23 Software cost estimation

Objectives

The objective of this chapter is to introduce techniques for estimating the cost and effort required for software production. When you have read this chapter, you will:

- understand the fundamentals of software costing and pricing and the complex relationship between them;
- have been introduced to three metrics that are used for software productivity assessment;
- appreciate that a range of different techniques should be used when estimating software costs and schedule;
- understand the principles of the COCOMO 2 model for algorithmic cost estimation.

Contents

23.1 Productivity
23.2 Estimation techniques
23.3 Algorithmic cost modelling
23.4 Project duration and staffing

In Chapter 4, I introduced the project planning process. During that process, a project is split into a number of activities which are enacted in parallel or in sequence. The earlier discussion of project planning concentrated on ways to represent these activities, their dependencies and the allocation of people to carry out these tasks.

In this chapter, I turn to the problem of associating estimates of effort and time with the identified project activities. Project managers must estimate the answers to the following questions:

1. How much effort is required to complete an activity?
2. How much calendar time is needed to complete an activity?
3. What is the total cost of an activity?

Project estimation and project scheduling are carried out together. However, some cost estimation may be required at an early stage of the project before detailed schedules are drawn up. These estimates may be needed to establish a budget for the project or to set a price for the software for a customer.

Once a project is under way, estimates should be updated regularly. This assists with the planning process and allows the effective use of resources. If actual expenditure is significantly greater than the estimates then the project manager must take some action. This may involve applying for additional resources for the project or modifying the work to be done.

There are three parameters involved in computing the total cost of a software development project:

- hardware and software costs, including maintenance;
- travel and training costs;
- effort costs (the costs of paying software engineers).

For most projects, the dominant cost is the effort cost. Computers that are powerful enough for software development are relatively cheap. Although travel costs can be significant where a project is developed at different sites, they are relatively low for most projects. Furthermore, the use of electronic mail, fax and teleconferencing can reduce the travel required.

Effort costs are not simply the costs of the salaries of the software engineers involved in the project. Organisations compute effort costs in terms of overhead costs where they take the total cost of running the organisation and divide this by the number of productive staff. Therefore, the following costs are all part of the total effort cost:

1. costs of providing, heating and lighting office space;
2. costs of support staff such as accountants, secretaries, cleaners and technicians;
3. costs of networking and communications;
4. costs of central facilities such as a library, recreational facilities, etc.;
5. costs of social security and employee benefits such as pensions and health insurance.

Figure 23.1 Factors affecting software pricing

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices may be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

Typically, this overhead factor is somewhere around twice the software engineer's salary depending on the size of the organisation and its associated overheads. Therefore, if a software engineer is paid $90,000 per year, the total cost to the organisation is $180,000 per year or $15,000 per month.

Software costing should be carried out objectively with the aim of accurately predicting the cost to the contractor of developing the software. If the project cost has been computed as part of a project bid to a customer, a decision then has to be made about the price quoted to the customer. Classically, price is simply cost plus profit. However, the relationship between the project cost and the price to the customer is not usually so simple.

Software pricing must take into account broader organisational, economic, political and business considerations. Factors which may be taken into account are shown in Figure 23.1. Therefore, there may not be a simple relationship between the price to the customer for the software and the development costs. Because of the organisational considerations involved, project pricing usually involves senior management in the organisation as well as software project managers.

23.1 Productivity

Productivity in a manufacturing system can be measured by counting the number of units which are produced and dividing this by the number of person-hours required

to produce them. However, for any software problem, there are many different solutions which have different attributes. One solution may execute more efficiently while another may be more readable and easier to maintain. When solutions with different attributes are produced, comparing their production rates is not really meaningful.

Nevertheless, the productivity of engineers in the software development process may have to be estimated by managers. These estimates may be needed for project estimation and to assess whether process or technology improvements are effective. These productivity estimates are usually based on measuring some attributes of the software and dividing this by the total effort required for development. There are two types of measure which have been used:

1. *Size-related measures* These are related to the size of some output from an activity. The most common size-related measure is lines of delivered source code. Other measures which may be used are the number of delivered object code instructions or the number of pages of system documentation.

2. *Function-related measures* These are related to the overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function points and object points are the best known measures of this type.

Lines of source code per programmer-month is a widely-used metric in productivity measurement. This is computed by counting the total number of lines of source code which are delivered. The count is divided by the total time in programmer-months required to complete the project. This time therefore includes the time required for analysis and design, coding, testing and documentation.

This approach was first developed when most programming was in FORTRAN, assembly language or COBOL. Then, programs were typed on cards, with one statement on each card. The number of lines of code was easy to compute. It corresponded to the number of cards in the program deck. However, programs in languages like Java or C++ consist of declarations, executable statements, and commentary. They may include macro instructions that expand to several lines of code. There may be more than one statement per line. There is not, therefore, a simple relationship between program statements and lines on a listing.

Some line counting techniques consider executable statements only; others count executable statements and data declarations; some count each non-blank line in the program, irrespective of what is on that line. Standards for line counting in different languages have been proposed (Park, 1992) but these are not widely known. Productivity comparisons across organisations are impossible unless each organisation uses the same method for counting lines of code.

Comparing productivity across different programming languages can also give misleading impressions of programmer productivity. The more expressive the programming language, the lower the apparent productivity. This anomaly arises because all software development activities are considered together when computing productivity but the metric used only applies to the programming process.

Figure 23.2 System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	8 weeks	6 weeks	2 weeks

	Size	Effort	Productivity
Assembly code	5000 lines	28 weeks	714 lines/month
High-level language	1500 lines	20 weeks	300 lines/month

For example, consider a system which might be coded in 5000 lines of assembly code or 1500 lines of high-level language code. The development time for the various phases is shown in Figure 23.2. The assembler programmer has a productivity of 714 lines/month and the high-level language programmer less than half of this, 300 lines/month. Yet the development costs for the high-level language system are lower and it is produced in less time.

An alternative to using code size as the estimated product attribute is to use some measure of the functionality of the code. This avoids the above anomaly as functionality is independent of implementation language. MacDonell (1994) briefly describes and compares several different function-based measures.

The best-known of these measures is the function-point count. This was proposed by Albrecht (1979) and refined by Albrecht and Gaffney (1983). Function points are language-independent so productivity in different programming languages can be compared. Productivity is expressed as function points produced per person-month. Function points are biased towards data-processing systems which are dominated by input and output operations. A function point is not a single characteristic but is a combination of program characteristics. The total number of function points in a program is computed by measuring or estimating the following program features:

- external inputs and outputs
- user interactions
- external interfaces
- files used by the system.

Each of these is individually assessed for complexity and given a weighting value that varies from 3 (for simple external inputs) to 15 for complex internal files. Either the weighting values proposed by Albrecht may be used or values based on local experience.

The unadjusted function-point count (UFC) is computed by multiplying each raw count by the estimated weight and summing all values.

$$UFC = \sum(\text{number of elements of given type}) \times (\text{weight})$$

This initial function-point count is then further modified by factors whose value is based on the overall complexity of the project. This takes into account the degree of distributed processing, the amount of reuse, the performance, etc. The unadjusted function-point count is multiplied by the project complexity factors to produce a final function-point count.

Symons (1988) notes that the subjective nature of complexity estimates means that the function-point count in a program depends on the estimator. Different people have different notions of complexity. There are wide variations in function-point count depending on the estimator's judgement. For this reason, there are differing views about the value of function points (Furey and Kitchenham, 1997). However, users argue that, in spite of their flaws, they are effective in practical situations (Kemerer, 1993).

Object points (Banker *et al.*, 1992) are an alternative to function points when 4GLs or comparable languages are used for software development. Object points are used in the COCOMO 2 estimation model discussed later in this chapter. Object points are not object classes that may be produced when an object-oriented approach is taken to software development. Rather, the number of object points in a program is a weighted estimate of:

1. The number of separate screens that are displayed. Simple screens count as 1 object point, moderately complex screens count as 2 and very complex screens count as 3 object points.
2. The number of reports that are produced. For simple reports, count 2 object points, for moderately complex reports, count 5 and for reports which are likely to be difficult to produce, count 8 object points.
3. The number of 3GL modules that must be developed to supplement the 4GL code. Each 3GL module counts as 10 object points.

The advantage of using object points rather than function points is that they are easier to estimate from a high-level software specification. Object points are only concerned with screens, reports and 3GL modules.

If function points or object points are used they can be estimated at an early stage in the development process. Estimates of these parameters can be made as soon as the external interactions of the system have been designed. At this stage, it is very difficult to produce an accurate estimate of the size of a program in lines of source code. Indeed, the programming language which is to be used may not even have been decided. Early estimates are essential when using the algorithmic cost estimation models that are discussed later in this chapter.

Function-point counts can be used in conjunction with lines of code estimation techniques. The number of function points is used to estimate the final code size. Using historical data analysis, the average number of lines of code, AVC, in a particular language required to implement a function point can be estimated. The estimated code size for a new application is computed as follows:

$$\text{Code size} = \text{AVC} \times \text{Number of function points}$$

Figure 23.3 Factors affecting software engineering productivity

Factor	Description
Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 25.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, supportive configuration management systems, etc. can improve productivity.
Working environment	As discussed in Chapter 22, a quiet working environment with private work areas contributes to improved productivity.

Values of AVC vary from 200–300 LOC/FP in assembly language to 2–40 LOC/FP for a fourth-generation language.

The productivity of individual engineers working in an organisation is affected by a number of factors. Some of the most important of these are summarised in Figure 23.3. However, individual differences in ability are more significant than any of these factors. In an early assessment of productivity, Sackman *et al.* (1968) found that some programmers were more than 10 times more productive than others. My experience is that this is still true. Large teams are likely to have a mix of abilities so will have 'average' productivity. In small teams, however, overall productivity is mostly dependent on individual aptitudes and abilities.

There is no such thing as an 'average' value for productivity that applies across application domains and organisations. For large, complex embedded systems, productivity may be as low as 30 lines/programmer-month. For straightforward, well-understood application systems it may be as high as 900 lines/month. When measured in terms of object points, Boehm *et al.* (1995) suggest that productivity varies from four object points per month to 50 object points/month depending on tool support and developer capability.

The problem with measures expressed as volume/time is that they take no account of non-functional software characteristics such as reliability, maintainability, etc. They imply that more always means better. Beck (2000) makes the excellent point that if you have an approach that is based around continuous code simplification and improvement then counting lines of code doesn't mean much.

These measures also do not take into account the possibility of reusing the software produced. What we really want to estimate is the cost of deriving a particular system with given functionality, quality, performance, maintainability, etc. This is only indirectly related to tangible measures such as the system size.

This becomes a particular problem if managers use productivity measurements to judge the abilities of staff. In such situations, engineers may compromise on quality in order to become more 'productive'. It may be the case that the 'less productive' programmer produces more reliable code which is easier to understand and cheaper to maintain. Productivity measures must therefore be used only as a guide. They should not be used without careful analysis.

23.2 Estimation techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system. Initial estimates may have to be made on the basis of a high-level user requirements definition. The software may have to run on unfamiliar computers or use new development technology. The people involved in the project and their skills will probably not be known. All of these factors mean that it is difficult to produce an accurate estimate of system development costs at an early stage in the project.

Furthermore, there is a fundamental difficulty in assessing the accuracy of different approaches to cost estimation techniques. Project cost estimates are often self-fulfilling. The estimate is used to define the project budget and the product is adjusted so that the budget figure is realised. I do not know of reports of controlled experiments with project costing where the estimated costs were not used to bias the experiment. A controlled experiment would not reveal the cost estimate to the project manager. The actual costs would then be compared with the estimated project costs.

Nevertheless, organisations need to make software effort and cost estimates. To do so, one or more of the techniques described in Figure 23.4 may be used (Boehm, 1981).

Hihn and Habib-agahi (1991) describe an experiment where they asked managers to estimate the size of a software system to be developed and the effort required. The managers used expert judgement and estimation by analogy. It was found that they were reasonably accurate in estimating required effort but their estimates of code size were much less accurate. This means that cost estimates based on a code size estimate were also inaccurate.

These approaches to cost estimation can be tackled using either a top-down or a bottom-up approach. A top-down approach starts at the system level. The estimator starts by examining the overall functionality of the product and how that functionality is provided by interacting sub-functions. The costs of system-level activities such as integration, configuration management and documentation are taken into account.

The bottom-up approach, by contrast, starts at the component level. The system is decomposed into components and the effort required to develop each of these

Figure 23.4 Cost estimation techniques

Technique	Description
Algorithmic cost modelling	A model is developed using historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.
Expert judgement	Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.
Estimation by analogy	This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (1989) gives a very clear description of this approach.
Parkinson's Law	Parkinson's Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and five people are available, the effort required is estimated to be 60 person-months.
Pricing to win	The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer's budget and not on the software functionality.

is computed. These costs are then added to give the effort required for the whole system development.

The disadvantages of the top-down approach are the advantages of the bottom-up approach and vice versa. Top-down estimation can underestimate the costs of solving difficult technical problems associated with specific components such as interfaces to non-standard hardware. There is no detailed justification of the estimate that is produced. By contrast, bottom-up estimation produces such a justification and considers each component. However, this approach is more likely to underestimate the costs of system activities such as integration. Bottom-up estimation is also more expensive. There must be an initial system design to identify the components to be costed.

Each estimation technique has its own strengths and weaknesses. For large projects, you should use several cost estimation techniques and compare their results. If these predict radically different costs, this suggests that you do not have enough costing information. You should look for more information and repeat the costing process until the estimates converge.

These estimation techniques are applicable where a requirements document for the system has been produced. This should define all users and system requirements. You can therefore make a reasonable estimate of the extent of the system functionality which is to be developed. In general, large systems engineering projects will normally have such a requirements document.

However, in many cases, the costs of many projects must be estimated using only an outline of the user requirements for the system. This means that the estimators have very little information to work with. Requirements analysis and specification is expensive and the managers in a company may need an initial cost estimate for the system before they can have a budget approved for this process.

Under these circumstances, 'pricing to win' is a commonly used strategy. The notion of 'pricing to win' may seem unethical and unbusinesslike. However, it does have some advantages. A project cost is agreed on the basis of an outline proposal. Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality. The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the cost is not exceeded.

When estimating software costs, managers must take into account that there may be important differences between past projects and future projects. A range of new development methods and techniques have been introduced in the last 10 years. Many managers have little experience of these techniques or knowledge of how they affect project costs. Some examples of the changes which may affect estimates based on experience include:

- object-oriented development rather than function-oriented development;
- client–server systems rather than mainframe-based systems;
- use of off-the-shelf software components rather than component development;
- development for and with reuse rather than new development of all parts of a system;
- the use of CASE tools and program generators rather than unsupported software development.

All of these factors make it more difficult for managers to produce accurate estimates of the costs of software production. Their previous experience may not be relevant in helping them estimate software project costs.

23.3 Algorithmic cost modelling

The most systematic, although not necessarily the most accurate, approach to software estimation is algorithmic cost estimation. An algorithmic cost model can be built by analysing the costs and attributes of completed projects. A mathematical formula is used to predict costs based on estimates of project size, number of programmers and other process and product factors. Kitchenham (1990) describes 13 algorithmic cost models that have been developed from empirical observations.

Most algorithmic estimation models have an exponential component. This reflects the fact that costs do not normally increase linearly with project size. As the size of the software increases, extra costs are incurred because of the communication overhead of larger teams, more complex configuration management, more difficult system integration, etc. They may also include multipliers reflecting the attributes of the product being developed, the development platform, the process and the software developers.

In its most general form, an algorithmic cost estimate for software cost can be expressed as:

$$\text{Effort} = A \times \text{Size}^B \times M$$

A is a constant factor which depends on local organisational practices and the type of software that is developed. Size may be either an assessment of the code size of the software or a functionality estimate expressed in function or object points. The value of exponent B usually lies between 1 and 1.5. It reflects the disproportionate effort required for large projects. M is a multiplier made up by combining different process, product and development attributes.

All algorithmic models suffer from the same basic difficulties:

1. It is often difficult to estimate Size at an early stage in a project where only a specification is available. Function point and object point estimates are easier to produce than estimates of code size but may still be inaccurate.
2. The estimates of the factors contributing to B and M are subjective. Estimates vary from one person to another depending on their background and experience.

The number of lines of source code in the finished system is the basic metric used in most algorithmic cost models. Size estimation may involve estimation by analogy with other projects, estimation by converting function points to code size, estimation by ranking the sizes of system components and using a known reference component to estimate the component size or may simply be a question of engineering judgement.

Design decisions, which may not be known when the initial estimates are made, influence the size of the final system. For example, an application which requires complex data management may either use a commercial database or implement its own data manager. If a commercial database is used, the code size will be smaller. The programming language used is also significant. A language like Java might mean that more lines of code are necessary than if C (say) were used. However, this extra code allows more compile-time checking so validation costs are likely to be reduced. How should this be taken into account? Furthermore, the extent of reuse in the software development process must also be estimated and the size estimate adjusted to take this into account.

If algorithmic models are used for project costing, they should be calibrated using data for the type of project being estimated and their outputs must be carefully

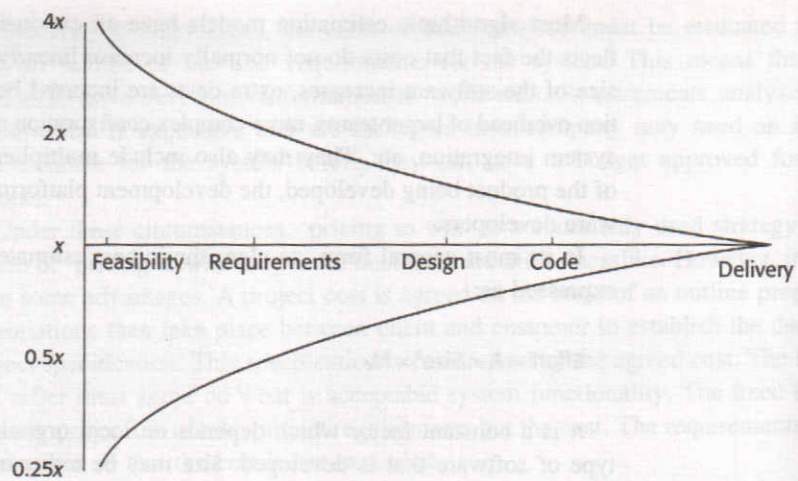

Figure 23.5 Estimate uncertainty

intepreted. The estimator should develop a range of estimates (worst, expected and best) rather than a single estimate. The costing formula should be applied to all of these. Errors in initial estimates are likely to be significant. Estimates are most likely to be accurate when the product is well understood, the model has been calibrated for the organisation using it, and language and hardware choices are predefined.

The accuracy of the estimates produced by an algorithmic model depends on the system information that is available. As the software process proceeds, more information becomes available so estimates become more and more accurate. If the initial estimate of effort required is x months of effort, this range may be from $0.25x$ to $4x$ when the system is first proposed. This narrows during the development process as shown in Figure 23.5. This figure, adapted from Boehm's paper (Boehm *et al.*, 1995), reflects experience of a large number of software development projects.

23.3.1 The COCOMO model

There are a number of algorithmic models that have been proposed as a basis for estimating the effort, schedule and costs of a software project. These are conceptually similar but use different parameter values. The specific model which I discuss here is the COCOMO model. The COCOMO model is an empirical model. It was derived by collecting data from a large number of software projects, then analysing that data to discover formulae that were the best-fit to the observations. I have chosen COCOMO for the following reasons:

1. It is well documented, in the public domain and is supported by public domain and commercial tools.
2. It has been widely used and evaluated.

Project complexity	Formula	Description
Simple	PM = 2.4 $(KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	PM = 3.0 $(KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	PM = 3.6 $(KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

Figure 23.6 The basic COCOMO 81 model

3. It has a long pedigree from its first instantiation in 1981 (Boehm, 1981), through a refinement tailored to Ada software development (Boehm and Royce, 1989) to its most recent version, published in 1995 (Boehm *et al.*, 1995).

The first version of the COCOMO model (now known as COCOMO 81) was a three-level model where the levels reflected the detail of the analysis of the cost estimate. The first level (basic) provided an initial, rough estimate, the second level modified this using a number of project and process multipliers and the most detailed level produced estimates for different phases of the project. Figure 23.6 shows the basic COCOMO formula for different types of project. The multiplier M is similar to that discussed below for COCOMO 2.

COCOMO 81 assumed that the software would be developed according to a waterfall process and that the vast majority of the software would be developed from scratch. However, there have been radical changes to software development since this initial version was proposed. Software may be created by assembling reusable components and linking them via some scripting language. Prototyping and incremental development are commonly used process models. In many cases, off-the-shelf sub-systems are used. Existing software is re-engineered to create new software. CASE tool support for most software process activities is now available.

To take these changes into account, the COCOMO 2 model recognises different approaches to software development such as prototyping, development by component composition, use of 4GLs, etc. The model levels do not simply reflect increasingly complex and detailed estimates. Levels are associated with activities in the software process so that initial estimates may be made early in the process with more detailed estimating carried out after the system architecture has been defined. The levels identified in COCOMO 2 are:

1. *The early prototyping level* Size estimates are based on object points and a simple size/productivity formula is used to estimate the effort required.
2. *The early design level* This level corresponds to the completion of the system requirements with (perhaps) some initial design. Estimates are based on function points which are then converted to number of lines of source code. The formula follows the standard form discussed above with a simple set of multipliers associated with it.

Figure 23.7 Object point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

3. *The post-architecture level* Once the system architecture has been designed a reasonably accurate estimate of the software size can be made. The estimate at this level uses a more extensive set of multipliers reflecting personnel capability, product and project characteristics.

The early prototyping level

The early prototyping level was introduced into COCOMO to support the estimation of effort required for prototyping projects and for projects where the software was developed by composing existing components. It is based on an estimate of weighted object points (discussed in section 23.1) that is divided by a standard figure for estimated productivity. Programmer productivity depends on the developer's experience and capability and the capabilities of the CASE tools used to support development. Figure 23.7 shows the different levels of productivity suggested by the developers of the model (Boehm *et al.*, 1995).

At this level, reuse is common so the number of object points used in the schedule estimate is adjusted to take into account the percentage of reuse (%reuse) that is expected. Therefore, the final formula for schedule computation is:

$$PM = (NOP \times (1 - \%reuse/100))/PROD$$

PM is the effort in person-months, NOP is the number of object points and PROD is the productivity as shown in Figure 23.7.

The early design level

The estimates produced at this stage are based on the standard formula for algorithmic models, namely

$$Effort = A \times Size^{B} \times M$$

Based on his own large data-set, Boehm proposes that the coefficient A should be 2.5 for estimates made at this level. The size of the system is expressed in KSLOC, namely the number of thousands of lines of source code. This is computed by estimating the number of function points in the software and converting this to KSLOC using standard tables which relate software size to function points for different programming languages. This size estimate refers to the code that is implemented manually rather than generated or reused.

The exponent B reflects the increased effort required as the size of the project increases. This is not fixed as in the first version of COCOMO but can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team and the process maturity level (see Chapter 25) of the organisation. The way in which this exponent is computed is discussed in the following section.

The multiplier M is based on a simplified set of seven project and process drivers which include product reliability and complexity (RCPX), reuse required (RUSE), platform difficulty (PDIF), personnel capability (PERS), personnel experience (PREX), schedule (SCED) and support facilities (FCIL). These can be estimated directly on a six-point scale where 1 corresponds to very low values for these multipliers and 6 corresponds to very high values. Alternatively, they can be computed by combining values of the more detailed multipliers used in the post-architecture level.

This results in an effort computation as follows:

$$PM = A \times Size^{B} \times M + PM_m \text{ where}$$

$$M = PERS \times RCPX \times RUSE \times PDIF \times PREX \times FCIL \times SCED$$

The last term in the formula (PM_m) is a factor that is used when a significant percentage of the code is generated automatically. Some manual input is usually needed to tailor this code but the productivity level is much higher than for manually created code. Therefore, the effort required (PM_m) is computed separately using the following formula and then added to the effort computed for manually developed code.

$$PM_m = (ASLOC \times (AT/100))/ATPROD$$

ASLOC is the number of automatically generated lines of source code and ATPROD is the productivity level for this type of code production. However, some effort is required to interface generated code with the remainder of the system. This depends on the percentage of the total system code which is automatically generated (AT). The actual productivity therefore depends on the number of modules which are generated. The smaller the amount of generated code, the greater the overhead involved in integrating this with other code in the system.

The post-architecture level

The estimates produced at the post-architecture level are based on the same basic formula that is used in the early design estimates. However, the size estimate for the software should be more accurate by this stage in the estimation process and 17 rather than seven attributes are used to refine the initial effort computation.

The estimate of the total number of lines of source code is adjusted to take two important project factors into account:

- *The requirements volatility* An estimate is made of the rework which may be required to accommodate changes to the system requirements. This estimate is expressed as the number of lines of source code which must be modified and this number is added to the initial size estimate.
- *The extent of possible reuse* Extensive reuse means that the number of lines of source code which will actually be developed must be amended. However, Selby (1988) discovered that reuse costs are non-linear because of the initial effort required to discover and select components and because of the effort required to understand reusable components and their interfaces if they need to be modified.

The effects of reuse are taken into account in COCOMO 2 by adjusting the size effort according to the following formula:

$$\text{ESLOC} = \text{ASLOC} \times (\text{AA} + \text{SU} + 0.4\text{DM} + 0.3\text{CM} + 0.3\text{IM})/100$$

where ESLOC is equivalent number of lines of new code, ASLOC is the number of lines of reusable code which must be modified, DM is the percentage of design modified, CM is the percentage of the code that is modified and IM is the percentage of the original integration effort required for integrating the reused software. SU is a factor based on the cost of software understanding which ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code. AA is a factor which reflects the initial assessment costs of deciding if software may be reused. It depends on the amount of testing and evaluation that is required. Its value varies from 0 to 8.

The exponent term in the effort computation formula had three possible values in COCOMO 1. These were related to different levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. However, organisational practices and procedures can control this 'diseconomy of scale' and this is recognised in COCOMO 2. The exponent is estimated by considering five scale factors as shown in Figure 23.8. These factors are rated on a six-point scale from Very low to Extra high (5 to 0). The resulting ratings are added, divided by 100 and the result is added to 1.01 to give the exponent term.

To illustrate this, imagine that an organisation is taking on a project in a domain where it has little previous experience. The project client has not defined the process to be used and has not allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organisation has recently put a process improvement programme in place and has been rated as a Level 2 organisation according to the CMM model (see Chapter 25). Possible values for the ratings used in exponent calculation are:

1. *Precedentedness* This is a new project for the organisation – rated Low (4)
2. *Development flexibility* No client involvement – rated Very high (1)
3. *Architecture/risk resolution* No risk analysis carried out – rated Very low (5)

Figure 23.8 Scale factors used in the COCOMO 2 exponent computation

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience; Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client sets only general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; Extra high means a complete and thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions; Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

4. *Team cohesion* New team so no information – rated Nominal (3)
5. *Process maturity* Some process control in place – rated Nominal (3)

The sum of these values is 16 so the resulting exponent is 1.17.

The attributes (Figure 23.9) that are used to adjust the initial estimates in the post-architecture model fall into four classes:

1. Product attributes are concerned with required characteristics of the software product being developed.
2. Computer attributes are constraints imposed on the software by the hardware platform.
3. Personnel attributes are multipliers that take the experience and capabilities of the people working on the project into account.
4. Project attributes are concerned with the particular characteristics of the software development project.

Figure 23.10 shows an example of how these cost drivers influence effort estimates. I have taken a value for the exponent of 1.17 as discussed in the above example and I assume that RELY, CPLX, STOR, TOOL and SCED are the key cost drivers in the project. All of the other cost drivers have a nominal value of 1 so do not affect the computation of the effort.

Figure 23.9 Project cost drivers

Product attributes			
RELY	Required system reliability	DATA	Size of database used
CPLX	Complexity of system modules	RUSE	Required percentage of reusable components
DOCU	Extent of documentation required		
Computer attributes			
TIME	Execution time constraints	STOR	Memory constraints
PVOL	Volatility of development platform		
Personnel attributes			
ACAP	Capability of project analysts	PCAP	Programmer capability
PCON	Personnel continuity	AEXP	Analyst experience in project domain
PEXP	Programmer experience in project domain	LTEX	Language and tool experience
Project attributes			
TOOL	Use of software tools	SITE	Extent of multi-site working and quality of site communications
SCED	Development schedule compression		

Figure 23.10 The effect of cost drivers on effort estimates

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

In the example, I have assigned maximum and minimum values to the key cost drivers to show how they influence the effort estimate. The values taken are those from the COCOMO 2 reference manual (Boehm, 1997). You can see that high values for the cost drivers lead to an effort estimate that is more than three times the initial estimate whereas low values reduce the estimate to about 1/3 of the original.

This highlights the vast differences between different types of project and the difficulties of transferring experience from one application domain to another.

This formula proposed by the developers of the COCOMO 2 model reflects their experience and data but, in my view, it seems too complex for practical use. There are too many attributes and too much scope for uncertainty in estimating their values. In principle, each user of the model should calibrate the model and the attribute values according to its own historical project data as this will reflect local circumstances which affect the model. In practice, however, few organisations have collected enough data from past projects in a form that supports model calibration. Practical use of COCOMO 2 would normally start with the published values for the model parameters.

23.3.2 Algorithmic cost models in project planning

Project managers can use an algorithmic cost model to compare different ways of investing money to reduce project costs. This is particularly important where there must be hardware/software cost trade-offs and where there may be a need to recruit new staff with specific project skills.

Consider an embedded system to control an experiment that is to be launched into space. Spaceborne experiments have to be very reliable and are subject to stringent weight limits. The number of chips on a circuit board may have to be minimised. In terms of the COCOMO model, the multipliers based on computer constraints and reliability are greater than 1.

There are three components to be taken into account in costing this project:

1. The cost of the target hardware to execute the system.
2. The cost of the platform (computer plus software) to develop the system.
3. The cost of the effort required to develop the software.

Figure 23.11 shows some possible options that may be considered. These include spending more on target hardware to reduce software costs or investing in better development tools.

Additional hardware costs may be acceptable in this case because the system is a specialised system that does not have to be mass-produced. If hardware is embedded in consumer products, however, investing in target hardware to reduce software costs is rarely acceptable because it increases the unit cost of the product.

Figure 23.12 shows the hardware, software and total costs for the options A–F shown in Figure 23.11. Applying the COCOMO 2 model without cost drivers predicts an effort of 45 p.m. to develop an embedded software system for this application. The average cost for one person-month of effort is $15,000.

The relevant multipliers are based on storage and execution time constraints (TIME and STOR), the availability of tool support (cross-compilers, etc.) for the development system (TOOL) and development team's experience platform experience

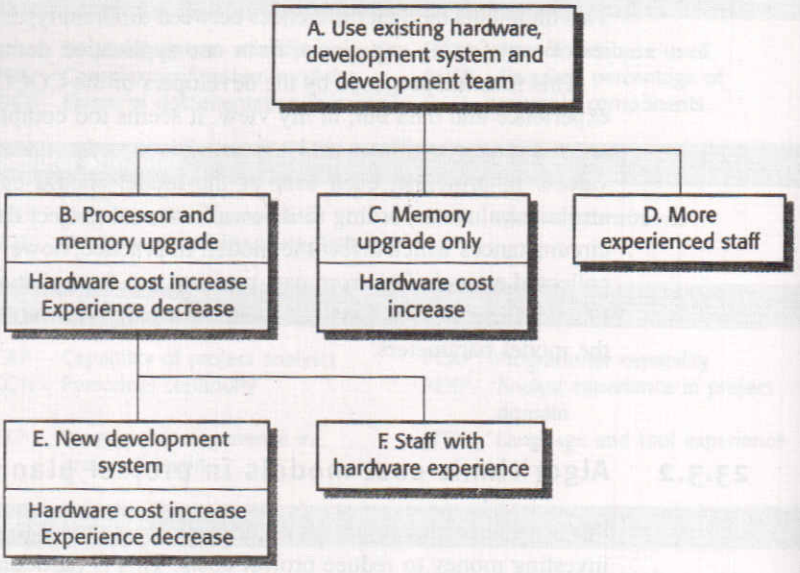

Figure 23.11
Management options

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949,393	100,000	1,049,393
B	1.39	1	1	1.12	1.22	88	1,313,550	120,000	1,402,025
C	1.39	1	1.11	0.86	1	60	895,653	105,000	1,000,653
D	*1.39*	*1.06*	*1.11*	*0.86*	*0.84*	*51*	*769,008*	*100,000*	*897,490*
E	1.39	1	1	0.72	1.22	56	844,425	220,000	1,044,159
F	1.39	1	1	1.12	0.84	57	851,180	120,000	1,002,706

Figure 23.12 Costs of management options

(LTEX). In all options, the reliability multiplier (RELY) is 1.39, indicating that significant extra effort is needed to develop a reliable system.

The software cost (SC) is computed as follows:

$$SC = \text{Effort estimate} \times RELY \times TIME \times STOR \times TOOL \times EXP \times \$15{,}000$$

Option A represents the cost of building the system with existing support and staff. It represents a baseline for comparison. All other options involve either more hardware expenditure or the recruitment (with associated costs and risks) of new staff. Option B shows that upgrading hardware does not necessarily reduce

costs because the experience multiplier is more significant. It is actually more cost-effective to upgrade memory rather than the whole computer configuration.

Option D appears to offer the lowest costs for all basic estimates. No additional hardware expenditure is involved but new staff must be recruited onto the project. If these are already available in the company, this is probably the best option to choose. If not, they must be recruited externally and this involves significant costs and risks. These may mean that the cost advantages of this option are much less significant than suggested by Figure 23.12. Option C offers a saving of almost $50,000 with virtually no associated risk. Conservative project managers would probably select this option rather than the riskier Option D.

The comparisons show the importance of staff experience as a multiplier. If good quality people with the right experience are recruited, this can significantly reduce project costs. This is consistent with the discussion of productivity factors in section 23.1. It also reveals that investment in new hardware and tools may not be cost-effective. Such strategies are often promoted by developers who like to work with new systems. However, the loss of experience is a more significant effect on the system cost than the savings which come from the new hardware system.

23.4 Project duration and staffing

As well as estimating the effort required to develop a software system and the overall cost of that effort, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project. The development time for the project is called the project schedule. Increasingly, organisations are demanding shorter development schedules so that their products can be brought to market before their competitor's.

The relationship between the number of staff working on a project, the total effort required and the development time is not linear. As the number of staff increases, more effort may be needed. People must spend more time communicating. More time is required to define interfaces between the parts of the system. Doubling the number of staff (for example) does not mean that the duration of the project will be halved.

The COCOMO model includes a formula to estimate the calendar time (TDEV) required to complete a project. The time computation formula is the same for all COCOMO levels:

$$TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$$

PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project.

However, the predicted project schedule and the schedule which is required by the project plan are not necessarily the same thing. The planned schedule may be shorter or longer than the nominal predicted schedule. However, there is obviously a limit to the extent of schedule changes and this is predicted by the COCOMO 2 model:

$$\text{TDEV} = 3 \times (\text{PM})^{(0.33+0.2*(\text{B}-1.01))} \times \text{SCEDPercentage}/100$$

SCEDPercentage is the percentage increase or decrease in the nominal schedule. If the predicted figure then differs significantly from the planned schedule, it suggests that there is a high risk that there will be problems delivering the software as planned.

To illustrate the COCOMO development schedule computation, assume that a software project has been estimated to require 60 months of development effort (Option C in Figure 23.12). Assume that the value of exponent B is 1.17.

From the schedule equation, the time required to complete the project is:

$$\text{TDEV} = 3(60)^{0.36} = 13 \text{ months}$$

In this case, there is no schedule compression or expansion so the last term in the formula has no effect on the computation.

An interesting implication of the COCOMO model is that the time required to complete the project is a function of the total effort required for the project. It does not depend on the number of software engineers working on the project. This confirms the notion that adding more people to a project that is behind schedule is unlikely to help that schedule to be regained. Myers (1989) discusses the problems of schedule acceleration. He suggests that projects are likely to run into significant problems if they try to develop software without allowing sufficient calendar time.

Dividing the effort required on a project by the development schedule does not give a useful indication of the number of people required for the project team. Generally, the number of people employed on a software project builds up from a relatively small number to a peak and then declines.

Only a small number of people are needed at the beginning of a project to carry out planning and specification. As the project progresses and more detailed work is required, the number of staff builds up to a peak. After implementation and unit testing is complete, the number of staff required starts to fall until it reaches one or two when the product is delivered. A very rapid build-up of project staff often correlates with project schedule slippage. Project managers should therefore avoid adding too many staff to a project early in its lifetime.

The effort build-up can be modelled by what is called a Rayleigh curve (Londeix, 1987) and Putnam's estimation model (Putnam, 1978) incorporates a model of project staffing based around these curves. Putnam's model also includes development time as a key factor. As development time is reduced, the effort required to develop the system grows exponentially.

KEY POINTS

- Factors that affect productivity include individual aptitude (the dominant factor), domain experience, the development process, the size of the project, tool support and the working environment.
- There are various techniques of software cost estimation. In preparing an estimate, several of these should be used. If the estimates diverge widely, this means that inadequate estimating information is available.
- Software is often priced to gain a contract and the functionality of the system is then adjusted to meet the estimated price.
- Algorithmic cost modelling suffers from the fundamental difficulty that it relies on attributes of the finished product to make the cost estimate. At early stages of the project, these attributes are impossible to estimate accurately.
- The COCOMO costing model is a well-developed model that takes project, product, hardware and personnel attributes into account when formulating a cost estimate. It also includes a means of estimating development schedules.
- Algorithmic cost models are valuable to management as they support quantitative option analysis. They allow the cost of various options to be computed and, even with errors, the options can be compared on an objective basis.
- The time required to complete a project is not simply proportional to the number of people working on the project. Adding more people to a late project can make it later.

FURTHER READING

Software Project Management: Readings and Cases. A selection of papers and case studies on software project management that is particularly strong in its coverage of algorithmic cost modelling. (C. F. Kemerer (ed.), 1997, Irwin.)

'Cost models for future software life cycle processes: COCOMO 2'. A comprehensive introduction to the COCOMO 2 cost estimation model which includes the rationale for the formulae used. (B. Boehm *et al.*, *Annals of Software Engineering*, 1, Balzer Science Publishers, 1995.)

'Nine management guidelines for better cost estimating'. Sound practical advice on cost estimation. (A. Lederer and J. Pasad, *Comm. ACM*, 35(2), February 1992.)

EXERCISES

23.1 Describe two metrics that have been used to measure programmer productivity. Comment briefly on the advantages and disadvantages of these metrics.

23.2 In the development of large, embedded real-time systems, suggest five factors which are likely to have a significant effect on the productivity of the software development team.

23.3 Cost estimates are inherently risky irrespective of the estimation technique used. Suggest four ways in which the risk in a cost estimate can be reduced.

23.4 A software manager is in charge of the development of a safety-critical software system which is designed to control a radiotherapy machine to treat patients suffering from cancer. This system is embedded in the machine and must run on a special-purpose processor with a fixed amount of memory (8 Mbytes). The machine communicates with a patient database system to obtain the details of the patient and, after treatment, automatically records the radiation dose delivered and other treatment details in the database.

The COCOMO method is used to estimate the effort required to develop this system and an estimate of 26 person-months is computed. All cost driver multipliers were set to 1 when making this estimate.

Explain why this estimate should be adjusted to take project, personnel, product and organisational factors into account. Suggest four factors that might have significant effects on the initial COCOMO estimate and propose possible values for these factors. Justify why you have included each factor.

23.5 Give three reasons why algorithmic cost estimates prepared in different organisations are not directly comparable.

23.6 Explain how the algorithmic approach to cost estimation may be used by project managers for option analysis. Suggest a situation where managers may choose an approach that is not based on the lowest project cost.

23.7 Implement the COCOMO model using a spreadsheet such as Microsoft Excel. Details of the model can be downloaded from the COCOMO 2 web site. I have included a link to this site in the book's web pages.

23.8 Some very large software projects involve the writing of millions of lines of code. Suggest how useful the cost estimation models are likely to be for such systems. Why might the assumptions on which they are based be invalid for very large software systems?

23.9 Is it ethical for a company to quote a low price for a software contract knowing that the requirements are ambiguous and that they can charge a high price for subsequent changes requested by the customer?

23.10 Should measured productivity be used by managers during the staff appraisal process? What safeguards are necessary to ensure that quality is not affected by this?