

# Software Project Planning

# Project Planning and Project Estimation Techniques

## Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Identify the job responsibilities of a software project manager.
- Identify the necessary skills required in order to perform software project management.
- Identify the essential activities of project planning.
- Determine the different project related estimates performed by a project manager and suitably order those estimates.
- Explain what is meant by Sliding Window Planning.
- Explain what is Software Project Management Plan (SPMP).
- Identify and explain two metrics for software project size estimation.
- Identify the shortcomings of function point (FP) metric.
- Explain the necessity of feature point metric in the context of project size estimation.
- Identify the types of project-parameter estimation technique.

## Responsibilities of a software project manager

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

## Skills necessary for software project management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and

controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

## Project planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

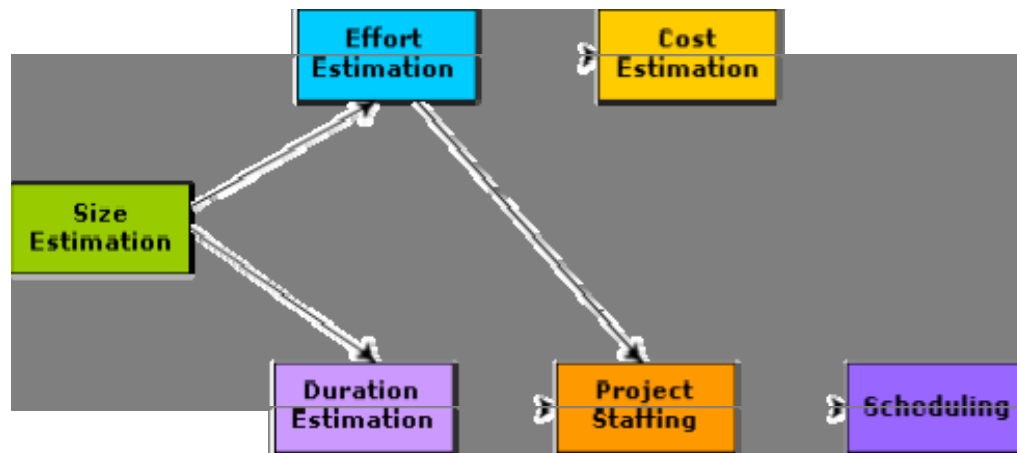
- Estimating the following attributes of the project:
  - Project size:** What will be problem complexity in terms of the effort and time required to develop the product?
  - Cost:** How much is it going to cost to develop the project?
  - Duration:** How long is it going to take to complete development?
  - Effort:** How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

- Scheduling manpower and other resources
- Staff organization and staffing plans
- Risk identification, analysis, and abatement planning
- Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

## Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Fig. 11.1 shows the order in which important project planning activities may be undertaken. From fig. 11.1 it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.



**Fig. 11.1:** Precedence ordering among planning activities

## Sliding Window Planning

Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

## Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document.

Organization of the Software Project Management Plan (SPMP) Document

## **1. Introduction**

- (a) Objectives
- (b) Major Functions
- (c) Performance Issues
- (d) Management and Technical Constraints

## **2. Project Estimates**

- (a) Historical Data Used
- (b) Estimation Techniques Used
- (c) Effort, Resource, Cost, and Project Duration Estimates

## **3. Schedule**

- (a) Work Breakdown Structure
- (b) Task Network Representation
- (c) Gantt Chart Representation
- (d) PERT Chart Representation

## **4. Project Resources**

- (a) People
- (b) Hardware and Software
- (c) Special Resources

## **5. Staff Organization**

- (a) Team Structure
- (b) Management Reporting

## **6. Risk Management Plan**

- (a) Risk Analysis
- (b) Risk Identification
- (c) Risk Estimation
- (d) Risk Abatement Procedures

## **7. Project Tracking and Control Plan**

## **8. Miscellaneous Plans**

- (a) Process Tailoring
- (b) Quality Assurance Plan
- (c) Configuration Management Plan

- (d) Validation and Verification
- (e) System Testing Plan
- (f) Delivery, Installation, and Maintenance Plan

## 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. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

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.

### Lines of Code (LOC)

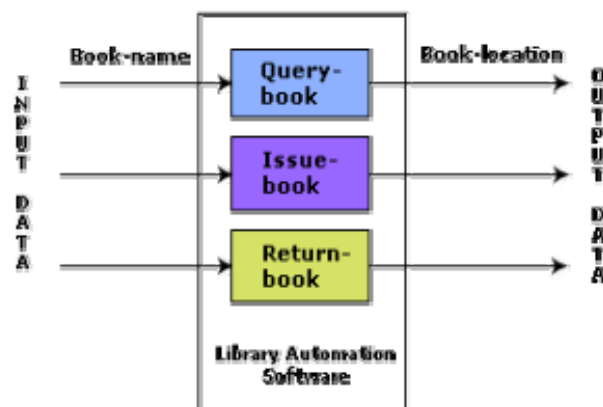
LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, 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 submodules 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.

### Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. 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 fig. 11.2) 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.



**Fig. 11.2:** 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).

$$\text{UFP} = (\text{Number of inputs}) \times 4 + (\text{Number of outputs}) \times 5 + \\ (\text{Number of inquiries}) \times 4 + (\text{Number of files}) \times 10 + \\ (\text{Number of interfaces}) \times 10$$

**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 unadjusted 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  $(0.65 + 0.01 \cdot DI)$ . As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally,  $FP = UFP \cdot TCF$ .

## Shortcomings of function point (FP) metric

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. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.

- 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. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.
- 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. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
- 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. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
- 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. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

## 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.

## Project Estimation techniques

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

### Empirical Estimation Techniques

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: Expert judgment technique and Delphi cost estimation.

#### Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert

estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

### Delphi cost estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

### Heuristic Techniques

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the

mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

$$\text{Estimated Parameter} = c_1 * e^{d_1}$$

In the above expression,  $e$  is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc.  $c_1$  and  $d_1$  are constants. The values of the constants  $c_1$  and  $d_1$  are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated Resource} = c_1 * e_1^{d_1} + c_2 * e_2^{d_2} + \dots$$

Where  $e_1, e_2, \dots$  are the basic (independent) characteristics of the software already estimated, and  $c_1, c_2, d_1, d_2, \dots$  are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants  $c_1, c_2, d_1, d_2, \dots$ . Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

### **Analytical Estimation Techniques**

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead's software science is an example of an analytical technique. Halstead's software science can be used to derive some interesting results starting with a few simple assumptions. Halstead's software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

## Halstead's Software Science – An Analytical Technique

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for over all program length, potential minimum value, actual volume, effort, and development time.

For a given program, let:

- $\eta_1$  be the number of unique operators used in the program,
- $\eta_2$  be the number of unique operands used in the program,
- $N_1$  be the total number of operators used in the program,
- $N_2$  be the total number of operands used in the program.

### Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length  $N = N_1 + N_2$ . Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, *program vocabulary*  $\eta = \eta_1 + \eta_2$ .

### Program Volume

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

$$V = N \log_2 \eta$$

Here the program volume  $V$  is the minimum number of bits needed to encode the program. In fact, to represent  $\eta$  different identifiers uniquely, at least  $\log_2 \eta$  bits (where  $\eta$  is the program vocabulary) will be needed. In this scheme,  $N \log_2 \eta$  bits will be needed to store a program of length  $N$ . Therefore, the volume  $V$  represents the size of the program by approximately compensating for the effect of the programming language used.

## Potential Minimum Volume

The potential minimum volume  $V^*$  is defined as the volume of most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction., say a function call like `foo( )` ;. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands.

Thus, if an algorithm operates on input and output data  $d_1, d_2, \dots d_n$ , the most succinct program would be  $f(d_1, d_2, \dots d_n)$ ; for which  $\eta_1 = 2, \eta_2 = n$ . Therefore,  $V^* = (2 + \eta_2)\log_2(2 + \eta_2)$ .

The program level  $L$  is given by  $L = V^*/V$ . The concept of program level  $L$  is introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

## Effort and Time

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort  $E = V/L$ , where  $E$  is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort  $E = V^2/V^*$  (since  $L = V^*/V$ ) varies as the square of the volume. Experience shows that  $E$  is well correlated to the effort needed for maintenance of an existing program.

The programmer's time  $T = E/S$ , where  $S$  the speed of mental discriminations. The value of  $S$  has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

## Length Estimation

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity. His method is summarized below.

Halstead assumed that it is quite unlikely that a program has several identical parts – in formal language terminology identical

substrings – of length greater than  $\eta$  ( $\eta$  being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is made into a procedure or a function. Thus, it can be assumed that any program of length  $N$  consists of  $N/\eta$  unique strings of length  $\eta$ . Now, it is standard combinatorial result that for any given alphabet of size  $K$ , there are exactly  $K^r$  different strings of length  $r$ .

Thus.

$$N/\eta \leq \eta^n \quad \text{Or, } N \leq \eta^{n+1}$$

Since operators and operands usually alternate in a program, the upper bound can be further refined into  $N \leq \eta \eta_1^{n_1} \eta_2^{n_2}$ . Also,  $N$  must include not only the ordered set of  $n$  elements, but it should also include all possible subsets of that ordered sets, i.e. the power set of  $N$  strings (This particular reasoning of Halstead is not very convincing!!!).

Therefore,

$$2^N = \eta \eta_1^{n_1} \eta_2^{n_2}$$

Or, taking logarithm on both sides,

$$N = \log_2 \eta + \log_2 (\eta_1^{n_1} \eta_2^{n_2})$$

So we get,

$$N = \log_2 (\eta_1^{n_1} \eta_2^{n_2})$$

(approximately, by ignoring  $\log_2 \eta$ )

Or,

$$\begin{aligned} N &= \log_2 \eta_1^{n_1} + \log_2 \eta_2^{n_2} \\ &= n_1 \log_2 \eta_1 + n_2 \log_2 \eta_2 \end{aligned}$$

Experimental evidence gathered from the analysis of larger number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs when considered individually.

In conclusion, Halstead's theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead's software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.



**Example:**

Let us consider the following C program:

```
main( )
{
    int a, b, c, avg;

    scanf("%d %d %d", &a, &b, &c);
    avg = (a+b+c)/3;
    printf("avg = %d", avg);
}
```

The unique operators are:

**main,(), {}, int, scanf, &, " ", ":", " ", =, +, /, printf**

The unique operands are:

**a, b, c, &a, &b, &c, a+b+c, avg, 3,  
"%d %d %d", "avg = %d"**

Therefore,

$$\eta_1 = 12, \eta_2 = 11$$

$$\begin{aligned} \text{Estimated Length} &= (12 \cdot \log 12 + 11 \cdot \log 11) \\ &= (12 \cdot 3.58 + 11 \cdot 3.45) \\ &= (43 + 38) = 81 \end{aligned}$$

$$\begin{aligned} \text{Volume} &= \text{Length} \cdot \log(23) \\ &= 81 \cdot 4.52 \\ &= 366 \end{aligned}$$

# COCOMO Model

## Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Differentiate among organic, semidetached and embedded software projects.
- Explain basic COCOMO.
- Differentiate between basic COCOMO model and intermediate COCOMO model.
- Explain the complete COCOMO model.

## Organic, Semidetached and Embedded software projects

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm's [1981] definition of organic, semidetached, and embedded systems are elaborated below.

**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.

**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.

**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.

## COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. 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:

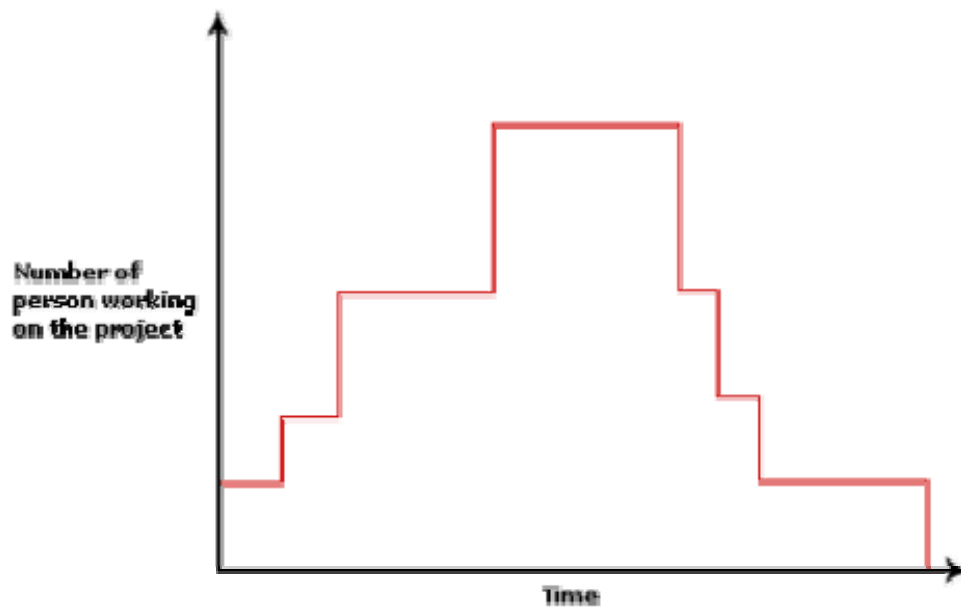
$$\text{Effort} = a_1 \times (\text{KLOC})^{a_2} \text{ PM}$$

$$\text{Tdev} = b_1 \times (\text{Effort})^{b_2} \text{ Months}$$

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- $a_1$ ,  $a_2$ ,  $b_1$ ,  $b_2$  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).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig. 11.3). 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 (as shown in fig. 11.3).



**Fig. 11.3:** Person-month curve

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say  $n$  lines), it is considered to be  $n$ LOC. The values of  $a_1$ ,  $a_2$ ,  $b_1$ ,  $b_2$  for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

### 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:

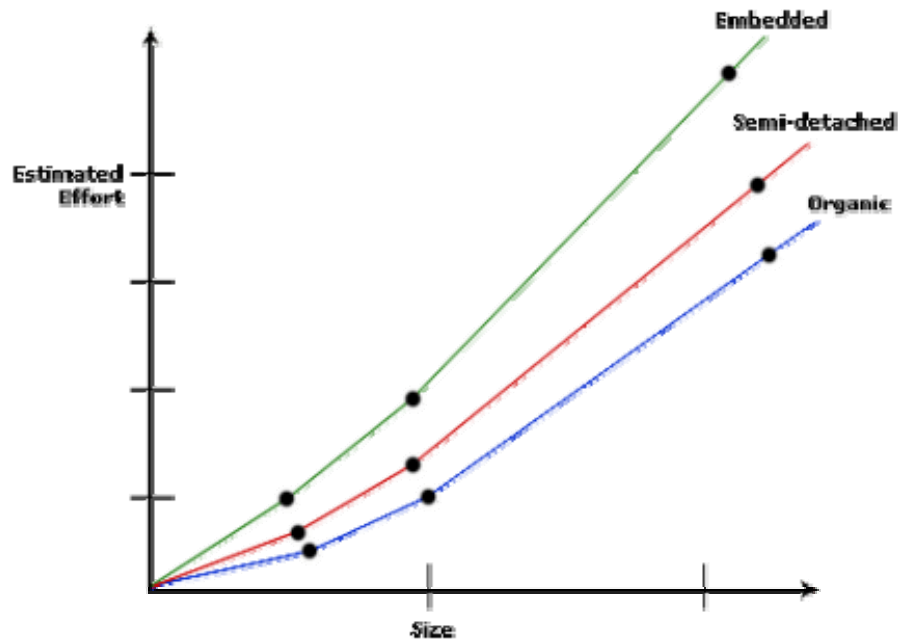
Organic	: <b>Effort = <math>2.4(KLOC)^{1.05}</math></b>	<b>PM</b>
Semi-detached	: <b>Effort = <math>3.0(KLOC)^{1.12}</math></b>	<b>PM</b>
Embedded	: <b>Effort = <math>3.6(KLOC)^{1.20}</math></b>	<b>PM</b>

### 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:

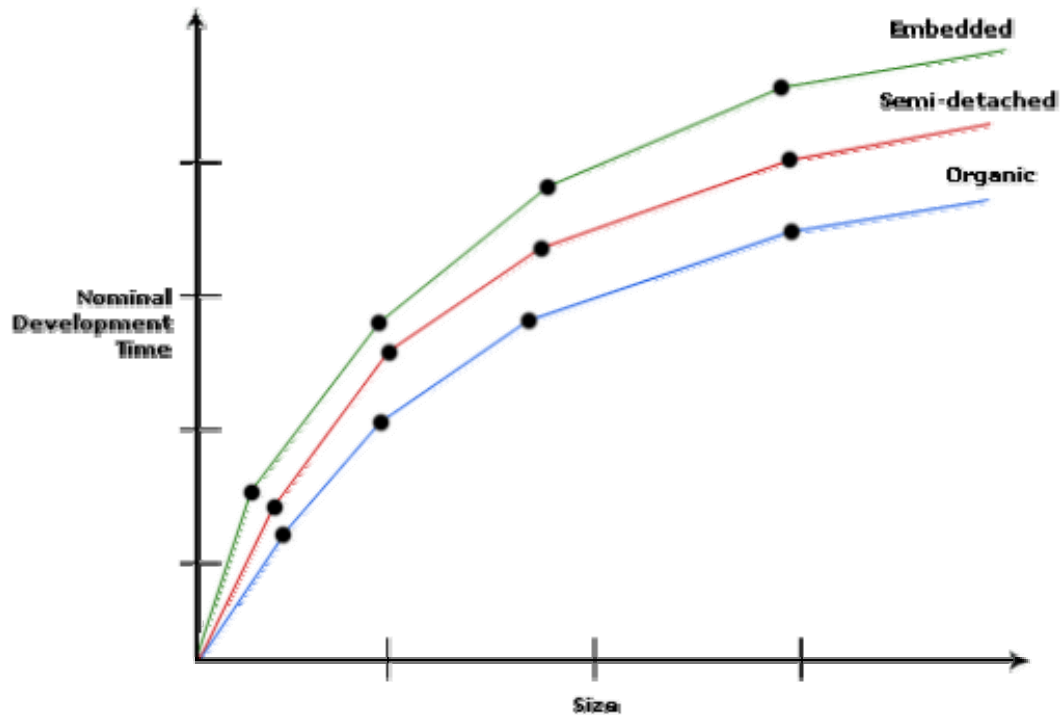
Organic	: <b>Tdev = <math>2.5(Effort)^{0.38}</math></b>	<b>Months</b>
Semi-detached	: <b>Tdev = <math>2.5(Effort)^{0.35}</math></b>	<b>Months</b>
Embedded	: <b>Tdev = <math>2.5(Effort)^{0.32}</math></b>	<b>Months</b>

some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 11.4 shows a plot of estimated effort versus product size. From fig. 11.4, we can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.



**Fig. 11.4:** Effort versus product size

The development time versus the product size in KLOC is plotted in fig. 11.5. From fig. 11.5, it can be observed that the development time is a sublinear 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. Further, from fig. 11.5, 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.

### 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:

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

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

$$\begin{aligned}\text{Cost required to develop the product} &= 14 \times 15,000 \\ &= \text{Rs. } 210,000/-\end{aligned}$$

## Intermediate COCOMO model

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

**Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

**Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required etc.

**Personnel:** The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.

**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 model

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 sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems



may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. 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.

# Staffing Level Estimation and Scheduling

## Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Identify why careful planning of staffing pattern for a project is so important.
- Determine numerically how change of project duration affects the overall effort and cost.
- Identify five necessary tasks taken by a project manager in order to perform project scheduling.
- Explain the usefulness of work breakdown structure.
- Explain activity networks and critical path method.
- Develop the Gantt chart for a project.
- Develop PERT chart for a project.

## Staffing level estimation

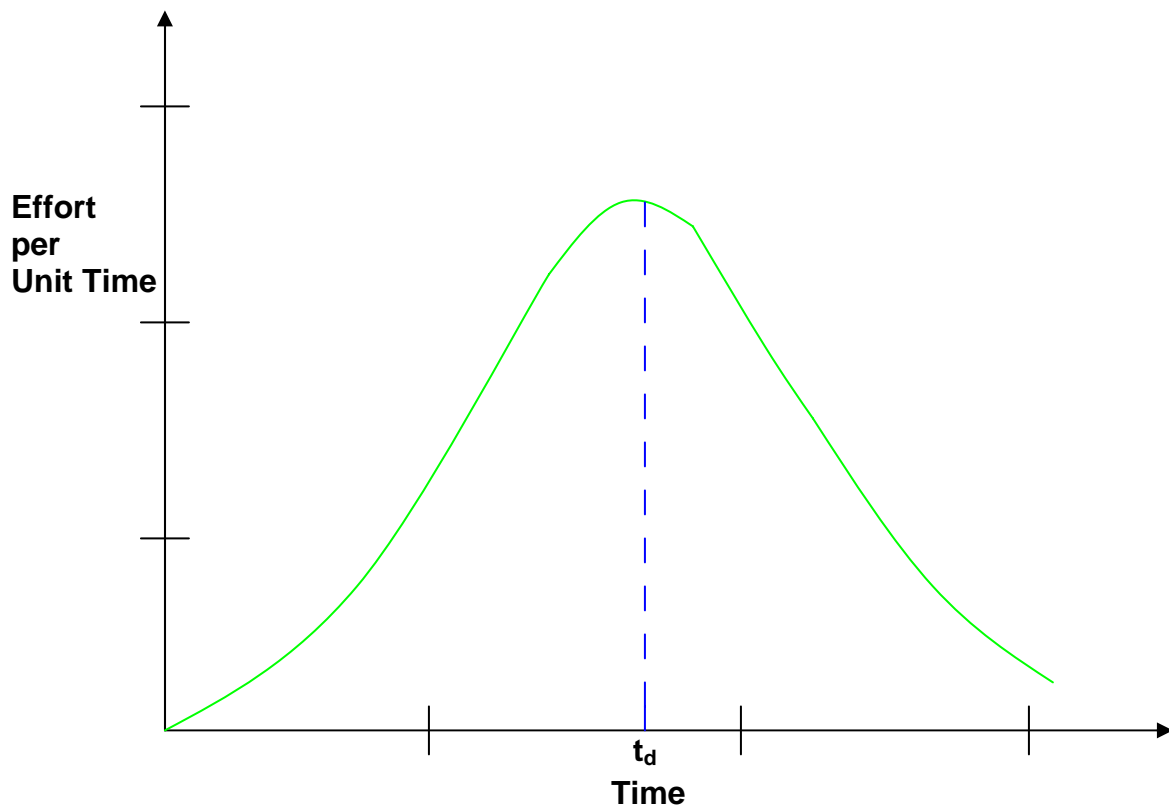
Once the effort required to develop a software has been determined, it is necessary to determine the staffing requirement for the project. Putnam first studied the problem of what should be a proper staffing pattern for software projects. He extended the work of Norden who had earlier investigated the staffing pattern of research and development (R&D) type of projects. In order to appreciate the staffing pattern of software projects, Norden's and Putnam's results must be understood.

### Norden's Work

Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve (as shown in fig. 11.6). Norden represented the Rayleigh curve by the following equation:

$$E = K/t_d^2 * t * e^{-t^2 / 2 t_d^2}$$

Where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and  $t_d$  is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects.



**Fig. 11.6:** Rayleigh curve

### Putnam's Work

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.
- $t_d$  corresponds to the time of system and integration testing. Therefore,  $t_d$  can be approximately considered as the time required to develop the software.

- $C_k$  is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of  $C_k = 2$  for poor development environment (no methodology, poor documentation, and review, etc.),  $C_k = 8$  for good software development environment (software engineering principles are adhered to),  $C_k = 11$  for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of  $C_k$  for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve. Experience shows that a very rapid build up of project staff any time during the project development correlates with schedule slippage.

It should be clear that a constant level of manpower through out the project duration would lead to wastage of effort and increase the time and effort required to develop the product. If a constant number of engineers are used over all the phases of a project, some phases would be overstaffed and the other phases would be understaffed causing inefficient use of manpower, leading to schedule slippage and increase in cost.

## Effect of schedule change on cost

By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

Where,  $K$  is the total effort expended (in PM) in the product development and  $L$  is the product size in KLOC,  $t_d$  corresponds to the time of system and integration testing and  $C_k$  is the state of technology constant and reflects constraints that impede the progress of the programmer

Now by using the above expression it is obtained that,

$$K = L^3 / C_k^3 t_d^4$$

Or,

$$K = C/t_d^4$$

For the same product size,  $C = L^3 / C_k^3$  is a constant.

or,  $K_1/K_2 = t_{d2}^4/t_{d1}^4$

or,  $K \propto 1/t_d^4$

or,  $\text{cost} \propto 1/t_d$

(as project development effort is equally proportional to project development cost)

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in substantial penalty of human effort as well as development cost. For example, if the estimated development time is 1 year, then in order to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

## 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.

The first step in scheduling a software project involves identifying all the tasks necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work breakdown structure formalism helps the manager to breakdown the tasks systematically.

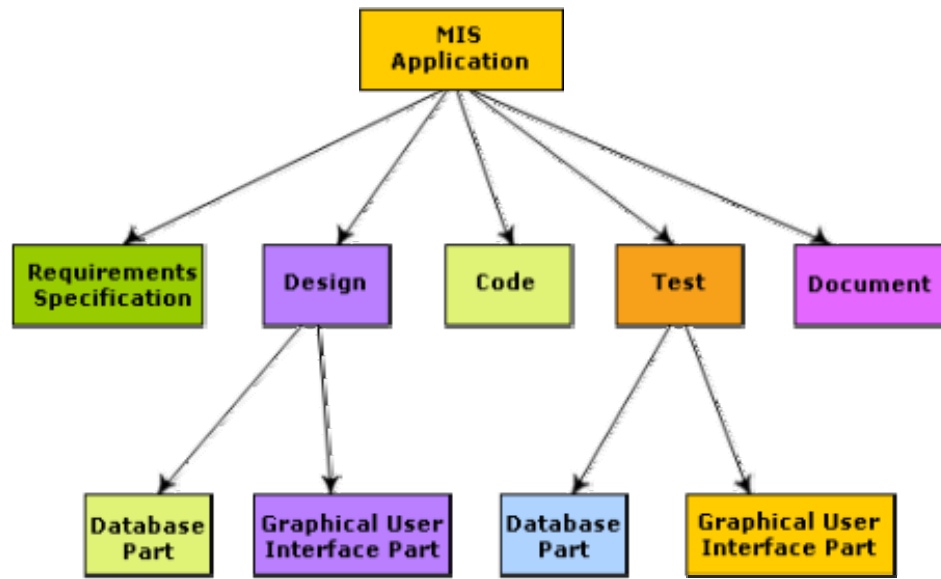
After the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each task may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities are represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

## Work breakdown 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 labeled 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. [Fig. 11.7](#) represents the WBS of an MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.



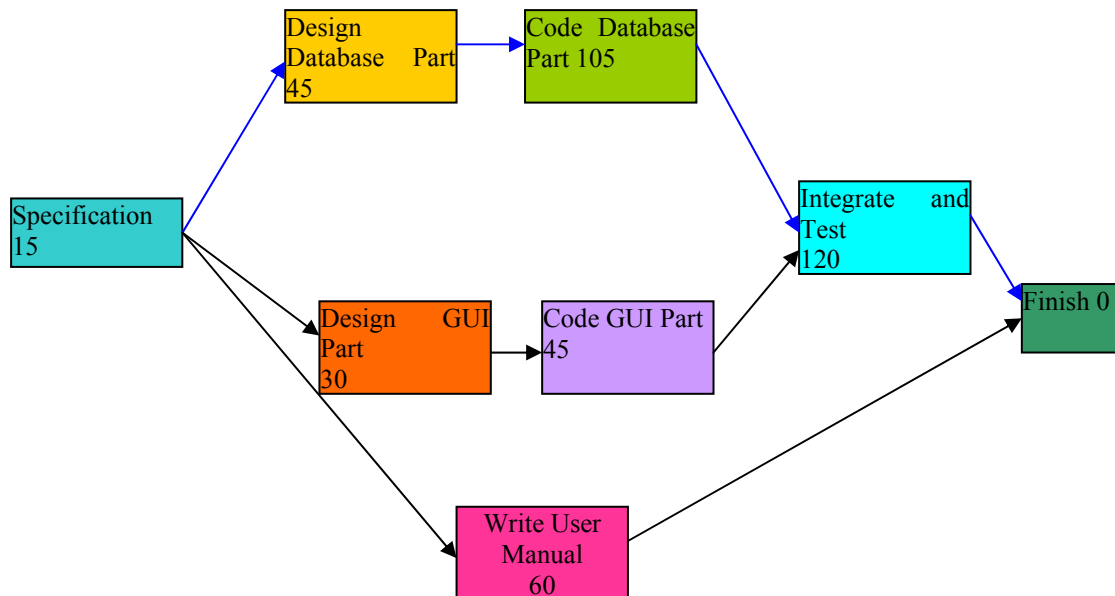
**Fig. 11.7:** Work breakdown structure of an MIS problem

## Activity networks and critical path method

WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies (as shown in [fig. 11.8](#)). Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

Managers can estimate the time durations for the different tasks in several ways. One possibility is that they can empirically assign durations to different tasks. This however is not a good idea, because software engineers often resent such unilateral decisions. A possible alternative is to let engineer himself estimate the time for an activity he can assigned to. However, some managers prefer to estimate the time for various activities themselves. Many managers believe that an aggressive schedule motivates the engineers to do a better and faster job. However, careful experiments have shown that unrealistically aggressive schedules not only cause engineers to compromise on intangible quality aspects, but also are a cause for schedule delays. A good way to achieve accurately in estimation of the task durations without creating undue schedule pressures is to have people set their own schedules.





**Fig. 11.8:** Activity network representation of the MIS problem

### Critical Path Method (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is  $LS - EF$  and equivalently can be written as  $LF - EF$ . The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the “flexibility” in starting and completion of tasks. A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75

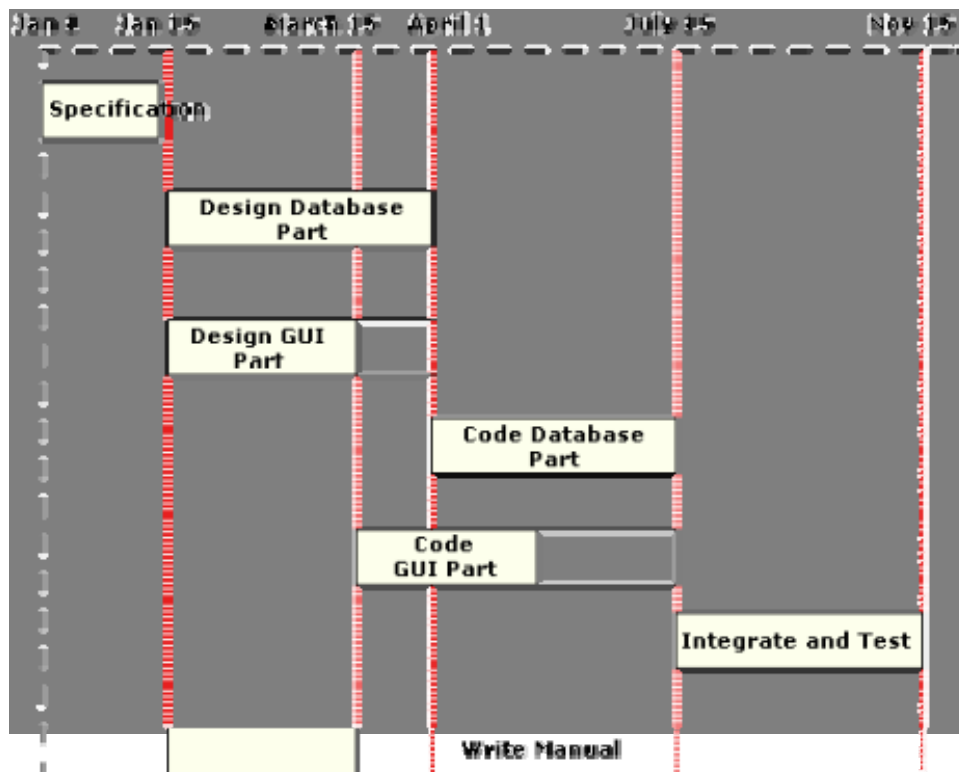
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in fig. 11.8 is shown with a blue arrow.

## Gantt chart

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt charts are used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem of fig. 11.8 is shown in the fig. 11.9.

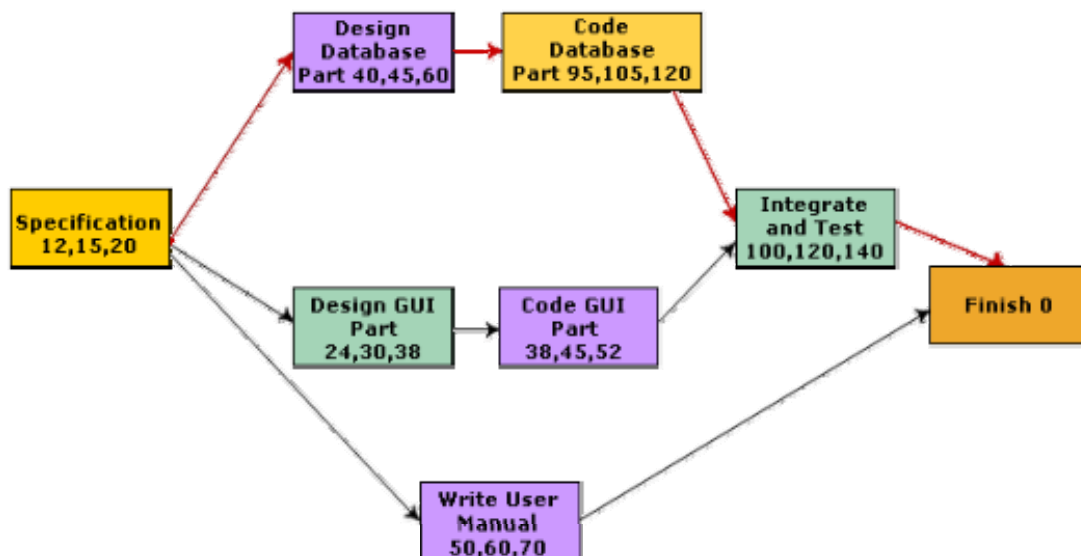


**Fig. 11.9:** Gantt chart representation of the MIS problem

## 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. The boxes of PERT charts are usually annotated with the pessimistic, likely, and optimistic estimates for every task. 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. The PERT chart representation of the MIS problem of fig. 11.8 is shown in fig. 11.10. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited.

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.



**Fig. 11.10:** PERT chart representation of the MIS problem

The following questions have been designed to test the objectives identified for this module:

1. List the major responsibilities of a software project manager.
2. What should be the necessary skills of a software project manager in order to perform the task of software project management?
3. When does the software planning activity start and end in software life cycle? List some important activities that a software project manager performs during software project planning.
4. What are the project related estimates performed by a project manager and also mention the order of project related estimates.
5. What do you understand by Sliding Window Planning? Explain using a few examples the types of projects for which this form of planning is especially suitable. What are its advantages over conventional planning?
6. List the important items that a Software Project Management Plan (SPMP) document should discuss.
7. Point out the major shortcomings of Lines of Code (LOC) metric in order to use it as a software project size metric.
8. List out the major shortcomings of function point metric in order to use it as a software project size metric.
9. What is the necessity of a feature point metric in the context of software project size estimation?
10. Write down the major differences in between empirical estimation technique and heuristic technique.
11. How are the software project related parameters such as program length, program vocabulary, program volume, potential minimum volume, effort to develop the project, project development time estimated using analytical estimation technique?
12. Write down the major differences between expert judgment technique and delphi cost estimation technique.
13. Write down the differences among organic, semidetached and embedded software product.
14. Differentiate among basic COCOMO model, intermediate COCOMO model and complete COCOMO model.
15. As the manager of a software project to develop a product for business application, if you estimate the effort required for completion of the project to be 50 person-months, can you complete the project by employing 50 engineers for a period of one month? Justify your answer.

16. For the same number of lines of code and the same development team size, rank the following software projects in order of their estimated development time. Show reasons behind your answer.
  - A text editor
  - An employee pay roll system
  - An operating system for a new computer
17. Explain Norden's model in the context of staffing requirements for a software project.
18. Explain how can Putnam's model be used to compute the change in project cost with project duration. What are the main disadvantages of using Putnam's model to compute the additional costs incurred due to schedule compression? How can you overcome them?
19. Explain why adding more manpower to an already late project makes it later.
20. Suppose you have estimated the normal development time of a moderate-sized software product to be 5 months. You have also estimated that it will cost Rs. 50,000/- to develop the software product. Now, the customer comes and tells you that he wants you to accelerate the delivery time by 10%. How much additional cost would you charge the customer for this accelerated delivery? Irrespective of whether you take less time or more time to develop the product, you are essentially developing the same product. Why then does the effort depend on the duration over which you develop the product?
21. How does the change of project duration affect the overall project development effort and development cost?
22. Write down the necessary tasks performed by a project manager in order to perform project scheduling.
23. Write down the major differences between work breakdown structure and activity network model.
24. Explain critical path method.
25. Explain when you should use PERT charts and when you should use Gantt charts while you are performing the duties of a project manager.

**Mark all options which are true.**

**1. Normally software project planning activity is undertaken**

- ☐ before the development starts to plan the activities to be undertaken during development
- ☐ once the development activities start
- ☐ after the completion of the project monitoring and control
- ☐ none of the above

**2. Which of the following estimation is carried out first by a project manager during project planning?**

- ☐ estimation of cost
- ☐ estimation of the duration of the project
- ☐ project size estimation
- ☐ estimation of development effort

**3. Sliding Window Planning involves**

- ☐ planning a project before development starts
- ☐ planning progressively as development proceeds
- ☐ planning a project after development starts
- ☐ none of the above

**4. A project estimation technique based on making an educated guess of the project parameters (such as project size, effort required to develop the software, project duration, cost etc.) is**

- ☐ analytical estimation technique
- ☐ heuristic estimation technique
- ☐ empirical estimation technique
- ☐ none of the above

**5. An example of single variable heuristic cost estimation model is**

- ☐ Halstead's software science
- ☐ basic COCOMO model
- ☐ intermediate COCOMO model
- ☐ complete COCOMO model

**6. Operating systems and real-time system programs can be considered as**

- ☐ application programs
- ☐ utility programs
- ☐ system programs
- ☐ none of the above

7. Compilers, linkers, etc. can be considered as

- ☐ application programs
- ☐ utility programs
- ☐ system programs
- ☐ none of the above

8. Data processing programs are considered as

- ☐ utility programs
- ☐ system programs
- ☐ application programs
- ☐ none of the above

9. During project scheduling, resource allocation to different activities is done using which of the following representations?

- ☐ PERT chart
- ☐ activity network representation
- ☐ work breakdown structure
- ☐ Gantt chart

**Mark the following as either True or False. Justify your answer.**

1. Size of a project, as used in COCOMO is the size of the final executable code in bytes.
2. According to the COCOMO model, cost is the fundamental attribute of a software product, based on which size and effort are estimated.
3. If we increase the size of a software product by two times then the time required to develop that software product would be double.
4. The number of development personnel required for any software development project can be obtained by dividing the total (estimated) effort by the total (estimated) duration of the project.
5. For the development of the same product, the larger is the size of a software development team, the faster is the product development. (for simplicity, assume that all engineers are equally proficient and have exactly similar experience).
6. As a project manager it would be worthwhile on your part to reduce the project duration by half provided the customer agrees to pay for the increased manpower requirements.
7. PERT charts are a sophisticated form of activity chart.