

26

Software cost estimation

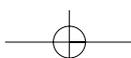
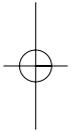
Objectives

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 reasons why the price of the software may not be directly related to its development cost;
- have been introduced to three metrics that are used for software productivity assessment;
- appreciate why a range of techniques should be used when estimating software costs and schedule;
- understand the principles of the COCOMO II model for algorithmic cost estimation.

Contents

- 26.1** Software productivity
- 26.2** Estimation techniques
- 26.3** Algorithmic cost modelling
- 26.4** Project duration and staffing



In Chapter 5, I introduced the project planning process where the work in a project is split into a number of separate activities. This 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 project activities. Estimation involves answering the following questions:

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

Project cost estimation and project scheduling are normally carried out together. The costs of development are primarily the costs of the effort involved, so the effort computation is used in both the cost and the schedule estimate. However, you may have to do some cost estimation before detailed schedules are drawn up. These initial estimates may be used to establish a budget for the project or to set a price for the software for a customer.

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 extensive travel costs may be needed when a project is developed at different sites, the travel costs are usually a small fraction of the effort costs. Furthermore, using electronic communications systems such as e-mail, shared web sites and videoconferencing can significantly reduce the travel required. Electronic conferencing also means that travelling time is reduced and time can be used more productively in software development. In one project where I worked, making every other meeting a videoconference rather than a face-to-face meeting reduced travel costs and time by almost 50%.

Effort costs are not just the salaries of the software engineers who are 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, administrators, system managers, cleaners and technicians
3. Costs of networking and communications

4. Costs of central facilities such as a library or recreational facilities
5. Costs of Social Security and employee benefits such as pensions and health insurance.

This overhead factor is usually at least twice the software engineer's salary, depending on the size of the organisation and its associated overheads. Therefore, if a company pays a software engineer \$90,000 per year, its total costs are at least \$180,000 per year or \$15,000 per month.

Once a project is underway, project managers should regularly update their cost and schedule estimates. This helps with the planning process and 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.

Software costing should be carried out objectively with the aim of accurately predicting the cost 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, such as those shown in Figure 26.1. 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 should involve senior management (i.e., those who can make strategic decisions), as well as software project managers.

For example, say a small oil services software company employs 10 engineers at the beginning of a year, but only has contracts in place that require 5 members of the development staff. However, it is Bidding for a very large contract with a major oil company that requires 30 person years of effort over 2 years. The project will not start up for at least 12 months but, if granted, it will transform the finances of the small company. The oil services company gets an opportunity to bid on a project that requires 6 people and has to be completed in 10 months. The costs (including overheads of this project) are estimated at \$1.2 million. However, to improve its competitive position, the oil services company bids a price to the customer of \$0.8 million. This means that, although it loses money on this contract, it can retain specialist staff for more profitable future projects.

26.1 Software productivity

You can measure productivity in a manufacturing system by counting the number of units that are produced and dividing this by the number of person-hours required

Figure 26.1 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 organisation the opportunity to make a greater profit later. The experience gained may also help it develop new products.
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 can 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.

to produce them. However, for any software problem, there are many different solutions, each of which has 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, as a project manager, you may be faced with the problem of estimating the productivity of software engineers. You may need these productivity estimates to help define the project cost or schedule, to inform investment decisions or to assess whether process or technology improvements are effective.

Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development. There are two types of metric that have been used:

1. *Size-related metrics* These are related to the size of some output from an activity. The most commonly used size-related metric is lines of delivered source code. Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.
2. *Function-related metrics* 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 metrics of this type.

Lines of source code per programmer-month (LOC/pm) is a widely used software productivity metric. You can compute LOC/pm by counting the total number of lines of source code that are delivered, then divide the count by the total time in programmer-months required to complete the project. This time therefore includes the time required for all other activities (requirements, design, coding, testing and documentation) involved in software development.

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 count: It corresponded to the number of cards in the program deck. However, programs in languages such as 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.

Comparing productivity across 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 the development time, but the LOC metric applies only to the programming process. Therefore, if one language requires more lines than another to implement the same functionality, productivity estimates will be anomalous.

For example, consider an embedded real-time system that might be coded in 5,000 lines of assembly code or 1,500 lines of C. The development time for the various phases is shown in Figure 26.2. 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 system developed in C are lower and it is delivered earlier.

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 (MacDonell, 1994) briefly describes and compares several function-based measures. The best known of these measures is the function-point count. This was proposed by Albrecht (Albrecht, 1979) and refined by Albrecht and Gaffney (Albrecht and Gaffney, 1983). Garmus and Herron (Garmus and Herron, 2000) describe the practical use of function points in software projects.

Productivity is expressed as the number of function points that are implemented per person-month. A function point is not a single characteristic but is computed by combining several different measurements or estimates. You compute the total number of function points in a program by measuring or estimating the following program features:

Figure 26.2 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	4 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		

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

Obviously, some inputs and outputs, interactions, and so on are more complex than others and take longer to implement. The function-point metric takes this into account by multiplying the initial function-point estimate by a complexity-weighting factor. You should assess each of these features for complexity and then assign the weighting factor that varies from 3 (for simple external inputs) to 15 for complex internal files. Either the weighting values proposed by Albrecht or values based on local experience may be used.

You can then compute the so-called unadjusted function-point count (UFC) by multiplying each initial count by the estimated weight and summing all values.

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

You then modify this unadjusted function-point count by additional complexity factors that are related to the complexity of the system as a whole. This takes into account the degree of distributed processing, the amount of reuse, the performance, and so on. The unadjusted function-point count is multiplied by these project complexity factors to produce a final function-point count for the overall system.

Symons (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 therefore wide variations in function-point count depending on the estimator's judgement and the type of system being developed. Furthermore, function points are biased towards data-processing systems that are dominated by input and output operations. It is harder to estimate function-point counts for event-driven systems. For this reason, some people think that function points are not a very useful way to measure software productivity (Furey and Kitchenham, 1997; Armour, 2002). However, users of function points argue that,

in spite of their flaws, they are effective in practical situations (Banker, et al., 1993; Garmus and Herron, 2000).

Object points (Banker, et al., 1994) are an alternative to function points. They can be used with languages such as database programming languages or scripting languages. 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 that are likely to be difficult to produce, count 8 object points.
3. *The number of modules in imperative programming languages such as Java or C++ that must be developed to supplement the database programming code* Each of these modules counts as 10 object points.

Object points are used in the COCOMO II estimation model (where they are called application points) that I cover later in this chapter. The advantage of object points over function points is that they are easier to estimate from a high-level software specification. Object points are only concerned with screens, reports and modules in conventional programming languages. They are not concerned with implementation details, and the complexity factor estimation is much simpler.

If function points or object points are used, they can be estimated at an early stage in the development process before decisions that affect the program size have been made. 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.

Function-point and object-point counts can be used in conjunction with lines of code-estimation models. The final code size is calculated from the number of function points. 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. Values of AVC vary from 200 to 300 LOC/FP in assembly language to 2 to 40 LOC/FP for a database programming language such as SQL. The estimated code size for a new application is then computed as follows:

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

The programming productivity of individuals working in an organisation is affected by a number of factors. Some of the most important of these are summarised in Figure 26.3. However, individual differences in ability are usually more significant than any of these factors. In an early assessment of productivity, Sackman et al. (Sackman, et al., 1968) found that some programmers were more than 10 times

Figure 26.3 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 28.
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 and configuration management systems can improve productivity.
Working environment	As I discussed in Chapter 25, a quiet working environment with private work areas contributes to improved productivity.

more productive than others. My experience is that this is still true. Large teams are likely to have a mix of abilities and experience and so will have ‘average’ productivity. In small teams, however, overall productivity is mostly dependent on individual aptitudes and abilities.

Software development productivity varies dramatically across application domains and organisations. For large, complex, embedded systems, productivity has been estimated to be as low as 30 LOC/pm. For straightforward, well-understood application systems, written in a language such as Java, it may be as high as 900 LOC/pm. When measured in terms of object points, Boehm et al. (Boehm, et al., 1995) suggest that productivity varies from 4 object points per month to 50 per month, depending on the type of application, tool support and developer capability.

The problem with measures that rely on the amount produced in a given time period is that they take no account of quality characteristics such as reliability and maintainability. They imply that more always means better. Beck (Beck, 2000), in his discussion of extreme programming, makes an excellent point about estimation. If your approach is based on 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, using code generators and other tools that help create the software. What we really want to estimate is the cost of deriving a particular system with given functionality, quality, performance, maintainability, and so on. This is only indirectly related to tangible measures such as the system size.

As a manager, you should not use productivity measurements to make hasty judgements about the abilities of the engineers on your team. If you do, 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—code that is easier to understand and cheaper to maintain. You should always, therefore, think of productivity measures as providing partial information about programmer productivity. You also need to consider other information about the quality of the programs that are produced.

26.2 Estimation techniques

There is no simple way to make an accurate estimate of the effort required to develop a software system. You may have to make initial estimates 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 mean that it is impossible to estimate system development costs accurately at an early stage in a 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 any 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. However, such an experiment is probably impossible because of the high costs involved and the number of variables that cannot be controlled.

Nevertheless, organisations need to make software effort and cost estimates. To do so, one or more of the techniques described in Figure 26.4 may be used (Boehm, 1981). All of these techniques rely on experience-based judgements by project managers who use their knowledge of previous projects to arrive at an estimate of the resources required for the project. However, there may be important differences between past and future projects. Many new development methods and techniques have been introduced in the last 10 years. Some examples of the changes that may affect estimates based on experience include:

1. Distributed object systems rather than mainframe-based systems
2. Use of web services
3. Use of ERP or database-centred systems
4. Use of off-the-shelf software rather than original system development
5. Development for and with reuse rather than new development of all parts of a system

Figure 26.4 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 (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 5 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.

6. Development using scripting languages such as TCL or Perl (Ousterhout, 1998)
7. The use of CASE tools and program generators rather than unsupported software development.

If project managers have not worked with these techniques, their previous experience may not help them estimate software project costs. This makes it more difficult for them to produce accurate costs and schedule estimates.

You can tackle the approaches to cost estimation shown in Figure 26.4 using either a top-down or a bottom-up approach. A top-down approach starts at the system level. You start 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 you estimate the effort required to develop each of these components. You then add these component costs to compute 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 nonstandard 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. Each uses different information about the project and the development team, so if you use a single model and this information is not accurate, your final estimate will be wrong. For large projects, therefore, you should use several cost estimation techniques and compare their results. If these predict radically different costs, you probably do not have enough information about the product or the development process. You should look for more information about the product, process or team 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 system functionality that is to be developed. In general, large systems engineering projects will have such a requirements document.

However, in many cases, the costs of many projects must be estimated using only incomplete user requirements for the system. This means that the estimators have very little information with which to work. 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 to develop more detailed requirements or a system prototype.

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.

For example, say a company is bidding for a contract to develop a new fuel delivery system for an oil company that schedules deliveries of fuel to its service stations. There is no detailed requirements document for this system so the developers estimate that a price of \$900,000 is likely to be competitive and within the oil company’s budget. After they are granted the contract, they negotiate the detailed requirements of the system so that basic functionality is delivered; then they estimate the additional costs for other requirements. The oil company does not necessarily lose here because it has awarded the contract to a company that it can trust. The additional requirements may be funded from a future budget, so that the oil company’s budgeting is not disrupted by a very high initial software cost.

26.3 Algorithmic cost modelling

Algorithmic cost modelling uses a mathematical formula to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors. An algorithmic cost model can be built by analysing the costs and attributes of completed projects and finding the closest fit formula to actual experience.

Algorithmic cost models are primarily used to make estimates of software development costs, but Boehm (Boehm, et al., 2000) discusses a range of other uses for algorithmic cost estimates, including estimates for investors in software companies, estimates of alternative strategies to help assess risks, and estimates to inform decisions about reuse, redevelopment or outsourcing.

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 that 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. M is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team

Most algorithmic estimation models have an exponential component (B in the above equation) that is associated with the size estimate. 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, and so on. Therefore, the larger the system, the larger the value of this exponent.

Unfortunately, all algorithmic models suffer from the same fundamental difficulties:

1. *It is often difficult to estimate Size at an early stage in a project when only a specification is available.* Function-point and object-point estimates are easier to produce than estimates of code size but are often still 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 with the type of system that is being developed.

The number of lines of source code in the delivered system is the basic metric used in many algorithmic cost models. Size estimation may involve estimation by analogy with other projects, estimation by converting function or object points to

code size, estimation by ranking the sizes of system components and using a known reference component to estimate the component size, or it may simply be a question of engineering judgement.

Accurate code size estimation is difficult at an early stage in a project because the code size is affected by design decisions that have not yet been made. For example, an application that requires complex data management may either use a commercial database or implement its own data-management system. If a commercial database is used, the code size will be smaller but additional effort may be needed to overcome the performance limitations of the commercial product.

The programming language used for system development also affects the number of lines of code to be developed. A language such as 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, it may be possible to reuse a significant amount of code from previous projects and the size estimate has to be adjusted to take this into account.

If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected and best) rather than a single estimate and apply the costing formula to all of them. Estimates are most likely to be accurate when you understand the type of software that is being developed, when you have calibrated the costing model using local data, and when programming language and hardware choices are predefined.

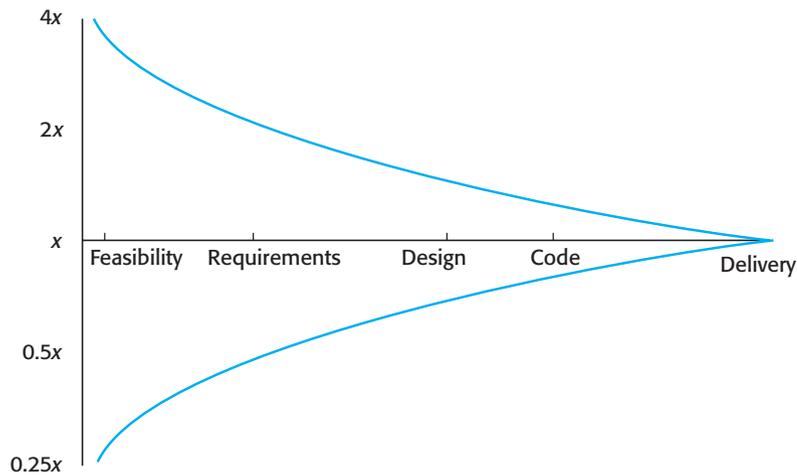
The accuracy of the estimates produced by an algorithmic model depends on the system information that is available. As the software process proceeds, more information becomes available so estimates become more and more accurate. If the initial estimate of effort required is x months of effort, this range may be from $0.25x$ to $4x$ when the system is first proposed. This narrows during the development process, as shown in Figure 26.5. This figure, adapted from Boehm's paper (Boehm, et al., 1995), reflects experience of a large number of software development projects. Of course, just before the system is delivered, a very accurate estimate can be made.

26.3.1 The COCOMO model

A number of algorithmic models have been proposed as the basis for estimating the effort, schedule and costs of a software project. These are conceptually similar but use different parameter values. The model that I discuss here is the COCOMO model. The COCOMO model is an empirical model that was derived by collecting data from a large number of software projects. These data were analysed to discover formulae that were the best fit to the observations. These formulae link the size of the system and product, project and team factors to the effort to develop the system.

I have chosen to use the COCOMO model for several reasons:

Figure 26.5 Estimate uncertainty



1. It is well documented, available in the public domain and supported by public domain and commercial tools.
2. It has been widely used and evaluated in a range of organisations.
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, COCOMO II, published in 2000 (Boehm, et al., 2000).

The COCOMO models are comprehensive, with a large number of parameters that can each take a range of values. They are so complex that I cannot give a complete description here. Rather, I simply discuss their essential characteristics to give you a basic understanding of algorithmic cost models.

The first version of the COCOMO model (COCOMO 81) was a three-level model where the levels corresponded to 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 26.6 shows the basic COCOMO formula for different types of projects. The multiplier M reflects product, project and team characteristics.

COCOMO 81 assumed that the software would be developed according to a waterfall process (see Chapter 4) using standard imperative programming languages such as C or FORTRAN. However, there have been radical changes to software development since this initial version was proposed. Prototyping and incremental development are commonly used process models. Software is now often developed by

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 26.6 The basic COCOMO 81 model

assembling reusable components with off-the-shelf systems and ‘gluing’ them together with scripting language. Data-intensive systems are developed using a database programming language such as SQL and a commercial database management system. 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 II model recognises different approaches to software development such as prototyping, development by component composition and use of database programming. COCOMO II supports a spiral model of development (see Chapter 4) and embeds several sub-models that produce increasingly detailed estimates. These can be used in successive rounds of the development spiral. Figure 26.7 shows COCOMO II sub-models and where they are used.

The sub-models that are part of the COCOMO II model are:

1. *An application-composition model* This assumes that systems are created from reusable components, scripting or database programming. It is designed to make estimates of prototype development. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required. Application points are the same as object points discussed in Section 26.1, but the name was changed to avoid confusion with objects in object-oriented development.
2. *An early design model* This model is used during early stages of the system design after the requirements have been established. 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 simplified set of seven multipliers.
3. *A reuse model* This model is used to compute the effort required to integrate reusable components and/or program code that is automatically generated by design or program translation tools. It is usually used in conjunction with the post-architecture model.
4. *A post-architecture model* Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again this model uses

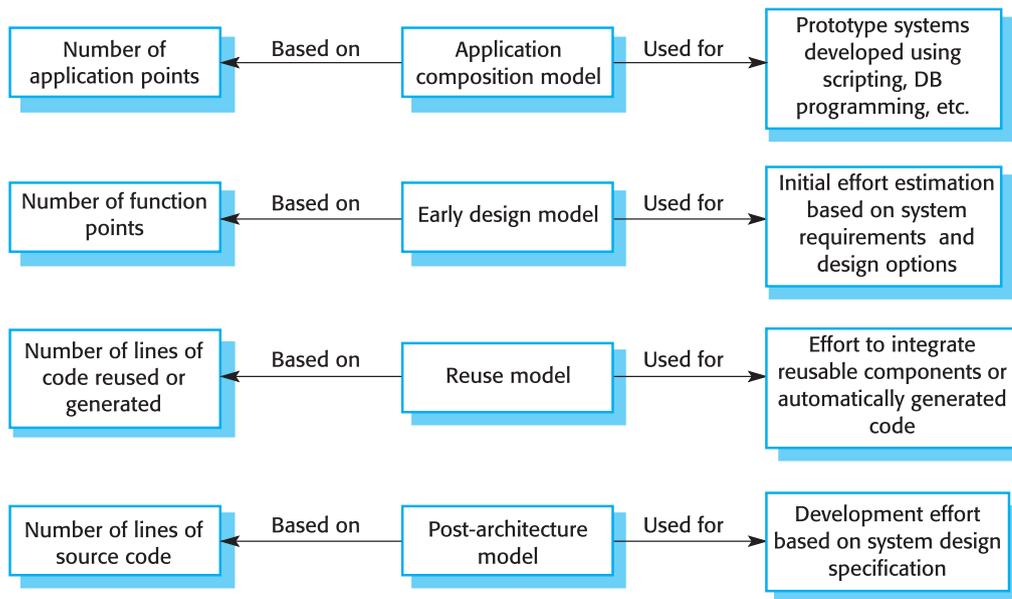


Figure 26.7 The COCOMO II models

the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability and product and project characteristics.

Of course, in large systems, different parts may be developed using different technologies, and you may not have to estimate all parts of the system to the same level of accuracy. In such cases, you can use the appropriate sub-model for each part of the system and combine the results to create a composite estimate.

The application-composition model

The application-composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components. It is based on an estimate of weighted application points (object points) divided by a standard estimate of application-point productivity. The estimate is then adjusted according to the difficulty of developing each object point (Boehm, et al., 2000). Programmer productivity also depends on the developer's experience and capability as well as the capabilities of the CASE tools used to support development. Figure 26.8 shows the levels of object-point productivity suggested by the model developers (Boehm, et al., 1995).

Application composition usually involves significant software reuse, and some of the total number of application points in the system may be implemented with reusable components. Consequently, you have to adjust the estimate based on the

Figure 26.8 Object-point productivity

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

total number of application points to take into account the percentage of reuse expected. Therefore, the final formula for effort computation for system prototypes is:

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

PM is the effort estimate in person-months. NAP is the total number of application points in the delivered system. %reuse is an estimate of the amount of reused code in the development. PROD is the object-point productivity as shown in Figure 26.8. The model simplistically assumes that there is no additional effort involved in reuse.

The early design model

This model is used once user requirements have been agreed and initial stages of the system design process are underway. However, you don't need a detailed architectural design to make these initial estimates. Your goal at this stage should be to make an approximate estimate without undue effort. Consequently, you make various simplifying assumptions, such as that the effort involved in integrating reusable code is zero. Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.

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

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

Based on his own large data set, Boehm proposes that the coefficient A should be 2.94. The size of the system is expressed in KSLOC, which is the number of thousands of lines of source code. You calculate KSLOC by estimating the number of function points in the software. You then use standard tables that relate software size to function points for different programming languages to compute an initial estimate of the system size in KSLOC.

The exponent B reflects the increased effort required as the size of the project increases. This is not fixed for different types of systems, as in COCOMO 81, 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 28) of the organisation. I discuss how

the value of this exponent is calculated using these parameters in the description of the COCOMO II post-architecture model.

The multiplier M in COCOMO II is based on a simplified set of seven project and process characteristics that influence the estimate. These can increase or decrease the effort required. These characteristics used in the early design model are product reliability and complexity (RCPX), reuse required (RUSE), platform difficulty (PDIF), personnel capability (PERS), personnel experience (PREX), schedule (SCED) and support facilities (FCIL). You estimate values for these attributes using a six-point scale where 1 corresponds to very low values for these multipliers and 6 corresponds to very high values.

This results in an effort computation as follows:

$$PM = 2.94 \times \text{Size}^B \times M$$

where:

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

The reuse model

As I have discussed in Chapters 18 and 19, software reuse is now common, and most large systems include a significant percentage of code that is reused from previous developments. The reuse model is used to estimate the effort required to integrate reusable or generated code.

COCOMO II considers reused code to be of two types. Black-box code is code that can be reused without understanding the code or making changes to it. The development effort for black-box code is taken to be zero. Code that has to be adapted to integrate it with new code or other reused components is called white-box code. Some development effort is required to reuse this because it has to be understood and modified before it can work correctly in the system.

In addition, many systems include automatically generated code from program translators that generate code from system models. This is a form of reuse where standard templates are embedded in the generator. The system model is analysed, and code based on these standard templates with additional details from the system model is generated. The COCOMO II reuse model includes a separate model to estimate the costs associated with this generated code.

For code that is automatically generated, the model estimates the number of person months required to integrate this code. The formula for effort estimation is:

$$PM_{\text{Auto}} = (\text{ASLOC} \times \text{AT}/100) / \text{ATPROD} \quad // \text{ Estimate for generated code}$$

AT is the percentage of adapted code that is automatically generated and $ATPROD$ is the productivity of engineers in integrating such code. Boehm et al. (Boehm, et al., 2000) have measured $ATPROD$ to be about 2,400 source statements per month. Therefore, if there is a total of 20,000 lines of white-box reused code in a system

and 30% of this is automatically generated, then the effort required to integrate this generated code is:

$$(20,000 \times 30/100) / 2400 = 2.5 \text{ person months} \quad // \text{Generated code example}$$

The other component of the reuse model is used when a system includes some new code and some reused white-box components that have to be integrated. In this case, the reuse model does not compute the effort directly. Rather, based on the number of lines of code that are reused, it calculates a figure that represents the equivalent number of lines of new code.

Therefore, if 30,000 lines of code are to be reused, the new equivalent size estimate might be 6,000. Essentially, reusing 30,000 lines of code is taken to be equivalent to writing 6,000 lines of new code. This calculated figure is added to the number of lines of new code to be developed in the COCOMO II post-architecture model.

The estimates in this reuse model are:

ASLOC—the number of lines of code in the components that have to be adapted;
ESLOC—the equivalent number of lines of new source code.

The formula used to compute **ESLOC** takes into account the effort required for software understanding, for making changes to the reused code and for making changes to the system to integrate that code. It also takes into account the amount of code that is automatically generated where the development effort is calculated, as explained earlier in this section.

The following formula is used to calculate the number of equivalent lines of source code:

$$\text{ESLOC} = \text{ASLOC} \times (1 - \text{AT}/100) \times \text{AAM}$$

ASLOC is reduced according to the percentage of automatically generated code. **AAM** is the Adaptation Adjustment Multiplier, which takes into account the effort required to reuse code. Simplistically, **AAM** is the sum of three components:

1. An adaptation component (referred to as **AAF**) that represents the costs of making changes to the reused code. This includes components that take into account design, code and integration changes.
2. An understanding component (referred to as **SU**) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. **SU** ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.
3. An assessment factor (referred to as **AA**) that represents the costs of reuse decision-making. That is, some analysis is always required to decide whether code can be reused, and this is included in the cost as **AA**. **AA** varies from 0 to 8 depending on the amount of analysis effort required.

The reuse model is a nonlinear model. Some effort is required if reuse is considered to make an assessment of whether reuse is possible. Furthermore, as more and more reuse is contemplated, the costs per code unit reused drop as the fixed understanding and assessment costs are spread across more lines of code.

The post-architecture level

The post-architecture model is the most detailed of the COCOMO II models. It is used once an initial architectural design for the system is available so the sub-system structure is known.

The estimates produced at the post-architecture level are based on the same basic formula ($PM = A \times \text{Size}^B \times M$) used in the early design estimates. However, the size estimate for the software should be more accurate by this stage in the estimation process. In addition, a much more extensive set of product, process and organisational attributes (17 rather than 7) are used to refine the initial effort computation. It is possible to use more attributes at this stage because you have more information about the software to be developed and the development process.

The estimate of the code size in the post-architecture model is computed using three components:

1. An estimate of the total number of lines of new code to be developed
2. An estimate of the equivalent number of source lines of code (ESLOC) calculated using the reuse model
3. An estimate of the number of lines of code that have to be modified because of changes to the requirements.

These three estimates are added to give the total code size in KSLOC that you use in the effort computation formula. The final component in the estimate—the number of lines of modified code—reflects the fact that software requirements always change. The system programs have to reflect these requirements changes so additional code has to be developed. Of course, estimating the number of lines of code that will change is not easy, and there will often be even more uncertainty in this figure than in development estimates.

The exponent term (**B**) in the effort computation formula had three possible values in COCOMO I. These were related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. However, good organisational practices and procedures can control this 'diseconomy of scale'. This is recognised in COCOMO II, where the range of values for the exponent **B** is continuous rather than discrete. The exponent is based on five scale factors, as shown in Figure 26.9. These factors are rated on a six-point scale from Very low to Extra high (5 to 0). You should then add the ratings, divide them by 100 and add the result to 1.01 to get the exponent that should be used.

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

632 Chapter 26 ■ Software cost estimation

Figure 26.9 Scale factors used in the COCOMO II 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.

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 28). Possible values for the ratings used in exponent calculation are:

- *Precedentedness* This is a new project for the organisation—rated Low (4)
- *Development flexibility* No client involvement—rated Very high (1)
- *Architecture/risk resolution* No risk analysis carried out—rated Very low (5)
- *Team cohesion* New team so no information—rated Nominal (3)
- *Process maturity* Some process control in place—rated Nominal (3)

The sum of these values is 16, so you calculate the exponent by adding 0.16 to 1.01, getting a value of 1.17.

The attributes (Figure 26.10) that are used to adjust the initial estimates and create multiplier *M* 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.

Figure 26.10 Project cost drivers

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

- Personnel attributes are multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes are concerned with the particular characteristics of the software development project.

Figure 26.11 shows 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 assumed 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 they do not affect the computation of the effort.

Figure 26.11 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 Figure 26.11, 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 II reference manual (Boehm, 1997). You can see that high values for the cost drivers lead an effort estimate that is more than three times the initial estimate, whereas low values reduce the estimate to about one third 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 formulae proposed by the developers of the COCOMO II model reflects their experience and data, but it is an extremely complex model to understand and use. There are many attributes and considerable 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 that 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 II therefore has to start with the published values for the model parameters, and it is impossible for a user to know how closely these relate to their own situation. This means that the practical use of the COCOMO model is limited. Very large organisations may have the resources to employ a cost-modelling expert to adapt and use the COCOMO II models. However, for the majority of companies, the cost of calibrating and learning to use an algorithmic model such as the COCOMO model is so high that they are unlikely to introduce this approach.

26.3.2 Algorithmic cost models in project planning

One of the most valuable uses of algorithmic cost modelling is to compare different ways of investing money to reduce project costs. This is particularly important

where you have to make hardware/software cost trade-offs and where you may have to recruit new staff with specific project skills. The algorithmic code model helps you assess the risks of each option. Applying the cost model reveals the financial exposure that is associated with different management decisions.

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 26.13 shows some possible options for this project. These include spending more on target hardware to reduce software costs or investing in better development tools.

Additional hardware costs may be acceptable 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 increases the unit cost of the product, irrespective of the number sold, which is usually undesirable.

Figure 26.13 shows the hardware, software and total costs for the options A–F shown in Figure 26.12. Applying the COCOMO II model without cost drivers predicts an effort of 45 person-months to develop an embedded software system for this application. The average cost for one person-month of effort is \$15,000.

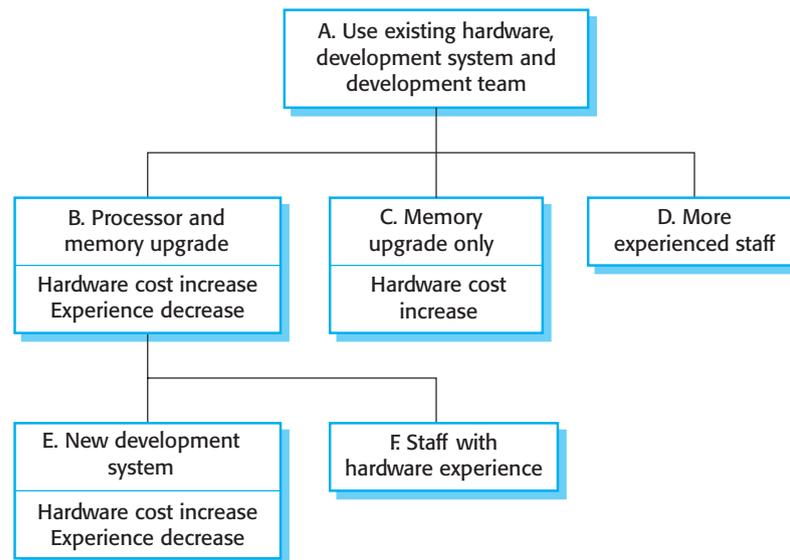
The relevant multipliers are based on storage and execution time constraints (**TIME** and **STOR**), the availability of tool support (cross-compilers, etc.) for the development system (**TOOL**), and development team's experience platform experience (**LTEX**). In all options, the reliability multiplier (**RELY**) is 1.39, indicating that significant extra effort is needed to develop a reliable system.

The software cost (**SC**) is computed as follows:

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

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. The staff lack experience with the new hardware so the increase in the experience multiplier negates the reduction in the **STOR** and **TIME** multipliers. It is actually more cost-effective to upgrade memory rather than the whole computer configuration.

636 Chapter 26 ■ Software cost estimation

Figure 26.12
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	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	1.39	1.06	1.11	0.86	0.84	51	769008	100000	897490
EX	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

Figure 26.13 Cost of
Management options

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, which involves significant costs and risks. These may mean that the cost advantages of this option are much less significant than suggested by Figure 26.13. 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 26.1. It also reveals that investment in new hardware and tools may not be cost-effective. Some engineers may prefer this option because it gives them an opportunity to learn about and work with new systems. However, the loss of experience is a more significant effect on the system cost than the savings that arise from using the new hardware system.

26.4 Project duration and staffing

As well as estimating the effort required to develop a software system and the overall project costs, 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. The reason for this is that people spend more time communicating and defining interfaces between the parts of the system developed by other people. Doubling the number of staff (for example) therefore 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:

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

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 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 the COCOMO II model predicts this:

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

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 of problems delivering the software as planned.

To illustrate the COCOMO development schedule computation, assume that 60 months of effort are estimated to develop a software system (Option C in Figure

26.12). Assume that the value of exponent B is 1.17. From the schedule equation, the time required to complete the project is:

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

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 (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.



KEY POINTS

- There is not necessarily a simple relationship between the price charged for a system and its development costs. Organisational factors may mean that the price charged is increased to compensate for increased risk or decreased to gain competitive advantage.
- Factors that affect software productivity include individual aptitude (the dominant factor), domain experience, the development process, the size of the project, tool support and the working environment.
- Software is often priced to gain a contract, and the functionality of the system is then adjusted to meet the estimated price.
- There are various techniques of software cost estimation. In preparing an estimate, several different techniques should be used. If the estimates diverge widely, this means that inadequate estimating information is available.
- The COCOMO II costing model is a well-developed algorithmic cost 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 can be used to 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 increase rather than decrease the time required to finish the project.

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, only a small number of people are needed at the start of a project to carry out the initial design. The team then builds up to a peak during the development and testing of the system, and finally the team size declines as the system is prepared for deployment. A very rapid buildup of project staff has been shown to correlate 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), which 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.

FURTHER READING

'Ten unmyths of project estimation'. A pragmatic article that discusses the practical difficulties of project estimation and challenges some fundamental assumptions in this area. (P. Armour, *Comm. ACM*, 45(11), November 2002.)

Software Cost Estimation with COCOMO II. This is the definitive book on the COCOMO II model. It provides a complete description of the model with many examples and includes software that implements the model. It's extremely detailed and not light reading. The Boehm's paper below is, in my view, an easier introduction to the model. (B. Boehm, et al., 2000, Prentice Hall.)

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 II'. An introduction to the COCOMO II cost estimation model that includes rationale for the formulae used. Easier to read than the definitive book. (B. Boehm et al., *Annals of Software Engineering*, 1, Balzer Science Publishers, 1995.)

EXERCISES

- 26.1 Under what circumstance might a company charge a much higher price for a software system than that suggested by the cost estimate plus a normal profit margin?
- 26.2 Describe two metrics that have been used to measure programmer productivity. Comment briefly on the advantages and disadvantages of each of these metrics.

- 26.3** In the development of large, embedded real-time systems, suggest five factors that are likely to have a significant effect on the productivity of the software development team.
- 26.4** 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.
- 26.5** Why should several estimation techniques be used to produce a cost estimate for a large, complex software system?
- 26.6** A software manager is in charge of the development of a safety-critical software system that 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.

- 26.7** Give three reasons why algorithmic cost estimates prepared in different organisations are not directly comparable
- 26.8** 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.
- 26.9** Some very large software projects involve writing 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?
- 26.10** 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?
- 26.11** 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?