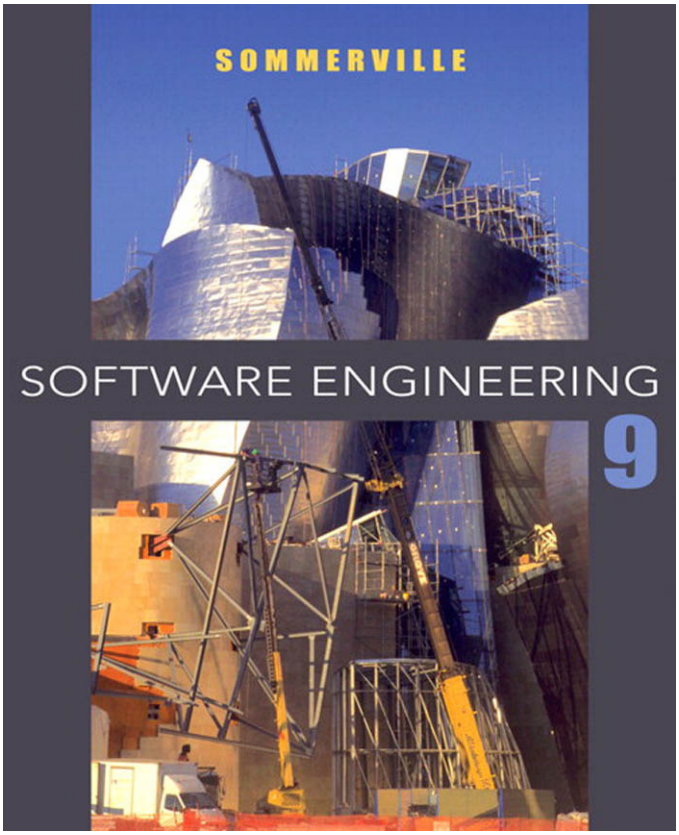

SOMMERVILLE
SOFTWARE ENGINEERING
9

SOFTWARE ENGINEERING

Ninth Edition

Ian Sommerville

Addison-Wesley

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Contents at a glance

PART 4 Software Management

It is sometimes suggested that the key difference between software engineering and other types of programming is that software engineering is a managed process. By this, I mean that the software development takes place within an organization and is subject to a range of schedule, budget, and organizational constraints. Management is therefore very important to software engineering. I introduce a range of management topics in this part of the book with a focus on technical management issues rather than 'softer' management issues such as people management, or the more strategic management of enterprise systems.

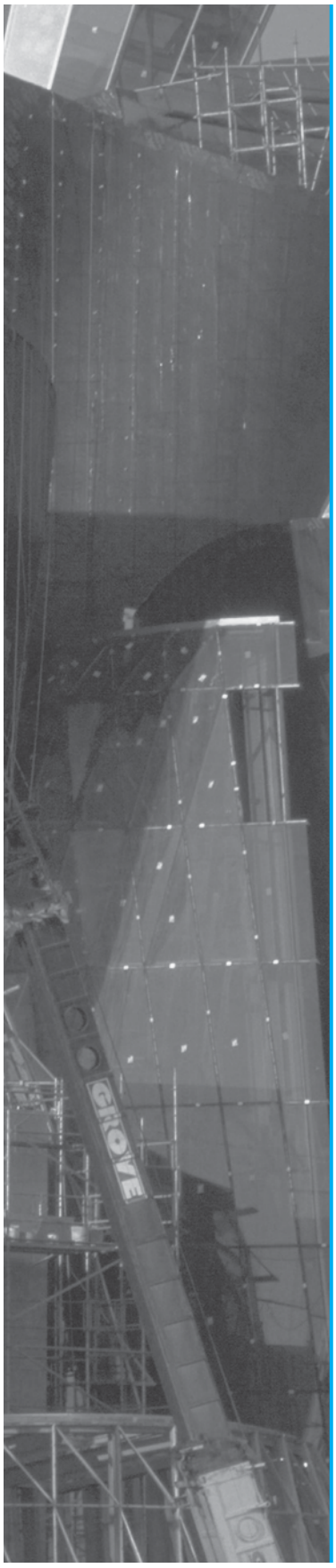
Chapter 22 introduces software project management and its first major section is concerned with risk management. Along with project planning, risk management, where managers identify what might go wrong and plan what they might do about it, is a key project management responsibility. This chapter also includes sections on people management and team working.

Chapter 23 covers project planning and estimation. I introduce bar charts as fundamental planning tools and explain why plan-driven development will remain an important development approach, in spite of the success of agile methods. I also discuss issues that influence the price charged for a system and techniques of software cost estimation. I use the COCOMO family of cost models to describe algorithmic cost modeling and explain the benefits and disadvantages of this approach.

Chapters 24 to 26 are concerned with issues of quality management. Quality management is concerned with processes and techniques for ensuring and improving the quality of software and I introduce this topic in Chapter 24. I discuss the importance of standards in quality management, the use of reviews and inspections in the quality assurance process, and the role of software measurement in quality management.

Chapter 25 discusses configuration management. This is an issue that is important for all large systems, which are developed by teams. However, the need for configuration management is not always obvious to students who have only been concerned with personal software development, so I describe the various aspects of this topic here, including configuration planning, version management, system building, and change management.

Finally, Chapter 26 covers software process improvement—how can processes be modified so that both product and process attributes are improved? I discuss the stages of a generic process improvement process, namely, process measurement, process analysis, and process change. I then go on to cover the SEI's capability-based approach to process improvement and briefly explain capability maturity models.



22

Project management

Objectives

The objective of this chapter is to introduce software project management and two important management activities, namely risk management and people management. When you have read the chapter you will:

- know the principal tasks of software project managers;
- have been introduced to the notion of risk management and some of the risks that can arise in software projects;
- understand factors that influence personal motivation and what these might mean for software project managers;
- understand key issues that influence team working, such as team composition, organization, and communication.

Contents

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: the software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers.

The success criteria for project management obviously vary from project to project but, for most projects, important goals are:

1. Deliver the software to the customer at the agreed time.
2. Keep overall costs within budget.
3. Deliver software that meets the customer's expectations.
4. Maintain a happy and well-functioning development team.

These goals are not unique to software engineering but are the goals of all engineering projects. However, software engineering is different from other types of engineering in a number of ways that make software management particularly challenging. Some of these differences are:

1. *The product is intangible* A manager of a shipbuilding or a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible—parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress by simply looking at the artifact that is being constructed. Rather, they rely on others to produce evidence that they can use to review the progress of the work.
2. *Large software projects are often 'one-off' projects* Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager's experience obsolete. Lessons learned from previous projects may not be transferable to new projects.
3. *Software processes are variable and organization-specific* The engineering process for some types of system, such as bridges and buildings, is well understood. However, software processes vary quite significantly from one organization to another. Although there has been significant progress in process standardization and improvement, we still cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project.

Because of these issues, it is not surprising that some software projects are late, over budget, and behind schedule. Software systems are often new and technically

innovative. Engineering projects (such as new transport systems) that are innovative often also have schedule problems. Given the difficulties involved, it is perhaps remarkable that so many software projects are delivered on time and to budget!

It is impossible to write a standard job description for a software project manager. The job varies tremendously depending on the organization and the software product being developed. However, most managers take responsibility at some stage for some or all of the following activities:

1. *Project planning* Project managers are responsible for planning, estimating and scheduling project development, and assigning people to tasks. They supervise the work to ensure that it is carried out to the required standards and monitor progress to check that the development is on time and within budget.
2. *Reporting* Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews.
3. *Risk management* Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise.
4. *People management* Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.
5. *Proposal writing* The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded. There can be no set guidelines for this task; proposal writing is a skill that you acquire through practice and experience.

In this chapter, I focus on risk management and people management. Project planning is an important topic in its own right, which I discuss in Chapter 23.

22.1 Risk management

Risk management is one of the most important jobs for a project manager. Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks

(Hall, 1998; Ould, 1999). You can think of a risk as something that you'd prefer not to have happen. Risks may threaten the project, the software that is being developed, or the organization. There are, therefore, three related categories of risk:

1. *Project risks* Risks that affect the project schedule or resources. An example of a project risk is the loss of an experienced designer. Finding a replacement designer with appropriate skills and experience may take a long time and, consequently, the software design will take longer to complete.
2. *Product risks* Risks that affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform as expected. This may affect the overall performance of the system so that it is slower than expected.
3. *Business risks* Risks that affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Of course, these risk types overlap. If an experienced programmer leaves a project this can be a project risk because, even if they are immediately replaced, the schedule will be affected. It inevitably takes time for a new project member to understand the work that has been done, so they cannot be immediately productive. Consequently, the delivery of the system may be delayed. The loss of a team member can also be a product risk because a replacement may not be as experienced and so could make programming errors. Finally, it can be a business risk because that programmer's experience may be crucial in winning new contracts.

You should record the results of the risk analysis in the project plan along with a consequence analysis, which sets out the consequences of the risk for the project, product, and business. Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage.

The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed. However, there are also common risks that are not related to the type of software being developed and these can occur in any project. Some of these common risks are shown in Figure 22.1.

Risk management is particularly important for software projects because of the inherent uncertainties that most projects face. These stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills. You have to anticipate risks; understand the impact of these risks on the project, the product, and the business; and take steps to avoid these risks. You may need to draw up contingency plans so that, if the risks do occur, you can take immediate recovery action.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Figure 22.1 Examples of common project, product, and business risks

An outline of the process of risk management is illustrated in Figure 22.2. It involves several stages:

1. *Risk identification* You should identify possible project, product, and business risks.
2. *Risk analysis* You should assess the likelihood and consequences of these risks.
3. *Risk planning* You should make plans to address the risk, either by avoiding it or minimizing its effects on the project.
4. *Risk monitoring* You should regularly assess the risk and your plans for risk mitigation and revise these when you learn more about the risk.

You should document the outcomes of the risk management process in a risk management plan. This should include a discussion of the risks faced by the project, an analysis of these risks, and information on how you propose to manage the risk if it seems likely to be a problem.

The risk management process is an iterative process that continues throughout the project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes available, you have to reanalyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.

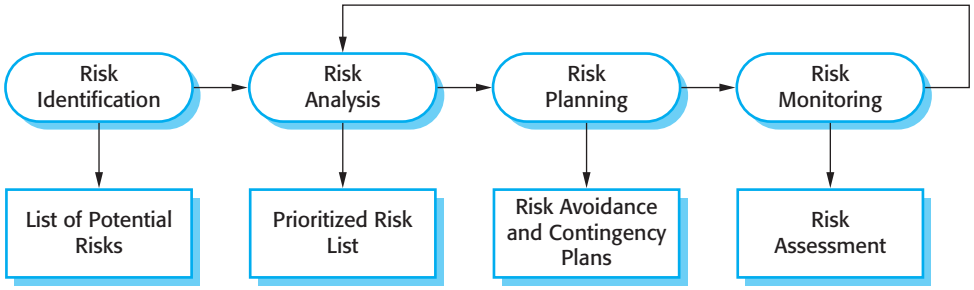


Figure 22.2 The risk management process

22.1.1 Risk identification

Risk identification is the first stage of the risk management process. It is concerned with identifying the risks that could pose a major threat to the software engineering process, the software being developed, or the development organization. Risk identification may be a team process where a team get together to brainstorm possible risks. Alternatively, the project manager may simply use his or her experience to identify the most probable or critical risks.

As a starting point for risk identification, a checklist of different types of risk may be used. There are at least six types of risk that may be included in a risk checklist:

1. *Technology risks* Risks that derive from the software or hardware technologies that are used to develop the system.
2. *People risks* Risks that are associated with the people in the development team.
3. *Organizational risks* Risks that derive from the organizational environment where the software is being developed.
4. *Tools risks* Risks that derive from the software tools and other support software used to develop the system.
5. *Requirements risks* Risks that derive from changes to the customer requirements and the process of managing the requirements change.
6. *Estimation risks* Risks that derive from the management estimates of the resources required to build the system.

Figure 22.3 gives some examples of possible risks in each of these categories. When you have finished the risk identification process, you should have a long list of risks that could occur and which could affect the product, the process, and the business. You then need to prune this list to a manageable size. If you have too many risks, it is practically impossible to keep track of all of them.

22.1.2 Risk analysis

During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk. There is no easy way to do this. You have to rely on your own judgment and experience of previous projects

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

Figure 22.3 Examples of different types of risks

and the problems that arose in them. It is not possible to make precise, numeric assessment of the probability and seriousness of each risk. Rather, you should assign the risk to one of a number of bands:

1. The probability of the risk might be assessed as very low (< 10%), low (10–25%), moderate (25–50%), high (50–75%), or very high (> 75%).
2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.

You should then tabulate the results of this analysis process using a table ordered according to the seriousness of the risk. Figure 22.4 illustrates this for the risks that I have identified in Figure 22.3. Obviously, the assessment of probability and seriousness is arbitrary here. To make this assessment, you need detailed information about the project, the process, the development team, and the organization.

Of course, both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and as risk management plans are implemented. Therefore, you should update this table during each iteration of the risk process.

Once the risks have been analyzed and ranked, you should assess which of these risks are most significant. Your judgment must depend on a combination of the probability of the risk arising and the effects of that risk. In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

Figure 22.4 Risk types and examples

Boehm (1988) recommends identifying and monitoring the top 10 risks, but I think that this figure is rather arbitrary. The right number of risks to monitor must depend on the project. It might be 5 or it might be 15. However, the number of risks chosen for monitoring should be manageable. A very large number of risks would simply require too much information to be collected. From the risks identified in Figure 22.4, it is appropriate to consider the 8 risks that have catastrophic or serious consequences (Figure 22.5).

22.1.3 Risk planning

The risk planning process considers each of the key risks that have been identified, and develops strategies to manage these risks. For each of the risks, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in the risk occurs. You also should think about information that you might need to collect while monitoring the project so that problems can be anticipated.

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Figure 22.5 Strategies to help manage risk

Again, there is no simple process that can be followed for contingency planning. It relies on the judgment and experience of the project manager.

Figure 22.5 shows possible risk management strategies that have been identified for the key risks (i.e., those that are serious or intolerable) shown in Figure 22.4. These strategies fall into three categories:

1. *Avoidance strategies* Following these strategies means that the probability that the risk will arise will be reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 22.5.
2. *Minimization strategies* Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 22.5.
3. *Contingency plans* Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 22.5.

You can see a clear analogy here with the strategies used in critical systems to ensure reliability, security, and safety, where you must avoid, tolerate, or recover from failures. Obviously, it is best to use a strategy that avoids the risk. If this is not possible, you should use a strategy that reduces the chances that the risk will have serious effects. Finally, you should have strategies in place to cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

Figure 22.6 Risk indicators

22.1.4 Risk monitoring

Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed. You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable. You should also think about whether or not the effects of the risk have changed. To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are obviously dependent on the types of risk. Figure 22.6 gives some examples of factors that may be helpful in assessing these risk types.

You should monitor risks regularly at all stages in a project. At every management review, you should consider and discuss each of the key risks separately. You should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

22.2 Managing people

The people working in a software organization are its greatest assets. It costs a lot to recruit and retain good people and it is up to software managers to ensure that the organization gets the best possible return on its investment. In successful companies and economies, this is achieved when people are respected by the organization and are assigned responsibilities that reflect their skills and experience.

It is important that software project managers understand the technical issues that influence the work of software development. Unfortunately, however, good software engineers are not necessarily good people managers. Software engineers often have strong technical skills but may lack the softer skills that enable them to

motivate and lead a project development team. As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

In my view, there are four critical factors in people management:

1. *Consistency* People in a project team should all be treated in a comparable way. No one expects all rewards to be identical but people should not feel that their contribution to the organization 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. In some cases, of course, you will find that people simply don't fit into a team and they cannot continue, but it is important not to jump to conclusions about this at an early stage in the project.
3. *Inclusion* People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a working environment where all views, even those of the most junior staff, are considered.
4. *Honesty* As a manager, you should always be honest about what is going well and what is going badly in the team. You should also be honest about your level of technical knowledge and willing to defer to staff with more knowledge when necessary. If you try to cover up ignorance or problems you will eventually be found out and will lose the respect of the group.

People management, in my view, is something that has to be based on experience, rather than learned from a book. My aim in this section and the following section on teamwork is simply to introduce some of the most important people and team management problems that affect software project management. I hope the material here will sensitize you to some of the problems that managers may encounter when dealing with teams of technically talented individuals.

22.2.1 Motivating people

As a project manager, you need to motivate the people that work with you so that they contribute to the best of their abilities. Motivation means organizing the work and the working environment to encourage people to work as effectively as possible. If people are not motivated, they will not be interested in the work they are doing. They will work slowly, be more likely to make mistakes, and will not contribute to the broader goals of the team or the organization.

To provide this encouragement, you should understand a little about what motivates people. Maslow (1954) suggests that people are motivated by satisfying their needs. These needs are arranged in a series of levels, as shown in Figure 22.7. The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, and the need to feel secure in an environment. Social needs are concerned with the need to feel part of a social grouping. Esteem needs represent the need to feel

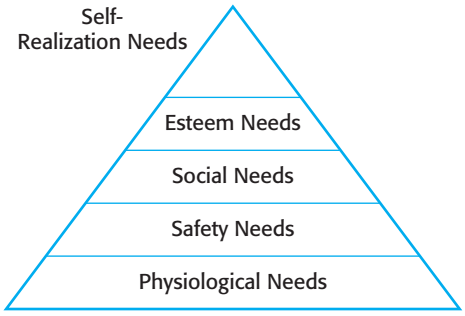


Figure 22.7 Human needs hierarchy

respected by others, and self-realization needs are concerned with personal development. People need to satisfy lower-level needs like hunger before the more abstract, higher-level needs.

People working in software development organizations are not usually hungry or thirsty or physically threatened by their environment. Therefore, making sure that people's social, esteem, and self-realization needs are satisfied is most important from a management point of view.

1. To satisfy social needs, you need to give people time to meet their co-workers and 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. They may work for different organizations or from home most of the time.

   Social networking systems and teleconferencing can be used to facilitate communications but my experience with electronic systems is that they are most effective once people know each other. You therefore need to arrange some face-to-face meetings early in the project so that people can directly interact with other members of the team. Through this direct interaction, people become part of a social group and accept the goals and priorities of that group.

2. To satisfy esteem needs, you need to show people that they are valued by the organization. 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.

3. Finally, to satisfy self-realization needs, you 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. Training is an important motivating influence as people like to gain new knowledge and learn new skills.

In Figure 22.8, I illustrate a problem of motivation that managers often have to face. In this example, a competent group member loses interest in the work and in

Case study: Motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of 6 developers than can develop new products based around the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team and creative new ideas are developed. The team decides to develop a peer-to-peer messaging system using digital televisions linked to the alarm network for communications. However, some months into the project, Alice notices that Dorothy, a hardware design expert, starts coming into work late, the quality of her work deteriorates and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials that there is a problem, Dorothy admits that she has l ost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity for this. Basically, she is working as a C programmer with other team members.

Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Figure 22.8 Individual motivation

the group as a whole. The quality of her work falls and becomes unacceptable. This situation has to be dealt with quickly. If you don't sort out the problem, the other group members will become dissatisfied and feel that they are doing an unfair share of the work.

In this example, Alice tries to find out if Dorothy's personal circumstances could be the problem. Personal difficulties commonly affect motivation because people cannot concentrate on their work. You may have to give them time and support to resolve these issues, although you also have to make it clear that they still have a responsibility to their employer.

Dorothy's motivation problem is one that is quite common when projects develop in an unexpected direction. People who expect to do one type of work may end up doing something completely different. This becomes a problem when team members want to develop their skills in a way that is different from that taken by the project. In those circumstances, you may decide that the team member should leave the team and find opportunities elsewhere. In this example, however, Alice decides to try to convince Dorothy that broadening her experience is a positive career step. She gives Dorothy more design autonomy and organizes training courses in software engineering that will give her more opportunities after her current project has finished.

The People Capability Maturity Model

The People Capability Maturity Model (P-CMM) is a framework for assessing how well organizations manage the development of their staff. It highlights best practice in people management and provides a basis for organizations to improve their people management processes.

http://www.SoftwareEngineering-9.com/Web/Management/P-CMM.html

Maslow's model of motivation is helpful up to a point but I think that a problem with it is that it takes an exclusively personal viewpoint on motivation. It does not take adequate account of the fact that people feel themselves to be part of an organization, a professional group, and one or more cultures. This is not simply a question of satisfying social needs—people can be motivated through helping a group achieve shared goals.

Being a member of a cohesive group is highly motivating for most people. People with fulfilling jobs often like to go to work because they are motivated by the people they work with and the work that they do. Therefore, as well as thinking about individual motivation you also have to think about how a group as a whole can be motivated to achieve the organization's goals. I discuss group management issues in the next section.

Personality type also influences motivation. Bass and Dunteman (1963) classify professionals into three types:

1. Task-oriented people, who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.
2. Self-oriented people, who 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 and they wish to be successful in their work to help realize these goals.
3. Interaction-oriented people, who are motivated by the presence and actions of co-workers. As software development becomes more user-centered, interaction-oriented individuals are becoming more involved in software engineering.

Interaction-oriented personalities usually like to work as part of a group, whereas task-oriented and self-oriented people usually prefer to act as individuals. Women are more likely to be interaction-oriented than men. They are often more effective communicators. I discuss the mix of these different personality types in groups in the case study in Figure 22.10.

Each individual's motivation is made up of elements of each class but one type of motivation is usually dominant at any one time. However, individuals can change. For example, technical people who feel they are not being properly rewarded can become self-oriented and put personal interests before technical concerns. If a group works particularly well, self-oriented people can become more interaction-oriented.

22.3 Teamwork

Most professional software is developed by project teams that range in size from two to several hundred people. However, as it is clearly impossible for everyone in a large group to work together on a single problem, large teams are usually split into a number of groups. Each group is responsible for developing part of the overall system. As a general rule, software engineering project groups should not have more than 10 members. When small groups are used, communication problems are reduced. Everyone knows everyone else and the whole group can get around a table for a meeting to discuss the project and the software that they are developing.

Putting together a group that has the right balance of technical skills, experience, and personalities is a critical management task. However, successful groups are more than simply a collection of individuals with the right balance of skills. A good group is cohesive and has a team spirit. The people involved are motivated by the success of the group as well as by their own personal goals.

In a cohesive group, members think of the group as more important than the individuals who are group members. Members of a well-led, cohesive group are loyal to the group. They identify with group goals and other group members. 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.

The benefits of creating a cohesive group are:

1. *The group can establish its own quality standards* Because these standards are established by consensus, they are more likely to be observed than external standards imposed on the group.
2. *Individuals learn from and support each other* People in the group learn from each other. Inhibitions caused by ignorance are minimized as mutual learning is encouraged.
3. *Knowledge is shared* 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.
4. *Refactoring and continual improvement is encouraged* Group members work collectively to deliver high-quality results and fix problems, irrespective of the individuals who originally created the design or program.

Case study: Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As they are developing a new product, she takes the opportunity of involving all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She also encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an 'away day' for the group where the team spends two days on 'technology updating'. Each team member prepares an update on a relevant technology and presents it to the group. This is an off-site meeting in a good hotel and plenty of time is scheduled for discussion and social interaction.

Figure 22.9 Group cohesion

Good project managers should always try to encourage group cohesiveness. They may organize social events for group members and their families, try to establish a sense of group identity by naming the group and establishing a group identity and territory, or they may get involved in explicit group-building activities such as sports and games.

One of the most effective ways of promoting cohesion is to be inclusive. This means that you should treat group members as responsible and trustworthy, and make information freely available. Sometimes, managers feel that they cannot reveal certain information to everyone in the group. This invariably creates a climate of mistrust. Simple information exchange is an effective way of making people feel valued and that they are part of a group.

You can see an example of this in the case study in Figure 22.9. Alice arranges regular informal meetings where she tells the other group members what is going on. She makes a point of involving people in the product development by asking them to come up with new ideas derived from their own family experiences. The 'away days' are also good ways of promoting cohesion—people relax together while they help each other learn about new technologies.

Whether or not a group is effective depends, to some extent, on the nature of the project and the organization doing the work. If an organization is in a state of turmoil with constant reorganizations and job insecurity, it is very difficult for team members to focus on software development. However, apart from project and organizational issues, there are three generic factors that affect team working:

1. *The people in the group* You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing, and documentation.

2. *The group organization* A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.
3. *Technical and managerial communications* Good communications between group members, and between the software engineering team and other project stakeholders, is essential.

As with all management issues, getting the right team cannot guarantee project success. Too many other things can go wrong, including changes to the business and the business environment. However, if you don't pay attention to group composition, organization, and communications, you increase the likelihood that your project will run into difficulties.

22.3.1 Selecting group members

A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively. This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively. Sometimes, people are hired from outside the organization; more often, however, software engineering groups are put together from current employees who have experience on other projects. However, managers rarely have a completely free hand in team selection. They often have to use the people who are available in the company, even when they may not be the ideal people for the job.

As I discussed in Section 22.2.1, many software engineers are motivated primarily by their work. Software development groups, therefore, are often composed of people who have their own ideas about how technical problems should be solved. This is reflected in regularly reported problems of interface standards being ignored, systems being redesigned as they are coded, unnecessary system embellishments, and so on.

A group that has complementary personalities may work better than a group that is selected solely on technical ability. People who are motivated by the work are likely to be the strongest technically. People who are self-oriented will probably be best at pushing the work forward to finish the job. People who are interaction-oriented help facilitate communications within the group. I think that it is particularly important to have interaction-oriented people in a group. They like to talk to people and can detect tensions and disagreements at an early stage, before these have a serious impact on the group.

In the case study in Figure 22.10, I have suggested how Alice, the project manager, has tried to create a group with complementary personalities. This particular group has a good mix of interaction- and task-oriented people but I have already discussed, in Figure 22.8, how Dorothy's self-oriented personality has caused problems because she has not been doing the work that she expected. Fred's part-time role in the group as a domain expert might also be a problem. He is mostly interested in

Case study: Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Bob—task-oriented
Carol—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fred—task-oriented

Figure 22.10 Group composition

technical challenges, so he may not interact well with other group members. The fact that he is not always part of the team means that he may not relate well to the team's goals.

It is sometimes impossible to choose a group with complementary personalities. If this is the case, the project manager has to control the group so that individual goals do not take precedence over organizational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely when group members are given instructions without being aware of the part that their task plays in the overall project.

For example, say a software engineer is given a program design for coding and notices what appears to be possible improvements that could be made to the design. If he or she implements these improvements without understanding the rationale for the original design, any changes, though well intentioned, 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. They may then identify with these decisions rather than oppose them.

22.3.2 Group organization

The way that a group is organized affects the decisions that are made by that group, the ways that information is exchanged, and the interactions between the development group and external project stakeholders. Important organizational questions for project managers include:

1. Should the project manager be the technical leader of the group? The technical leader or system architect is responsible for the critical technical decisions made

Hiring the right people

Project managers are often responsible for selecting the people in the organization who will join their software engineering team. Getting the best possible people in this process is very important as poor selection decisions may be a serious risk to the project.

Key factors that should influence the selection of staff are education and training, application domain and technology experience, communication ability, adaptability, and problem-solving ability.

http://www.SoftwareEngineering-9.com/Web/Management/Selection.html

during software development. Sometimes, the project manager has the skill and experience to take on this role. However, for large projects, it is best to appoint a senior engineer to be the project architect, who will take responsibility for technical leadership.

2. Who will be involved in making critical technical decisions, and how will these be made? Will decisions be made by the system architect, the project manager, or by reaching consensus amongst a wider range of team members?
3. How will interactions with external stakeholders and senior company management be handled? In many cases, the project manager will be responsible for these interactions, assisted by the system architect if there is one. However, an alternative organizational model is to create a dedicated role concerned with external liaison, and appoint someone with appropriate interaction skills to that role.
4. How can groups integrate people who are not colocated? It is now common for groups to include members from different organizations and people to work from home as well as in a shared office. This has to be taken into account in group decision-making processes.
5. How can knowledge be shared across the group? Group organization affects information sharing as certain methods of organization are better for sharing than others. However, you should avoid too much information sharing as people become overloaded and excessive information distracts them from their work.

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

Extreme programming groups (Beck, 2000) are always informal groups. XP enthusiasts claim that formal structure inhibits information exchange. In XP, many

decisions that are usually seen as management decisions (such as decisions on schedule) are devolved to group members. Programmers work together in pairs to develop code and take joint responsibility for the programs that are developed.

Informal groups can be very successful, particularly when most group members are experienced and competent. Such a group makes decisions by consensus, which improves cohesiveness and performance. However, if a group is composed mostly of inexperienced or incompetent members, informality can be a hindrance because no definite authority exists to direct the work, causing a lack of coordination between group members and, possibly, eventual project failure.

Hierarchical groups are groups that have a hierarchical structure with the group leader at the top of the hierarchy. He or she has more formal authority than the group members and so can direct their work. There is a clear organizational structure and decisions are made towards the top of the hierarchy and implemented by people lower down the hierarchy. Communications are primarily instructions from senior staff and there is relatively little 'upward' communication from the lower levels to the upper levels in the hierarchy.

This approach can work well when a well-understood problem can be easily broken into subproblems with subproblem solutions developed in different parts of the hierarchy. In those situations, relatively little communication across the hierarchy is required. However, such situations are relatively rare in software engineering for the following reasons:

1. Changes to the software often require changes to several parts of the system and this requires discussion and negotiation at all levels in the hierarchy.
2. Software technologies change so fast that more junior staff often know more about the technology than experienced staff. Top-down communications may mean that the project manager does not find out about the opportunities of using new technologies. More junior staff may become frustrated because of what they see as old-fashioned technologies being used for development.

Democratic and hierarchic group organizations do not formally recognize that there may be very large differences in technical ability between group members. The best programmers may be up to 25 times more productive as the worst programmers. It makes sense to use the best people in the most effective way and to provide them with as much support as possible. An early organizational model that was intended to provide this support was the chief programmer team.

To make the most effective use of highly skilled programmers, Baker (1972) and others (Aron, 1974; Brooks, 1975) suggested that teams should be built around an individual, highly skilled chief programmer. The underlying principle of the chief programmer team is that skilled and experienced staff should be responsible for all software development. They should not be concerned with routine matters and should have good technical and administrative support for their work. They should focus on the software to be developed and not spend a lot of time in external meetings.

The physical work environment

The environment in which people work affects both group communications and individual productivity. Individual workspaces are better for concentration on detailed technical work as people are less likely to be distracted by interruptions. However, shared workspaces are better for communications. A well-designed work environment takes both of these needs into account.

http://www.SoftwareEngineering-9.com/Web/Management/workspace.html

However, the chief programmer team organization is, in my view, overdependent on the chief programmer and their assistant. Other team members who are not given sufficient responsibility may become demotivated because they feel their skills are underused. They do not have the information to cope if things go wrong and are not given the opportunity to participate in decision making. There are significant project risks associated with this group organization and these may outweigh any benefits that this kind of organization might bring.

22.3.3 Group communications

It is absolutely essential that group members communicate effectively and efficiently with each other and with other project stakeholders. Group members must exchange information on the status of their work, the design decisions that have been made, and changes to previous design decisions. They have to resolve problems that arise with other stakeholders and inform these stakeholders of changes to the system, the group, and delivery plans. Good communication also helps strengthen group cohesiveness. Group members come to understand the motivations, strengths, and weaknesses of other people in the group.

The effectiveness and efficiency of communications is influenced by:

1. *Group size* As a group gets bigger, it gets harder for members to communicate effectively. The number of one-way communication links is $n * (n - 1)$, where n is the group size, so, with a group of eight members, there are 56 possible communication pathways. This means that it is quite possible that some people will rarely communicate with each other. Status differences between group members mean that communications are often one-way. Managers and experienced engineers tend to dominate communications with less experienced staff, who may be reluctant to start a conversation or make critical remarks.
2. *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. This is a particular problem in a

large project with several development groups. If people working on different subsystems only communicate through their managers, then there are more likely to be delays and misunderstandings.

3. *Group composition* People with the same personality types (discussed in Section 22.2) may clash and, as a result, communications can be inhibited. Communication is also usually better in mixed-sex groups (Marshall and Heslin, 1975) than in single-sex groups. Women are often more interaction-oriented than men and may act as interaction controllers and facilitators for the group.

4. *The physical work environment* The organization of the workplace is a major factor in facilitating or inhibiting communications. See the book's webpage for more information.

5. *The available communication channels* There are many different forms of communication—face-to-face, e-mail messages, formal documents, telephone, and Web 2.0 technologies such as social networking and wikis. As project teams become increasingly distributed, with team members working remotely, you need to make use of a range of technologies to facilitate communications.

Project managers usually work to tight deadlines and, consequently, they may try to use communication channels that don't take up too much of their time. They may therefore rely on meetings and formal documents to pass on information to project staff and stakeholders. Although this may be an efficient approach to communication from a project manager's perspective, it is not usually very effective. There are often good reasons why people can't attend meetings and so they don't hear the presentation. Long documents are often never read because readers don't know if the documents are relevant. When several versions of the same document are produced, readers find it difficult to keep track of the changes.

Effective communication is achieved when communications are two way, and the people involved can discuss issues and information and establish a common understanding of proposals and problems. This can be done through meetings, although these are often dominated by powerful personalities. It is sometimes impractical to arrange meetings at short notice. More and more project teams include remote members, which also makes meetings more difficult.

To counter these problems, you may make use of web technologies such as wikis and blogs to support information exchange. Wikis support the collaborative creation and editing of documents, and blogs support threaded discussions about questions and comments made by group members. Wikis and blogs allow project members and external stakeholders to exchange information, irrespective of their location. They help manage information and keep track of discussion threads, which often become confusing when conducted by e-mail. You can also use instant messaging and teleconferences, which can be easily arranged, to resolve issues that need discussion.

KEY POINTS

- Good software project management is essential if software engineering projects are to be developed on schedule and within budget.
- Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative so there is no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
- Risk management is now recognized as one of the most important project management tasks.
- Risk management involves identifying and assessing major project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage, or deal with likely risks if or when they arise.
- People are motivated by interaction with other people, the recognition of management and their peers, and by being given opportunities for personal development.
- Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized, and the communication between group members.
- Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities, and available communication channels.

FURTHER READING

The Mythical Man Month (Anniversary Edition). The problems of software management remain largely unchanged since the 1960s and this is one of the best books on the topic. An interesting and readable account of the management of one of the first very large software projects, the IBM OS/360 operating system. The anniversary edition (published 20 years after the original edition in 1975) includes other classic papers by Brooks. (F. P. Brooks, 1995, Addison-Wesley.)

Software Project Survival Guide. This is a very pragmatic account of software management that contains good practical advice for project managers with a software engineering background. It is easy to read and understand. (S. McConnell, 1998, Microsoft Press.)

Peopleware: Productive Projects and Teams, 2nd edition. This is a new edition of the classic book on the importance of treating people properly when managing software projects. It is one of the few books that recognizes the importance of the place where people work. Strongly recommended. (T. DeMarco and T. Lister, 1999, Dorset House.)

Waltzing with Bears: Managing Risk on Software Projects. A very practical and easy-to-read introduction to risks and risk management. (T. DeMarco and T. Lister, 2003, Dorset House.)

EXERCISES

22.1. Explain why the intangibility of software systems poses special problems for software project management.

22.2. Explain why the best programmers do not always make the best software managers. You may find it helpful to base your answer on the list of management activities in Section 22.1.

22.3. Using reported instances of project problems in the literature, list management difficulties and errors that occurred in these failed programming projects. (I suggest that you start with *The Mythical Man Month*, by Fred Brooks)

22.4. In addition to the risks shown in Figure 22.1, identify at least six other possible risks that could arise in software projects.

22.5. Fixed-price contracts, where the contractor bids a fixed price to complete a system development, may be used to move project risk from client to contractor. If anything goes wrong, the contractor has to pay. Suggest how the use of such contracts may increase the likelihood that product risks will arise.

22.6. Explain why keeping all members of a group informed about progress and technical decisions in a project can improve group cohesiveness.

22.7. What problems do you think might arise in extreme programming teams where many management decisions are devolved to the team members?

22.8. Write a case study in the style used here to illustrate the importance of communications in a project team. Assume that some team members work remotely and it is not possible to get the whole team together at short notice.

22.9. You are asked by your manager to deliver software to a schedule that you know can only be met by asking your project team to work unpaid overtime. All team members have young children. Discuss whether you should accept this demand from your manager or whether you should persuade your team to give their time to the organization rather than to their families. What factors might be significant in your decision?

22.10. As a programmer, you are offered promotion to a project management position but you feel that you can make a more effective contribution in a technical rather than a managerial role. Discuss whether you should accept the promotion.

REFERENCES

Aron, J. D. (1974). *The Program Development Process*. Reading, Mass.: Addison-Wesley.

Baker, F. T. (1972). 'Chief Programmer Team Management of Production Programming'. *IBM Systems J.*, 11 (1), 56–73.

Bass, B. M. and Dunteman, G. (1963). 'Behaviour in groups as a function of self, interaction and task orientation'. *J. Abnorm. Soc. Psychology.,* **66** (4), 19–28.

Beck, K. (2000). *extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. W. (1988). 'A Spiral Model of Software Development and Enhancement'. *IEEE Computer,* **21** (5), 61–72.

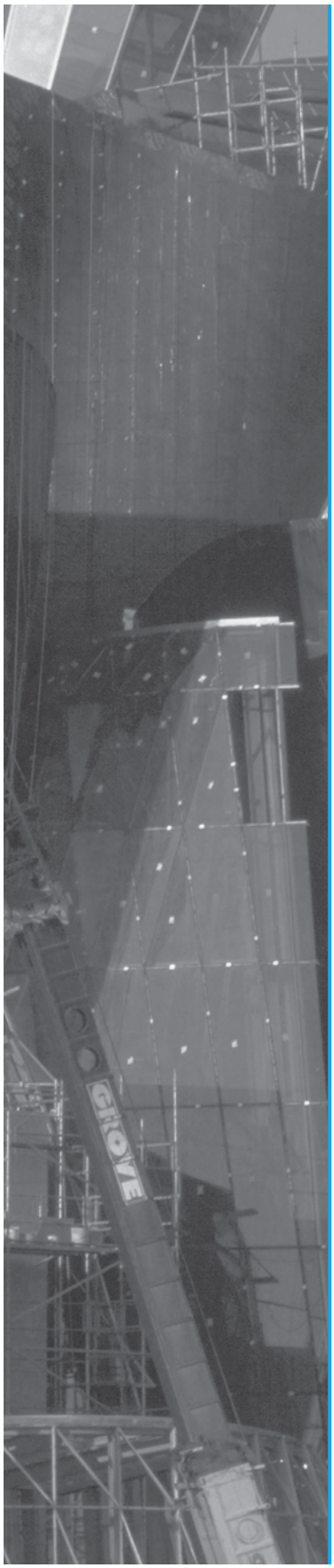
Brooks, F. P. (1975). *The Mythical Man Month*. Reading, Mass.: Addison-Wesley.

Hall, E. (1998). *Managing Risk: Methods for Software Systems Development*. Reading, Mass.: Addison-Wesley.

Marshall, J. E. and Heslin, R. (1975). 'Boys and Girls Together. Sexual composition and the effect of density on group size and cohesiveness'. *J. of Personality and Social Psychology,* **35** (5), 952–61.

Maslow, A. A. (1954). *Motivation and Personality*. New York: Harper and Row.

Ould, M. (1999). *Managing Software Quality and Business Risk*. Chichester: John Wiley & Sons.



23

Project planning

Objectives

The objective of this chapter is to introduce project planning, scheduling, and cost estimation. When you have read the 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;
- know what sections should be included in a project plan that is created within a plan-driven development process;
- understand what is involved in project scheduling and the use of bar charts to present a project schedule;
- have been introduced to the 'planning game', which is used to support project planning in extreme programming;
- understand how the COCOMO II model can be used for algorithmic cost estimation.

Contents

Project planning is one of the most important jobs of a software project manager. As a manager, you have to break down the work into parts and assign these to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems. The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.

Project planning takes place at three stages in a project life cycle:

1. At the proposal stage, when you are bidding for a contract to develop or provide a software system. You need a plan at this stage to help you decide if you have the resources to complete the work and to work out the price that you should quote to a customer.
2. During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc. Here, you have more information than at the proposal stage, and can therefore refine the initial effort estimates that you have prepared.
3. Periodically throughout the project, when you modify your plan in light of experience gained and information from monitoring the progress of the work. You learn more about the system being implemented and capabilities of your development team. This information allows you to make more accurate estimates of how long the work will take. Furthermore, the software requirements are likely to change and this usually means that the work breakdown has to be altered and the schedule extended. For traditional development projects, this means that the plan created during the startup phase has to be modified. However, when an agile approach is used, plans are shorter term and continually change as the software evolves. I discuss agile planning in Section 23.4.

Planning at the proposal stage is inevitably speculative, as you do not usually have a complete set of requirements for the software to be developed. Rather, you have to respond to a call for proposals based on a high-level description of the software functionality that is required. A plan is often a required part of a proposal, so you have to produce a credible plan for carrying out the work. If you win the contract, you then usually have to replan the project, taking into account changes since the proposal was made.

When you are bidding for a contract, you have to work out the price that you will propose to the customer for developing the software. As a starting point for calculating this price, you need to draw up an estimate of your costs for completing the project work. Estimation involves working out how much effort is required to complete each activity and, from this, calculating the total cost of activities. You should always calculate software costs objectively, with the aim of accurately predicting the cost of developing the software. Once you have a reasonable estimate of the likely costs, you are then in a position to calculate the price that you will quote to

Overhead costs

When you estimate the costs of effort on a software project, you don't simply multiply the salaries of the people involved by the time spent on the project. You have to take into account all of the organizational overheads (office space, administration, etc.) that must be covered by the income from a project. You calculate the costs by computing these overheads and adding a proportion to the costs of each engineer working on a project.

http://www.SoftwareEngineering-9.com/Web/Planning/overheadcosts.html

the customer. As I discuss in the next section, many factors influence the pricing of a software project—it is not simply cost + profit.

There are three main parameters that you should use when computing the costs of a software development project:

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

For most projects, the biggest cost is the effort cost. You have to estimate the total effort (in person-months) that is likely to be required to complete the work of a project. Obviously, you have limited information to make such an estimate, so you have to make the best possible estimate and then add significant contingency (extra time and effort) in case your initial estimate is optimistic.

For commercial systems, you normally use commodity hardware, which is relatively cheap. However, software costs can be significant if you have to license middleware and platform software. Extensive travel may be needed when a project is developed at different sites. Although travel costs themselves are usually a small fraction of the effort costs, the time spent traveling is often wasted and adds significantly to the effort costs of the project. Electronic meeting systems and other software that supports remote collaboration can reduce the amount of travel required. The time saved can be devoted to more productive project work.

Once a contract to develop a system has been awarded, the outline project plan for the project has to be refined to create a project startup plan. At this stage, you should know more about the requirements for this system. However, you may not have a complete requirements specification, especially if you are using an agile approach to development. Your aim at this stage should be to create a project plan that can be used to support decision making about project staffing and budgeting. You use the plan as a basis for allocating resources to the project from within the organization and to help decide if you need to hire new staff.

The plan should also define project monitoring mechanisms. You must keep track of the progress of the project and compare actual and planned progress and costs. Although most organizations have formal procedures for monitoring, a good

manager should be able to form a clear picture of what is going on through informal discussions with project staff. Informal monitoring can predict potential project problems by revealing difficulties as they occur. For example, daily discussions with project staff might reveal a particular problem in finding a software fault. Rather than waiting for a schedule slippage to be reported, the project manager could then immediately assign an expert to the problem, or decide to program around it.

The project plan always evolves during the development process. Development planning is intended to ensure that the project plan remains a useful document for staff to understand what is to be achieved and when it is to be delivered. Therefore, the schedule, cost estimate, and risks all have to be revised as the software is developed.

If an agile method is used, there is still a need for a project startup plan, as regardless of the approach used, the company still needs to plan how resources will be allocated to a project. However, this is not a detailed plan and should include only limited information about the work breakdown and project schedule. During development, an informal project plan and effort estimates are drawn up for each release of the software, with the whole team involved in the planning process.

23.1 Software pricing

In principle, the price of a software product to a customer is simply the cost of development plus profit for the developer. In practice, however, the relationship between the project cost and the price quoted to the customer is not usually so simple. When calculating a price, you should take broader organizational, economic, political, and business considerations into account, such as those shown in Figure 23.1. You need to think about organizational concerns, the risks associated with the project, and the type of contract that will be used. These may cause the price to be adjusted upwards or downwards. Because of the organizational considerations involved, deciding on a project price should be a group activity involving marketing and sales staff, senior management, and project managers.

To illustrate some of the project pricing issues, consider the following scenario:

A small software company, PharmaSoft, employs 10 software engineers. It has just finished a large project but only has contracts in place that require five development staff. However, it is bidding for a very large contract with a major pharmaceutical company that requires 30 person-years of effort over two years. The project will not start for at least 12 months but, if granted, it will transform the finances of the company.

PharmaSoft gets an opportunity to bid on a project that requires six 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, PharmaSoft decides to bid a price to the customer of $0.8 million.

Factor	Description
Market opportunity	A development organization 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 organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a 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 organization 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. Cash flow is more important than profit in difficult economic times.

Figure 23.1 Factors affecting software pricing

This means that, although it loses money on this contract, it can retain specialist staff for the more profitable future projects that are likely to come on stream in a year's time.

As the cost of a project is only loosely related to the price quoted to a customer, 'pricing to win' is a commonly used strategy. Pricing to win means that a company has some idea of the price that the customer expects to pay and makes a bid for the contract based on the customer's expected price. This may seem unethical and unbusinesslike, but it does have advantages for both the customer and the system provider.

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 (OilSoft) is bidding for a contract to develop a 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 OilSoft estimates that a price of $900,000 is likely to be competitive and within the oil company's budget. After they are granted the contract, OilSoft then negotiates the detailed requirements of the system so that basic functionality is delivered. They then 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 high initial software cost.

23.2 Plan-driven development

Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail. A project plan is created that records the work to be done, who will do it, the development schedule, and the work products. Managers use the plan to support project decision making and as a way of measuring progress. Plan-driven development is based on engineering project management techniques and can be thought of as the 'traditional' way of managing large software development projects. This contrasts with agile development, where many decisions affecting the development are delayed and made later, as required, during the development process.

The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used. Delaying such decisions is sensible because it avoids unnecessary rework. The arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be closely taken into account, and that potential problems and dependencies are discovered before the project starts, rather than once the project is under way.

In my view, the best approach to project planning involves a judicious mixture of plan-based and agile development. The balance depends on the type of project and skills of the people who are available. At one extreme, large security and safety-critical systems require extensive up-front analysis and may have to be certified before they are put into use. These should be mostly plan-driven. At the other extreme, small to medium-size information systems, to be used in a rapidly changing competitive environment, should be mostly agile. Where several companies are involved in a development project, a plan-driven approach is normally used to coordinate the work across each development site.

23.2.1 Project plans

In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work. The plan should identify risks to the project and the software under development, and the approach that is taken to risk management. Although the specific details of project plans vary depending on the type of project and organization, plans normally include the following sections:

1. *Introduction* This briefly describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project.

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Staff development plan	Describes how the skills and experience of the project team members will be developed.

Figure 23.2 Project plan supplements

2. *Project organization* This describes the way in which the development team is organized, the people involved, and their roles in the team.
3. *Risk analysis* This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed. I have covered risk management in Chapter 22.
4. *Hardware and software resource requirements* This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.
5. *Work breakdown* This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity. Milestones are key stages in the project where progress can be assessed; deliverables are work products that are delivered to the customer.
6. *Project schedule* This shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities. The ways in which the schedule may be presented are discussed in the next section of the chapter.
7. *Monitoring and reporting mechanisms* This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

As well as the principal project plan, which should focus on the risks to the projects and the project schedule, you may develop a number of supplementary plans to support other process activities such as testing and configuration management. Examples of possible supplementary plans are shown in Figure 23.2.

23.2.2 The planning process

Project planning is an iterative process that starts when you create an initial project plan during the project startup phase. Figure 23.3 is a UML activity diagram that

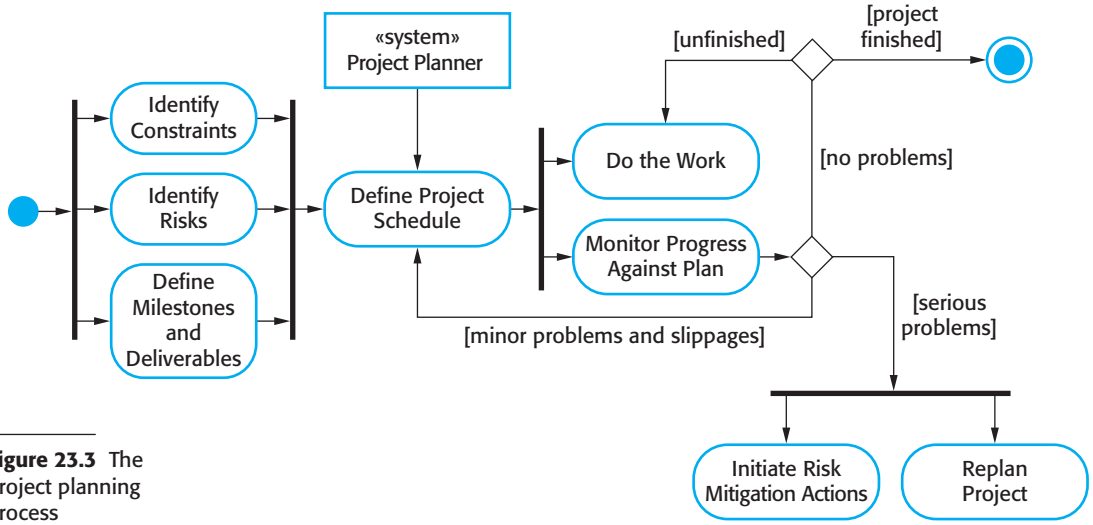


Figure 23.3 The project planning process

shows a typical workflow for a project planning process. Plan changes are inevitable. As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule, and risk changes. Changing business goals also leads to changes in project plans. As business goals change, this could affect all projects, which may then have to be replanned.

At the beginning of a planning process, you should assess the constraints affecting the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on. In conjunction with this, you should also identify the project milestones and deliverables. Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing. Deliverables are work products that are delivered to the customer (e.g., a requirements document for the system).

The process then enters a loop. You draw up an estimated schedule for the project and the activities defined in the schedule are initiated or given permission to continue. After some time (usually about two to three weeks), you should review progress and note discrepancies from the planned schedule. Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and you will have to make modifications to the original plan.

It is important to be realistic when you are creating a project plan. Problems of some description nearly always arise during a project, and these can lead to project delays. Your initial assumptions and scheduling should therefore be pessimistic rather than optimistic. There should be sufficient contingency built into your plan so that the project constraints and milestones don't need to be renegotiated every time you go around the planning loop.

If there are serious problems with the development work that are likely to lead to significant delays, you need to initiate risk mitigation actions to reduce the risks of project failure. In conjunction with these actions, you also have to replan the project.

This may involve renegotiating the project constraints and deliverables with the customer. A new schedule of when work should be completed also has to be established and agreed with the customer.

If this renegotiation is unsuccessful or the risk mitigation actions are ineffective, then you should arrange for a formal project technical review. The objectives of this review are to find an alternative approach that will allow the project to continue, and to check whether the project and the goals of the customer and software developer are still aligned.

The outcome of a review may be a decision to cancel a project. This may be a result of technical or managerial failings but, more often, is a consequence of external changes that affect the project. The development time for a large software project is often several years. During that time, the business objectives and priorities inevitably change. These changes may mean that the software is no longer required or that the original project requirements are inappropriate. Management may then decide to stop software development or to make major changes to the project to reflect the changes in the organizational objectives.

23.3 Project scheduling

Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed. You estimate the calendar time needed to complete each task, the effort required, and who will work on the tasks that have been identified. You also have to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be. In terms of the planning stages that I discussed in the introduction of this chapter, an initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development planning.

Both plan-based and agile processes need an initial project schedule, although the level of detail may be less in an agile project plan. This initial schedule is used to plan how people will be allocated to projects and to check the progress of the project against its contractual commitments. In traditional development processes, the complete schedule is initially developed and then modified as the project progresses. In agile processes, there has to be an overall schedule that identifies when the major phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

Scheduling in plan-driven projects (Figure 23.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task. Tasks should normally last at least a week, and no longer than 2 months. Finer subdivision means that a disproportionate amount of time must be spent on replanning and updating the project plan. The maximum amount of time for

Activity charts

An activity chart is a project schedule representation that shows which tasks can be carried out in parallel and those that must be executed in sequence, due to their dependencies on earlier activities. If a task is dependent on several other tasks then all of these must finish before it can start. The 'critical path' through the activity chart is the longest sequence of dependent tasks. This defines the project duration.

http://www.SoftwareEngineering-9.com/Web/Planning/activities.html

any task should be around 8 to 10 weeks. If it takes longer than this, the task should be subdivided for project planning and scheduling.

Some of these tasks are carried out in parallel, with different people working on different components of the system. You have to coordinate these parallel tasks and organize the work so that the workforce is used optimally and you don't introduce unnecessary dependencies between the tasks. It is important to avoid a situation where the whole project is delayed because a critical task is unfinished.

If a project is technically advanced, initial estimates will almost certainly be optimistic even when you try to consider all eventualities. In this respect, software scheduling is no different from scheduling any other type of large advanced project. New aircraft, bridges, and even new models of cars are frequently late because of unanticipated problems. Schedules, therefore, must be continually updated as better progress information becomes available. If the project being scheduled is similar to a previous project, previous estimates may be reused. However, projects may use different design methods and implementation languages, so experience from previous projects may not be applicable in the planning of a new project.

As I have already suggested, when you are estimating schedules, you must take into account the possibility that things will go wrong. People working on a project may fall ill or leave, hardware may fail, and essential support software or hardware may be delivered late. If the project is new and technically advanced, parts of it may turn out to be more difficult and take longer than originally anticipated.

A good rule of thumb is to estimate as if nothing will go wrong, then increase your estimate to cover anticipated problems. A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.), and the quality and experience of the software engineers working on the project. Contingency estimates may add 30% to 50% to the effort and time required for the project.

23.3.1 Schedule representation

Project schedules may simply be represented in a table or spreadsheet showing the tasks, effort, expected duration, and task dependencies (Figure 23.5). However, this style of representation makes it difficult to see the relationships and dependencies

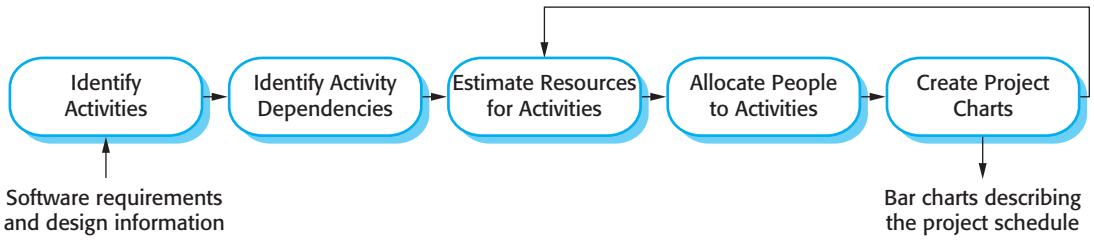


Figure 23.4 The project scheduling process

between the different activities. For this reason, alternative graphical representations of project schedules have been developed that are often easier to read and understand. There are two types of representation that are commonly used:

1. Bar charts, which are calendar-based, show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are sometimes called 'Gantt charts', after their inventor, Henry Gantt.
2. Activity networks, which are network diagrams, show the dependencies between the different activities making up a project.

Normally, a project planning tool is used to manage project schedule information. These tools usually expect you to input project information into a table and will then create a database of project information. Bar charts and activity charts can then be generated automatically from this database.

Project activities are the basic planning element. Each activity has:

1. A duration in calendar days or months.
2. An effort estimate, which reflects the number of person-days or person-months to complete the work.
3. A deadline by which the activity should be completed.
4. A defined endpoint. This represents the tangible result of completing the activity. This could be a document, the holding of a review meeting, the successful execution of all tests, etc.

When planning a project, you should also define milestones; that is, each stage in the project where a progress assessment can be made. Each milestone should be documented by a short report that summarizes the progress made and the work done. Milestones may be associated with a single task or with groups of related activities. For example, in Figure 23.5, milestone M1 is associated with task T1 and milestone M3 is associated with a pair of tasks, T2 and T4.

A special kind of milestone is the production of a project deliverable. A deliverable is a work product that is delivered to the customer. It is the outcome of a significant project phase such as specification or design. Usually, the deliverables that are

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figure 23.5 Tasks, durations, and dependencies

required are specified in the project contract and the customer's view of the project's progress depends on these deliverables.

To illustrate how bar charts are used, I have created a hypothetical set of tasks as shown in Figure 23.5. This table shows tasks, estimated effort, duration, and task interdependencies. From Figure 23.5, you can see that task T3 is dependent on task T1. Task T1 must, therefore, be completed before T3 starts. For example, T1 might be the preparation of a component design and T3, the implementation of that design. Before implementation starts, the design should be complete. Notice that the estimated duration for some tasks is more than the effort required and vice versa. If the effort is less than the duration, this means that the people allocated to that task are not working full-time on it. If the effort exceeds the duration, this means that several team members are working on the task at the same time.

Figure 23.6 takes the information in Figure 23.5 and presents the project schedule in a graphical format. It is a bar chart showing a project calendar and the start and finish dates of tasks. Reading from left to right, the bar chart clearly shows when tasks start and end. The milestones (M1, M2, etc.) are also shown on the bar chart. Notice that tasks that are independent are carried out in parallel (e.g., tasks T1, T2, and T4 all start at the beginning of the project).

As well as planning the delivery schedule for the software, project managers have to allocate resources to tasks. The key resource is, of course, the software engineers who will do the work, and they have to be assigned to project activities. The resource allocation can also be input to project management tools and a bar chart generated, which shows when staff are working on the project (Figure 23.7). People may be

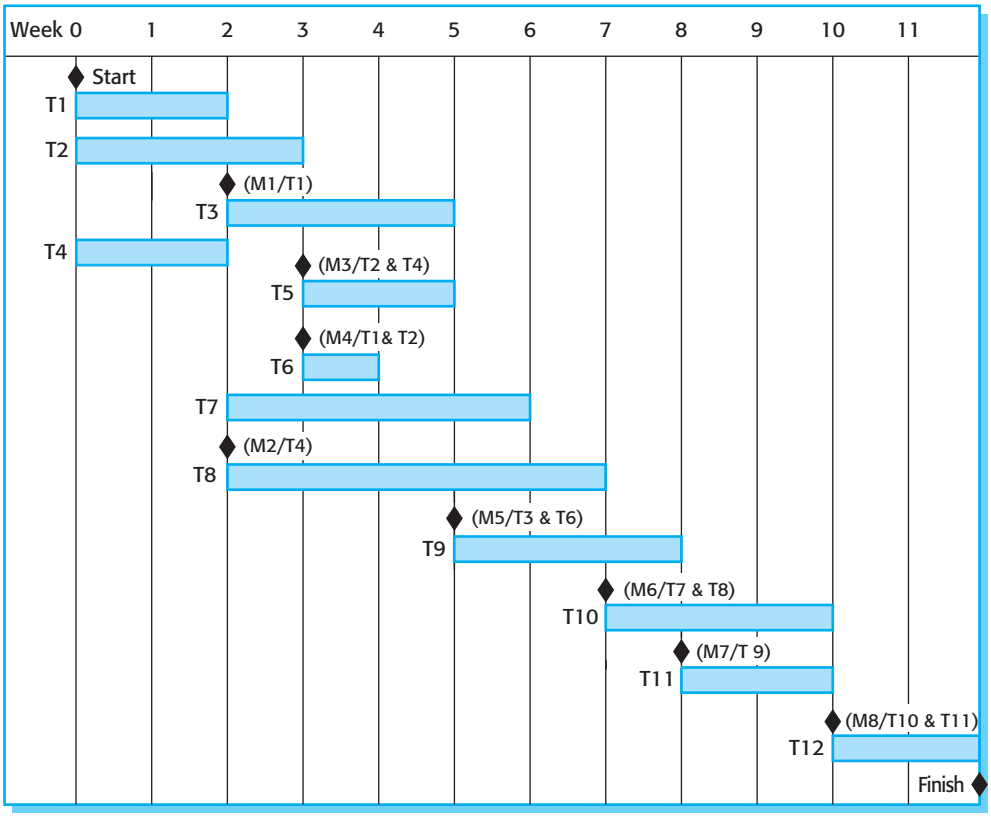


Figure 23.6
Activity bar chart

working on more than one task at the same time and, sometimes, they are not working on the project. They may be on holiday, working on other projects, attending training courses, or engaging in some other activity. I show part-time assignments using a diagonal line crossing the bar.

Large organizations usually employ a number of specialists who work on a project when needed. In Figure 23.7, you can see that Mary is a specialist, who works on only a single task in the project. This can cause scheduling problems. If one project is delayed while a specialist is working on it, this may have a knock-on effect on other projects where the specialist is also required. These may then be delayed because the specialist is not available.

If a task is delayed, this can obviously affect later tasks that are dependent on it. They cannot start until the delayed task is completed. Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time. If a task (T) is delayed, the people allocated may be assigned to other work (W). To complete this may take longer than the delay but, once assigned, they cannot simply be reassigned back to the original task, T. This may then lead to further delays in T as they complete W.

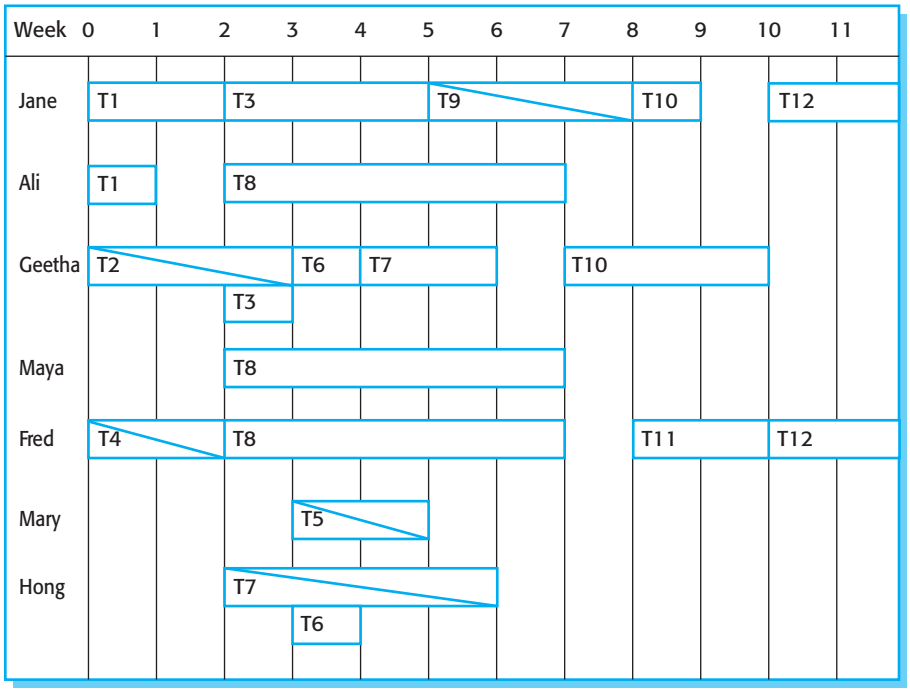


Figure 23.7 Staff allocation chart

23.4 Agile planning

Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments. Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development. The decision on what to include in an increment depends on progress and on the customer's priorities. The argument for this approach is that the customer's priorities and requirements change so it makes sense to have a flexible plan that can accommodate these changes. Cohn's book {Cohn, 2005 ##1735} is a comprehensive discussion of planning issues in agile projects.

The most commonly used agile approaches such as Scrum (Schwaber, 2004) and extreme programming (Beck, 2000) have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:

1. Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
2. Iteration planning, which has a shorter-term outlook, and focuses on planning the next increment of a system. This is typically 2 to 4 weeks of work for the team.

Figure 23.8 Planning in XP

I have already discussed the Scrum approach to planning in Chapter 3, so I concentrate here on planning in extreme programming (XP). This is called the 'planning game' and it usually involves the whole development team, including customer representatives. Figure 23.8 shows the stages in the planning game.

The system specification in XP is based on user stories that reflect the features that should be included in the system. At the start of the project, the team and the customer try to identify a set of stories, which covers all of the functionality that will be included in the final system. Some functionality will inevitably be missing, but this is not important at this stage.

The next stage is an estimation stage. The project team reads and discusses the stories and ranks them in order of the amount of time they think it will take to implement the story. This may involve breaking large stories into smaller stories. Relative estimation is often easier than absolute estimation. People often find it difficult to estimate how much effort or time is needed to do something. However, when they are presented with several things to do, they can make judgments about which stories will take the longest time and most effort. Once the ranking has been completed, the team then allocates notional effort points to the stories. A complex story may have 8 points and a simple story 2 points. You do this for all of the stories in the ranked list.

Once the stories have been estimated, the relative effort is translated into the first estimate of the total effort required by using the notion of 'velocity'. In XP, velocity is the number of effort points implemented by the team, per day. This can be estimated either from previous experience or by developing one or two stories to see how much time is required. The velocity estimate is approximate, but is refined during the development process. Once you have a velocity estimate, you can calculate the total effort in person-days to implement the system.

Release planning involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented. The customer has to be involved in this process. A release date is then chosen and the stories are examined to see if the effort estimate is consistent with that date. If not, stories are added or removed from the list.

Iteration planning is the first stage into the iteration development process. Stories to be implemented for that iteration are chosen, with the number of stories reflecting the time to deliver an iteration (usually 2 or 3 weeks) and the team's velocity. When the iteration delivery date is reached, that iteration is complete, even if all of the stories have not been implemented. The team considers the stories that have been implemented and adds up their effort points. The velocity can then be recalculated and this is used in planning the next release of the system.

At the start of each iteration, there is a more detailed planning stage where the developers break stories down into development tasks. A development task should

take 4–16 hours. All of the tasks that must be completed to implement all of the stories in that iteration are listed. The individual developers then sign up for the specific tasks that they will implement. Each developer knows their individual velocity so should not sign up for more tasks than they can implement in the time.

There are two important benefits from this approach to task allocation:

1. The whole team gets an overview of the tasks to be completed in an iteration. They therefore have an understanding of what other team members are doing and who to talk to if task dependencies are identified.
2. Individual developers choose the tasks to implement; they are not simply allocated tasks by a project manager. They therefore have a sense of ownership in these tasks and this is likely to motivate them to complete the task.

Halfway though an iteration, progress is reviewed. At this stage, half of the story effort points should have been completed. So, if an iteration involves 24 story points and 36 tasks, 12 story points and 18 tasks should have been completed. If this is not the case, then the customer has to be consulted and some stories removed from the iteration.

This approach to planning has the advantage that the software is always released as planned and there is no schedule slippage. If the work cannot be completed in the time allowed, the XP philosophy is to reduce the scope of the work rather than extend the schedule. However, in some cases, the increment may not be enough to be useful. Reducing the scope may create extra work for customers if they have to use an incomplete system or change their work practices between one release of the system and another.

A major difficulty in agile planning is that it is reliant on customer involvement and availability. In practice, this can be difficult to arrange, as the customer representative must sometimes give priority to other work. Customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning project.

Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented. However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management. Consequently, large projects are usually planned using traditional approaches to project management.

23.5 Estimation techniques

Project schedule estimation is difficult. 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. There are so many uncertainties

that it is impossible to estimate system development costs accurately during the early stages of a project.

There is even a fundamental difficulty in assessing the accuracy of different approaches to cost and effort estimation. Project 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 realized. A project that is within budget may have achieved this at the expense of features in the software being developed.

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. Nevertheless, organizations need to make software effort and cost estimates. There are two types of technique that can be used to do this:

1. *Experience-based techniques* The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
2. *Algorithmic cost modeling* In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.

In both cases, you need to use your judgment to estimate either the effort directly, or estimate the project and product characteristics. In the startup phase of a project, these estimates have a wide margin of error. Based on data collected from a large number of projects, Boehm, et al. (1995) discovered that startup estimates vary significantly. If the initial estimate of effort required is x months of effort, they found that the range may be from $0.25x$ to $4x$ of the actual effort as measured when the system was delivered. During development planning, estimates become more and more accurate as the project progresses (Figure 23.9).

Experience-based techniques rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development. Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed. You document these in a spreadsheet, estimate them individually, and compute the total effort required. It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate. This often reveals factors that others have not considered and you then iterate towards an agreed group estimate.

The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects. Software development changes very quickly and a project will often use unfamiliar techniques such as web services, COTS-based development, or AJAX. If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.

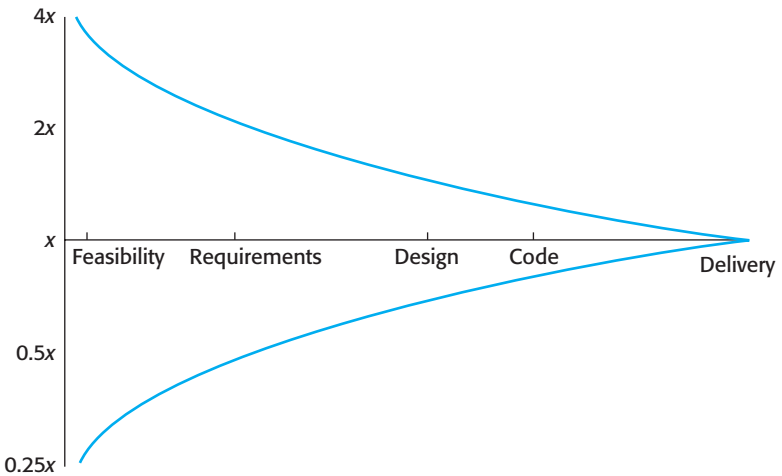


Figure 23.9
Estimate uncertainty

23.5.1 Algorithmic cost modeling

Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size; the type of software being developed; and other team, process, and product factors. An algorithmic cost model can be built by analyzing 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. However, Boehm and his collaborators (2000) discuss a range of other uses for these models, such as the preparation of estimates for investors in software companies; alternative strategies to help assess risks; and to informed decisions about reuse, redevelopment, or outsourcing.

Algorithmic models for estimating effort in a software project are mostly based on a simple formula:

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

A is a constant factor which depends on local organizational 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 application 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.

The number of lines of source code (SLOC) in the delivered system is the fundamental size metric that is used in many algorithmic cost models. Size estimation may involve estimation by analogy with other projects, estimation by converting function or application 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 judgment.

Most algorithmic estimation models have an exponential component (B in the above equation) that is related to the size and complexity of the system. This reflects the fact that costs do not usually increase linearly with project size. As the size and complexity 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. The more complex the system, the more these factors affect the cost. Therefore, the value of B usually increases with the size and complexity of the system.

All algorithmic models have similar problems:

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

Accurate code size estimation is difficult at an early stage in a project because the size of the final program depends on design decisions that may not have been made when the estimate is required. For example, an application that requires high-performance data management may either implement its own data management system or use a commercial database system. In the initial cost estimation, you are unlikely to know if there is a commercial database system that performs well enough to meet the performance requirements. You therefore don't know how much data management code will be included in the system.

The programming language used for system development also affects the number of lines of code to be developed. A language like Java might mean that more lines of code are necessary than if C (say) was 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.

Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use. There are many attributes and considerable scope for uncertainty in estimating their values. This complexity discourages potential users and hence the practical application of algorithmic cost modeling has been limited to a small number of companies.

Another barrier that discourages the use of algorithmic models is the need for calibration. Model users should calibrate their model and the attribute values using their own historical project data, as this reflects local practice and experience. However, very few organizations have collected enough data from past projects in a form that supports model calibration. Practical use of algorithmic models, therefore, has to start with the published values for the model parameters. It is practically impossible for a modeler to know how closely these relate to their own organization.

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

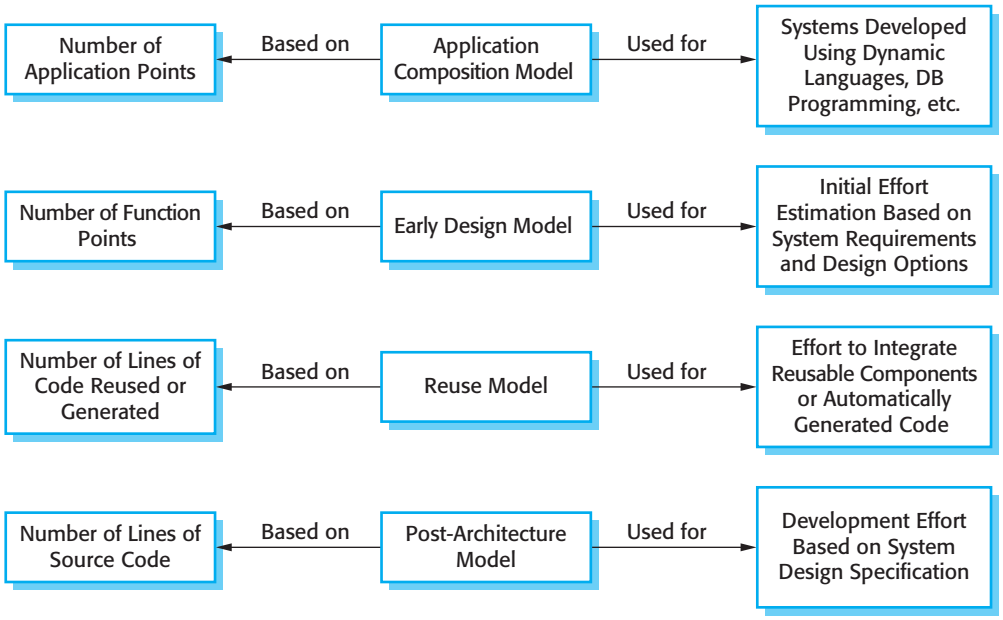


Figure 23.10 COCOMO estimation models

formula to all of them. Estimates are most likely to be accurate when you understand the type of software that is being developed, have calibrated the costing model using local data, or when programming language and hardware choices are predefined.

23.5.2 The COCOMO II model

Several similar models have been proposed to help estimate the effort, schedule, and costs of a software project. The model that I discuss here is the COCOMO II model. This is an empirical model that was derived by collecting data from a large number of software projects. These data were analyzed to discover the formulae that were the best fit to the observations. These formulae linked the size of the system and product, project and team factors to the effort to develop the system. COCOMO II is a well-documented and nonproprietary estimation model.

COCOMO II was developed from earlier COCOMO cost estimation models, which were largely based on original code development (Boehm, 1981; Boehm and Royce, 1989). The COCOMO II model takes into account more modern approaches to software development, such as rapid development using dynamic languages, development by component composition, and use of database programming. COCOMO II supports the spiral model of development, described in Chapter 2, and embeds submodels that produce increasingly detailed estimates.

The submodels (Figure 23.10) that are part of the COCOMO II model are:

1. *An application-composition model* This models the effort required to develop systems that are created from reusable components, scripting, or

Software productivity

Software productivity is an estimate of the average amount of development work that software engineers complete in a week or a month. It is therefore expressed as lines of code/month, function points/month, etc.

However, whilst productivity can be easily measured where there is a tangible outcome (e.g., a clerk processes *N* invoices/day), software productivity is more difficult to define. Different people may implement the same functionality in different ways, using different numbers of lines of code. The quality of the code is also important but is, to some extent, subjective. Productivity comparisons between software engineers are, therefore, unreliable and so are not very useful for project planning.

http://www.SoftwareEngineering-9.com/Web/Planning/productivity.html

database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required. The number of application points in a program is a weighted estimate of the number of separate screens that are displayed, the number of reports that are produced, the number of modules in imperative programming languages (such as Java), and the number of lines of scripting language or database programming code.

2. *An early design model* This model is used during early stages of the system design after the requirements have been established. The estimate is based on the standard estimation formula that I discussed in the introduction, with a simplified set of seven multipliers. Estimates are based on function points, which are then converted to number of lines of source code. Function points are a language-independent way of quantifying program functionality. You compute the total number of function points in a program by measuring or estimating the number of external inputs and outputs, user interactions, external interfaces, and files or database tables used by the system.
3. *A reuse model* This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally 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 the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability, product, and project characteristics.

Of course, in large systems, different parts of the system 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

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

Figure 23.11 Application-point productivity

submodel 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 (sometimes called object points), divided by a standard estimate of application point productivity. The estimate is then adjusted according to the difficulty of developing each application point (Boehm, et al., 2000). Productivity depends on the developer's experience and capability as well as the capabilities of the software tools (ICASE) used to support development. Figure 23.11 shows the levels of application-point productivity suggested by the COCOMO developers (Boehm, et al., 1995).

Application composition usually involves significant software reuse. It is almost certain that some of the application points in the system will be implemented using reusable components. Consequently, you have to adjust the estimate to take into account the percentage of reuse expected. Therefore, the final formula for effort computation for system prototypes is:

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

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 application-point productivity, as shown in Figure 23.11. The model produces an approximate estimate as it does not take into account the additional effort involved in reuse.

The early design model

This model may be used during the early stages of a project, before a detailed architectural design for the system is available. Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements. The early design model assumes that user requirements have

been agreed and initial stages of the system design process are under way. Your goal at this stage should be to make a quick and approximate cost estimate. Therefore, you have to make simplifying assumptions, for example, that the effort involved in integrating reusable code is zero.

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 proposed 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 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 26) of the organization. I discuss how the value of this exponent is calculated using these parameters in the description of the COCOMO II post-architecture model.

This results in an effort computation as follows:

$$\text{PM} = 2.94 \times \text{Size}^{(1.1\ -\ 1.24)} \times M$$

where

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

The multiplier **M** is based on seven project and process attributes that increase or decrease the estimate. The attributes 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**). I explain these attributes on the book's webpages. You estimate values for these attributes using a six-point scale, where 1 corresponds to 'very low' and 6 corresponds to 'very high'.

The reuse model

As I have discussed in Chapter 16, software reuse is now common. Most large systems include a significant amount of code that has been reused from previous development projects. The reuse model is used to estimate the effort required to integrate reusable or generated code.

COCOMO II considers two types of reused code. '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. 'White box' code has to be adapted to integrate it with new code or other reused components. Development

effort is required for reuse because the code has to be understood and modified before it can work correctly in the system.

Many systems include automatically generated code from system models, as discussed in Chapter 5. A model (often in UML) is analyzed and code is generated to implement the objects specified in the model. The COCOMO II reuse model includes a formula to estimate the effort required to integrate this generated code:

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

ASLOC is the total number of lines of reused code, including code that is automatically generated.

AT is the percentage of reused code that is automatically generated.

ATPROD is the productivity of engineers in integrating such code.

Boehm, et al. (2000) have measured **ATPROD** to be about 2,400 source statements per month. Therefore, if there are a total of 20,000 lines of reused source code in a system and 30% of this is automatically generated, then the effort required to integrate the generated code is:

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

A separate effort computation is used to estimate the effort required to integrate the reused code from other systems. The reuse model does not compute the effort directly from an estimate of the number of reused components. Rather, based on the number of lines of code that are reused, the model provides a basis for calculating the equivalent number of lines of new code (**ESLOC**). This is based on the number of lines of reusable code that have to be changed and a multiplier that reflects the amount of work you need to do to reuse the components. The formula to compute **ESLOC** takes into account the effort required for software understanding, making changes to the reused code, and making changes to the system to integrate that code.

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

$$ESLOC = ASLOC \times AAM$$

ESLOC is the equivalent number of lines of new source code.

ASLOC is the number of lines of code in the components that have to be changed.

AAM is an Adaptation Adjustment Multiplier, as discussed below.

Reuse is never free and some costs are incurred even if no reuse proves to be possible. However, reuse costs decrease as the amount of code reused increases. The fixed understanding and assessment costs are spread across more lines of code. The Adaptation Adjustment Multiplier (**AAM**) adjusts the estimate to reflect

COCOMO II cost drivers

COCOMO II cost drivers are attributes that reflect some of the product, team, process, and organizational factors that affect the amount of effort needed to develop a software system. For example, if a high level of reliability is required, extra effort will be needed; if there is a need for rapid delivery, extra effort will be required; if the team members change, extra effort will be required.

There are 17 of these attributes in the COCOMO II model, which have been assigned values by the model developers.

http://www.SoftwareEngineering-9.com/Web/Planning/costdrivers.html

the additional 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. The adaptation component includes subcomponents 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 or not 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.

If some code adaptation can be done automatically, this reduces the effort required. You therefore adjust the estimate by estimating the percentage of automatically adapted code (AT) and using this to adjust ASLOC. Therefore, the final formula is:

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

Once ESLOC has been calculated, you then apply the standard estimation formula to calculate the total effort required, where the Size parameter = ESLOC. You then add this to the effort to integrate automatically generated code that you have already computed, thus computing the total effort required.

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

subsystem structure is known. You can then make estimates for each part of the system.

The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

$$\mathsf{PM} = \mathsf{A} \times \mathsf{Size}^{\mathsf{B}} \times \mathsf{M}$$

By this stage in the process, you should be able to make a more accurate estimate of the project size as you know how the system will be decomposed into objects or modules. You make this estimate of the code size using three parameters:

1. An estimate of the total number of lines of new code to be developed (**SLOC**).
2. An estimate of the reuse costs based on an equivalent number of source lines of code (**ESLOC**), calculated using the reuse model.
3. An estimate of the number of lines of code that are likely to be modified because of changes to the system requirements.

You add the values of these parameters to compute 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. This leads to rework and development of extra code, which you have to take into account. Of course there will often be even more uncertainty in this figure than in the estimates of new code to be developed.

The exponent term (**B**) in the effort computation formula is related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. However, good organizational practices and procedures can control the diseconomy of scale that is a consequence of increasing complexity. The value of the exponent **B** is therefore based on five factors, as shown in Figure 23.12. These factors are rated on a six-point scale from 0 to 5, where 0 means 'extra high' and 5 means 'very low'. To calculate **B**, you add the ratings, divide them by 100, and add the result to 1.01 to get the exponent that should be used.

For example, imagine that an organization is taking on a project in a domain in which it has little previous experience. The project client has not defined the process to be used or allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organization has recently put in place a process improvement program and has been rated as a Level 2 organization according to the SEI capability assessment, as discussed in Chapter 26. Possible values for the ratings used in exponent calculation are therefore:

1. *Precedentedness*, rated low (4). This is a new project for the organization.
2. *Development flexibility*, rated very high (1). No client involvement in the development process so there are few externally imposed changes.

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization 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 knows 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 organization. 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.

Figure 23.12 Scale factors used in the exponent computation in the post-architecture model

3. *Architecture/risk resolution*, rated very low (5). There has been no risk analysis carried out.
4. *Team cohesion*, rated nominal (3). This is a new team so there is no information available on cohesion.
5. *Process maturity*, rated nominal (3). Some process control is in place.

The sum of these values is 16. You then calculate the exponent by dividing this by 100 and adding the result to 0.01. The adjusted value of B is therefore 1.17.

The overall effort estimate is refined using an extensive set of 17 product, process, and organizational attributes (cost drivers), rather than the seven attributes used in the early design model. You can estimate values for these attributes because you have more information about the software itself, its non-functional requirements, the development team, and the development process.

Figure 23.13 shows how the cost driver attributes can influence effort estimates. I have taken a value for the exponent of 1.17 as discussed in the previous 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 23.13, 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, 2000). 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 significant differences between different types of projects and the difficulties of transferring experience from one application domain to another.

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	**2,306 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**

Figure 23.13 The effect of cost drivers on effort estimates

23.5.3 Project duration and staffing

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

The COCOMO model includes a formula to estimate the calendar time required to complete a project:

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

TDEV is the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.

PM is the effort computed by the COCOMO model.

B is the complexity-related exponent, as discussed in Section 23.5.2.

If B = 1.17 and PM = 60 then

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

However, the nominal project schedule predicted by the COCOMO model and the schedule required by the project plan are not necessarily the same thing. There may be a requirement to deliver the software earlier or (more rarely) later than the date suggested by the nominal schedule. If the schedule is to be compressed, this increases the effort required for the project. This is taken into account by the SCED multiplier in the effort estimation computation.

Assume that a project estimated TDEV as 13 months, as suggested above, but the actual schedule required was 11 months. This represents a schedule compression of approximately 25%. Using the values for the SCED multiplier as derived by Boehm's team, the effort multiplier for such a schedule compression is 1.43. Therefore, the actual effort that will be required if this accelerated schedule is to be met is almost 50% more than the effort required to deliver the software according to the nominal schedule.

There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project, and the project delivery schedule. If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you can complete the work in 11 months (55 person-months of effort). However, the COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).

The reason for this is that adding people actually reduces the productivity of existing team members and so the actual increment of effort added is less than one person. As the project team increases in size, team members 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. If the development team is large, it is sometimes the case that adding more people to a project increases rather than reduces the development schedule. 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 to complete the work.

You cannot simply estimate the number of people required for a project team by dividing the total effort by the required project schedule. Usually, 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 then declines in size 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.

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

KEY POINTS

- The price charged for a system does not just depend on its estimated development costs and the profit required by the development company. Organizational factors may mean that the price is increased to compensate for increased risk or decreased to gain competitive advantage.
- Software is often priced to gain a contract and the functionality of the system is then adjusted to meet the estimated price.
- Plan-driven development is organized around a complete project plan that defines the project activities, the planned effort, the activity schedule, and who is responsible for each activity.
- Project scheduling involves the creation of various graphical representations of part of the project plan. Bar charts, which show the activity duration and staffing timelines, are the most commonly used schedule representations.
- A project milestone is a predictable outcome of an activity or set of activities. At each milestone, a formal report of progress should be presented to management. A deliverable is a work product that is delivered to the project customer.
- The XP planning game involves the whole team in project planning. The plan is developed incrementally and, if problems arise, it is adjusted so that software functionality is reduced instead of delaying the delivery of an increment.
- Estimation techniques for software may be experience-based, where managers judge the effort required, or algorithmic, where the effort required is computed from other estimated project parameters.
- The COCOMO II costing model is a mature algorithmic cost model that takes project, product, hardware, and personnel attributes into account when formulating a cost estimate.

FURTHER READING

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. (B. Boehm et al., Prentice Hall, 2000.)

'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.)

Agile Estimating and Planning. This book is a comprehensive description of story-based planning as used in XP, as well as a rationale for using an agile approach to project planning. However, it also includes a good, general introduction to project planning issues. (M. Cohn, Prentice Hall, 2005.)

'Achievements and Challenges in Cocomo-based Software Resource Estimation'. This article presents a history of the COCOMO models and influences on these models, and discusses the variants of these models that have been developed. It also identifies further possible developments in the COCOMO approach. (B. W. Boehm and R. Valeridi, *IEEE Software*, **25** (5), September/October 2008.) http://dx.doi.org/10.1109/MS.2008.133.

EXERCISES

23.1. Under what circumstances might a company justifiably charge a much higher price for a software system than the software cost estimate plus a reasonable profit margin?

23.2. Explain why the process of project planning is iterative and why a plan must be continually reviewed during a software project.

23.3. Briefly explain the purpose of each of the sections in a software project plan.

23.4. 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.

23.5. Figure 23.14 sets out a number of tasks, their durations, and their dependencies. Draw a bar chart showing the project schedule.

23.6. Figure 23.14 shows the task durations for software project activities. Assume that a serious, unanticipated setback occurs and instead of taking 10 days, task T5 takes 40 days. Draw up new bar charts showing how the project might be reorganized.

23.7. The XP planning game is based around the notion of planning to implement the stories that represent the system requirements. Explain the potential problems with this approach when software has high performance or dependability requirements.

23.8. A software manager is in charge of the development of a safety-critical software system, which is designed to control a radiotherapy machine to treat patients suffering from cancer. This system is embedded in the machine and must run on a special-purpose processor with a fixed amount of memory (256 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 organizational 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.

Task	Duration (days)	Dependencies
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T2, T14
T16	10	T15

Figure 23.14
Scheduling example

23.9. Some very large software projects involve writing millions of lines of code. Explain why the effort estimation models, such as COCOMO, might not work well when applied to very large systems.

23.10. 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?

REFERENCES

Beck, K. (2000). *extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. 2000. 'COCOMO II Model Definition Manual'. Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. and Selby, R. (1995). 'Cost models for future life cycle processes: COCOMO 2'. *Annals of Software Engineering,* **1** 57–94.

Boehm, B. and Royce, W. (1989). 'Ada COCOMO and the Ada Process Model'. *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh: Software Engineering Institute.

Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.

Londeix, B. (1987). *Cost Estimation for Software Development*. Wokingham: Addison-Wesley.

Myers, W. (1989). 'Allow Plenty of Time for Large-Scale Software'. *IEEE Software,* **6** (4), 92–9.

Putnam, L. H. (1978). 'A General Empirical Solution to the Macro Software Sizing and Estimating Problem'. *IEEE Trans. on Software Engineering.*, **SE-4** (3), 345–61.

Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.