11 Managing the software process

Earlier in this book, we made the point that software engineering is a labor-intensive activity. Performing software engineering successfully therefore means carefully managing the people who perform it and organizing the tasks they perform. Project management is an activity that all software engineers will perform to some extent, even though they might not be officially given the title of manager or leader. In this chapter we introduce some of the basic principles of project management that all software engineers should know.

In this chapter you will learn about the following

- The different process models and methodologies that can be used to plan and conduct a software project.
- Techniques for estimating the amount of work it will take to develop a system.
- The basics of organizing software engineering teams.
- Techniques for planning, scheduling and tracking the work.

11.1 What is project management?

Project management encompasses all the activities needed to plan and execute a project. The following are specific activities often done by a project manager:

1. **Deciding what needs to be done**. Finding customers; working with customers to determine their problem and the scope of the project; prioritizing the work; selecting the overall processes that will be followed, and negotiating contracts.
2. **Estimating costs**. The most important aspect of this is estimating the amount of elapsed time and effort that will be required to complete the project. We will discuss this in Section 11.3.

3. **Ensuring there are suitable people to undertake the project**. This includes finding people, and ensuring that people have appropriate training. It can also include firing people who are not performing adequately.

4. **Defining responsibilities**. Determining how people will work together in teams and who will be responsible for what. Teams are the subject of Section 11.4.

5. **Scheduling**. Determining the sequence of tasks, plus setting deadlines for when tasks must be complete. Major deadlines are called *milestones*. Scheduling is discussed in Section 11.5.

6. **Making arrangements for the work**. Initiating the paperwork involved in hiring or subcontracting; setting up training courses; finding office space; ensuring that hardware and software is available; ensuring that people have the requisite security clearance, etc.

7. **Directing**. Telling subordinates and contractors what to do. Many of the other activities in this list involve making decisions; but acting on those decisions by ordering people to do things is a distinct activity. Directing is not as simple as issuing orders – you have to get people to commit to deliver what they promise.

8. **Being a technical leader**. Giving advice about engineering problems; helping people solve problems by leading discussions; pointing people to appropriate sources of information; acting as a mentor, and making high-level decisions about requirements and design.

9. **Reviewing and approving decisions made by others**. In certain types of projects, the project manager will have to take the ultimate legal responsibility for declaring that proper engineering practice has been followed, and that the manager believes the resulting system will be safe. However, a certain amount of reviewing and approving is a part of every project.

10. **Building morale and supporting staff**. Helping resolve interpersonal conflicts; ensuring that people feel rewarded, respected and motivated; giving people feedback to help them improve their work; and ensuring that people always have somebody to talk to about problems.

11. **Monitoring and controlling**. Finding out what is going on, determining how the plans need to change, and taking action to keep the project on track. The risk management process, which we have talked about throughout this book, is a central aspect of this.

12. **Co-ordinating the work with managers of other projects**.

13. **Reporting**. Telling customers and higher-level managers what they need and want to know.

14. **Continually striving to improve the process**.

Project management disasters

Periodically in the news, there are reports of major failures of software projects: projects that are canceled, in whole or in part, after spending large amounts of money and running far over budget and behind schedule. There are always many contributing factors to these failures, such as changing requirements or unexpected technological problems. However, with improved project management, the problems could have been detected and resolved without wasting so much money.

- The United States Internal Revenue Service struggled for years to update its software. The cost overruns and opportunity costs (failure to collect revenue) were estimated to exceed $50 billion per year.
- Attempts from the early 1980s to the mid-1990s to update the air traffic control systems in the United States met with many failures. One after another, components of the system were abandoned after billions of dollars had been spent and many years of elapsed time had passed.
- Early in 1993, the London Stock Exchange abandoned the development of its Taurus paperless share settlement system after 10 years and £400m were wasted.
- In 1995, the Canadian Auditor General's annual report found that large amounts of money had been wasted due to poorly managed projects in the Canadian Federal Government (see http://www.oag-bvg.gc.ca/domino/reports.nsf/html/9512ce.html).

Other references that discuss project management failures such as these can be found under the 'Important software failures' heading of the 'For more information' section in the previous chapter.

In some organizations, the role of 'project manager' is separated from the role of 'departmental manager'. Activities such as hiring, building morale, and issuing the final directions may then be performed by the latter individual instead.

We are introducing project management in this book since all software engineers need to have a basic understanding of how their managers run projects. Furthermore, all software engineers from time to time perform project management tasks. For example, software engineers have to estimate the amount of time their personal work will take; schedule their personal work; help to define the project by working on requirements analysis; act as a leader in their area of technical expertise; review the work of others; participate in risk management; co-ordinate with others; report to managers and work with others to improve the process.

In the next four sections, we will look at key project management skills: choosing an overall process model, estimating costs, building a team and scheduling. We will conclude the chapter by describing the project plan – an essential document maintained by the project manager.

Exercise

E213 Review the work you performed while you were doing group-oriented exercises in this book. For each of the project management activities, listed above, write down a) what you did (if anything), and b) how you believe you could do better.

11.2 Software process models

Software process models are general approaches for organizing a project into activities. They help the project manager and his or her team to decide what work should be done and in what sequence to perform the work. The models should be seen as *aids to thinking*, not rigid prescriptions of the way to do things – each project will end up with its own unique plan.

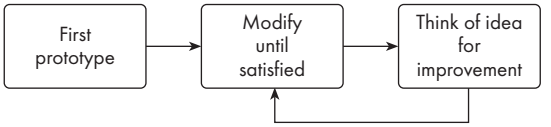


Figure 11.1 **The opportunistic approach, a poor model for software development**

Organizations that don't follow good software engineering practices often end up following what is often called an *opportunistic approach*, illustrated in Figure 11.1. In the opportunistic approach, developers keep on modifying the software until they or their users are satisfied. This approach has several important problems:

- It does not acknowledge the great importance of working out the requirements and the design before starting implementation. The system might satisfy certain user needs, but reaching a high level of user satisfaction will require many changes.
- The design of software deteriorates faster if it is not well designed. In the opportunistic model, design is an ad-hoc activity, therefore rapid deterioration is to be expected.
- Since there are no plans, there is nothing to aim towards. Since there is nothing to aim towards, you can never know if you are doing well or poorly. Therefore there is no *control* of costs or schedule in an opportunistic project.
- There is no explicit recognition of the need for systematic testing and other forms of quality assurance. Many undetected defects therefore remain, giving rise to never-ending changes that make the system worse and worse. As a result, the system most often collapses at some point and becomes unusable and irreparable. The only solution is then to rebuild the system from scratch (until it collapses again).
- The above problems make the cost of developing and maintaining software very high.

The waterfall model

The *waterfall* model is a significant improvement over the opportunistic approach. It is a classic way of looking at software engineering that accounts for the importance of requirements, design and quality assurance. The model is so named because diagrams of it, such as Figure 11.2, tend to look like cascading waterfalls.

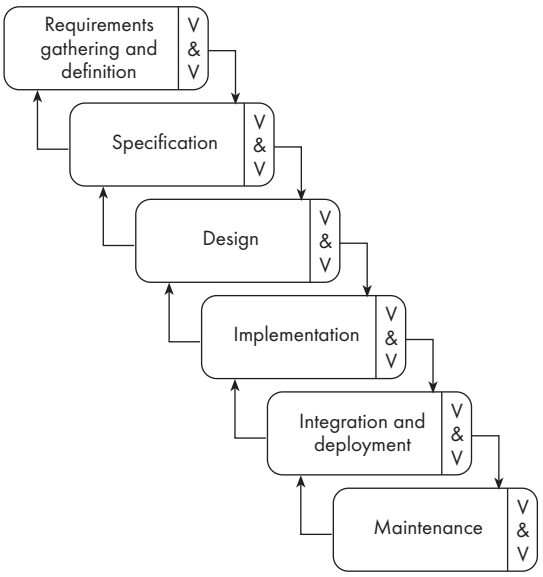


Figure 11.2 The waterfall model

Rather than jumping in and immediately developing a product, the waterfall model suggests that software engineers should work in a series of stages. Before completing each stage, they should perform quality assurance (verification and validation) so that the next stage can be built on a good foundation. The waterfall model also recognizes, to a limited extent, that you sometimes have to step back to earlier stages when you discover a problem in a subsequent stage.

The waterfall model forms the foundation of many software development methodologies in use today. However, it has some limitations and, if followed too strictly, can lead to the following types of problems:

- The model implies that you should attempt to complete the entire specification before moving on to the design, and the entire design before moving on to implementation. This is an overly rigid viewpoint since it does not account for the fact that requirements constantly change. It also means that customers cannot use anything until the entire system is complete.
- The waterfall model makes no allowances for prototyping and implies that you can get the requirements right by simply writing them down and reviewing them. In practice, this only works for simple, well-understood types of software development.
- The model implies that once the product is finished, everything else is maintenance. Relegating maintenance to be the 'last step' makes it appear that all you have to do at that point is to make minor adjustments, perhaps without requirements and design. Unfortunately, most of the costs occur after the initial system is developed as the system is repeatedly changed.

The phased-release model

The *phased-release model* of software development, shown in Figure 11.3, rectifies some, but not all, of the problems of the waterfall model. The most important change is that it introduces the notion of *incremental* development. It suggests that, after requirements gathering and planning, the project should be broken into separate subprojects, or phases. Each phase can then be released to customers when ready. Parts of the system will be available earlier than would have been possible when using a strict waterfall approach.

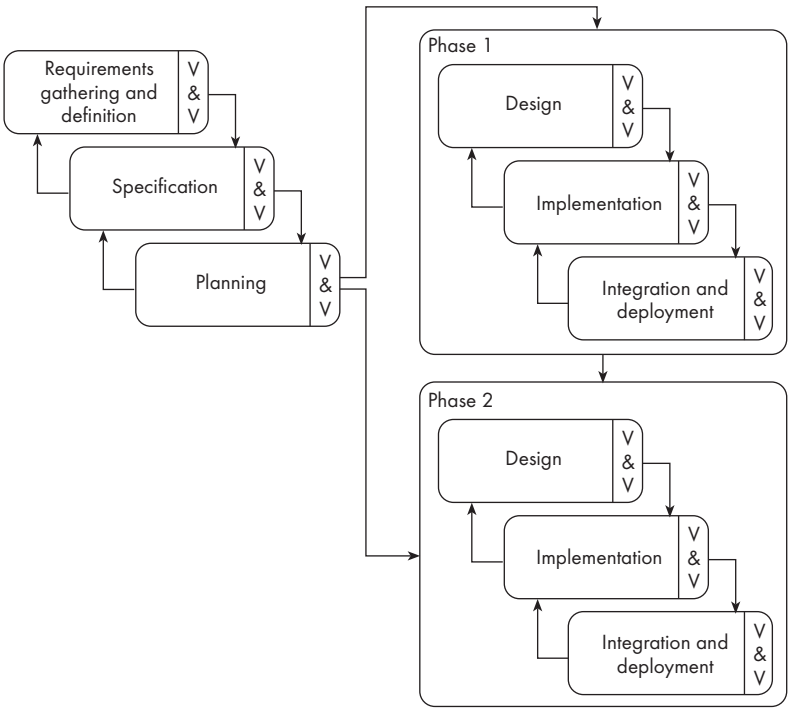


Figure 11.3 **The phased-release model**

The phased-release model still retains some of the important weaknesses of the waterfall model. It continues to suggest that all requirements be finalized at the start of development, and it continues to downplay the possibility of prototyping. When the time comes to develop Phase 2, and subsequent phases, the design is based on the original specifications – in other words, the model does not facilitate learning lessons from Phase 1, which could result in improvements to subsequent phases.

The spiral model

The *spiral model*, as shown in Figure 11.4, is another view of incremental development that explicitly embraces prototyping and an *iterative* approach to software development. This model takes the position that you should start to

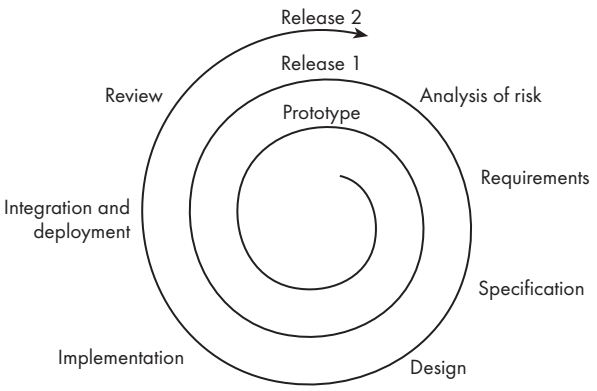


Figure 11.4 **The spiral model**

develop software by developing a small prototype (innermost loop of the spiral). This first prototype follows a mini-waterfall process, but is very quickly developed and serves primarily to gather requirements. The last stage of this inner loop is review, or evaluation, of the first prototype.

Project managers embrace the spiral model because it acknowledges one of the most famous quotations in software engineering. It was Fred Brooks who said, 'The question is not whether to build a pilot system and throw it away. You will do that. The question is whether to plan in advance to build a throwaway.'

In subsequent loops of the spiral, the project team performs further requirements, design, implementation and review. There may be several cycles of prototyping; however, subsequent cycles become official releases.

Before each cycle of the spiral ends, a review is held. At this review, the stakeholders discuss their experiences with the previous release and decide whether to proceed to another cycle.

The spiral model also adds the notion of *risk analysis* to process modeling. The first thing to do before embarking on each new loop of the spiral is to decide what are the major difficulties to be handled. After determining these, you then make adjustments to the architecture, the requirements or the project plan as necessary.

When following the spiral model, a project undergoes a large number of cycles. The cycling only ends when the system is finally retired. This model therefore incorporates the idea that maintenance is simply a type of ongoing development.

The evolutionary model

The *evolutionary model* (Figure 11.5) shows software development as a series of hills, each representing a separate loop of the spiral. This is a third way of thinking about incremental development.

This model shows two things that are not always clear from the spiral model. First, it shows that loops, or releases, tend to overlap each other. As testing and

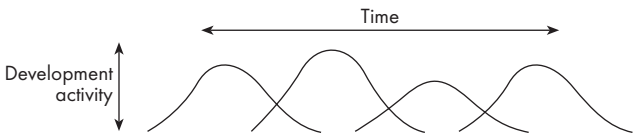


Figure 11.5 The evolutionary model

preparations for deployment of one release are under way, planning for the next release has already started.

Secondly, the evolutionary model makes it clear that development work tends to reach a peak, at around the time of the deadline for completion of implementation. Finally, the model shows that each prototype or release can take different amounts of time to deliver, and can take differing amounts of effort.

The concurrent engineering model

The *concurrent engineering model* (Figure 11.6) explicitly accounts for the divide and conquer principle. Each team works on its own component, typically following a spiral or evolutionary approach.

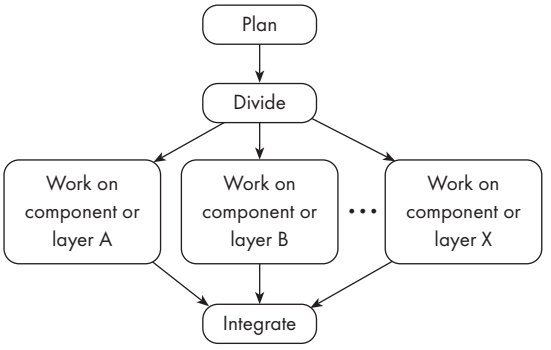


Figure 11.6 The concurrent engineering model

In the concurrent engineering model, there has to be some initial planning, and periodic integration. The diagram therefore only shows a picture of a single iteration.

The Rational Unified Process

Many of the features of the above models are embodied in what is known as the *Rational Unified Process (RUP)*. This is the most widely known published methodology that embraces UML and most of the other techniques discussed in this book.

RUP is designed to be adaptable: it suggests a framework and a set of disciplined practices to follow, but encourages you to vary the level of rigor depending on your organization's needs.

See the 'For more information' section at the end of Chapter 5 for the definitive book about RUP.

Agile approaches

We have discussed agile approaches periodically throughout this book. These approaches embrace the iterative model, encouraging the development of particularly small iterations, and the reassessment of the direction of the project as each new iteration is being considered. The techniques also explicitly disavow 'big' processes: by which we mean processes that require the development of lots of documentation and have delivery dates far in the future.

Agile approaches are gaining popularity for small projects that involve uncertain, changing requirements and other sources of high risk. Risk is reduced because nothing big is produced – if a small iteration fails to meet the needs of the users and has to be re-done, only a small amount of money is wasted.

Some people, upon initially hearing of agile techniques, think that they are perhaps throwing away all the discipline inherent in some of the other techniques we have talked about. This misconception is far from the truth: all the agile approaches promote using disciplined techniques; the key is that each of these techniques is lightweight – that is, it is not something that will bog down the project or cost a lot of money. In fact many of the techniques are explicitly designed to save money.

The most famous of the agile techniques is eXtreme Programming (XP). Some of the tenets of eXtreme Programming are:

- The development team includes all the stakeholders, who work very closely together. In particular, it is considered important to have a customer or user on site.
- You do not develop large requirements documents. Instead, you write a series of up to about 80 *user stories* that describe things the users want the system to do for them (see the sidebar in Section 4.8). Each user story must take a couple of weeks to develop and test; if developers estimate it would take longer, then it must be broken down into smaller stories.
- Project planning is based on the user stories. There must be a series of small and frequent releases. A release is divided into iterations of between 1 and 3 weeks. The stakeholders work together to determine which stories will be released in each iteration. Planning takes place just before the start of each iteration, and *project velocity* is measured and used to gauge how much time a set of user stories will actually take to develop. During each iteration, programmers sign up to do tasks that must take between 1 and 3 days – programmers do their own estimates of how much time each task should take.
- There are three project variables that can be changed: *scope*, *resources* and *time*. Management is only allowed to dictate any two of these; the developers determine the third. Quality, a fourth variable, should not be sacrificed.

- In order to ensure high quality, design for testability and test-first development (see Chapter 10) are emphasized: automated test cases are written before the software is developed. As failures occur, new automated test cases are written. Acceptance testing is based on the user stories.
- A large amount of *refactoring* is encouraged. Refactoring means transforming the design to ensure internal qualities such as maintainability are promoted. Simple examples of refactoring include adding a superclass that contains common features of existing classes, and adding a Façade design pattern around a package to reduce coupling. Frequent refactoring ensures the design retains high internal quality despite the fact that design only considers one small iteration at a time. We will have more to say about refactoring in the context of re-engineering below.
- *Pair programming* is recommended. The idea behind this is that defects are much less likely to creep into code if two people are involved in its creation. Both people also keep each other engaged and focused on the task.

Extreme programming also promotes many other ideas, such as the use of CRC cards, a focus on simplicity, creation of 'spike' throwaway prototypes when difficult technical issues are encountered, frequent integration, and continual improvement. Another part of the philosophy is that software developers should not need to be overworked; overtime is therefore frowned on.

There are some drawbacks to agile methods. It is harder to write a contract for development of a system when neither party knows the direction the project will take. Also, some of the techniques become less and less appropriate as the system becomes larger and larger. Nevertheless, many developers and managers now enthusiastically support the use of agile techniques for a wide variety of types of project.

The open source model

A model that is widely used for software development is the open source model. In this model software is distributed for free along with the source code, and interested people contribute improvements, without being paid by users.

Many open source projects were founded by enthusiasts who initially developed a system largely as a hobby, or perhaps as a student research project. People who contribute to the projects often benefit because they are users of the software and want it to become better and better. The idea is that community members will make the software ever better for themselves and others in the community.

The open source movement has resulted in a wide variety of important systems, including the Linux operating system and the GNU tools. Some companies allow their developers to participate in open source projects; IBM, for example, has heavily promoted the Eclipse software development tool.

In the open source model, a small group of leaders must retain control of which contributions make it into the final release. Quality assurance is

performed by the community: those interested in the project scrutinize and criticize any suggested contributions.

Choosing a process model

When planning a particular project, the important thing to recognize is that you can combine the features of the models that apply best to your current project.

From the waterfall model, you will always incorporate the notion of stages, but you will most likely want to avoid having a single cascade ending in maintenance.

From the phased-release model, you can incorporate the notion of doing some initial high-level analysis, and then dividing the project into releases. You can also incorporate the notions of prototyping and risk analysis from the spiral model.

Next, from the evolutionary model you can incorporate the notion of varying amounts of time and work, with overlapping releases. And from the concurrent engineering model, you can break the system down into components and develop them in parallel.

Finally, even if you are developing a large system, you can adopt many of the features of the agile model: smaller releases, on-site users, test-first development and frequent refactoring can all be adopted in projects both large and small.

Re-engineering

No matter what process model you use, in any large or long-lived project, the design will deteriorate. Periodically, therefore, project managers should set aside some time to re-engineer part or all of the system. The extent of this work can vary considerably: it could include cleaning up the code to make it more readable, completely replacing a layer, or refactoring part of the design.

In general, the objective of a re-engineering activity is to increase maintainability. It should not normally involve adding any new features for users, but should make the system more flexible so that adding such features in the future becomes easier.

Although periodic re-engineering will reduce long-term costs, it is often hard to convince managers and customers of this. You need to present them with a detailed analysis of how much the re-engineering will cost, and what cost savings will result. As with many aspects of software engineering, you should follow the 80–20 rule: you can get 80 per cent of the benefits of re-engineering with only 20 per cent of the work, therefore focus re-engineering efforts on the parts of the system most in need of it.

11.3 Cost estimation

One of the biggest challenges in software engineering is accurately forecasting how much time and effort it will take either to develop a system, or to make a specific set of changes. All software developers have to participate in cost

Chaos in project management

The Standish Group's reports on project management chaos contain some of the most respected statistics about software project failures. According to the Standish Group, the average project exceeds its budget by 90%, and its schedule by 120%. Furthermore, over 30% of projects are canceled before completion. In the United States alone, this represents about $100 *billion* in wasted money per year. See http://www.standishgroup.com.

estimation, no matter whether they are the managers or architects of a large system, or whether they are the engineers responsible for the design and programming of a single component.

There is an old cliché that 'time is money'. In software project management, this is literally true. When you estimate the cost of doing some work, you focus on estimating how much software-engineering time will be required. However, there are two distinct aspects of time. The first is *elapsed time*; this is the difference in time from the start date to the end date of a task or project. The second aspect of time is probably better called *development effort* (or simply *effort*); this measures the amount of labor used and is expressed in *person-months* or *person-days*.

When planning a project, you budget a certain amount of effort for each task, and you schedule the start and end times for that task. If you schedule a task to take two weeks of elapsed time, but take six person-weeks of effort, that means that you must also ensure that an average of three people are working on the task at any given point in time.

The elapsed time of a project is important – you might have a contractual obligation to deliver software by a certain day, or you might want to ensure that your product reaches the market before the competition. However, the actual cost of a project is primarily a function of development effort.

To convert an estimate of development effort to an amount of money, you multiply it by the *weighted average cost* (also called the *burdened* cost) of employing a software engineer for a month (or a day). The weighted average cost not only includes the average salary of a software engineer, but also the cost of providing that person with benefits, an office, a desk, a computer as well as technical and managerial support. The weighted average cost is therefore often two to three times average salary.

Example 11.1 *In your organization, although the average salary is $4,000 /month, the weighted average salary for cost estimation purposes is $11,000/month. You have determined that a particular project will take 7 person-months to complete. How much would you estimate this project will cost financially?*

You estimate that the project will cost 7×$11,000 = $77,000.

Principles of effective cost estimation

Cost estimation is notoriously difficult, as witnessed by the large number of projects that are completed behind schedule and over budget, or are not completed at all. There are several key principles that can help you to make better estimates. These can be applied whether you are estimating the time for your personal work, or for an entire project.

Cost Estimation Principle 1: Divide and conquer

In Chapter 9, we said that divide and conquer was one of the essential principles of design. In Chapter 10, we said it also applies to testing. It turns out to be just as important to cost estimation.

If you try to estimate the entire cost of a project as a single number, you are likely to be very inaccurate. To make a better estimate, you should divide the project up into individual subsystems, and then divide each subsystem further into the activities that will be required to develop it. Next, you make a series of detailed estimates for each individual activity, and sum the results to arrive at the grand total estimate for the project.

Although your detailed estimates may be inaccurate, your final estimate will be more accurate, for two reasons. Firstly, you will be more likely to account for all the subsystems and activities. Secondly, if you underestimate the time required for some subsystems or activities, this should be at least partly compensated for by overestimates in other places.

Although it will help you improve estimates, the divide and conquer principle is not a panacea. You might miss activities or systematically underestimate each activity, leading to a total estimate that is too low. The other principles listed below will help to combat this.

Cost Estimation Principle 2: Include all activities when making estimates

If you do not appreciate the amount of effort required for certain activities, or omit them entirely, then your estimate of total effort will be too low.

For example, when asked to estimate the cost of a new feature for SimpleChat, a beginner might focus primarily in the amount of time required to write the requirements document and the code. However, the time required for *all* development activities must be taken into account, including prototyping, design, inspecting, testing, debugging, writing user documentation and deployment.

Cost Estimation Principle 3: Base your estimates on past experience combined with what you can observe of the current project

The only way to predict the future is to reason by analogy with the past. If you are developing a project that has many similarities with a past project, then you can expect it to take a similar amount of work.

In practice, no two projects are the same. However, the more you follow Principle 1 and divide the estimation task into fine-grained detailed estimates, the more likely you are to be able to find similarities with aspects of past projects.

There are two general strategies for using past experience. The first is to base your estimates purely on the personal judgment of experts within your team. Such people will have worked on other projects and can extrapolate their experience to the current project. The second strategy is to use algorithmic models that have been developed in the software industry as a whole by analyzing a wide range of

development projects. They take into account various aspects of a project's size and complexity, and provide formulas to compute anticipated cost.

The algorithmic models allow you to systematically base your estimate of development effort on an estimate of some other factor that you can measure, or that is easier to estimate. Project managers base their estimates on factors such as the following:

- The number of use cases.
- The number of distinct requirements.
- The number of classes in the domain model.
- The number of widgets in the prototype user interface.
- An estimate of the number of lines of code.

Which of the above factors you use as a basis for your estimate depends on how far you have progressed in development. For example, you may have completed an initial use case model, but have not yet finalized the detailed requirements or developed a prototype user interface.

Example 11.2 *You have been asked to estimate the cost of a project for which you developed a system domain model composed of 24 classes. In a recent similar project, the average development effort (including all testing etc.) was 10 person-days per class. What would be your cost estimate for the current project?*

You might first assume from past experience that the system domain model classes represent half of the total number of classes in the final system. The other classes would be user interface and architectural classes. The development effort estimate would therefore be $24 \times 2 \times 10 = 480$ person-days = 16 person-months. Using the burdened cost from Example 11.1, this results in a cost estimate of 16x$11000 = $176,000.

Project managers often use lines of code to give an intermediate estimate of system size that people can easily understand. For example, you might convert an estimate of use cases to an estimate of lines of code, and then convert the lines of code to an effort estimate using one of the formulas discussed below. However, you can only effectively base cost estimates on lines of code when you have almost completed design.

Example 11.3 *Your records of previous projects show that an average class has 6 attributes, and 14 methods, averaging 6 lines of code each. The average lines of code per class is 90. How many lines of code would you expect to have to develop in the system of Example 11.2?*

You would expect to develop $90 \times 48 = 4.3$KLOC.

A typical algorithmic model uses a formula such as the following:

$$E = a + bN^c$$

In this formula, E is the effort estimate and N is the estimate or measure being used as the basis for the effort estimate (e.g. number of use cases or lines of code). The values a, b and c are obtained by extensive analysis of past projects, and determinations of the differences that will effect the current project (these will be discussed in Cost Estimation Principle 4).

The exponent c is particularly interesting. If it is not equal to one, it means that the effort increases *non-linearly*. If $c < 1$, there would be *economies of scale* as a project gets larger, which turns out *not* to be the case in software. In real projects $c > 1$, which means that the effort grows increasingly rapidly relative to project size; this is due to the increasingly large amount of co-ordination and complexity involved.

Figure 11.7 shows the effect of typical values of the c parameter used by one of the best-known algorithmic cost estimation models, COCOMO II (COnstructive COst Model, version II). COCOMO II computes effort, in person-months, from an estimate of size, measured in lines of code.

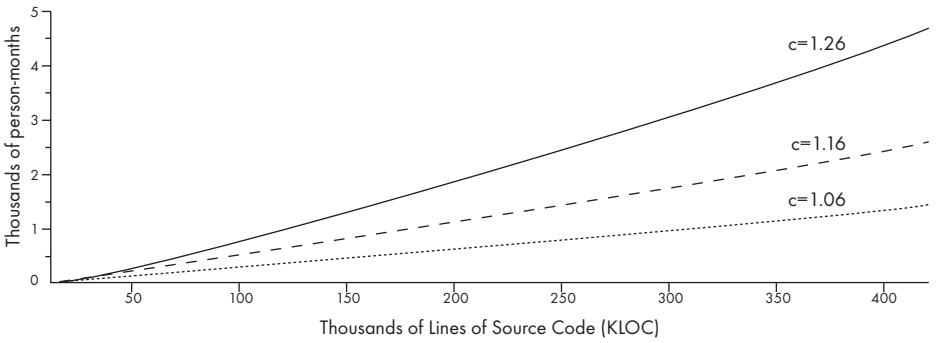


Figure 11.7 **Effect of non-linearity in cost estimation**

Another important algorithmic method is *Function Point Analysis*. In this approach, and several related approaches, you count features of the requirements and use these to compute an estimate of the system's size. You can then apply COCOMO to the size estimates.

The basic equation used by approaches like Function Point Analysis is:

$$S = W_1F_1 + W_2F_2 + W_3F_3 + \ldots$$

The F_i represent counts of features of the requirements such as number of inputs, number of tables in the database, number of use cases etc. These counts are multiplied by weights (the W_i values) calculated using industry-wide experience. The results are summed to produce a system size, S.

To find out more about Function Points and COCOMO, see the references in the 'For more information' section at the end of the chapter.

Estimating the cost of construction – false analogies

Some people yearn for the day when it will be as easy to estimate software development time as it is for a civil engineer to estimate the cost to construct a small building. Civil engineers and builders have tables that allow them, for example, to calculate how much time it will take to lay a certain amount of concrete, or to perform each of the other construction tasks.

Why can't we do this in software engineering? The answer, as we hinted in Chapter 1, is that it is false to draw analogies between 'construction' of buildings and the latter stages of software development that are sometimes (unadvisedly we believe) also called 'construction'.

Software development is an intensely mental activity. Like the *design* of buildings, it is full of decision-making and other intellectual challenges, all the way to the end of programming and testing. Constructing a building should actually be seen as analogous to the printing of disks or making the executables available for downloading. Because software is largely intangible, this real construction process is very inexpensive.

The dominant costs in software engineering are therefore design costs, which are hard to estimate. The dominant costs in many other branches of engineering are true construction costs, which are much easier to estimate.

This does not mean we are doomed to failure; it just means that we should expect a realistic level of uncertainty in our estimates, and that we should practice iterative development and risk management so as to mitigate the effects of those uncertainties.

Cost Estimation Principle 4: Be sure to account for differences when extrapolating from other projects

Although experience from other projects can be a good guide, there are often many subtle differences between projects. You have to consider carefully the effect of differences such as the following when making an estimate for a new project:

- **Different software developers**. People differ dramatically in their skill and experience levels; a skilled programmer can be up to ten times as productive as a less skilled programmer. In projects with only a few people, this can therefore significantly influence the accuracy of estimates.
- **Different development processes and maturity levels**. Teams that have skilled management and a mature development methodology that includes such things as quality assurance, risk management and iterative development will be able to work more efficiently than organizations that follow an ad-hoc approach.
- **Different types of customers and users**. Project teams that have good access to and rapport with both their customers and users can often proceed faster than organizations lacking these advantages. In the latter case, delays are caused by slow or poor decision making. Projects that have a single user or customer will often also proceed faster than projects with many users and customers. Not only does it take time to negotiate decisions when many people are involved, but the resulting compromises may be difficult to develop efficiently.

- **Different schedule demands**. Ironically, if a project team is under intense pressure to deliver software by a certain date they may cut corners and make mistakes that actually end up delaying them. On the other hand, a certain amount of deadline pressure can have a positive effect.
- **Different technology**. The effort required can be affected by changes in hardware, other software systems with which your system must interact, database management systems, operating systems and programming languages. For example, if you change to a new programming language, that language can take considerable time to learn. Also, some languages are known to be less productive than others. For example, programming in assembler is unproductive because it takes many statements to express a given computation, and furthermore the complexity of assembler gives rise to more bugs. Programming in C or C++ is likely to result in faster development. Using Java or C# is likely to decrease development time still further.
- **Different technical complexity of the requirements**. Some systems are intrinsically more complex than others, because they require greater reliability, are distributed, or for many other reasons.
- **Different domains**. A team embarking on a project in a domain in which it has not worked before will undoubtedly take more time than a team that has been working in the same domain for years.
- **Different levels of requirement stability**. Some projects are easier to define in advance. On the other hand, some projects must proceed in an exploratory fashion since the requirements will not be clearly known until prototyping is complete. The total effort will thus vary substantially. Also, some domains are more prone to changes in requirements than others.

The algorithmic cost-estimation techniques explicitly take into account factors such as the above, by multiplying the estimate by a small factor, or increasing the value of an exponent.

In COCOMO II, the basic value of the exponent c in the formula presented earlier is 1.01. You add to this exponent amounts ranging from 0.00 to 0.05 depending on the impact you expect from: a) the level of experience with this kind of project; b) the amount of flexibility you have in the development process; c) the amount of risk management performed; d) the quality of the development team, and e) the process maturity of the organization.

COCOMO II also has a series of multiplicative factors that modify the value of b in the formula. These factors are divided into four categories: Personnel, Product, Project and Platform attributes. Typical initial values of b might be 2.4 for a small project and 3.0 for a medium-sized project.

Example 11.4 *This example continues examples 11.2 and 11.3. Assume that you are working on a small project and will use 2.4 as your multiplicative value,* b. *To compute the exponent, suppose we have, for this project, good experience with similar project*

(0.01), no involvement of the client in the process (0.00), little risk analysis carried out (0.04), a relatively good team of developers (0.02) and an average process maturity (0.03). Use this data to compute a cost estimate in person-months.

The exponent value will be 1.01+0.10 = 1.11. The effort would then be $2.4 \times 4.3^{1.11} = 12.1$ person months.

Example 11.5 *Looking up in COCOMO II tables, you find that the following multiplicative factors need to be applied to adjust the value of the* b *(multiplicative) parameter:*

(a) Personnel factor: your team has good experience with the programming language to be used: 0.95

(b) Product factor: the system will be running under very tight memory constraints: 1.21

(c) Project factor: you have to put up with a schedule that is a bit tight: 1.04

(d) Platform factor: particularly high reliability is required: 1.4

Revise your effort estimate from Example 11.4, given this new data, and convert the result to dollars.

The effort estimate is $0.95 \times 1.21 \times 1.04 \times 1.4 \times 12.1 = 20$ person-months, for a cost of $20 \times \$11{,}000 = \$220{,}000$.

Exercise

E214 Using your judgment, what would be reasonable multiplicative parameters for the following (you could do this with reference to the COCOMO II documents).

(a) Personnel factor: your team has in the past developed many similar applications.

(b) Product factor: the system includes a particularly large database.

(c) Project factor: some of your development team will be located in the US, some in Canada, some in India and some in South Africa.

(d) Platform factor: the application will run under the Linux operating system.

Cost Estimation Principle 5: Anticipate the worst case and plan for contingencies

In accordance with Murphy's Law, if something can go wrong in a project, it probably will. For example, you might have difficulty deciding on the requirements; there might be unexpected technical challenges in the design process; or somebody might quit or not perform up to expectations.

One way to plan for contingencies is to prioritize all the use cases according to their benefit to the customer. If, as the project progresses, your revised cost

estimates show that you will exceed your budget or deliver the software too late, you can drop the lowest priority use cases, and update your estimates accordingly.

Another important thing to do is to build enough 'cushion' into your cost estimate so as to account for typical delays. One way to do this is as follows. For every detailed estimate, make an 'optimistic' (O) estimate, a 'likely' (L) estimate and a 'pessimistic' (P) estimate. Your O estimate suggests the minimum time you reasonably think the activity might take. Your L estimate accounts for what you think would be a typical number of things going wrong. Your P estimate suggests what you think the activity would consume if many difficulties were experienced. You then add up the O estimates, the L estimates and the P estimates separately to arrive a global estimates of the best-case, typical-case and worst-case cost for the project.

Having these three separate estimates ensures that you are not making the mistake of providing only a best-case estimate. Your pessimistic estimate also helps you to become conscious of all the things that could go wrong, so that you can take active steps to avoid them.

Example 11.6 *You are asked to estimate the cost of developing the Release 2 of the GANA system introduced in Chapter 4. The new release would:*

- *Constantly update the estimated time of arrival at the destination, based on typical travel speeds on the roads to be taken, the time of day, and day of the week (accounting for rush hours).*
- *Dynamically adjust the scale of the map based on the local population density, the distance of the driver to his or her origin or destination, and the driver's speed. The map would not jump from one scale to another but would smoothly increase or decrease its scale.*

The following table shows Optimistic, Likely and Pessimistic estimates for the work required, expressed in person-months. Note that more detailed estimates could be created if some of the tasks were divided into subtasks.

	Optimistic	*Likely*	*Pessimistic*
Requirements gathering	2.5	3	4
Prototyping	2	3	5
Specification	1	1.5	3
Design	4	6	8
Implementation	5	8	16
Software inspection and testing	6	10	13
Hardware integration and system test	1.5	2.5	4
Total Time	22	34	53

Exercise

E215 Evaluate the cost estimates presented in Example 11.6; discuss potential sources of error in these estimates.

E216 Search the Internet for the source code of an open-source Java project (you can also select a package from the Java API). Then do the following:

(a) Compute some basic metrics: the average number of attributes and methods per class; and the average number of lines per method.

(b) Imagine you are planning to add a new subsystem of 40 classes very similar to the system you have studied. Use the numbers computed in part a) to derive likely, optimistic and pessimistic estimates for the number of lines of code you would expect to find in this system.

(c) Use the COCOMO approach discussed earlier to arrive at optimistic, likely and pessimistic estimates for the effort required to develop the subsystem.

Cost Estimation Principle 6: Combine multiple independent estimates

No matter how good your estimation process, you are likely to have overlooked some factors that will affect the accuracy of your estimate.

You should therefore use several different techniques and compare the results. If there are discrepancies, you can analyze your calculations to discover what factors are causing the differences. For example, you can make one estimate based on your experience with a similar project; you can then use COCOMO or Function Points to make a second estimate.

A well-respected approach to making multiple estimates is the Delphi technique. To use this technique, several individuals initially make cost estimates in private. They then share their estimates to discover the discrepancies. Each individual repeatedly adjusts his or her estimates until a consensus is reached.

Example 11.7 *Discuss the discrepancies between the cost estimates presented in Examples 11.2 and 11.5, and suggest a course of action.*

The estimate of $176,000 from example 11.2 was based purely on extrapolating from this organization's typical costs per class developed. The $220,000 estimate of example 11.5 was based on a much more sophisticated sequence of computations, with a lines-of-code estimate as an intermediate value. In particular, the specific attributes of the *current project* that differentiate it from previous projects were taken into account. You should probably therefore put more weight on the second estimate, although it has one weakness: some factors used in the calculations were based on industry-wide data, not your own company's data.

You should probably attempt to use other cost-estimation techniques to see if they provide any more evidence that would help make the estimate more accurate. You should also closely monitor progress and revise your estimate regularly.

Cost Estimation Principle 7: Revise and refine estimates as work progresses

When you set out to solve a customer's problem you will have very little idea what software is needed (if any), therefore your estimate of the amount of work required can only be very rough. You will refine your estimates for three reasons:

- **As you add detail**. As you gather requirements and begin specifying details, you will be able to increase the accuracy of your estimate. As you move into the design phase, you can again increase the accuracy of your estimates.
- **As the requirements change**. You will adjust your estimates as requirements change, or features are dropped in order to meet a budget or deadline.
- **As the risk management process uncovers problems**. Similarly, as you encounter problems during design and implementation, you will be able to adjust your estimates to take these into account.

Cost estimation should therefore be a continuous activity. Updated estimates should be reported regularly to other members of the team, to management and to customers. A classic project management failure is to keep reporting that the project is 'on time and on budget' until the expected delivery date, and then to announce that the project is six months behind schedule.

Exercise

E217 Working in groups of four, estimate the cost of developing Release 1 of the GANA software, whose requirements were described in Chapter 4. Use as many of the techniques described in this section as possible. Among other things, this means you will use the 'divide and conquer' strategy to divide the software into parts, and divide the development process into detailed activities. You should also make optimistic, likely and pessimistic estimates. Each group should initially split into pairs, with each pair doing the estimation independently. Then the two pairs can come together to compare and reconcile their estimates.

11.4 Building software engineering teams

Software engineering is a human process. Choosing appropriate people for a team, and assigning roles and responsibilities to the team members, is therefore an important project management skill.

Software development contracts based on uncertain cost estimates

Customers often want a contract for custom software development to have a 'fixed price' so that they are not exposed to the risk of inaccurate estimates. On the other hand, software developers want a contract to allow them to charge for whatever 'time and materials' they use; this protects the developers from inaccurate cost estimates but exposes the customers to all the risk.

A solution to this conflict is to make an estimate for the entire project, but only initially sign a fixed-price contract for the first step. After each step, the developers and the customer revise the estimates. Either party can walk away if they believe their risk is growing too high; otherwise they repeat the process, signing a new fixed price contract for each successive step.

This approach reduces risk for both parties. It also encourages the developers to control costs, and discourages the customers from trying to add requirements.

Strict hierarchy versus more flexible arrangements

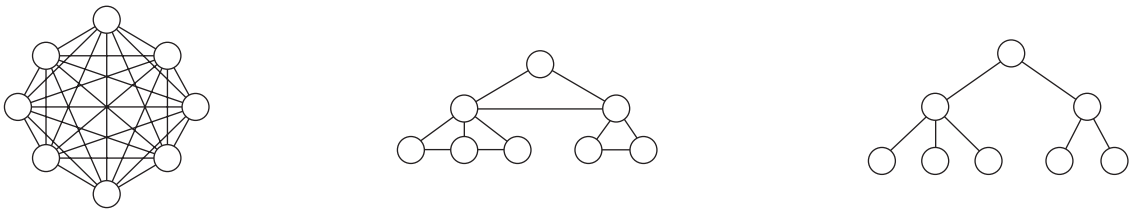
Software engineering teams can be organized in many different ways. One approach is to use a hierarchical manager–subordinate structure. In this approach, each individual reports to a manager and is responsible for performing the tasks delegated by that manager. At the opposite extreme is the egoless team: in such a team everybody is equal, and the team works together to achieve a common goal. In most organizations today, software engineering is performed using teams that fall in the middle of the spectrum.

In the *egoless approach*, decisions are made by consensus; this can lead to more creative solutions, since group members spontaneously get together to solve problems when they arise. In general, the egoless approach is most suited to difficult projects with many technical challenges. However, it is dependent on the personalities of the individuals, and hence is more likely to run into trouble if personal conflicts arise. Also, the assumption that everybody is equal is often wrong when a team is composed of people whose levels of skill and knowledge vary widely.

The *hierarchical approach* is reminiscent of the military, or large bureaucratic organizations. It is suitable for large projects with a strict schedule and where everybody is well trained and has a well-defined role. However, since everybody is responsible only for their own work, problems may go unnoticed.

The '*chief programmer team*' is a model that is midway between egoless and hierarchical. It works very much like the surgical team in an operating room. The chief programmer leads and guides the project; however, he or she consults with, and relies on, individual specialists. As with the egoless approach, everybody's goal is the success of the project.

Figure 11.8 shows how the main channels of communication differ among the team structures. In each case the same eight people are involved. In the idealized egoless approach shown on the left, everybody communicates with everybody else: this can result in a lot of communication, which may be reasonable when difficult technical problems are to be solved. The total number of potential communication channels is $(n^2-n)/2$. The rightmost diagram shows the



(a) Egoless (b) Chief programmer (c) Strict hierachy

Figure 11.8 **Idealized communication patterns in different team structures**

idealized hierarchical approach, where communication links are dramatically reduced (to n–1), but the risk is run that important information will not be communicated.

In reality, whatever team structure is formally adopted, people will tend to consult people with whom they do not have a formal relationship, and may fail to communicate details to people who need to know them. A project manager should therefore monitor and encourage all necessary communication.

Choosing an effective size for a team

We have already discussed the fact that you should divide a system into subsystems. Each subsystem is then assigned to a team; the various teams have to co-ordinate their work.

For a given estimated development effort in person-months there is an optimal team size. Doubling the size of a team will not halve the development time.

Making subsystems, and hence teams, too large or too small tends to make the development more complex and lead to designs that have higher coupling. Subsystems and teams should be sized such that the total amount of required knowledge and exchange of information is reduced – each team needs to understand only the overall software architecture, the details of its own subsystem, plus the interfaces to related subsystems.

For a given project or project iteration, the number of people on a team will not be constant. Initially, just a few people will be involved in defining the scope; later on, additional people will become involved as requirements, design, implementation and testing get under way. As the project or iteration nears completion, people will start moving on to the next iteration or to other work, leaving only a few people to undertake deployment. This change in team size is illustrated in Figure 11.5.

It is important to remember the following rule, however: if your team has an appropriate number of people to start with, then you cannot generally add people if you get behind schedule, in the hope of catching up. In fact, in his book *The Mythical Man Month*, Fred Brooks made one of the most well-known statements in software project management: 'Adding more people to a late project makes it even later.' This occurs because the new people will take time to

learn what has been done, and will require support from the other people in the meantime, slowing them down. Furthermore, as mentioned above, if the team was the 'right' size to start with, adding more people may well be making it too big and inefficient.

Skills needed on a team

No matter how the team is organized, individual people are often assigned specific roles based on their particular skills. The following are some of the more common roles found on a development team:

- **Architect**. This person is responsible for leading the decision making about the architecture, and maintaining the architectural model.
- **Project manager**. Responsible for doing the project management tasks described in this chapter. Even in an egoless team, somebody has to be the custodian of the cost estimates and the schedule.
- **Configuration management and build specialist**. This person ensures that, as changes are made, no new problems are introduced. Everyone relies on builds as the baselines for quality assurance and subsequent development. This person also makes sure that documentation for each change is properly updated.
- **User interface specialist**. Although everybody should interact with users, this person has the particular responsibility to make sure that usability is kept at the forefront of the design process.
- **Technology specialist**. Such a person has specialized knowledge and expertise in a technology such as databases, networking, operating systems etc.
- **Hardware and third-party software specialist**. This person makes sure that the development team has appropriate types of hardware and third-party software on which to develop and test the software. This person will install and perform acceptance testing on any reusable components the team plans to use.
- **User documentation specialist**. This person, who should have a technical writing background, ensures that online help and user manuals are well written.
- **Tester**. Even though there should be an independent test group, the development group may have a person who is responsible for the first stage of testing.

Within any development team, the same individual may have more than one of these roles. The important point, from a project management perspective, is that the needed roles be filled. If not, then some of these tasks may not be properly done. It is also important that there be at least two people capable of performing each role, so that if somebody leaves or is sick, the project is not paralyzed.

The roles listed above require specific skills and knowledge. The project manager can either select people with appropriate background, or ensure that people obtain appropriate training. The project manager should also ensure that all team members continue to augment their education in software engineering, as well as in general skills such as leadership, technical writing, and running meetings.

Exercises

E218 Discuss what you think the best team structure would be for each of the following projects:

(a) The replacement of the income tax system of a country.

(b) The GANA software, whose requirements we introduced in Chapter 4.

(c) The control software for a new interplanetary probe.

(d) Software to provide a more useful front-end to a university registration system, so that students can make better choices of courses.

E219 Design a team for implementing Release 1 of the GANA software whose cost you estimated in Exercise E217. Specify an appropriate team structure, the number of people you believe should be on the team, how the team structure would evolve as the project progresses, and the skills you believe team members should have.

11.5 Project scheduling and tracking

Scheduling is the process of deciding in what sequence a set of activities will be performed, as well as when they should start and be completed. Tracking is the process of determining how well you are sticking to the cost estimate and schedule.

Two types of diagram are particularly important in scheduling: *PERT charts* and *Gantt charts*. *Earned value charts* are useful for tracking. In this section, we will show you how to use all three diagram types. A variety of commercial tools are available to help draw these diagrams.

PERT charts

A PERT chart shows the sequence in which tasks must be completed. Each task has zero or more predecessors on which it depends, and zero or more successors, which depend on it. The whole diagram therefore forms a graph, whose nodes are tasks, and whose arcs are dependencies.

In each node of a PERT chart, you typically show the elapsed time and effort estimates. You can also show optimistic, likely and pessimistic estimates.

PERT

PERT stands for 'Program Evaluation Review Technique' and is very similar to techniques called the 'Critical Path Method' and 'Precedence Networks'.

One of the most important uses of a PERT chart is to determine the *critical path*. The critical path indicates the minimum time in which it is possible to complete the project. It is computed by searching for the path through the chart that has the greatest cumulative elapsed time and no idle time. If any task on the critical path is delayed, then the completion date of the project will be delayed.

Figure 11.9 shows a PERT chart for Release 2 of the GANA system, corresponding to the cost estimates in Example 11.6. Note that some of the tasks have been broken down into subtasks. Each task box shows both the expected elapsed time, in weeks, as well as the expected effort in person-months. The critical path is shown in bold. You can calculate the minimum elapsed time (35 weeks) by summing the elapsed times on this path.

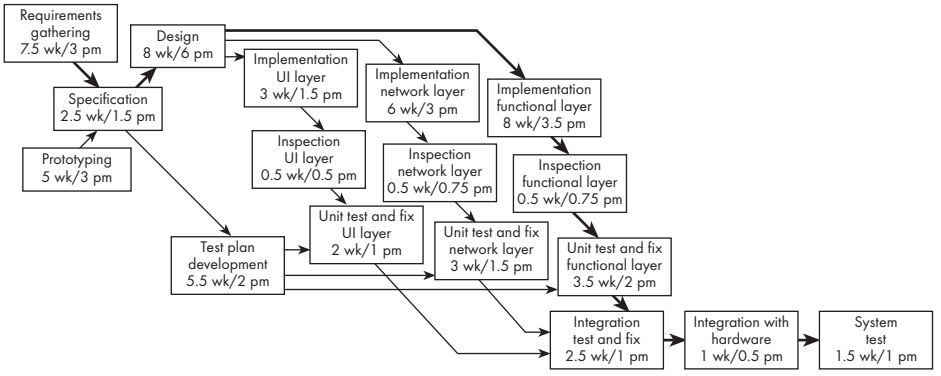


Figure 11.9 **A PERT chart for Release 2 of the GANA system**

Gantt charts

A Gantt chart is used to graphically present the start and end dates of each software engineering task. A Gantt chart is like a UML sequence chart: one axis shows time and the other axis shows the activities that will be performed.

Figure 11.10 shows a Gantt chart for Release 2 of the GANA system. The black bars are the top-level tasks; the white bars are subtasks, and the diamonds are milestones – important deadline dates, at which specific events may occur.

Note how requirements gathering and prototyping overlap each other. Also note that inspection can be started as soon as the first parts of the implementation are complete, and testing can start as soon as some inspection is complete.

This chart does not yet show the allocation of people to the tasks. Nor have we chosen to show the breakdown of implementation and testing by layer; this latter detail can be added once requirements are clearer.

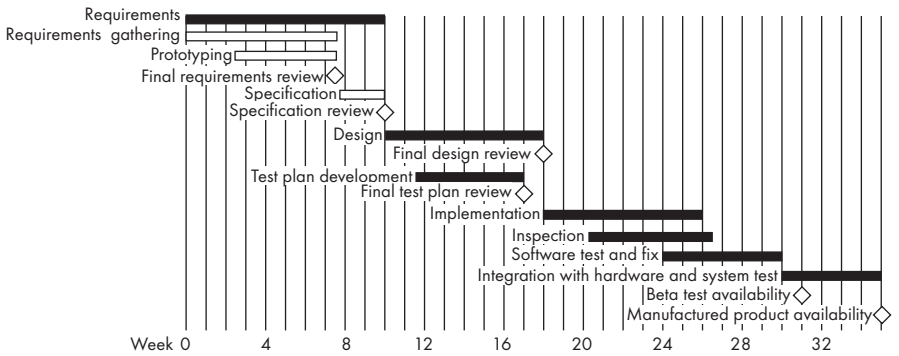


Figure 11.10 **A Gantt chart for Release 2 of the GANA system**

Earned value charts

Earned value is the amount of work completed, measured according to the *budgeted* effort that the work was supposed to consume. It is also called the *budgeted cost of work performed*. As each task is completed, the number of person-months originally planned for that task is added to the earned value of the project.

For example, in the GANA Release 2 project, the total planned effort for all the tasks is 32.5 person-months. Therefore, when the project is complete, its earned value will be 32.5 person-months.

However, tasks may be completed late, and may take more effort than planned. An *earned value chart* such as Figure 11.11 can be used to measure the extent to which the project is behind schedule and over budget.

An earned value chart has three curves:

- **The budgeted cost of work scheduled**. This is the planned amount of effort that was supposed to have been expended by any point in time. It is computed by examining at the cost estimates and the Gantt chart. It is shown here as the solid black curve.
- **The earned value** – that is, the budgeted cost of the work performed. This is shown here as the dotted curve.
- **The actual cost of the work performed so far**. This is shown as a dashed curve.

At any point in the project, you can tell how far behind schedule you are by measuring the horizontal distance between the earned value curve and the budgeted cost of work scheduled. You can also tell the extent to which you are over budget by measuring the vertical distance between the earned value curve and the actual cost curve. Armed with this information, you can take steps to get the project back on track, for example by postponing less important requirements.

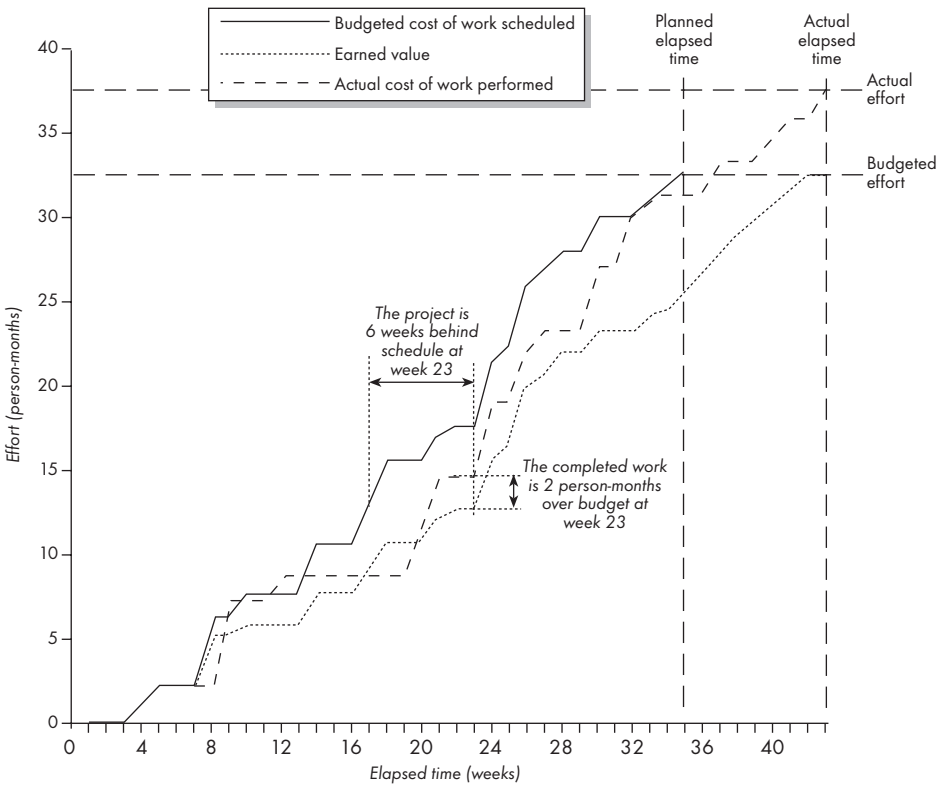


Figure 11.11 **Earned value chart showing how development of Release 2 of the GANA system was 8 weeks behind schedule and cost 10 person-months more than expected**

Exercises

E220 Draw a PERT and a Gantt chart for the project to develop Release 1 of the GANA software, whose cost you estimated in Exercise E217. Make sure you show the critical path on your PERT chart.

E221 In light of the last exercise, review your proposed cost estimate and team structure for the GANA system. Explain any changes you would make.

11.6 Contents of a project plan

You should write a project plan document that has the following types of information. This document should be regularly updated as the project progresses. As with other documents, when writing a project plan, be concise and understand the audience. The audience will be all the team members, the customers, the project managers of related projects, anybody else who has to implement aspects of the plan, and anybody who has to take over as project manager for any reason.

A. **Purpose**. Describe the problem to be solved, as in the requirements document. State the business objectives for the project, including anticipated benefits. Quantify the benefits as much as possible.

B. **Background information**. Give a brief history of the project to date – you can update this as the project proceeds. Describe the stakeholders. Give references to related projects and any documents produced so far, such as requirements definitions.

C. **Processes to be used**. Describe the overall process model (e.g. spiral model, eXtreme Programming, etc.). Outline the techniques to be employed for requirements gathering, design, implementation, quality assurance, change management, risk management, and ongoing project management. Also describe the documents to be produced, including the contents of these documents, and how you will measure and track the project. Some of this information can be given as pointers to standards.

D. **Subsystems and planned releases**. Show the division of the system into subsystems and releases that can be allocated to people or teams. This can be provided as a reference to the architectural model.

E. **Risks and challenges**. Describe the risks and difficulties that are expected to be most critical to this project, or to specific subsystems. Indicate how they are to be monitored and resolved. This will be updated as the project progresses.

F. **Tasks**. List the tasks to be completed for each subsystem and release.

G. **Cost estimates**. Give the present cost estimates for the tasks and subsystems. Show all calculations, and include pessimistic and optimistic values.

H. **Team**. Describe the team structure, the skills needed on that team, the plan for ongoing training and the allocation of general responsibilities to team members.

I. **Schedule and milestones**. Give reasonable deadlines for completion of tasks. Use Gantt and PERT charts showing the allocations of tasks to people, the periods of time they will work on each task, as well as the critical path(s).

11.7 Difficulties and risks in project management

The problems faced by software engineers in general, as discussed in earlier chapters, are also faced by project managers. The following are some of the particular difficulties faced by those given management tasks.

- **Accurately estimating costs is a constant challenge**. As discussed in this chapter, there are many sources of uncertainty that result in inaccurate estimates.
  Resolution. Follow the cost estimation guidelines presented in this chapter.

- **It is very difficult to measure progress and meet deadlines**. Despite your best planning efforts, last-minute problems and inaccurate progress reports can confound attempts to meet deadlines with a high-quality result. Quite often, members of a team have a false sense of optimism, blindly ignore some of the tasks that lie ahead, or are fearful to report potential delays.
  Resolution. Improve your cost estimation skills so as to account for the kinds of problems that may occur. Develop a closer relationship with other members of the team so that you are more keenly aware at all times about the progress achieved, and the potential risks. Be realistic in initial requirements gathering, and follow an iterative approach, since smaller projects are easier to deliver on time. Use earned value charts to monitor progress.

- **It is difficult to deal with lack of human resources or technology that is needed to successfully run a project**. People differ widely in skills, and there may be a shortage of software developers with the skills you need. Much of the technology that we use also has deficiencies.
  Resolution. When determining the requirements and the project plan, take into realistic consideration the resources available. If you cannot find skilled people or suitable technology then you must limit the scope of your project.

- **Communicating effectively in a large project is hard**. Many people lack training in the listening and communications skills needed to exchange information effectively with other stakeholders. This results in errors and delays. Both lack of information and information overload are equally problematic.
  Resolution. Take courses in communication, both written and oral. Learn how to run effective meetings. Review what information everybody should have, and make sure they have it. Make sure that project information is readily available for browsing, e.g. using an intranet web site. Use 'groupware' technology to help specific groups of people exchange the information they need to know.

- **It is hard to obtain agreement and commitment from others**. People have different ideas about the requirements, the design and the project plan. Progress can be paralyzed by indecision. People can be reluctant to commit to plans if they feel they have too much work to do, or if they feel there are not enough benefits accruing from the plan as compared to the costs.
  Resolution. Take courses in negotiating skills and leadership. Ensure that everybody understands the position of everybody else, the costs and benefits of each alternative, and the rationale behind any compromises. Ensure that everybody's proposed responsibility is clearly expressed. Listen to everybody's opinion, but take assertive action, when needed, to ensure progress occurs.

Exercise

E222 Now you have completed this chapter, go back and review your answer to Exercise E213. What have you learned in the chapter that would have enabled

Software patents: a big source of risk

A patent is a license, given by a government, to hold a monopoly on a new invention for a limited time, typically 20 years. Although laws differ from country to country, you can generally obtain a patent if you can show that your invention is new, useful and non-obvious.

In the past, patents have been very important, ensuring that those who invest effort in developing an innovation will not be immediately undercut in price by people who merely copy their idea. Within the last 20 years, however, patents on *software* have become available in many jurisdictions. The majority of software engineers see this as a bad thing, although many patent holders will obviously hold the opposite opinion.

Software patents certainly are a big source of risk for developers. In the early days of technology, if you marketed an innovation it tended to represent a *single* invention, or perhaps a few. You could therefore reasonably easily tell whether you were infringing on someone's patent. Developing software, however, involves building a system by composing vast numbers of ideas; therefore you are quite likely to accidentally infringe on patents and hence risk being sued by patent holders.

Many people feel software patents should be disallowed since research has shown they probably have the opposite effect from what was intended. They most likely stifle invention – they consume time and money as people defend themselves against claims, or file their own patents so that they can counter-claim against others. They also result in people having to be very cautious in their development and therefore avoid innovation.

To make matters worse, it seems that many patents that have been issued have in fact been neither new nor non-obvious. However, patent examiners have often lacked the time or the sources of information to determine this. Although a bad patent can be overturned in court, doing so is very expensive. This means that if you are sued for infringement of a patent that you know should be invalid, it may still put you out of business.

you to do a better job, if you could go back in time and repeat the group work you did in this book?

11.8 Summary

In this chapter we have briefly introduced some important techniques required to manage a software project.

We started by discussing process models. Although the waterfall model is the classic model, it is recognized today that a more iterative approach is generally superior, and an agile approach may be ideal for a smaller project.

Next we discussed cost estimation, an activity that all software engineers need to perform to some extent, but which is particularly difficult. We outlined a series of principles you can apply, including: dividing and conquering, making sure you account for all activities, basing your estimates on past experience, combining estimates from different people, making pessimistic and optimistic estimates, and regularly revising estimates.

Then we discussed structuring teams. Members of a rigid, hierarchical team will consume less time communicating with each other, but may not exchange essential information. More flexible team structures such as chief programmer or egoless teams work best, especially for smaller and high-risk projects.

We discussed using Gantt, PERT and Earned Value charts for scheduling and tracking projects. Gantt charts show a series of timelines for each task, whereas PERT charts show the interrelationships among tasks, including the critical path. Earned value charts help you determine the extent to which the project is behind schedule and over budget.

All of the information discussed above should be summarized in a project plan that team members can read.

11.9 For more information

The following are some important resources related to project management:

- B. Hughes and M. Cotterell, *Software Project Management*, 3rd edition, McGraw-Hill, 2002. http://www.mcgraw-hill.co.uk/hughes
- S. McConnell, *Software Project Survival Guide*, Microsoft Press, 1997, http://www.stevemcconnell.com/sg.htm
- R. L. Glass, *Software Runaways*, Prentice Hall, 1998
- F. Brooks, *The Mythical Man-Month*, 20th Anniversary Edition, Addison-Wesley, 1995
- R. M. Belbin, *Beyond the Team*, Butterworth-Heinemann, 2000, http://www.belbin.com/book-btt.html
- A good website for COCOMO: http://sunset.usc.edu/research/COCOMOII/
- Dave Farthing's software project management page at the University of Glamorgan: http://www.comp.glam.ac.uk/pages/staff/dwfarthi/projman.htm
- The Extreme Programming web site: http://www.extremeprogramming.org. This web site is a particularly good project management resource, whether or not you are currently thinking of doing eXtreme Programming
- M. Stephens and D. Rosenberg, *Extreme Programming Refactored: The Case Against XP*, Apress, 2003. http://www.softwarereality.com/lifecycle/xp/case_against_xp.jsp
- All project management: http://www.allpm.com
- The project management institute http://www.pmi.org. They have also published their Project Management Body of Knowledge in book form as *A Guide to the Project Management Body of Knowledge*

Standards

The following are some of the IEEE, British and ISO standards covering project management. As mentioned in Chapter 4, standards require a subscription. For ISO standards, see http://www.iso.ch. For IEEE standards, see: http://www.standards.ieee.org/software/index.html. For British standards, see: http://bsonline.techindex.co.uk.

- IEEE Standard 828, *Software Configuration Management Plans*
- IEEE Standard 1219, *Software Maintenance*
- IEEE Standard 1490, *Adoption of the Project Management Institute's Project Management Body of Knowledge*
- ISO Standard/British Standard/ IEEE Standard 12207, *Software Lifecycle Processes*
- ISO Standard/British Standard 15504, *Software Process Assessment*

Project exercises

E223 Look back over all the work you and your group have performed on project work from this book, and write a report about the results. Include the following information:

(a) How much effort you spent in total, for all aspects of the work.

(b) How much you produced, measured in use cases implemented, test cases written, and lines of code.

(c) Your productivity: how much you produced, per person-hour.

(d) Sources of inaccuracy in your estimation of the above.

(e) The technological problems you encountered.

(f) The project management problems you encountered.

E224 Compare your results of the previous exercise with the results of other groups. Explain any differences.

E225 Develop an outline of requirements and a project plan for the following problem. Imagine you were asked to extend SimpleChat Phase 5 so that it can be used to transmit low-resolution, low frame-rate video images of the people participating in a channel. The objective is to make SimpleChat work like video-conferencing software. Include in your project plan the information suggested in Section 11.6. Use data from the last exercise to help calibrate your cost estimates, and use OLP and Delphi estimating techniques.