3.3 METRICS FOR PROJECT SIZE ESTIMATION

As already mentioned, accurate estimation of the problem size is fundamental to satisfactory estimation of other project parameters such as effort, time duration for completing the project and the total cost for developing the software. Before discussing appropriate metrics to estimate the size of a project, let us examine what the term problem size means in the context of software projects. The size of a project 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 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 which are discussed in the following.

3.3.1 Lines of Code (LOC)

(LOC is the simplest among all metrics available to estimate project size. Consequently, this metric is extremely popular. This metric measures the size of a project 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 are ignored)

Determining the LOC count at the end of a project is very simple. 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, one would have to make a systematic guess.

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 predict the LOC count for the various leaf-level modules sufficiently accurately, past experience in developing similar products is very helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation. However, 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 in stead of lines of code.

- LOC is a measure of the coding activity alone. On the other hand, a good problem size measure should consider the total 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 by different developers (that is, productivity), they would be discouraging code reuse by developers!

- 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 only be accurately computed only after the code has been fully developed. Therefore, the LOC metric is of 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.

3.3.2 Function Point Metric

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 query book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding input data. Thus, a computation of the number of input and output data values to a system gives some indication of the number of functions supported by the system.

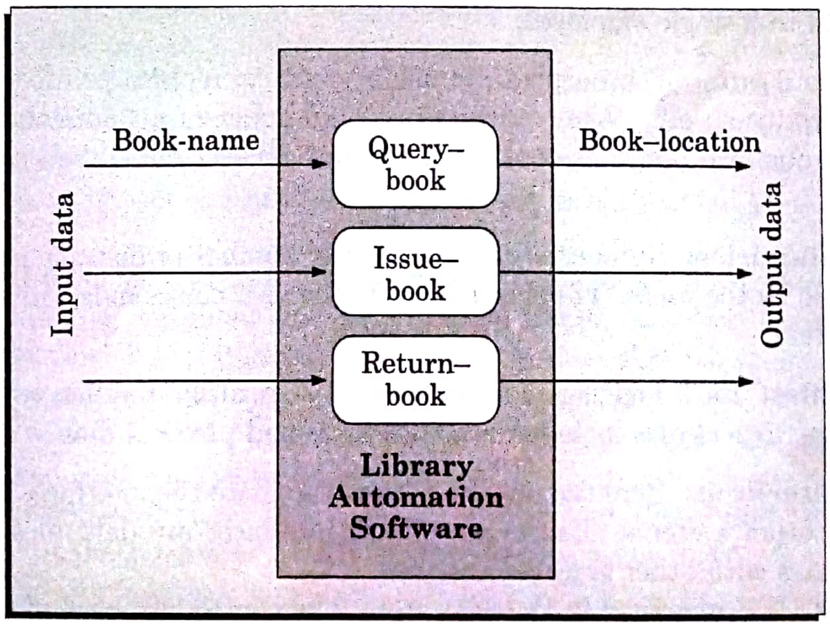

Figure 3.2: System function as a map of input data to output data.

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. Interfaces refer to the different mechanisms that need to be supported for data transfer with other external systems. Besides using the number of input and output data values, function point metric computes the size of a software product (in units of function points or FPs) using three other characteristics of the product discussed above and shown in the following expression.

Function point is computed in three steps. The first step is to compute the unadjusted function point (UFP). In the next step, the UFP is refined to reflect the differences in the complexities of the different parameters of the expression for UFP computation (shown below). In the third and the final step, FP is computed by further refining UFP to account for the specific characteristics of the project that can influence the development effort.

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

The expression shows the computation of the unadjusted function points (UFP) as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed by Albrecht empirically and was validated through data gathered from many projects.

The meaning of the different parameters of this expression is as follows:

1. **Number of inputs:** Each data item input by the user is counted. Data inputs should be distinguished from user inquires. 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 simply added up to compute 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 payroll 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.

2. **Number of outputs:** The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While computing 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 output.

3. **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.

4. **Number of files:** Each logical file is counted. A logical file implies a group of logically related data. Thus, logical files include data structures and physical files.

5. **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.

The computed UFP is refined in the next step. The complexity level of each of the parameters are graded as simple, average, or complex. The weights for the different parameters can then be computed based on Table 3.1. Thus, rather than each input being computed as four function points, very simple inputs can be computed as three function points and very complex inputs as six function points.

Table 3.1: Refinement of function point entities

Type	*Simple*	*Average*	*Complex*
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15
Number of interfaces	5	7	10

A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of various project parameters that can influence the development effort such as high transaction rates, response time requirements, scope for reuse, etc. Albrecht identified 14 parameters that can influence the development effort. Each of these 14 factors is assigned a value 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 84, TCF can vary from 0.65 to 1.35. Finally, FP is given as the product of UFP and TCF. Thar is, FP=UFP*TCF.

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. For example, in a library automation software, the `create-member` feature would be much simpler compared to the `loan-from-remote-library` feature. 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* has been proposed.

Feature point metric incorporates *algorithm* complexity as an extra parameter. 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.

Proponents of function point and feature point metrics claim that these metrics are language-independent and can be easily computed from the SRS document during project planning, whereas opponents claim that these metrics are subjective and require a sleight of hand. An example of the subjective nature of the function point metric can be that the way one would group logically related data items can be very subjective. For example, consider that certain data `employee-details` consists of the employee name and employee address. Then, it is possible that one can consider it as a single unit of data. Also, someone else can consider the employees address as one unit and name as another. Therefore, there is sufficient scope for different project managers to arrive at different function point measures for the same problem.

3.4 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 an appropriate project cost to the customer but also form the basis for resource planning and scheduling. There are three broad categories of estimation techniques:

1. Empirical estimation techniques
2. Heuristic techniques
3. Analytical estimation techniques

In the following, we provide an overview of the different categories of estimation techniques.

3.4.1 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, over the years, different activities involved in estimation have been formalized to certain extent. We shall discuss two such formalizations of the basic empirical estimation techniques in sections 3.5.1 and 3.5.2.

3.4.2 Heuristic Techniques

Heuristic techniques assume that the relationships among the different project parameters can be modelled 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 multivariable 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 a 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 COCOMO model (discussed in section 3.6.1), is an example of a single variable cost estimation model.

A multivariable cost estimation model takes the following form:

$$\text{Estimated resource} = c_1 * ep_1^{d_1} + c_2 * ep_2^{d_2} + ...$$

where ep_1, ep_2, ... are the basic (independent) characteristics of the software already estimated, and c_1, c_2, d_1, d_2, ... are constants. Multivariable estimation models are expected to ponsibility of compiling the results and preparing the

3.6 COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COCOMO (COnstructive COst estimation MOdel) was proposed by Boehm, 1981. Boehm postulated that any software development project can be classified into any 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 requires us to consider not only the characteristics of the product but also those of the development team and development environment. Roughly speaking, the three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs[1] 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.

Brooks, 1975 states that utility programs are roughly three times as difficult to write as application programs, and system programs are roughly three times as difficult as utility programs. Thus, according to Brooks, the relative levels of product development complexity for the three categories (application, utility and system programs) of products are 1:3:9.

Boehm's [1981] definitions of organic, semidetached, and embedded systems are elaborated as follows:

1. Organic: We can consider a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development

[1]A data processing program is one which processes large volumes of data using a simple algorithm. An example of a data processing application is a payroll software. A payroll software computes the salaries of the employees and prints cheques for them. In a payroll software, the algorithm for pay computation is fairly simple. The only complexity that arises while developing such a software product arises from the fact that the pay computation has to be done for a large number of employees.

team is reasonably small, and the team members are experienced in developing similar types of projects.

2. **Semidetached:** A development project can be considered to be of semidetached type, if the development team 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 stringent regulations on the operational procedures exist.

Observe that Boehm in addition to considering the characteristics of the product being developed, considers the characteristics of the team members in deciding the category of the development project. Thus, a simple data processing program may be classified as semidetached if the team members are inexperienced in the development of similar products. For the three product categories, Boehm provides different sets of expressions to predict the effort (in units of person-months) and development time from the size estimation given in KLOC (Kilo Lines of Source Code). One person-month is the effort an individual can typically put in a month. This effort estimate takes into account the productivity losses that may occur due to lost time such as holidays, weekly offs, coffee breaks, etc.

Note that effort estimation is expressed in units of person-months (PM). Person-month (PM) is considered to be an appropriate unit for measuring effort because developers are typically assigned to a project for a certain number of months. The person-month unit indicates the work done by one person working on the project for one month. It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither nor does it imply that 1 person should be employed for 100 months. The effort estimation simply denotes the area under the person-month curve (see Figure 3.3) for the project. The plot in Figure 3.3 shows that different number of personnel may work at

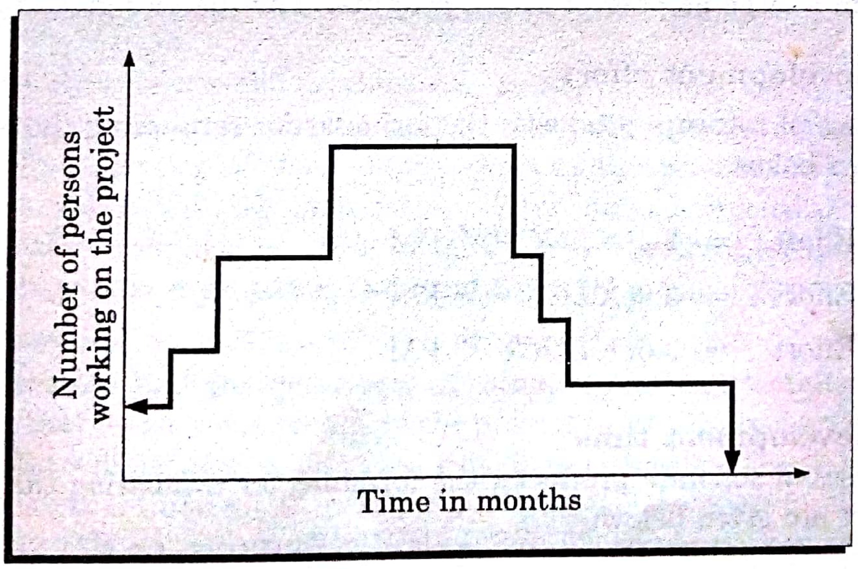

Figure 3.3: Person-month curve.

different point in the project development, as is typical in a practical industry scenario. The number of personnel working on the project usually increases and decreases by an integral number, resulting in the sharp edges in the plot. We shall elaborate in Section 3.8 how the number of persons to work at any time on the product development is determined.

According to Boehm, software cost estimation should be done through three stages: basic COCOMO, intermediate COCOMO, and complete COCOMO. We discuss these stages as follows:

3.6.1 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:

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

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

Where

(a) KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

(b) a_1, a_2, b_1, b_2 are constants for each category of software products,

(c) Tdev is the estimated time to develop the software, expressed in months,

(d) Effort is the total effort required to develop the software product, expressed in person months (PMs).

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 nLOC. The values of a_1, a_2, b_1, b_2 for different categories of products as given by Boehm [1981]. He derived the above expressions by examining historical data collected from a large number of actual projects. The theories given by Boehm were:

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
Semidetached	: 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
Semidetached	: Tdev	$= 2.5(Effort)^{0.35}$ Months
Embedded	: Tdev	$= 2.5(Effort)^{0.32}$ Months

We can gain some insight into the basic COCOMO model, if we plot the estimated characteristics for different software sizes. Figure 3.4 shows a plot of estimated effort versus product size. From Figure 3.4, we can observe that the effort is somewhat superlinear (slope of the curve ≥ 1) in the size of the software product. This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. However, observe that the increase in effort with size is not as bad as that was portrayed in Chapter 1. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles.

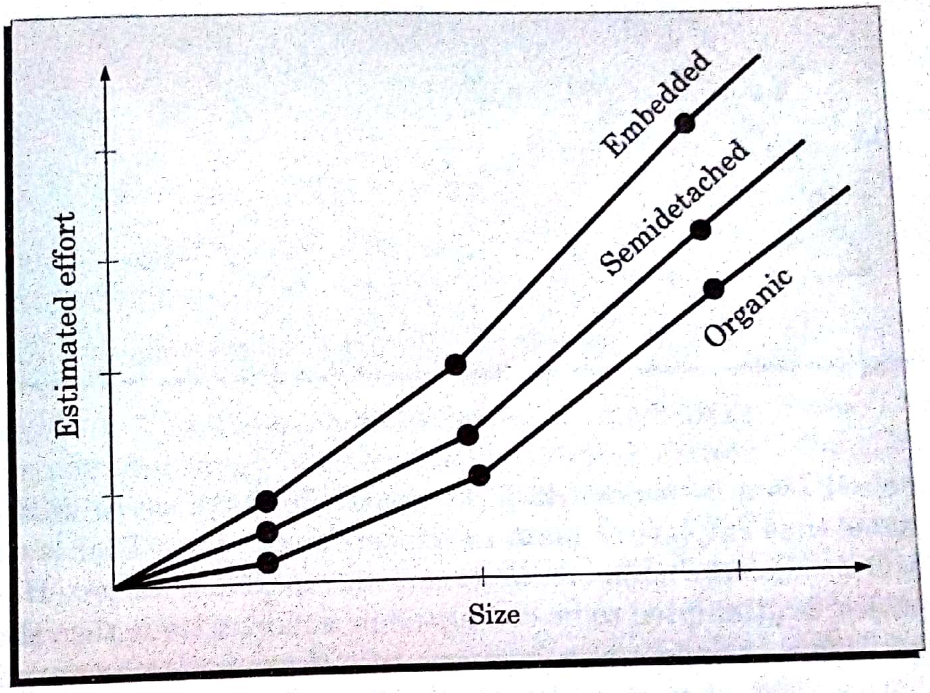

Figure 3.4: Effort versus product size.

The development time versus the product size in KLOC is plotted in Figure 3.5. From Figure 3.5, we can observe that the development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately. It may appear surprising that the duration curve does not increase superlinearly. The apparent anomaly can be explained by the fact that COCOMO assumes that a project is carried out not by a single person but by a team of developers.

It is important to note that the effort and duration estimations obtained using the COCOMO model imply that if you try to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if you complete the project over a longer period of time than that estimated, then there is almost no decrease in the estimated cost value. The reasons for this are discussed in Section 3.8. Thus, we can consider that the computed effort and duration values to indicate the following.

The effort and duration values computed by COCOMO are the values for doing the work in the shortest time without unduly increasing manpower cost.

termine the staffing level by a simple division. However, we are going to examine the staffing problem in more detail in Section 3.8. From the discussion in Section 3.8 it would become clear that the simple division approach to obtain the staff size is highly improper.

Example 3.1 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 developers is Rs. 15,000 per month. Determine the effort required to develop the software product, the nominal development time, and the cost to develop the product.

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 $= 91 \times 15,000 =$ Rs. 1,465,000

3.6.2 Intermediate COCOMO

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:

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, 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.

We have discussed only the basic ideas behind the COCOMO model. A detailed discussion on the COCOMO model are beyond the scope of this book and the interested reader may refer [Boehm, 81].

3.6.3 Complete COCOMO

A major shortcoming of both the basic and the intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up of several smaller subsystems. These subsystems may have widely different characteristics. For example, some subsystems 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 subsystem 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.

Let us consider the following development project 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 subcomponents:

- 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 semidetached 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.

To improve the accuracy of their results, the different parameter values of the model can be fine-tuned and validated against an organization's historical project database to obtain more accurate estimations. Estimation models such as COCOMO are not accurate and lack a full scientific justification. Still, software cost estimation models such as COCOMO are required for an engineering approach to software project management. Companies consider computed estimates to be satisfactory, if these are within about 80% of final cost. Although these estimates are gross approximations—without such models, one has only subjective judgements to rely on.

3.6.4 COCOMO 2

Since the time that COCOMO estimation model proposed in the early 1980s, both the software development paradigm and problems have undergone a sea change. The present day software projects are much larger in size and reuse of existing software to develop new products has become pervasive. This has given rise to component-based development. New life cycle models and development paradigms are being deployed for web-based and component-based software. During the 1980s rarely any program was interactive, and graphical user interfaces were almost non-existent. On the other hand, most of the present day software is highly interactive and has elaborate graphical user interface. To make COCOMO suitable in the changed scenario, Boehm proposed COCOMO 2 [Boehm, 95].

COCOMO 2 provides three increasingly detailed cost estimation models. These can be used to estimate project costs at different phases of the software. As the project progresses through these models can be applied at the different stages of the same project.

1. **Application composition:** This model as the name suggests, can be used to estimate cost for prototyping, e.g. to resolve user interface issues.

2. **Early design:** This supports estimation of cost at the architectural design stage.

3. **Post-architecture stage:** This provides cost estimation during detailed design and coding stage.

The post-architectural model can be considered as an update of the original COCOMO. We discuss these three models in the following.

Application composition model

The application composition model is based on counting the number of screens, reports and 3GL modules. Each of these components is considered to be an object (this has nothing to do with the concept of objects in the object-oriented paradigm). These are used to compute the object points of the application.

Effort is estimated in the application composition model as follows:

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 a report is determined by the number of tables and views it contains.

Table 3.2: SCREEN complexity assignments for the number of views and data tables

Number of views	*Tables* < 4	*Tables* < 8	*Tables* ≥ 8
< 3	simple	simple	medium
3–7	simple	medium	difficult
>8	medium	difficult	difficult

3. Use the weight values in Tables 3.2 to 3.4.

Table 3.3: Report complexity assignments for the number and source of the data tables

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

The weights are supposed to correspond to the amount of effort required to implement an instance of an object at the assigned complexity class.

4. Determine the number of object points
   Add all the assigned complexity values for the object instances together—The Object Count.
5. Estimate percentage of reuse expected in the system (reuse refers to the amount of pre-developed software that will be used within the system). Then, evaluate New Object-Point count (NOP),

$$NOP = \frac{(\text{Object-Points})(100 - \%\text{ of reuse})}{100}$$

Table 3.4: Table of complexity weights for each class for each object type

Object type	*Simple*	*Medium*	*Difficult*
Screen	1	2	3
Report	2	5	8
3GL component	—	—	10

6. Determine a productivity rate, PROD = NOP/ person-month, using Table 3.5. The productivity depends on the experience of the developers as well as the maturity of the CASE environment used.
7. Finally, the person-month is computed as E=NOP/PROD.

Table 3.5: Productivity table

Developers' experience	*Very low*	*Low*	*Nominal*	*High*	*Very high*
CASE maturity	Very Low	Low	Nominal	High	Very High
PROD	4	7	13	25	50

Early design model

The unadjusted function points (UFP) are counted and converted to Source Lines of Code (SLOC). In a typical programming environment, each UFP would correspond to about 128 lines of C, 29 lines of C++, or 320 lines of assembly code. Seven cost drivers that characterize the post-architecture model are used. These are rated on a seven points scale. The cost drivers include product reliability and complexity, the extent of reuse, platform difficulty, personnel experience, CASE support, and schedule.

The effort is then calculated using the following formula:

$$\text{Effort} = KSLOC \times \Pi_i \text{ costdriver}_i$$

Post-architecture model

The effort is then calculated using the following formula, which is similar to the original COCOMO model.

$$\text{Effort} = a \times KSLOC^b \times \Pi_i \text{ cost driver}_i$$

The post-architecture model differs from the original COCOMO model in the choice of the set of cost drivers and the range of values of the exponent b. The exponent b can take values in the range of 1.01 to 1.26. The details of the COCOMO 2 model, and the exact values of b and the cost drivers can be found in [Boehm 97].

3.7 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 volume, actual volume, language level, effort and development time.

For a given program, let η_1 be the number of unique operators used in the program, η_2 be the number of unique operands used in the program, N_1 be the total number of operators used in the program, and N_2 be the total number of operands used in the program.

Although the terms *operators* and *operands* have intuitive meanings, a precise definition of these terms is needed to avoid ambiguities. But, unfortunately we would not be able to provide a precise definition of these two terms. There is no general agreement among researchers on what is the most meaningful way to define the operators and operands for different programming languages. However, a few general guidelines regarding identification of operators and operands for any programming language can be provided. For instance, assignment, arithmetic, and logical operators are usually counted as operators. A pair of parentheses, as well as a block begin—block end pair, are considered as single operators. A label is considered to be an operator, if it is used as the target of a GOTO statement. The constructs `if ... then ... else ... endif` and a `while ... do` are considered as single operators. A sequence (statement termination) operator ';' is considered as a single operator. Subroutine declarations and variable declarations comprise the operands. Function name in a function call statement is considered as an operator, and the arguments of the function call are considered as operands. However, the parameter list of a function in the function declaration statement is not considered as operands. We list below what we consider to be the set of operators and operands for the ANSI C language. However, it should be realized that there is considerable disagreement among various researchers in this regard.

Operators and Operands for the ANSI C Language

The following is a suggested list of operators for the ANSI C language:

```
( [ . , -> * + - ~ ! ++ -- * / % + - << >> < > <= >= !=
== & ^ | && || = *= /= %= += -= <<= >>= &= ^= |= : ? { ;
CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
```
and a `function name in a function call`

Operands are those variables and constants which are being used with operators in expressions. Note that variable names appearing in declarations are not considered as operands.

Example 3.2 Consider the expression `a = &b ;`
a, b are the operands and =, & are the operators.

Example 3.3 The function name in a function definition is not counted as an operator.

```
int func ( int a, int b )
{
    . . .
}
```

For the above example code, the operators are: {}, ( ) We do not consider func, a, and b as operands, since these are a part of the function definition.

Example 3.4 Consider the function call statement: `func ( a, b );`. In this, `func ','` and ; are considered as operators and variables a, b are treated as operands.