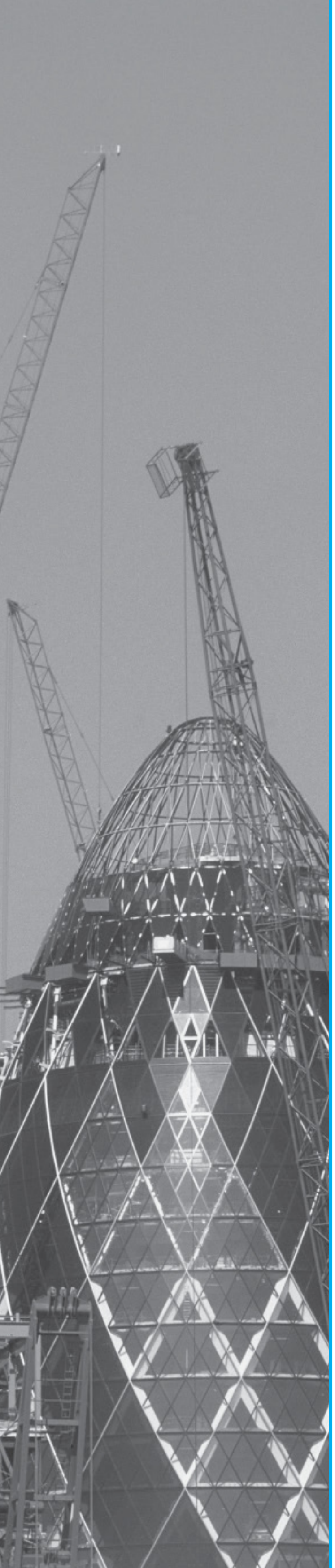

23

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 the factors that affect the price of a software system to be developed for external clients;
- 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 agile project planning based on the "planning game";
- understand cost estimation techniques and how the COCOMO II model can be used for software 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 them 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 and updated as the project progresses, is used to show how the work will be done and to 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, and so on. Here, you have more information than at the proposal stage, and you can therefore refine the initial effort estimates that you have prepared.

3. Periodically throughout the project, when you update your plan to reflect new information about the software and its development. You learn more about the system being implemented and the capabilities of your development team. As software requirements change, the work breakdown has to be altered and the schedule extended. This information allows you to make more accurate estimates of how long the work will take.

Planning at the proposal stage is inevitably speculative, as you do not have a complete set of requirements for the software to be developed. 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 have to re-plan the project, taking into account changes since the proposal was made and new information about the system, the development process, and the development team.

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

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://software-engineering-book.com/web/overhead-costs/

You should use three main parameters when computing the costs of a software development project:

- effort costs (the costs of paying software engineers and managers);
- hardware and software costs, including hardware maintenance and software support; and
- 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. You therefore make the best possible estimate and then add 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. While 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. You can use electronic meeting systems and other collaborative software to reduce travel and so have more time available for productive 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. Your aim should be to create a project plan with enough detail to help make decisions about project staffing and budgeting. You use this 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 companies 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 that the team is having problems with a software fault in the communications systems. The project manager can then immediately assign a communications expert to the problem to help find and solve the problem.

The project plan always evolves during the development process because of requirements changes, technology issues, and development problems. 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 because 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 you only need to include essential 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. Some aspects of agile planning have already been covered in Chapter 3, and I discuss other approaches in Section 23.4.

23.1 Software pricing

In principle, the price of a software system developed for 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 take broader organizational, economic, political, and business considerations into account (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 issues may cause the price to be adjusted upward or downward.

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. 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.

Factor	Description
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 might then be reduced to reflect the value of the source code to the developer.
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.
Financial health	Companies with financial problems 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.
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.
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.

Figure 23.1 Factors affecting software pricing

This is an example of an approach to software pricing called "pricing to win." 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 project costs remain within budget.

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 being granted the contract, OilSoft then negotiates the detailed requirements of the system so that basic functionality is delivered. It then estimates the additional costs for other requirements.

This approach has advantages for both the software developer and the customer. The requirements are negotiated to avoid requirements that are difficult to implement and potentially very expensive. Flexible requirements make it easier to reuse software. The oil company has awarded the contract to a known company that it can trust. Furthermore, it may be possible to spread the cost of the project over several versions of the system. This may reduce the costs of system deployment and allow the client to budget for the project cost over several financial years.

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. Agile development involves a different planning process, discussed in Section 23.4, where decisions are delayed.

The problem with plan-driven development is that early decisions have to be revised because of changes to the environments in which the software is developed and used. Delaying planning decisions avoids unnecessary rework. However, the arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be taken into account. Potential problems and dependencies are discovered before the project starts, rather than once the project is underway.

In my view, the best approach to project planning involves a sensible 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 systems 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 the approach that is taken to risk management as well as risks to the project and the software under development. The details of project plans vary depending on the type of project and organization but plans normally include the following sections:

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

2. *Project organization* Describes the way in which the development team is organized, the people involved, and their roles in the team.

Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
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.

Figure 23.2 Project plan supplements

3. *Risk analysis* Describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies (discussed in Chapter 22) that are proposed.
4. *Hardware and software resource requirements* Specifies the hardware and support software required to carry out the development. If hardware has to be purchased, estimates of the prices and the delivery schedule may be included.
5. *Work breakdown* Sets out the breakdown of the project into activities and identifies the inputs to and the outputs from each project activity.
6. *Project schedule* 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* Defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

The main project plan should always include a project risk assessment and a schedule for the project. In addition, you may develop a number of supplementary plans for activities such as testing and configuration management. Figure 23.2 shows some supplementary plans that may be developed. These are all usually needed in large projects developing large, complex systems.

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 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

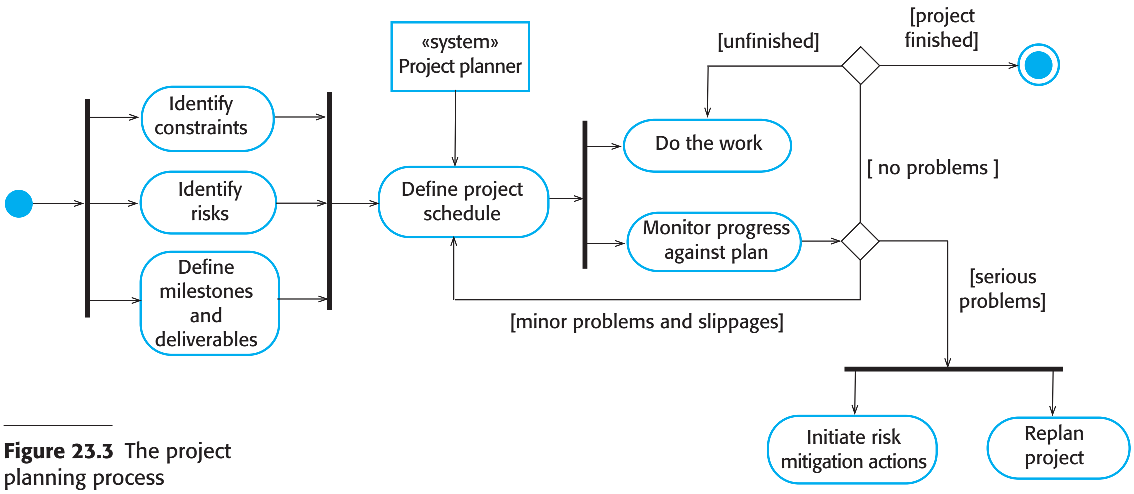

Figure 23.3 The project planning process

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 re-planned.

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 assessment, 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, for example, a requirements document for the system.

The process then enters a loop that terminates when the project is complete. You draw up an estimated schedule for the project, and the activities defined in the schedule are initiated or are approved 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.

You should make realistic rather than optimistic assumptions when you are defining a project plan. Problems of some description always arise during a project, and these lead to project delays. Your initial assumptions and scheduling should therefore be pessimistic and take unexpected problems into account. You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.

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 re-plan 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 to 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. Reviews should also check that the customer's goals are unchanged and that the project remains aligned with these goals.

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 and the effort required, and you suggest who will work on the tasks that have been identified. You also have to estimate the hardware and software resources that are needed to complete each task. For example, if you are developing an embedded system, you have to estimate the time that you need on specialized hardware and the costs of running a system simulator. In terms of the planning stages that I introduced 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 less detail is included 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 re-planning and updating the project plan. The maximum amount of time for any task should be 6 to 8 weeks. If a task will take longer than this, it should be split into subtasks 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

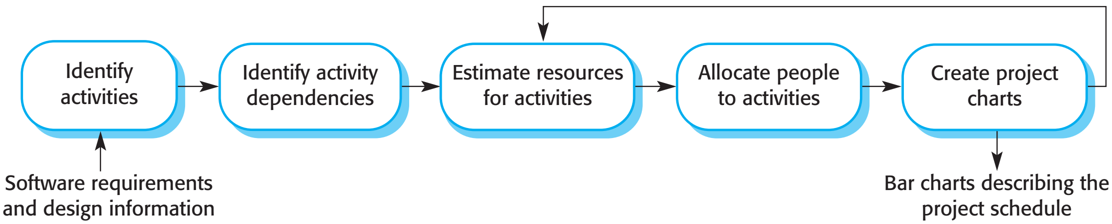

Figure 23.4 The project scheduling process

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.

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 and 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 presentation

Project schedules may simply be documented in a table or spreadsheet showing the tasks, estimated effort, duration, and task dependencies (Figure 23.5). However, this style of presentation makes it difficult to see the relationships and dependencies between the different activities. For this reason, alternative graphical visualizations of project schedules have been developed that are often easier to read and understand. Two types of visualization are commonly used:

1. Calendar-based bar charts show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are also called Gantt charts, after their inventor, Henry Gantt.

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

2. Activity networks show the dependencies between the different activities making up a project. These networks are described in an associated web section.

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

- a duration in calendar days or months;
- an effort estimate, which shows the number of person-days or person-months to complete the work;
- a deadline by which the activity should be complete; and
- a defined endpoint, which might be a document, the holding of a review meeting, the successful execution of all tests, or the like.

When planning a project, you may decide to define project milestones. A milestone is a logical end to a stage of the project where the progress of the work can be reviewed. Each milestone should be documented by a brief report (often simply an email) that summarizes the work done and whether or not the work has been completed as planned. 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 marks the end of that activity. Milestone M3 is associated with a pair of tasks T2 and T4; there is no individual milestone at the end of these tasks.

Activity charts

An activity chart is a project schedule representation that presents the project plan as a directed graph. It 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 tasks must be completed before it can start. The "critical path" through the activity chart is the longest sequence of dependent tasks. This defines the project duration.

http://software-engineering-book.com/web/planning-activities/

Some activities create project deliverables—outputs that are delivered to the software customer. Usually, the deliverables that are required are specified in the project contract, and the customer's view of the project's progress depends on these deliverables. Milestones and deliverables are not the same thing. Milestones are short reports that are used for progress reporting, whereas deliverables are more substantial project outputs such as a requirements document or the initial implementation of a system.

Figure 23.5 shows a hypothetical set of tasks, their estimated effort and duration, and task dependencies. From this table, you can see that task T3 is dependent on task T1. This means that task T1 has to be completed before T3 starts. For example, T1 might be the selection of a system for reuse and T3, the configuration of the selected system. You can't start system configuration until you have chosen and installed the application system to be modified.

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, 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 as 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 may be carried out in parallel. For example, 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. They have to be assigned to project activities. The resource allocation can be analyzed by project management tools, and a bar chart can be generated showing when staff are working on the project (Figure 23.7). People may be 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, or attending training courses. 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

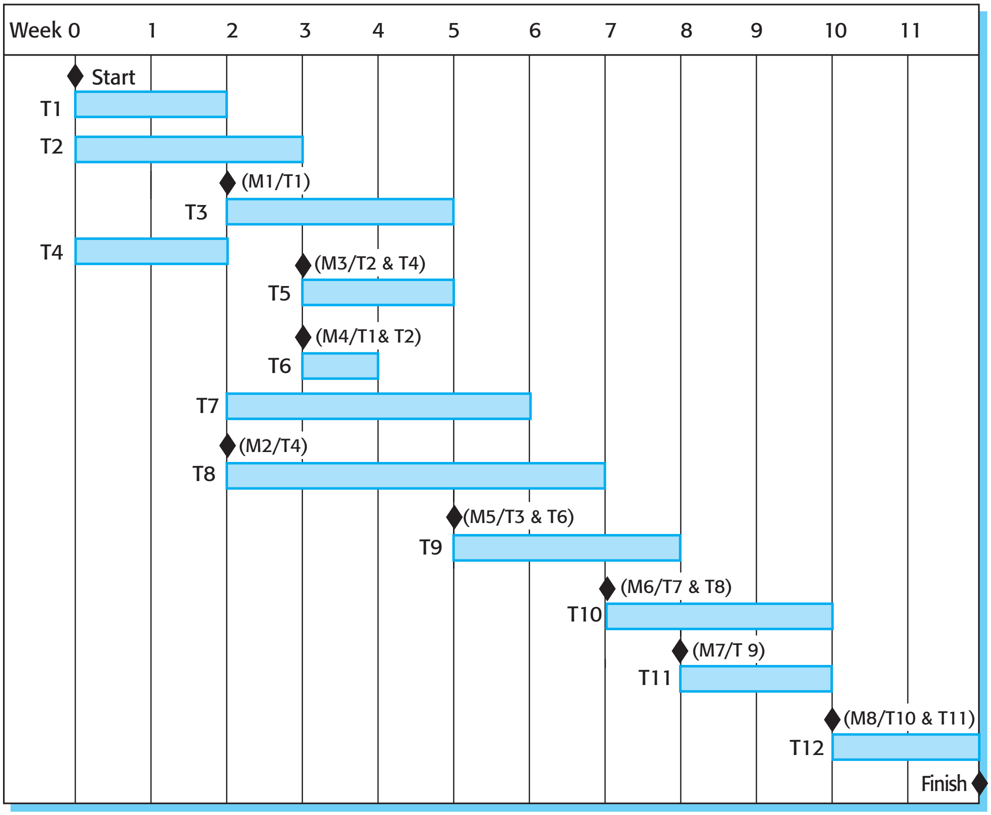

Figure 23.6 Activity bar chart

only a single task (T5) in the project. The use of specialists is unavoidable when complex systems are being developed, but it can lead to scheduling problems. If one project is delayed while a specialist is working on it, this may affect other projects where the specialist is also required. These projects may be delayed because the specialist is not available.

If a task is delayed, later tasks that are dependent on it may be affected. 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 to it may be assigned to other work (W). To complete this work may take longer than the delay, but, once assigned, they cannot simply be reassigned back to the original task. This may then lead to further delays in T as they complete W.

Normally, you should use a project planning tool, such as the Basecamp or Microsoft project, to create, update, and analyze project schedule information. Project management tools usually expect you to input project information into a table, and they create a database of project information. Bar charts and activity charts can then be generated automatically from this database.

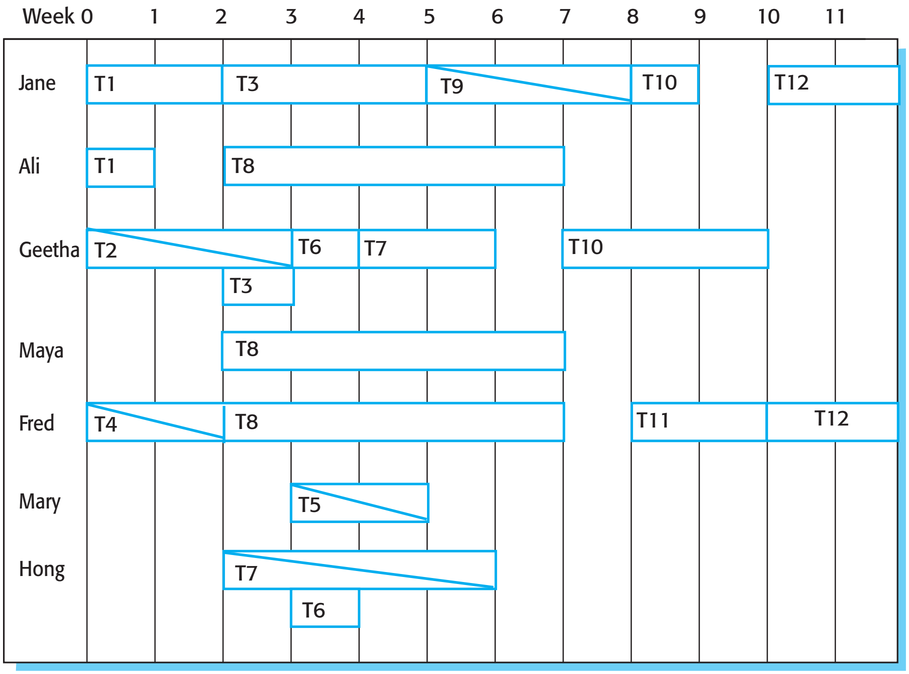

Figure 23.7 Staff allocation chart

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) is an excellent introduction to agile planning.

Agile development methods such as Scrum (Rubin 2013) and Extreme Programming (Beck and Andres 2004) 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 usually represents 2 to 4 weeks of work for the team.

I have already explained the Scrum approach to planning in Chapter 3, which is based on project backlogs and daily reviews of work to be done. It is primarily geared

Figure 23.8 The "planning game"

to iteration planning. Another approach to agile planning, which was developed as part of Extreme Programming, is based on user stories. The so-called planning game can be used in both release planning and iteration planning.

The basis of the planning game (Figure 23.8) is a set of user stories (see Chapter 3) that cover all of the functionality to be included in the final system. The development team and the software customer work together to develop these stories. The team members read and discuss the stories and rank them based on the amount of time they think it will take to implement the story. Some stories may be too large to implement in a single iteration, and these are broken down into smaller stories.

The problem with ranking stories is that people often find it difficult to estimate how much effort or time is needed to do something. To make this easier, relative ranking may be used. The team compares stories in pairs and decides which will take the most time and effort, without assessing exactly how much effort will be required. At the end of this process, the list of stories has been ordered, with the stories at the top of the list taking the most effort to implement. The team then allocates notional effort points to all of the stories in the list. A complex story may have 8 points and a simple story 2 points.

Once the stories have been estimated, the relative effort is translated into the first estimate of the total effort required by using the idea of "velocity." 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 in developing a deliverable system increment. Stories to be implemented during that iteration are chosen, with the number of stories reflecting the time to deliver an workable system (usually 2 or 3 weeks) and the team's velocity. When the delivery date is reached, the development 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 measure is used in planning the next version of the system.

At the start of each development iteration, there is a task planning stage where the developers break down stories 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 and so should not sign up for more tasks than they can implement in the time allotted.

This approach to task allocation has two important benefits:

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 through 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 there has to be discussions with the customer about which stories should be removed from the system increment that is being developed.

This approach to planning has the advantage that a software increment is always delivered at the end of each project iteration. If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced. The delivery schedule is never extended. However, this can cause problems as it means that customer plans may be affected. Reducing the scope may create extra work for customers if they have to use an incomplete system or change the way they work between one release of the system and another.

A major difficulty in agile planning is that it relies on customer involvement and availability. This involvement can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game. Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

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

Estimating project schedules is difficult. You have to make initial estimates on the basis of an incomplete user requirements definition. The software may have to run on unfamiliar platforms 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

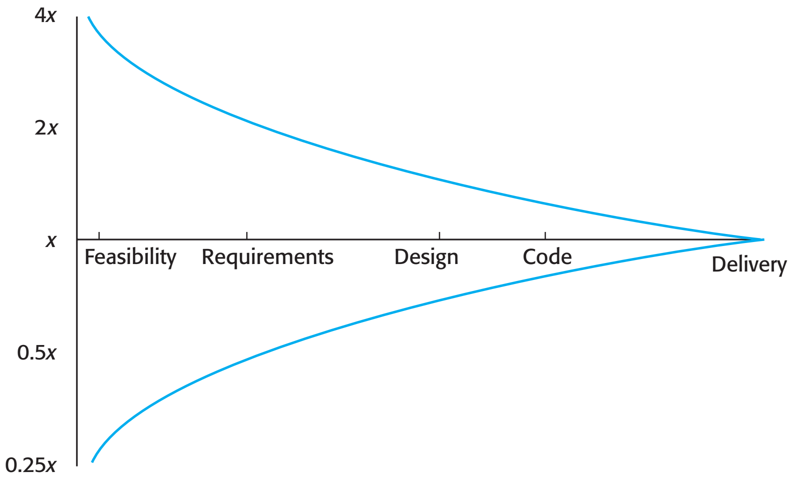

Figure 23.9 Estimate uncertainty

stages of a project. Nevertheless, organizations need to make software effort and cost estimates. Two types of techniques can be used for making estimates:

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, process characteristics, and experience of staff involved.

In both cases, you need to use your judgment to estimate either the effort directly or 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. (B. 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 toward 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, application system configuration, or HTML5. 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.

It is impossible to say whether experience-based or algorithmic approaches are more accurate. 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.

To make a comparison of the accuracy of these techniques, a number of controlled experiments would be required where several techniques were used independently to estimate the project effort and costs. No changes to the project would be allowed, and the final effort could them be compared. The project manager would not know the effort estimates, so no bias would be introduced. However, this scenario is completely impossible in real projects, so we will never have an objective comparison of these approaches.

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. Algorithmic cost models are developed by analyzing the costs and attributes of completed projects, then finding the closest-fit formula to the actual costs incurred.

Algorithmic cost models are primarily used to make estimates of software development costs. However, Boehm and his collaborators (B. W. Boehm et al. 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 inform decisions about reuse, redevelopment, or outsourcing.

Most algorithmic models for estimating effort in a software project are based on a simple formula:

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

A: a constant factor, which depends on local organizational practices and the type of software that is developed.

Size: an assessment of the code size of the software or a functionality estimate expressed in function or application points.

B: represents the complexity of the software and usually lies between 1 and 1.5.

M: is a factor that takes into account process, product and development attributes, such as the dependability requirements for the software and the experience of the development team. These attributes may increase or decrease the overall difficulty of developing the system.

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. To estimate the number of lines of code in a system, you may use a combination of approaches:

1. Compare the system to be developed with similar systems and use their code size as the basis for your estimate.

2. Estimate the number of function or application points in the system (see the following section) and formulaically convert these to lines of code in the programming language used.

3. Rank the system components using judgment of their relative sizes and use a known reference component to translate this ranking to code sizes.

Most algorithmic estimation models have an exponential component (B in the above equation) that increases with 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 increase, 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.

The idea of using a scientific and objective approach to cost estimation is an attractive one, but all algorithmic cost models suffer from two key problems:

1. It is practically impossible to estimate Size accurately 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 also usually inaccurate.

2. The estimates of the complexity and process 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. It is not clear how this should be taken into account in the estimation process. Code reuse also

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, and so forth.

However, while productivity can be easily measured where there is a tangible outcome (e.g., an administrator processes *N* travel claims/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. Therefore, you can't really compare the productivity of individual engineers. It only makes sense to use productivity measures with large groups.

http://software-engineering-book.com/web/productivity/

makes a difference, and some models explicitly estimate the number of lines of code reused. However, if application systems or external services are reused, it is very difficult to compute the number of lines of source code that these replace.

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 means that the practical application of algorithmic cost modeling has been limited to a relatively small number of large companies, mostly working in defense and aerospace systems engineering.

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

23.6 COCOMO cost modeling

The best known algorithmic cost modeling technique and tool is the COCOMO II model. This empirical model was derived by collecting data from a large number of software projects of different sizes. These data were analyzed to discover the formulas that were the best fit to the observations. These formulas linked the size of the

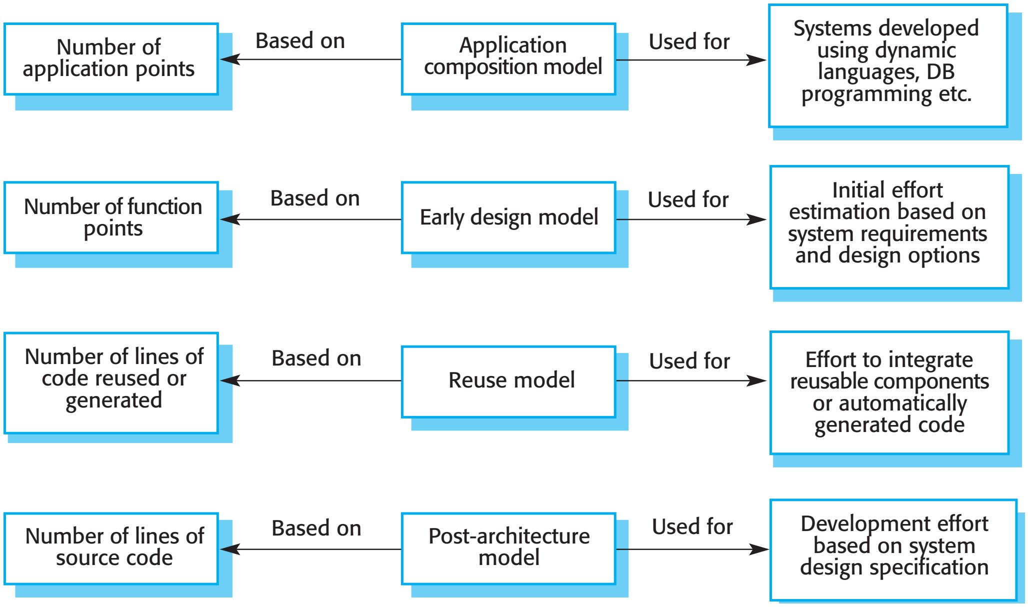

Figure 23.10 COCOMO estimation models

system and product, project, and team factors to the effort to develop the system. COCOMO II is a freely available model that is supported with open-source tools.

COCOMO II was developed from earlier COCOMO (Constructive Cost Modeling) cost estimation models, which were largely based on original code development (B. W. Boehm 1981; B. Boehm and Royce 1989). The COCOMO II model takes into account modern approaches to software development, such as rapid development using dynamic languages, development with reuse, and database programming. COCOMO II embeds several submodels based on these techniques, which 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 database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.

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 of this chapter, 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 submodel for each part of the system and combine the results to create a composite estimate.

The COCOMO II model is a very complex model and, to make it easier to explain, I have simplified its presentation. You could use the models as I have explained them here for simple cost estimation. However, to use COCOMO properly, you should refer to Boehm's book and the manual for the COCOMO II model (B. W. Boehm et al. 2000; Abts et al. 2000).

23.6.1 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 (B. W. Boehm et al. 2000). The number of application points in a program is derived from four simpler estimates:

- the number of separate screens or web pages 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.

This estimate is then adjusted according to the difficulty of developing each application point. 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 model developers.

Application composition usually relies on reusing existing software and configuring application systems. Some of the application points in the system will therefore be implemented using reusable components. Consequently, you have to adjust the

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

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

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

PM: the effort estimate in person-months.

NAP: the total number of application points in the delivered system.

%reuse: an estimate of the amount of reused code in the development.

PROD: the application-point productivity as shown in Figure 23.11.

23.6.2 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. The early design model assumes that user requirements have been agreed and initial stages of the system design process are underway. Your goal at this stage should be to make a quick and approximate cost estimate. Therefore, you have to make simplifying assumptions, such as the assumption that there is no effort involved in integrating reusable code.

Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements. The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

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

Based on his own large dataset, Boehm proposed that the co-efficient 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, which relate software size to function points for different programming languages (QSM 2014) 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 web 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:

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

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

PERS: personnel capability

PREX: personnel experience

RCPX: product reliability and complexity

RUSE: reuse required

PDIF: platform difficulty

SCED: schedule

FSIL: support facilities

The multiplier M is based on seven project and process attributes that increase or decrease the estimate. I explain these attributes on the book's web pages. You estimate values for these attributes using a six-point scale, where 1 corresponds to "very low" and 6 corresponds to "very high"; for example, PERS = 6 means that expert staff are available to work on the project.

23.6.3 The reuse model

The COCOMO reuse model is used to estimate the effort required to integrate reusable or generated code. As I have discussed in Chapter 15, software reuse is now the norm in all software development. Most large systems include a significant amount of code that has been reused from previous development projects.

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. Examples of black-box code are components that are automatically generated from UML models or application libraries such as graphics libraries. It is assumed that the development effort for black-box code is zero. Its size is not taken into account in the overall effort computation.

White-box code is reusable code that 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. White-box code could be automatically generated code that needs manual changes or additions. Alternatively, it can be reused components from other systems that have to be modified in the system that is being developed.

Three factors contribute to the effort involved in reusing white-box code components:

1. The effort involved in assessing whether or not a component could be reused in a system that is being developed.
2. The effort required to understand the code that is being reused.
3. The effort required to modify the reused code to adapt it and integrate it with the system being developed.

The development effort in the reuse model is calculated using the COCOMO early design model and is based on the total number of lines of code in the system. The code size includes new code developed for components that are not reused plus an additional factor that allows for the effort involved in reusing and integrating existing code. This additional factor is called ESLOC, the equivalent number of lines of new source code. That is, you express the reuse effort as the effort that would be involved in developing some additional source code.

The formula used to calculate the source code equivalence is:

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

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

ASLOC: an estimate of the number of lines of code in the reused components that have to be changed.

AT: the percentage of reused code that can be modified automatically.

AAM: an Adaptation Adjustment Multiplier that reflects the additional effort required to reuse components.

In some cases, the adjustments required to reuse code are syntactic and can be implemented by an automated tool. These do not involve significant effort, so you should estimate what fraction of the changes made to reused code can be automated (AT). This reduces the total number of lines of code that have to be adapted.

The Adaptation Adjustment Multiplier (AAM) adjusts the estimate to reflect the additional effort required to reuse code. The COCOMO model documentation (Abts et al. 2000) discusses in detail how AAM should be calculated. Simplistically, AAM is the sum of three components:

1. An assessment factor (referred to as AA) that represents the effort involved in deciding whether or not to reuse components. AA varies from 0 to 8 depending on the amount of time you need to spend looking for and assessing potential candidates for reuse.
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 that is being reused. SU ranges from 50 for complex, unstructured code to 10 for well-written, object-oriented code.
3. An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. These include design, code, and integration changes.

Once you have calculated a value for ESLOC, you apply the standard estimation formula to calculate the total effort required, where the Size parameter = ESLOC. Therefore, the formula to estimate the reuse effort is:

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

where A, B, and M have the same values as used in the early design model.

COCOMO 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 estimated values by the model developers.

http://software-engineering-book.com/web/cost-drivers/

23.6.4 The post-architecture level

The post-architecture model is the most detailed of the COCOMO II models. It is used when you have an initial architectural design for the system. The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

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

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 subsystems and components. You make this estimate of the overall code size by adding three code size estimates:

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 may be changed because of changes to the system requirements.

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. The value of the exponent B is 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.

Scale factor	Explanation
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.
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.
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.
Team cohesion	Reflects how well the development team knows each other and works 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 as discussed in web chapter 26. 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

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 (web chapter). These characteristics lead to estimates of the ratings used in exponent calculation as follows:

1. *Precedentedness,* rated low (4). This is a new project for the organization.
2. *Development flexibility,* rated very high (1). There is no client involvement in the development process, so there are few externally imposed changes.
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 final value of the exponent by dividing this sum by 100 and adding 0.01 to the result. 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 (see breakout box) 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.

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

Figure 23.13 The effect of cost drivers on effort estimates

Figure 23.13 shows how the cost driver attributes influence effort estimates. Assume that the exponent value is 1.17 as discussed in the above example. Reliability (RELY), complexity (CPLX), storage (STOR), tools (TOOL), and schedule (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 effort computation.

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 used are those from the COCOMO II reference manual (Abts et al. 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 project and the difficulties of transferring experience from one application domain to another.

23.6.5 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:

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

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

PM: the effort computed by the COCOMO model.

B: a complexity-related exponent, as discussed in section 23.5.2.

If B = 1.17 and PM = 60 then

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

The nominal project schedule predicted by the COCOMO model does not necessarily correspond with the schedule required by the software customer. You may have to deliver the software earlier or (more rarely) later than the date suggested by the nominal schedule. If the schedule is to be compressed (i.e., software is to be developed more quickly), 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 10 months. This represents a schedule compression of approximately 25%. Using the values for the SCED multiplier as derived by Boehm's team, we see that the effort multiplier for this level of 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 could 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 to a project reduces the productivity of existing team members. 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.

Consequently, when you add an extra person, the actual increment of effort added is less than one person as others become less productive. If the development team is large, adding more people to a project sometimes increases rather than reduces the development schedule because of the overall effect on productivity.

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 build-up of project staff has been shown to correlate with project schedule slippage. As a project manager, you should therefore avoid adding too many staff to a project early in its lifetime.

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 agile 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

Further reading suggested in Chapter 22 is also relevant to this chapter.

"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). http://dx.doi.org/10.1145/581571.581582

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. The book also includes a good, general introduction to project planning issues. (M. Cohn, 2005, Prentice-Hall).

"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

All About Agile; Agile Planning. This website on agile methods includes an excellent set of articles on agile planning from a number of different authors. (2007–2012). http://www.allaboutagile.com/category/agile-planning/

Project Management Knowhow: Project Planning. This website has a number of useful articles on project management in general. These are aimed at people who don't have previous experience in this area. (P. Stoemmer, 2009–2014). http://www.project-management-knowhow.com/project_planning.html

WEBSITE

PowerPoint slides for this chapter:

www.pearsonglobaleditions.com/Sommerville

Links to supporting videos:

http://software-engineering-book.com/videos/software-management/

EXERCISES

23.1. Describe the factors that affect software pricing. Define the "pricing to win" approach in software pricing.

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. Define project scheduling. What are the things to be considered while estimating schedules?

23.4. What is algorithmic cost modeling? What problems does it suffer from when compared with other approaches to cost estimation?

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

Task	Duration	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	T12, T14
T16	10	T15

Figure 23.14 Scheduling example

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 planning game is based on 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.

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

Abts, C., B. Clark, S. Devnani-Chulani, and B. W. Boehm. 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

Beck, K., and C. Andres. 2004. *Extreme Programming Explained: 2nd ed*. Boston: Addison-Wesley.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. 1995. “Cost Models for Future Software Life Cycle Processes: COCOMO 2.” *Annals of Software Engineering*: 1–31. doi:10.1007/BF02249046.

Boehm, B., and W. Royce. 1989. “Ada COCOMO and the Ada Process Model.” In *Proc. 5th COCOMO Users’ Group Meeting*. Pittsburgh: Software Engineering Institute. http://www.dtic.mil/dtic/tr/fulltext/u2/a243476.pdf

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

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

Cohn, M. 2005. *Agile Estimating and Planning*. Englewood-Cliffs, NJ: Prentice Hall.

QSM. 2014. “Function Point Languages Table.” http://www.qsm.com/resources/function-point-languages-table

Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.