# 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

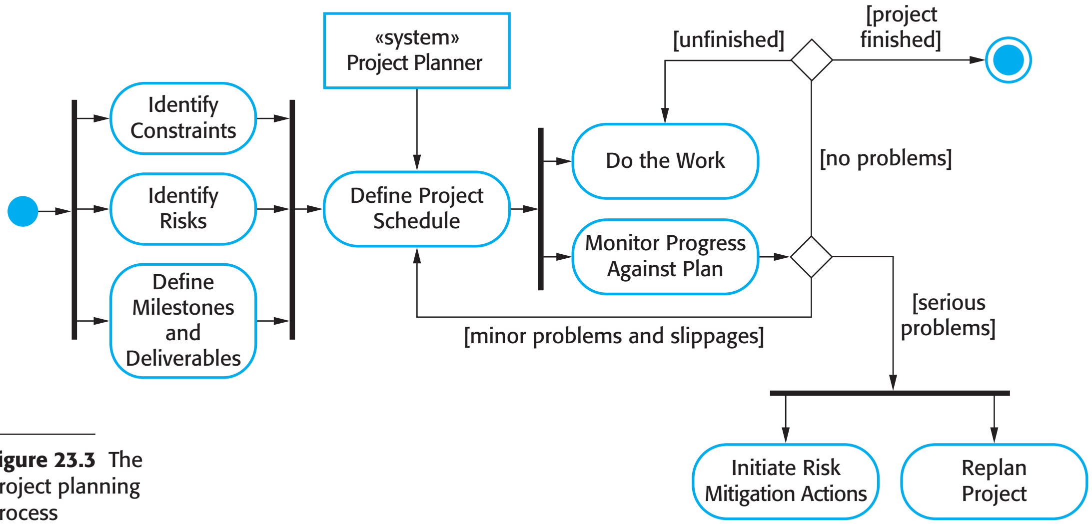


**Figure 23.3** The project planning process

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

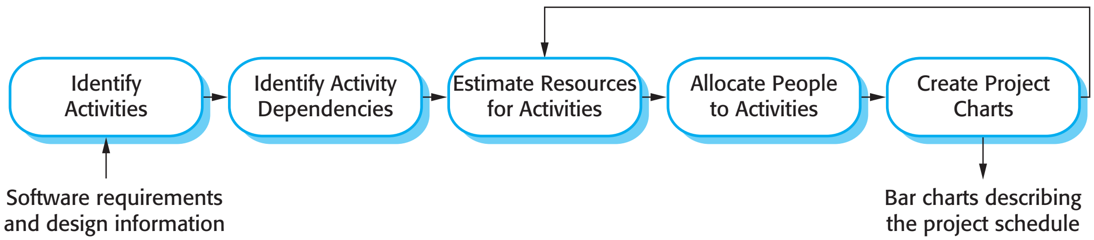


**Figure 23.4** The project scheduling process

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

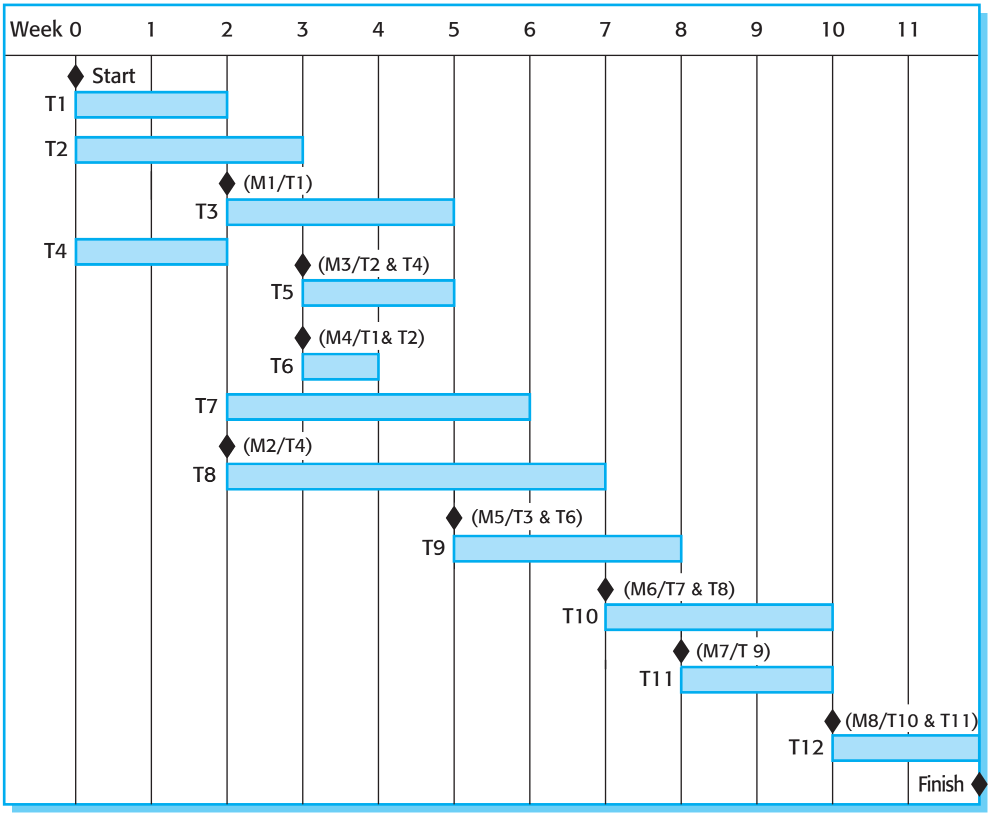


**Figure 23.6**
Activity bar chart

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.

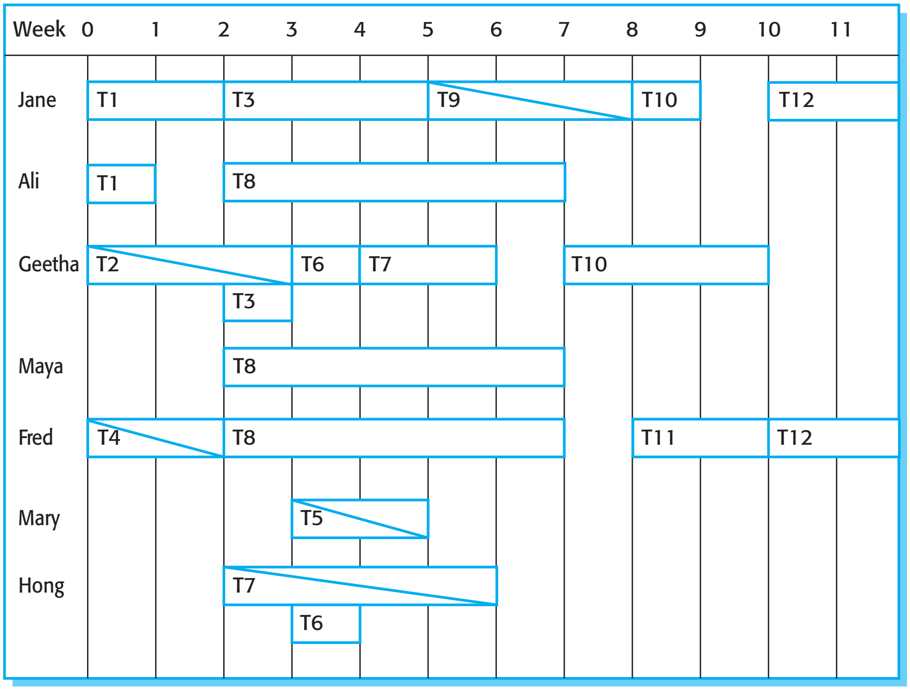


**Figure 23.7** Staff allocation chart

## 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.

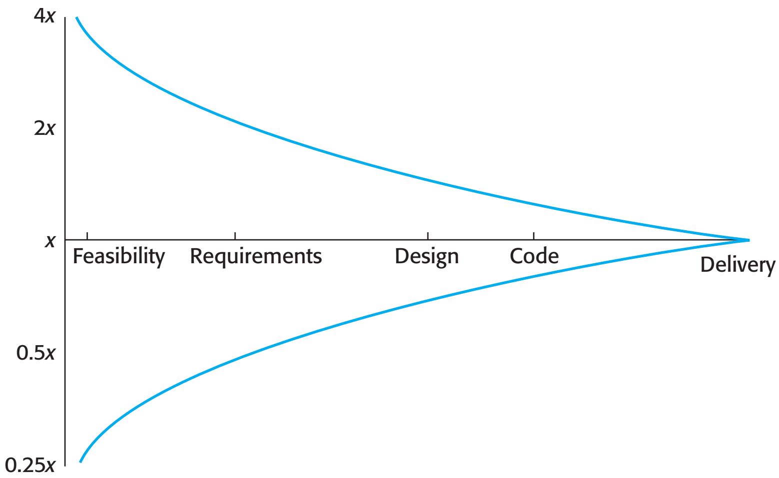


**Figure 23.9**
Estimate uncertainty

## 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:

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

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

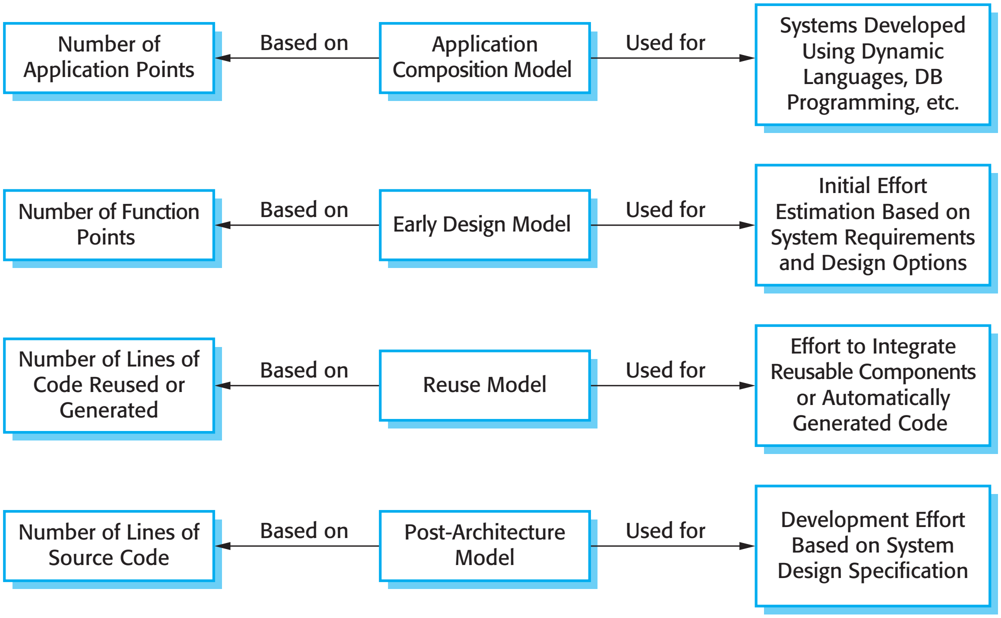


**Figure 23.10** COCOMO estimation models

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:

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

PM is the effort estimate in person-months. NAP is the total number of application points in the delivered system. "%reuse" is an estimate of the amount of reused code in the development. PROD is the 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:

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

where

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

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:

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

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:

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

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 sub-components that take into account design, code, and integration changes.
2. An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.
3. An assessment factor (referred to as AA) that represents the costs of reuse decision making. That is, some analysis is always required to decide whether 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:

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

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 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.