Software Engineering

Introduction:- Software engineering as the engineering approach to develop software. Software engineering principles are based on error prevention. Even when errors occur, software engineering principles emphasize detection of errors as close to the point where the errors are committed as possible.

It is the systematic collection of decades of programming experience together with the innovations made by researchers towards developing high quality software in a cost-effective manner. In other word we can say that it is the engineering approach to develop software.

Software engineering techniques are essential for the development of large software products where a group of engineers work in a team to develop the product.

Software Processes: - The process that deals with the technical and management issues of software development is called a software process. A software development project must have at least development activities and project management activities. The fundamental objectives of a process are optimality and scalability. Optimality means that the process should be able to produce high-quality software at low cost, and scalability means that it should also be applicable for large software projects.

Software characteristics Software characteristics are classified into 6 major components:

Functionality: It refers to the degree of performance of the software against its intended purpose. It basically means are the required functions.

Reliability: A set of attribute that Bear on the capability of software to maintain its level of performances understated conditions for a stated period of time.

Efficiency: It refers to the ability of the software to use System Resources in the most effective and Efficient Manner. The software should make effective use of storage space and executive commands as per desired timing requirement.

Usability: It refers to the extent to which the software can be used with ease. Or the amount of effort or time required learning how to use the software should be less.

Maintainability: Refers to the ease with which the modifications can be made in a software system to extend its functionality, improvement, performance or correct errors.

Portability: A set of attributes that bears on the ability of the software to be transferred from one environment to another, without or minimum changes.

Software life cycle: - A software life cycle is the series of identifiable stages that a software product undergoes during its lifetime. The first stage in the life cycle of any software product is usually the feasibility study stage. Commonly, the subsequent stages are : requirements analysis and specification, design, coding, testing and maintenance. Each of these stages is called a life cycle phase.

Model: - A software life cycle model is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. A life cycle model maps the different activities performed on a software product from its inception to retirement.

Classical Waterfall model: - The classical waterfall model divides the life cycle into the phases. This model breaks down the life cycle into an intuitive set of phases. The different phases of this model are:

Feasibility study
Requirements analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

1. Feasibility Study: - The main aim of the feasibility study activity is to determine whether it would be financially and technically feasible to develop the product. The feasibility study activity involves the analysis of the problem and collection of all relevant information relating to the product such as the different data items which would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the behaviour of the system.
2. Requirement analysis and specification: - The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely requirements gathering and analysis, and requirements specification.

a) Requirements gathering and analysis: - The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed with a view to clearly understanding the customer requirements and weeding out the incompleteness and inconsistencies in these requirements.

The requirement analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussion.

b) Requirement specification: - The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document.

3. Design: - The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. The two different design approaches are – traditional design approach and the object-oriented design approach.

a) Traditional design approach: - This approach consists of two different activities – first a structured analysis of the requirement specification is carried out. Second is the structured design activity where the results of structured analysis are transformed into the software design.

b) Object-oriented design approach: - In this approach, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The OOD approach has several benefits such as lower development time and effort, and better maintainability of the product.

4. Coding and Unit testing: - The purpose of this phase is to translate the software design into source code. Each component of the design is implemented as a programme module. The end-product of this phase is a set of programme modules that have been individually tested.

During this phase, each module is unit tested to determine the correct working of all the individual modules.

5. Integration and System Testing: - During this phase, the modules are integrated in a planned manner. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally when all the modules have been successfully integrated and tested, system testing is carried out. System testing usually consists of three different kinds of testing activities: -

a) α – testing : - it is the system testing performed by the development team

b) β – testing: - it is system testing performed by a friendly set of customer.

c) acceptance testing: - it is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered output.

6. Maintenance: - Maintenance involves performing any one or more of the following three kinds of activities:

a) Correcting errors that were not discovered during the product development. This is called corrective maintenance

b) Improving the implementation of the system according to the customer's requirements. This is called perfective maintenance.

c) Porting the software to work in a new environment. This is called adaptive maintenance.

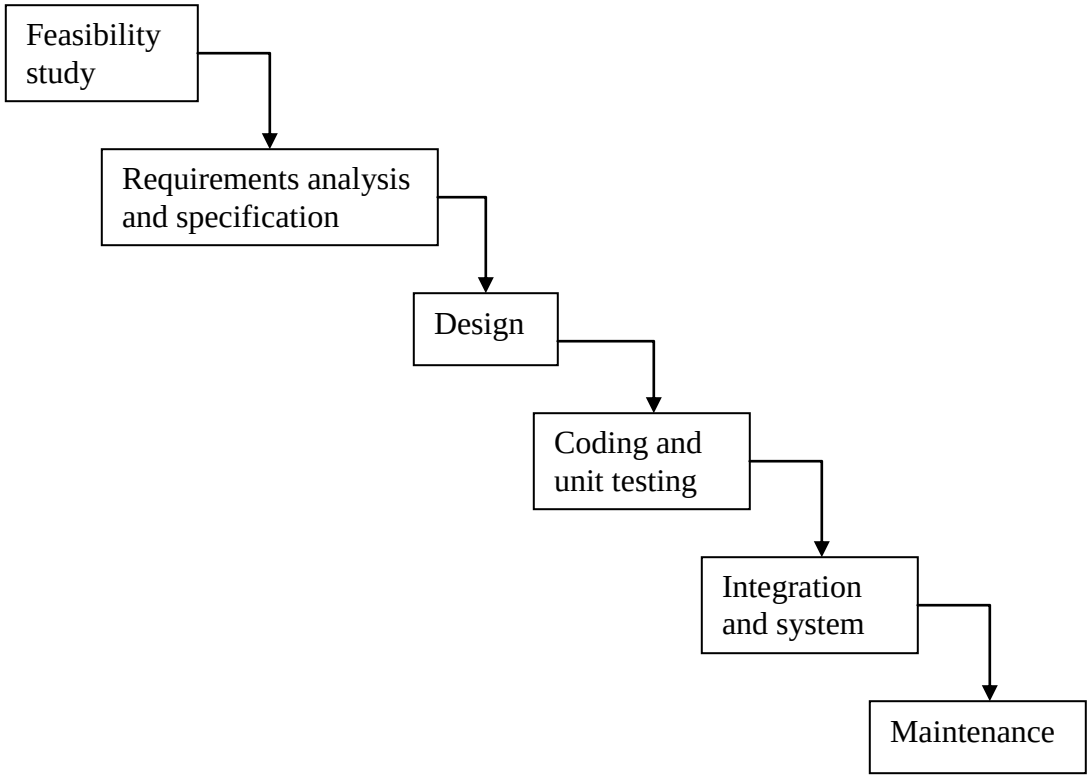


Fig: Classical waterfall model

Disadvantage of waterfall model: -

1. It can not satisfactorily handle the different types of risks that a real life software project is subjected to .
2. To achieve better efficiency and higher productivity, most real life projects can not follow the ridig phase sequence imposed by the waterfall model.

Prototyping model: - The prototyping model suggests that before carrying out the deelopment of the actual software, a working prototype of the system should be built. A prototype usually exhibits limited functional capabilities, low reliability and inefficient perforamnce compared to the actual software. A prototype is usually built using several shortcuts.

In this model, product development starts with an initial requirements gathering phase. A quick design is carried out and the prototype is built. The developed prototype is submitted to the customer for his evaluation. Based on the customer feedback, the requirements are refined and the prototype continues till the customer approves the prototype. The actual system is developed using the iterative waterfall approach.

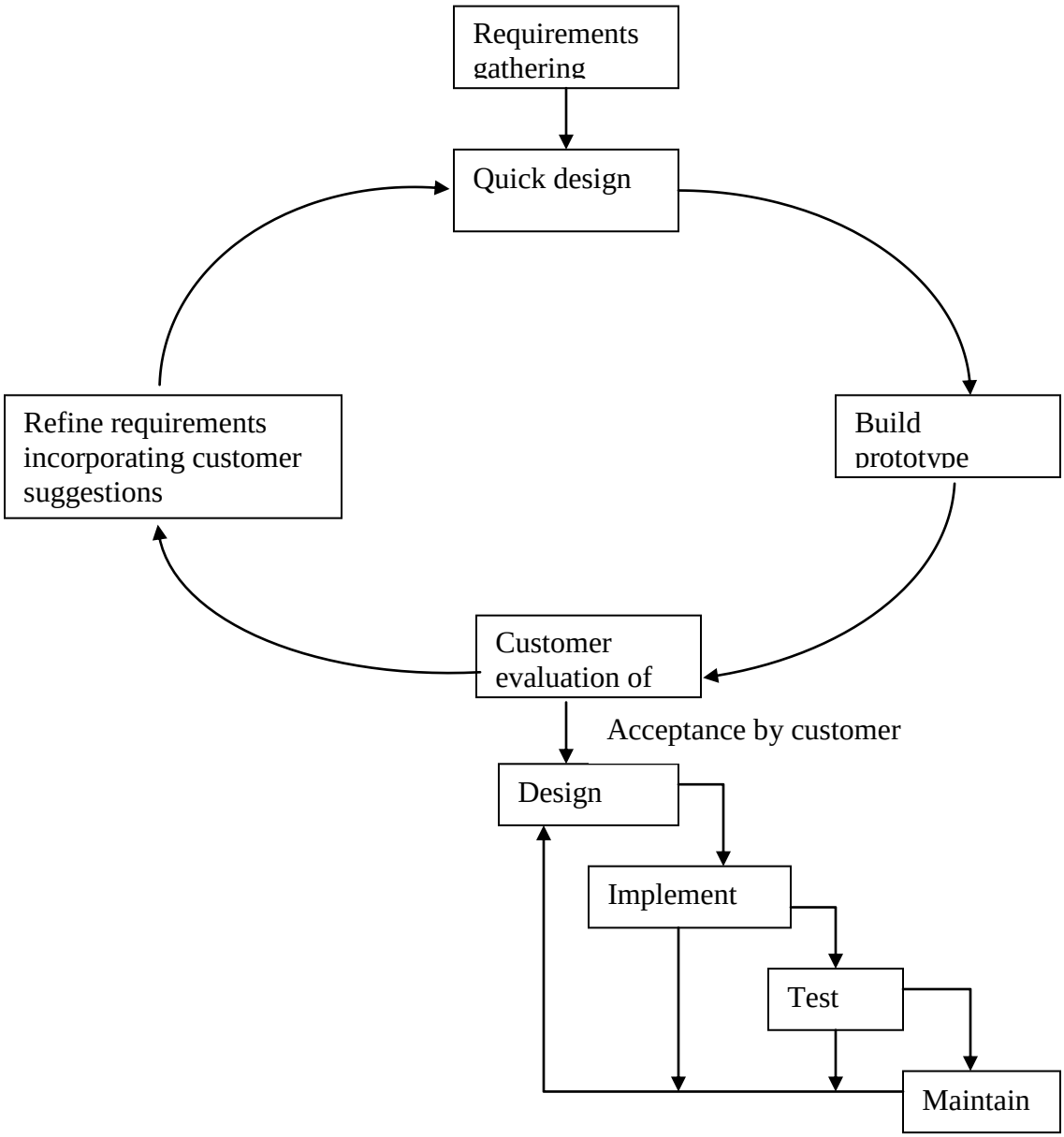


Fig: Prototyping model of software development

Use of prototyping model: -

1. To illustrate the input data formats, messages, reports and the interactive dialogues to the customer.
2. when the technical solutions are unclear to the development team.

Evolutionary model: - In this model, the software is first broken down into several modules which can be incrementally constructed and delivered. The development team first develops the core modules of the system. This initial product is refined into increasing levels of capability by adding new functionalities in successive versions. This life cycle model is also referred to as the successive versions model or incremental model.

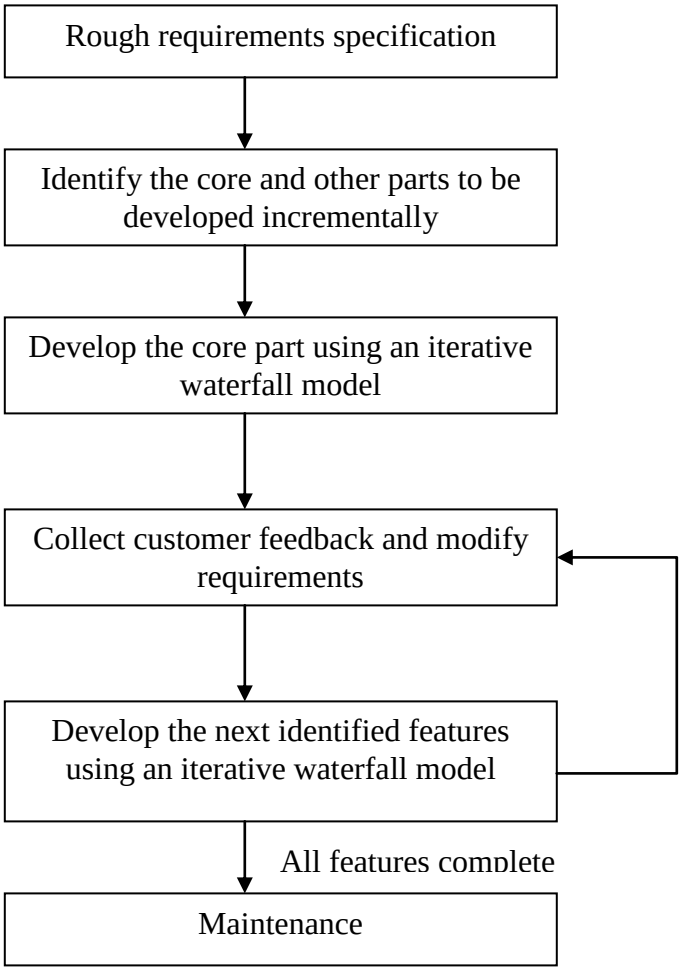


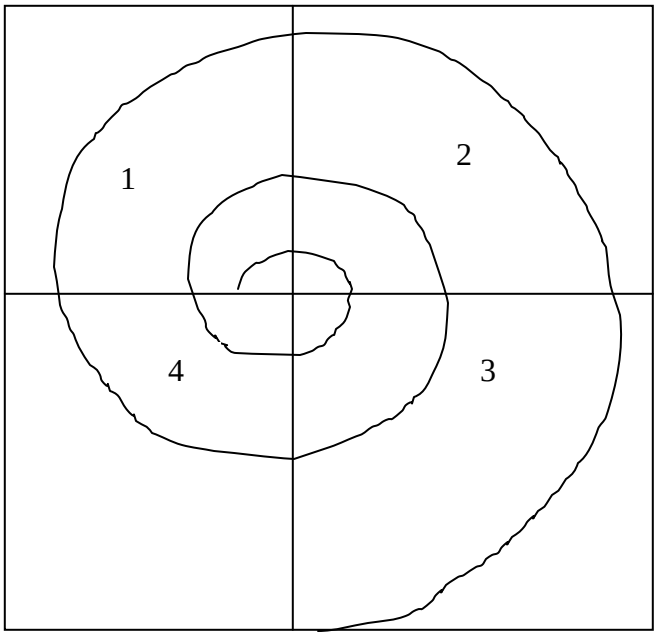
Fig: Evolutionary model of software development

Advantage: -

1. The user gets a chance to experiment with partially developed software.
2. The core module get tested thoroughly

Disadvantage: - For most practical problem it is difficult to divide the problem into several functional units.

Spiral model: - The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process.



Each phase in this model is split into four sectors. The first quadrant identifies the objectives of the phase and the alternative solutions possible for the phase under consideration. During the second quadrant, the alternative solutions are evaluated to select the best solution possible. For the chosen solution, the potential risks are identified and dealt with by developing an appropriate prototype.

Activities during the third quadrant consist of developing and verifying the next level of the product. Activities during the fourth quadrant concern reviewing the results of the stages traversed so far with the customer and planning the next iteration around the spiral. Usually, after several iterations along the spiral, all risks are resolved and the software is ready for development. At this point, a waterfall model of software development is adopted.

Most distinguishing feature of this model is its ability to handle risks. During each iteration, risk analysis through prototype construction allows weighing different alternative available to handle the risks.

Software Requirement Analysis and Specification: *see in classical waterfall model*

Requirement engineering:

Requirements engineering is the discipline that involves establishing and documenting requirements. The various activities associated with requirements engineering are elicitation, specification, analysis, verification and validation, and management.

Requirement Elicitation Technique:

Requirements elicitation is perhaps the most difficult, most error-prone and most communication intensive software development. It can be successful only through an effective customer-developer partnership. It is needed to know what the users really need.

There are a number of requirements elicitation methods. Few of them are listed below –

1. Interviews
2. Brainstorming Sessions
3. Facilitated Application Specification Technique (FAST)
4. Quality Function Deployment (QFD)
5. Use Case Approach

FAST: (**Facilitated Application Specification Technique)**

Its objective is to bridge the expectation gap – difference between what the developers think they are supposed to build and what customers think they are going to get. A team oriented approach is developed for requirements gathering.

Each attendee is asked to make a list of objects that are-

1. Part of the environment that surrounds the system
2. Produced by the system
3. Used by the system

Each participant prepares his/her list, different lists are then combined, redundant entries are eliminated, team is divided into smaller sub-teams to develop mini-specifications and finally a draft of specifications is written down using all the inputs from the meeting.

QFD: (Quality Function Deployment)

In this technique customer satisfaction is of prime concern, hence it emphasizes on the requirements which are valuable to the customer.

3 types of requirements are identified –

- **Normal requirements –** In this the objective and goals of the proposed software are discussed with the customer. Example – normal requirements for a result management system may be entry of marks, calculation of results, etc
- **Expected requirements –** These requirements are so obvious that the customer need not explicitly state them. Example – protection from unauthorized access.
- **Exciting requirements –** It includes features that are beyond customer's expectations and prove to be very satisfying when present. Example – when unauthorized access is detected, it should backup and shutdown all processes.

Data Flow Diagram (DFD): - A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchanged among these functions. In the DFD terminology, it is useful to consider each function as a processing station that consumes some input data and produces some output data.

DFD Symbols: - There are four symbols used in the DFD

1. A square defines a source or destination of system data
2. An arrow identifies data flow
3. A circle represents a process
4. An open rectangle is a data store.

   A DFD describe what data flow rather than how they are processed, so it does not depend on hardware, software, data structure, or file organization.

Rule of DFD: -

1. Processes should be named and numbered for easy reference
2. The direction of flow is from top to bottom and from left to right, although they may flow back to a source.
3. When a process is exploded into lower-level details, they are numbered.
4. The name of data stores, sources, and destination are written in capital letters.

Data Dictionary: - A data dictionary is a structured repository of data about data. It lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on all the DFD.

A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For the smallest units of data items, the data dictionary lists their name and their type.

A data dictionary plays a very important role in any software development process because of the following reasons:

a) A data dictionary provides a standard terminology for all relevant data for use by all engineers working in the same project.

b) The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

Requirement Document:

A software requirements document (also known as software requirements specifications) is a document that describes the intended use-case, features, and challenges of a software application.

These documents are created before the project has started development in order to get every stakeholder on the same page regarding the software's functionality.

SRS:

The production of the requirements stage of the software development process is **Software Requirements Specifications (SRS)** (also called a **requirements document**). This report lays a foundation for software engineering activities and is constructing when entire requirements are elicited and analyzed. **SRS** is a formal report, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements. Also, it comprises user requirements for a system as well as detailed specifications of the system requirements.

Characteristics of **SRS:**

Software requirements specification should be unambiguous, accurate, complete, efficient, and of high quality, so that it does not affect the entire project plan. An **SRS** is said to be of high quality when the developer and user easily understand the prepared document.

i) Concise: - The SRS document should be concise and at the same time unambiguous, consistent and complete.

ii) Structured: - SRS should be well structured. A well-structured document is easy to understand and modify. SRS document undergoes several revisions to cope up with the customer requirement.

iii) Black box view: - The SRS document should specify the external behavior of the system and not discuss the implementation issue.

iv) Traceable: - It should be possible to trace a specific requirement to the design elements that implement it and vice versa. similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and vice versa.

v) Response to undesired events: - It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

vi) Verifiable: - All requirements of the system as documented in the SRS document should be verifiable. This means that it should be possible to determine whether or not requirements have been met in the implementation.

Software Project Planning

Matrics for Project Size Estimation: - In order to be able to accurately estimate the project size, we need to define some appropriate metric or unit in terms of which we can express the project size. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Two metrics are widely used to estimate size: Lines of code (LOC) and Function point (FP).

Lines of Code: - Using this metric, the project size is estimated by counting the number of source instructions in the developed programme.

In order to estimate the LOC count at the beginning of a project, project managers usually divided 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. By using the estimation of the lowest level modules, project managers arrive at the total size estimation. However, LOC has several shortcomings:

1. LOC gives a numerical value of problem size that can vary widely with individual coding style
2. A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it
3. LOC measure correlates poorly with the quality and efficiency of the code
4. LOC metric penalizes use of higher-level programming languages, code, reuse, etc

5. LOC metric measures the lexical complexity of a programme and does not address the more important but subtle issues of logical or structural complexities
6. It is very difficult to accurately estimate LOC in the final product from the problem specification.

Function Count: - Function point metric was proposed by Albercht (1983). One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification.

The conceptual idea is that the size of a software product is directly dependent on the number of different functions or feature it supports. Albercht postulated that in addition to the number of basic functions that software performs, the size is also dependent on the number of files and the number of interfaces.

Besides these, function point metric computes the size of a software product using three other characteristics of the product. The size of a product in function points (FPs) can be expressed as the weighted sum of these five problem characteristics. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP)]

*UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)* 10*

1. Number of inputs: - Each data item input by the user is counted.
2. Number of outputs: - The outputs considered refer to reports printed, screen outputs error messages produced etc.
3. number if inquiries: - Number of inquiries is the number of distinct interactive queries which can be made by the users
4. Number of files: - Each logical file is counted
5. Number of interfaces: - Here the interfaces considered are the interfaces used to exchange information with other system.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. The TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput and response time requirements etc. each of these 14 factors is assigned a value from 0 to 6. The resulting numbers are summed, yielding the total degree of influence (DI). Now, the TCF is computed as (0.65 + 0.01 * DI). As DI can vary from 0 to 70, the TCF can vary from 0.65 to 1.35. Finally, FP = UFP * TCF.

Cost Estimation Model:

COnstructive COst estimation MOdel (COCOMO) : - COCOMO was proposed by Boehm (1981). According to him, any software development project can be classified into three category – organic, semidetached and embedded.

a) **Organic**: - This type of project deal with developing a well-understood application program. The size of the development team is small and members are experienced in developing similar type of projects.

b) **Semidetached**: - The development team of this type of project consists of mixture of experienced and inexperienced staff.

c) **Embedded**: - This type of project is strongly coupled to complex hardware.

Boehm provides different sets of expressions to predict the effort and development time from the size estimation given in KLOC (Kilo Lines of Source Code). One person-month (PM) is the effort an individual can typically put in a month.

According to Boehm, software cost estimation should be done through three category – basic COCOMO, intermediate COCOMO, and complete COCOMO.

1. **Basic COCOMO Model**: - The basic COCOMO model gives an approximate estimate of the project parameters. The expressions are

$$\text{Effort} = a_1 * (\text{KLOC})^{a2} \quad \text{PM}$$

$$\text{Tdev} = b_1 * (\text{Effort})^{b2} \quad \text{month}$$

Where KLOC is Kilo Line of Source Code,

a_1, a_2, b_1, b_2 are constants for each category of software product

Tdev is the estimated time to develop the software

Effort is the total effort required to develop the software product

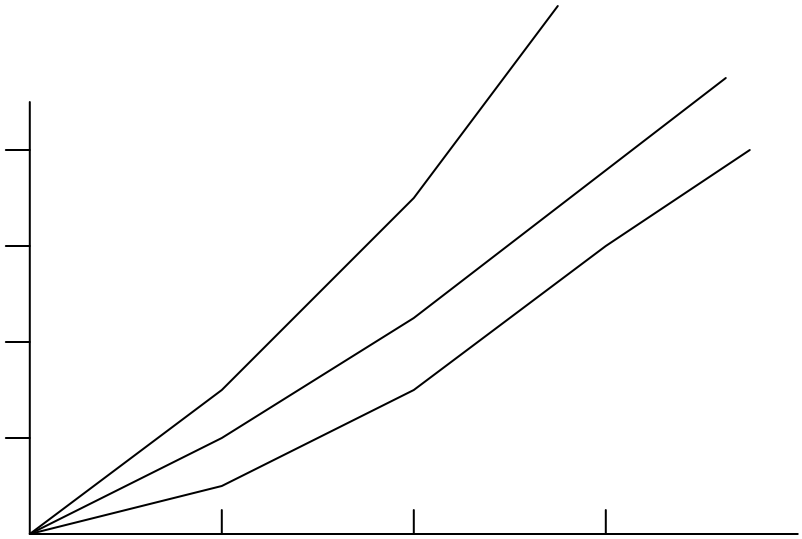
PM for person-month

The Effort for three different categories of software can be estimated as follows

Organic	:	Effort	=	$2.4 (\text{KLOC})^{1.05}$	PM
Semi-detached	:	Effort	=	$3.0 (\text{KLOC})^{1.12}$	PM
Embedded	:	Effort	=	$3.6 (\text{KLOC})^{1.20}$	PM

The developing time can be calculated as

Organic	:	Tdev	=	$2.5 (\text{Effort})^{0.38}$	months
Semi-detached	:	Tdev	=	$2.5 (\text{Effort})^{0.35}$	months
Embedded	:	Tdev	=	$2.5 (\text{Effort})^{0.32}$	months



Risk Management

A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway. If a risk becomes real, it can adversely affect the project and hamper the successful and timely completion of the project. Therefore, it is necessary for the project manager to anticipate and identify different risks that a project is susceptible to, so that contingency plans can be prepared to contain each risk.

Risk management consists of three essential activities – Risk Identification, Risk Assessment and Risk Containment.

i) **Risk Identification**: - The Project manager needs to anticipate the risks in the project as early as possible so that the impact of the risk can be minim9ized by making effective risk management plans. For example, certain modules might not complete in time, poor quality work, whether some of key personnel might leave the organization, etc.

A project can be affected by a large variety of risks. It is necessary to categorize risks in to different classes. There are three main categories of risks which can affect a software project as follows:

a) **Project risks**: Project risk concerns various forms of budgetary, schedule, personnel, resource and customer related problem. An important project risk is schedule slippage.

b) **Technical risks**: Technical risk concern potential design, implementation, interfacing, testing and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty and technical obsolescence.

c) **Business Risk**: - This type of risks include risks of building an excellent product that no one wants, losing budgetary or personnel commitments.

ii) **Risk Assessment**: - The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in two ways:

1. The likelihood of risk coming true (r)
2. The consequence of the problems associated with the risk (s)

The priority of each risk can be computed as P = r * s

The most likely and damaging risks can be handled first and more comprehensive risk abatement procedures can be designed for these risks.

iii) **Risk Containment**: - After all the identified risks of a project are assessed, plans must be made to first contain the most damaging and the most likely risks. Different risks require different containment procedures. There are three main strategies to plan for risk containment:

1) Avoid the risk: - Risks can be avoided in several ways, such as discussing with the customer to change the requirement to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover.

2) Transfer the risk: - This strategy involves getting the risky component developed by a third party, buying insurance cover and so on.

3) Risk reduction: - This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned.

UNIT – 3

Software Design

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.

Characteristics of Good Design:

i) Correctness: - A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

ii) Understandability: - A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

iii) Efficiency: - A good design solution should adequately address resource, time and cost optimization issues.

iv) Maintainability: - A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Data design**,**

Architectural design**,**

Interface design,

Approaches to Software Design:

There are two fundamentally different approaches to software design that are in use today – Function-oriented design, and Object-oriented design.

1. Function Oriented Design: - The salient features of the function-oriented design are:

i) Top-down decomposition: - In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

ii) Centralized system state: - The system state can be designed as the values of certain data items that determine the response of the system to a user action or external event.

2. Object-oriented Design: - In the object-oriented design approach, a system si viewed as being made up of a collection of object. Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it.

Cohesion: -

Cohesion is a measure of the functional strength of a module. When the functions of a module cooperate with each other for performing a single objective, then the module has good cohesion, otherwise it has very poor cohesion.

Classification of Cohesiveness: - Different types of cohesion are

i) **Coincidental Cohesion**: - A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely. It means that the module contains a random collection of functions.

ii) **Functional cohesion**: - A module is said to posses functional cohesion, if different functions of the module cooperate to complete a single task. For example, a module containing all the function required to manage students' admission.

iii) **Logical Cohesion**: - A module is said to be logically cohesive, if all the elements of the module perform similar operations such as error handling data input, data output etc.

iv) **Temporal cohesion**: - when a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.

v) **Procedural cohesion**: - A module is said to posses procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.

vi) **Communicational cohesion**: - A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

vii) **Sequential cohesion**: - A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output form one function is input to the next in the sequence.

Coupling: -

Coupling between two module sis measure of the degree of interaction or independence between the two modules. Two modules are said to be highly coupled, in either of the following two situations arises:

i) If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled

ii) If the interactions occur through some shared data, then also we say that they are highly coupled.

Classification of Coupling

i) Data Coupling: - Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float or a character etc.

ii) Stamp Coupling: - Two modules are stamp coupled, if they communicate using a composite dta item such as a structure in C language.

iii) Control Coupling: - Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another.

iv) Common Coupling: - Two modules are common coupled, if they share same global data items.

v) Content Coupling: - Content coupling exists between two modules, if they share code.

Software Metrics: different types of project matrics

Software Testing and Maintenance

Testing Process

The aim of testing is to identify all defects in a program. However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free.

Design of Test Cases

When test cases are designed based on random input data, many of the test cases do not contribute to the significance of the test suite. That is, they do not help detect any additional defects not already being detected by other test cases in the suite. A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contract to testing using some random input values.

There are essentially two main approaches to systematically design test cases:

i) Black-box approach ii) White-box (or glass-box) approach.

i) Black-box approach: - In this approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input / output behavior and does not require any knowledge of the internal structure of a program.

The two main approaches of black-box approach are:

a) Equivalence Class Partitioning: - In this approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.

b) Boundary Value Analysis: - Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

ii) White-box approach: - This approach requires a thorough knowledge of the internal structure of a program. A white-box testing strategy can either be coverage-based or fault-based.

a) Fault-based testing: - A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy.

b) Coverage-based testing: - A coverage-based testing strategy attempts to execute certain elements of a program. Popular examples of coverage-based testing strategies are statements coverage, branch coverage, and path coverage-based testing.

Types of Testing

Functional Testing

Structural Testing

Test Activities

Unit Testing

Unit testing is undertaken after a module has been coded and reviewed. Before carrying out unit testing, unit test cases have to be designed and the test environment for the unit under test has to be developed.

Integration Testing

Integration testing is carried out after all the modules have been unit tested. The objective of integration testing is to detect the errors at the module interfaces that is there are no errors in parameter passing. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. Any one of the following approaches can be used to develop the test plan:

i) Big-bang approach: - In this approach, all the modules making up a system are integrated in a single step. However, this technique can meaningfully be used only for very small system.

ii) Bottom-up integration testing: - In this testing, first the modules for the each subsystems are integrated. Thus the subsystems can be integrated separately and independently.

iii) Top-down integration testing: - This testing starts with the root module are one or two subordinate modules in the system. After the top-level skeleton has been tested, the modules that are at the immediately lower layer of the skeleton are combined with it and tested.

iv) Mixed integration testing: - The mixed integration testing follows a combination of tow-down and bottom-up testing approaches. In this testing, testing can start as and when modules become available after unit testing.

System Testing

After all the units of a program have been integrated together and tested, system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

There are essentially three main kinds of system testing:

i) Alpha testing: - Alpha testing referes to the system testing carried out by the test team within the developing organization.

ii) Beta testing: - Beta testing is the system testing performed by a selected group of friendly customers.

iii) Acceptance testing: - Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

Debugging Activities

After a failure has been detected, it is necessary to first identify the program statements that are in error and are responsible for the failure, the error can then be fixed. The following are some of the approaches that are popularly adopted by the programmers for debugging:

i) **Brute force method**: - In this approach, print statements are inserted through out the program to print the intermediate values with hope that some of the printed values will help to identify the statement in error.

ii) **Backtracking**: - In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.

iii) **Cause elimination method**: - In this approach, once a failure is observed, the symptoms of the failure are noted. Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and test are conducted to eliminate each.

iv) **Program slicing**: - In this approach, program is decomposed into some part called slice. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Software Maintenance:
Management of Maintenance
Maintenance Process
Reverse Engineering
Software Re-engineering
Configuration Management
Documentation
Software quality Assurance
CASE tools:
Analysis tools,
design tools,
SQA tools,
software testing tools.