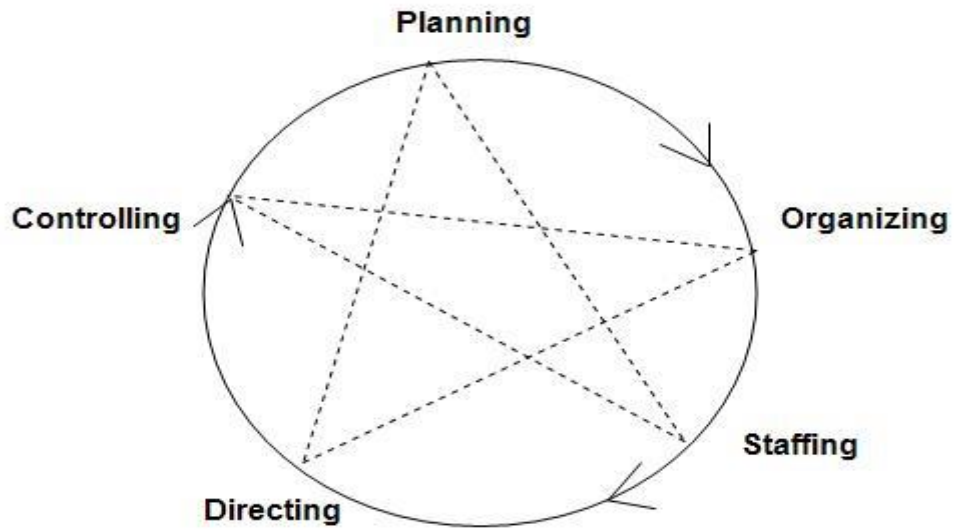
13. SOFTWARE PROJECT MANAGEMENT

A system of management procedures, practices, technologies, skills, and experience necessary to successfully manage a software project

- The main goal of software project management is to enable a group of software developers to work efficiently towards successful completion of the project.
- The entire SPM consist of five phases: Planning, Staffing, Organizing, Controlling & Directing.



PLANNING

- ✓ It is the basic function of management. It deals with chalking out a future course of action & deciding in advance the most appropriate course of actions for achievement of pre-determined goals.
- ✓ Planning is deciding in advance - what to do, when to do & how to do. It bridges the gap from where we are & where we want to be".
- ✓ A plan is a future course of actions. It is an exercise in problem solving & decision making.
- ✓ Planning is determination of courses of action to achieve desired goals. Thus, planning is a systematic thinking about ways & means for accomplishment of pre-determined goals.

Planning activities

- Set objectives and goals
- Develop strategies
- Develop policies
- Forecast future situations
- Conduct a risk assessment
- Determine possible courses of action
- Make planning decisions
- Set procedures and rules
- Develop project plans
- Prepare budgets
- Document project plans

ORGANIZING

- ✓ It is the process of bringing together physical, financial and human resources and developing productive relationship amongst them for achievement of organizational goals.
- ✓ To organize a business is to provide it with everything useful or its functioning i.e. raw material, tools, capital and personnel's.
- ✓ To organize a business involves determining & providing human and non-human resources to the organizational structure.

Organizing activities

- *Identify and group project function, activities, and tasks*
- *Select organizational structures*
- Create organizational positions
- Define responsibilities and authority
- Establish position qualifications
- Document organizational decisions

STAFFING

- ✓ It is the function of manning the organization structure and keeping it manned.
- ✓ Staffing has assumed greater importance in the recent years due to advancement of technology, increase in size of business, complexity of human behavior etc.
- ✓ The main purpose of staffing is to put right man on right job.
- ✓ Managerial function of staffing involves manning the organization structure through proper and effective selection, appraisal & development of personnel to fill the roles designed under the structure.

Staffing activities

- Fill organizational positions
- Assimilate newly assigned personnel
- Educate or train personnel
- Provide for general development
- Evaluate and appraise personnel
- Compensate
- Terminate assignments
- Document staffing decisions

DIRECTING

- ✓ It is that part of managerial function which actuates the organizational methods to work efficiently for achievement of organizational purposes.
- ✓ It is considered life-spark of the enterprise which sets it in motion the action of people because planning, organizing and staffing are the mere preparations for doing the work.
- ✓ Direction is that inert-personnel aspect of management which deals directly with influencing, guiding, supervising, motivating sub-ordinate for the achievement of organizational goals.
- ✓ Direction has following elements:

Supervision- implies overseeing the work of subordinates by their superiors. It is the act of watching & directing work & workers.

Motivation- means inspiring, stimulating or encouraging the sub-ordinates with zeal to work. Positive, negative, monetary, non-monetary incentives may be used for this purpose.

Leadership- may be defined as a process by which manager guides and influences the work of subordinates in desired direction.

Communications- is the process of passing information, experience, opinion etc from one person to another. It is a bridge of understanding.

Directing activities

- *Provide leadership*
- *Supervise personnel*
- *Delegate authority*
- *Motivate personnel*
- Build teams
- Coordinate activities
- Facilitate communication
- Resolve conflicts
- Manage changes
- Document directing decisions

CONTROLLING

- ✓ It implies measurement of accomplishment against the standards and correction of deviation if any to ensure achievement of organizational goals.
- ✓ The purpose of controlling is to ensure that everything occurs in conformities with the standards. An efficient system of control helps to predict deviations before they actually occur.
- ✓ Controlling is the process of checking whether or not proper progress is being made towards the objectives and goals and acting if necessary, to correct any deviation
- ✓ Controlling is the measurement & correction of performance activities of subordinates in order to make sure that the enterprise objectives and plans desired to obtain them as being accomplished.

Controlling activities

- Develop standards of performance
- Establish monitoring and reporting systems
- Measure and analyze results
- Initiate corrective actions
- Reward and discipline
- Document controlling methods

ORGANIZATION STRUCTURES

- ✓ Usually every software development organization handles several projects at any time. Software organizations assign different teams of engineers to handle different software projects.
- ✓ There are essentially two broad ways in which a software development organization can be structured: functional format and project format.

PROJECT FORMAT

- ➢ In the project format, the project development staff are divided based on the project for which they work as shown below.

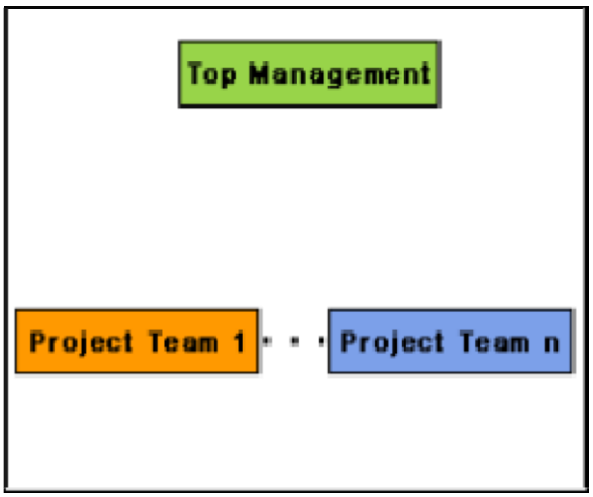


Figure: Project Organization

- ➢ In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities.
- ➢ A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and are under tremendous pressure in the later phase of the development.
- ➢ The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. So a team member can fill any slots in the development team.
- ➢ In project format, each individual team consist of a representative to do various SDLC phases like analysis, design, coding, testing, architecture design, reviewer etc.
- ➢ Though each member is aware about various SDLC activities, it is easy to rotate the roles of team members.
- ➢ The team members can improve their knowledge and skills in every domains of SDLC.

FUNCTIONAL FORMAT

- In the functional format, the development staff are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.

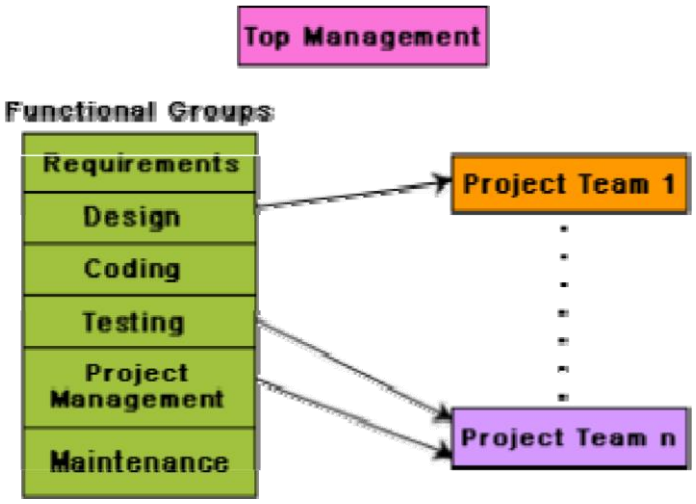


Figure: Functional Organization

- In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.
- The main advantages of a functional organization are:

  - Ease of staffing
  - Production of good quality documents
  - Job specialization
  - Efficient handling of the problems associated with manpower turnover.

- The functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.
- The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work.

- It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization also provides an efficient solution to the staffing problem.
- The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organization.
- Functional format is not suitable for small organizations handling just one or two projects.
- Another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects.

TEAM STRUCTURE

- Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized.
- There are mainly three formal team structures: chief programmer, democratic, and the mixed team organization.

1. Chief Programmer Team

- In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members.
- The structure of the chief programmer team is shown below.

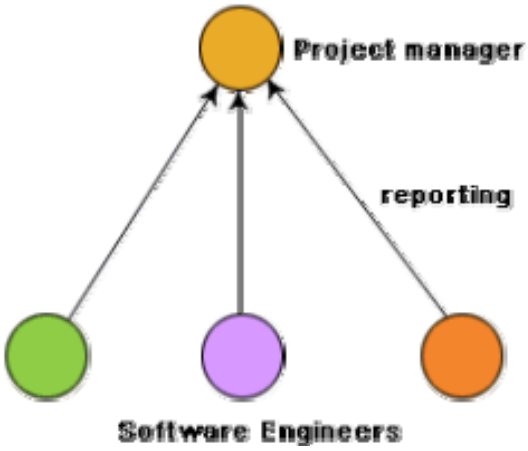


Figure: Chief programmer team structure

- The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking.
- The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

- The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution.
- For simple and well-understood problems, an organization must be selective in adopting the chief programmer structure.
- The chief programmer team structure should not be used unless the importance of early project completion outweighs other factors such as team morale, personal developments, life-cycle cost etc.

2. Democratic Team

- The democratic team structure, as the name implies, does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

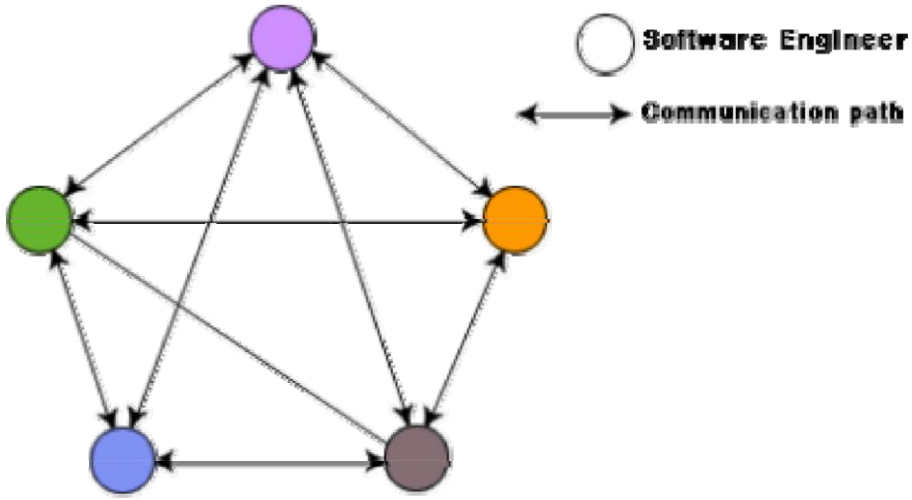


Figure: Democratic team structure

- The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover.
- Democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team.
- A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic.
- The democratic team organization encourages egoless programming as programmers can share and review one another's work.

3. Mixed Control Team Organization

- The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization.
- The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior engineer's level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs.
- This team structure is extremely popular and is being used in many software development companies.
- The mixed control team organization is shown below. This team organization incorporates both hierarchical reporting and democratic set up. In figure, the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows.

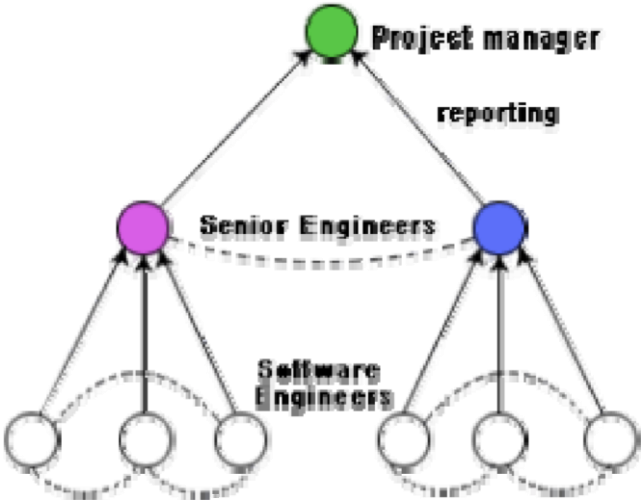


Figure: Mixed team structure

CHARACTERISTICS OF A GOOD SOFTWARE ENGINEER

The attributes that good software engineers should possess are as follows:

- Exposure to systematic techniques, i.e. familiarity with software engineering principles.
- Good technical knowledge of the project areas (Domain knowledge).
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation.
- Sound knowledge of fundamentals of computer science.
- Intelligence.
- Ability to work in a team.
- Discipline, etc.

14. SOFTWARE PROJECT COST ESTIMATION & PROJECT SCHEDULING

COST ESTIMATION

Cost estimation can be defined as the approximate judgement of the costs for a project.

- Cost estimation is usually measured in terms of effort. The most common metric used is person months or years (or man months or years). The effort is the amount of time for one person to work for a certain period of time.
- A cost estimate done at the beginning of a project will help determine which features can be included within the resource constraints of the project (e.g., time). Requirements can be prioritized to ensure that the most important features are included in the product. The risk of a project is reduced when the most important features are included at the beginning because the complexity of a project increases with its size, which means there is more opportunity for mistakes as development progresses. Thus, cost estimation can have a big impact on the life cycle and schedule for a project.
- Cost estimation can also have an important effect on resource allocation. It is prudent for a company to allocate better resources, such as more experienced personnel, to costly projects.

Metrics for software project size estimation

- Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed.
- Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

1. **Lines of Code (LOC)**
   - LOC is the simplest and most widely used metric to estimate project size. The project size is estimated by counting the number of source instructions in the developed program. Lines used for commenting the code and the header lines should be ignored.
   - Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually 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. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

LOC as a measure of problem size has several shortcomings:

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program.

- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.

- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic
- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed.

2. **Function point (FP)**

- Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. 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. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.
- The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data.
- For example, the issue book feature (as shown in below figure) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.

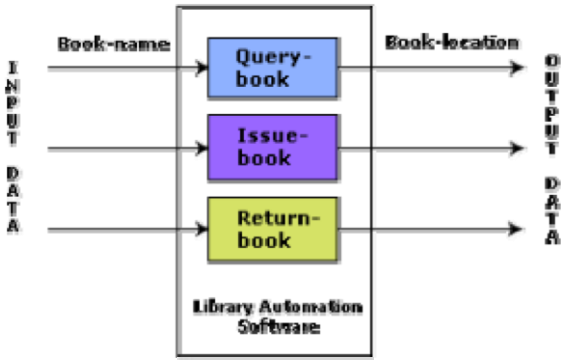


Figure: System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. 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

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance. Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input.

For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the un-adjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. 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 from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP*TCF.

3. **Feature point metric**
   - A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of

developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

- ❖ Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

COCOMO (Constructive Cost Model)

- ➢ COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. COCOMO is used to estimate the total effort required to develop a software project.
- ➢ COCOMO divides the software or projects into three categories.

1. **Organic:** A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
2. **Semidetached:** A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
3. **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.
   - ➢ According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

$$\textbf{Effort} = c1 \times (\text{KLOC})^{c2} \ \textbf{PM}$$

$$\textbf{Tdev} = c3 \times (\text{Effort})^{c4} \ \textbf{Months}$$

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- c1,c2,c3,c4 are constants for each category of software products,
- Tdev is the estimated time to develop the software, expressed in months,
- Effort is the total effort required to develop the software product, expressed in person months (PMs).

Software category	c1	c2	c3	c4
Organic	2.4	1.05	2.5	0.38
Semi- Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table: Estimated value of constants for various software categories

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in below figure). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve.

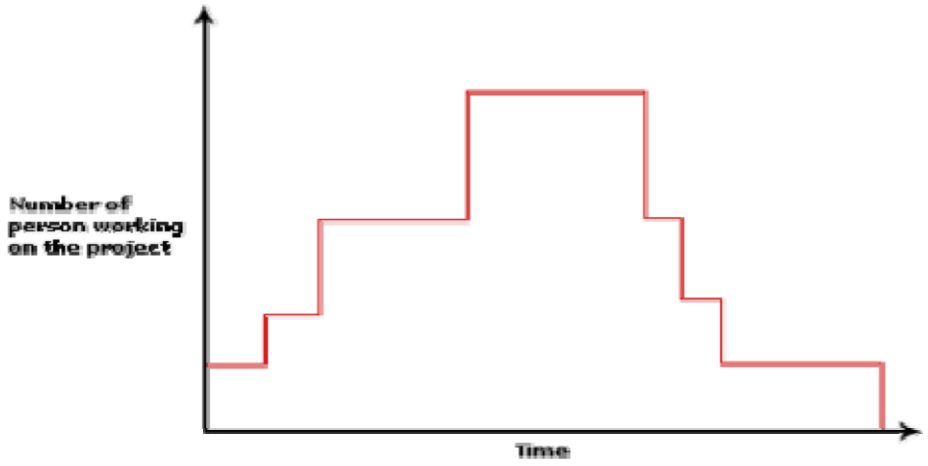


Figure: Person-month curve

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : **Effort = 2.4(*KLOC*)$^{1.05}$ PM**

Semi- detached : **Effort = 3.0(*KLOC*)$^{1.12}$ PM**

Embedded : **Effort = 3.6(*KLOC*)$^{1.20}$ PM**

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : **Tdev = 2.5(*Effort*)$^{0.38}$ Months**

Semi- detached : **Tdev = 2.5(*Effort*)$^{0.35}$ Months**

Embedded : **Tdev = 2.5(*Effort*)$^{0.32}$ Months**

The effort is somewhat super-linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size as shown in the graph.

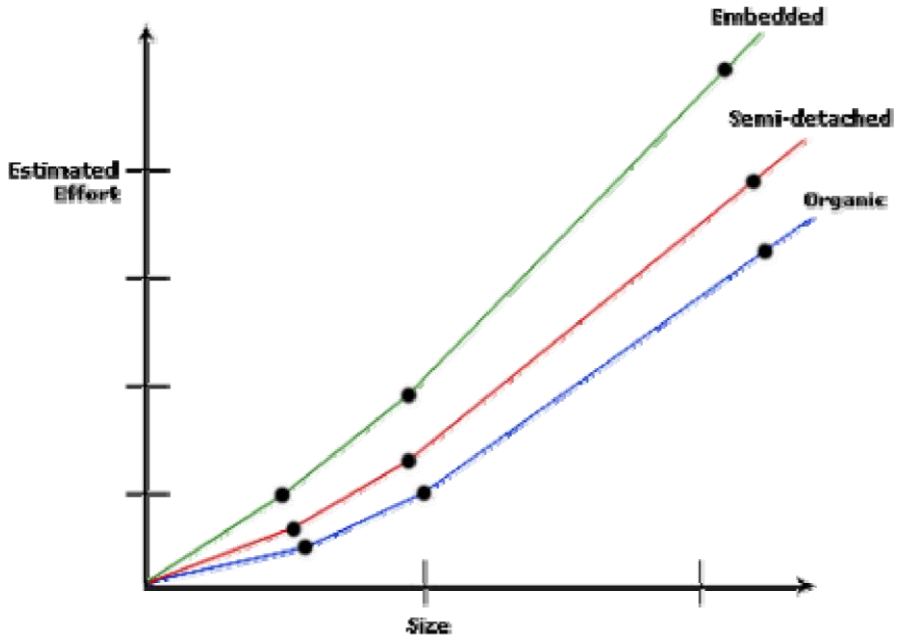


Figure Effort versus product size

The development time is a sub-linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. From the graph below, it can be observed that the development time is roughly the same for all the three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

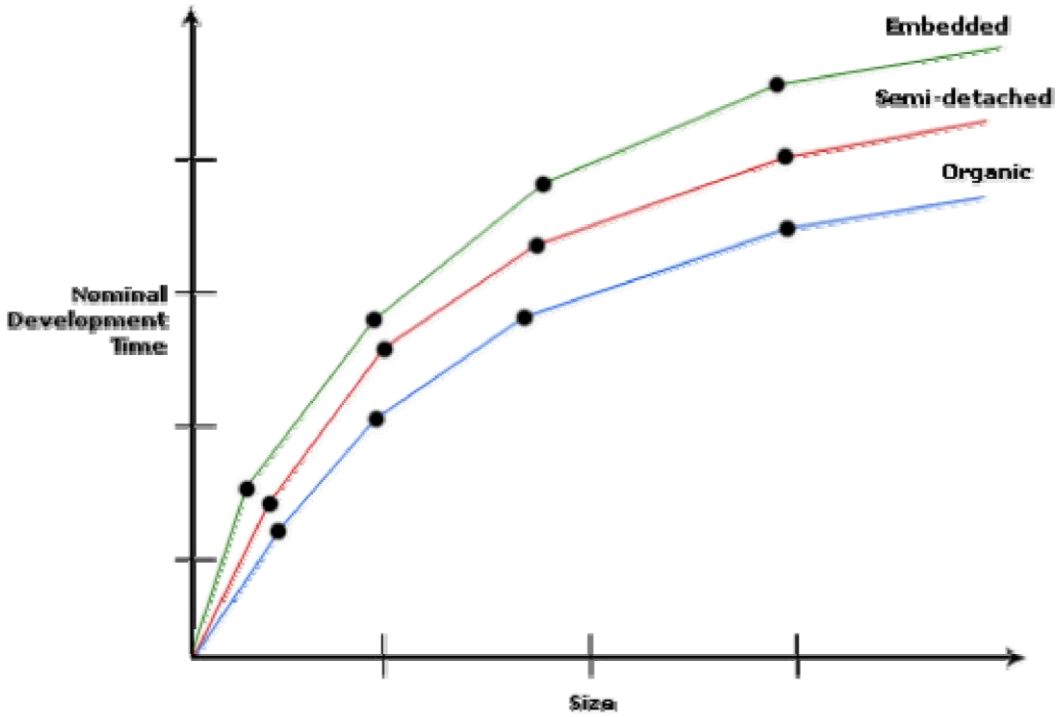


Fig. 11.5: Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

STEPS TO ESTIMATE COST IN BASIC COCOMO

1. Identify the category to which the software belong to.
2. Estimate the Lines of Code(LOC) and convert it into KLOC
3. Estimate the effort using the appropriate equation constants c1,c2.
4. Estimate the time for development using the computed effort value.
5. Then estimate the development cost

Example:

Assume that the size of an organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05} = 91$ PM**

Nominal development time = **$2.5 \times (91)^{0.38} = 14$ months**

Cost required to develop the product = **$14 \times 15{,}000$**

= **Rs. 210,000/-**

INTERMEDIATE COCOMO

- Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

- Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers" each with a number of subsidiary attributes:-

1. **Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.
2. **Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required, memory constraints etc.
3. **Personnel:** The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.
4. **Development Environment:** Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

COMPLETE COCOMO

- A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These subsystems may have widely different characteristics.
- The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.
- The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

- Database part
- Graphical User Interface (GUI) part
- Communication part

➢ Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

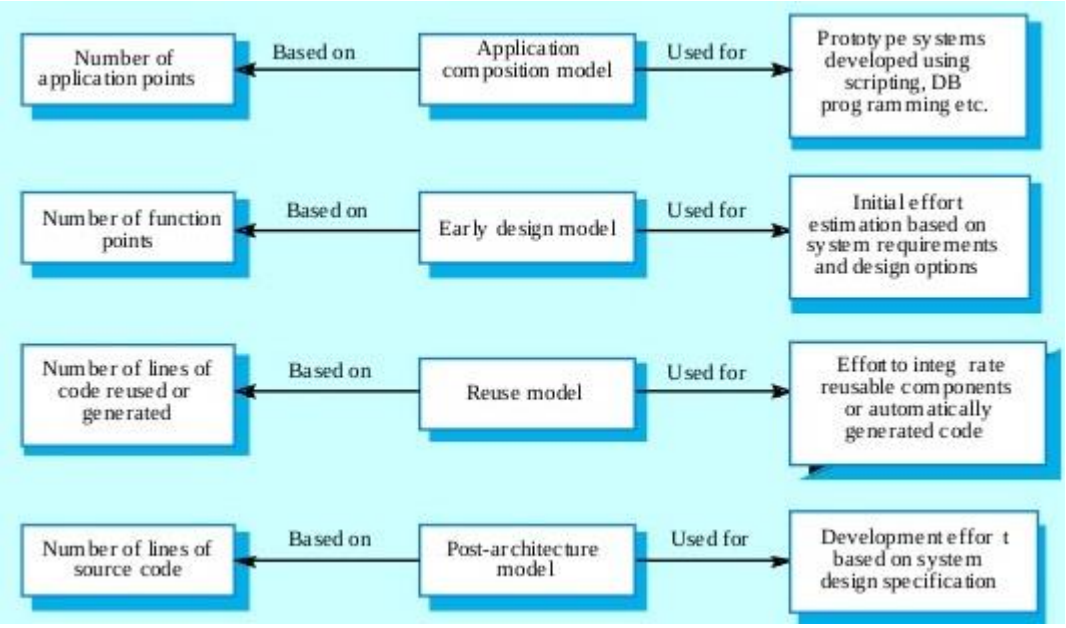
COCOMO – 2

➢ Basic COCOMO is not suitable for projects of larger size as well as projects which uses reuse – approach. COCOMO – 2 deals with the above cases and is most suitable for projects developed in Third Generation Languages (3GL).

➢ COCOMO – 2 is used to estimate project costs at different phases of the software. As the project progresses, though these models can be applied at different stages of the same project.

➢ The various models in COCOMO – 2 are

1. **Application composition:** Used to estimate cost for prototyping.
2. **Early design:** Used to estimate cost at the architectural design stage.
3. **Post –Architecture :** Used to estimate cost during detailed design and coding stage.
4. **Reuse model:** Used to estimate cost, if code reuse is adapted.



➢ **STEPS TO ESTIMATE EFFORT IN APPLICATION COMPOSTION**

1. Estimate the number of screens, reports and 3GL components from an analysis of the SRS document.
2. Determine the complexity level of each screen and report, and rate these as either simple, medium or difficult. The complexity of a screen or report is determined by the number of tables and views it contains.

Table for Screen complexity assignment			
Number of views	**Tables <4**	**Tables<8**	**Tables>=8**
<3	Simple	Simple	Medium
3-7	Simple	Medium	Difficult
>8	Medium	Difficult	Difficult
Table for Report complexity assignment			
Number of sections	**Tables <4**	**Tables<8**	**Tables>=8**
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 or more	Medium	Difficult	Difficult

3. Estimate the complexity and find out the equivalent weight value using the below table. Weights are the amount of effort required to implement an instance of an object at the assigned complexity class.

Table for complexity weights for each class of objects			
Object type	**Simple**	**Medium**	**Difficult**
Screen	1	2	3
Report	2	5	8
3GL components	--	--	10

4. Determine the number of object points (OP). The object point count is the sum of all the assigned complexity values for the object instances together.
5. Estimate the expected percentage of reuse in the system. Then evaluate the number of New object point count (NOP).

NOP= ((Object-Points)*(100-% of reuse)) / 100

6. Determine the productivity rate, PROD = NOP / person – month based on CASE maturity value.
7. Finally Effort is computed as E = NOP / PROD.

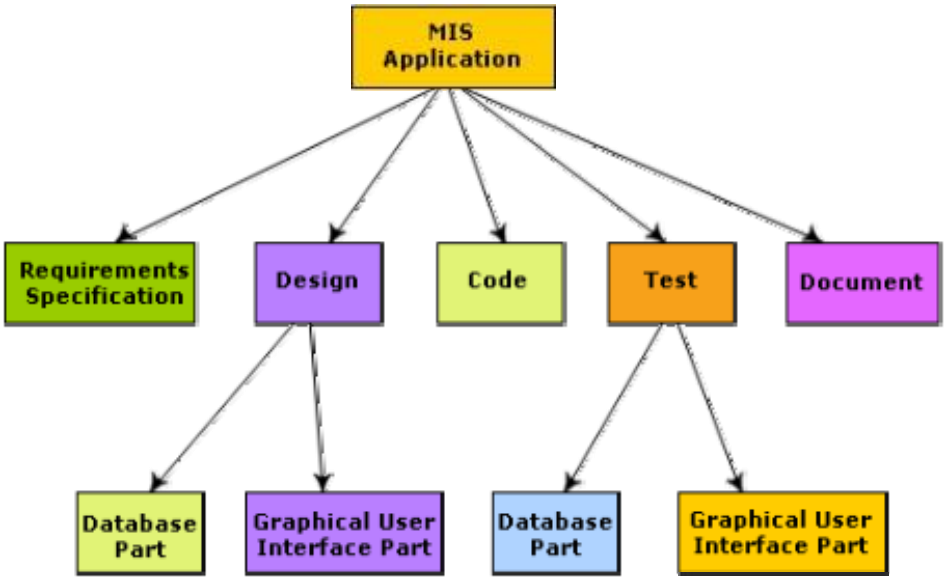
SOFTWARE PROJECT SCHEDULING

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

WORK BREAK DOWN STRUCTURE

- Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem.
- The root of the tree is labelled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. The following figure represents the WBS of an MIS (Management Information System) software.



GANTT CHART

- The simplest project management tool used to represent the timeline of activities is the Gantt chart. Henry L. Gantt lent his name to a simple and very useful graphical representation of a project development schedule. The Gantt chart shows almost all of the information contained in the schedule activity list, but in a much more digestible way. The schedule information is more easily grasped and understood, and the activities can be easily compared.
- The Gantt chart enables us to see at any given time, which activities should be occurring in the project. A Gantt chart has horizontal bars plotted on a chart to represent a schedule.
- In a Gantt chart, Time is plotted on the horizontal axis and activities on the vertical axis. An activity is represented by a horizontal bar on the Gantt chart. The position of a horizontal bar shows the start and end time of an activity and the length of the bar show its duration. Gantt chart can be used to analyse the progress of the project.

ID	Task Name	Expected Start	Expected Finish	Duration
1	Requirement Gathering & analysis	02-11-2015	12-11-2015	1w 4d
2	Architectural and Detailed design	13-11-2015	30-11-2015	2w 2d
3	Database & GUI Design	18-11-2015	23-11-2015	4d
4	Module 1 implementation	01-12-2015	15-12-2015	2w 1d
5	Module 2 implementation	02-12-2015	30-12-2015	4w 1d
6	Unit Testing	02-12-2015	30-12-2015	4w 1d
7	Integration and System Testing	31-12-2015	18-01-2016	2w 3d
8	Documentation	02-11-2015	25-01-2016	12w 1d

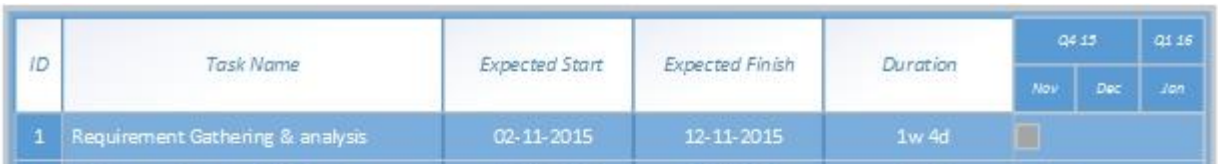


Figure: GANTT chart

PERT CHART

- ✓ PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies.
- ✓ PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made.
- ✓ Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex.
- ✓ A critical path in a PERT chart is shown by using thicker arrows.
- ✓ Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different engineers.

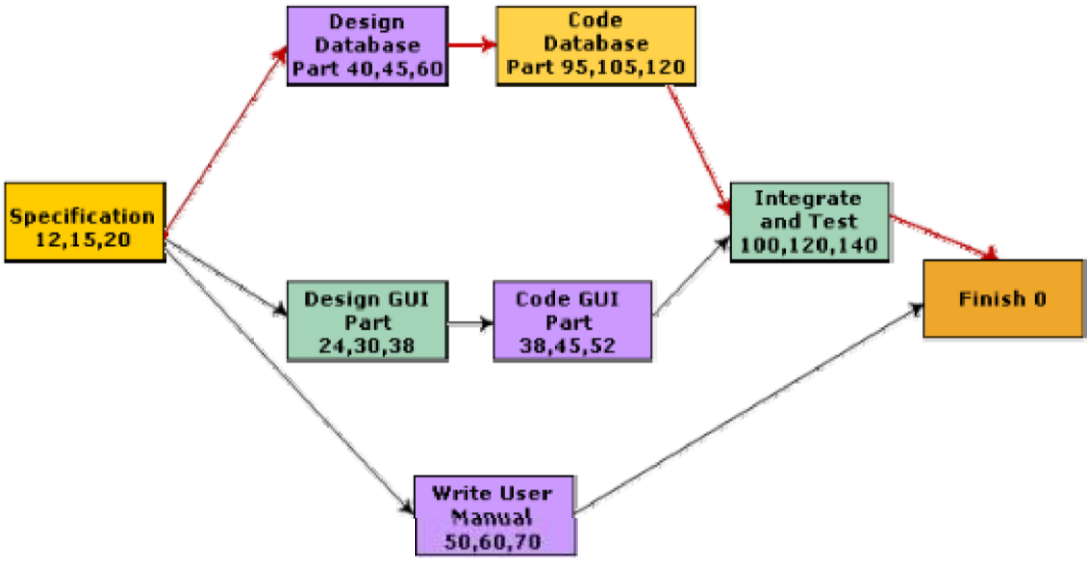


Figure: PERT chart representation of the MIS problem

15. CASE

CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.

Benefits of CASE tools

- Reduces the software development time and cost by automating many repetitive manual tasks.
- Helps to create good quality documentation and thus provides a better quality product.
- Helps to create more maintainable software systems.
- Reduces the burden of software engineer.
- Provides more structured and ordered development methodology.

Characteristics of a successful CASE tool

- It should support standard software development methodology and modelling techniques.
- It must provide an integrated environment for software development.
- It must be flexible, so that user can make necessary changes.
- It should support reverse engineering process.
- It must support integration with automated testing tools.
- It must provide online help.

CASE CLASSIFICATIONS

- **Upper Case Tools** – Tools that mainly concentrate on high level activities of SDLC. Upper CASE tools are used in planning, analysis and design stages of SDLC.

- **Lower Case Tools** – tools mainly focuses on the implementation of the system. Lower CASE tools are used in implementation, testing and maintenance.

- **Integrated Case Tools** - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.