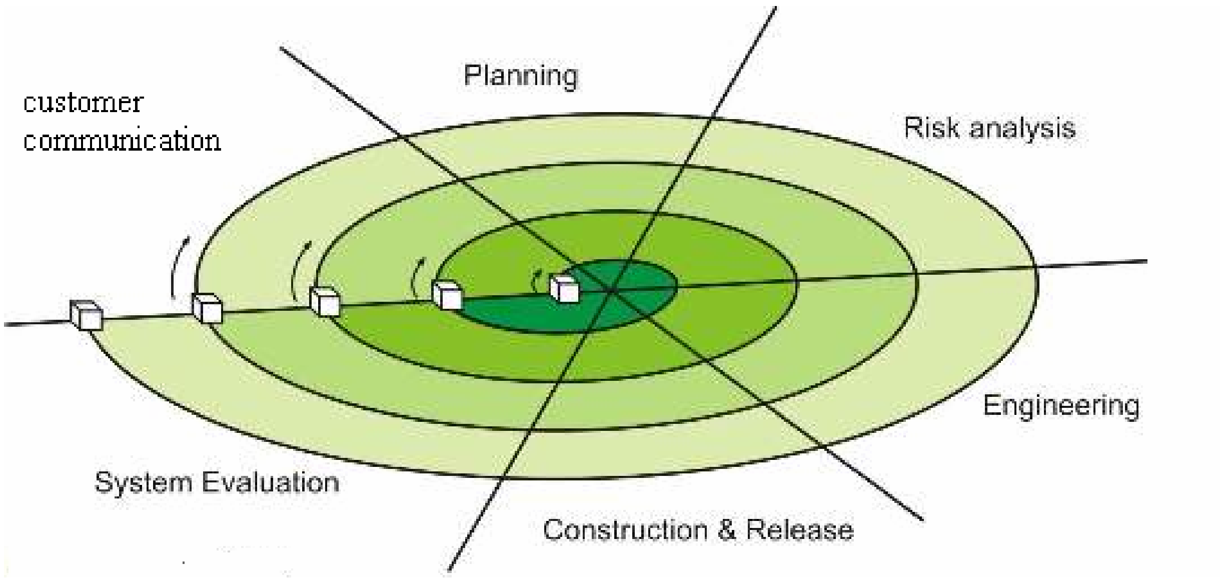
Planning: tasks required to defined resources, timelines and other project related information.

Risk analysis: tasks required to assess both technical and management risks.

Engineering: tasks required to build prototype of the application.

Construction and release: tasks required to construct, test, install and provide user support.

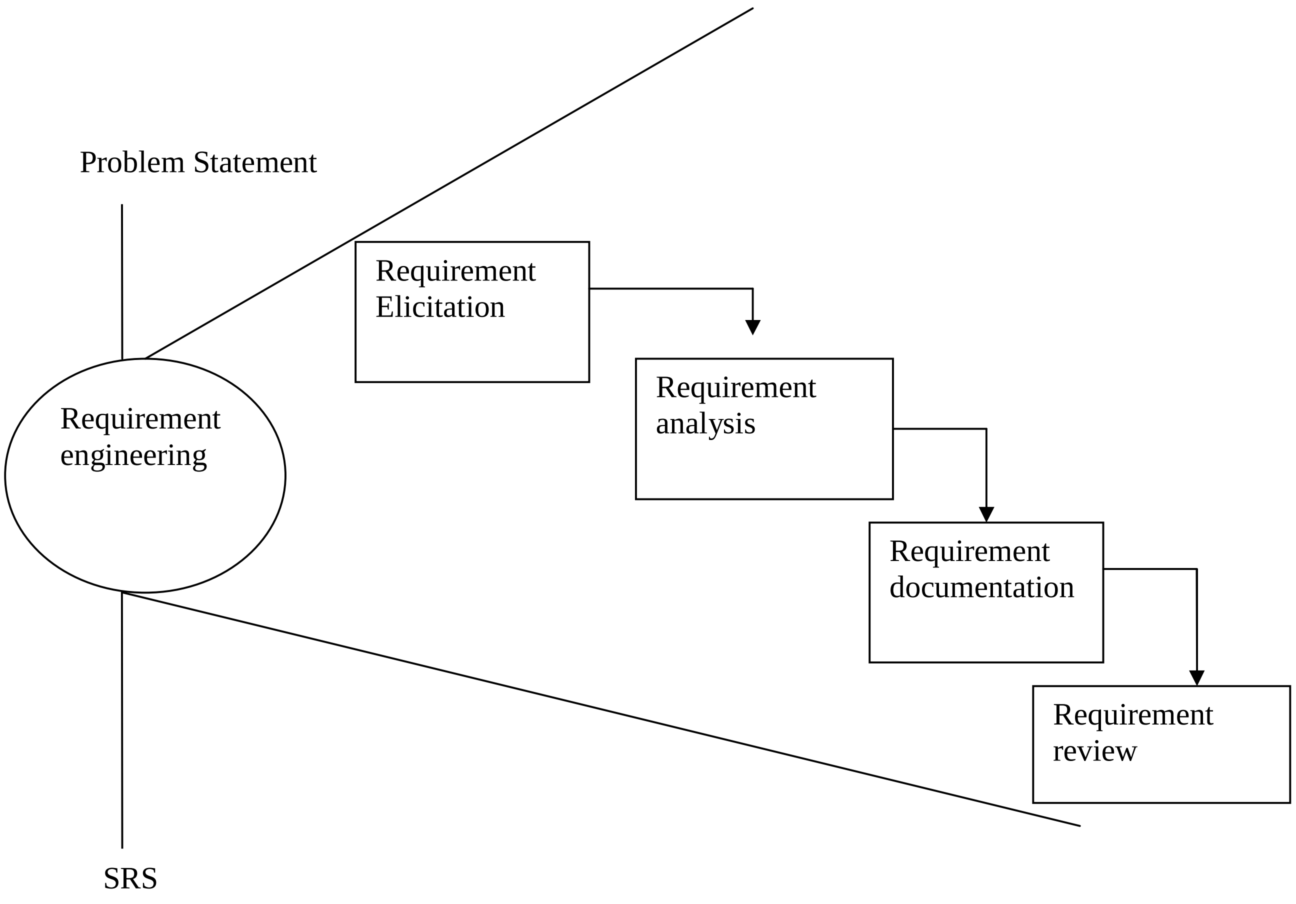
Customer evaluation: task required to obtain customer feedback.



Requirement Engineering:

Requirement engineering produces one large document contains a description of what the system will do without describing how it will do. The input to requirements engineering is the problem statement prepared by the customer. The problem statement may give an overview of the existing system along with broad expectations from the new system. The software engineering process involves the following activities (crucial process steps):

Requirement elicitation: In Requirement elicitation, requirements are identified with the help of customer and existing system processes.
Requirements analysis: The requirements are analyzed in order to identify inconsistencies, ambiguities, missing requirements.
Requirement Documentation: After Requirement elicitation and analysis, one large document is prepared. This document is called known as software requirement specification (SRS) and this documentation works as the foundation for the design of the software.
Requirements review: the review process is carried out to improve the quality of the SRS. It may also be called as requirements verification.



Requirements Elicitation:

In requirements engineering**,** requirements elicitation is the practice of obtaining the requirements of a system from users and customers. The real requirement actually resides in user's mind; and Requirements elicitation is to find out what users really need. Requirements elicitation is perhaps the most difficult, most critical, most error-prone, and most communication intensive aspect of software development. Elicitation can succeed only through an effective customer-developer partnership. Requirements elicitation is a part of the requirements engineering process, usually followed by Requirement analysis and Requirement documentation. The requirements elicitation techniques are: interviews, Brainstorming Sessions, Facilitated Application Specification Technique (FAST), and Qualified Function Deployment (QFD), Use Case.

Facilitated Application Specification Technique (FAST):

This approach is similar to brainstorming session. The objective of the approach is to bridge the expectation gap – a difference between what developers think they are supposed to build and what customers think they are going to get. In order to reduce expectation gap, a team oriented approach is developed for requirement elicitation and is called Facilitated Application Specification Technique (FAST). In this technique, meeting is conducted at some neutral site which is attended both by developers and customers. The meeting is controlled by a facilitator. Participants also agree not to debate. Each participant presents his or her list of objectives and services required, and these lists are displayed in the meeting. Discussions are held on these objectives and services. And after this, combined lists for these topics are prepared by eliminating redundant entries and adding new ideas. And the same process is repeated many times and many sessions are held for this; till a consensus (agreement) is reached.

Quality Function Deployment (QFD):

It is one of the Requirement elicitation techniques and it used to find out the requirements that enhance the quality of the Software Product. Here, in QFD Prime concern is the customer satisfaction and emphasizes on the requirements which are valuable to the customer and to add these valuable requirements to the SRS. To satisfy the customer needs, in QFD technique, every requirement can be given some value on a scale of 1 to 5 according to their importance to the customer. And in QFD, it has been suggested that Quality in the software product is achieved if we fulfill following requirements:

Normal requirement: The objective and goals of the proposed software comes under the category of Normal requirements and these normal requirements must be fulfilled. e.g. entry of the marks, calculation of the marks, reports generation are the Normal requirements for the result management system.
Expected requirements: Expected requirements are implicit requirements, and customers do not state them explicitly. e.g. for wrong entry of data, there must be proper message, and protection from unauthorized access is also an expected requirement.

Exciting requirement: Exciting requirements are the requirements which go beyond the customer's expectations and prove to very satisfy when present. In Result Management System, if reports can be generated in various formats; even customers have not demanded, and this type of requirement is called exciting requirement and enhance the quality of product.

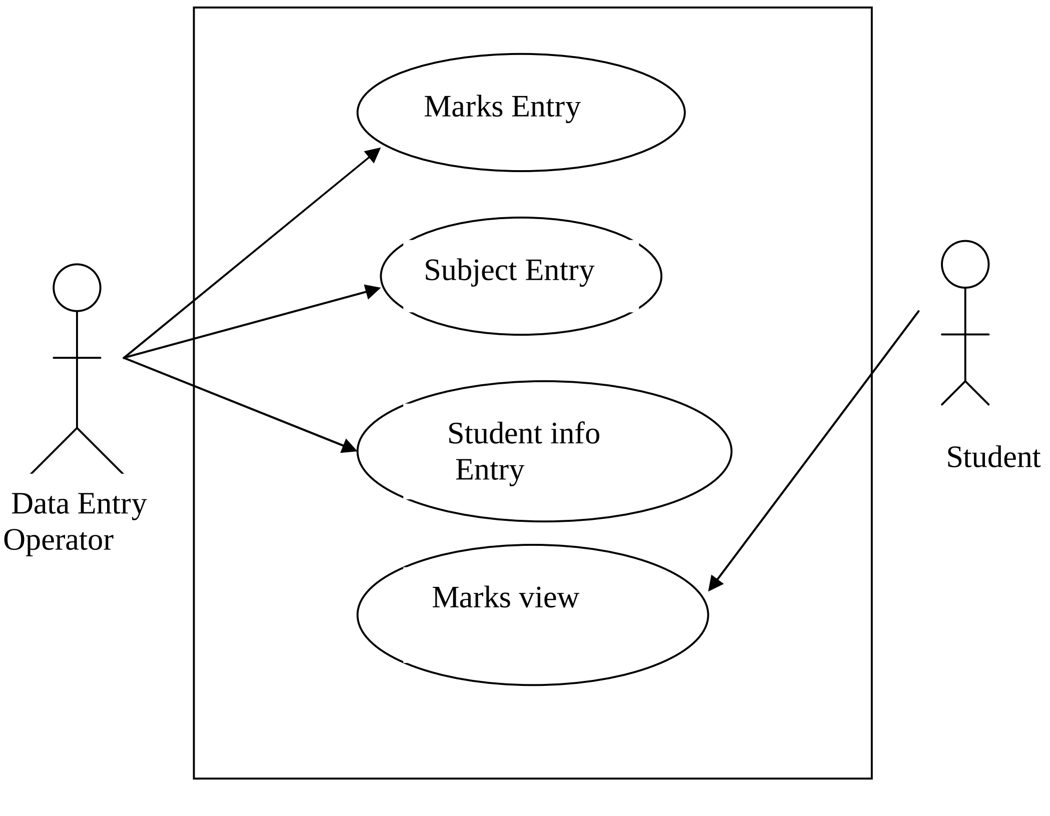
Use case Approach:

Use case approach is the one of approach of Requirement Elicitation in which interaction between the user and proposed software system is studied to find out Requiements for the proposed system. The terms use case, use case scenario, and use case diagram are different.

Use case scenario: use scenario is the unstructured description of user requirements. eg. If we explain the interaction between the user and system as a story description.

Use case: use case is the structured description of user requirements. Eg. if we represent all the interaction in a template or in a table for requirement elicitation.

Use case diagram: use case diagram are graphically represent of interaction between the user and system so that requirements can be elicited. Use case diagram for Result Management System is given

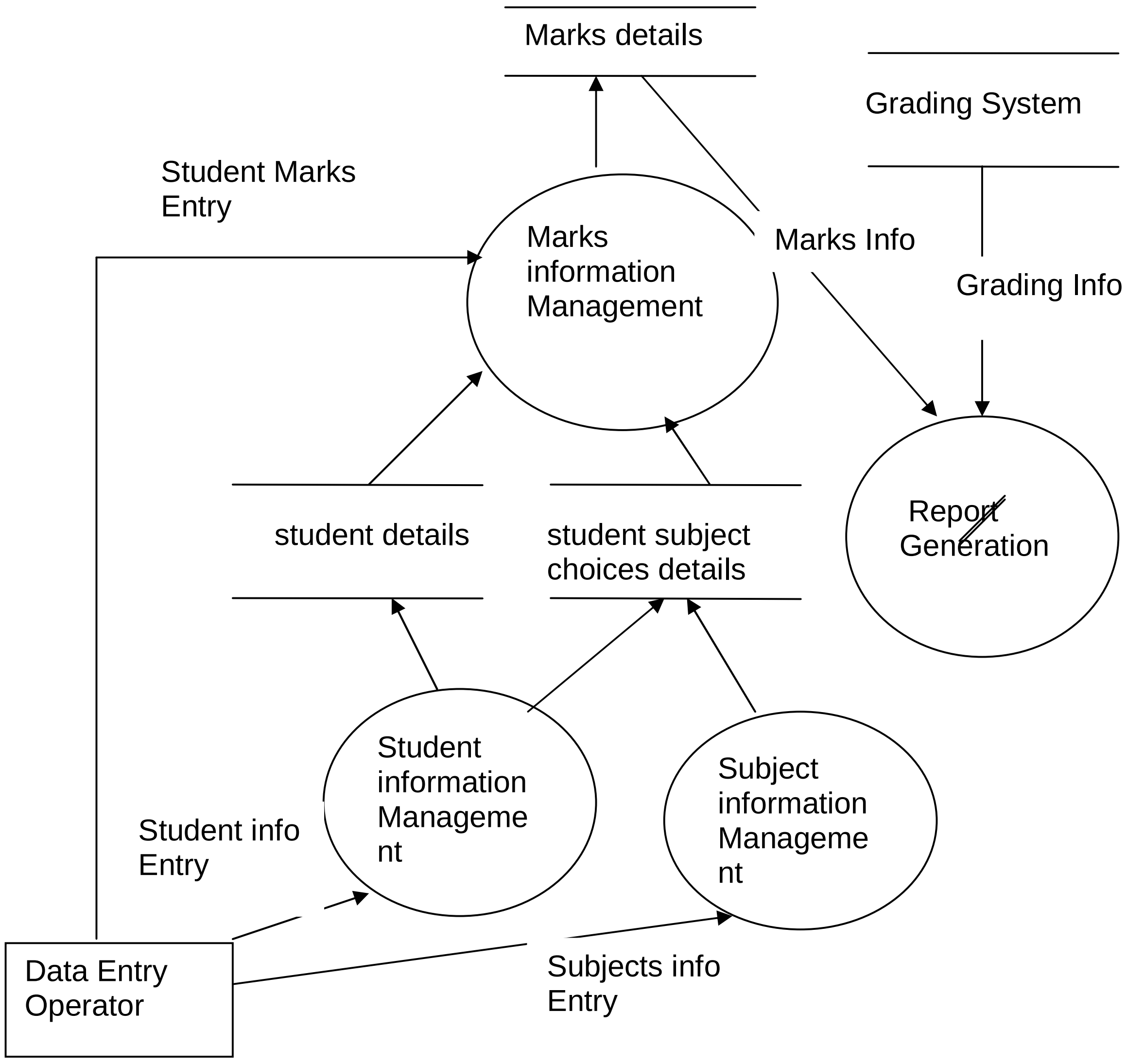


Result Management System

Data flow diagram (DFD):

Data flow diagram (DFD) is a simple graphical notation that can be used to represent a system in terms of the input data to the system, various processing is carried out on these data, and the output data is generated by the system. The following are the basic element of a DFD. DFD are used widely for modeling the requirements. The DFD is also known as a data flow graph. The following are the basic elements of a DFD:

Process: Perform some transformation of input data to yield output data.
Data Flow: used to connect processes to each other, to sources sinks; the arrowhead indicates direction of data flow.
Source or Sink: A source of system inputs or sinks of system outputs.
Data store: it is a repository of data.

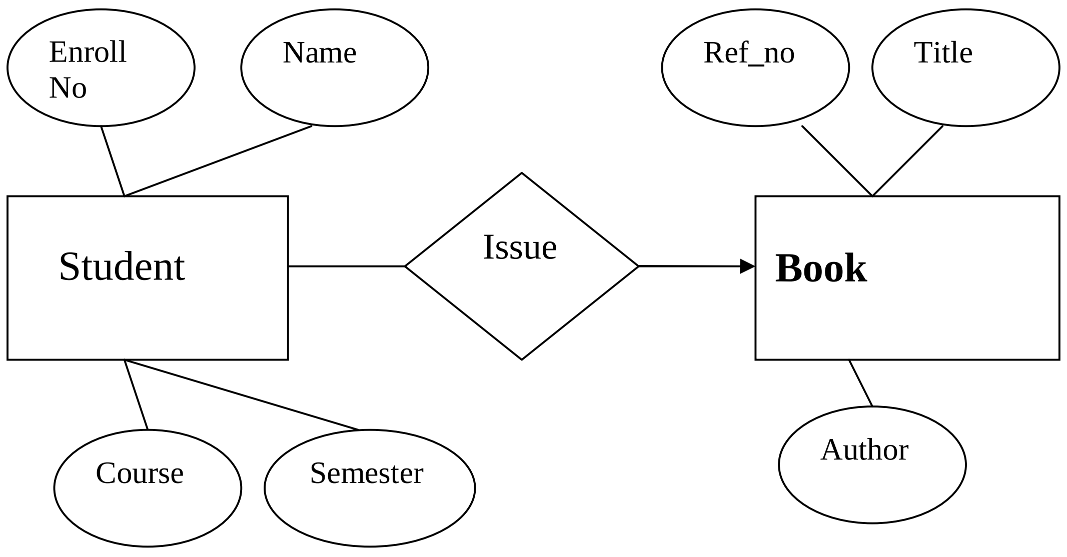


Result Management System

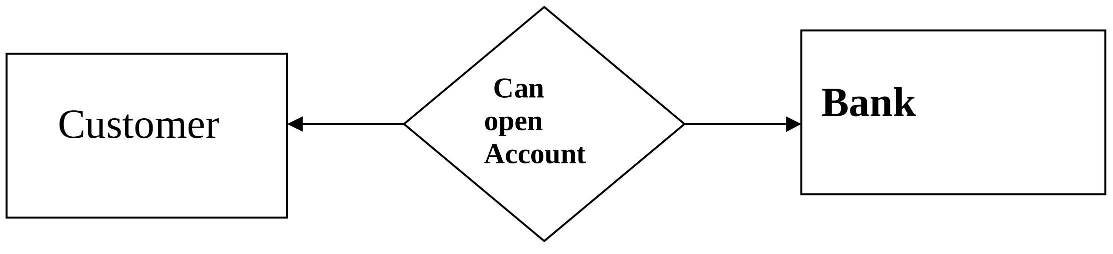
Entity Relationship Diagram:

An Entity Relationship Diagram (ER Diagram) is a logical representation of data. Entity Set, Relationship set, and attributes are the main elements of ER Diagram

Entity Set: An object is called an entity and entities with the similar properties are called Entity Set. An Entity Set is represented by a rectangle.
Attributes: the properties of an entity are called its attributes, and these are represented by Ovals.
Relationship Set: Associations between two entities are called Relationship and Relationships with the similar properties are called Relationship Set. Relationship Set is represented by a rectangle. Examples of ER-Diagrams are:



(ER-Diagram of Student-Book)



(ER-Diagram of Customer-Bank)

Software Design:

Design Phase of software development deals with transforming the customer requirements as described in the SRS document into a form that can be implementable using a programming language. In order to be easily implementable in a conventional language, the following items must be designed during the design phase:

- Different modules of the proposed software.
- Interfaces among Different Modules.
- Algorithms required to implement the individual Modules.

Design is highly significant phase in the software development where the designer plans "how" software should be produced in order to make it functional, reliable and reasonable easy to understand, modify and maintain.

Modularity:

Modularity is single most important characteristic of a well-designed software system. Each module should be well defined and has a single, well defined purpose. And each module can be separately compiled and reusable in other software systems. Modular system are easier to understand and easier to code, test and debugging.

Cohesion and Coupling:

Cohesion is a measure of the degree to which the elements of a module are functionally related and Coupling is the measure of the degree of interdependence between modules.
Two modules with high coupling are strongly interconnected to each other and two modules with low coupling are not dependent on each other. In software designing good software system must be decomposed into modules which have high cohesion and low coupling.

Classification of Cohesiveness:

There are different types of Cohesiveness and the strength of Cohesiveness from highest Cohesiveness (best) to lowest Cohesiveness (worst) is given below:

Functional Cohesion	Sequence Cohesion	Communication Cohesion	Procedural Cohesion	Temporal Cohesion	Logical Cohesion	Coincidental Cohesion

Best $\xrightarrow{\hspace{10em}}$ Worst

Functional Cohesion: Function Cohesion is said to exist if different elements of a module cooperate to achieve a single objective.

Sequence Cohesion: Sequence Cohesion is said to exist if elements are of this type that output of some elements are input to other elements.

Communication Cohesion: Communication Cohesion is said to exist if elements of the module refer to the same storage area e.g. array or stack.

Procedural Cohesion: Procedural Cohesion is said to exist if elements of the module are structured in the same way.

Temporal Cohesion: Temporal Cohesion is said to exist if elements of the module execute in the same time span.

Logical Cohesion: Logical Cohesion is said to exist if elements of the module execute similar type of operations.

Coincidental Cohesion: Coincidental Cohesion is said to exist if elements of the module have no relation.

Classification of Coupling:

There are different types of Coupling and the strength of coupling from lowest (best) to highest coupling (worst) is given below:

Data Coupling	Stamp Coupling	Control Coupling	External Coupling	Common Coupling	Content Coupling

Best $\xrightarrow{\hspace{10cm}}$ Worst

Data Coupling: Two modules are data coupled; if they communicate through elementary data item e.g. by passing int, float data as parameter.

Stamp Coupling: Two modules are stamp coupled; if they communicate through composite data e.g. by passing array as parameter.

Control Coupling: Two modules are control coupled; if one module pass any control information to another module.

Common Coupling: Two modules are common coupled; if they share common data e.g. global variables

Content Coupling: Two modules are content coupled; if two modules shared the common code.

Size Estimation:

Estimation of project size is fundamental to estimating the effort and time required to complete the planned software project. The size of a program indicates the development complexity. There are two important metrics to estimate size:

- **Line of Code**
- **Function Count**

Lines of Code (LOC):

The simplest way of problem size is lines of code. This metric is very popular primarily due to the fact that it is simple to use. This metric measure the number of source instructions required to solve a problem. While counting the number of source instructions, lines used for commenting and blank lines are ignored. However, LOC as a measure of problem size has several shortcomings:

Shortcomings/Limitation of LOC:

- Estimating the loc count at the end of a project is very simple, its estimation at the beginning of a project is very tricky. In order to estimate the LOC measures at the beginning of a project, project managers divide the problem into modules, and each module into sub modules, and so on until the sizes of the different leaf-level modules can be approximately predicted.

- LOC gives a numeric value of the problem size that varies with coding style as different programmer’s layout their code in different styles. E.g. one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines.

- LOC focuses on the coding activity alone, ignoring the relative complexity of design, testing etc. Coding is only a small part of the overall software development activities. Also for some problems the design might be very complex, whereas coding might be straightforward.

- Some programmers produce a lengthy and complicated code structure as they do not make effective use of the available instruction set. Such a poorly written piece of code can not be a good metric for size Estimation.

- If a programmer develop very efficient modules and reuse these modules and results less Line of Codes. And if Lines of code is the metric to estimate the effort then it will discourage him to reuse the code.

Function Count:

Function Count measure functionality from user point of view. The base of the function count is what the user requests and what he receives in return from the system.

For Function Count measure functional units are divided in two categories:

Data function types:

- Internal Logical Files (ILF): A user identifiable group of logically related data maintained with the system.

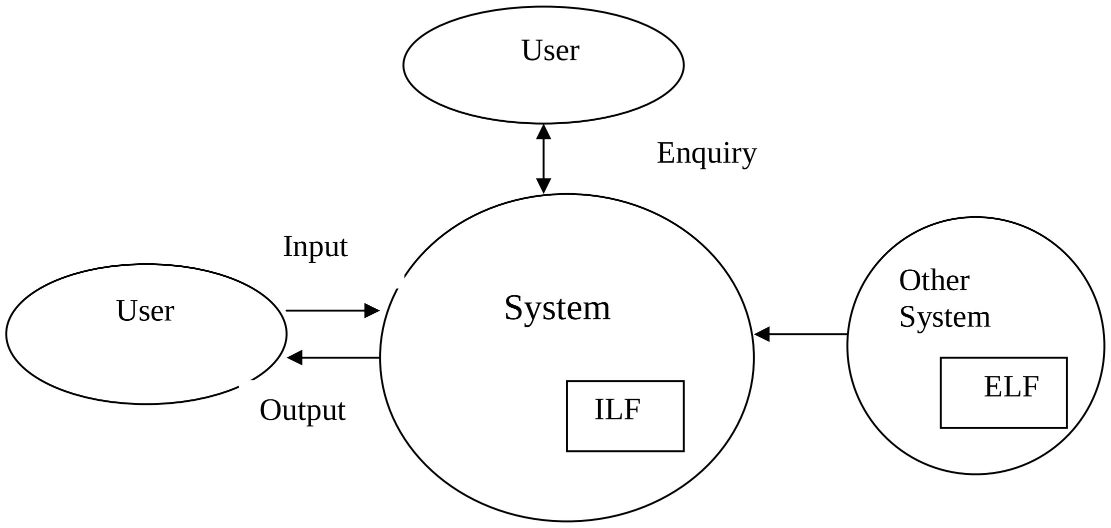
- External Logical Files (ELF): A user identifiable group of logically related data referenced by the system, but maintained within another system.

Transaction function types:

- External Input (EI): An EI is the information that comes from outside to the system for the processing.

- External Output (EO): An EO is the processed information to the outside world.

- External Inquiry (EQ): An EQ is the processed information to the outside world according to the information required by the user.



Functional units for Function Count

The Constructive cost Model (COCOMO):

COCOMO is a hierarchy of software cost estimation models, which include Basic, Intermediate and Detailed sub models, developed by B.W.Boehm in 1981. These models are following:

Basic Model:

Basic Model is used to find the software cost quickly and roughly fashion. Three modes of software development are considered in this model. These modes are:

Organic mode:

- Project Size: 2-50 KLOC
- Team Size: Small
- Nature of Project: Familiar
- Deadline: Not tight

Semi detached mode:

- Project Size: 50-300 KLOC
- Team Size: Medium
- Nature of Project: Medium
- Deadline: Medium

Embedded mode:

- Project Size: Over 300 KLOC
- Team Size: Large
- Nature of Project: Complex
- Deadline: Tight

In Basic model Efforts and Development time are calculated by using following formulas:

$$\mathbf{E = a(KLOC)^{b} * EAF}$$

$$\mathbf{D = c(E)^{d}}$$

Here, E is Effort
D is Development time
a, b, c, d are coefficients

For Organic mode:	a =2.4, b=1.05, c=2.5, d=0.38
For Semi detached mode:	a =3.0, b=1.12, c=2.5, d=0.35
For Embedded mode:	a =3.6, b=1.20, c=2.5, d=0.32

Intermediate Model:

Basic Model is used for a quick and rough estimate, but it resulted in a lack of accuracy. Boehm introduced an additional set of 15 predictors called cost drivers. By computing these cost drivers, we can reach close to the actual price of the project. These cost drivers are further divided into four groups:

Product attributes:

- Required Software Reliability (RELY)
- Database size (DATA)
- Product complexity (CPLX)

Computer attributes:

- Execution time constraints (TIME)
- Main storage constraints (STOR)
- Virtual machine volatility (VIRT)
- Computer Turnaround time (TURN)

Personal attributes:

- Analyst capability (ACAP)
- Application Experience (AEXP)
- Programming Capability (PCAP)
- Virtual machine experience (VEXP)
- Programming Language experience (LEXP)

Project attributes:

- Modern programming practices (MODP)
- Use of software tools (TOOL)
- Required development schedule (SCED)

By multiply the values of these cost drivers, EAF (Effort Adjustment Factor) is computed. In Intermediate model Efforts and Development time are calculated by using following formulas:

$$\mathbf{E = a\ (KLOC)^{\mathit{b}}.\ EAF}$$

$$\mathbf{D = c\ (E)^{d}}$$

Here, in intermediate model values for a, b, c, and d coefficients are:

For Organic mode:	a =3.2, b=1.05, c=2.5, d=0.38
For Semi detached mode:	a =3.0, b=1.12, c=2.5, d=0.35
For Embedded mode:	a =2.8, b=1.20, c=2.5, d=0.32

Detailed COCOMO Model:

Basic Model is used to find the software cost quickly and roughly fashion. In the Intermediate mode, Boehm introduced an additional set of 15 predictors to reach close to the actual price of the project. And in the Detailed model, Boehm further refined his COCOMOL model to find the cost of project.

In the Detailed model, the effort and time for each phase of development can be found and for these phase-sensitive multipliers have been introduced. And here the five phases for development are considered:

- **Plan & requirement**
- **System Design**
- **Detail Design**
- **Module Code & test**
- **Integration and test**

Effort and Schedule for each phase can be found by these formulas

$$E = \tau . E$$

Here, **τ** is a phase sensitive multiplier

$$E = a\,(KLOC)^{b} . EAF$$

$$D = \tau . D$$

Here, **τ** is a phase sensitive multiplier

$$D = c\,(E)^{d}$$

Advantage of COCOMO:

- Easy to use and documented properly
- Adjusted to realistic values to some extent

Shortcomings/ Limitations:

- Mode choice offers some difficulties, since it is not always possible to be sure which of the three models is appropriate.
- Coefficients values may be very for organization to organization.
- It is silent about the involvement of customer

Risk Management:

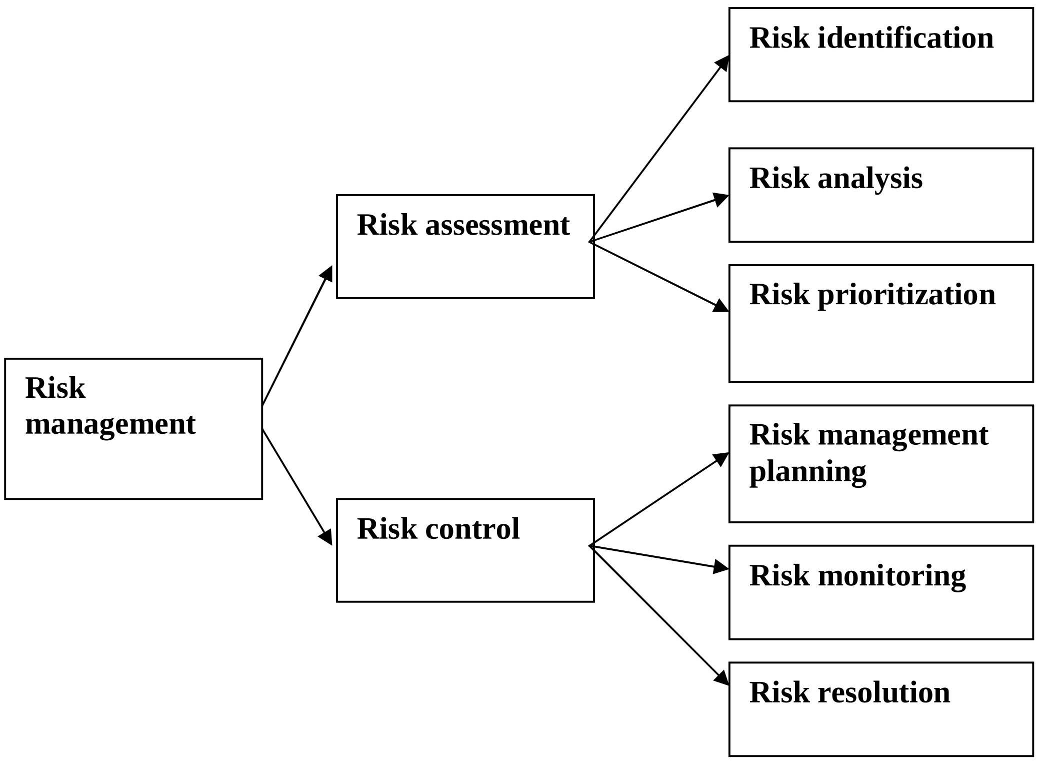
Risk Management is the process of identifying, addressing and eliminating these problems before they can damage the project

Types of Software Risks?

- Dependencies
- Requirement issues
- Management issues
- Lack of Knowledge

Risk Management Activities:

- **Risk Assessment**
- **Risk Control**



Risk Assessment has following activities:

- **Risk identification:** In this activity common risk areas of the project are identified.

- **Risk analysis:** Risk analysis involves how the Risk can be minimized by adopting risk management plans.

- **Risk prioritization:** In this activity loss due to risks are measured and risks are assigned priorities on some scale.

Risk Control has following activities:

- **Risk management planning:** In this activity, a plan is produced for dealing with each significant risk.

- **Risk monitoring:** projects are continuously monitored to resolve the risk as it occurs.

- **Risk resolution:** Risk Resolution is the execution of the risk management plans for dealing with the risk**.**

Software Metrics:
(What and why?):

"The continue application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products".

The conclusion of the above definition is:

Software metrics are all about measurement which, in turn, involves numbers, the use of numbers to make things better, to improve the process of developing software and to improve all aspects of the management of that process. Software metrics are applicable to the whole development life cycle from initiation, when costs must be estimated, to monitoring the reliability of the end product in the field, and the way that product change over time with enhancement.

There are three categories of Metrics:

Product metrics:
Describe the characteristics of the product.
E.g. performance, efficiency, reliability, portability, reusability etc.

Process metrics:
Describe the characteristics of the processes used to develop the software product. E.g. effort, time, number of bugs found during testing etc.

Project metrics:
Describe the characteristics of the project. E.g. number of software developers, cost, schedule etc.

Token Count:
(Halstead software science measures)

Operands and operator of any computer language is called tokens. Any symbol or keyword in a program that specifies an algorithmic action is considered an operator, while a symbol used to represent data is considered an operand.

Halstead at Purdue University developed software sciences family of measures that survive till today. These measures are:

Program Length (N):

$$N=N1+N2$$

Here,

N1: total occurrence of operators
N2: total occurrence of operands

Program Vocabulary (η):

$$\eta= \eta1 + \eta2$$

Here,

$\eta1$: no of unique of operators
$\eta2$: no of unique operands

Volume of a Program (V):

$$V= N * \log_2\eta$$

Here,

N: Program length
η: Program vocabulary

Potential Volume of a Program (V*):

$$V^*= (2+ \eta_2^*) \log_2 (2+ \eta_2^*)$$

Here,

η_2: unique input and output parameters

Program Level (L):

$$L=V^* / V$$

Here,

V*: Potential volume
V : volume

Program Difficulty (D):

$$\mathbf{D = 1 / L}$$

Here, L: Program Level

Estimated Program Length (N):

$$\mathbf{N=\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2}$$

Here,

η_1: no of unique of operators
η_2: no of unique operands

Estimated Program Level (L):

$$\mathbf{L=2\, \eta_2 / (\eta_1 N2)}$$

Here,

η_1: no of unique of operators
η_2: no of unique operands
N2: total occurrence of operands

Estimated Program Difficulty (D):

$$\mathbf{D = 1 / L= \eta_1 N2 / 2\, \eta_2}$$

Here,

η_1: no of unique of operators
η_2: no of unique operands
N2: total occurrence of operands

Effort (E):

$$\mathbf{E = V / L= V*D}$$
$$\mathbf{= (\eta_1 N2 N Log_2\, \eta)\ 2\eta_2}$$

Here,

η_1: no of unique of operators
η_2: no of unique operands
N2: total occurrence of operands
N: total occurrence of operators and operands

Time (T):

$$\mathbf{T=E/ \beta}$$

Here,

E: Effort
β: Stroud number (given by John Stroud), value is 18

Data Structure Metrics:

Line of Code, Function Point, and Token Count are important metrics to find out the effort and time required to complete the project. There are some Data Structure metrics to compute the effort and time required to complete the project. There metrics are:

- **The Amount of Data.**
- **The Usage of data within a Module.**
- **Program weakness.**
- **The sharing of Data among Modules.**

The Amount of Data:

To measure Amount of Data, there are further many different metrics and these are:

- **Number of variable (VARS):**
  In this metric, Number of variables used in the program are counted.

- **Number of Operands (η_2):**
  In this metric, Number of operand used in the program are counted**.**

  $$\eta_2 = \text{VARS} + \text{Constants} + \text{Labels}$$

- **Total number of occurrence of variable (N2):**
  In this metric, total number of occurrence of variables are computed

The Usage of data within a Module:

To measure this metric, average number of live variables are computed. A variable is live from its first to its last references with in procedure.

Average no of Live variables (LV) =

(Sum of count of live variables/ Sum of count of executable statements)

Program weakness:

Program weakness depends on its Modules weakness. If Modules are weak (less Cohesive), then it increase the effort and time metrics required to complete the project.

Average life of variables (γ) =

(Sum of count of live variables/ No. of unique variables)

Module Weakness (WM) = LV* γ

Here,
LV: average no. of live variables.
γ: average life of variables.

Program Weakness (WP) = (ΣWM)/m

Here, **WM**: weakness of module.
m: number of modules in the program.

The Sharing of Data among Module:

As the data sharing between the Modules increases (higher Coupling), no parameter passing between Modules also increased, as a result more effort and time are required to complete the project. So Sharing Data among Module is an important metrics to calculate effort and time

Information Flow Metrics:

Program consists of modules and as the information flow between the Modules increased, the Modules are called low cohesive and high coupled, and as a result, more complex software and requires more effort and time. Therefore effort and time also depend on the Information Flow Metric (IF).

The Basic Information Flow Module:

IF (A) (Information Flow of Component A)

$$= [\text{FAN IN (A)} * \text{FAN OUT (A)}]^2$$

Here,

FAN IN (A) = the number of components calling Component (Module) A.
FAN Out (A) = the number of components that are called by A

The Sophisticated Information Flow Module:

The only difference between the simple and the sophisticated Information Flow (IF) models lies in the definition of FAN IN and FAN OUT .

FAN IN (A) = a+ b + c +d

Here,
a =the number of components that call A
b =the no of parameters passed to A from other components higher in the hierarchy.
c =the no of parameters passed to A from other components lower in the hierarchy.
d =the no of data elements read by component A.

FAN OUT (A) = a+ b + c +d

Here,
a =the number of components that called by A
b =the no of parameters passed from A to other components higher in the hierarchy.
c =the no of parameters passed from A to other components lower in the hierarchy.
d =the no of data elements written to by A.

Hardware Reliability:

There are three phase in the life of a hardware component i.e. burn-in, useful life, wear-out.
Burn-in: in burn-in case failure rate is quite high.
Useful life: during this period, failure rate is quite low and remain constant. The best period is useful life period.
Wear-out: failure rate increases in wear-out phase due to wearing out/aging of component

The hardware reliability curve is given below:

For Diagram refer the Book (K.K.Aggarwal)

Software Reliability:

“Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specific period of time.”

“it is a probability of a failure free operation of a program for a specific time in a specific environment”.

The Software reliability curve is given below:

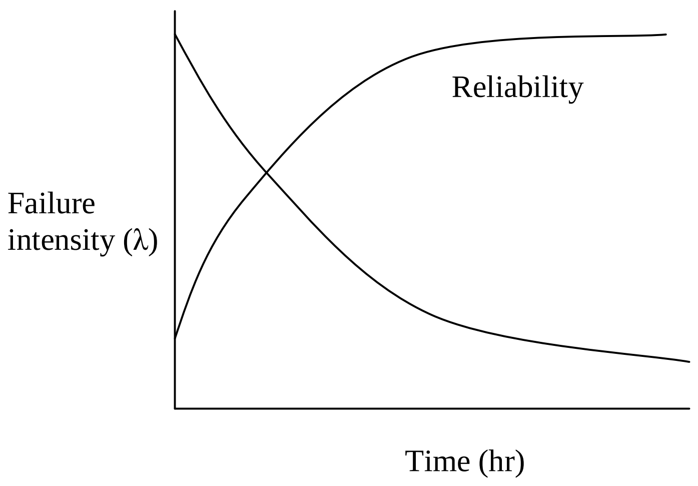
For Diagram refer the Book (K K Aggarwal)

Fault:

- A fault is defect in the program.
- When program makes error it is called fault.
- It is a property of the program.

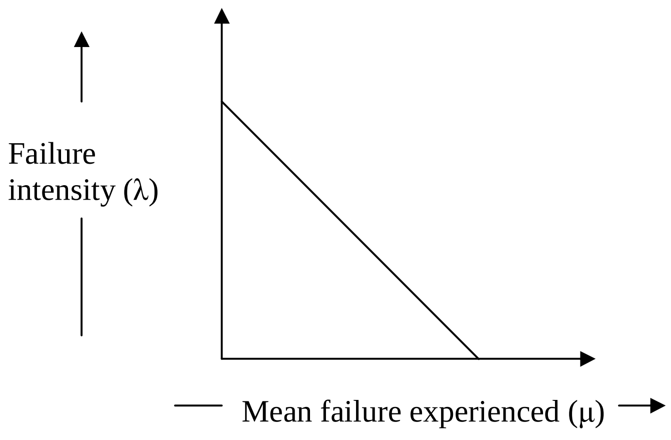
Failure:

- When fault is executed under particular condition, causes a failure.
- Failure is the property of behavior of program and is effected by two principal Factors:
    1. The number of faults in the software being executed.
    2. The execution environment (operational profile).



Basic Execution time Model:

The model was developed by J.D. MUSA in 1979 and is based on execution time. According to this model, the decrease in failure intensity, as a function of the number of failures observed, is constant and is given in the below figure:



Current failure intensity:

$$\lambda(\mu) = \lambda_0(1 - \mu/V_0)$$

Here,

λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period
μ: Average or expected no of failures experienced at a given point in time.

Decrement of failure intensity per failure:

$$d\lambda/d\mu = - \lambda_0 / V_0$$

Here,

λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period

Failure experienced after some time (execution time given):

$$\mu (\tau)= V_0 (1-\exp(- \lambda_0 \tau / V_0))$$

Here,

τ : execution time
λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period

Failure intensity after some time (execution time given):

$$\lambda (\tau)= \lambda_0 \exp (- \lambda_0 \tau / V_0)$$

Here,

τ : execution time
λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period

Additional failures required to reach the failure intensity objective (which are given):

$$\Delta\mu = (V_0 /\lambda_0) (\lambda_p - \lambda_f)$$

Here,

λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period
λ_p: present failure intensity.
λ_f: failure intensity objective.

Additional execution required to reach the failure intensity objective (which are given):

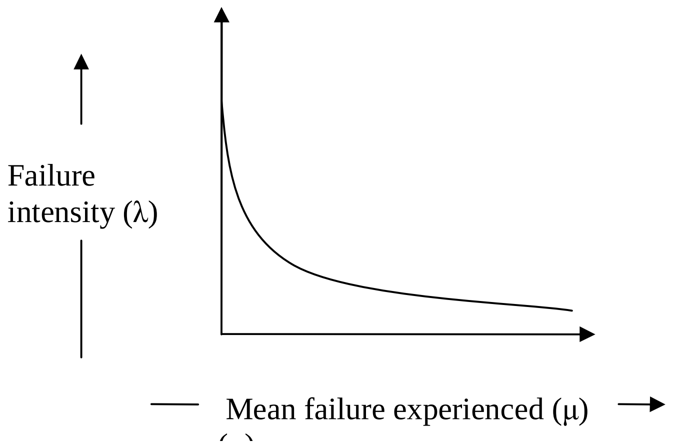
$$\Delta\tau = (V_0 /\lambda_0) \mathrm{Ln} (\lambda_{p/} \lambda_f)$$

Here,

λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period
λ_p: present failure intensity.
λ_f: failure intensity objective.

Logarithmic Poisson Execution time Model:

The model was also developed by J.D. MUSA in 1979. According to this model, the decrease in failure intensity, as a function of the number of failures observed, is exponential and is given in the below figure:



Current failure intensity:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

Here,

λ_0: initial failure intensity at the start of execution
θ: failure intensity decay parameter.

Decrement of failure intensity per failure:

$$d\lambda / d\mu = -\theta \lambda_0$$

Here,

λ_0: initial failure intensity at the start of execution
θ: failure intensity decay parameter.

Failure experienced after some time (execution time given):

$$\mu(\tau) = 1/\ \theta\ \text{Ln}\ (\lambda_0\ \theta\ \tau + 1)$$

Here,

τ: execution time
λ_0: initial failure intensity at the start of execution
θ: failure intensity decay parameter.

Failure intensity after some time (execution time given):

$$\lambda(\tau) = \lambda_0 / (-\lambda_0 \theta\ \tau + 1)$$

Here,

τ: execution time
λ_0: initial failure intensity at the start of execution
V_0: Number of failure experienced, if program is executed for infinite time Period

Additional failures required to reach the failure intensity objective (which are given):

$$\Delta\mu = 1/\ \theta\ \text{Ln}(\ \lambda_p - \lambda_f)$$

Here,

λ_p: present failure intensity.
λ_f: failure intensity objective.

Additional failures required to reach the failure intensity objective (which are given):

$$\Delta\ \tau = 1/\ \theta\ (1/\ \lambda_f - 1/\lambda_p)$$

Here,

λ_p: present failure intensity.
λ_f: failure intensity objective.

Capability Maturity Model (CMM):

CMM was developed by Software Engineering Institute and it is a strategy to improve software quality by improving the process by which software is developed. The five levels of CMM and their characteristics are given below:

Maturity Level	**Characterization**
Maturity Level 1 **(initial)**	**Adhoc process**
Maturity Level 2 **(Repeatable)**	**Basic Project Management**
Maturity Level 3 **(Defined)**	**Process Definition**
Maturity Level 4 (**Managed)**	**Process measurement**
Maturity Level 5 **(Optimizing)**	**Process Control**

ISO-9000:

ISO 9000, maintained by International Organization for Standardization is an internationally recognized family of standards for quality management of businesses ISO-9000 is applicable to wide variety of industries activities; including design, production, installation, and servicing. Within the ISO-9000 Series, ISO-9001 is the standard that is most applicable to software development. Any organization's Registration to ISO 9001 by an accredited certification body shows committed to quality, customers, and a willingness to work towards improving efficiency. An ISO 9001 certificate enhances company image in the eyes of customers, employees and shareholders alike. It also gives a competitive edge to an organization's marketing.

Comparison between CMM and ISO-9000

- CMM emphasis on continue process improvement. ISO-9001 addresses the minimum criteria for an acceptable quality system.
- CMM focuses strictly on software, while ISO-9001 has a much broader scope: hardware, software, processed materials and services.
- Although either document could be used to structure a process improvement program, the more detailed guidance and greater breadth provided to software organization by the CMM suggest that it is a better choice.

Introduction to Software Testing:

Definition:

"Testing is the process of executing a program with intent of finding errors".

Effective testing will contribute to the delivery of higher quality software products, more satisfied users, and lower maintenance costs, more accurate and reliable results. Hence, software testing is necessary and important activity of software development process. It is a very expensive process and consumes one-third to one-half of the cost of a typical development project.

There are three level of software testing:

Unit testing:
In this testing individual module is tested in all possible ways so as to detect any error.

Integration testing:
All modules are integrated and test their interfaces. Concentration is given of internal structures and functions of the software. .

System testing:
In system testing internal structure of the software is ignored and concentration is given on how it responds to the typical kind of operations that will be requested by the user.