Chapter 9

Planning and Estimating

**Learning Objectives**

After studying this chapter, you should be able to

* Explain the importance of planning.
* Estimate the size and cost of building a software product.
* Appreciate the importance of updating and tracking estimates.
* Draw up a project management plan that conforms to the IEEE standard

The challenges of constructing a software product have no easy solution. To put together a large software product takes time and resources. And, like any other large construction project, careful planning at the beginning of the project perhaps is the single most important factor that distinguishes success from failure. This initial planning, however, by no means is enough. Planning, like testing, must continue throughout the software development and maintenance process. Notwithstanding the need for continual planning, these activities reach a peak after the specifications have been drawn up but before design activities commence. At this point in the process, meaningful duration and cost estimates are computed and a detailed plan for completing the project produced.

I n this chapter, we distinguish these two types of **planning**, the planning that proceeds throughout the project and the intense planning that must be carried out once the specifications are complete.

# 9.1 Planning and the Software Process

Ideally, we would like to plan the entire software project at the very beginning of the process, and then follow that plan until the target software finally has been delivered to the client. This is impossible, however, because we lack enough information during the initial

## 268

**FIGURE 9.1** A model for estimating the relative range of a cost estimate for each life-cycle workflow.

workflows to be able to draw up a meaningful plan for the complete project. For example, during the requirements workflow, any sort of planning (other than just for the requirements workflow itself) is futile.

There is a world of difference between the information at the developers’ disposal at the end of the requirements workflow and at the end of the analysis workflow, analogous to the difference between a rough sketch and a detailed blueprint. By the end of the requirements workflow, the developers at best have an informal understanding of what the client needs. In contrast, by the end of the analysis workflow, at which time the client signs a document stating precisely what is going to be built, the developers have a detailed appreciation of most (but usually still not all) aspects of the target product. This is the earliest point in the process at which accurate duration and cost estimates can be determined.

Nevertheless, in some situations, an organization may be required to produce duration and cost estimates before the specifications can be drawn up. In the worst case a client may insist on a bid on the basis of an hour or two of preliminary discussion. F figure 9.1 shows how problematic this can be. Based on a model in [Boehm et al., 2000], it depicts the relative range of cost estimates for the various workflows of the life cycle. For example, suppose that, when a product passes its acceptance test at the end of the implementation workflow and is delivered to the client, its cost is found to be $1 million. If a cost estimate had been made midway through the requirements workflow, it is likely that it would have been somewhere in the range ($0.25 million, $4 million), as shown in Figure 9.2. Similarly, if the cost estimate had been made midway through the analysis workflow, the range of likely estimates would have shrunk to ($0.5 million, $2 million). Furthermore, if the cost estimate had been made at the end of the analysis workflow, that is, at the appropriate time, the result probably would have been in the still relatively wide range of ($0.67 million, $1.5 million). All four points are marked on the upper and lower bound lines in Figure 9.2, which has a logarithmic scale on the vertical axis. This model is called the **cone of uncertainty**. It is

Relative range of cost estimate

4

3

2

1

Requirements Analysis Design Implementation

Workflow during which the cost estimate is made

Requirements Analysis Design Implementation

Cost estimate (in millions of dollars)

Workflow during which the cost estimate is made

4.00

3.00

2.00

0

0.25

0.67

0.50

1.00

1.50

Upper bound

Lower bound

**FIGURE 9.2**

Range of cost

estimates for

a softw

are

product that cost

$1 million to

build.

clear from Figures 9.1 and 9 .2 that cost estimation is not an exact science; reasons for this are given in Section 9.2.

The data on which the cone of uncertainty model is based are old, including five proposals submitted to the U.S. Air Force Electronic Systems Division [Devenny, 1976], and estimation techniques have improved since that time. Nevertheless, the overall shape of the curve in Figure 9.1 probably has not changed overmuch. Consequently, a premature duration or cost estimate, that is, an estimate made before the specifications have been signed off on by the client, is likely to be considerably less accurate than an estimate made when sufficient data have accumulated.

We now examine techniques for estimating duration and cost. The assumption throughout the remainder of this chapter is that the analysis workflow has been completed; that is, meaningful estimating and planning now can be carried out.

# 9.2 Estimating Duration and Cost

The budget is an integral part of any software project management plan. Before design commences, the client needs to know how much he or she will have to pay for the product. If the development team underestimates the actual cost, the development organization can lose money on the project. On the other hand, if the development team overestimates, then the client may decide that, on the basis of cost–benefit analysis or return on investment, there is no point in having the product built. Alternatively, the client may give the job to another development organization whose estimate is more reasonable. Either way, it is clear that accurate cost estimation is critical.

In fact, two types of costs are associated with software development. The first is the **internal cost**, the **cost** to the developers; the second is the **external cost**, the **price** that the client will pay. The internal cost includes the salaries of the development teams, managers, and support personnel involved in the project; the cost of the hardware and software for developing the product; and the cost of overhead such as rent, utilities, and salaries of senior management. Although the price generally is based on the cost plus a profit margin, in some cases economic and psychological factors are important. For example, developers who desperately need the work may be prepared to charge the client at cost. A different situation arises when a contract is to be awarded on the basis of bids. The client may reject a bid that is significantly lower than all the other bids on the grounds that the quality of the resulting product probably also would be significantly lower. A development team therefore may try to come up with a bid that will be slightly, but not significantly, lower than what it believes will be the competitors’ bids.

Another important part of any plan is estimating the duration of the project. The client certainly wants to know when the finished product will be delivered. If the development organization is unable to keep to its schedule, then at best the organization loses credibility, at worst penalty clauses are invoked. In all cases, the managers responsible for the software project management plan have a lot of explaining to do. Conversely, if the development organization overestimates the time needed to build the product, then there is a good chance that the client will go elsewhere.

Unfortunately, it is by no means easy to obtain an accurate **cost estimate** and **duration estimate**. Too many variables are involved to be able to get an accurate handle on either cost or duration. One big difficulty is the human factor. Over 40 years ago, Sackman and coworkers observed differences of up to 28 to 1 between pairs of programmers [Sackman, Erikson, and Grant, 1968]. It is easy to try to brush off their results by saying that experienced programmers always outperform beginners, but Sackman and his colleagues compared matched pairs of programmers. They observed, for example, two programmers with 10 years of experience on similar types of projects and measured the time it took them to perform tasks like coding and debugging. Then they observed, say, two beginners who had been in the profession for the same short length of time and had similar educational backgrounds. Comparing worst and best performances, they observed differences of 6 to 1 in product size, 8 to 1 in product execution time, 9 to 1 in development time, 18 to 1 in coding time, and 28 to 1 in debugging time. A particularly alarming observation is that the best and worst performances on one product were by two programmers, each of whom had 11 years of experience. Even when the best and worst cases were removed from Sackman et al.’s sample, observed differences were still on the order of 5 to 1. On the basis of these results, clearly, we cannot hope to estimate software cost or duration with any degree of accuracy (unless we have detailed information regarding all the skills of all the employees, which would be most unusual). It has been argued that, on a large project, differences among individuals tend to cancel out, but this perhaps is wishful thinking; the presence of one or two very good (or very bad) team members can cause marked deviations from schedules and significantly affect the budget. Another human factor that can affect estimation is that, in a free country, there is no way of ensuring that a critical staff member will not resign during the project. Time and money then are spent attempting to fill the vacated position and integrate the replacement into the team, or in reorganizing the remaining team members to compensate for the loss. Either way, schedules slip and estimates come unstuck.

Underlying the cost estimation problem is another issue: How is the size of a product to be measured?

## 9.2.1 Metrics for the Size of a Product

The most common metric for the size of a product is the number of lines of code. Two units commonly are used: **lines of code (LOC)** and **thousand delivered source instructions (KDSI)**. Many problems are associated with the use of lines of code [van der Poel and Schach, 1983].

* Creation of source code is only a small part of the total software development effort. It seems somewhat far-fetched that the time required for the requirements, analysis, design, implementation, and testing workflows (which include planning and documentation activities) can be expressed solely as a function of the number of lines of code in the final product.
* Implementing the same product in two different languages results in versions with different numbers of lines of code. Also, with languages such as Lisp or with many nonprocedural 4GLs (Section 15.2), the concept of a line of code is not defined.
* It often is unclear exactly how to count lines of code. Should only executable lines of code be counted or data definitions as well? And should comments be counted? If not, there is a danger that programmers will be reluctant to spend time on what they perceive to be “nonproductive” comments, but if comments are counted, then the opposite danger is that programmers will write reams of comments in an attempt to boost their apparent productivity. Also, what about counting job control language statements? Another problem is how changed lines or deleted lines are counted—in the course of enhancing a product to improve its performance, sometimes the number of lines of code is decreased. Reuse of code (Section 8.1) also complicates line counting: If reused code is modified, how is it counted? And, what if code is inherited from a parent class (Section 7.8)? In short, the apparently straightforward metric of lines of code is anything but straightforward to count.
* Not all the code implemented is delivered to the client. It is not uncommon for half the code to consist of tools needed to support the development effort.
* Suppose that a software developer uses a code generator, such as a report generator, a screen generator, or a graphical user interface (GUI) generator. After a few minutes of design activity on the part of the developer, the tool may generate many thousands of lines of code.
* The number of lines of code in the final product can be determined only when the product is completely finished. Therefore, basing cost estimation on lines of code is doubly dangerous. To start the estimation process, the number of lines of code in the finished product must be estimated. Then, this estimate is used to estimate the cost of the product. Not only is there uncertainty in every costing technique, but if the input to an uncertain cost estimator itself is uncertain (that is, the number of lines of code in a product that has not yet been built), then the reliability of the resulting cost estimate is unlikely to be very high.

Because the number of lines of code is so unreliable, other metrics must be considered. An alternative approach to estimating the size of a product is the use of metrics based on

measurable quantities that can be determined early in the software process. For example, van der Poel and Schach [1983] put forward the **FFP metric** for cost estimation of medium- scale data-processing products. The three basic structural elements of a data-processing product are its fi les, flows, and processes; the name FFP is an acronym formed from the initial letters of those elements. A *fi le* is defined as a collection of logically or physically related records permanently resident in the product; transaction and temporary fi les are excluded. A *flow* is a data interface between the product and the environment, such as a screen or a report. A *process* is a functionally defined logical or arithmetic manipulation of data; examples include sorting, validating, or updating. Given the number of fi les *Fi*, flows *Fl*, and processes *Pr* in a product, its size *S* and cost *C* are given by

* 1. *Fi*  *Fl*  *Pr* **(9.1)**  *C d*  *S*  **(9.2)**

where *d* is a constant that varies from organization to organization. Constant *d* is a measure of the **efficiency** (**productivity**) of the software development process within that organization. The size of a product simply is the sum of the number of fi les, flows, and processes, a quantity that can be determined once the architectural design is complete. The cost then is proportional to the size, the constant of proportionality *d* being determined by a least-squares fi t to cost data relating to products previously developed by that organization. Unlike metrics based on the number of lines of code, the cost can be estimated before coding begins.

* 1. he validity and reliability of the FFP metric were demonstrated using a purposive sample that covered a range of medium-scale data-processing applications. Unfortunately, the metric was never extended to include databases, an essential component of many data-processing products.

A similar, but independently developed, metric for the size of a product was developed by Albrecht [1979] based on function points; Albrecht’s metric is based on the number of input items *Inp*, output items *Out*, inquiries *Inq*, master fi les *Maf*, and interfaces *Inf*. In its simplest form the number of **function points** *FP* is given by the equation

*FP*  4 *Inp*  5 *Out*  4 *Inq*  10 *Maf*  7 *Inf*  **(9.3)**

Because this is a measure of the product’s size, it can be used for cost estimation and productivity estimation.

Equation (9.3) is an oversimplification of a three-step calculation. First, the unadjusted function points are computed:

1. Each of the components of a product—*Inp*, *Out*, *Inq*, *Maf*, and *Inf—*must be classified as simple, average, or complex (see Figure 9.3).
2. Each component is assigned a number of function points depending on its level. For example, an average input is assigned four function points, as reflected in equation (9.3), but a simple input is assigned only three, whereas a complex input is assigned six function points. The data needed for this step appear in Figure 9.3.
3. The function points assigned to each component are then summed, yielding the **unadjusted function points (*UFP*)**.

**FIGURE 9.3**

**L**

**evel of Complexity**

**Component**

**S**

**imple**

**Average**

**C**

**omplex**

Input item

4

3

6

Output item

4

5

7

Inquiry

3

4

6

Master fi le

7

1

0

15

Interface

5

1

0

7

Table of function point values.

**FIGURE 9.4**

Technical 1. Data communication

factors for 2. Distributed data processing function point

3.Performance criteria computation.

4.Heavily utilized hardware

5.High transaction rates

6.Online data entry

7.End-user efficiency

8.Online updating

9.Complex computations

10.Reusability

11.Ease of installation

12.Ease of operation

13.Portability

14. Maintainability

Second, the **technical complexity factor (*TCF*)** is computed. This is a measure of the effect of 14 technical factors, such as high transaction rates, performance criteria (for example, throughput or response time), and online updating; the complete set of factors is shown in Figure 9.4. Each of these 14 factors is assigned a value from 0 (“not present or no influence”) to 5 (“strong influence throughout”). The resulting 14 numbers are summed, yielding the total degree of influence (*DI*). The *TCF* is then given by

*TCF*  0.65 0.01 *DI* **(9.4)**

Because *DI* can vary from 0 to 70, *TCF* varies from 0.65 to 1.35. Third, *FP*, the number of function points, is given by

*FP*  *UFP*  *TCF* **(9.5)**

Experiments to measure software productivity rates have shown a better fit using function points than using KDSI. For example, Jones [1987] has stated that he observed errors

|  |
| --- |
| **Assembler Version Ada Version**  Source code size 70 KDSI 25 KDSI  Development costs $1,043,000 $590,000  KDSI per person-month 0.335 0.211  Cost per source statement $14.90 $23.60  Function points per person-month 1.65 2.92  Cost per function point $3,023 $1,170 |

**FIGURE 9.5** A comparison of assembler and Ada products [Jones, 1987].

(© 1987 IEEE.)

in excess of 800 percent counting KDSI, but *only* [emphasis added] 200 percent in counting function points, a most revealing remark.

To show the superiority of function points over lines of code, Jones [1987] cites the example shown in Figure 9.5. The same product was coded both in assembler and in Ada and the results compared. First, consider KDSI per person-month. This metric tells us that coding in assembler is apparently 60 percent more efficient than coding in Ada, which is patently false. Third-generation languages like Ada have superseded assembler simply because it is much more efficient to code in a third-generation language. Now consider the second metric, cost per source statement. Note that one Ada statement in this product is equivalent to 2.8 assembler statements. Use of cost per source statement as a measure of efficiency again implies that it is more efficient to code in assembler than in Ada. However, when function points per person-month is taken as the metric of programming efficiency, the superiority of Ada over assembler is reflected clearly.

On the other hand, both function points and the FFP metric of equations (9.1) and (9.2) suffer from the same weakness: Product maintenance often is inaccurately measured. When a product is maintained, major changes to the product can be made without changing the number of fi les, flows, and processes or the number of inputs, outputs, inquiries, master fi les, and interfaces. Lines of code is no better in this respect. To take an extreme case, it is possible to replace every line of a product with a completely different line without changing the total number of lines of code.

At least 40 variants of and extensions to Albrecht’s function points have been proposed [Maxwell and Forselius, 2000]. Mk II function points were put forward by Symons [1991] to provide a more accurate way of computing the unadjusted function points (*UFP*). The software is decomposed into a set of component transactions, each consisting of an input, a process, and an output. The value of *UFP* then is computed from these inputs, processes, and outputs. Mk II function points are widely used all over the world.

## 9.2.2 Techniques of Cost Estimation

Notwithstanding the difficulties with estimating size, it is essential that software developers simply do the best they can to obtain accurate estimates of both project duration and project cost, while taking into account as many as possible of the factors that can affect their estimates. These include the skill levels of the personnel, the complexity of the project, the size of the project (cost increases with size but much more than linearly), familiarity of the development team with the application area, the hardware on which the product is to be

run, and availability of CASE tools. Another factor is the deadline effect. If a project has to be completed by a certain time, the effort in person-months is greater than if no constraint is placed on completion time; hence, the greater the cost. This shows that duration and cost are not independent; the shorter the deadline, the greater the effort and, hence, the greater the cost.

From the preceding list, which is by no means comprehensive, clearly estimation is a difficult problem. A number of approaches have been used, with greater or lesser success.

### 1. Expert Judgment by Analogy

In the **expert judgment by analogy** technique, a number of experts are consulted. An expert arrives at an estimate by comparing the target product to completed products with which the expert was actively involved and noting the similarities and differences. For example, an expert may compare the target product to a similar product developed 2 years ago for which the data were entered in batch mode, whereas the target product is to have online data capture. Because the organization is familiar with the type of product to be developed, the expert reduces development time and effort by 15 percent. However, the graphical user interface is somewhat complex; this increases time and effort by 25 percent. Finally, the target product has to be developed in a language with which most of the team members are unfamiliar, thereby increasing time by 15 percent and effort by 20 percent. Combining these three figures, the expert decides that the target product will take 25 percent more time and 30 percent more effort than the previous one. Because the previous product took 12 months to complete and required 100 person-months, the target product is estimated to take 15 months and consume 130 person-months.

T wo other experts within the organization compare the same two products. One concludes that the target product will take 13.5 months and 140 person-months. The other comes up with the figures of 16 months and 95 person-months. How can the predictions of these three experts be reconciled? One technique is the **Delphi technique**: It allows experts to arrive at a consensus without having group meetings, which can have the undesirable side effect of one persuasive member swaying the group. In this technique, the experts work independently. Each produces an estimate and a rationale for that estimate. These estimates and rationales then are distributed to all the experts, who now produce a second estimate. This process of estimation and distribution continues until the experts can agree within an accepted tolerance. No group meetings take place during the iteration process.

Valuation of real estate frequently is done on the basis of expert judgment by analogy. An appraiser arrives at a valuation by comparing a house with similar houses that have been sold recently. Suppose that house A is to be valued, house B next door has just sold for $205,000, and house C on the next street sold 3 months ago for $218,000. The appraiser may reason as follows: House A has one more bathroom than house B, and the yard is 5000 square feet larger. House C is approximately the same size as house A, but its roof is in poor condition. On the other hand, house C has a Jacuzzi. After careful thought, the appraiser may arrive at a figure of $215,000 for house A.

In the case of software products, expert judgment by analogy is less accurate than real estate valuation. Recall that our first software expert claimed that using an unfamiliar language would increase time by 15 percent and effort by 20 percent. Unless the expert has

some validated data from which the effect of each difference can be determined (a highly unlikely possibility), errors induced by what can be described only as guesses will result in hopelessly incorrect cost estimates. In addition, unless the experts are blessed with total recall (or have kept detailed records), their recollections of completed products may be sufficiently inaccurate as to invalidate their predictions. Finally, experts are human and, therefore, have biases that may affect their predictions. At the same time, the results of estimation by a group of experts should reflect their collective experience; if this is broad enough, the result well may be accurate.

### 2. Bottom-Up Approach

One way of trying to reduce the errors resulting from evaluating a product as a whole is to break the product into smaller components. Estimates of duration and cost are made for each component separately and combined to provide an overall figure. This **bottom-up approach** has the advantage that estimating costs for several smaller components generally is quicker and more accurate than for one large one. In addition, the estimation process is likely to be more detailed than with one large, monolithic product. The weakness of this approach is that a product is more than the sum of its components.

With the object-oriented paradigm, the independence of the various classes helps the bottom-up approach. However, interactions among the various objects in the product complicate the estimation process.

### 3. Algorithmic Cost Estimation Models

I n this approach, a metric, such as function points or the FFP metric, is used as input to a model for determining product cost. The estimator computes the value of the metric; duration and cost estimates then can be computed using the model. On the surface, an **algorithmic cost estimation model** is superior to expert opinion, because a human expert, as pointed out previously, is subject to biases and may overlook certain aspects of both the completed and target products. In contrast, an algorithmic cost estimation model is unbiased; every product is treated the same way. The danger with such a model is that its estimates are only as good as the underlying assumptions. For example, underlying the function point model is the assumption that every aspect of a product is embodied in the five quantities on the right-hand side of equation (9.3) and the 14 technical factors. A further problem is that a significant amount of subjective judgment often is needed in deciding what values to assign to the parameters of the model. For example, frequently it is unclear whether a specific technical factor of the function point model should be rated a 3 or a 4.

Many algorithmic cost estimation models have been proposed. Some are based on mathematical theories as to how software is developed. Other models are statistically based; large numbers of projects are studied and empirical rules determined from the data. Hybrid models incorporate mathematical equations, statistical modeling, and expert judgment. The most important hybrid model is Boehm’s COCOMO, which is described in detail in Section 9.2.3. (See Just in Case You Wanted to Know Box 9.1 for a discussion of the acronym COCOMO.)

# Just in Case You Wanted to Know Box 9.1

COCOMO is an acronym formed from the first two letters of each word in COnstructive COst MOdel. Any connection with Kokomo, Indiana, is purely coincidental.

The *MO* in COCOMO stands for “model,” so the phrase *COCOMO model* should not be used. That phrase falls into the same category as “ATM machine” and “PIN number,” both of which were dreamed up by the Department of Redundant Information Department.



## 9.2.3 Intermediate COCOMO

**COCOMO** actually is a series of three models, ranging from a macro estimation model that treats the product as a whole to a micro estimation model that treats the product in detail. In this section, a description is given of intermediate COCOMO, which has a middle level of complexity and detail. COCOMO is described in detail in [Boehm, 1981]; an overview is presented in [Boehm, 1984].

Computing development time using intermediate COCOMO is done in two stages. First, a rough estimate of the development effort is provided. Two parameters have to be estimated: the length of the product in KDSI and the product’s development mode, a measure of the intrinsic level of difficulty of developing that product. There are three modes: *organic* (small and straightforward), *semidetached* (medium sized), and *embedded* (complex).

From these two parameters, the **nominal effort** can be computed. For example, if the project is judged to be essentially straightforward (organic), then the nominal effort (in person-months) is given by the equation

Nominal effort 3.2 (KDSI) 1.05  person-months **(9.6)**

The constants 3.2 and 1.05 are the values that best fitted the data on the organic mode products used by Boehm to develop intermediate COCOMO.

F or example, if the product to be built is organic and estimated to be 12,000 delivered source statements (12 KDSI), then the nominal effort is

3.2 (12) 1.05  43 person-months

(but read Just in Case You Wanted to Know Box 9.2 for a comment on this value).

Next, this nominal value must be multiplied by 15 **software development effort multipliers**. These multipliers and their values are given in Figure 9.6. Each multiplier can have up to six values. For example, the product complexity multiplier is assigned the values 0.70, 0.85, 1.00, 1.15, 1.30, or 1.65, according to whether the developers rate the project complexity as very low, low, nominal (average), high, very high, or extra high. As can be seen from Figure 9.6, all 15 multipliers take on the value 1.00 when the corresponding parameter is nominal.

Boehm provides guidelines to help the developer determine whether the parameter should indeed be rated nominal or whether the rating is lower or higher. For example, consider again the module complexity multiplier. If the control operations of the module essentially consist of a sequence of the constructs of structured programming (such as **if-then-else**, **do-while**, **case**), then the complexity is rated *very low*. If these operators are nested, then the rating is *low*. Adding intermodule control and decision tables increases the rating to *nominal*. If the operators are highly nested, with compound predicates, and queues and stacks, then the rating is *high.* The presence of reentrant and recursive coding and

# Just in Case You Wanted to Know Box 9.2

One reaction to the value of the nominal effort might be, “If 43 person-months of effort are needed to produce 12,000 delivered source instructions, then on average each programmer is turning out fewer than 300 lines of code a month—I have implemented more than that in one night!”



A 300-line product usually is just that: 300 lines of code. In contrast, a maintainable 12,000-line product has to go through all the workflows of the life cycle. In other words, the total effort of 43 person-months is shared among many activities, including coding.

**FIGURE 9.6** Intermediate COCOMO software development effort multipliers [Boehm, 1984]. (© 1984 IEEE)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Rating**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Cost Drivers** | **Very Low** | **Low** | **Nominal** | **High** | **Very High** | **Extra High** | | Product Attributes  Required software reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |  | | Database size |  | 0.94 | 1.00 | 1.08 | 1.16 |  | | Product complexity | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 | | Computer Attributes  Execution time constraint |  |  | 1.00 | 1.11 | 1.30 | 1.66 | | Main storage constraint |  |  | 1.00 | 1.06 | 1.21 | 1.56 | | Virtual machine volatility\* |  | 0.87 | 1.00 | 1.15 | 1.30 |  | | Computer turnaround time |  | 0.87 | 1.00 | 1.07 | 1.15 |  | | Personnel Attributes  Analyst capabilities | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |  | | Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |  | | Programmer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |  | | Virtual machine experience\* | 1.21 | 1.10 | 1.00 | 0.90 |  |  | | Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |  | | Project Attributes  Use of modern programming practices | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |  | | Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |  | | Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |  |   \*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task. |

fixed-priority interrupt handling pushes the rating to *very high*. Finally, multiple resource scheduling with dynamically changing priorities and microcode-level control ensures that the rating is *extra high.* These ratings apply to control operations. A module also has to be evaluated from the viewpoint of computational operations, device-dependent operations, and data management operations. For details on the criteria for computing each of the 15 multipliers, refer to [Boehm, 1981].

To see how this works, Boehm [1984] gives the example of microprocessor-based communications processing software for a highly reliable new electronic funds transfer network, with performance, development schedule, and interface requirements. This product fits the description of embedded mode and is estimated to be 10,000 delivered source instructions (10 KDSI) in length, so the nominal development effort is given by

Nominal effort 2.8 (KDSI) 1.20 **(9.7)**

|  |
| --- |
| **Effort**  **Cost Drivers Situation Rating Multiplier**  Required software reliability Serious financial consequences High 1.15 of software fault  Database size 20,000 bytes Low 0.94  Product complexity Communications processing Very high 1.30  Execution time constraint Will use 70% of available time High 1.11  Main storage constraint 45K of 64K store (70%) High 1.06  Virtual machine volatility Based on commercial Nominal 1.00  microprocessor hardware  Computer turnaround time 2-hour average turnaround time Nominal 1.00  Analyst capabilities Good senior analysts High 0.86  Applications experience 3 years Nominal 1.00  Programmer capability Good senior programmers High 0.86  Virtual machine experience 6 months Low 1.10  Programming language experience 12 months Nominal 1.00  Use of modern programming Most techniques in use over High 0.91 practices 1 year  Use of software tools at basic minicomputer Low 1.10  tool level  Required development schedule 9 months Nominal 1.00 |

**FIGURE 9.7**

Intermediate

COCOMO

effort multiplier ratings for microprocessor communications software [Boehm, 1984].

(© 1984 IEEE)

(Again, the constants 2.8 and 1.20 are the values that best fitted the data on embedded products.) Because the project is estimated to be 10 KDSI in length, the nominal effort is

2.8 (10) 1.20 44 person-months

The estimated development effort is obtained by multiplying the nominal effort by the 15 software development effort multipliers. The ratings of these multipliers and their values are given in Figure 9.7. Using these values, the product of the multipliers is found to be

1.35, so the estimated effort for the project is

1.35 44 59 person-months

This number is then used in additional formulas to determine dollar costs, development schedules, phase and activity distributions, computer costs, annual maintenance costs, and other related items; for details, see [Boehm, 1981]. Intermediate COCOMO is a complete algorithmic cost estimation model, giving the user virtually every conceivable assistance in project planning.

Intermediate COCOMO has been validated with respect to a broad sample of 63 projects covering a wide variety of application areas. The results of applying intermediate COCOMO to this sample are that the actual values come within 20 percent of the predicted values about 68 percent of the time. Attempts to improve on this accuracy make little sense because in most organizations, the input data for intermediate COCOMO generally are accurate to within only about 20 percent. Nevertheless, the accuracy obtained by experienced estimators placed intermediate COCOMO at the cutting edge of cost estimation research during the 1980s; no other technique was consistently as accurate.

The major problem with intermediate COCOMO is that its most important input is the number of lines of code in the target product. If this estimate is incorrect, then every single prediction of the model may be incorrect. Because of the possibility that the predictions of intermediate COCOMO or any other estimation technique may be inaccurate, management must monitor all predictions throughout software development.

## 9.2.4 COCOMO II

C OCOMO was put forward in 1981. At that time, the only life-cycle model in use was the waterfall model. Most software was run on mainframes. Technologies such as client–server and object orientation essentially were unknown. Accordingly, COCOMO did not incorporate any of these factors. However, as newer technologies began to become accepted software engineering practice, COCOMO started to become less accurate.

**COCOMO II** [Boehm et al., 2000] is a major revision of the 1981 COCOMO.

COCOMO II can handle a wide variety of modern software engineering techniques, including object-orientation, the various life-cycle models described in Chapter 2, rapid prototyping (Section 11.13), fourth-generation languages (Section 15.2), reuse (Section 8.1), and COTS software (Section 1.11). COCOMO II is both flexible and sophisticated. Unfortunately, to achieve this goal, COCOMO II is considerably more complex than the original COCOMO. Accordingly, the reader who wishes to utilize COCOMO II should study [Boehm et al., 2000] in detail; only an overview of the major differences between COCOMO II and intermediate COCOMO is given here.

First, intermediate COCOMO consists of one overall model based on lines of code (KDSI). On the other hand, COCOMO II consists of three different models. The **application composition model**, based on object points (similar to function points), is applied at the earliest workflows, when minimal knowledge is available regarding the product to be built. Then, as more knowledge becomes available, the **early design model** is used; this model is based on function points. Finally, when the developers have maximal information, the **post architecture model** is used. This model uses function points or lines of code (KDSI). The output from intermediate COCOMO is a cost and size estimate; the output from each of the three models of COCOMO II is a range of cost and size estimates. Accordingly, if the most likely estimate of the effort is *E*, then the application composition model returns the range (0.50*E*, 2.0*E*), and the post architecture model returns the range (0.80*E*, 1.25*E*). This reflects the increasing accuracy of the progression of models of COCOMO II. A second difference lies in the effort model underlying COCOMO:

Effort a(size)*b* **(9.8)**

where *a* and *b* are constants. In intermediate COCOMO, the exponent *b* takes on three different values, depending on whether the mode of the product to be built is organic (*b* = 1.05), semidetached (*b* = 1.12), or embedded (*b* = 1.20). In COCOMO II, the value of *b* varies between 1.01 and 1.26, depending on a variety of parameters of the model. These include familiarity with products of that type, process maturity level (Section 3.13), extent of risk resolution (Section 2.7), and degree of team cooperation (Section 4.1).

A third difference is the assumption regarding reuse. Intermediate COCOMO assumes that the savings due to reuse are directly proportional to the amount of reuse. COCOMO II takes into account that small changes to reused software incur disproportionately large costs (because the code has been understood in detail for even a small change and the cost of testing a modified module is relatively large).

Fourth, there now are 17 multiplicative cost drivers, instead of 15 in intermediate COCOMO. Seven of the cost drivers are new, such as required reusability in future products, annual personnel turnover, and whether the product is being developed at multiple sites. C OCOMO II has been calibrated using 83 projects from a variety of different domains. The model still is too new for there to be many results regarding its accuracy and, in particular, the extent to which it is an improvement over its predecessor, the original (1981)

COCOMO.

## 9.2.5 Tracking Duration and Cost Estimates

While the product is being developed, the actual development effort must constantly be compared against predictions. For example, suppose that the estimation metric used by the software developers predicted that the **duration** of the analysis workflow would last 3 months and require 7 person-months of effort. However, 4 months have gone by and 10 person-months of effort have been expended, yet the specifications are by no means complete. Deviations of this kind can serve as an early warning that something has gone wrong and corrective action must be taken. The problem could be that the size of the product was seriously underestimated or the development team is not as competent as it was thought to be. Whatever the reason, there are going to be serious duration and cost overruns, and management must take appropriate action to minimize the effects.

Careful tracking of predictions must be done throughout the development process, irrespective of the techniques by which the predictions were made. Deviations could be due to metrics that are poor predictors, inefficient software development, a combination of both, or some other reason. The important thing is to detect deviations early and take immediate corrective action. In addition, it is essential to continually update predictions in the light of additional information as it becomes available.

N ow that metrics for estimating duration and cost have been discussed, the components of the software project management plan are described.

## 9.3 Components of a Software Project Management Plan

A software project management plan has three main components: the work to be done, the resources with which to do it, and the money to pay for it all. In this section, these three ingredients of the plan are discussed. The terminology is taken from [IEEE 1058, 1998], which is discussed in greater detail in Section 9.4.

Software development requires *resources*. The major **resources** required are the people who will develop the software, the hardware on which the software is run, and the support software such as operating systems, text editors, and version control software (Section 5.9). U se of resources such as personnel varies with time. Norden [1958] has shown that for large projects, the **Rayleigh distribution** is a good approximation of the way that resource consumption, *Rc*, varies with time, *t*, that is,

*t t* 2*/* 2*k* 2 0 *t* **(9.9)**

*Rc*   *—* 2e

*k*

Parameter *k* is a constant, the time at which consumption is at its peak, and e = 2.71828. . ., the base of Naperian (natural) logarithms. A typical Rayleigh curve is shown in Figure 9.8.

**FIGURE 9.8**

Rayleigh

cur

ve showing

how resource

consumption

varies with time.

Resource consumption

Time

*k*

Resource consumption starts small, climbs rapidly to a peak, and then decreases at a slower rate. Putnam [1978] investigated the applicability of Norden’s results to software development and found that personnel and other resource consumption was modeled with some degree of accuracy by the Rayleigh distribution.

I t therefore is insufficient in a software plan merely to state that three senior programmers with at least 5 years of experience are required. What is needed is something like the following:

Three senior programmers with at least 5 years of experience in real-time programming are needed, two to start 3 months after the project commences, the third to start 6 months after that. Two will be phased out when product testing commences, the third when postdelivery maintenance begins.

The fact that resource needs depend on time applies not only to personnel but also to computer time, support software, computer hardware, office facilities, and even travel. Consequently, the software project management plan is a function of time.

The work to be done falls into two categories. First is work that continues throughout the project and does not relate to any specific workflow of software development. Such work is termed a **project function**. Examples are project management and quality control. Second is work that relates to a specific workflow in the development of the product; such work is termed an *activity* or a *task*. An **activity** is a major unit of work that has precise beginning and ending dates; consumes resources, such as computer time or person-days; and results in **work products**, such as a budget, design documents, schedules, source code, or a user’s manual. An activity, in turn, comprises a set of tasks, a **task** being the smallest unit of work subject to management accountability. There are therefore three kinds of work in a software project management plan: project functions carried on throughout the project, activities (major units of work), and tasks (minor units of work).

A critical aspect of the plan concerns completion of work products. The date on which a work product is deemed completed is termed a **milestone**. To determine whether a work product indeed has reached a milestone, it must first pass a series of **reviews** performed by fellow team members, management, or the client. A typical milestone is the date on which the design is completed and passes review. Once a work product has been reviewed and agreed on, it becomes a **baseline** and can be changed only through formal procedures, as described in Section 5.10.2.

I n reality, there is more to a work product than merely the product itself. A **work package** defines not just the work product but also the staffing requirements, duration, resources, name of the responsible individual, and acceptance criteria for the work product. **Money** of course is a vital component of the plan. A detailed budget must be worked out and the money allocated, as a function of time, to the project functions and activities.

The issue of how to draw up a plan for software production is addressed next.

## 9.4 Software Project Management Plan Framework

There are many ways of drawing up a project management plan. One of the best is IEEE Standard 1058 [1998]. The components of the plan are shown in Figure 9.9.

* The standard was drawn up by representatives of numerous major organizations involved in software development. Input came from both industry and universities, and the members of the working group and reviewing teams had many years of experience in drawing up project management plans. The standard incorporates this experience.
* The IEEE project management plan is designed for use with all types of software products. It does not impose a specific life-cycle model or prescribe a specific methodology. The plan essentially is a framework, the contents of which are tailored by each organization for a particular domain, development team, or technique.
* The IEEE project management plan framework supports process improvement. For example, many of the sections of the framework reflect CMM key process areas (Section 3.13) such as configuration management and metrics.
* The IEEE project management plan framework is ideal for the Unifi ed Process. For instance, one section of the plan is devoted to requirements control and another to risk management, both central aspects of the Unifi ed Process.

On the other hand, although the claim is made in IEEE Standard 1058 [1998] that the IEEE project management plan is applicable to software projects of all sizes, some of the sections are not relevant to small-scale software. For example, section 7.7 of the plan framework is headed “Subcontractor Management Plan,” but it is all but unheard of for subcontractors to be used in small-scale projects.

Accordingly, we now present the plan framework in two different ways. First, the full framework is described in Section 9.5. Second, a slightly abbreviated version of the framework is used in Appendix F for a management plan for a small-scale project, the MSG Foundation case study.

|  |
| --- |
| 1. Overview    1. Project summary       1. Purpose, scope, and objectives       2. Assumptions and constraints       3. Project deliverables       4. Schedule and budget summary    2. Evolution of the project management plan 2. Reference materials 3. Definitions and acronyms 4. Project organization    1. External interfaces    2. Internal structure    3. Roles and responsibilities 5. Managerial process plans    1. Start-up plan       1. Estimation plan       2. Staffing plan       3. Resource acquisition plan       4. Project staff training plan    2. Work plan       1. Work activities       2. Schedule allocation       3. Resource allocation       4. Budget allocation    3. Control plan       1. Requirements control plan       2. Schedule control plan       3. Budget control plan       4. Quality control plan       5. Reporting plan       6. Metrics collection plan    4. Risk management plan    5. Project close-out plan 6. Technical process plans    1. Process model    2. Methods, tools, and techniques    3. Infrastructure plan    4. Product acceptance plan 7. Supporting process plans    1. Configuration management plan    2. Testing plan    3. Documentation plan    4. Quality assurance plan    5. Reviews and audits plan    6. Problem resolution plan    7. Subcontractor management plan    8. Process improvement plan 8. Additional plans |

**FIGURE 9.9** The IEEE project

management plan framework.

## 9.5 IEEE Software Project Management Plan

The **IEEE software project management plan** (SPMP) framework itself now is described in detail. The numbers and headings in the text correspond to the entries in Figure 9.9. The various terms used have been defined in Section 9.3.

1. **Overview.**
   1. **Project summary.**
      1. **Purpose, scope, and objectives.** A brief description is given of the purpose and scope of the software product to be delivered, as well as project objectives. Business needs are included in this subsection.
      2. **Assumptions and constraints.** Any assumptions underlying the project are stated here, together with constraints, such as the delivery date, budget, resources, and artifacts to be reused.
      3. **Project deliverables.** All the items to be delivered to the client are listed here, together with the delivery dates.
      4. **Schedule and budget summary.** The overall schedule is presented here, together with the overall budget.
   2. **Evolution of the project management plan.** No plan can be cast in concrete. The project management plan, like any other plan, requires continual updating in the light of experience and change within both the client organization and the software development organization. In this section, the formal procedures and mechanisms for changing the plan are described, including the mechanism for placing the project management plan itself under configuration control.
2. **Reference materials.** All documents referenced in the project management plan are listed here.
3. **Definitions and acronyms.** This information ensures that the project management plan will be understood the same way by everyone.
4. **Project organization.**
   1. **External interfaces.** No project is constructed in a vacuum. The project members have to interact with the client organization and other members of their own organization. In addition, subcontractors may be involved in a large project. Administrative and managerial boundaries between the project and these other entities must be laid down.
   2. **Internal structure.** In this section, the structure of the development organization itself is described. For example, many software development organizations are divided into two types of groups: development groups that work on a single project and support groups that provide support functions, such as configuration management and quality assurance, on an organization-wide basis. Administrative and managerial boundaries between the project group and the support groups also must be defined clearly.
   3. **Roles and responsibilities.** For each project function, such as quality assurance, and for each activity, such as product testing, the individual responsible must be identified.
5. **Managerial process plans.**
   1. **Start-up plan.**
      1. **Estimation plan.** The techniques used to estimate project duration and cost are listed here, as well as the way these estimates are tracked and, if necessary, modified while the project is in progress.
      2. **Staffing plan.** The numbers and types of personnel required are listed, together with the durations for which they are needed.
      3. **Resource acquisition plan.** The way of acquiring the necessary resources, including hardware, software, service contracts, and administrative services, is given here.
      4. **Project staff training plan.** All training needed for successful completion of the project is listed in this subsection.
   2. **Work plan.**
      1. **Work activities.** In this subsection, the work activities are specified, down to the task level if appropriate.
      2. **Schedule allocation.** In general, the work packages are interdependent and further dependent on external events. For example, the implementation workflow follows the design workflow and precedes product testing. In this subsection, the relevant dependencies are specified.
      3. **Resource allocation.** The various resources previously listed are allocated to the appropriate project functions, activities, and tasks.
      4. **Budget allocation.** In this subsection, the overall budget is broken down at the project function, activity, and task levels.
   3. **Control plan.**
      1. **Requirements control plan.** As described in Part B of this book, while a software product is being developed, the requirements frequently change. The mechanisms used to monitor and control the changes to the requirements are given in this section.
      2. **Schedule control plan.** In this subsection, mechanisms for measuring prog-

ress are listed, together with a description of the actions to be taken if actual progress lags behind planned progress.

* + 1. **Budget control plan.** It is important that spending should not exceed the budgeted amount. Control mechanisms for monitoring when actual cost exceeds budgeted cost, as well as the actions to be taken should this happen, are described in this subsection.
    2. **Quality control plan.** The ways in which quality is measured and controlled are described in this subsection.
    3. **Reporting plan.** To monitor the requirements, schedule, budget, and quality, reporting mechanisms need to be in place. These mechanisms are described in this subsection.
    4. **Metrics collection plan.** As explained in Section 5.5, it is not possible to manage the development process without measuring relevant metrics. The metrics to be collected are listed in this subsection.
  1. **Risk management plan.** Risks have to be identified, prioritized, mitigated, and tracked. All aspects of risk management are described in this section.
  2. **Project close-out plan.** The actions to be taken once the project is completed, including reassignment of staff and archiving of artifacts, are presented here.

1. **Technical process plans.**
   1. **Process model.** In this section, a detailed description is given of the life-cycle model to be used.
   2. **Methods, tools, and techniques.** The development methodologies and programming languages to be used are described here.
   3. **Infrastructure plan.** Technical aspects of hardware and software are described in detail in this section. Items that should be covered include the computing systems (hardware, operating systems, network, and software) to be used for developing the software product, as well as the target computing systems on which the software product will be run and CASE tools to be employed.
   4. **Product acceptance plan.** To ensure that the completed software product passes its acceptance test, acceptance criteria must be drawn up, the client must agree to the criteria in writing, and the developers must then ensure that these criteria are indeed met. The way that these three stages of the acceptance process will be carried out is described in this section.
2. **Supporting process plans.**
   1. **Configuration management plan.** In this section, a detailed description is given of the means by which all artifacts are put under configuration management.
   2. **Testing plan.** Testing, like all other aspects of software development, needs careful planning.
   3. **Documentation plan.** A description of documentation of all kinds, whether or not to be delivered to the client at the end of the project, is included in this section.
   4. **Quality assurance plan.** All aspects of quality assurance, including testing, standards, and reviews, are encompassed by this section.
   5. **Reviews and audits plan.** Details as to how reviews are conducted are presented in this section.
   6. **Problem resolution plan.** In the course of developing a software product, problems are all but certain to arise. For example, a design review may bring to light a critical fault in the analysis workflow that requires major changes to almost all the artifacts already completed. In this section, the way such problems are handled is described.
   7. **Subcontractor management plan.** This section is applicable when subcontractors are to supply certain work products. The approach to selecting and managing subcontractors then appears here.
   8. **Process improvement plan.** Process improvement strategies are included in this section.
3. **Additional plans.** For certain projects, additional components may need to appear in the plan. In terms of the IEEE framework, they appear at the end of the plan. Additional components may include security plans, safety plans, data conversion plans, installation plans, and the software project postdelivery maintenance plan.

## 9.6 Planning Testing

One component of the SPMP frequently overlooked is **test planning**. Like every other activity of software development, testing must be planned. The SPMP must include resources for testing, and the detailed schedule must explicitly indicate the testing to be done during each workflow.

Without a test plan, a project can go awry in a number of ways. For example, during product testing (Section 3.7.4), the SQA group must check that every aspect of the specification document, as signed off on by the client, has been implemented in the completed product. A good way of assisting the SQA group in this task is to require that the development be traceable (Section 3.7). That is, it must be possible to connect each statement in the specification document to a part of the design, and each part of the design must be reflected explicitly in the code. One technique for achieving this is to number each statement in the specification document and ensure that these numbers are reflected in both the design and the resulting code. However, if the test plan does not specify that this is to be done, it is highly unlikely that the analysis, design, and code artifacts will be labeled appropriately. Consequently, when the product testing finally is performed, it will be extremely difficult for the SQA group to determine that the product is a complete implementation of the specifications. In fact, traceability should start with the requirements; each statement in the requirements artifacts (or each portion of the rapid prototype) must be connected to part of the analysis artifacts.

One powerful aspect of inspections is the detailed list of faults detected during an inspection. Suppose that a team is inspecting the specifications of a product. As explained in Section 6.2.3, the list of faults is used in two ways. First, the fault statistics from this inspection must be compared with the accumulated averages of fault statistics from previous specification inspections. Deviations from previous norms indicate problems within the project. Second, the fault statistics from the current specification inspection must be carried forward to the design and code inspections of the product. After all, if there is a large number of faults of a particular type, it is possible that not all of them were detected during the inspection of the specifications, and the design and code inspections provide an additional opportunity for locating any remaining faults of this type. However, unless the test plan states that details of all faults have to be carefully recorded, it is unlikely that this task will be done.

An important way of testing code modules is so-called black-box testing (Section 15.11) in which the code is executed with test cases based on the specifications. Members of the SQA group read through the specifications and draw up test cases to check whether the code obeys the specification document. The best time to draw up black-box test cases is at the end of the analysis workflow, when the details of the specification document still are fresh in the minds of the members of the SQA group that inspected them. However, unless the test plan explicitly states that the black-box test cases are to be selected at this time, in all probability only a few black-box test cases will be hurriedly thrown together later. That is, a limited number of test cases will be rapidly assembled only when pressure starts mounting from the programming team for the SQA group to approve its modules so that they can be integrated into the product as a whole. As a result, the quality of the product as a whole suffers.

Therefore, every test plan must specify what testing is to be performed, when it is to be performed, and how it is to be performed. Such a test plan is an essential part of section 7.2 of the SPMP. Without it, the quality of the overall product undoubtedly will suffer.

## 9.7 Planning Object-Oriented Projects

Suppose the classical paradigm is used. From a conceptual viewpoint, the resulting product generally is one large unit, even though it is composed of separate modules. In contrast, use of the object-oriented paradigm results in a product consisting of a number of relatively

independent smaller components, namely, the classes. This makes planning considerably easier, in that cost and duration estimates can be computed more easily and more accurately for smaller units. Of course, the estimates must take into account that a product is more than just the sum of its parts. The separate components are not totally independent; they can invoke one another, and these effects must not be overlooked.

A re the techniques for estimating cost and duration described in this chapter applicable to the object-oriented paradigm? COCOMO II (Section 9.2.4) was designed to handle modern software technology, including object orientation, but what about earlier metrics such as function points (Section 9.2.1) and intermediate COCOMO (Section 9.2.3)? In the case of intermediate COCOMO, minor changes to some of the cost multipliers are required [Pittman, 1993]. Other than that, the estimation tools of the classical paradigm appear to work reasonably well on object-oriented projects—provided that there is no reuse. Reuse enters the object-oriented paradigm in two ways: reuse of existing components during development and the deliberate production (during the current project) of components to be reused in future products. Both forms of reuse affect the estimating process. Reuse during development clearly reduces the cost and duration. Formulas have been published showing the savings as a function of this reuse [Schach, 1994], but these results relate to the classical paradigm. At present, no information is available as to how the cost and duration change when reuse is utilized in the development of an object-oriented product.

We turn now to the goal of reusing parts of the current project. It can take about three times as long to design, implement, test, and document a reusable component as a similar nonreusable component [Pittman, 1993]. Cost and duration estimates must be modified to incorporate this additional labor, and the SPMP as a whole must be adjusted to incorporate the effect of the reuse endeavor. Therefore, the two reuse activities work in opposite directions. Reuse of existing components reduces the overall effort in developing an object-oriented product, whereas designing components for reuse in future products increases the effort. It is expected that, in the long term, the savings due to reuse of classes will outweigh the costs of the original developments, and already some evidence supports this [Lim, 1994].

## 9.8 Training Requirements

When the subject of **training** is raised in discussions with the client, a common response is, “We don’t need to worry about training until the product is finished, then we can train the users.” This is a somewhat unfortunate remark, implying as it does that only users require training. In fact, training also may be needed by members of the development team, starting with training in software planning and estimating. When new software development techniques, such as new design techniques or testing procedures, are used, training must be provided to every member of the team using the new technique.

Introduction of the object-oriented paradigm has major training consequences. The introduction of hardware or software tools such as workstations or an integrated environment (see Section 15.24.2) also requires training. Programmers may need training in the operating system of the machine to be used for product development as well as in the implementation language. Documentation preparation training frequently is overlooked, as evidenced by the poor quality of so much documentation. Computer operators certainly require some

sort of training to be able to run the new product; they also may require additional training if new hardware is utilized.

The required training can be obtained in a number of ways. The easiest and least disruptive is in-house training, by either fellow employees or consultants. Many companies offer a variety of training courses, and colleges often offer training courses in the evenings. World Wide Web–based courses are another alternative.

Once the training needs have been determined and the training plan drawn up, the plan must be incorporated into the SPMP.

## 9.9 Documentation Standards

The development of a software product is accompanied by a wide variety of **documentation**. Jones found that 28 pages of documentation were generated per 1000 instructions (KDSI) for an IBM internal commercial product around 50 KDSI in size, and about 66 pages per KDSI for a commercial software product of the same size. Operating system IMS/360 Version 2.3 was about 166 KDSI in size, and 157 pages of documentation per KDSI were produced. The documentation was of various types, including planning, control, financial, and technical [Jones, 1986a]. In addition to these types of documentation, the source code itself is a form of documentation; comments within the code constitute further documentation.

A considerable portion of the software development effort is absorbed by documentation. A survey of 63 development projects and 25 postdelivery maintenance projects showed that, for every 100 hours spent on activities related to code, 150 hours were spent on activities related to documentation [Boehm, 1981]. For large TRW products, the proportion of time devoted to documentation-related activities rose to 200 hours per 100 code-related hours [Boehm et al., 1984].

Standards are needed for every type of documentation. For instance, uniformity in design documentation reduces misunderstandings between team members and aids the SQA group. Although new employees have to be trained in the documentation standards, no further training is needed when existing employees move from project to project within the organization. From the viewpoint of postdelivery maintenance, uniform coding standards assist maintenance programmers in understanding source code. Standardization is even more important for user manuals, because these have to be read by a wide variety of individuals, few of whom are computer experts. The IEEE has developed a standard for user manuals (IEEE Standard 1063 for Software User Documentation).

As part of the planning process, standards must be established for all documentation to be produced during software production. These standards are incorporated in the SPMP. Where an existing standard is to be used, such as the ANSI/IEEE Standard for Software Test Documentation [ANSI/IEEE 829, 1991], the standard is listed in section 2 of the SPMP (reference materials). If a standard is specially written for the development effort, then it appears in section 6.2 (methods, tools, and techniques).

Documentation is an essential aspect of the software production effort. In a very real sense, the product *is* the documentation, because without documentation the product cannot be maintained. Planning the documentation effort in every detail, and then ensuring that the plan is adhered to, is a critical component of successful software production.

## 9.10 CASE Tools for Planning and Estimating

A number of tools are available that automate intermediate COCOMO and COCOMO II. For speed of computation when the value of a parameter is modified, several implementations of intermediate COCOMO have been implemented in spreadsheet languages such as Lotus 1-2-3 and Excel. For developing and updating the plan itself, a word processor is essential.

Management information tools also are useful for planning. For example, suppose that a large software organization has 150 programmers. A scheduling tool can help planners keep track of which programmers already are assigned to specific tasks and which are available for the current project.

More general types of management information also are needed. A number of commercially available management tools can be used both to assist with the planning and estimating process and to monitor the development process as a whole. These include MacProject and Microsoft Project.

## 9.11 Testing the Software Project Management Plan

|  |  |
| --- | --- |
|  | As pointed out at the beginning of this chapter, a fault in the software project management plan can have serious financial implications for the developers. It is critical that the development organization neither overestimate nor underestimate the cost of the project or its duration. For this reason, the entire SPMP must be checked by the SQA group before estimates are given to the client. The best way to test the plan is by a plan inspection.  The plan inspection team must review the SPMP in detail, paying particular attention to the cost and duration estimates. To reduce risks even further, irrespective of the metrics used, the duration and cost estimates should be computed independently by a member of the SQA group as soon as the members of the planning team have determined their estimates. |
| **Chapter Review** | The main theme of this chapter is the importance of planning in the software process (Section 9.1). A vital component of any software project management plan is estimating the duration and the cost (Section 9.2). Several metrics are put forward for estimating the size of a product, including function points (Section 9.2.1). Next, various metrics for cost estimation are described, especially intermediate COCOMO (Section 9.2.3) and COCOMO II (Section 9.2.4). As described in Section 9.2.5, it is essential to track all estimates. The three major components of a software project management plan—the work to be done, the resources with which to do it, and the money to pay for it—are explained in Section 9.3. One particular SPMP, the IEEE standard, is outlined in Section 9.4 and described in detail in Section 9.5. Next follow sections on planning testing (Section 9.6), planning object-oriented projects (Section 9.7), and training requirements and documentation standards and their implications for the planning process (Sections 9.8 and 9.9). CASE tools for planning and estimating are described in Section 9.10. The chapter concludes with material on testing the software project management plan (Section 9.11). |
| **For**  **Further**  **Reading** | Weinberg’s four-volume work [Weinberg, 1992; 1993; 1994; 1997] provides detailed information on many aspects of software management, as do [Bennatan, 2000] and [Reifer, 2000]. The September– October 2005 issue of *IEEE Software* contains a number of articles on software management, especially [Royce, 2005] and [Venugopal, 2005]; there are additional articles in the May–June 2008 issue. The way | |

managers defi ne success is explained in [Procaccino and Verner, 2006]. The mechanisms used by project managers to monitor and control software development projects are discussed in [McBride, 2008]. F or further information on IEEE Standard 1058 for Software Project Management Plans, the standard itself should be read carefully [IEEE 1058, 1998]. The need for careful planning is described in [McConnell, 2001].

Sackman’s classic work is described in [Sackman, Erikson, and Grant, 1968]. A more detailed source is [Sackman, 1970]. The impact of programmer expertise on pair programming is described in [Arisholm, Gallis, Dybå, and Sjøberg, 2007].

A careful analysis of function points, as well as suggested improvements, appears in [Symons, 1991]. Strengths and weaknesses of function points are presented in [Furey and Kitchenham, 1997]. Class points, an extension of function points to classes, are introduced in [Costagliola, Ferrucci, Tortora, and Vitiello, 2005].

The theoretical justification for intermediate COCOMO, together with full details for implementing it, appears in [Boehm, 1981]. COCOMO II is described in [Boehm et al., 2000]. Ways of enhancing COCOMO predictions are presented in [Smith, Hale, and Parrish, 2001]. An extension of COCOMO to software product lines appears in [In, Baik, Kim, Yang, and Boehm, 2006].

activity

*283*

algorithmic cost estimation

model

*277*

application composition

model

*281*

baseline

*284*

bottom-up approach

*277*

COCOMO

*278*

COCOMO II

*281*

cone of uncertainty

*269*

cost

*271*

cost estimate

*271*

Delphi technique

*276*

documentation

*291*

duration

*282*

duration estimate

*271*

early design model

*281*

efficiency

*273*

expert judgment by analogy

*276*

external cost

*271*

FFP metric

*273*

function point (FP)

*273*

IEEE software project

management plan

*286*

internal cost

*271*

lines of code (LOC)

*272*

milestone

*284*

money

*284*

nominal effort

*278*

planning

*268*

postarchitecture model

*281*

price

*271*

productivity

*273*

project

function

*283*

R

ayleigh distribution

*282*

resources

*282*

review

*284*

software development effort

multipliers (SPMP)

*278*

**Key Terms**

Ariand and Wüst [2001] describe how to estimate the development effort for object-oriented products. Estimating both the size and defects of object-oriented software products is described in [Cartwright and Shepperd, 2000].

Software productivity data for a variety of business data-processing products are presented in [Maxwell and Forselius, 2000]; the unit of productivity utilized is function points per hour. Other measures of productivity are discussed in [Kitchenham and Mendes, 2004]. Errors in estimating software effort are analyzed in [Jorgensen and Moløkken-Østvold, 2004]. A critique of a frequently used research procedure for comparing estimation models is given in [Myrtveit, Stensrud, and Shepperd, 2005]. A probabilist model for predicting software development effort appears in [Pendharkar, Subramanian, and Rodger, 2005]. An analysis of cost overruns for software products constructed with various life-cycle models appears in [Moløkken-Østvold and Jorgensen, 2005]. Having an effective requirements workfl ow can have a positive impact on productivity; this is shown in [Damian and Chisan, 2006]. The impact of the cone of uncertainty on schedule estimate is analyzed in [Little, 2006]. A comprehensive review of 304 development cost estimation studies in 76 journals is presented in [Jorgensen and Shepperd, 2007]. An evidence-based approach to selecting an appropriate cost-estimation model for a given project is described in [Menzies and Hihn, 2006].

**294** Part A *Software Engineering Concepts*

task *283* thousand delivered unadjusted function points technical complexity factor source instructions (UFP) *273*

(TCF) *274* (KDSI) *272* work package *284* test planning *288* training *290* work product *283*

**Problems** 9 .1 Why do you think that some cynical software organizations refer to *milestones* as *millstones*? (Hint: Look up the figurative meaning of *millstone* in a dictionary.)

9.2 You are a software engineer at Pretoriuskop Software Developers. A year ago, your manager announced that your next product would comprise 8 fi les, 48 flows, and 91 processes.

* 1. Using the FFP metric, determine its size.
  2. For Pretoriuskop Software Developers, the constant *d* in equation (9.2) has been determined to be $1021. What cost estimate did the FFP metric predict?
  3. The product recently was completed at a cost of $135,200. What does this tell you about the productivity of your development team?

9.3 A target product has 8 simple inputs, 3 average inputs, and 11 complex inputs. There are 57 average outputs, 9 simple inquiries, 13 average master fi les, and 18 complex interfaces. Determine the unadjusted function points (*UFP*).

9.4 If the total degree of influence for the product of Problem 9.3 is 47, determine the number of function points.

9.5 Why do you think that, despite its drawbacks, lines of code (LOC or KDSI) is so widely used as a metric of product size?

9.6 You are in charge of developing a 62-KDSI embedded product that is nominal except that the database size is rated very high and the use of software tools is low. Using intermediate COCOMO, what is the estimated effort in person-months?

9.7 You are in charge of developing two 31-KDSI organic-mode products. Both are nominal in every respect except that product P1 has extra-high complexity and product P2 has extra-low complexity. To develop the product, you have two teams at your disposal. Team A has very high analyst capability, applications experience, and programmer capability. Team A also has high virtual machine experience and programming language experience. Team B is rated very low on all five attributes.

* 1. What is the total effort (in person-months) if team A develops product P1 and team B develops product P2?
  2. What is the total effort (in person-months) if team B develops product P1 and team A develops product P2?
  3. Which of the two preceding staffing assignments makes more sense? Is your intuition backed by the predictions of intermediate COCOMO?

9.8 You are in charge of developing a 48-KDSI organic-mode product that is nominal in every respect.

* 1. Assuming a cost of $10,100 per person-month, how much is the project estimated to cost?
  2. Your entire development team resigns at the start of the project. You are fortunate enough to be able to replace the nominal team with a very highly experienced and capable team, but the cost per person-month will rise to $13,400. How much money do you expect to gain (or lose) as a result of the personnel change?

9.9 You are in charge of developing the software for a product that uses a set of newly developed algorithms to compute the most cost-effective routes for a large trucking company. Using

Chapter 9 *Planning and Estimating* **295**

intermediate COCOMO, you determine that the cost of the product will be $470,000. However, as a check, you ask a member of your team to estimate the effort using function points. She reports that the function point metric predicts a cost of $985,000, more than twice as large as your COCOMO prediction. What do you do now?

9.10 Show that the Rayleigh distribution [equation (9.9)] attains its maximum value when *t* = *k*. Find the corresponding resource consumption.

9.11 A product postdelivery maintenance plan is considered an “additional component” of an IEEE software project management plan. Bearing in mind that every nontrivial product is maintained and that the cost of postdelivery maintenance, on average, is about twice or three times the cost of developing the product, how can this be justifi ed?

9.12 Why do software development projects generate so much documentation?

9.13 (Term project) Consider the Chocoholics Anonymous project described in Appendix A. Why is it not possible to estimate the cost and duration purely on the basis of the information in Appendix A?

9.14 (Readings in Software Engineering) Your instructor will distribute copies of [Costagliola, Ferrucci, Tortora, and Vitiello, 2005]. Are you convinced by the empirical validation of class points?

**References** [Albrecht, 1979] A. J. ALBRECHT, “Measuring Application Development Productivity,” *Proceedings of the IBM SHARE/GUIDE Applications Development Symposium*, Monterey, CA, October 1979, pp. 83–92.

[ANSI/IEEE 829, 1991] *Software Test Documentation*, ANSI/IEEE 829-1991, American National Standards Institute, Institute of Electrical and Electronic Engineers, New York, 1991.

[ Arisholm, Gallis, Dybå, and Sjøberg, 2007] E. ARISHOLM, H. GALLIS, T. DYBÅ, AND D. I. K. SJØBERG, “Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise,” *IEEE Transactions on Software Engineering* **33** (February 2007), pp. 65–86.

[Bennatan, 2000] E. M. BENNATAN, *On Time within Budget: Software Project Management Practices and Techniques*, 3rd ed., John Wiley and Sons, New York, 2000.

[Boehm, 1981] B. W. BOEHM, *Software Engineering Economics,* Prentice Hall, Englewood Cliffs, NJ, 1981.

[ Boehm, 1984] B. W. BOEHM, “Software Engineering Economics,” *IEEE Transactions on Software Engineering* **SE-10** (January 1984), pp. 4–21.

[Boehm et al., 1984] B. W. BOEHM, M. H. PENEDO, E. D. STUCKLE, R. D. WILLIAMS, AND A. B. PYSTER, “A Software Development Environment for Improving Productivity,” *IEEE Computer* **17** (June 1984), pp. 30–44.

[ Boehm et al., 2000] B. W. BOEHM, C. ABTS, A. W. BROWN, S. CHULANI, B. K. CLARK, E. HOROWITZ, R. MADACHY, D. REIFER, AND B. STEECE, *Software Cost Estimation with COCOMO II*, Prentice Hall, Upper Saddle River, NJ, 2000.

[Briand and Wüst, 2001] L. C. BRIAND AND J. WÜST, “Modeling Development Effort in ObjectOriented Systems Using Design Properties,” *IEEE Transactions on Software Engineering* **27** (November 2001), pp. 963–86.

[Cartwright and Shepperd, 2000] M. CARTWRIGHT AND M. SHEPPERD, “An Empirical Investigation of an Object-Oriented Software System,” *IEEE Transactions on Software Engineering* **26** (August 2000), pp. 786–95.

[ Costagliola, Ferrucci, Tortora, and Vitiello, 2005] G. COSTAGLIOLA, F. FERRUCCI, G. TORTORA, AND G. VITIELLO, “Class Point: An Approach for the Size Estimation of Object-Oriented Systems,” *IEEE Transactions on Software Engineering* **31** (January 2005), pp. 52–74.

**296** Part A *Software Engineering Concepts*

[ Damian and Chisan, 2006] D. DAMIAN AND J. CHISAN, “An Empirical Study of the Complex Relationships between Requirements Engineering Processes and Other Processes That Lead to Payoffs in Productivity, Quality, and Risk Management,” *IEEE Transactions on Software Engineering* **32** (July 2006), pp. 433–53.

[Devenny, 1976] T. DEVENNY, “An Exploratory Study of Software Cost Estimating at the Electronic Systems Division,” Thesis No. GSM/SM/765–4, Air Force Institute of Technology, Dayton, OH, 1976.

[Furey and Kitchenham, 1997] S. FUREY AND B. KITCHENHAM, “Function Points,” *IEEE Software* **14** (March–April 1997), pp. 28–32.

[IEEE 1058, 1998] “IEEE Standard for Software Project Management Plans.” IEEE Std. 1058-1998, Institute of Electrical and Electronic Engineers, New York, 1998.

[In, Baik, Kim, Yang, and Boehm, 2006] H. P. IN, J. BAIK, S. KIM, Y. YANG, AND B. BOEHM, “A QualityBased Cost Estimation Model for the Product Line Life Cycle,” *Communications of the ACM* **49** (December 2006), pp. 85–88.

[Jones, 1986a] C. JONES, *Programming Productivity,* McGraw-Hill, New York, 1986.

[Jones, 1987] C. JONES, Letter to the Editor, *IEEE Computer* **20** (December 1987), p. 4.

[ Jorgensen and Moløkken-Østvold, 2004] M. JORGENSEN and K. MOLØKKEN-ØSTVOLD, “Reasons for Software Effort Estimation Error: Impact of Respondent Role, Information Collection Approach, and Data Analysis Method,” *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 993–1007.

[ Jorgensen and Shepperd, 2007] M. JORGENSEN AND M. SHEPPERD, “A Systematic Review of Software Development Cost Estimation Studies,” *IEEE Transactions on Software Engineering* **32** (January 2007), pp. 33–53.

[ Kitchenham and Mendes, 2004] B. KITCHENHAM AND E. MENDES, “Software Productivity Measurement Using Multiple Size Measures,” *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 1023–35.

[ Lim, 1994] W. C. LIM, “Effects of Reuse on Quality, Productivity, and Economics,” *IEEE Software* **11** (September 1994), pp. 23–30.

[Little, 2006] T. LITTLE, “Schedule Estimation and Uncertainty Surrounding the Cone of Uncertainty,” *IEEE Software* **23** (May–June 2006), pp. 48–54.

[ Maxwell and Forselius, 2000] K. D. MAXWELL AND P. FORSELIUS, “Benchmarking Software Development Productivity,” *IEEE Software* **17** (January–February 2000), pp. 80–88.

[McBride, 2008] T. MCBRIDE, “The Mechanisms of Project Management of Software Development,” *Journal of Systems and Software* **81** (December 2008), pp. 2386–95.

[ McConnell, 2001] S. MCCONNELL, “The Nine Deadly Sins of Project Planning,” *IEEE Software* **18** (November–December 2001), pp. 5–7.

[Menzies and Hihn, 2006] T. MENZIES AND J. HIHN, “Evidence-Based Cost Estimation for BetterQuality Software,” *IEEE Software* **23** (July–August 2006), pp. 64–66.

[ Moløkken-Østvold and Jorgensen, 2005] K. MOLØKKEN-ØSTVOLD AND M. JORGENSEN, “A Comparison of Software Project Overruns—Flexible versus Sequential Development Models,” *IEEE Transactions on Software Engineering* **31** (September 2005), pp. 754–66.

[Myrtveit, Stensrud, and Shepperd, 2005] I. MYRTVEIT, E. STENSRUD, AND M. SHEPPERD, “Reliability and Validity in Comparative Studies of Software Prediction Models,” *IEEE Transactions on Software Engineering* **31** (May 2005), pp. 380–91.

[Norden, 1958] P. V. NORDEN, “Curve Fitting for a Model of Applied Research and Development Scheduling,” *IBM Journal of Research and Development* **2** (July 1958), pp. 232–48.

Chapter 9 *Planning and Estimating* **297**

[Pendharkar, Subramanian, and Rodger, 2005] P. C. PENDHARKAR, G. H. SUBRAMANIAN, AND J. A. RODGER, “A Probabilistic Model for Predicting Software Development Effort,” *IEEE Transactions on Software Engineering* **31** (July 2005), pp. 615–24.

[ Pittman, 1993] M. PITTMAN, “Lessons Learned in Managing Object-Oriented Development,” *IEEE Software* **10** (January 1993), pp. 43–53.

[ Procaccino and Verner, 2006] J. D. PROCACCINO AND J. M. VERNER, “How Agile Are Industrial Software Development Practices?” *Journal of Systems and Software* **79** (November 2006), pp. 1541–51.

[ Putnam, 1978] L. H. PUTNAM, “A General Empirical Solution to the Macro Software Sizing and Estimating Problem,” *IEEE Transactions on Software Engineering* **SE-4** (July 1978), pp. 345–61.

[Reifer, 2000] D. J. REIFER, “Software Management: The Good, the Bad, and the Ugly,” *IEEE Software* **17** (March–April 2000), pp. 73–75.

[Royce, 2005] W. ROYCE, “Successful Software Management Style: Steering and Balance,” *IEEE Software* **22** (September–October 2005), pp. 40–47.

[Sackman, 1970] H. SACKMAN, *Man–Computer Problem Solving: Experimental Evaluation of TimeSharing and Batch Processing,* Auerbach, Princeton, NJ, 1970.

[Sackman, Erikson, and Grant, 1968] H. SACKMAN, W. J. ERIKSON, AND E. E. GRANT, “Exploratory Experimental Studies Comparing Online and Offl ine Programming Performance,” *Communications of the ACM* **11** (January 1968), pp. 3–11.

[ Schach, 1994] S. R. SCHACH, “The Economic Impact of Software Reuse on Maintenance,” *Journal of Software Maintenance: Research and Practice* **6** (July–August 1994), pp. 185–96.

[Smith, Hale, and Parrish, 2001] R. K. SMITH, J. E. HALE, AND A. S. PARRISH, “An Empirical Study Using Task Assignment Patterns to Improve the Accuracy of Software Effort Estimation,” *IEEE Transactions on Software Engineering* **27** (March 2001), pp. 264–71.

[ Symons, 1991] C. R. SYMONS, *Software Sizing and Estimating: Mk II FPA*, John Wiley and Sons, Chichester, UK, 1991.

[ van der Poel and Schach, 1983] K. G. VAN DER POEL AND S. R. SCHACH, “A Software Metric for Cost Estimation and Effi ciency Measurement in Data Processing System Development,” *Journal of Systems and Software* **3** (September 1983), pp. 187–91.

[Venugopal, 2005] C. VENUGOPAL, “Single Goal Set: A New Paradigm for IT Megaproject Success,” *IEEE Software* **22** (September–October 2005), pp. 48–53.

[Weinberg, 1992] G. M. WEINBERG, *Quality Software Management: Systems Thinking*, Vol. 1, Dorset House, New York, 1992.

[ Weinberg, 1993] G. M. WEINBERG, *Quality Software Management: First-Order Measurement*, Vol. 2, Dorset House, New York, 1993.

[Weinberg, 1994] G. M. WEINBERG, *Quality Software Management: Congruent Action*, Vol. 3, Dorset House, New York, 1994.

[Weinberg, 1997] G. M. WEINBERG, *Quality Software Management: Anticipating Change*, Vol. 4, Dorset House, New York, 1997.

### This page intentionally left blank