7

Cost Estimation

LEARNING OBJECTIVES

- To appreciate the use of quantitative, objective approaches to software cost estimation
- To have insight in the factors that affect software development productivity
- To understand well-known techniques for estimating software cost and effort
- To understand techniques for relating effort to development time

Software development takes time and money. When commissioning a building project, you expect a reliable estimate of the cost and development time up front. Getting reliable cost and schedule estimates for software development projects is still largely a dream. Software development cost is notoriously difficult to estimate reliably at an early stage. Since progress is difficult to 'see' --just when is a piece of software 50% complete? --schedule slippages often go undetected for quite a while, and schedule overruns are the rule, rather than the exception. In this chapter, we look at various ways to estimate software cost and schedule.

When commissioning a house construction, decorating the bathroom, or laying-out a garden, we expect a precise estimate of the costs to be incurred before the operation is started. A gardener is capable of giving a rough indication of the cost on the basis of, say, the area of land, the desired size of the terrace or grass area, whether or not a pond is required, and similar information. The estimate can be made more precise in further dialog, before the first bit of earth is turned. If you expect a similar accuracy as regards the cost estimate for a software development project, you are in for a surprise.

Estimating the cost of a software development project is a rather unexplored field, in which one all too often relies on mere guesstimates. There are exceptions to this procedure, fortunately. There now exist a number of algorithmic models that allow us to estimate total cost and development time of a software development project, based on estimates for a limited number of relevant cost drivers. Some of the important algorithmic cost estimation models are discussed in section 7.1.

In most cost estimation models, a simple relation between cost and effort is assumed. The effort may be measured in man-months, for instance, and each man-month is taken to incur a fixed amount, say, of $5000. The total estimated cost is then obtained by simply multiplying the estimated number of man-months by this constant factor. In this chapter, we freely use the terms cost and effort as if they are synonymous.

The notion of total cost is usually taken to indicate the cost of the initial software development effort, i.e. the cost of the requirements engineering, design, implementation and testing phases. Thus, maintenance costs are not taken into account. Unless explicitly stated otherwise, this notion of cost will also be used by us. In the same vein, development time will be taken to mean: the time between the start of the requirements engineering phase and the point in time when the software is delivered to the customer. Lastly, the notion of cost as it is used here, does not include possible hardware costs either. It concerns only personnel costs involved in software development.

Research in the area of cost estimation is far from crystallized. Different models use different measures and cost drivers, so that mutual comparisons are very difficult.

Suppose some model uses an equation of the form:

$$E = 2.7KLOC^{1.05}$$

This equation shows a certain relation between effort needed (E) and the size of the product ($KLOC$ = Kilo Lines Of Code = Lines Of Code/1000). The effort measure could be the number of man-months needed. Several questions come to mind immediately: What is a line of code? Do we count machine code, or the source code in some high-level language? Do we count comment lines, or blank lines that increase readability? Do we take into account holidays, sick-leave, and the like, in our notion of the man-month, or does it concern a net measure? Different interpretations of these notions may lead to widely different results. Unfortunately, different models do use different definitions of these notions. Sometimes, it is not even known which definitions were used in the derivation of the model.

To determine the equations of an algorithmic cost estimation model, we may follow several approaches. Firstly, we may base our equations on the results of experiments. In such an experiment, we in general vary one parameter, while the other parameters are kept constant. In this way, we may try to determine the influence of the parameter that is being varied. As a typical example, we may consider the question of whether or not comments help to build up our understanding of a program. Under careful control of the circumstances, we may pose a number of questions about one and the same program text to two groups of programmers. The first group gets program text without comments, the second group gets the same program text, with comments. We may check our hypothesis using the results of the two groups. The, probably realistic, assumption in this experiment is that a better and faster understanding of the program text has a positive effect on the maintainability of that program.

This type of laboratory experiment is often performed at universities, where students play the role of programmers. It is not self-evident that the results thus obtained also holds in industrial settings. In practice, there may be a rather complicated interaction between different relevant factors. Also, the subjects need not be representative. Finally, the generalization from laboratory experiments that are (of necessity) limited in size to big software development projects with which professionals are confronted is not possible. The general opinion is that results thus obtained have limited validity, and certainly need further testing.

A second way to arrive at algorithmic cost estimation models is based on an analysis of real project data, in combination with some theoretical underpinning. An organization may collect data about a number of software systems that have been developed. These data may concern the time spent on the various phases that are being distinguished, the qualifications of the personnel involved, the points in time at which errors occurred, both during testing and after installation, the complexity, reliability and other relevant project factors, the size of the resulting code, etc. Based on a sound hypothesis of the relations between the various entities involved and a (statistical) analysis of these data we may derive equations that numerically

characterize these relations. An example of such a relation is the one given above, which relates E to $KLOC$. The usability and reliability of such equations is obviously very much dependent upon the reliability of the data on which they are based. Also, the hypothesis that underlies the form of the equation must be sound.

The findings obtained in this way reflect an average, a best possible approximation based on available data. We therefore have to be very careful in applying the results obtained. If the software to be developed in the course of a new project cannot be compared with earlier products because of the degree of innovation involved, one is in for a big surprise. For example, estimating the cost of the Space Shuttle project cannot be done through a simple extrapolation from earlier projects. We may hope, however, that the average software development project has a higher predictability as regards effort needed and the corresponding cost.

The way in which we obtain quantitative relations implies further constraints on the use of these models. The model used is based on an analysis of data from earlier projects. Application of the model to new projects is possible only insofar as those new projects resemble old projects, i.e. the projects on whose data the model is based. If we have collected data on projects of a certain kind and within a particular organization, a model based on these data cannot be used without amendment for different projects in a possibly different organization. A model based on data about administrative projects in a government environment has little predictive value for the development of real-time software in the aerospace industry. This is one of the reasons why the models of, for example, Walston and Felix (1977) and Boehm (1981) (see section 7.1 for more detailed discussions of these models) yield such different results for one and the same problem description.

The lesson to be learned is that blind application of the formulae from existing models will not solve your cost estimation problem. Each model needs tuning to the environment in which it is going to be used. This implies the need to continuously collect your own project data, and to apply statistical techniques to calibrate model parameters.

Other reasons for the discrepancies between existing models are:

- Most models give a relation between man-months needed and size (in lines of code). As remarked before, widely different definitions of these notions are used.

- The notion 'effort' does not always mean the same thing. Sometimes, one only counts the activities starting from the design, i.e. after the requirements specification has been fixed. Sometimes also, one includes maintenance effort.

Despite these discrepancies, the various cost estimation models do have a number of characteristics in common. These common characteristics reflect important factors that bear on development cost and effort. The increased understanding of software costs allows us to identify strategies for improving software productivity, the most important of which are:

- Writing less code. System size is one of the main determinants of effort and cost. Techniques that try to reduce size, such as software reuse and the use of high-level languages, can obtain significant savings.
- Getting the best from people. Individual and team capabilities have a large impact on productivity. The best people are usually a bargain. Better incentives, better work environments, training programs and the like provide further productivity improvement opportunities.
- Avoiding rework. Studies have shown that a considerable effort is spent redoing earlier work. The application of prototyping or evolutionary development process models and the use of modern programming practices (information hiding) can yield considerable savings.
- Developing and using integrated project support environments. Tools can help us eliminate steps or make steps more efficient.

In the next section, we discuss and compare some of the well-known algorithmic models for cost estimation. In many organizations, software cost is estimated by human experts, who use their expertise and gut feeling, rather than a formula, to arrive at a cost estimate. Some of the do's and don'ts of expert-based cost estimation are discussed in section 7.2.

Given an estimate of the size of a project, we will next be interested in the development time needed. With a naive view, we may conjecture that a project with an estimated effort of 100 man-months can be done in 1 year with a team of 8.5 people, but equally well in one month with a team of 100 people. This view is too naive. A project of a certain size corresponds to a certain nominal physical time period. If we try to shorten this nominal development time too much, we get into the 'impossible region' and the chance of failure sharply increases. This phenomenon is further discussed in section 7.3.

The topics addressed in this chapter fit planning-driven development projects more than they do agile projects. In agile projects, iterations are usually fairly small, and do not warrant the effort required by the algorithmic models discussed in section 7.1. In agile projects (see chapter 3, increments correspond to one or a few user stories or scenarios. These user stories are estimated in terms of development weeks, bucks, or some artificial unit, say Points. Next, it is determined which user stories will be realized in the current increment, and development proceeds. Usually, the time box agreed upon is sacred, i.e., if some of the user stories cannot be realized within the agreed upon time frame, they are moved to a next iteration. Estimation accuracy is assumed to improve in the course of the project.

7.1 Algorithmic Models

To be able to get reliable estimates, we need to extensively record historical data. These historical data can be used to produce estimates for new projects. In doing

so, we predict the expected cost on account of *measurable* properties of the project at hand. Just as the cost of laying out a garden might be a weighted combination of a number of relevant attributes (size of the garden, size of the grass area, yes/no for a pond), so we would like to estimate the cost of a software development project. In this section, we discuss efforts to get at algorithmic models to estimate software cost.

In the introduction to this chapter, we noticed that programming effort is strongly correlated with program size. There exist various (non-linear) models which express this correlation. A general form is

$$E = (a + bKLOC^c)f(x_1, \ldots, x_n)$$

Here, $KLOC$ again denotes the size of the software (lines of code/1000), while E denotes the effort in man-months. a, b and c are constants, and $f(x_1, \ldots, x_n)$ is a correction which depends on the values of the entities $x_1, \ldots, x_n$. In general, the base formula

$$E = a + bKLOC^c$$

is obtained through a regression analysis of available project data. Thus, the primary cost driver is software size, measured in lines of code. This nominal cost estimate is tuned by correcting it for a number of factors that influence productivity (so-called cost drivers). For instance, if one of the factors used is 'experience of the programming team', this could incur a correction to the nominal cost estimate of 1.50, 1.20, 1.00, 0.80 and 0.60 for a very low, low, average, high and very high level of expertise, respectively.

Figure 7.1 contains some of the well-known base formulae for the relation between software size and effort. For reasons mentioned before, it is difficult to compare these models. It is interesting to note, though, that the value of c fluctuates around the value 1 in most models.

Origin	Base formula	See section
Halstead	$E = 0.7KLOC^{1.50}$	12.1.4
Boehm	$E = 2.4KLOC^{1.05}$	7.1.2
Walston--Felix	$E = 5.2KLOC^{0.91}$	7.1.1

Figure 7.1 Some base formulae for the relation between size and effort

This phenomenon is well known from the theory of economics. In a so-called economy of scale, one assumes that it is cheaper to produce large quantities of the same product. The fixed costs are then distributed over a larger number of units, which

decreases the cost per unit. We thus realize an increasing return on investment. In the opposite case, we find a diseconomy of scale: after a certain point the production of additional units incurs extra costs.

In the case of software, the lines of code are the product. If we assume that producing a lot of code will cost less per line of code, formulae like those of Walston--Felix ($c < 1$) result. This may occur, for example, because the cost of expensive tools like program generators, programming environments and test tools can be distributed over a larger number of lines of code. Alternatively, we may reason that large software projects are more expensive, relatively speaking. There is a larger overhead because of the increased need for communication and management control, because of the problems and interfaces getting more complex, and so on. Thus, each additional line of code requires more effort. In such cases, we obtain formulae like those of Boehm and Halstead ($c > 1$).

There is no really convincing argument for either type of relation, though the latter ($c > 1$) may seem more plausible. Certainly for large projects, the effort required does seem to increase more than linearly with size.

It is clear that the value of the exponent c strongly influences the computed value E, certainly for large values of $KLOC$. Figure 7.2 gives the values for E, as they are computed for the earlier-mentioned models and some values for $KLOC$. The reader will notice large differences between the models. For small programs, Halstead's model yields the lowest cost estimates. For projects in the order of one million lines of code, this same model yields a cost estimate which is an order of magnitude higher than that of Walston--Felix.

$KLOC$	$E = 0.7KLOC^{1.50}$	$E = 2.4KLOC^{1.05}$	$E = 5.2KLOC^{0.91}$
1	0.7	2.4	5.2
10	22.1	26.9	42.3
50	247.5	145.9	182.8
100	700.0	302.1	343.6
1000	22135.9	3390.1	2792.6

Figure 7.2 E versus $KLOC$ for various base models

However, we should not immediately conclude that these models are useless. It is much more likely that there are big differences in the characteristics between the sets of projects on which the various models are based. Recall that the actual numbers used in those models result from an analysis of real project data. If these data reflect widely different project types or development environments, so will the models. We cannot simply copy those formulae. Each environment has its own specific characteristics and

tuning the model parameters to the specific environment (a process called calibration) is necessary.

The most important problem with this type of model is to get a *reliable* estimate of the software size early on. How should we estimate the number of pages in a novel not yet written? Even if we know the number of characters, the number of locations and the time interval in which the story takes place, we should not expect a realistic size estimate up front. The further advanced we are with the project, the more accurate our size estimate will get. If the design is more or less finished, we may (possibly) form a reasonable impression of the size of the resulting software. Only if the system has been delivered, do we know the exact number.

The customer, however, needs a reliable cost estimate early on. In such a case, lines of code is a measure which is too inexact to act as a base for a cost estimate. We therefore have to look for an alternative. In section 7.1.4 we discuss a model based on quantities which are known at an earlier stage.

We may also switch to another model during the execution of a project, since we may expect to get more reliable data as the project is making progress. We then get a cascade of increasingly detailed cost estimation models. COCOMO 2 is an example of this; see section 7.1.5.

7.1.1 Walston--Felix

The base equation of Walston and Felix' model (Walston and Felix, 1977) is

$$E = 5.2KLOC^{0.91}$$

Some 60 projects from IBM were used in the derivation of this model. These projects differed widely in size and the software was written in a variety of programming languages. It therefore comes as no surprise that the model, applied to a subset of these 60 projects, yields unsatisfactory results.

In an effort to explain these wide-ranging results, Walston and Felix identified 29 variables that clearly influenced productivity. For each of these variables, three levels were distinguished: high, average and low. For a number of projects (51) Walston and Felix determined the level of each of these 29 variables, together with the productivity obtained (in terms of lines of code per man-month) in those projects. These results are given in figure 7.3 for some of the most important variables. Thus, the average productivity turned out to be 500 lines of code per man-month for projects with a user interface of low complexity. With a user interface of average or high complexity, the productivity is 295 and 124 lines of code per man-month, respectively. The last column contains the productivity change PC, the absolute value of the difference between the high and low scores.

According to Walston and Felix, a productivity index I can now be determined for a new project, as follows:

$$I = \sum_{i=1}^{29} W_i X_i$$

Variable	Value of variable Average productivity (LOC)			| high − low | (PC)
Complexity of user interface	<normal 500	normal 295	>normal 124	376
User participation during requirements specification	none 491	some 267	much 205	286
User-originated changes in design	few 297	--	many 196	101
User-experience with application area	none 318	some 340	much 206	112
Qualification, experience of personnel	low 132	average 257	high 410	278
Percentage programmers participating in design	<25% 153	25 --50% 242	>50% 391	238
Previous experience with operational computer	minimal 146	average 270	extensive 312	166
Previous experience with programming languages	minimal 122	average 225	extensive 385	263
Previous experience with application of similar or greater size and complexity	minimal 146	average 221	extensive 410	264
Ratio of average team size to duration (people/month)	<0.5 305	0.5-0.9 310	>0.9 171	134

Figure 7.3 Some productivity intervals (*Source: C.E. Walston and C.P. Felix, A method for programming measurement and estimation,* IBM Systems Journal, *1977.*)

The weights W_i are defined by

$$W_i = 0.5 \log(PC_i)$$

Here, PC_i is the productivity change of factor i. For the first factor from figure 7.3 (complexity of the user interface), the following holds: $PC_1 = 376$, so $W_1 = 1.29$. The variables X_i can take on values $+1$, 0 and -1, where the corresponding factor scores as low, average or high (and thus results in a high, average or low productivity, respectively). The productivity index obtained can be translated into an expected productivity (lines of code produced per man-month). Details of the latter are not given in (Walston and Felix, 1977).

The number of factors considered in this model is rather high (29 factors out of 51 projects). Also it is not clear to what extent the various factors influence each other. Finally, the number of alternatives per factor is only three, and does not seem to offer enough choice in practical situations.

Nevertheless, the approach taken by Walston and Felix and their list of cost drivers have played a very important role in directing later research in this area.

7.1.2 COCOMO

COCOMO (COnstructive COst MOdel) is one of the algorithmic cost estimation models best documented. In its simplest form, called Basic COCOMO, the formula that relates effort to software size, reads

$$E = bKLOC^c$$

Here, b and c are constants that depend on the kind of project that is being executed. COCOMO distinguishes three classes of project:

- **Organic** A relatively small team develops software in a known environment. The people involved generally have a lot of experience with similar projects in their organization. They are thus able to contribute at an early stage, since there is no initial overhead. Projects of this type will seldom be very large projects.

- **Embedded** The product will be embedded in an environment which is very inflexible and poses severe constraints. An example of this type of project might be air traffic control, or an embedded weapon system.

- **Semidetached** This is an intermediate form. The team may show a mixture of experienced and inexperienced people, the project may be fairly large, though not excessively large, etc.

For the various classes, the parameters of Basic COCOMO take on the following values:

organic:	$b = 2.4, c = 1.05$
semidetached:	$b = 3.0, c = 1.12$
embedded:	$b = 3.6, c = 1.20$

Figure 7.4 gives the estimated effort for projects of each of those three modes, for different values of $KLOC$ (though an 'organic' project of one million lines is not very realistic). Amongst others, we may read from this figure that the constant c soon starts to have a major impact on the estimate obtained.

Basic COCOMO yields a simple, and hence a crude, cost estimate based on a simple classification of projects into three classes. In his book *Software Engineering*

KLOC	Effort in man-months organic ($E = 2.4KLOC^{1.05}$)	semidetached ($E = 3.0KLOC^{1.12}$)	embedded ($E = 3.6KLOC^{1.20}$)
1	2.4	3.0	3.6
10	26.9	39.6	57.1
50	145.9	239.4	392.9
100	302.1	521.3	904.2
1000	3390.0	6872.0	14333.0

Figure 7.4 Size versus effort in Basic COCOMO

Economics, Boehm also discusses two other, more complicated, models, termed Intermediate COCOMO and Detailed COCOMO, respectively. Both these models take into account 15 cost drivers --attributes that affect productivity, and hence costs.

All these cost drivers yield a multiplicative correction factor to the nominal estimate of the effort. (Both these models also use values for b which slightly differ from that of Basic COCOMO.) Suppose we found a nominal effort estimate of 40 man-months for a certain project. If the complexity of the resulting software is low, then the model tells us to correct this estimate by a factor of 0.85. A better estimate then would be 34 man-months. On the other hand, if the complexity is high, we get an estimate of $1.15 \times 40 = 46$ man-months.

The nominal value of each cost driver in Intermediate COCOMO is 1.00 (see also figure 7.11. So we may say that Basic COCOMO is based on nominal values for each of the cost drivers.

On top of this set of cost drivers, the detailed model adds a further level of refinement. First of all, this model is phase-sensitive, the idea being that not all cost drivers influence each phase of the development cycle in the same way. So, rather than having one table with effort multipliers as in Intermediate COCOMO, Detailed COCOMO uses a set of such tables. These tables show, for each cost driver, a separate effort multiplier for each major development phase. Furthermore, Detailed COCOMO uses a hierarchy for the product to be developed, in which some cost drivers have an impact on the estimate at the module level, while others have an impact at the (sub)system level.

The COCOMO formulae are based on a combination of expert judgment, an analysis of available project data, other models, etc. The basic model does not yield very accurate results for the projects on which the model has been based. The intermediate version yields good results and, if one extra cost driver (volatility of the requirements specification) is added, it even yields very good results. Further validation of the COCOMO models using other project data is not straightforward,

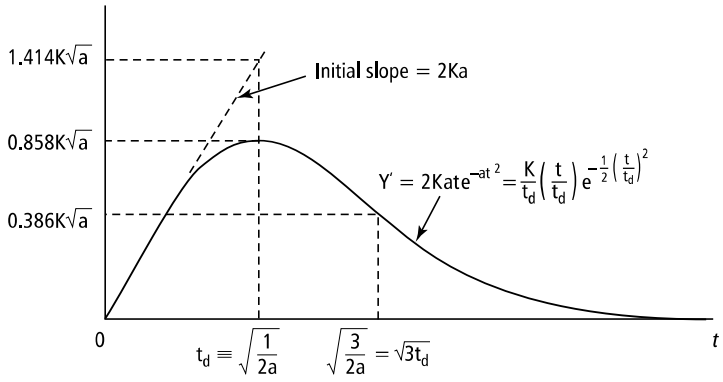

Figure 7.5 The Rayleigh-curve for software schedules (*Source: M.L. Shooman*, Tutorial on software cost models, *IEEE Catalog nr TH0067-9 (1979), 1979 IEEE.*)

since the necessary information to determine the ratings of the various cost drivers is in general not available. So we are left with the possibility of only testing the basic model. Here, we obtain fairly large discrepancies between the effort estimated and the actual effort needed.

A major advantage of COCOMO is that we know all its details. A major update of the COCOMO model, better reflecting current and future software practices, is discussed in section 7.1.5.

7.1.3 Putnam

Norden studied the distribution of manpower over time in a number of software development projects in the 1960s. He found that this distribution often had a very characteristic shape which is well-approximated by a Rayleigh distribution. Based upon this finding, Putnam developed a cost estimation model in which the manpower required (MR) at time t is given by

$$MR(t) = 2Kate^{-at^2}$$

a is a speed-up factor which determines the initial slope of the curve, while K denotes the total manpower required, including the maintenance phase. K equals the volume of the area delineated by the Rayleigh curve (see figure 7.5).

The shape of this curve can be explained theoretically as follows. Suppose a project consists of a number of problems for which a solution must be found. Let $W(t)$ be the fraction of problems for which a solution has been found at time t. Let $p(t)$ be the problem-solving capacity at time t. Progress at time t then is proportional

to the product of the available problem-solving capacity and the fraction of problems yet unsolved. If the total amount of work to be done is set to 1, this yields:

$$\frac{\mathrm{d}W}{\mathrm{d}t} = p(t)(1 - W(t))$$

After integration, we get

$$W(t) = 1 - \exp(-\int^{t} p(\alpha)\mathrm{d}\alpha)$$

If we next assume that the problem-solving capacity is well approximated by an equation of the form $p(t) = at$, i.e. the problem-solving capacity shows a linear increase over time, the progress is given by a Rayleigh distribution:

$$\frac{\mathrm{d}W}{\mathrm{d}t} = ate^{-(at^2)/2}$$

Integration of the equation for $MR(t)$ that was given earlier yields the cumulative effort I:

$$I(t) = K(1 - e^{-at^2})$$

In particular, we get $I(\infty) = K$. If we denote the point in time at which the Rayleigh-curve assumes its maximum value by T, then $a = 1/(2T^2)$. This point T will be close to the point in time at which the software is being delivered to the customer. The volume of the area delineated by the Rayleigh curve between points 0 and T then is a good approximation of the initial development effort. For this, we get

$$E = I(T) = 0.3945K$$

This result is remarkably close to the often-used rule of thumb: 40% of the total effort is spent on the actual development, while 60% is spent on maintenance.

Various studies indicate that Putnam's model is well suited to estimating the cost of very large software development projects (projects that involve more than 15 man-years). The model seems to be less suitable for small projects.

A serious objection to Putnam's model, in our opinion, concerns the relation it assumes between effort and development time if the schedule is compressed relative to the nominal schedule estimate: $E = c/T^4$. Compressing a project's schedule in this model entails an extraordinary large penalty (see also section 7.3).

7.1.4 Function Point Analysis

Function point analysis (FPA) is a method of estimating costs in which the problems associated with determining the expected amount of code are circumvented. FPA is based on counting the number of different data structures that are used. In the FPA method, it is assumed that the number of different data structures is a good

size indicator. FPA is particularly suitable for projects aimed at realizing business applications for, in these applications, the structure of the data plays a very dominant role. The method is less suited to projects in which the structure of the data plays a less prominent role, and the emphasis is on algorithms (such as compilers and most real-time software).

The following five entities play a central role in the FPA-model:

- Number of input types (I). The input types refer only to user input that results in changes in data structures. It does not concern user input which is solely concerned with controlling the program's execution. Each input type that has a different format, or is treated differently, is counted. So, though the records of a master file and those of a mutation file may have the same format, they are still counted separately.

- Number of output types (O). For the output types, the same counting scheme is used.

- Number of inquiry types (E). Inquiry types concern input that controls the execution of the program and does not change internal data structures. Examples of inquiry types are: menu selection and query criteria.

- Number of logical internal files (L). This concerns internal data generated by the system, and used and maintained by the system, such as, for example, an index file.

- Number of interfaces (F). This concerns data that is output to another application, or is shared with some other application.

By trial and error, weights have been associated with each of these entities. The number of (unadjusted) function points, UFP, is a weighted sum of these five entities:

$$UFP = 4I + 5O + 4E + 10L + 7F$$

With FPA too, a further refinement is possible, by applying corrections to reflect differences in complexity of the data types. In that case, the constants used in the above formula depend on the estimated complexity of the data type in question. Figure 7.6 gives the counting rules when three levels of complexity are distinguished. So, rather than having each input type count as four function points, we may count three, four or six function points, based on an assessment of the complexity of each input type.

Each input type has a number of data element types (attributes), and refers to zero or more other file types. The complexity of an input type increases as the number of its data element types or referenced file types increases. For input types, the mapping of these numbers to complexity levels is given in figure 7.7. For the other file types, these tables have the same format, with slightly different numbers along the axes.

Type	Complexity level		
	Simple	Average	Complex
Input (I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Logical internal (L)	7	10	15
Interfaces (F)	5	7	10

Figure 7.6 Counting rules for (unadjusted) function points

# of file types	# of data elements		
	1 --4	5 --15	>15
0 or 1	simple	simple	average
2 --3	simple	average	complex
>3	average	complex	complex

Figure 7.7 Complexity levels for input types

As in other cost estimation models, the unadjusted function point measure is adjusted by taking into account a number of application characteristics that influence development effort. Figure 7.8 contains the 14 characteristics used in the FPA model. The degree of influence of each of these characteristics is valued on a six-point scale, ranging from zero (no influence, not present) to five (strong influence). The total degree of influence DI is the sum of the scores for all characteristics. This number is then converted to a technical complexity factor (TCF) using the formula

$$TCF = 0.65 + 0.01DI$$

The (adjusted) function point measure FP is now obtained through

$$FP = UFP \times TCF$$

Finally, there is a direct mapping from (adjusted) function points to lines of code. For instance, in (Albrecht, 1979) one function point corresponds to 65 lines of PL/I, or to 100 lines of COBOL, on average.

In FPA, it is not simple to decide exactly when two data types should be counted as separate. Also, the difference between, for example, input types, inquiry types, and

Data communications
Distributed functions
Performance
Heavily used configuration
Transaction rate
Online data entry
End-user efficiency
Online update
Complex processing
Re-usability
Installation ease
Operational ease
Multiple sites
Facilitate change

Figure 7.8 Application characteristics in FPA

interfaces remains somewhat vague. The International Function Point User Group (IFPUG) has published extensive guidelines on how to classify and count the various entities involved. This should overcome many of the difficulties that analysts have in counting function points in a uniform way.

Further problems with FPA have to do with its use of ordinal scales and the way complexity is handled. FPA distinguishes three levels of component complexity only. A component with 100 elements thus gets at most twice the number of function points of a component with one element. It has been suggested that a model which uses the raw complexity data, i.e. the number of data elements and file types referenced, might work as well as, or even better than, a model which uses an ordinal derivative thereof. In a sense, complexity is counted twice: both through the complexity level of the component and through one of the application characteristics. Yet it is felt that highly complex systems are not adequately dealt with, since FPA is predominantly concerned with counting externally visible inputs and outputs.

In applying the FPA cost estimation method, it still remains necessary to calibrate the various entities to your own environment. This holds the more for the corrections that reflect different application characteristics, and the transition from function points to lines of code.

7.1.5 COCOMO 2: Variations on a Theme

COCOMO 2 is a revision of the 1981 COCOMO model, tuned to the life cycle practices of the 1990s and 2000s. It reflects our cumulative experience with and

knowledge of cost estimation. By comparing its constituents with those of previous cost estimation models, it also offers us a means to learn about significant changes in our trade over the past decades.

COCOMO 2 provides three increasingly detailed cost estimation models. These models can be used for different types of projects, as well as during different stages of a single project:

- the **Application Composition** model, mainly intended for prototyping efforts, for instance to resolve user interface issues (Its name suggests heavy use of existing components, presumably in the context of a powerful CASE environment.)
- the **Early Design** model, aimed at the architectural design stage
- the **Post-Architecture** model for the actual development stage of a software product

The Post-Architecture model can be considered an update of the original COCOMO model; the Early Design model is an FPA-like model; and the Application Composition model is based on counting system components of a large granularity, such as screens and reports.

The Application Composition model is based on counting Object Points. Object Points have nothing to do with objects as in object-oriented development. In this context, objects are screens, reports, and 3GL modules.

The roots of this type of model can be traced back to several variations on FPA-type size measures. Function points as used in FPA are intended to be a user-oriented measure of system function. The user functions measured are the inputs, outputs, inquiries, etc. We may conjecture that these user-functions are technology-dependent, and that FPA primarily reflects the batch-oriented world of the 1970s.

Present-day administrative systems are perhaps better characterized by their number of menus or screens. This line of thought has been pursued in various studies. Banker et al. (1991) compared Object Points with Function Points for a sample of software projects, and found that Object Points did almost as well as Function Points. Object Points, however, are easier to determine, and at an earlier point in time.

Total effort is estimated in the Application Composition model as follows:

1. Estimate the number of screens, reports, and 3GL components in the application.
2. Determine the complexity level of each screen and report (simple, medium or difficult). 3GL components are assumed to be always difficult. The complexity of a screen depends on the number of views and tables it contains. The complexity of a report depends on the number of sections and tables it contains. A classification table similar to those in FPA (see figure 7.9 for an example) is used to determine these complexity levels.

3. Use the numbers given in figure 7.10 to determine the relative effort (in Object Points) to implement the object.

4. The sum of the Object Points for the individual objects yields the number of Object Points for the whole system.

5. Estimate the reuse percentage, resulting in the number of New Object Points (NOP) as follows: $NOP = ObjectPoints \times (100 - \%Reuse)/100$.

6. Determine a productivity rate $PROD = NOP/$*man-month*. This productivity rate depends on the experience and capability of both the developers and the maturity of the CASE environment they use. It varies from 4 (very low) to 50 (very high).

7. Estimate the number of man-months needed for the project: $E = NOP/PROD$.

# of views	# and source of data tables		
	total < 4 (< 2 on server < 3 on client)	total < 8 ($2-3$ on server $3-5$ on client)	total ≥ 8 (> 3 on server > 5 on client)
< 3	simple	simple	medium
$3-7$	simple	medium	difficult
> 8	medium	difficult	difficult

Figure 7.9 Complexity levels for screens

Object type	Complexity		
	simple	medium	difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Figure 7.10 Counting Object Points

The Early Design model uses unadjusted function points (UFPs) as its basic size measure. These unadjusted function points are counted in the same way they are counted in FPA. Next, the unadjusted function points are converted to Source Lines Of Code (SLOC), using a ratio SLOC/UFP which depends on the programming language used. In a typical environment, each UFP may correspond to, say, 91 lines of Pascal, 128 lines of C, 29 lines of C++, or 320 lines of assembly language. Obviously, these numbers are environment-specific.

The Early Design model does not use the FPA scheme to account for application characteristics. Instead, it uses a set of seven cost drivers, which are a combination of the full set of cost drivers of the Post-Architecture model. The intermediate, reduced set of cost drivers is:

- product reliability and complexity, which is a combination of the required software reliability, database size, product complexity and documentation needs cost drivers
- required reuse, which is equivalent to its Post-Architecture counterpart
- platform difficulty, which combines execution time, main storage constraints, and platform volatility
- personnel experience, which combines application, platform, and tool experience
- personnel capability, which combines analyst and programmer capability and personnel continuity.
- facilities, which is a combination of the use of software tools and multi-site development
- schedule, which again equals its Post-Architecture counterpart

These cost drivers are rated on a seven-point scale, ranging from extra low to extra high. The values assigned are similar to those in figure 7.11. Thus, the nominal values are always 1.00, and the values become larger or smaller as the cost driver is estimated to deviate further from the nominal rating. After the unadjusted function points have been converted to Kilo Source Lines Of Code ($KSLOC$), the cumulative effect of the cost drivers is accounted for by the formula

$$E = KSLOC \times \prod_i \mathit{cost\ driver}_i$$

Finally, the Post-Architecture model is the most detailed model. Its basic effort equation is very similar to that of the original COCOMO model:

$$E = a \times KSLOC^b \times \prod_i \mathit{cost\ driver}_i$$

It differs from the original COCOMO model in its set of cost drivers, the use of lines of code as its base measure, and the range of values of the exponent b.

The differences between the COCOMO and COCOMO 2 set of cost drivers reflect major changes in the field. The set of COCOMO 2 cost drivers and the associated effort multipliers are given in figure 7.11. The values of the effort multipliers in this figure are the result of calibration on a certain set of projects. The changes are as follows:

- Four new cost drivers have been introduced: required reusability, documentation needs, personnel continuity, and multi-site development. They reflect the growing influence of the corresponding aspects on development cost.
- Two cost drivers have been dropped: computer turnaround time and use of modern programming practices. Nowadays, developers use workstations and (batch-processing) turnaround time is no longer an issue. Modern programming practices have evolved into the broader notion of mature software engineering practices, which are dealt with in the exponent b of the COCOMO 2 effort equation.
- The productivity influence, i.e. the ratio between the highest and lowest value, of some cost drivers has been increased (analyst capability, platform experience, language and tools experience) or decreased (programmer capability).

In COCOMO 2, the user may use both $KSLOC$ and UFP as a base measure. It is also possible to use UFP for part of the system. The UFP counts are converted to $KSLOC$ counts as in the Early Design model, after which the effort equation applies.

Rather than having three 'modes', with slightly different values for the exponent b in the effort equation, COCOMO 2 has a much more elaborate scaling model. This model uses five scale factors W_i, each of which is rated on a six-point scale from very low (5) to extra high (0). The exponent b for the effort equation is then determined by the formula:

$$b = 1.01 + 0.01 \times \sum_i W_i$$

So, b can take on values in the range 1.01 to 1.26, thus giving a more flexible rating scheme than that used in the original COCOMO model.

The scale factors used in COCOMO 2 are:

- precedentedness, indicating the novelty of the project to the development organization. Aspects like the experience with similar systems, the need for innovative architectures and algorithms, and the concurrent development of hardware and software are reflected in this factor.
- development flexibility, reflecting the need for conformance with pre-established and external interface requirements, and a possible premium on early completion.

- architecture/risk resolution, which reflects the percentage of significant risks that have been eliminated. In many cases, this percentage will be correlated with the percentage of significant module interfaces specified, i.e. architectural choices made.

- team cohesion, accounting for possible difficulties in stakeholder interactions. This factor reflects aspects like the consistency of stakeholder objectives and cultures, and the experience of the stakeholders in acting as a team.

- process maturity, reflecting the maturity of the project organization according to the Capability Maturity Model (see section 6.6).

Only the first two of these factors were, in a crude form, accounted for in the original COCOMO model.

The original COCOMO model allows us to handle reuse in the following way. The three main development phases, design, coding and integration, are estimated to take 40%, 30% and 30% of the average effort, respectively. Reuse can be catered for by separately considering the fractions of the system that require redesign (DM), recoding (CM) and re-integration (IM). An adjustment factor AAF is then given by the formula

$$AAF = 0.4DM + 0.3CM + 0.3IM$$

An adjusted value $AKLOC$, given by

$$AKLOC = KLOC \times AAF/100$$

is next used in the COCOMO formulae, instead of the unadjusted value $KLOC$. In this way a lower cost estimate is obtained if part of the system is reused.

By treating reuse this way, it is assumed that developing reusable components does not require any extra effort. You may simply reap the benefits when part of a system can be reused from an earlier effort. This assumption does not seem to be very realistic. Reuse does not come for free (see also chapter ??).

COCOMO 2 uses a more elaborate scheme to handle reuse effects. This scheme reflects two additional factors that impact the cost of reuse: the quality of the code being reused and the amount of effort needed to test the applicability of the component to be reused.

If the software to be reused is strongly modular, strongly matches the application in which it is to be reused, and the code is well-organized and properly documented, then the extra effort needed to reuse this code is relatively low, and estimated to be 10%. This penalty may be as high as 50% if the software exhibits low coupling and cohesion, is poorly documented, and so on. This extra effort is denoted by the software understanding increment SU.

The degree of assessment and assimilation (AA) denotes the effort needed to determine whether a component is appropriate for the present application. It

Cost drivers	Rating					
	Very low	Low	Nominal	High	Very high	Extra high
Product factors						
Reliability required	0.75	0.88	1.00	1.15	1.39	
Database size		0.93	1.00	1.09	1.19	
Product complexity	0.75	0.88	1.00	1.15	1.30	1.66
Required reusability		0.91	1.00	1.14	1.29	1.49
Documentation needs	0.89	0.95	1.00	1.06	1.13	
Platform factors						
Execution time constraints			1.00	1.11	1.31	1.67
Main storage constraints			1.00	1.06	1.21	1.57
Platform volatility		0.87	1.00	1.15	1.30	
Personnel factors						
Analyst capability	1.50	1.22	1.00	0.83	0.67	
Programmer capability	1.37	1.16	1.00	0.87	0.74	
Application experience	1.22	1.10	1.00	0.89	0.81	
Platform experience	1.24	1.10	1.00	0.92	0.84	
Language and tool experience	1.25	1.12	1.00	0.88	0.81	
Personnel continuity	1.24	1.10	1.00	0.92	0.84	
Project factors						
Use of software tools	1.24	1.12	1.00	0.86	0.72	
Multi-site development	1.25	1.10	1.00	0.92	0.84	0.78
Required development schedule	1.29	1.10	1.00	1.00	1.00	

Figure 7.11 Cost drivers and associated effort multipliers in COCOMO 2 (*Source: B.W. Boehm et al.*, COCOMO II Model Definition Manual, *University of Southern California, 1997.*)

ranges from 0% (no extra effort required) to 8% (extensive test, evaluation and documentation required).

Both these percentages are added to the adjustment factor AAF, yielding the equivalent kilo number of new lines of code, $EKLOC$:

$$EKLOC = KLOC \times (AAF + SU + AA)/100$$

7.2 Guidelines for Estimating Cost

The models discussed in the preceeding section are based on data about past projects. One of the main problems in applying these models is the sheer *lack* of quantitative data about past projects. There simply is not enough data available. Though the importance of such a database is now widely recognized we still do not routinely collect data on current projects. It seems as if we cannot spare the time to collect data; we have to write software. DeMarco (1982) makes a comparison with the medieval barber who also acted as a physician. He could have made the same objection: 'We cannot afford the time to take our patient's temperature, since we have to cut his hair.'

We thus have to shift to other methods to estimate costs. These other methods are based on the expertise of the estimators. In doing so, certain traps have to be circumvented. It is particularly important to prevent political arguments from entering the arena. Typical lines of reasoning that reflect political reasoning are:

- We were given 12 months to do the job, so it will take 12 months. This might be seen as a variation of Parkinson's Law: work fills the time available.
- We know that our competitor put in a bid of $1M, so we need to schedule a bid of $0.9M. This is sometimes referred to as 'price to win'.
- We want to show our product at the trade show next year, so the software needs to be written and tested within the next nine months, though we realize that this is rather tight. This could be termed the 'budget' method of cost estimation.
- Actually, the project needs one year, but I can't sell that to my boss. We know that ten months is acceptable, so we settle for ten months.

Politically-colored estimates can have disastrous effects, as has been shown all too often during the short history of our field. Political arguments almost always play a role if estimates are being given by people directly involved in the project, such as the project manager, or someone reporting to the project manager. Very soon, then, estimates will influence, or be influenced by, the future assessment of those persons. To quote DeMarco (1982): "one chief villain is the policy that estimates shall be used to create incentives."

Jørgensen (2005) gives the following guidelines for expert-based effort estimation:

- Do not mix estimation, planning, and bidding,
- Combine estimation methods,
- Ask for justification,
- Select experts with experience from similar projects,
- Accept and assess uncertainty,

- Provide learning opportunities, and
- Consider postponing or avoiding effort estimation.

The politically-colored estimation methods mentioned above all mix up estimation, planning and bidding. These have different goals, though. Estimation's only goal is accuracy. Planning involves risk assessment and schedule. Bidding is about winning a contract. Though these activities have different goals, they are of course related. A low bid for instance generally incurs a tight schedule and higher risks.

An interesting experiment on the effects of bidding on the remainder of a project is described in (Jørgensen and Grimstad, 2004). In this experiment, the authors study what is called *the winner's curse*, a phenomenon known from auctions, where players are uncertain of the value of a good when they bid. The highest bid wins, but the winner may be left with an item that's worth less than paid for. The term was first coined in the 1950s, when oil industries had no accurate way to estimate the value of their oil fields. In the software field, it has the following characteristics:

- Software providers differ in optimism in their estimates of most likely cost: some are over-optimistic, some are realistic, and some are pessimistic.
- Software providers with over-optimistic estimates tend to have the lowest bids.
- Software clients require a fixed-price contract.
- Software clients tend to select a provider with a low bid.

The result often is a Pyrrhic victory, a contract that results in low or negative profits to the bidder. But such a contract might also be risky for the client. Jørgensen and Grimstad (2004) describe an experiment in which they actually asked 35 companies for bids on a certain requirements specification. Next, four companies were asked to implement the system. They found that the companies with the lowest bids incurred the greatest risks.

Vacuuming a rug in two orthogonal directions is likely to pick up more dirt than vacuum that rug twice in the same direction. Likewise, the combination of sufficiently different estimation methods gives better estimates. So one may combine a COCOMO estimate with that of an expert, or estimates from experts with a different background. In this way, the bias that is inherent in a method or class of experts is mitigated.

Estimators should be held accountable for their estimates. Lederer and Prasad (2000) found that the use of estimates in performance evaluations of software managers and professionals is the only practice that leads to better estimates. In a slightly weaker form, one may at least ask for a justification of the estimate. Such a justification could refer to a calibrated model used, or a work breakdown structure in which cost estimates of components are derived from those in similar projects.

For lack of hard data, the cost of a software development project is often estimated through a comparison with earlier projects. If the estimator is very

experienced, reasonable cost estimates may result. However, the learning effect of earlier experiences may lead to estimates that are too pessimistic in this case. We may expect that experience gained with a certain type of application leads to a higher productivity for subsequent projects. Similar applications thus give rise to lower costs.

(McClure, 1968) describes a situation in which a team was asked to develop a FORTRAN compiler for three different machines. The effort needed (in man-months) for these three projects is given in figure 7.12.

Compiler	Number of man-months needed
1	72
2	36
3	14

Figure 7.12 Learning effect in writing a FORTRAN compiler

On the other hand, peculiar circumstances and particular characteristics of a specific project tend to get insufficient attention if cost is estimated through comparison with earlier projects. For example, a simple change of scale (automation of a local library with 25 000 volumes as opposed to a university library with over 1 000 000 volumes), slightly harsher performance requirements, a compressed schedule (which incurs a larger team and thus increases overhead because of communication) may have a significant impact on the effort required in terms of man-months.

Careless application of the comparison method of cost estimation leads to estimates like: the cost of this project is equal to the cost of the previous project.

We may also involve more than one expert in the estimation process. In doing so, each expert gives an estimate based on his own experience and expertise. Factors that are hard to quantify, such as personality characteristics and peculiar project characteristics, may thus be taken into account. Here too, the quality of the estimate cannot exceed the quality of the experts. The experts that participate in the estimate then should have experience in simliar projects. It does not help all that much to ask advice from an expert in office automation type systems to provide an estimate for an air-traffic control system.

Estimates incur uncertainty. A cost estimate of, say, 100 man months might mean that there is a 75% probability that the real cost of this project is between 80 and 120 manmonths. It is *not* a point estimate. One method that aims to get a more reliable estimate is to have the expert produce more than one estimate. We all have the tendency to conceive an optimistic estimate as being realistic. (Have you ever heard of a software system that got delivered ahead of time?) To obviate this tendency, we may employ a technique in which the expert is asked for three estimates: an

optimistic estimate a, a realistic estimate m, and a pessimistic estimate b. Using a beta-distribution, the expected effort then is $E = (a + 4m + b)/6$. Though this estimate will probably be better than the one simply based on the average of a and b, it seems justified to warn against too much optimism. Software has the tendency to grow, and projects have the tendency to far exceed the estimated effort.

Training improves performance. This holds for skaters as well as software cost estimators. Studies in other fields show that inexperienced people tend to overestimate their abilities and performance. There is no reason to expect the software field to be any different. The resulting cost and schedule overruns are all to common. Harrison (2004) suggests that a prime reason for more mature organizations to have fewer cost overruns is not so much higher productivity or better processes, but greater self-knowledge. I concur the same is true for people estimating software cost.

While executing a task, people have to make a number of decisions. These decisions are strongly influenced by requirements set or proposed. The cost estimate is one such requirement which will have an impact on the end result. We may imagine a hypothetical case in which model A estimates the cost at 300 man-months. Now suppose the project actually takes 400 man-months. If model B would have estimated the project at 450 man-months, is model B better than model A? It is quite possible that, starting from the estimate given by model B, the eventual cost would have been 600 man-months. The project's behavior is also influenced by the cost estimate. Choices made during the execution of a project are influenced by cost estimates derived earlier on. If a cost estimate is not needed, it is wise not making one either.

7.3 Distribution of Manpower over Time

Having obtained an estimate of the total number of man-months needed for a given project, we are still left with the question of how many calendar months it will take. For a project estimated at 20 man-months, the kind of schedules you might think of, include:

- 20 people work on the project for 1 month;
- 4 people work on the project for 5 months;
- 1 person works on the project for 20 months.

These are not realistic schedules. We noticed earlier that the manpower needed is not evenly distributed over the time period of the project. From the shape of the Rayleigh curve we find that we need a slowly increasing manpower during the development stages of the project.

Cost estimation models generally provide us with an estimate of the development time (schedule) T as well. Contrary to the effort equations, the various models show a remarkable consistency when it comes to estimating the development time, as is shown in figure 7.13.

Walston--Felix	$T = 2.5E^{0.35}$
COCOMO (organic)	$T = 2.5E^{0.38}$
COCOMO 2 (nominal schedule)	$T = 3.0E^{0.33+0.2\times(b-1.01)}$
Putnam	$T = 2.4E^{1/3}$

Figure 7.13 Relation between development time and effort

The values T thus computed represent nominal development times. It is worthwhile studying ways to shorten these nominal schedules. Obviously, shortening the development time means an increase in the number of people involved in the project.

In terms of the Rayleigh curve model, shortening the development time amounts to an increase of the value a, the speed-up factor which determines the initial slope of the curve. The peak of the Rayleigh curve then shifts to the left and at the same time it shifts up. We thus get a faster increase of manpower required at the start of the project and a higher maximum workforce.

Such a shift does not go unpunished. Different studies show that individual productivity decreases as team size grows. There are two major causes of this phenomenon:

- As the team gets larger, the communication overhead increases, since more time will be needed for consultation with other team members, tuning of tasks, and the like.
- If manpower is added to a team during the execution of a project, the total team productivity decreases at first. New team members are not productive right from the start. At the same time, they require time from the other team members during their learning process. Taken together, this causes a decrease in total productivity.

The combination of these two observations leads to the phenomenon that has become known as Brooks' Law: Adding manpower to a late project only makes it later.

By analyzing a large amount of project data, Conte *et al.* found the following relation between average productivity L (measured in lines of code per man-month) and average team size P (Conte et al., 1986):

$$L = 777P^{-0.5}$$

In other words, individual productivity decreases exponentially with team size.

A theoretical underpinning hereof can be given on account of Brooks' observation regarding the number of communication links between the people involved in a project. This number is determined by the size and structure of the team. If, in a team of size P, each member has to coordinate his activities with those of all

other members, the number of communication links is $P(P-1)/2$. If each member needs to communicate with one other member only, this number is $P-1$. Less communication than that seems unreasonable, since we would then have essentially independent teams. (If we draw team members as nodes of a graph and communication links as edges, we expect the graph to be connected.)

The number of communication links thus varies from roughly P to roughly $P^2/2$. In a true hierarchical organization, this leads to P^α communication paths, with $1 < \alpha < 2$.

For an individual team member, the number of communication links varies from 1 to $P-1$. If the maximum individual productivity is L and each communication link results in a productivity loss l, the average productivity is

$$L_\gamma = L - l(P-1)^\gamma$$

where γ, with $0 < \gamma \leq 1$, is a measure of the number of communication links. (We assume that there is at least one person who communicates with more than one other person, so $\gamma > 0$.) For a team of size P, this leads to a total productivity

$$L_{tot} = P \times L_\gamma = P(L - l(P-1)^\gamma)$$

For a given set of values for L, l and γ, this is a function which, for increasing values of P, goes from 0 to some maximum and then decreases again. There thus is a certain optimum team size P_{opt} that leads to a maximum team productivity. The team productivity for different values of the P is given in figure 7.14. Here, we assume that individual productivity is 500 LOC/man-month ($L = 500$), and the productivity loss is 10% per communication link ($l = 50$). With full interaction between team members ($\gamma = 1$) this results in an optimum team size of 5.5 persons.

Everything takes time. We can not shorten a software development project indefinitely by exchanging time against people. Boehm sets the limit at 75% of the nominal development time, on empirical grounds. A system that has to be delivered too fast, gets into the 'impossible region'. The chance of success becomes almost nil if the schedule is pressed too far. See also figure 7.15.

In any case, a shorter development time induces higher costs. We may use the following rule of thumb: compressing the development time by X% results in a cost increase of X% relative to the nominal cost estimate (Boehm, 1984a).

7.4 Summary

It remains to be seen whether we will ever get one, general, cost estimation model. The number of parameters that impact productivity simply seems to be too large. Yet, each organization may develop a model which is well suited for projects to be undertaken within that organization. An organization may, and should, build a database with data on its own projects. Starting with a model like COCOMO 2, the different parameters, i.e. applicable cost drivers and values for the associated

Team size	Individual productivity	Total productivity
1	500	500
2	450	900
3	400	1200
4	350	1400
5	300	1500
5.5	275	1512
6	250	1500
7	200	1400
8	150	1200

Figure 7.14 Impact of team size on productivity

effort multipliers, may then be determined. In the course of time, the model becomes more closely tuned to the organizational environment, resulting in better and better estimates. Reifer (2000) for example describes how COCOMO II can be adapted to estimate Web-based software development projects.

Though we advocate the use of algorithmic cost estimation models, a word of caution should be made. Present-day models of this kind are not all that good yet. At best, they yield estimates which are at most 25% off, 75% of the time, *for projects used to derive the model*. For the time being, expert-based cost estimates are a viable alternative.

Even when a much better performance is realized, some problems remain when using the type of cost estimation model obtained in this way:

- Even though a model like COCOMO 2 looks objective, a fair amount of subjectivity is introduced through the need to assign values to the various levels of a number of cost drivers. Based on an analysis of historical project data, (Jones, 1986) lists 20 factors which certainly influence productivity, and another 25 for which it is probable. The set of COCOMO 2 cost drivers already allows for a variation of 1:800. A much smaller number of relevant cost drivers would reduce a model's vulnerability to the subjective assessment of project characteristics.

- The models are based on data from *old* projects and reflect the technology of those projects. In some cases, the project data are even fairly old. The impact of more recent developments cannot easily be taken into account, since we do not have sufficient data on projects which exhibit those characteristics.

- Almost all models take into account attributes that impact the initial development of software. Attributes which specifically relate to maintenance activities

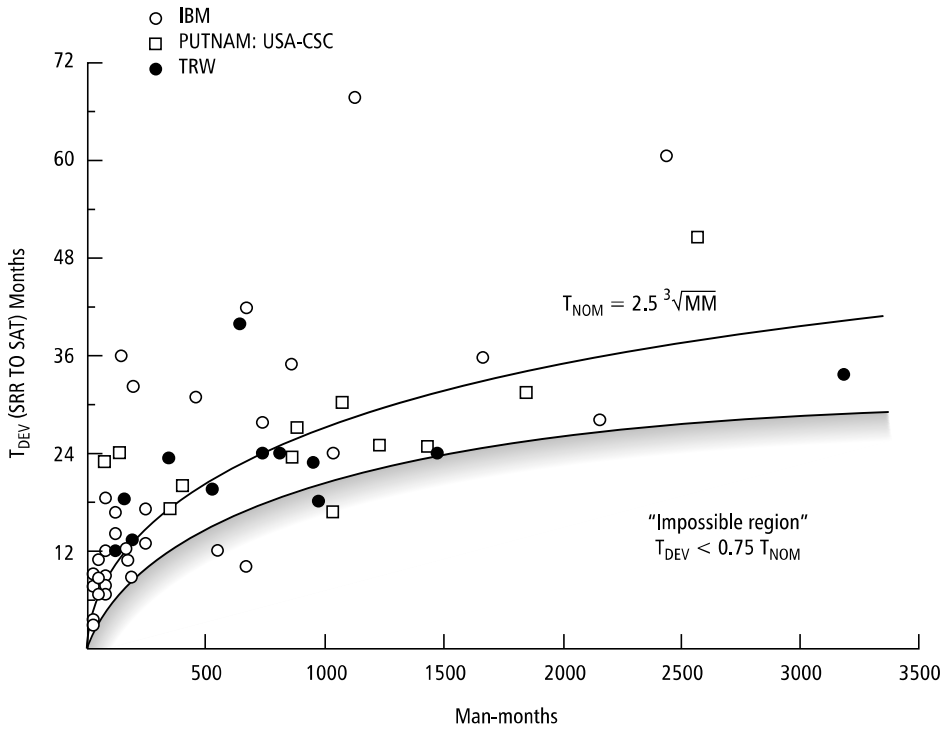

Figure 7.15 The impossible region (*Source: B.W. Boehm,* Software Engineering Economics, *fig. 27-8/page 471, 1981, Reprinted by permission of Prentice-Hall, Inc., Englewood Cliffs, NJ*)

are seldom taken into account. Also, factors like the amount of documentation required, the number of business trips (in case of multisite development) are often lacking. Yet, these factors may have a significant impact on the effort needed.

Algorithmic models usually result from applying statistical techniques like regression analysis to a given set of project data. For a new project, the parameters of the model have to be determined, and the model yields an estimate and, in some cases, a confidence interval.

A problem of a rather different nature is the following: In the introduction to this chapter we compared software cost estimation with cost estimation for laying out a garden. When laying out a garden, we often follow a rather different line of thought, namely: given a budget of, say, $10 000, what possibilities are there? What happens

if we trade off a pond against something else?

Something similar is also possible with software. Given a budget of $100 000 for library automation, what possibilities are there? Which user interface can we expect, what will the transaction speed be, how reliable will the system be? To be able to answer this type of question, we need to be able to analyze the sensitivity of an estimate to varying values of relevant attributes. Given the uncertainty about which attributes are relevant to start with, this trade-off problem is still largely unsolved.

Finally, estimating the cost of a software development project is a highly dynamic activity. Not only may we switch from one model to another during the course of a project, estimates will also be adjusted on the basis of experiences gained. Switching to another model during the execution of a project is possible, since we may expect to get more reliable data while the project is making progress. We may, for instance, imagine using the series of increasingly detailed COCOMO 2 models.

We cannot, and should not, rely on a one-shot statistical cost estimate. Controlling a software development project implies a regular check of progress, a regular check of estimates made, re-establishing priorities and weighing stakes, as the project is going on.

7.5 Further Reading

Early cost estimation models are described in (Nelson, 1966) and (Wolverton, 1974). The Walston--Felix model is described in (Walston and Felix, 1977). The model of Putnam and Norden is described in (Norden, 1970), (Putnam, 1978). (Boehm, 1981) is the definitive source on the original COCOMO model. COCOMO 2 is described in (Boehm et al., 1995) and (Boehm *et al.*, 1997).

Function point analysis (FPA) is developed by Albrecht (Albrecht, 1979; Albrecht and Gaffney, 1983). Critical appraisals of FPA can be found in (Symons, 1988), (Kemerer, 1993), (Kemerer and Porter, 1992), (Abran and Robillard, 1992) and (Abran and Robillard, 1996). A detailed discussion of function points, its counting process and some case studies, is provided by (Garmus and Herron, 1996).

The relation between project behavior and its cost estimate is discussed in (Abdel-Hamid et al., 1993). Guidelines for cost estimation are given in (Boehm and Sullivan, 1999), (Fairley, 2002) and (Jørgensen, 2005). (Software, 2000) is a special issue devoted to software estimation.

Exercises

1. In which ways may political arguments influence cost estimates?
2. What does the Walston--Felix model look like?
3. How may the Rayleigh-curve be related to software cost estimation?

4. Give a sketch of Function Point Analysis (FPA).
5. Give a sketch of COCOMO 2.
6. Discuss the major differences between COCOMO 2 and FPA.
7. Give a rationale for Brooks' Law.
8. In which sense does Function Point Analysis (FPA) reflect the batch-oriented world of the 1970s?
9. How may early cost estimates influence the way in which a project is executed?
10. Why is it difficult to compare different cost estimation models?
11. Suppose you are involved in a project which is estimated to take 100 man-months. How would you estimate the nominal calendar time required for this project? Suppose the project is to be finished within six calendar months. Do you think such a schedule compression is feasible?
12. Why should software cost models be recalibrated from time to time?
13. ♠ How would you calibrate the COCOMO 2 model to fit software development in your organization?
14. ♡ Suppose you are managing a project which is getting behind schedule. Possible actions include: renegotiating the time schedule, adding people to the project, and renegotiating quality requirements. In which ways can these actions shorten the time schedule? Can you think of other ways to finish the project on time?
15. ♡ Suppose you have a LOC-based cost estimation model available whose parameters are based on projects from your own organization that used COBOL as the implementation language. Can you use this model to estimate the cost of a project whose implementation language is Pascal? What if the model is based on projects that used C?
16. ♡ Can you give an intuitive rationale for the values of the COCOMO 2 cost drivers (figure 7.11) that relate to project attributes?