8th unit

25. Management

Contents

- The people working in a software organisation are its greatest assets. They represent intellectual capital, and it is up to software managers to ensure that the organization gets the best possible return on its investment in people.
- Effective management is therefore about managing the people in an organisation. Project managers have to solve technical and nontechnical problems by using the people in their team in the most effective way possible. They have to motivate people, plan and organise their work and ensure that the work is being done properly.
- Poor management of people is one of the most significant contributors to project failure.

Four critical factors in people management:

1. *Consistency* People in a project team should all be treated in a comparable way. people should not feel that their contribution to the organisation is undervalued.

2. *Respect* Different people have different skills and managers should respect these differences. All members of the team should be given an opportunity to make a contribution.

3. *Inclusion* It is important to develop a working environment where all views, even those of the most junior staff should feel that, others listen to them and take account of their proposals.

4. *Honesty* As a manager, he should always be honest about what is going well and what is going badly in the team. He should also be honest about level of technical knowledge and be willing to defer to staff with more knowledge when necessary.

25.1 Selecting staff

One of the most important project management tasks is team selection.

The decision on who to appoint to a project is usually made using **three types** of information:

1. Information provided by candidates about their background and experience (their résumé or CV). This is usually the most reliable evidence to judge whether candidates are likely to be suitable.

2. Information gained by interviewing candidates. Interviews can give a good impression of whether a candidate is a good communicator and whether he or she has good social skills. Consequently, interviews are not a reliable method for making judgements of technical capabilities.

3. Recommendations from people who have worked with the candidates. This can be effective when selection team know the people making the recommendation. Otherwise, the recommendations cannot be trusted.

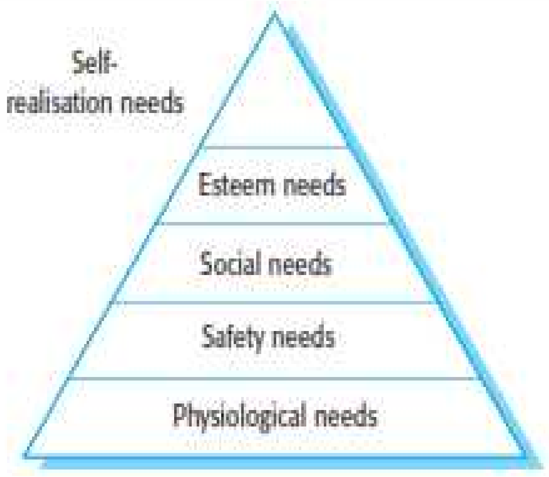
The factors that influence choosing a staff. are shown in Figure 25.2.

25.2 Motivating people

1. Motivation means organising the work and the working environment so that people are stimulated to work as effectively as possible.

2. Maslow (Maslow 1954) suggests that people are motivated by satisfying their needs and that needs are arranged in a series of levels, as shown in Figure 25.3.

Figure 25.3 Human needs hierarchy



3. The lower levels of this hierarchy represent fundamental needs for food, sleep.

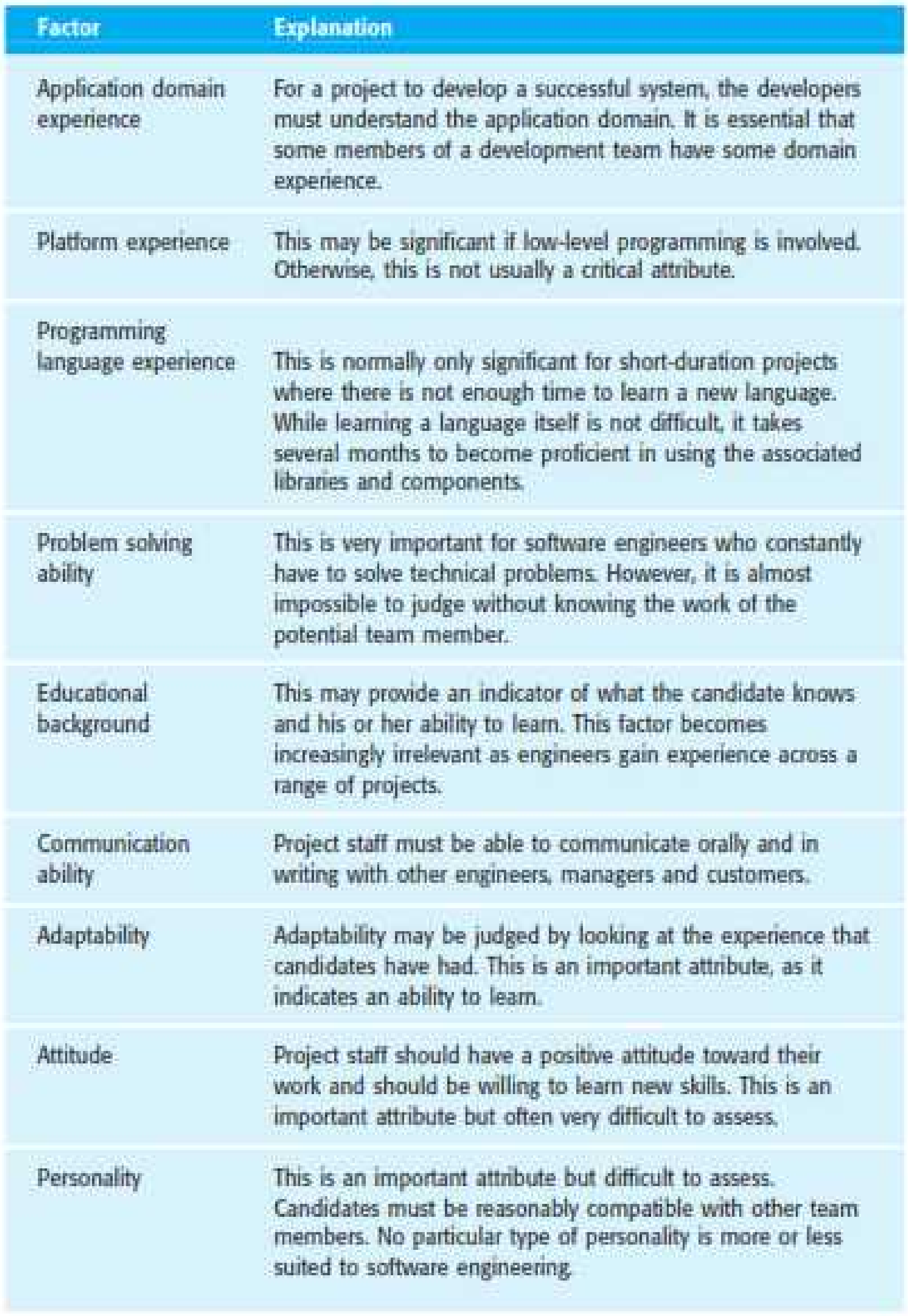
4. Safety needs as the need to feel secure in an environment.

5. Social needs are concerned with the need to feel part of a social grouping.

6. Esteem needs are the need to feel respected by others, and self-realisation needs are concerned with personal development.

.

Figure 25.2 Factors governing staff selection



Factor	Explanation
Application domain experience	For a project to develop a successful system, the developers must understand the application domain. It is essential that some members of a development team have some domain experience.
Platform experience	This may be significant if low-level programming is involved. Otherwise, this is not usually a critical attribute.
Programming language experience	This is normally only significant for short-duration projects where there is not enough time to learn a new language. While learning a language itself is not difficult, it takes several months to become proficient in using the associated libraries and components.
Problem solving ability	This is very important for software engineers who constantly have to solve technical problems. However, it is almost impossible to judge without knowing the work of the potential team member.
Educational background	This may provide an indicator of what the candidate knows and his or her ability to learn. This factor becomes increasingly irrelevant as engineers gain experience across a range of projects.
Communication ability	Project staff must be able to communicate orally and in writing with other engineers, managers and customers.
Adaptability	Adaptability may be judged by looking at the experience that candidates have had. This is an important attribute, as it indicates an ability to learn.
Attitude	Project staff should have a positive attitude toward their work and should be willing to learn new skills. This is an important attribute but often very difficult to assess.
Personality	This is an important attribute but difficult to assess. Candidates must be reasonably compatible with other team members. No particular type of personality is more or less suited to software engineering.

7. Therefore, ensuring the satisfaction of social, esteem and self-realisation needs is most significant from a management point of view.

a). To satisfy social needs, need to give people time to meet their co-workers and to provide places for them to meet. This is relatively easy when all of the members of a development team work in the same place but, increasingly, team members are not located in the same building or even the same town or state.

Electronic communications such as e-mail and teleconferencing may be used to support this remote working. However electronic communications do not really satisfy social needs.

If team is distributed, arrange periodic face-to-face meetings. Through this direct interaction, people become part of a social group and may be motivated by the goals and priorities of that group.

b). to satisfy esteem needs, need to show people that they are valued by the organisation. Public recognition of achievements is a simple yet effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.

c). Finally, to satisfy self-realisation needs, need to give people responsibility for their work, assign them demanding (but not impossible) tasks and provide a training programme where people can develop their skills.

8. Classification of professionals (Bass and Dunteman, 1963) into three types:

a). *Task-oriented* people are motivated by the work they do. In software engineering, they are technicians who are motivated by the intellectual challenge of software development.

b). *Self-oriented* people are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals. This does not mean that these people are selfish and think only of their own concerns. Rather, they often have longer-term goals such as career progression, that motivate them. They wish to be successful in their work to help realise these goals.

c). *Interaction-oriented* people are motivated by the presence and actions of coworkers. As software development becomes more user-centered, interactionoriented individuals are becoming more involved in software engineering.

25.3 Managing Groups

1. Most professional software is developed by project teams ranging in size from two to several hundred people.

2. However, as it is clearly impossible for all these people to work together on a single problem, these large teams are usually split into a number of groups. Each group is responsible for part of the overall system.

3. Putting together a group that works effectively is a critical management task. It is obviously important that the group should have the right balance of technical skills, experience and personalities.

4. There are a number **of factors that** influence group working:

a). *Group composition* Is there the right balance of skills, experience and personalities in the team?

b). *Group cohesiveness* Does the group think of itself as a team rather than as a collection of individuals who are working together?

c). *Group communications* Do the members of the group communicate effectively with each other?

d). *Group organisation* Is the team organised in such a way that everyone feels valued and is satisfied with his or her role in the group?

25.3.1 Group composition

1. A group that has complementary personalities may work better than a group selected solely on technical ability.

2. People who are motivated by the work are likely to be the strongest technically.

3. People who are self-oriented will probably be best at pushing the work forward to finish the job.

4. People who are interaction-oriented help facilitate communications within the group.

5. It is sometimes impossible to choose a group with complementary personalities. In that case, the project manager has to control the group so that individual goals do not transcend organisational and group objectives.

For example, say an engineer is given a program design for coding and notices possible design improvements. If he implements these improvements without understanding the rationale for the original design, they might have adverse implications for other parts of the system. If all the members of the group are involved in the design from the start, they will understand why design decisions have been made.

6. Roles of the group leader

a) He or she may be responsible for providing technical direction and project administration.

b) Group leaders must keep track of the day-to-day work of their group.

c) Ensure that people are working effectively and work closely with project managers on project planning.

7. Leaders are normally appointed and report to the overall project manager. However, the appointed leader may not be the real leader of the group as far as technical matters are concerned.

8. The group members may look to another group member for leadership. He or she may be the most technically competent engineer or may be a better motivator than the appointed group leader.

25.3.2 Group cohesiveness

1. In a cohesive group, members think of the group as more important than the individual in it.

2. They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

3. The advantages of a cohesive group are:

a). *A group quality standard can be developed* Because this standard is established by consensus, it is more likely to be observed than external standards imposed on the group.

b). *Group members work closely together* People in the group learn from each other.

c). *Group members can get to know each other's work* Continuity can be maintained if a group member leaves. Others in the group can take over critical tasks and ensure that the project is not unduly disrupted.

d). *Egoless programming can be practised* Programs are regarded as group property rather than personal property.

Advantages of Egoless programming

- Thus Group cohesiveness is improved because all members feel that they have a shared responsibility for the software. The idea of egoless programming is fundamental to extreme programming.
- In extreme programming, constant improvement of the code in the system, irrespective of who wrote that code.
- egoless programming also improves communications within the group.
- It encourages uninhibited discussion without regard to status, experience or gender

4. Group cohesiveness depends on many factors, including the organisational culture and the personalities in the group.

5. Managers can encourage cohesiveness in a number of ways:

a) They may organise social events for group members and their families;

b) They may try to establish a sense of group identity by naming the group and establishing a group identity and territory;

c) they may get involved in explicit groupbuilding activities such as sports and games.

6. One of the most effective ways to promote cohesion is to ensure that group members are treated as responsible and trustworthy and are given access to organizational information.

7. Strong, cohesive groups, however, can sometimes suffer from two problems:

a). ***Irrational resistance to a leadership change*** If the leader of a cohesive group has to be replaced by someone outside of the group, the group members may band together against the new leader. Group members may spend time resisting changes proposed by the new group leader with a consequent loss of productivity. Whenever possible, new leaders are therefore best appointed from within the group.

b). ***Groupthink*** is when the critical abilities of group members are eroded by group loyalties. To avoid groupthink, organise formal sessions in which group members are encouraged to question decisions that have been made. Outside experts may be introduced to review the group's decisions.

25.3.3 Group communications

1. Good communication between members of a software development group is essential.

2. The group members must exchange information on the status of their work, the design decisions that have been made and changes to previous decisions that are necessary.

3. Some key factors that influence the effectiveness of communication are:

a). ***Group size*** As a group increases in size, ensuring that all members communicate effectively with each other becomes more difficult.

b). ***Group structure*** People in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure.

In hierarchical groups, communications tend to flow up and down the hierarchy. People at the same level may not talk to each other.

c). ***Group composition*** People with the same personality types may clash and communications may be inhibited.

d). ***The physical work environment*** The organisation of the workplace is a major factor in facilitating or inhibiting communications.

25.3.4 Group organisation

1. Small programming groups are usually organised in a fairly informal way. The group leader gets involved in the software development with the other group members.

2. A technical leader may emerge who effectively controls software production.

3. In an **informal group**, the work to be carried out is discussed by the group as a whole, and tasks are allocated according to ability and experience.

4. More senior group members may be responsible for the architectural design. However, detailed design and implementation is the responsibility of the team member who is allocated to a particular task.

5. Informal groups can be very successful, particularly when the majority of group members are experienced and competent

6. To make the most effective use of highly skilled programmers, Baker (Baker, 1972) and others (Aron, 1974; Brooks, 1975) suggest that teams should be built around an individual, highly skilled chief programmer.

7. The underlying principle of the **chief programmer team** is that skilled and experienced staff should be responsible for all software development.

8. They should focus on the software to be developed and should not get involved in external meetings.

9. Disadvantages of chief programmer teams: But the chief programmer team organisation has serious problems because it is over-dependent on the chief programmer and their assistant.

10. Other team members, who are not given sufficient responsibility, become unmotivated because they feel their skills are underused.

25.3.5 Working environments

1. The workplace has important effects on people's performance and their job satisfaction.

2. Psychological experiments have shown that behaviour is affected by room size, furniture, equipment, temperature, humidity, brightness and quality of light, noise and the degree of privacy available.

3. Group behaviour is affected by architectural organisation and telecommunication facilities.

4. Communications within a group are affected by the building architecture and the structure of the workspace.

5. The most important environmental factors identified in the study were:

a). *Privacy* Programmers require an area where they can concentrate and work without interruption.

b). *Outside awareness* People prefer to work in natural light and with a view of the outside environment.

c). *Personalisation* Individuals adopt different working practices and have different opinions on decor. The ability to rearrange the workplace to suit working practices and to personalise that environment is important.

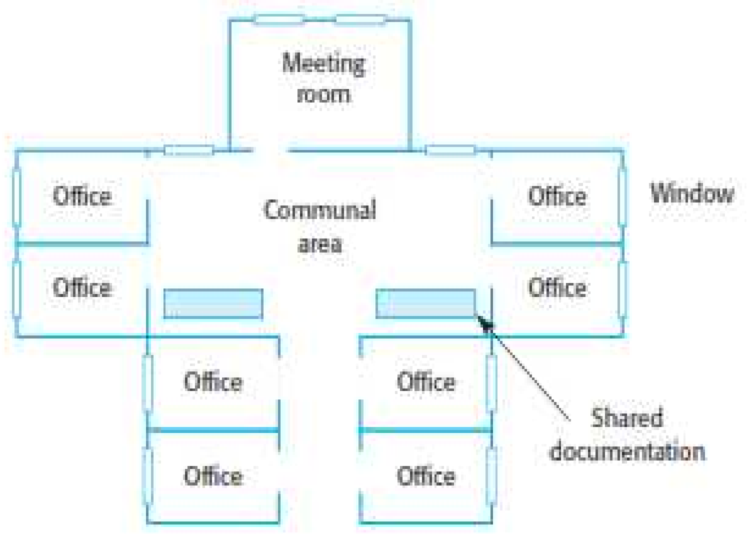
In short, people like individual offices that they can organise to their taste and needs.

6. Development groups need areas where all members of the group can get together and discuss their project, both formally and informally.

7. Meeting rooms must be able to accommodate the whole group in privacy.

8. McCue suggested grouping individual offices round larger group meeting rooms (Figure 25.7) was the best way to reconcile these conflicting requirements.

Figure 25.7 Office and meeting room grouping



25.4 The People Capability Maturity Model

1. The Software Engineering Institute (SEI) in the United States is engaged in a longterm programme of software process improvement. Part of this programme is the Capability Maturity Model (CMM) for software processes, to support this model; they have also proposed a People Capability Maturity Model (PCMM)

2, The P-CMM can be used as a framework for improving the way in which an organisation manages its human assets.

3. Like the CMM, the P-CMM is a five-level model, as shown in Figure 25.9.

4. The five levels are:

a). ***Initial*** **Ad hoc**, informal people management practices

b). ***Repeatable*** Establishment of policies for developing the capability of the staff

c). ***Defined*** Standardisation of best people management practice across the organisation

d). ***Managed*** Quantitative goals for people management

e). ***Optimizing*** **Continuous** focus on improving individual competence and workforce motivation

5. Curtis state that the strategic objectives of the P-CMM are:

a). To improve the capability of software organisations by increasing the capability of their workforce

b) To ensure that software development capability is an attribute of the organization rather than of a few individuals

c) To align the motivation of individuals with that of the organisation

d) To retain valuable human assets (i.e., people with critical knowledge and skills) within the organisation.

6. The P-CMM is a practical tool for improving the management of people in an organisation because it provides a framework for motivating, recognising, standardizing and improving good practice.

7. It reinforces the need to recognise the importance of people as individuals and to develop their capabilities.

8. it is a helpful guide that can lead to significant improvements in the capability of organisations to produce highquality software.

9. The complete application of this model is very expensive and probably unnecessary for most organisations.

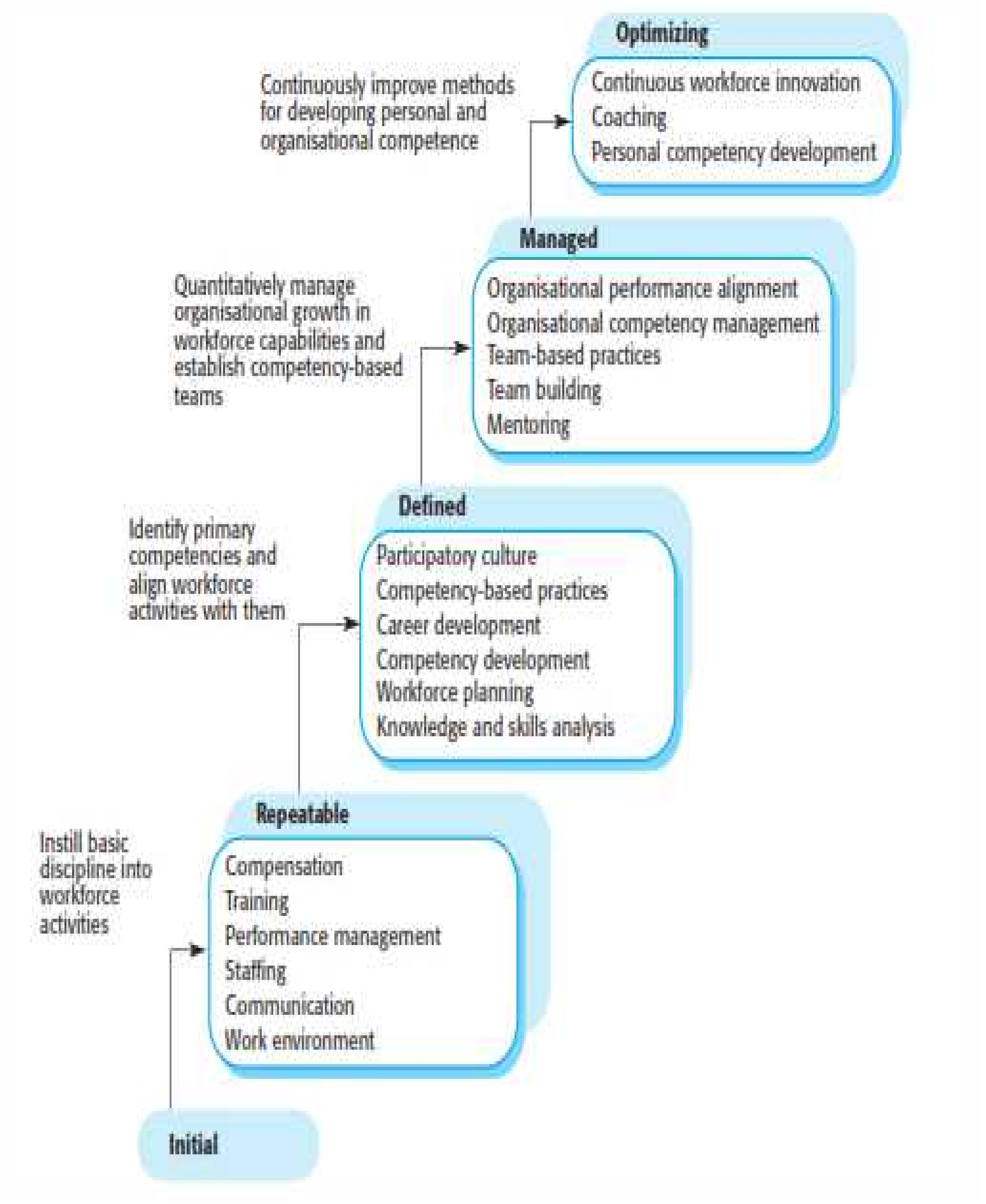

Optimizing
Continuous workforce innovation
Coaching
Personal competency development
Continuously improve methods for developing personal and organisational competence
Managed
Organisational performance alignment
Organisational competency management
Team-based practices
Team building
Mentoring
Quantitatively manage organisational growth in workforce capabilities and establish competency-based teams
Defined
Participatory culture
Competency-based practices
Career development
Competency development
Workforce planning
Knowledge and skills analysis
Identify primary competencies and align workforce activities with them
Repeatable
Compensation
Training
Performance management
Staffing
Communication
Work environment
Instill basic discipline into workforce activities
Initial

Figure 25.9 The People Capability Maturity Model

2nd chapter

26 Software cost estimation

Contents

Introduction

This chapter involves, 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?

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).

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.

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.

Software pricing must take into account broader organisational, economic, political and business considerations, such as those shown in Figure 26.1

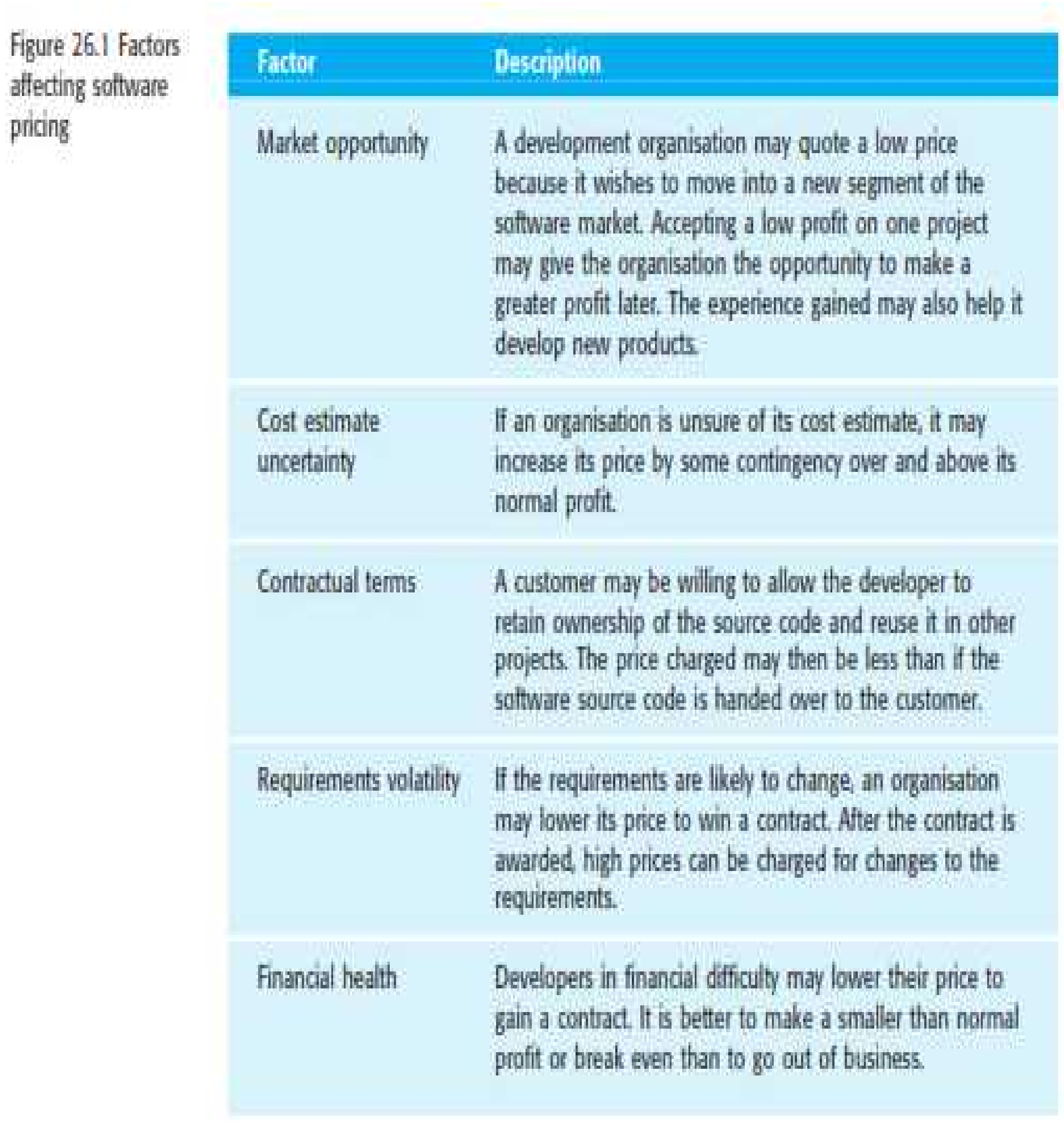


Figure 26.1 Factors affecting software pricing

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.

26.1 Software productivity

1. We 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 to produce them.

2. A project manager, estimate the productivity of software engineers. a project manager may need these productivity estimates to define the project cost or schedule, to inform investment decisions or to assess whether process or technology improvements are effective.

3. Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development.

4. There are two types of metric that have been used:

a). ***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.

b). ***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.

<u>Size-related metrics to measure software productivity</u>

1. The approach first developed to measure software productivity, when most programming was in FORTRAN, assembly language or COBOL

a). Lines of source code per programmer-month (LOC/pm) is a widely used software productivity metric.

LOC/pm= (total number of lines of source code that are delivered)/ (total time in programmer months required to complete the project)

b). This time therefore includes the time required for all other activities (requirements, design, coding, testing and documentation) involved in software development.

2. 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.

3. Comparing productivity across programming languages can also give misleading impressions of programmer productivity.

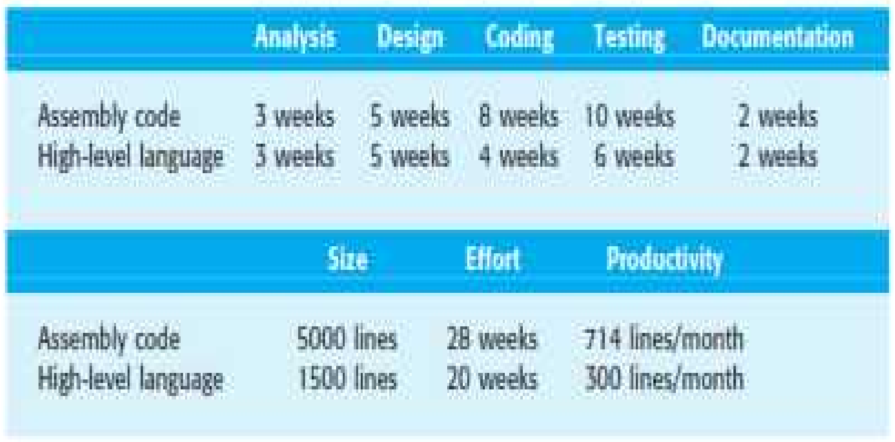
4. The more expressive the programming language, the lower the apparent productivity.

5. 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.

6. 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.

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

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.

7. 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.

function-related metrics to measure software productivity

- **Using function points**

1. This involves brief description and comparison of several function-based measures. The best known of these measures is the function-point count.

2. Productivity is expressed as the number of function points that are implemented per person-month.

3. We can compute the total number of function points in a program by measuring or estimating the following program features:

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

4. 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.

5. We 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.

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

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

7. Then modify this unadjusted function-point count by additional complexity factors that are related to the complexity of the system as a whole.

8. 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. They are effective in practical situations

9.Disadvantages

1. 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.

2. 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.

➢ **Using Object points**

1.Object points are an alternative to function points. They can be used with languages such as database programming languages or scripting languages.

2. the number of object points in a program is a weighted estimate of:

a). ***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.

b). ***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.

c). ***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).

3. 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

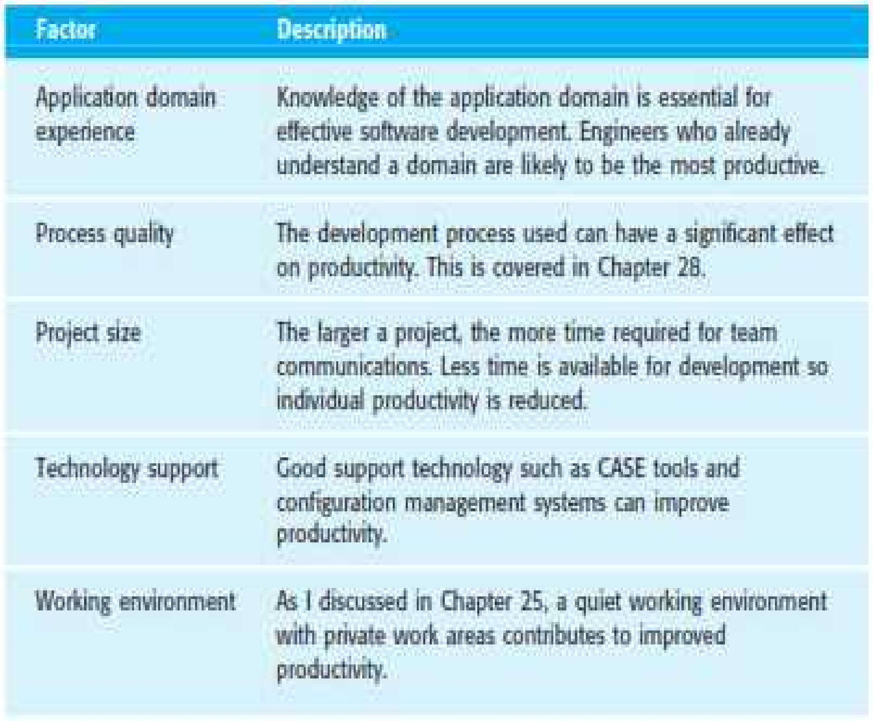
4. Function-point and object-point counts can be used in code-estimation models.

- The final code size is calculated from the number of function points in a code.
- AVC(the average number of lines of code) 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:

Code size = AVC X 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 summarized in Figure 26.3.

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.

26.2 Estimation techniques

1. There is no simple way to make an accurate estimate of the effort required to develop a software system.

2. We 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.

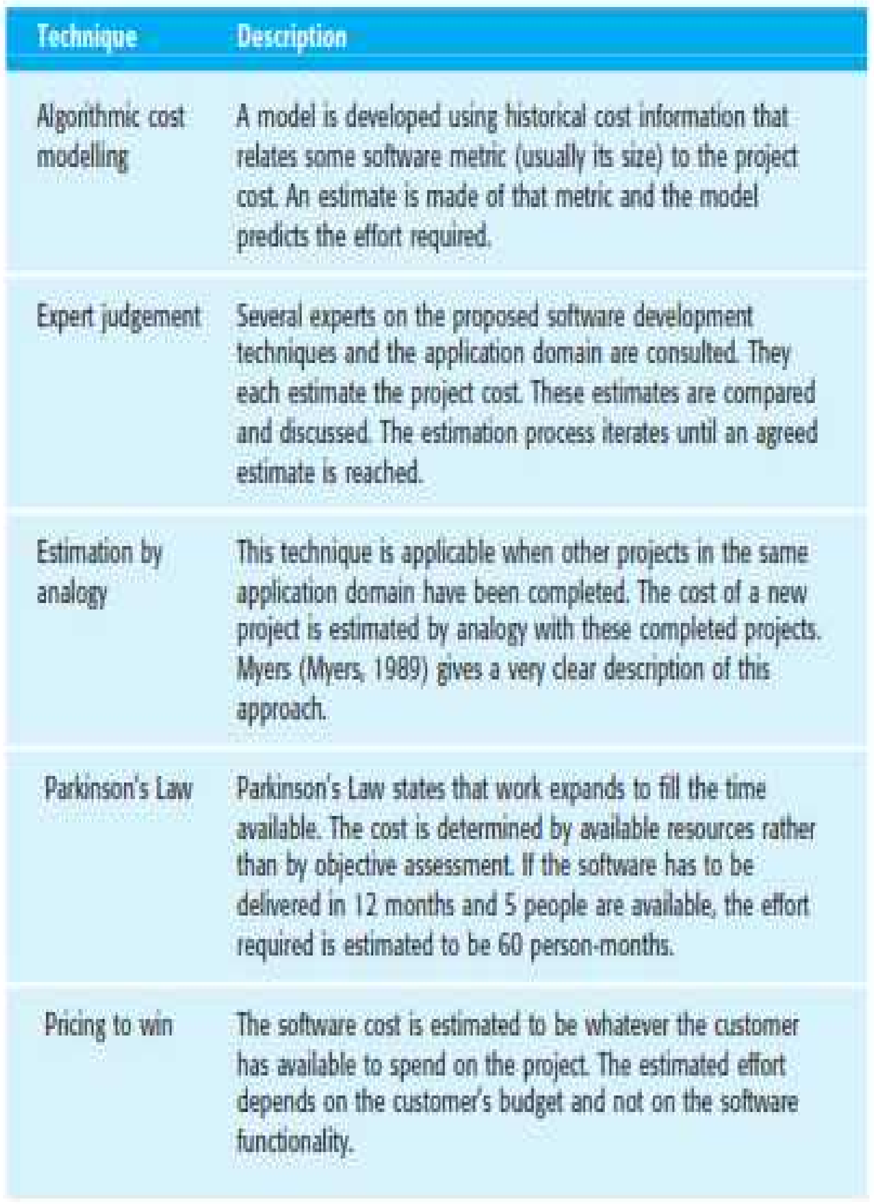
3. The people involved in the project and their skills will probably not be known. All of these mean that it is impossible to estimate system development costs accurately at an early stage in a project.

4. 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.

5. All of these techniques rely on experience-based judgements by project managers .

6. However, there may be important differences between past and future projects.

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.

7. Some examples of the changes that may affect estimates based on experience include:

- **a)** Distributed object systems rather than mainframe-based systems
- **b)** Use of web services
- **c)** Use of ERP or database-centred systems
- **d)** Use of off-the-shelf software rather than original system development
- **e)** Development for and with reuse rather than new development of all parts of a System
- **f)** Development using scripting languages such as TCL or Perl (Ousterhout, 1998)

g) The use of CASE tools and program generators rather than unsupported software development.

8. If project managers have not worked with these techniques, their previous experience may not help them estimate software project costs.

9. We 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

a) It starts at the system level by examining the overall functionality of the product and how that functionality is provided by interacting sub-functions.
b) The costs of system-level activities such as integration, configuration management and documentation are taken into account.

The bottom-up approach

a) It starts at the component level. The system is decomposed into components.
b) Then estimate the effort required to develop each of these components.
c) 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.

10 Each estimation technique has its own strengths and weaknesses. For large projects, therefore, we should use several cost estimation techniques and Compare their results.

11. These estimation techniques are applicable where a requirements document for the system has been produced. This should define all users and system requirements.

12. Sometimes the costs of many projects must be estimated using only incomplete user requirements for the system.

13. Under these circumstances, **“pricing to win”** is a commonly used strategy. The notion of pricing to win may seem unethical and unbusinesslike.

- 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.

26.3 Algorithmic cost modelling

1. 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.

2. 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.

3. Algorithmic cost models are primarily used to make estimates of software development costs.

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

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

Where

A= is a constant factor that depends on local organisational practices and the typeof 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.

Exponent B= The value of exponent B usually lies between 1 and 1.5.

M= It is a multiplier made by combining process, product and development attributes.

4. Exponential B, 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.Therefore, the larger the system, the larger the value of this exponent.

5. all algorithmic models suffer from the fundamental difficulties they are:

a). *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.

b). *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.

6. If we use an algorithmic cost estimation model, we should develop a range of estimates (worst, expected and best) rather than a single estimate and apply the costing formula to all of them.

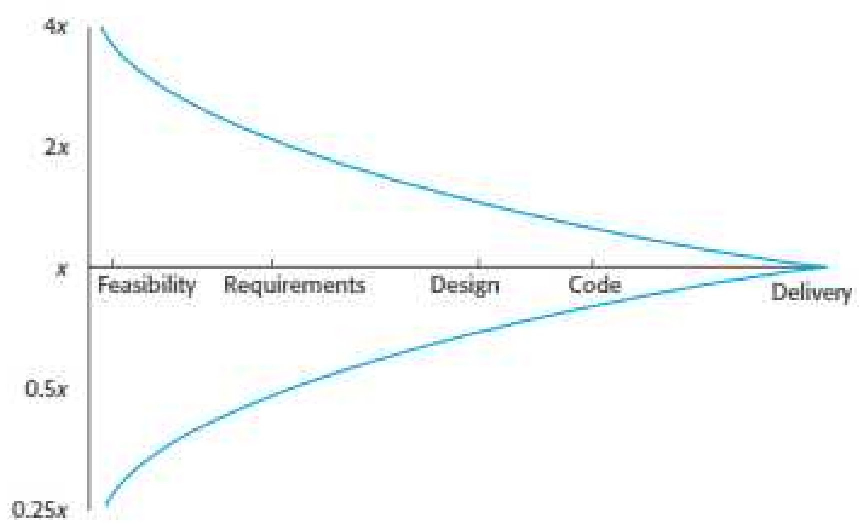
7. Estimates are most likely to be accurate when we understand the type of software that is being developed. when programming language and hardware choices are predefined.

8. 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.

9. If the initial estimate of effort required is *x* months of effort, this range may be from 0.25*x*

to 4*x* when the system is first proposed. This narrows during the development process, as shown in Figure 26.5.

Figure 26.5 Estimate uncertainty



26.3.1 The COCOMO model

1. The COCOMO model is an empirical model that was derived by collecting data from a large number of software projects.

2. 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.

3. Reasons to chose the COCOMO model

a). It is well documented, available in the public domain and supported by public domain and commercial tools.

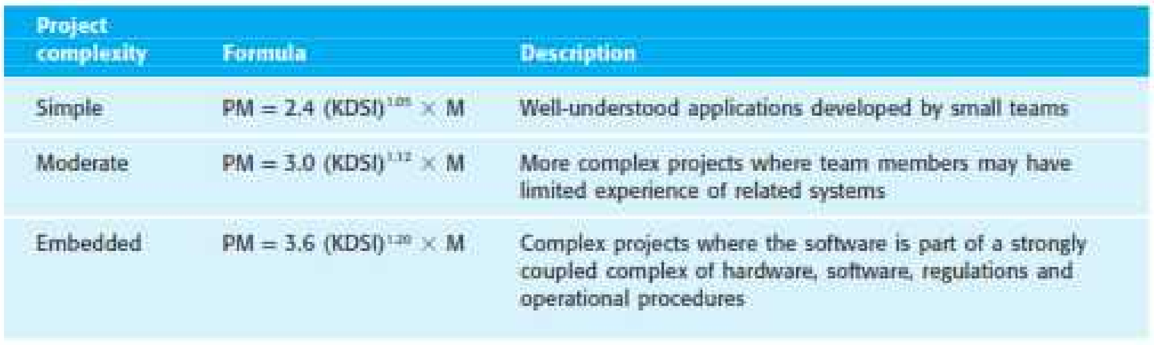
b). It has been widely used and evaluated in a range of organisations.

c). It has a long history from its first instantiation to its most recent version.

4. **COCOMO 81**

a) 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.

b) 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.



Project complexity	Formula	Description
Simple	PM = 2.4 $(KDSI)^{1.05}$ × M	Well-understood applications developed by small teams
Moderate	PM = 3.0 $(KDSI)^{1.12}$ × M	More complex projects where team members may have limited experience of related systems
Embedded	PM = 3.6 $(KDSI)^{1.20}$ × 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

c) The multiplier M reflects product, project and team characteristics.

d) COCOMO 81 suitable for software that would be developed according to a waterfall process using standard programming languages such as C or FORTRAN.

e) Not suitable for Prototyping and incremental software development,off-the-shelf and CASE tool support.

f) To take these changes into account, the COCOMO II model was developed.

5. COCOMO II

a) It recognises different approaches to software development such as prototyping, development by component composition and use of database programming.

b) COCOMO II supports a spiral model of development and embeds several sub-models that produce increasingly detailed estimates.

c) These can be used in successive rounds of the development spiral. Figure 26.7 shows COCOMO II sub-models and where they are used.

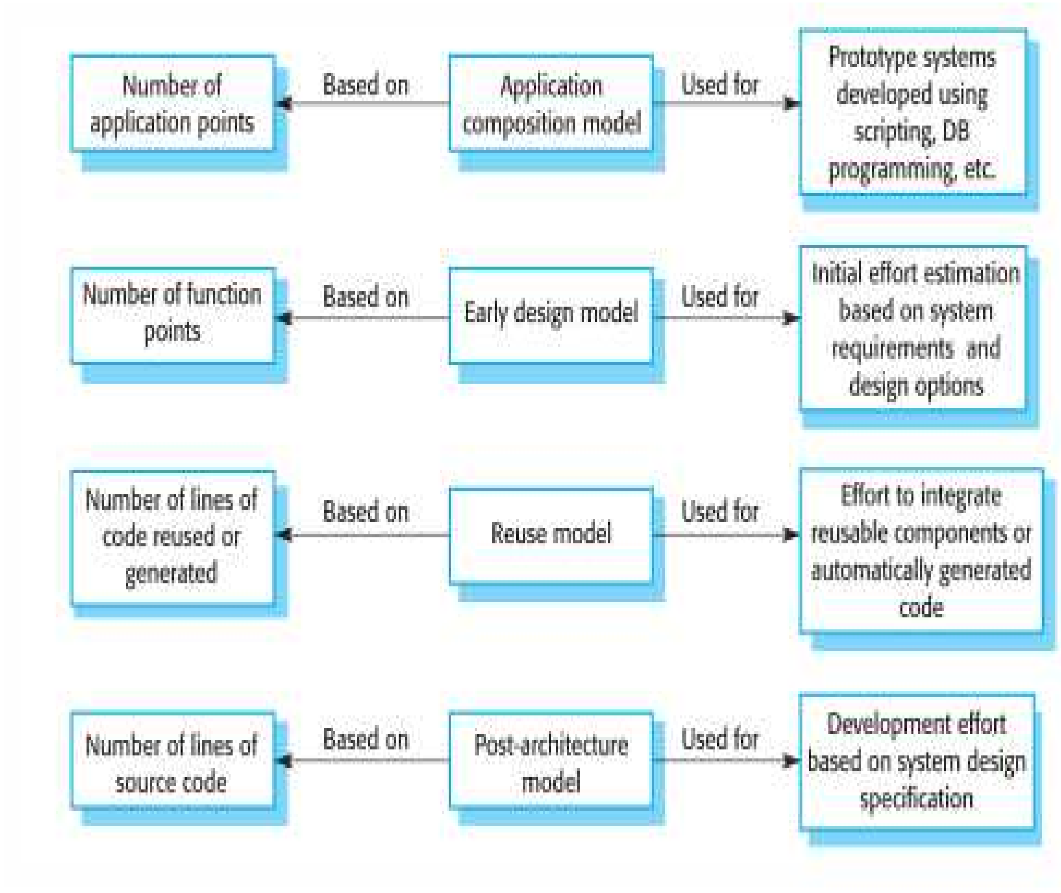

Number of application points
Based on
Application composition model
Used for
Prototype systems developed using scripting, DB programming, etc.
Number of function points
Based on
Early design model
Used for
Initial effort estimation based on system requirements and design options
Number of lines of code reused or generated
Based on
Reuse model
Used for
Effort to integrate reusable components or automatically generated code
Number of lines of source code
Based on
Post-architecture model
Used for
Development effort based on system design specification

Figure 26.7 The COCOMO II models

d) 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.

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.

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.

4. ***A post-architecture model*** Once the system architecture has been designed, a more accurate estimate of the software size can be made.

<u>The application-composition model</u>

1. 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.
2. It is based on an estimate of weighted application points (object points) divided by a standard estimate of application-point productivity.
3. The estimate is then adjusted according to the difficulty of developing each object point.
4. Figure 26.8 shows the levels of object-point productivity suggested by the model developers.

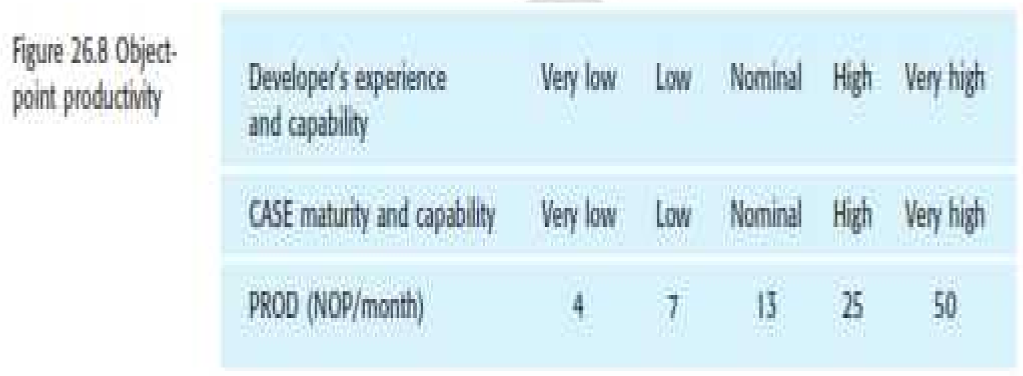


Figure 26.8 Object-point productivity

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

5. 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.
6. Therefore, the final formula for effort computation for system prototypes is:

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

Where

PM=is the effort estimate in person-months.

NAP = 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 early design model

1. This model is used once user requirements have been agreed and initial stages of the system design process are underway.
2. However, there is no need of a detailed architectural design to make initial estimates.
3. goal at this stage should be to make an approximate estimate without undue effort.
4. The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

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

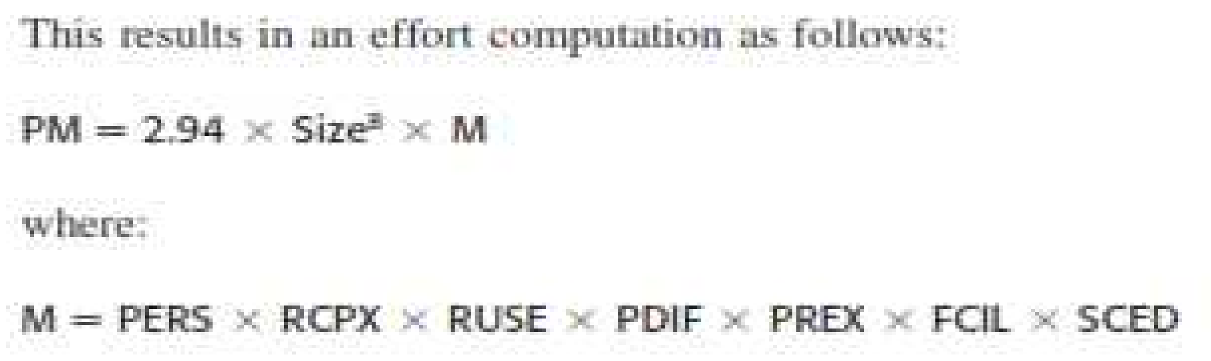
where

- ➢ coefficient A (Boehm proposes)=should be 2.94.
- ➢ size =of the system is expressed in KSLOC, which is the number of thousands of lines of source code.
- ➢ exponent B= reflects the increased effort required as the size of the project increases. can vary from 1.1 to 1.24.
- ➢ 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.

5. 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)
- support facilities (FCIL).

6.



This results in an effort computation as follows:

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

where:

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

<u>The reuse model</u>

1. software reuse is common, and most large systems include a significant percentage of code that is reused from previous developments.

2. The reuse model is used to estimate the effort required to integrate reusable or generated code.

3. 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 here.

4. **<u>Black-box code</u>**: 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.

- 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:

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

Where

AT=is the percentage of adapted code that is automatically generated

ATPROD=is the productivity of engineers in integrating such code.

(example: Boehm 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 X 30/100) / 2400 = 2.5 person months //Generated code example)

5. **White –box code**: 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 compute the effort, 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.

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

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

- 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.

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.

3. An assessment factor (referred to as AA) that represents the costs of reuse decisionmaking.

<u>**The post-architecture level**</u>

1. 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.

2. The estimates produced at the post-architecture level are based on the same basic formula ($PM = A \times Size^B \times M$) used in the early design estimates. In addition, a much more extensive set of product, process and organizational attributes (17 rather than 7) are used to refine the initial effort computation.

3. 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.

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

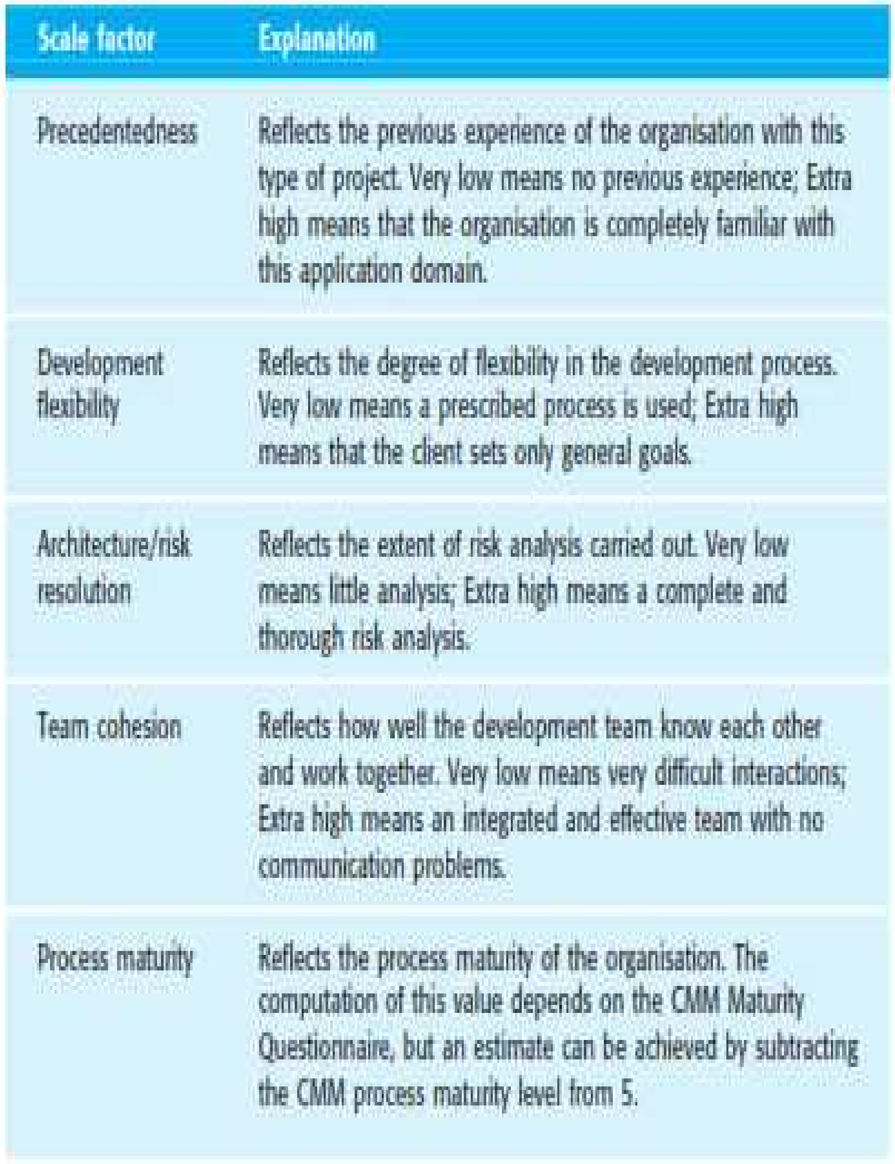
a). An estimate of the total number of lines of new code to be developed

b). An estimate of the equivalent number of source lines of code (ESLOC) calculated using the reuse model

c). An estimate of the number of lines of code that have to be modified because of changes to the requirements.

5. The exponent term (B) 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).

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.

6. Then add the ratings, divide them by 100 and add the result to 1.01 to get the exponent that should be used.

7. 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)

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

9. 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:

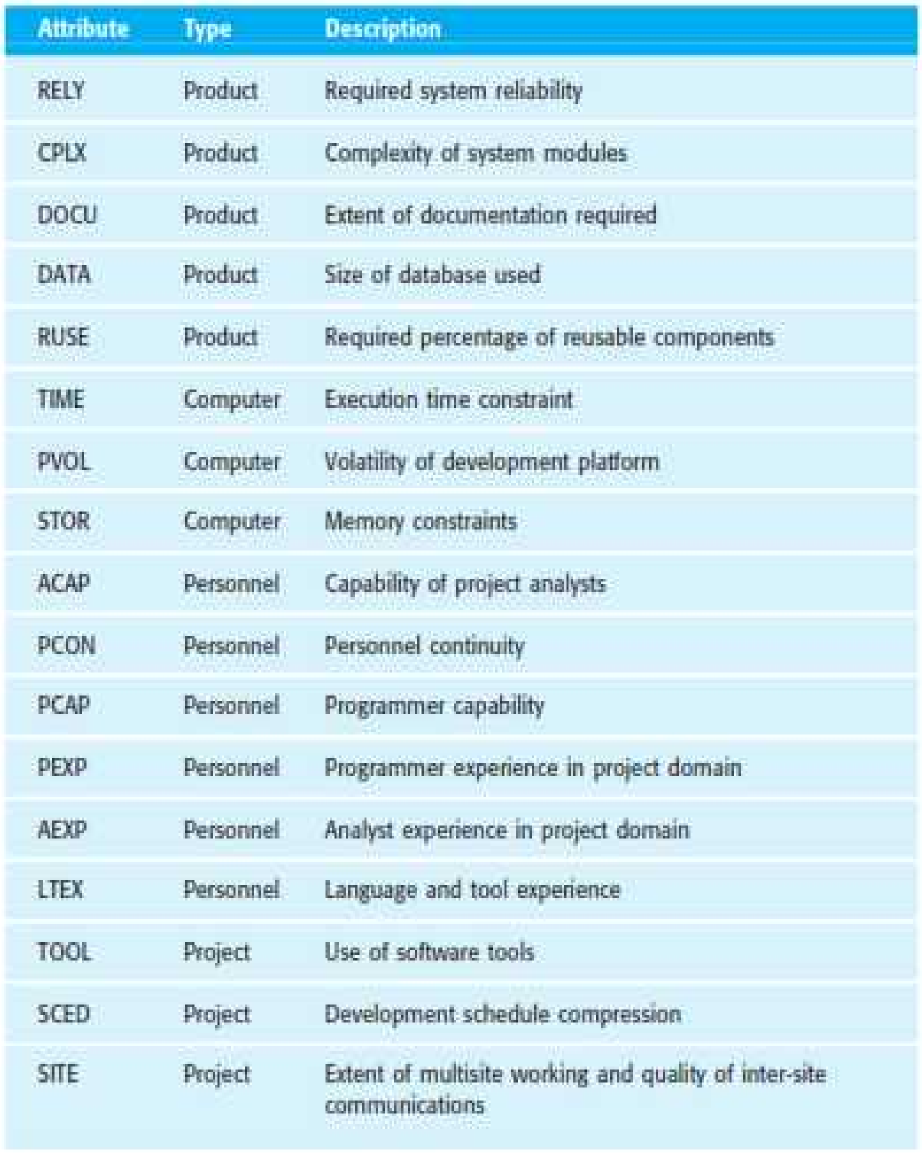
a). Product attributes are concerned with required characteristics of the software product being developed.

b). Computer attributes are constraints imposed on the software by the hardware platform.

c). Personnel attributes are multipliers that take the experience and capabilities of the people working on the project into account.

d). Project attributes are concerned with the particular characteristics of the software development project.

Figure 26.10 Project cost drivers



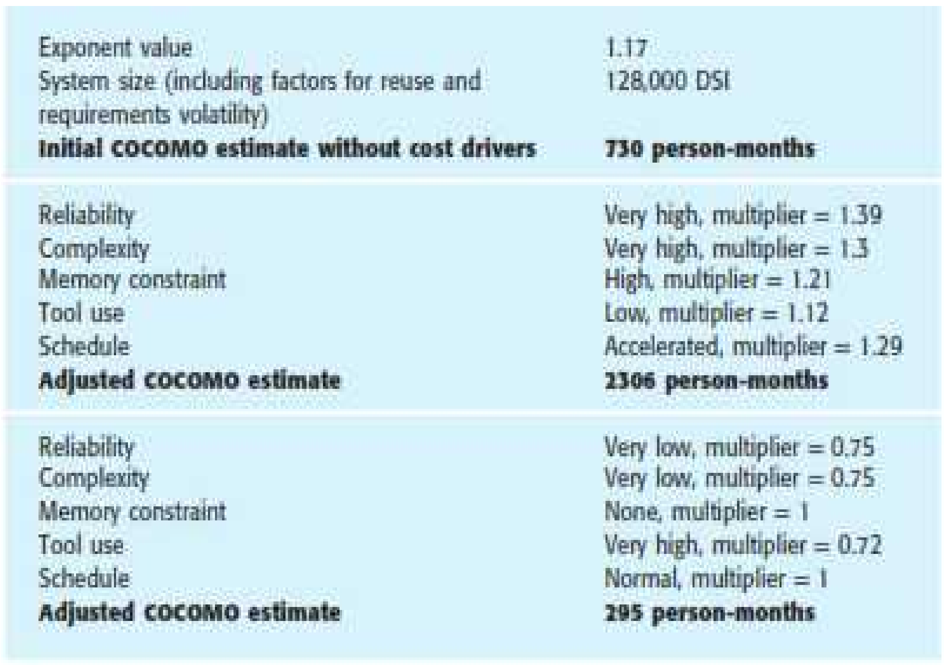
Attribute	Type	Description
RELY	Product	Required system reliability
CPLX	Product	Complexity of system modules
DOCU	Product	Extent of documentation required
DATA	Product	Size of database used
RUSE	Product	Required percentage of reusable components
TIME	Computer	Execution time constraint
PVOL	Computer	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

10. Figure 26.11 shows how these cost drivers influence effort estimates.

(Example: consider 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.

In Figure 26.11, assigned maximum and minimum values to the key cost drivers to show how they influence the effort estimate. 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 one third of the original. This highlights the vast differences between different types of project .)

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**

26.3.2 Algorithmic cost models in project planning

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

2. This is particularly important where we have to make hardware/software cost trade-offs and where we may have to recruit new staff with specific project skills.

3. Consider an embedded system to control an experiment that is to be launched into space.

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

  1. The cost of the target hardware to execute the system

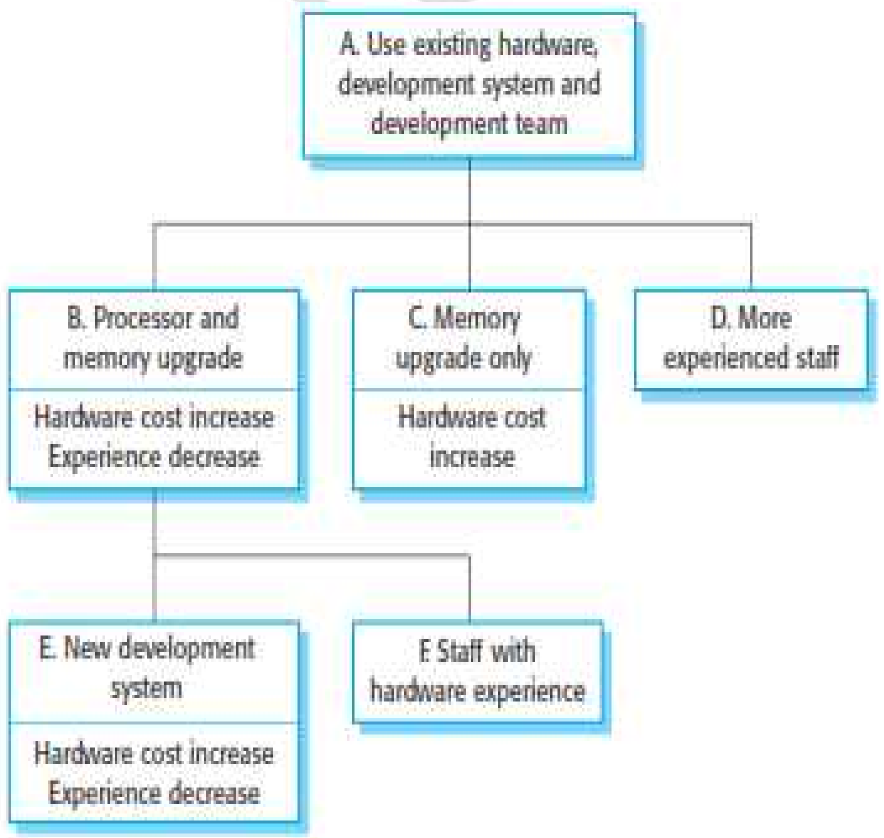
  2. The cost of the platform (computer plus software) to develop the system

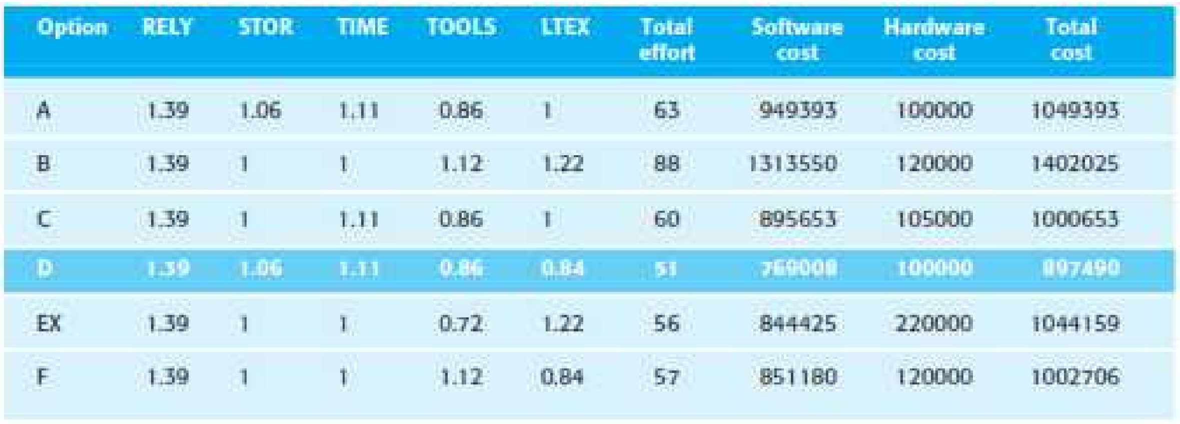
  3. The cost of the effort required to develop the software.

- Figure 26.13 shows 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:

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

Figure 26.12
Management options


A. Use existing hardware, development system and development team
B. Processor and memory upgrade
Hardware cost increase Experience decrease
C. Memory upgrade only
Hardware cost increase
D. More experienced staff
E. New development system
Hardware cost increase Experience decrease
F. Staff with hardware experience



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

26.4 Project duration and staffing

1. 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.

2. The development time for the project is called the project schedule. Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor's.

3. 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.

4. 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:

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

where

PM= is the effort computation

B= is the exponent computed

This computation predicts the nominal schedule for the project.

5. 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

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

where

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.

(Example:To illustrate the COCOMO development schedule computation, assume that 60 months of effort are estimated to develop a software system . Assume that the value of exponent B is 1.17. From the schedule equation, the time required to complete the project is:

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

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