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

## 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.

### 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.

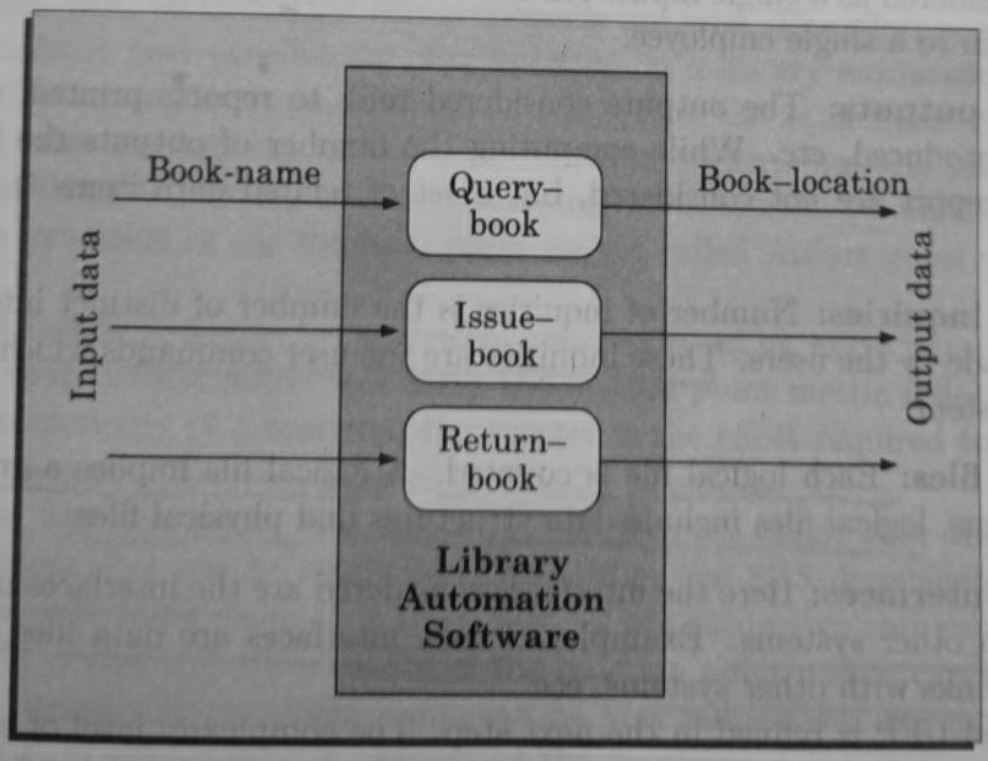


**Figure 3.2:** System function as a map of input data to output data.

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 FP) 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 as low). 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.

# 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

### 3.4.3 Analytical Estimation Techniques

Analytical estimation techniques derive the required results starting with certain basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have certain scientific basis. We shall discuss the Halstead's software science in Section 3.7 as an example of an analytical technique. As we will see, 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.

## 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.

### 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

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:

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

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

$$\text{Cost required to develop the product} = 91 \times 15,000 = \text{Rs. } 1,465,000$$

### 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.