5 Software Effort Estimation

Learning Objectives

- Avoid the dangers of unrealistic estimates
- Understand the range of estimating methods that can be used
- Estimate projects using a bottom-up approach
- Estimate the effort needed to implement software using a procedural programming language
- Count the function points for a system
- Understand the COCOMO II approach and use it to estimate project effort

5.1 Introduction

A successful project is one delivered '*on time, within budget and with the required quality*'. This implies that targets are set which the project manager then tries to meet. This assumes that the targets are reasonable – no account is taken of the possibility of project managers achieving record levels of productivity from their teams, but still not meeting a deadline because of incorrect initial estimates. Realistic estimates are therefore crucial.

A project manager like Amanda has to produce estimates of *effort*, which affect costs, and of activity *durations*, which affect the delivery time. These could be different, as in the case where two testers work on the same task for the same five days.

Some of the difficulties of estimating arise from the complexity and invisibility of software. Also, the intensely human activities which make up system development cannot be treated in a purely mechanistic way. Other difficulties include:

- *Subjective nature of estimating* For example, some research shows that people tend to underestimate the difficulty of small tasks and over-estimate that of large ones.

In Chapter 1, the special characteristics of software identified by Brooks, i.e. complexity, conformity, changeability and invisibility, were discussed.

- *Political implications* Different groups within an organization have different objectives. The IOE information systems development managers may, for example, want to generate work and will press estimators to reduce cost estimates to encourage higher management to approve projects. As Amanda is responsible for the development of the annual maintenance contracts subsystem, she will want to ensure that the project is within budget and timescale, otherwise this will reflect badly on herself. She might therefore try to increase the estimates to create a 'comfort zone'. To avoid these 'political' influences, one suggestion is that estimates be produced by a specialist estimating group, independent of the users and the project team. Not all agree with this, as developers will be more committed to targets they themselves have set.

The possibility of the different groups with stakes in a project having different and possibly conflicting objectives was discussed in Chapter 1.

- *Changing technology* Where technologies change rapidly, it is difficult to use the experience of previous projects on new ones.
- *Lack of homogeneity of project experience* Even where technologies have not changed, knowledge about typical task durations may not be easily transferred from one project to another because of other differences between projects.

The ISO 12207 standard, touched upon in Chapter 1, is an attempt to address this problem by standardizing on some of the terms used.

It would be very difficult on the basis of this information to advise a project manager about what sort of productivity to expect, or about the probable distribution of effort between the phases of design, coding and testing that could be expected from a new project.

Using existing project data for estimating is also difficult because of uncertainties in the way that various terms can be interpreted. For example, what exactly is meant by the term 'testing'? Does it cover the activities of the software developer when debugging code?

Exercise 5.1

Calculate the productivity (i.e. SLOC per work month) of each of the projects in Table 5.1 and also for the organization as a whole. If the project leaders for projects a and d had correctly estimated the source number of lines of code (SLOC) and then used the average productivity of the organization to calculate the effort needed to complete the projects, how far out would their estimates have been from the actual effort?

TABLE 5.1 Some project data – effort in work months (as percentage of total effort in brackets)

The figures are taken from B. A. Kitchenham and N. R. Taylor (1985) 'Software project development cost estimation' *Journal of Systems and Software (5)*. The abbreviation SLOC stands for 'source lines of code'. SLOC is one way of indicating the size of a system.

Project	Design		Coding		Testing		Total	
	wm	(%)	wm	(%)	wm	(%)	wm	SLOC
a	3.9	(23)	5.3	(32)	7.4	(44)	16.7	6050
b	2.7	(12)	13.4	(59)	6.5	(26)	22.6	8363
c	3.5	(11)	26.8	(83)	1.9	(6)	32.2	13334
d	0.8	(21)	2.4	(62)	0.7	(18)	3.9	5942

(Contd)

e	1.8	(10)	7.7	(44)	7.8	(45)	17.3	3315
f	19.0	(28)	29.7	(44)	19.0	(28)	67.7	38988
g	2.1	(21)	7.4	(74)	0.5	(5)	10.1	38614
h	1.3	(7)	12.7	(66)	5.3	(27)	19.3	12762
i	8.5	(14)	22.7	(38)	28.2	(47)	59.5	26500

Exercise 5.2

In the data presented in Table 5.1, observe that programmer productivity varies from 7 SLOC/day to 150 SLOC/day. In fact, in the industry the average productivity figure for programmers is only about 10 SLOC/day. Would you consider programmer productivity of 10 SLOC/day to be too low?

5.2 Where are the Estimates Done?

Estimates are carried out at various stages of a software project for a variety of reasons.

- *Strategic planning* Project portfolio management involves estimating the costs and benefits of new applications in order to allocate priorities. Such estimates may also influence the scale of development staff recruitment.

Chapter 2 discussed project portfolio management in some detail.

- *Feasibility study* This confirms that the benefits of the potential system will justify the costs.
- *System specification* Most system development methodologies usefully distinguish between the definition of the users' requirements and the design which shows how those requirements are to be fulfilled. The effort needed to implement different design proposals will need to be estimated. Estimates at the design stage will also confirm that the feasibility study is still valid.
- *Evaluation of suppliers' proposals* In the case of the IOE annual maintenance contracts subsystem, for example, IOE might consider putting development out to tender. Potential contractors would scrutinize the system specification and produce estimates as the basis of their bids. Amanda might still produce her own estimates so that IOE could question a proposal which seems too low in order to ensure that the proposer has properly understood the requirements. The cost of bids could also be compared to in-house development.

The estimate at this stage cannot be based only on the user requirement: some kind of technical plan is also needed – see Chapter 4.

- *Project planning* As the planning and implementation of the project becomes more detailed, more estimates of smaller work components will be made. These will confirm earlier broad-brush estimates, and will support more detailed planning, especially staff allocations.

As the project proceeds, so the accuracy of the estimates should improve as knowledge about the project increases. At the beginning of the project the user requirement is of paramount importance and premature consideration of the possible physical implementation is discouraged. However, in order to produce an estimate, there will need to be speculation about the eventual shape of the application.

To set estimating into the context of the Step Wise framework (Figure 5.1) presented in Chapter 3, re-estimating could take place at almost any step, but specific provision is made for the production of a relatively high-level estimate at Step 3, 'Analyse project characteristics', and for each individual activity in Step 5. As Steps 5–8 are repeated at progressively lower levels, so estimates will be done at a finer degree of detail. As we will see later in this chapter, different methods of estimating are needed at these different planning steps.

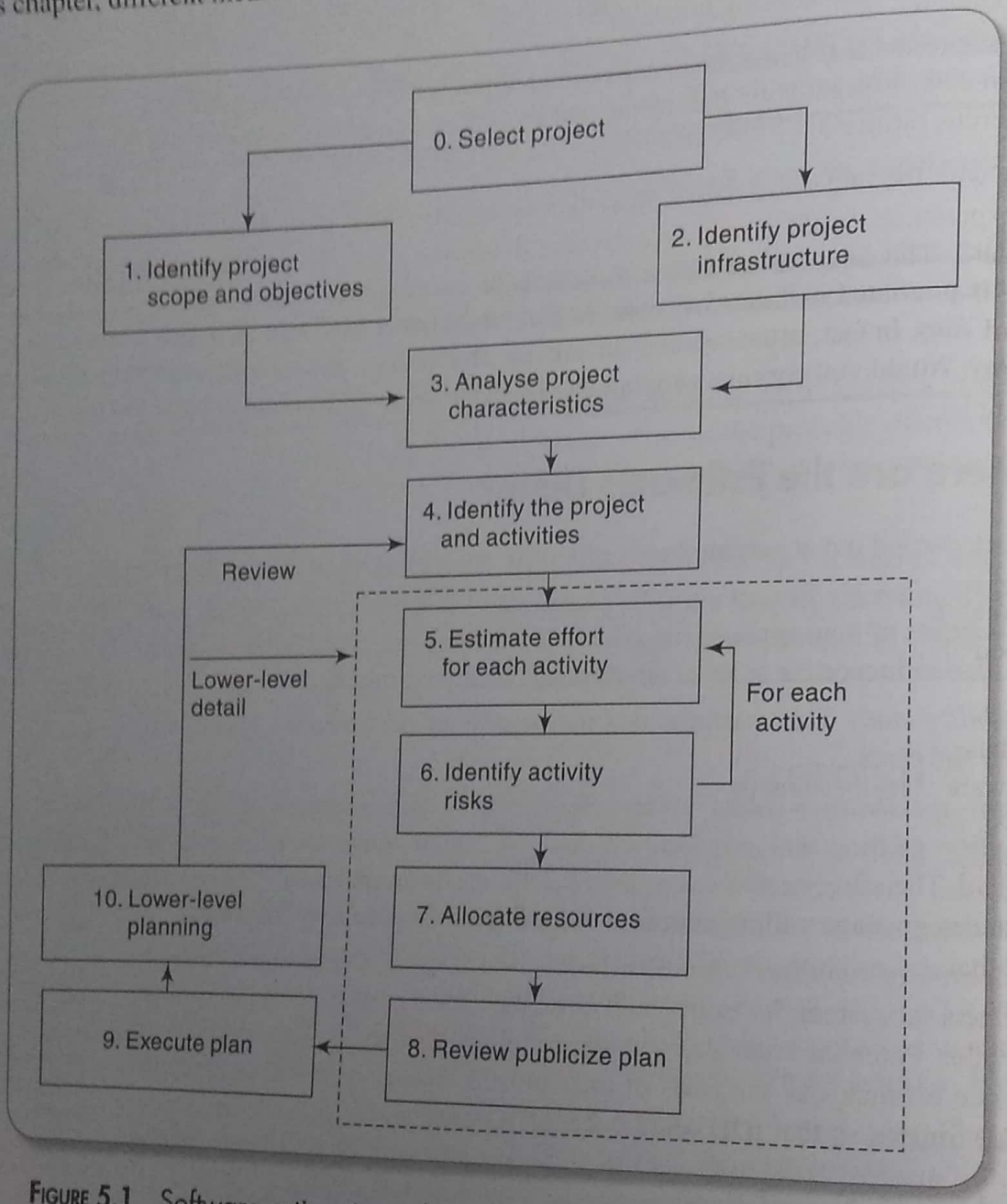

FIGURE 5.1 Software estimation takes place in Steps 3 and 5 in particular

5.3 Problems with Over- and Under-Estimates

A project leader such as Amanda will need to be aware that an over-estimate may cause the project to take longer than it would otherwise. This can be explained by the application of two 'laws'.

- *Parkinson's Law* '*Work expands to fill the time available*', that is, given an easy target staff will work less hard.

- *Brooks' Law* The effort of implementing a project will go up disproportionately with the number of staff assigned to the project. As the project team grows in size, so will the effort that has to go into management, coordination and communication. This has given rise, in extreme cases, to the notion of Brooks' Law: '*putting more people on a late job makes it later*'. If there is an over-estimate of the effort required, this could lead to more staff being allocated than needed and managerial overheads being increased.

Parkinson's law was originally expounded in C. Northcote Parkinson's tongue-in-cheek book *Parkinson's Law*, John Murray, 1957. Brooks' law comes from *The Mythical Man-month* which has been referred to already.

Some have suggested that while the under-estimated project might not be completed on time or to cost, it might still be implemented in a shorter time than a project with a more generous estimate.

The danger with the under-estimate is the effect on quality. Staff, particularly those with less experience, could respond to pressing deadlines by producing work that is substandard. This may be seen as a manifestation of Weinberg's zeroth law of reliability: '*if a system does not have to be reliable, it can meet any other objective*'. Substandard work might only become visible at the later, testing, phases of a project which are particularly difficult to control and where extensive rework can easily delay project completion.

See, for example, T. K. Hamid and S. E. Madnick (1986) 'Impact of schedule estimation on software project behaviour' *IEEE Software* July 3(4) 70–5.

Exercise 5.3

How do agile methods such as XP – see Chapter 4 – attempt to address the problems with estimates described above?

Research has found that motivation and morale are enhanced where targets are achievable. If, over time, staff become aware that the targets set are unattainable and that projects routinely miss targets, motivation is reduced. People like to think of themselves as winners and there is a general tendency to put success down to our own efforts and blame failure on the organization.

An estimate is not really a prediction, it is a *management goal*. Barry Boehm has suggested that if a software development cost is within 20% of the estimated cost for the job then a good manager can turn it into a self-fulfilling prophecy. A project leader like Amanda will work hard to make the actual performance conform to the estimate.

Barry Boehm devised the COCOMO estimating models which are described later in this chapter.

5.4 The Basis for Software Estimating

The need for historical data

Most estimating methods need information about past projects. However, care is needed when applying past performance to new projects because of possible differences in factors such as programming languages and the experience of staff. If past project data is lacking, externally maintained datasets of project performance data can be accessed. One well-known international database is that maintained by the International Software Benchmarking Standards Group (ISBSG), which currently contains data from 4800 projects.

Details of the work of the International Software Benchmarking Standards Group can be found at http://isbsg.org

Parameters to be estimated

The project manager needs to estimate two project parameters for carrying out project planning. These two parameters are effort and duration. Duration is usually measured in months. Work-month (wm) is a popular unit for effort measurement. We have already used this unit of effort measurement in Table 5.1. The term person-month (pm) is also frequently used to mean the same as work-month. One person-month is the effort an individual can typically put in a month. The person-month estimate implicitly takes into account the productivity losses that normally occur due to time lost in holidays, weekly offs, coffee breaks, etc. Person-month (pm) is considered to be an appropriate unit for measuring effort compared to person-days or person-years because developers are typically assigned to a project for a certain number of months.

Measure of work

Measure of work involved in completing a project is also called the size of the project. Work itself can be characterized by cost in accomplishing the project and the time over which it is to be completed. Direct calculation of cost or time is difficult at the early stages of planning. The time taken to write the software may vary according to the competence or experience of the software developers might not even have been identified. Implementation time may also vary depending on the extent to which CASE (Computer Aided Software Engineering) tools are used during development. It is therefore a standard practice to first estimate the project size; and by using it, the effort and time taken to develop the software can be computed. Thus, we can consider project size as an independent variable and the effort or time required to develop the software as dependent variables.

Let us examine the meaning of the term 'project size'. The size of a project is obviously not the number of bytes that the source code occupies, neither is it the 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. Two metrics are at present popularly being used to measure size. These are Source Lines of Code (SLOC) and Function Point (FP). The SLOC measure suffers from various types of disadvantages, which are to a great extent corrected in the FP measure. However, the SLOC measure is intuitively simpler, so it is still being widely used. It is important, however, to be aware of the major shortcomings of the SLOC measure.

- *No precise definition.* SLOC is a very imprecise measure. Unfortunately, researchers have not been consistent on points like does it include comment lines or are data declarations to be included? The writers' view is that comment lines are excluded in determining the SLOC measure. This can be debated, but the main point is that consistency is essential.
- *Difficult to estimate at start of a project.* From the project manager's perspective, the biggest shortcoming of the SLOC metric is that it is very difficult to estimate it during project planning stage, and can be accurately computed only after the development of the software is complete. The SLOC count can only be guessed at the beginning of a project, often leading to grossly inaccurate estimations.
- *Only a code measure.* SLOC is a measure of coding activity alone. This point has been illustrated in Exercise 5.2. A good problem size measure should consider the effort required for carrying out all the life cycle activities and not just coding.
- *Programmer-dependent.* SLOC gives a numerical value to the problem size that can vary widely with the coding style of individual programmers. This aspect alone renders any LOC-based size and effort estimations inaccurate.

R. E. Park has devised a standard for counting source statements that has been widely adopted – see *Software Size Measurement: A Framework for Counting Source Statements*, Software Engineering Institute, 1992.

- *Does not consider code complexity.* Two software components with the same KLOC will not necessarily take the same time to write, even if done by the same programmer in the same environment. One component might be more complex. Because of this, the effort estimate based on SLOC might have to be modified to take its complexity into account. Attempts have been made to find objective measures of complexity, but it depends to a large extent on the subjective judgment of the estimator.

Measure of effort

Man-month (MM) [also referred to as Person-month (PM)] is a popular unit for effort measurement. It quantifies the effort that can be put in by one person over 1 month. Person-month (PM) is considered to be an appropriate unit for measuring effort as compared to person-day or person-year, because developers are typically assigned to a project for a certain number of months. It should be noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither does it imply that one person should be employed for 100 months to complete the project.

Even though the effort that has been put in by a team can be measured in units of man-month based on the number of persons deployed and the number of months that they have worked, it may be fallacious to relate this to the progress achieved in the project. This aspect has been highlighted by Fred Brooks in his book 'The Mythical Man-Month'. This is a classic book on the human elements of software project management and provides several insights as to why projects fail. In the following, we discuss why inferring the project progress by measuring effort expended in units of person-month may be fallacious.

The unit of man-month as a measure of work completed in a project may suggest that the number of people working on a project and the duration for which they have worked to be tradable. That is, the progress achieved by deploying 100 persons for one day would be the same as one person working for 100 days. This aspect is expressed by Brooks as follows: 'Cost varies as product of men and months, progress does not.' Brooks concludes that man-month as a unit for measuring the progress achieved in a project is a deceptive myth.

Let us investigate why progress cannot be measured as the product of men and months. Men and months are tradable only when the project tasks can be partitioned equally among all the workers, there are no dependencies among the tasks, and the communication requirement among the workers is negligible. In such projects, the progress can be computed by the number of persons deployed and the number of months for which they have worked. However, this is far from the truth in software development projects. The progress that can be achieved by increasing the number of workers for such a project is shown in Figure 5.2(a). As can be seen in Figure 5.2(a), the completion time steadily decreases as the number of workers increases. However, in practical projects the situation is very different.

In practical projects, there is a substantial communication requirement among the team members, there exist intricate dependencies among various tasks and also for almost any task very fine-grained partitioning is not feasible. For example, consider the task of requirements analysis and requirements specification. It can be meaningfully carried out by four or six analysts. However, if hundreds or thousands of analysts are deployed, then it would not be possible to assign work to many. Also as the tasks are divided into finer subtasks, the communication and management overheads increase drastically. Further, dependencies exist among tasks. For example, design cannot be carried out for a feature for which requirements analysis and specification is not complete. This aspect is shown in Figure 5.2(b), where the completion time initially decreases with increase in the number of workers. However, after some time it actually increases indicating the drastically increased communication and management overheads.

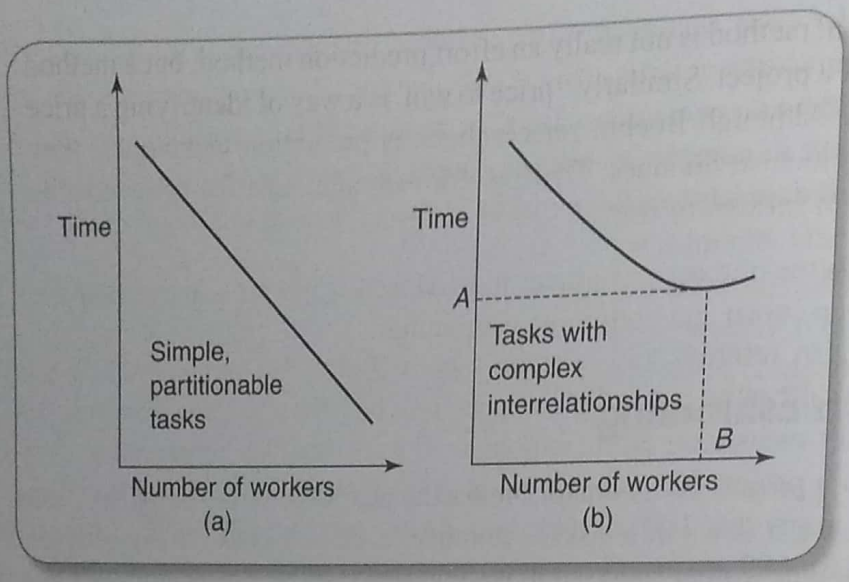

FIGURE 5.2 Impact of addition of workers on the completion time for various types of projects

Software development is an intellectual work, and requires significant amount of communication among the team members. In fact, often the effort in communication dominates the actual work. Communication is made up of two parts as follows: training and intercommunications. Training (technology, goals, overall strategy and work plan) cannot be partitioned and more effort does no good. The intercommunication overhead rises as the number of members in the team increases. Assuming that members talk pairwise, the communication overhead increases as the square of the number of persons in the team. For a team with n number of team members, the number of pairwise communications (meetings involving two persons) is $n(n-1)/2$. Interactions involving more than two people can make the required number of communications even worse. Therefore, in a typical project increasing manpower can become counter-productive after some number of workers. In Figure 5.2(b) observe, that A is the minimum time by which the project can be completed by deploying B number of persons. Increasing the number of deployed persons further, may actually result in increase in project completion time. The above discussions can be extended to explain why adding man power to a late project may make it later.

5.5 Software Effort Estimation Techniques

See B. W. Boehm (1981) *Software Engineering Economics*, Prentice-Hall.

Barry Boehm, in his classic work on software effort models, identified the main ways of deriving estimates of software development effort as:

- *Algorithmic models*, which use 'effort drivers' representing characteristics of the target system and the implementation environment to predict effort
- *Expert judgement*, based on the advice of knowledgeable staff
- *Analogy*, where a similar, completed, project is identified and its actual effort is used as the basis of the estimate
- *Parkinson*, where the staff effort available to do a project becomes the 'estimate'
- *Price to win*, where the 'estimate' is a figure that seems sufficiently low to win a contract
- *Top-down*, where an overall estimate for the whole project is broken down into the effort required for component tasks
- *Bottom-up*, where component tasks are identified and sized and these individual estimates are aggregated

Clearly, the 'Parkinson' method is not really an effort prediction method, but a method of setting the scope of a project. Similarly, 'price to win' is a way of identifying a price and not a prediction. Although Boehm rejects them as prediction techniques, they have value as management techniques. There is, for example, a perfectly acceptable engineering practice of '*design to cost*'.

This is also the principle behind the concept of time-boxing discussed in Chapter 4 in the context of incremental delivery.

We will now look at some of these techniques more closely. First we will examine the difference between top-down and bottom-up estimating.

5.6 Bottom-up Estimating

With the bottom-up approach the estimator breaks the project into its component tasks. With a large project, the process of breaking it down into tasks is iterative: each task is decomposed into its component subtasks and these in turn could be further analysed. It is suggested that this is repeated until you get tasks an individual could do in a week or two. Why is this not a 'top-down approach'? After all, you start from the top and work down. Although this top-down analysis is an essential precursor to bottom-up estimating, it is really a separate process – that of producing a work breakdown schedule (WBS). The bottom-up part comes in adding up the calculated effort for each activity to get an overall estimate.

The bottom-up approach is best at the later, more detailed, stages of project planning. If this method is used earlier, assumptions about the characteristics of the final system and project work methods will have to be made.

Where a project is completely novel or there is no historical data available, the estimator would be forced to use the bottom-up approach.

Exercise 5.4

Brigette at Brightmouth College has been told that there is a requirement, once the payroll system has been successfully installed, to create a subsystem that analyses the staffing costs for each course. Details of the pay that each member of staff receives may be obtained from the payroll standing data. The number of hours that each member of staff spends teaching on each course may be obtained from standing files in a computer-based timetabling system.

What tasks would have to be undertaken to implement this requirement? Try to identify tasks that would take one person about 1 or 2 weeks.

Which tasks are the ones whose durations are most difficult to estimate?

A procedural code-oriented approach

The bottom-up approach described above works at the level of activities. In software development a major activity is writing code. Here we describe how a bottom-up approach can be used at the level of software components.

(a) **Envisage the number and type of software modules in the final system**
Most information systems, for example, are built from a small set of system operations, e.g. Insert, Amend, Update, Display, Delete, Print. The same principle should equally apply to embedded systems, albeit with a different set of primitive functions.

'Software module' here implies a component that can be separately compiled and executed.

(b) **Estimate the SLOC of each identified module**

One way to judge the number of instructions likely to be in a program is to draw up a program structure diagram and to visualize how many instructions would be needed to implement each identified procedure. The estimator may look at existing programs which have a similar functional description to assist in this process.

(c) **Estimate the work content, taking into account complexity and technical difficulty**

The practice is to multiply the SLOC estimate by a factor for complexity and technical difficulty. This factor will depend largely on the subjective judgement of the estimator. For example, the requirement to meet particular highly constrained performance targets can greatly increase programming effort.

(d) **Calculate the work-days effort**

Historical data can be used to provide ratios to convert weighted SLOC to effort.

Note that the above steps can be used to derive an estimate of lines of code that can be used as an input to one of the COCOMO models which are described later.

Exercise 5.5

The IOE annual maintenance contracts subsystem for which Amanda is responsible will have a transaction which sets up details of new annual maintenance contract customers.

The operator will input:

- Customer account number
- Customer name
- Address
- Postcode
- Customer type
- Renewal date

All this information will be set up in a CUSTOMER record on the system's database. If a CUSTOMER account already exists for the account number that has been input, an error message will be displayed to the operator.

Draw up an outline program structure diagram for a program to do the processing described above. For each box on your diagram, estimate the number of lines of code needed to implement the routine in a programming language that you are familiar with, such as Java.

5.7 The Top-down Approach and Parametric Models

The top-down approach is normally associated with *parametric* (or *algorithmic*) *models*. These may be explained using the analogy of estimating the cost of rebuilding a house. This is of practical concern to house-owners who need insurance cover to rebuild their property if destroyed. Unless the house-owner is in the building trade he or she is unlikely to be able to calculate the numbers of bricklayer-hours, carpenter-hours, electrician-hours, and so on, required. Insurance companies, however, produce convenient tables where the

house-owner can find estimates of rebuilding costs based on such *parameters* as the number of storeys and the floor space of a house. This is a simple parametric model.

Project effort relates mainly to variables associated with characteristics of the final system. A parametric model will normally have one or more formulae in the form:

$$effort = (system\ size) \times (productivity\ rate)$$

For example, system size might be in the form 'thousands of lines of code' (KLOC) and have the specific value of 3 KLOC while the productivity rate was 40 days per KLOC. These values will often be matters of judgement.

A model to forecast software development effort therefore has two key components. The first is a method of assessing the amount of the work needed. The second assesses the rate of work at which the task can be done. For example, Amanda at IOE may estimate that the first software module to be constructed is 2 KLOC. She may then judge that if Kate undertook the development of the code, with her expertise she could work at a rate of 40 days per KLOC per day and complete the work in 2×40 days, i.e. 80 days, while Ken, who is less experienced, would need 55 days per KLOC and take 2×55, i.e. 110 days to complete the task. In this case KLOC is a *size driver* indicating the amount of work to be done, while developer experience is a productivity driver influencing the productivity or work rate.

If you have figures for the effort expended on past projects (in work-days for instance) and also the system sizes in KLOC, you should be able to work out a productivity rate as

$$productivity = effort/size$$

A more sophisticated way of doing this would be by using the statistical technique *least squares regression* to derive an equation in the form:

$$effort = constant_1 + (size \times constant_2)$$

Some parametric models, such as that implied by function points, are focused on system or task size, while others, such are COCOMO, are more concerned with productivity factors. Those particular models are described in more detail later in this chapter.

Exercise 5.6

Students on a course are required to produce a written report on an ICT-related topic each semester. If you wanted to create a model to estimate how long it should take a student to complete such an assignment, what measure of work content would you use? Some reports might be more difficult to produce than others: what factors might affect the degree of difficulty?

Having calculated the overall effort required, the problem is then to allocate proportions of that effort to the various activities within that project.

The top-down and bottom-up approaches are not mutually exclusive. Project managers will probably try to get a number of different estimates from different people using different methods. Some parts of an overall estimate could be derived using a top-down approach while other parts could be calculated using a bottom-up method.

At the earlier stages of a project, the top-down approach would tend to be used, while at later stages the bottom-up approach might be preferred.

5.8 Expert Judgement

See R. T. Hughes (1996) 'Expert judgement as an estimating method' Information and *Software Technology* 38(3) 67–75.

This is asking for an estimate of task effort from someone who is knowledgeable about either the application or the development environment. This method is often used when estimating the effort needed to change an existing piece of software. The estimator would have to examine the existing code in order to judge the proportion of code affected and from that derive an estimate. Someone already familiar with the software would be in the best position to do this.

Some have suggested that expert judgement is simply a matter of guessing, but our own research has shown that experts tend to use a combination of an informal analogy approach where similar projects from the past are identified (see below), supplemented by bottom-up estimating.

There may be cases where the opinions of more than one expert may need to be combined. The Delphi technique described in Section 12.3 tackles group decision-making.

5.9 Estimating by Analogy

See M. Shepperd and C. Schofield (1997) 'Estimating software project effort using analogies' *IEEE Transactions in Software Engineering* SE-23(11) 736–43.

This is also called *case-based reasoning*. The estimator identifies completed projects (*source cases*) with similar characteristics to the new project (the *target case*). The effort recorded for the matching source case is then used as a base estimate for the target. The estimator then identifies differences between the target and the source and adjusts the base estimate to produce an estimate for the new project.

This can be a good approach where you have information about some previous projects but not enough to draw generalized conclusions about what might be useful drivers or typical productivity rates.

A problem is identifying the similarities and differences between applications where you have a large number of past projects to analyse. One attempt to automate this selection process is the ANGEL software tool. This identifies the source case that is nearest the target by measuring the Euclidean distance between cases. The Euclidean distance is calculated as:

$$distance = \textit{square-root of } ((target\_parameter_1 - source\_parameter_1)^2 + \ldots (target\_parameter_n - source\_parameter_n)^2)$$

Example 5.1

Say that the cases are being matched on the basis of two parameters, the number of inputs to and the number of outputs from the application to be built. The new project is known to require 7 inputs and 15 outputs. One of the past cases, project A, has 8 inputs and 17 outputs. The Euclidean distance between the source and the target is therefore the square-root of $((7-8)^2 + (17-15)^2)$, that is 2.24.

Exercise 5.7

Project B has 5 inputs and 10 outputs. What would be the Euclidean distance between this project and the target new project being considered above? Is project B a better analogy with the target than project A?

The above explanation is simply to give an idea of how Euclidean distance may be calculated. The ANGEL package uses rather more sophisticated algorithms based on this principle.

5.10 Albrecht Function Point Analysis

This is a top-down method that was devised by Allan Albrecht when he worked for IBM. Albrecht was investigating programming productivity and needed to quantify the functional size of programs independently of their programming languages. He developed the idea of function points (FPs).

See A. J. Albrecht and J. E. Gaffney Jr., 'Software function, source lines of code, and development effort prediction: a software science validation', in M. Shepperd (ed.) (1993) *Software Engineering Metrics* (Vol. 1), McGraw-Hill.

The basis of function point analysis is that information systems comprise five major components, or '*external user types*' in Albrecht's terminology, that are of benefit to the users.

- *External input types* are input transactions which update internal computer files.
- *External output types* are transactions where data is output to the user. Typically these would be printed reports, as screen displays would tend to come under external inquiry types (see below).
- *External inquiry types* – note the US spelling of inquiry – are transactions initiated by the user which provide information but do not update the internal files. The user inputs some information that directs the system to the details required.
- *Logical internal file types* are the standing files used by the system. The term 'file' does not sit easily with modern information systems. It refers to a group of data items that is usually accessed together. It may be made up of one or more *record types*. For example, a purchase order file may be made up of a record type PURCHASE-ORDER plus a second which is repeated for each item ordered on the purchase order – PURCHASE-ORDER-ITEM. In structured systems analysis, a logical internal file would equate to a datastore, while record types would equate to relational tables or entity types.
- *External interface file types* allow for output and input that may pass to and from other computer applications. Examples of this would be the transmission of accounting data from an order processing system to the main ledger system or the production of a file of direct debit details on a magnetic or electronic medium to be passed to the Bankers Automated Clearing System (BACS). Files shared between applications would also be counted here.

Albrecht also dictates that outgoing external interface files should be double counted as logical internal file types as well.

The analyst identifies each instance of each external user type in the application. Each component is then classified as having either high, average or low complexity. The counts of each external user type in each complexity band are multiplied by specified weights (see Table 5.2) to get FP scores which are summed to obtain an overall FP count which indicates the information processing size.

TABLE 5.2 Albrecht complexity multipliers

External user type	Multiplier		
	Low	Average	High
External input type	3	4	6
External output type	4	5	7
External inquiry type	3	4	6
Logical internal file type	7	10	15
External interface file type	5	7	10

Exercise 5.8

The task for which Brigette has been made responsible in Exercise 5.4 needs a program which will extract yearly salaries from the payroll file, and hours taught on each course by each member of staff and the details of courses from two files maintained by the timetabling system. The program will produce a report showing for each course the hours taught by each member of staff and the cost of those hours.

Using the method described above, calculate the Albrecht function points for this subsystem assuming that the report is of high complexity, but that all the other elements are of average complexity.

The International FP User Group (IFPUG) have developed and published extensive rules governing FP counting. Hence Albrecht FPs are now often referred to as IFPUG FPs.

With FPs as originally defined by Albrecht, the question of whether the external user type was of high, low or average complexity was intuitive. The International FP User Group (IFPUG) has now promulgated rules on how this is assessed. For example, in the case of logical internal files and external interface files, the boundaries shown in Table 5.3 are used to decide the complexity level. Similar tables exist for external inputs and outputs.

TABLE 5.3 IFPUG file type complexity

Number of record types	Number of data types		
	< 20	20–50	> 50
1	Low	Low	Average
2 to 5	Low	Average	High
>5	Average	High	High

5.2 Example

A logical internal file might contain data about purchase orders. These purchase orders might be organized into two separate record types: the main PURCHASE-ORDER details, namely purchase order number, supplier reference and purchase order date, and then details for each PURCHASE-ORDER-ITEM specified in the order, namely the product code, the unit price and number ordered. The number of record types for that file would therefore be 2 and the number of data types would be 6. According to Table 5.3, this file type would be rated as 'low'. This would mean that according to Table 5.2, the FP count would be 7 for this file.

Function point analysis recognizes that the effort required to implement a computer-based information system relates not just to the number and complexity of the features provided but also to the operational environment of the system.

Fourteen factors have been identified which can influence the degree of difficulty associated with implementing a system. The list that Albrecht produced related particularly to the concerns of information system developers in the late 1970s and early 1980s. Some technology which was then new and relatively threatening is now well established.

Further details on TCA can be found in the Albrecht and Gaffney paper.

The technical complexity adjustment (TCA) calculation has had many problems. Some have even found that it produces less accurate estimates than using the unadjusted function point count. Because of these difficulties, we omit further discussion of the TCA.

The COCOMO II Model Definition Manual A contains a table of suggested conversion rates and can be downloaded from http://sunset.usc.edu/csse

Tables have been calculated to convert the FPs to lines of code for various languages. For example, it is suggested that 53 lines of Java are needed on average to implement an FP, while for Visual C++ the figure is 34. You can then use historical productivity data to convert the lines of code into an effort estimate, as previously described in Section 5.7.

Exercise 5.9

In the case of the subsystem described in Exercise 5.8 for which Brigette is responsible at Brightmouth HE College, how many lines of Java code should be needed to implement this subsystem, according to the standard conversion? Assuming a productivity rate of 50 lines of code a day, what would be the estimate of effort?

5.11 Function Points Mark II

The Mark II method was originally sponsored by what was then the CCTA (Central Computer and Telecommunications Agency, now the Office of Government Commerce or OGC), which lays down standards for UK government projects. The 'Mark II' label implies an improvement and replacement of the Albrecht method. The Albrecht (now IFPUG) method, however, has had many refinements made to it and FPA Mark II remains a minority method used mainly in the United Kingdom.

This method has came into the public domain with the publication of the book by Charles R. Symons (1991) *Software Sizing and Estimating – Mark II FPA*, John Wiley and Sons.

As with Albrecht, the information processing size is initially measured in unadjusted function points (UFPs) to which a technical complexity adjustment can then be applied (TCA). The assumption is that an information system comprises transactions which have the basic structure shown in Figure 5.3.

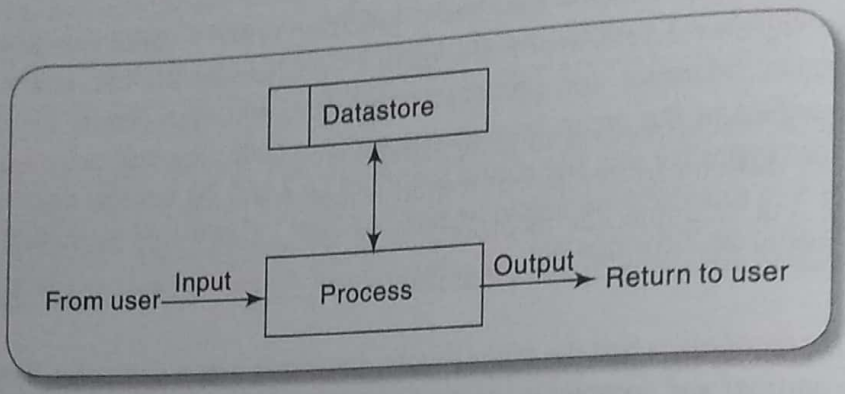

FIGURE 5.3 Model of a transaction

For each transaction the UFPs are calculated:

$W_i \times$ (*number of input data element types*) +

$W_e \times$ (*number of entity types referenced*) +

$W_o \times$ (*number of output data element types*)

W_i, W_e, and W_o are weightings derived by asking developers the proportions of effort spent in previous projects developing the code dealing respectively with inputs, accessing and modifying stored data and processing outputs.

The only reason why 2.5 was adopted here was to produce FP counts similar to the Albrecht equivalents.

The proportions of effort are then normalized into ratios, or weightings, which add up to 2.5. This process for calculating weightings is time consuming and most FP counters use industry averages which are currently 0.58 for W_i, 1 .66 for W_e and 0.26 for W_o.

Example 5.3

A cash receipt transaction in the IOE maintenance accounts subsystem accesses two entity types – INVOICE and CASH-RECEIPT.

The data inputs are:

- Invoice number
- Date received
- Cash received

If an INVOICE record is not found for the invoice number then an error message is issued. If the invoice number is found then a CASH-RECEIPT record is created. The error message is the only output of the transaction. The unadjusted function points, using the industry average weightings, for this transaction would therefore be:

$(0.58 \times 3) + (1.66 \times 2) + (0.26 \times 1) = 5.32$

Exercise 5.10

Calculate the number of unadjusted Mark II function points for the transaction described previously for Exercise 5.5, using the industry average weightings.

Mark II FPs follow the Albrecht method in recognizing that one system delivering the same functionality as another may be more difficult to implement (but also more valuable to the users) because of additional technical requirements. For example, the incorporation of additional security measures would increase the amount of effort to deliver the system. The identification of further factors to suit local circumstances is encouraged.

Symons is very much against the idea of using function points to estimate SLOC rather than effort. One finding by Symons is that productivity, that is, the effort per function point to implement a system, is influenced by the size of the project. In general, larger projects, up to a certain point, are more productive because of economies of scale. However, beyond a certain size they tend to become less productive because of additional management overheads.

Some of the rules and weightings used in FP counting, especially in the case of the Albrecht flavour, are rather arbitrary and have been criticized by academic writers on this account. FPs, however, have been found useful as a way of calculating the price for extensions to existing systems, as will be seen in Chapter 10 on managing contracts.

5.12 COSMIC Full Function Points

While approaches like that of IFPUG are suitable for information systems, they are not helpful when it comes to sizing real-time or embedded applications. This has resulted in the development of another version of function points – the COSMIC FFP method.

COSMIC-FFP stands for Common Software Measurement Consortium – Full Function Points.

The full function point (FFP) method has its origins in the work of two interlinked research groups in Québec, Canada. At the start, the developers were at pains to stress that this method should be seen as simply an extension to the IFPUG method for real-time systems. The original work of FFPs has been taken forward by the formation of the Common Software Measurement Consortium (COSMIC) which has involved not just the original developers in Canada, but others from many parts of the world, including Charles Symons, the originator of Mark II function points. Interestingly, there has been little participation by anyone from the United States.

The argument is that existing function point methods are effective in assessing the work content of information systems where the size of the internal procedures mirrors the number of external features. With a real-time, or embedded, system, its features will be hidden because the software's user will probably not be a human but a hardware device or another software component.

COSMIC deals with this by decomposing the system architecture into a hierarchy of software *layers*. The software component to be sized can receive requests for services from layers above and can request services from those below it. At the same time there could be separate software components at the same level that engage in *peer-to-peer communication*. This identifies the boundary of the software component to be assessed

and thus the points at which it receives inputs and transmits outputs. Inputs and outputs are aggregated into *data groups*, where each group brings together data items that relate to the same object of interest.

Data groups can be moved about in four ways:

- *entries* (E), which are effected by subprocesses that move the data group into the software component in question from a 'user' outside its boundary – this could be from another layer or another separate software component in the same layer via peer-to-peer communication;
- *exits* (X), which are effected by subprocesses that move the data group from the software component to a 'user' outside its boundary;
- *reads* (R), which are data movements that move data groups from persistent storage (such as a database) into the software component;
- *writes* (W), which are data movements that transfer data groups from the software component into persistent storage.

Exercise 5.11

A small computer system controls the entry of vehicles to a car park. Each time a vehicle pulls up before an entry barrier, a sensor notifies the computer system of the vehicle's presence. The system examines a count that it maintains of the number of vehicles that are currently in the car park. This count is kept on backing storage so that it will still be available if the system is temporarily shut down, for example because of a power cut. If the count does not exceed the maximum allowed then the barrier is lifted and the count is incremented. When a vehicle leaves the car park, a sensor detects the exit and reduces the count of vehicles.

There is a system administration system that can set the maximum number of cars allowed, and which can be used to adjust or replace the count of cars when the system is restarted.

Identify the entries, exits, reads and writes in this application.

The overall FFP count is derived by simply adding up the counts for each of the four types of data movement. The resulting units are *Cfsu* (COSMIC functional size units). The method does not take account of any processing of the data groups once they have been moved into the software component. The framers of the method do not recommend its use for systems involving complex mathematical algorithms, for example, but there is provision for the definition of local versions for specialized environments which could incorporate counts of other software features.

The NESMA FP method has been developed by the Netherlands Software Measurement Association.

COSMIC FFPs have been incorporated into an ISO standard – ISO/IEC 19761:2003. Prior to this there were attempts to produce a single ISO standard for 'functional size measurement' and there is an ISO document – ISO/IEC 14143–1:1998 – which lays down some general principles. ISO has decided, diplomatically, that it is unable to judge the relative merits of the four main methods in the field: IFPUG, Mark II, NESMA and COSMIC-FFP, and all four have been allowed to submit their methods as candidates to become ISO standards and then to '*let the market decide*'.

5.13 COCOMO II: A Parametric Productivity Model

Boehm's COCOMO (COnstructive COst MOdel) is often referred to in the literature on software project management, particularly in connection with software estimating. The term COCOMO really refers to a group of models.

Because there is now a newer COCOMO II, the older version is now referred to as COCOMO81.

Boehm originally based his models in the late 1970s on a study of 63 projects. Of these only seven were business systems and so the models could be used with applications other than information systems. The basic model was built around the equation

TABLE 5.4 COCOMO81 constants

System type	c	k
Organic	2.4	1.05
Semi-detached	3.0	1.12
Embedded	3.6	1.20

$$(effort) = c(size)^k$$

where *effort* was measured in *pm* or the number of 'person-months' consisting of units of 152 working hours, *size* was measured in *kdsi*, thousands of delivered source code instructions, and c and k were constants.

The first step was to derive an estimate of the system size in terms of *kdsi*. The constants, c and k (see Table 5.4), depended on whether the system could be classified, in Boehm's terms, as 'organic', 'semi-detached' or 'embedded'. These related to the technical nature of the system and the development environment.

- *Organic mode* This would typically be the case when relatively small teams developed software in a highly familiar in-house environment and when the system being developed was small and the interface requirements were flexible.
- *Embedded mode* This meant that the product being developed had to operate within very tight constraints and changes to the system were very costly.
- *Semi-detached mode* This combined elements of the organic and the embedded modes or had characteristics that came between the two.

Generally, information systems were regarded as organic while real-time systems were embedded.

The exponent value k, when it is greater than 1, means that larger projects are seen as requiring disproportionately more effort than smaller ones. This reflected Boehm's finding that larger projects tended to be less productive than smaller ones because they needed more effort for management and coordination.

Over the years, Barry Boehm and his co-workers have refined a family of cost estimation models of which the key one is COCOMO II. This approach uses various multipliers and exponents the values of which have been set initially by experts. However, a database containing the performance details of executed projects has been built up and periodically analysed so that the expert judgements can be progressively replaced by values derived from actual projects. The new models take into account that there is now a wider range of process models in common use than previously. As we noted earlier, estimates are required at different stages in the system life cycle and COCOMO II has been designed to accommodate this by having models for three different stages.

The detailed COCOMO II Model Definition Manual has been published by the Center for Software Engineering, University of Southern California.

- *Application composition* Here the external features of the system that the users will experience are designed. Prototyping will typically be employed to do this. With small applications that can be built using high-productivity application-building tools, development can stop at this point.
- *Early design* Here the fundamental software structures are designed. With larger, more demanding systems, where, for example, there will be large volumes of transactions and performance is important, careful attention will need to be paid to the architecture to be adopted.
- *Post architecture* Here the software structures undergo final construction, modification and tuning to create a system that will perform as required.

In COCOMO81, estimation was done at the start of the project even before the requirement specification document is fully written. In COCOMO81, the estimates on the average have 50–100 percent error in estimation. In COCOMO II, the estimations can be done with increasing accuracy as the project progresses (see Figure 5.4) through different life cycle phases as the project work advances with time and as more information about the project becomes available. The estimations with increasing accuracy can help the project manager refine the plans and make more accurate plans as the project progresses.

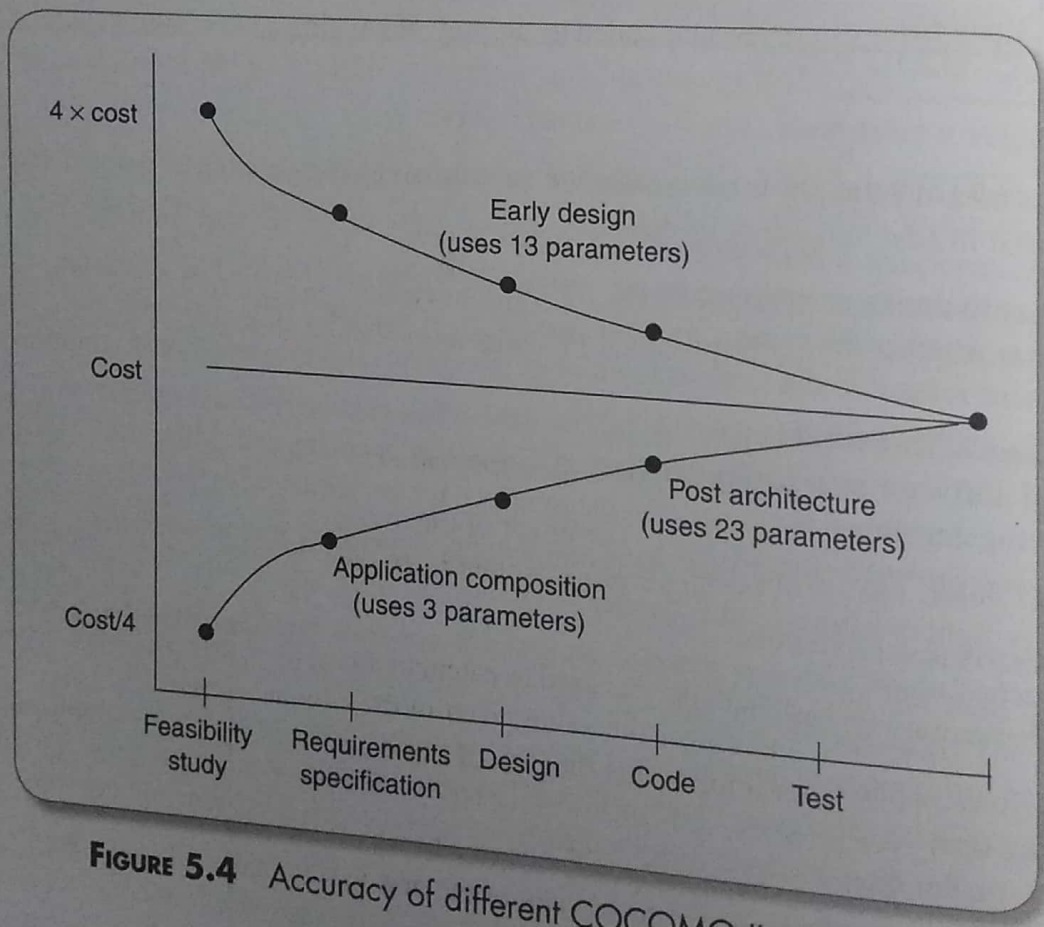

FIGURE 5.4 Accuracy of different COCOMO II estimations

As shown in Figure 5.4, at the feasibility study stage, the project manager makes rough estimates and the estimates can vary significantly often by 400 percent. The actual estimate would lie between the upper bound and lower bounds, which is plotted in Figure 5.4. COCOMO II application composition model can be used during requirements elicitation and prototype construction and gives much more accurate estimation than the rough estimation arrived by the project manager during the feasibility study stage. As shown, in the application composition model, estimates are made based on three parameters that are determined by the project

manager. The early design model can be used after the requirements specification and during the early design stage. This model requires the project manager to determine 13 independent parameters based on which the estimates are made. One of the main parameters to be estimated is the size of the software in function points, which is determined from the requirements document. This model typically produces much more accurate estimations than the early design model. The postarchitecture model is used after the high-level design is complete and requires the project manager to estimate 23 independent parameters based on which the effort and duration estimates are made. This model produces much more accurate estimation than the early design estimates.

To estimate the effort for *application composition*, the counting of *object points* is recommended by the developers of COCOMO II. This follows the function point approach of counting externally apparent features of the software. It differs by focusing on the physical features of the application, such as screens and reports, rather than 'logical' ones such as entity types. This is seen as being more useful where the requirements are being elicited via prototypes.

See R. D. Banker, R. Kauffman and R. Kumar (1992) 'An empirical test of object-based output measurement metrics' *Journal of MIS*, 8(3).

At the *early design* stage, FPs are recommended as the way of gauging a basic system size. An FP count may be converted to an LOC equivalent by multiplying the FPs by a factor for the programming language that is to be used – see Section 5.10.

The following model can then be used to calculate an estimate of person-months.

$$pm = A(size)^{(sf)} \times (em_1) \times (em_2) \times \ldots \times (em_n)$$

where *pm* is the effort in 'person-months', *A* is a constant (which was set in 2000 at 2.94), *size* is measured in *kdsi* (which may have been derived from an FP count as explained above), and *sf* is exponent scale factor.

The scale factor is derived thus:

$$sf = B + 0.01 \times \Sigma \text{ (exponent driver ratings)}$$

where *B* is a constant currently set at 0.91. The effect of the exponent ('. . . to the power of. . .') scale factor is to increase the effort predicted for larger projects, that is, to take account of diseconomies of scale which make larger projects less productive.

The qualities that govern the exponent drivers used to calculate the scale factor are listed below. Note that the less each quality is applicable, the bigger the value given to the exponent driver. The fact that these factors are used to calculate an exponent implies that the lack of these qualities increases the effort required disproportionately more on larger projects.

- *Precedentedness (PREC)* This quality is the degree to which there are precedents or similar past cases for the current project. The greater the novelty of the new system, the more uncertainty there is and the higher the value given to the exponent driver.
- *Development flexibility (FLEX)* This reflects the number of different ways there are of meeting the requirements. The less flexibility there is, the higher the value of the exponent driver.
- *Architecture/risk resolution (RESL)* This reflects the degree of uncertainty about the requirements. If they are liable to change then a high value would be given to this exponent driver.
- *Team cohesion (TEAM)* This reflects the degree to which there is a large dispersed team (perhaps in several countries) as opposed to there being a small tightly knit team.

- *Process maturity (PMAT)* Chapter 13 on software quality explains the process maturity model. The more structured and organized the way the software is produced, the lower the uncertainty and the lower the rating will be for this exponent driver.

Each of the scale factors for a project is rated according to a range of judgements: very low, low, nominal, high, very high, extra high. There is a number related to each rating of the individual scale factors – see Table 5.5. These are summed, then multiplied by 0.01 and added to the constant 0.91 to get the overall exponent scale factor.

TABLE 5.5 COCOMO II scale factor values

Driver	Very low	Low	Nominal	High	Very high	Extra high
PREC	6.20	4.96	3.72	2.48	1.24	0.00
FLEX	5.07	4.05	3.04	2.03	1.01	0.00
RESL	7.07	5.65	4.24	2.83	1.41	0.00
TEAM	5.48	4.38	3.29	2.19	1.10	0.00
PMAT	7.80	6.24	4.68	3.12	1.56	0.00

Exercise 5.12

A new project has 'average' novelty for the software supplier that is going to execute it and is thus given a nominal rating on this account for precedentedness. Development flexibility is high, but requirements may change radically and so the risk resolution exponent is rated very low. The development team are all located in the same office and this leads to team cohesion being rated as very high, but the software house as a whole tends to be very informal in its standards and procedures and the process maturity driver has therefore been given a rating of 'low'.

(i) What would be the scale factor (*sf*) in this case?

(ii) What would be the estimate of effort, if the size of the application was estimated as in the region of 2000 lines of code?

In the COCOMO II model the *effort multipliers* (*em*) adjust the estimate to take account of productivity factors, but do not involve economies or diseconomies of scale. The multipliers relevant to early design are in Table 5.6 and those used at the post architecture stage in Table 5.7. Each of these multipliers may, for a particular application, be given a rating of very low, low, nominal, high or very high. Each rating for each effort multiplier has an associated value. A value greater than 1 increases development effort, while a value less than 1 decreases it. The nominal rating means that the multiplier has no effect. The intention is that the values that these and other ratings use in COCOMO II will be refined over time as actual project details are added to the database.

Table 5.6 COCOMO II early design effort multipliers

Code	Effort modifier	Extra low	Very low	Low	Nominal	High	Very high	Extra high
RCPX	Product reliability and complexity	0.49	0.60	0.83	1.00	1.33	1.91	2.72
RUSE	Required reusability			0.95	1.00	1.07	1.15	1.24
PDIF	Platform difficulty			0.87	1.00	1.29	1.81	2.61
PERS	Personnel capability	2.12	1.62	1.26	1.00	0.83	0.63	0.50
PREX	Personnel experience	1.59	1.33	1.12	1.00	0.87	0.74	0.62
FCIL	Facilities available	1.43	1.30	1.10	1.00	0.87	0.73	0.62
SCED	Schedule pressure		1.43	1.14	1.00	1.00	1.00	

Table 5.7 COCOMO II post architecture effort multipliers

Modifier type	Code	Effort modifier
Product attributes	RELY	Required software reliability
	DATA	Database size
	DOCU	Documentation match to life-cycle needs
	CPLX	Product complexity
	REUSE	Required reusability
Platform attributes	TIME	Execution time constraint
	STOR	Main storage constraint
	PVOL	Platform volatility
Personnel attributes	ACAP	Analyst capabilities
	AEXP	Application experience
	PCAP	Programmer capabilities
	PEXP	Platform experience
	LEXP	Programming language experience
	PCON	Personnel continuity
Project attributes	TOOL	Use of software tools
	SITE	Multisite development
	SCED	Schedule pressure

Example 5.4

Consider a company is developing a software package for an educational institute that would automate various bookkeeping activities associated with the institute's academic activities such as course registration and grading. The institute has already installed other software applications that automate its various activity areas such as stores and purchase, accounting, and faculty pay roll, which are already operational. The size of code that is expected to be written for the academic package is 10,000 SLOC. The package to be developed by a vendor is very similar to a software developed by the same vendor for a different client. The software to be developed needs to seamlessly work with other applications running at the institute and has to use the existing DBMS and other hardware components. The requirements for the academic package are clear and are unlikely to change. The development team put together by the vendor is collocated and cohesive. Other aspects of the project such as required reliability and product complexity, required reusability, platform difficulty, personnel capability, facilities available and schedule pressure are nominal. Determine the effort required by the vendor assuming that it uses ad hoc development practices. Compare the effort to what would be incurred if the vendor had high process maturity.

Solution

From the description of the project, the following values can be assigned to the COCOMO II project parameters: PREC = Very high, FLEX = Low, RESL = High, Team = Very high, and PMAT would be very low for the organization using ad hoc development practices and very high for the organization having high-process maturity.

For the organization with ad hoc process, $sf = 0.91 + 0.01(1.24 + 4.05 + 2.83 + 1.1 + 7.8) = 0.91 + 0.17 = 1.08$

For the organization with mature process, $sf = 0.91 + 0.01(1.24 + 4.05 + 2.83 + 1.1 + 1.56) = 0.91 + 0.1 = 1.01$

From the project characteristics, the effort multipliers are nominal.

The effort required by the ad hoc organization = $2.94 \times (10)1.08 \times 1 = 2.94 \times 12.022 = 35.34$ man months

The effort required by the highly mature organization = $2.94 \times (10)1.01 \times 1 = 2.94 \times 10.23 = 30.08$ man months

Observe according to COCOMO II, the organization using ad hoc practices would require about 18 percent more effort for the project as compared to a mature organization.

Exercise 5.13

A software supplier has to produce an application that controls a piece of equipment in a factory. A high degree of reliability is needed as a malfunction could injure the operators. The algorithms to control the equipment are also complex. The product reliability and complexity are therefore rated as very high. The company would like to take the opportunity to exploit fully the investment that they made in the project by reusing the control system, with suitable modifications, on future contracts. The reusability requirement is therefore rated as very high. Developers are familiar with the platform and the possibility of potential problems in that respect is regarded as low. The current staff are generally very capable and are rated in this respect as very high, but the project is in a somewhat novel application domain for them so experience is rated as nominal. The toolsets available to the developers are judged

to be typical for the size of company and are rated as nominal, as is the degree of schedule pressure to meet a deadline.

Given the data in Table 5.6,

(i) What would be the value for each of the effort multipliers?

(ii) What would be the impact of all the effort multipliers on a project estimated as taking 200 staff-months?

At a later stage of the project, detailed design of the application will have been completed. There will be a clearer idea of application size in terms of lines of code, and the factors influencing productivity will be better known. A revised estimate of effort can be produced based on the broader range of effort modifiers seen in Table 5.7. The method of calculation is the same as for early design. Readers who wish to apply the model using the post architecture effort multipliers are directed to the COCOMO II Model Definition Manual which is available from the University of Southern California website http://sunset.usc.edu/csse/research/COCOMOII/COCOMO_main.html.

5.14 Cost Estimation

Project cost can be obtained by multiplying the estimated effort (in man-month, from the effort estimate) with the manpower cost per month. Implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. However, in addition to manpower cost, a project would incur several other types of costs which we shall refer to as the overhead costs. The overhead costs would include the costs of hardware and software required for the project and the company overheads for administration, office space, etc. Depending on the expected values of the overhead costs, the project manager has to suitably scale up the cost estimated by using the COCOMO formula.

Exercise 5.14

Assume that the size of an organic type software product is estimated to be 32,000 lines of source code. Assume that the average salary of a software developer is £2,000 per month. Determine the effort required to develop the software product, the nominal development time, and the staff cost to develop the product.

5.15 Staffing Pattern

After the effort required to complete a software project has been estimated, the staffing requirement for the project can be determined. Putnam was the first to study the problem of what should be a proper staffing pattern for software projects. He extended the classical work of Norden who had earlier investigated the staffing pattern of general research and development (R&D) type of projects. In order to appreciate the staffing pattern desirable for software projects, we must understand both Norden's and Putnam's results.

Norden's work

Norden studied the staffing patterns of several R&D projects. He found the staffing patterns of R&D projects to be very different from that of manufacturing or sales type of work. In a sales outlet, the number of sales staff

does not usually vary with time. For example, in a supermarket the number of sales personnel would depend on the number of sales counters alone and the number of sales personnel therefore remains fixed for years together. However, the staffing pattern of R&D type of projects changes dynamically over time for efficient manpower utilization. At the start of an R&D project, the activities of the project are planned and initial investigations are made. During this time, the manpower requirements are low. As the project progresses, the manpower requirement increases until it reaches a peak. Thereafter the manpower requirement gradually diminishes. Norden concluded that the staffing pattern for any R&D project can be approximated by the Rayleigh distribution curve shown in Figure 5.5.

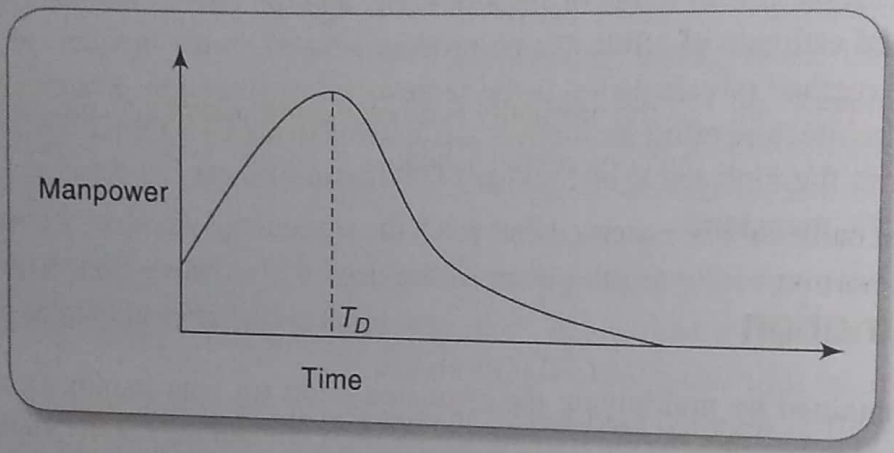

FIGURE 5.5 Rayleigh–Norden Curve

Putnam's work

Norden's work was carried out in the context of general R&D projects. Putnam studied the problem of staffing of software projects and found that the staffing pattern for software development projects has characteristics very similar to R&D projects. Putnam adapted the Rayleigh–Norden curve to relate the number of delivered lines of code to the effort and the time required to develop the product. Only a small number of developers are needed at the beginning of a project to carry out the planning and specification tasks. As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product delivery which has been shown to occur at time T_D in Figure 5.5. After product delivery, the number of project staff falls consistently during product maintenance.

Putnam suggested that starting from a small number of developers, there should be a staff build-up and after a peak size has been achieved, staff reduction is required. However, the staff build-up should not be carried out in large installments. Experience shows that a very rapid build-up of project staff any time during the project development correlates with schedule slippage.

Exercise 5.15

Suppose you are the project manager of a large development project. The top management informs that you would have to manage the project with a fixed team size throughout the duration of your project. What would be the likely impact of this decision on your project?

5.16 Effect of Schedule Compression

It is quite common for a project manager to encounter client requests to deliver products faster, that is, to compress the delivery schedule. It is therefore important to understand the impact of schedule compression on project cost. Putnam studied the effect of schedule compression on the development effort and expressed it in the form of the following equation:

$$pm_{new} = pm_{org} \times \left(\frac{td_{org}}{td_{new}} \right)^4$$

where pm_{new} is the new effort, pm_{org} is the originally estimated effort and td_{org} is the originally estimated time for project completion and td_{new} is the compressed schedule.

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

Boehm arrived at the result that there is a limit beyond which a software project cannot reduce its schedule by buying any more personnel or equipment. This limit occurs roughly at 75% of the nominal time estimate for small and medium sized projects. Thus, if a project manager accepts a customer demand to compress the development schedule of a typical project (medium or small project) by more than 25%, he is very unlikely to succeed. The main reason being, that every project has only a limited amount of activities which can be carried out in parallel, and the sequential activities cannot be speeded up by hiring any number of additional developers.

Exercise 5.16

The nominal effort and duration of a project is estimated to be 1000 pm and 15 months. The project cost is negotiated to be £200,000. This needs the product to be developed and delivered in 12 months time. What is the new cost that needs to be negotiated?

Exercise 5.17

Why does the effort requirement then increase drastically upon schedule compression (as per Putnam's results 16 times for schedule is compressed by 50%)? After all, isn't it the same product that is being developed?

5.17 Capers Jones Estimating Rules of Thumb

Capers Jones published a set of empirical rules in 1996 in the IEEE Computer journal. He formulated these rules based on his experience in estimating various parameters of a large number of software projects. Jones wanted that his rules should be as easy to use as possible, and yet should give the project manager a fairly good idea of various aspects of a project. Because of their simplicity, these rules are handy to use for making off-hand estimates. However, these rules should not be expected to yield very accurate estimations and are certainly not considered appropriate for working out formal cost contracts. Still, while working out formal contracts, these rules are used to carry out sanity checks for estimations arrived using other more rigorous techniques. An interesting aspect of Jones' rules is that these rules give an insight into many aspects of a project (such as the rate of requirements creep) for which no formal methodologies exist as yet. In the following section, we discuss a few of Jones' rules of thumb that are often useful.

Rule 1: *SLOC Function Point Equivalence* One function point = 125 SLOC for C programs.

We have already pointed out in Section 5.4 that the SLOC measure is intuitive and helps in developing a good understanding of the size of a project. SLOC is also used in several popular techniques for estimating several project parameters. However, often the size estimations for a software project are done using the function point analysis due to the inherent advantages of the function point metric. In this situation, it often becomes necessary for the project manager to come up with the SLOC measure for the project from its function point measurements. Jones determined the equivalence between SLOC and function point for several programming languages based on experimental data.

To gain an insight into why SLOC function point equivalence varies across different programming languages let us examine the following. According to Jones, it would take about 320 lines of assembly code to implement one function point. Why does assembly coding take as much as three times the number of instructions required in C language to code one function point? It can be argued that to express one SLOC of C would require several instructions in assembly language.

Rule 2: *Project Duration Estimation* Function points raised to the power 0.4 predicts the approximate development time in calendar months.

To illustrate the applicability of this rule, consider that the size of a project is estimated to be 150 function points (that is, approximately 18,750 SLOC by Rule 1). The development time (time necessary to complete the project) would be about eight months by Rule 2.

Rule 3: *Rate of Requirements Creep* User requirements creep in at an average rate of 2% per month from the design through coding phases.

In almost every project, the features required by the customer keep on increasing due to a variety of reasons. Of course, requirement creeps are normally not expected during project testing and installation stages. Observe that the rule has been carefully worded to take into account the fact that while predicting the total requirements creep, it is necessary to remember that the requirement creeps occur from the end of the requirements phase till the testing phase. Therefore, only an appropriate fraction of the project completion period needs to be considered to exclude the durations of the requirements and testing phases.

Assume that the size of a project is estimated to be 150 function points. Then, the duration for this project can be estimated to be eight months by Jones' Rule 2. Since we need to exclude the duration of requirements specification and testing phase, it is reasonable to assume that the requirements creep would occur for five months only. By Rule 3, the original requirements will grow by a rate of three function points per month. So, the total requirements creep would roughly be 15 function points. Thus, the total size of the project for which the project manager needs to plan would be 165 function points rather than 150 function points.

Rule 4: *Defect Removal Efficiency* Each software review, inspection, or test step will find and remove 30% of the bugs that are present.

This rule succinctly captures the reason why software development organizations use a series of defect removal steps, viz., requirements review, design review, code inspection, and code walk-through, followed by unit, integration, and system testing. In fact, a series of about ten consecutive defect removal operations must be utilized to achieve good product reliability.

Exercise 5.18

For a certain project, consider that in the design document 1000 defects are present at the end of the design stage. Compute how many of these defects would survive after the processes of code review, unit, integration, and system testing have been completed. Assume that the defect removal effectiveness of each error removal stage is 30%.

Rule 5: *Project Manpower Estimation* The size of the software (in function points) divided by 150 predicts the approximate number of the personnel required for developing the application.

To understand the use of this rule, consider a project whose size is estimated to be 500 function points. By Rule 5, the number of development personnel required would be four. Observe that Rule 5 predicts the manpower requirement without considering several other relevant aspects of a project that can significantly affect the required effort. These aspects include the project complexity, the level of usage of CASE tools, and the programming language being used. It is therefore natural to expect that the actual manpower requirement would defer from that predicted by Rule 5. This inaccuracy, however, is in keeping with Jones' objective of having rules that are as simple as possible, so that these can be used off-hand to get a gross understanding of the important project parameters.

Rule 6: *Software Development Effort Estimation* The approximate number of staff months of effort required to develop a software is given by the software development time multiplied with the number of personnel required.

It can be observed that this rule is actually a corollary of the Rules 2 and 5. As an example for application of this rule, consider a project whose size is estimated to be 150 function points. Using Rules 2 and 5, the estimated development effort would be 8 months $\times$ 1 person = 8 person-months.

Rule 7: *Function points divided by 500 predicts the approximate number of personnel required for regular maintenance activities.*

According to Rule 1, 500 function points is equivalent to about 62,500 SLOC for C programs. Thus, we can say that approximately for every 60,000 SLOC, one maintenance personnel would be required to carry out minor bug fixes and functionality adaptations during the operation phase of the software.