SOFTWARE COST ESTIMATION

Many modern software systems are expensive to develop. Most software is complex and there is an explosive demand for increased functionality in new software products, making new software development more expensive. Accurate estimation of the cost of a projected software project is essential if sufficient resources are to be allocated for the project's completion. In extreme cases, estimation of the cost of a projected software project can help determine if a proposed software project is simply too big or complex for the amount of resources that feasibly can be allocated to the project's completion. Many software projects, and indeed, many software companies, have failed because of inaccurate estimation of software costs (1).

As with any new product development, the cost of a software project depends upon several factors:

1. The sheer amount of software required for a project. Modern software packages such as the Microsoft Excel spreadsheet comprise over 1.2 million lines of source code (2). Such software is necessarily far more expensive to develop than was Visicalc, which was one of the original versions of personal computer spreadsheet software, and which was distributed on a single floppy disk and ran on computers with as little as 32 K of memory (3). Software cost appears to grow exponentially with project size.
2. The amount of new software that will be needed for the project. This in turn depends on the amount of software that can be reused as is, without modification, for the project; the amount that must be created to be created; and the amount of existing software that is to be reused after modification (4). The most extreme case of reusing existing software is incorporating an entire product into a new software system; such existing software is often called COTS (commercial, off-the-shelf) or GOTS (government, off-the-shelf) software, depending on its place of origin.
3. The complexity of the software that is to be created is an essential consideration, especially if there is any difficulty in meeting the requirements of the project due to tight time performance constraints or limited memory in the expected run-time environment. Systems that are required to react to external phenomena within prescribed time limits (so-called real-time software) are necessarily more expensive to build than other systems without these requirements, with some additional costs just for testing that the system meets the software timing constraints.
4. Very high general quality requirements for software correctness in certain application domains are a major factor in the cost of a software system. The potential effects of an error in software may depend on the application domain. For example, a software failure in a word processing system that forces a user to reboot a personal computer and results in the loss of a few paragraphs of work is annoying, with the level of annoyance directly correlated with the amount of work lost. Such system failures can be tolerated if they are infrequent and the software offers enough features at reasonable cost. However, such system failures cannot be tolerated in software that controls human life, such as monitoring medical devices, controlling processes in a chemical or nuclear power plant, or maintaining the safety of airline passengers in an air traffic control system. Correctness of other software systems, such as the ones used to coordinate banking transactions and payments to recipients of Social Security checks in the United States is also essential, although human life may not directly depend on the software systems functioning correctly. These extra demands for system correctness and robustness mean that the software must be of particularly high quality. This increased quality is obtained by a precise process with additional testing, reviews, and perhaps external certification of quality control. The increased quality requirement naturally increases software development costs.
5. The hardware and software resources available to the project become important if the project has a tight schedule. The cost of a project can only increase if there are not enough computers available for the software development team or if compilation is slow because of slow computers with limited memory. Inability to access existing software libraries because license renewals have not been paid can increase the cost of a project by wasting resources.
6. The development environment available to the project can have an important effect on programmer productivity. Software tools that allow a programmer to examine source code in one window, while seeing the value of certain variables in another window, can be valuable aids in the software development process. Other useful software tools allow automatic collection and analysis of the results of running the software on previously-developed test suites, or browsing through libraries of classes for object-oriented software development. Still other software tools encourage a consistent view of software throughout the requirements gathering, design, coding, testing, integration, and maintenance phases of the software's life cycle. Maintenance is costs incurred after the software is delivered. The software tools described in this paragraph are often referred to as CASE (computer aided software engineering) tools.
7. The training and experience of the project personnel is also an important factor in the cost of a software product. For instance, many CASE tools often have a steep learning curve before they can be used effectively, even by experienced software engineers. Thus a project that requires use of an unfamiliar CASE tool can incur greater costs than if the team was familiar with the CASE tool being used. Beginning programmers, more experienced software engineers who must learn a new programming language, or even software engineers unfamiliar with a particular application domain, are unlikely to be as productive as a software engineer experienced in the programming languages to be used and the terminology of the ap-

plication domain. This is true, even if the programmers are equally talented. Programmer productivity is an essential component of software development cost.

8. Changes in technology can have a major effect on the cost of a product's development. For example, the object-oriented paradigm of languages such as C++ (5) has had a major impact on standards for computer graphics and representation of graphical objects. The effect of the Java programming language (6) and Internet-based software development paradigms on the personal computer industry has had far-reaching consequences for both applications and operating systems development and this effect is certain to continue. The use of Java "applets" within programs allows the incorporation of small software applications to be considered as components that can be inserted easily into larger software systems. Recent, evolving, standards such as HTML (hypertext markup language) and HTTP (hypertext transfer protocol) allow such artifacts as graphical images and output of spreadsheets to be inserted into applications for the Internet. Many projects have been forced to change direction in mid-stream, due to changing technology and the resulting effects on the marketplace. This increases cost.
9. The expected lifetime of a software system can play a major role in the cost of the system. For example, some versions of the software that controls the flow of railroad traffic in the United States have been in use for over twenty-five years. This 25 year period has seen major changes in programming languages, operating systems, and the underlying computer hardware. Communications protocols have changed immensely, as have techniques for allowing concurrent access to shared resources such as railroad tracks. Clearly the cost of such software systems is much larger than it would be if the software had only been in use for one year. The maintenance costs for this software over a period of 25 years are much higher than if the software was discontinued after one year. Maintenance often accounts for 75 % of the total lifetime costs for many software systems.
10. The quality of the project's management affects the cost of a software development project. For example, a highly inefficient project schedule, with many software engineers unable to proceed because they are waiting for delivery of a critical product, will affect cost adversely. Lack of quality control can result in inadequate, incomplete, or contradictory requirements, which in turn can increase costs of producing software dependent on these requirements. Other, apparently minor, decisions can lead to enormous costs during the remainder of the software's development or during its maintenance. For example, the so-called "Year 2000 problem," which often was caused by a simple design decision to allocate only two digits for storage of years in a date field, required a major investment to fix a huge inventory of existing software for business, financial, medical, and other applications.

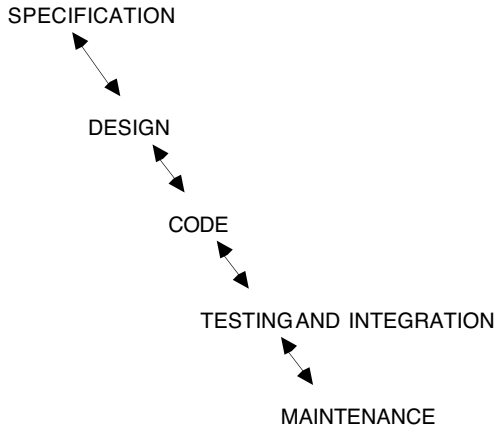

Figure 1. A stylized version of the classical waterfall model of software development.

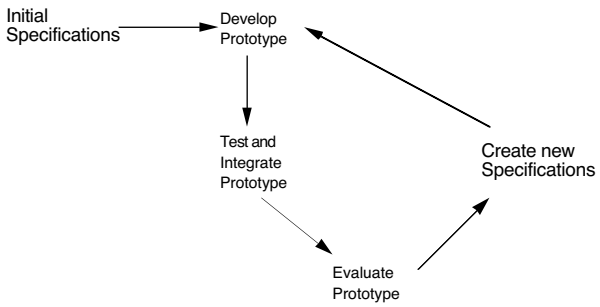

Figure 2. A stylized version of the rapid prototyping model of software development.

11. In many cases, the process used to develop a software project can also affect the software's cost. A rigidly followed process using the classical waterfall life cycle model will certainly increase overall costs for an organization's software products if technological changes are so rapid that they make the assumptions made in the initial requirements invalid. Figure 1 shows a stylized version of the classical waterfall model of software development in which the software is understood to go through distinct phases of requirements, design, coding, testing and integration, and maintenance. An alternative model of software development, based on iteration of a succession of prototypes, and known as the rapid prototyping model, is illustrated in Figure 2. Creation of too many prototypes can also slow down a software project's development.

It is clear that each of these eleven factors affects the cost of developing a software system. The major difficulty in software cost estimation is determining a quantitative basis for the effect that these factors have on the cost of a particular project. The problem is so complex that in most software projects, any other factors are ignored because they have at most second order effects and the first order effects listed above are difficult enough to quantify.

We note explicitly that the controlled experiments so important in many sciences are not usually practical in software engineering, because most organizations cannot afford the resources necessary to perform parallel experiments on actual production of software systems. In particular, the smaller scale experiments typically carried out in academic software engineering experiments using students typically have little relevance to typical cost estimation problems in industry or government.

SOFTWARE COST ESTIMATION FOR THE PRACTICING SOFTWARE ENGINEER

In general, software cost data and estimation models cannot be compared across organizations because of differences in how costs are recorded, how these costs are charged to particular sub-accounts for individual projects, and even how these costs are measured. Often there are major differences in software accounting procedures and standards, even between different units of the same organization. The only commonality between the software developed is typically mandated externally to the organization, such as in the case of software developed to meet the detailed specifications of a government contract. For an organization developing new software, the most important sources for detailed information often are to be found within the organization itself.

For the reasons listed above, most organizations have developed their own procedures for software cost estimation. The procedures are based on the organization's preferred methodology for software development. The procedures are used as the basis for software cost estimation models which are used by the organization.

In spite of the differences between various organizational approaches to software development, there are some common general approaches to software cost estimation. The approaches all have several features in common, and use feedback from other software projects that are deemed similar to the new project to improve the cost estimation process, at least for those organizations that are successful in predicting their software costs, which is illustrated in Figure 3:

1. A model for the organization's software costs estimation is selected. Such a cost model may be mathematically-based and consist of one or more formulas, each of which may have several parameters that must be determined. Initial, default, values may be used for these parameters if no additional information is available.
2. There is a systematic assessment of the size and relative complexity for each new software project. This assessment often is based on a decomposition of the project into smaller components which can be evaluated easily and the results are aggregated into an assessment of the entire project. The amount of software reuse, the number of complete COTS products used, and the underlying software technology used are very important at this stage.

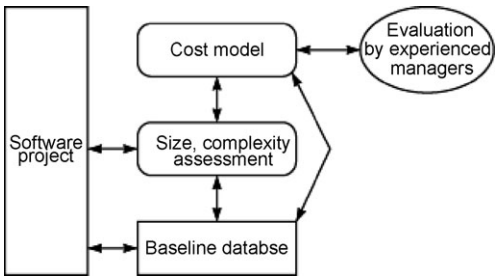

Figure 3. A model of the iterative process of developing cost estimation models and improving them

3. A database of experiences for other software development projects is created. This database is consulted for the new software development project and parameters such as those listed earlier are estimated for the new project, using the assessment of the size and complexity in the previous step.
4. The parameters estimated in step 2 are used as inputs into a cost model. The cost model will produce an estimate of cost and a range of likely values, together with an estimate of the time needed for the project's completion.
5. The cost estimates obtained in step 4 are then presented to a group of experienced managers for a review and a "sanity check" to make sure that the estimates are reasonable and that the development costs for the new software project is within the limits that the organization wishes to spend.
6. The project is then developed, with cost and schedule data collected and reviewed at different milestones (such as completed requirements or design, delivery of the first prototype, acceptance of the software by the customer, etc.).
7. Cost and schedule data for the project is compared with the estimated cost and schedule to determine any major deviations and to assess the reasons for the deviations, if any.
8. The detailed cost and schedule data obtained for the recently completed project is then incorporated into the database. Any deviations of the actual cost and schedule from those projected by the cost models are noted and then used to recalibrate the choice of parameters used in the mathematical model used for cost estimation. Statistical techniques such as regression analysis may be used (7).

Alternatively, a software cost model may be based on artificial intelligence techniques or even use neural nets. Such a model must, of course, be calibrated by data on the organizations relevant existing projects before being used. Generally speaking, such models are highly linked to the specific data and assumptions of the developing organization, and are not easily ported to different software development project cost modeling.

COCOMO SLIM and other mathematical cost models

A mathematical model of costs is generally given as one or more formulas that take a set of inputs, which are usually attributes of the particular software that is to be developed, and a set of parameters, which can be fine-tuned or recalibrated, according to the experiences of the organization developing the software. The output is an estimate of the cost, and in many cases, an estimate of the amount of time, needed for completion of the software development project.

The size of a software project is often an important component in a mathematical cost model. For example, the COCOMO cost model developed by Boehm requires a measure of the size of the software system in terms of the number of lines of code. The model is described in detail in his book, Software Engineering Economics, which is still one of the best general-purpose references in the field (7).

Boehm suggests the use of two formulas to compute the amount of effort (measured in person-months) and the time needed for completion of the project (measured in months). Boehm developed a hierarchy of three cost models: basic, intermediate, and advanced. We describe the basic and intermediate models briefly in this section, but will ignore the advanced model, referring the reader to Boehm's original book. Boehm's models are based on an assessment of the size of the system to be produced.

In the original COCOMO model, the first step is to estimate the size in lines of code. This total will be used as the variable K in the COCOMO formulas. It is measured in units of thousand lines of code. The article entitled "Software Metrics" in this encyclopedia describes some of the issues in software measurements and some approaches to formalizing the terminology to compare the concept of "lines of code" in different programming languages and application areas. Other measurements of software size are based on function points, which are a description of the software's functionality, rather than being based on the number of lines of code needed to construct software with this functionality. The book by Capers Jones (8) gives a good overview of function points and their use in software estimation. The number of lines of code and the number of function point are the metrics most commonly used for cost estimation.

Since the software system under consideration does not exist as an entirety at this point, the size (in lines of code or other measurement) must also be estimated, rather than measured exactly. The approach to size measurement is often is called a "work breakdown structure" because the software project to be developed is broken into smaller portions. Development of a work breakdown structure will be discussed in the next section.

A careful reader might object to this estimation process, because it replaces the estimate of the size of the entire system by a total of the estimates of the sizes of the individual components of the system. However, many practitioners of this approach believe that any errors in overestimating the size of individual components are likely to be balanced by other errors underestimating the size of other components. In any event, estimating the size of a project by a work breakdown structure is often used in practice.

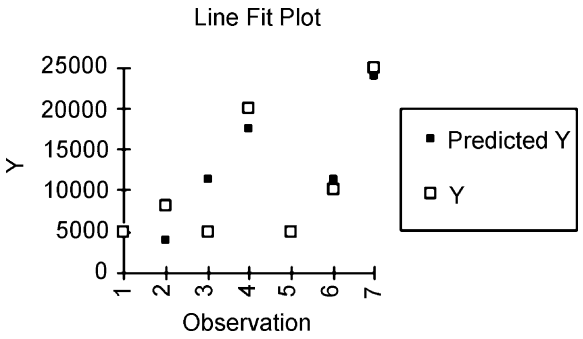

Figure 4. An attempt to fit a smooth curve to data in a scatter diagram using a COCOMO model approach.

Once the number of thousands of lines (K) has been estimated, the time and number of personnel can be estimated. We discuss the basic COCOMO model first. The relevant formulas are

$E = a_b * K * \exp(b_b)$

$D = c_b * E * \exp(d_b)$

where the coefficients a_b, b_b, c_b, and d_b, and d_b are based on relatively informal assessments of the relative complexity of the software. The computed quantities E and D are the amount of effort required for the project and the time needed for development of the project, respectively. The two quantities do not include the cost or time needed for software maintenance. The values of the constants a_b, b_b, c_b, and d_b should be taken from the appropriate entries in Table 1.

Note that the estimates for the quantities E and D are themselves based on estimates of the quantity K. Thus it is not reasonable to expect an exact match between estimates and actual values for the size and resources needed for a project. At best, an approximation with an expected range of accuracy can be determined, and this range of allowable error is heavily influenced by both the experience of the estimator and the quality of the information available in the organization for comparison with similar projects.

A typical relationship between the basic COCOMO model and some cost data is shown in Figure 4.

A technique such as linear regression (9) can be used to estimate the values of the parameters that provide a "best fit" to the existing software cost data in the database. If no such numbers are available, the default values of the formula should be used.

The basic COCOMO model can be extended to the so-called "intermediate COCOMO model." The intermediate COCOM model uses a set of "test driver attributes" which are given in Table 2.

The weights of these test driver attributes are to be entered on a scale from 1 to 6 and the resulting sum is used to create a multiplication factor that is used to modify the results of the basic COCOMO model. The rationale behind the extension of the basic COCOMO model to the intermediate COCOMO model is that mathematical cost estimation models should have a mechanism for incorporation of additional information into the cost estimates for projects (7).

The advanced COCOMO model extends the intermediate COCOMO model in much the same way that the in-

Table 1. Coefficients for the Basic COCOMO Model

Software project type	a_b	b_b	c_b	d_b
Small project, experienced team, flexible requirements ("organic")	2.4	1.05	2.5	0.38
Hard real-time requirements and strict interoperability ("embedded")	3.6	1.2	2.5	0.32
A mixture of the other two type of projects ("intermediate")	3.0	1.12	2.5	0.35

Table 2. "Test Driver Attributes" for the Intermediate COCOMO Model

Test Driver Attribute	Weight
Product attributes:	
Reliability requirements	
Size of application's database	
Software complexity	
Hardware attributes	
Run-time performance constraints	
Memory limitations	
Other processes competing for virtual memory	
Personnel attributes	
Analyst experience	
Software engineer experience	
Application domain experience	
Virtual machine experience	
Programming language experience	
Project attributes	
Use of software tools	
Use of software engineering methods	
Required development schedule	
TOTAL	

termediate COCOMO model extends the basic COCOMO model (7). It is considerably more complex.

The COCOMO model has been extended, and, to some extent, superseded recently by the COCOMO-2 model (10). The COCOMO-2 model includes assessment of the amount of software reuse and the difficulty of incorporating existing software components and COTS products into a new software system.

The SLIM estimation model is based on a model by Putnam of the assignment of effort during the various phases of a software project during its development lifetime, and often farther (10). The primary equation relates three parameters: effort, size, and productivity, of a project in a simple way, although the determination of the three parameters is often difficult. SLIM has been implemented as part of a suite of commercially available software estimation tools as well as in academic projects.

Many implementations of SLIM include an equation similar to:

$$S = E * Effort^{1/3} * t_d^{4/3}$$

Here t_d represents the software delivery time; E is the so-called *environment factor* that represents the productivity in the development environment, the size S (usually measured in LOC), and the $Effort$ (usually measured in person-years).

Estimation of Size and Relative Complexity

The term "work breakdown structure" is used in cost estimation to describe the result of repeated decomposition and stepwise refinement of a software project into smaller components whose size can be measured and whose cost presumably can therefore be estimated. A work breakdown structure is created by the following five step process:

1. Examine the list of detailed requirements.
2. For each requirement, estimate the number of lines of code needed to implement the requirement. Alternatively, estimate the number of function points needed to implement the requirement.
3. If the number of lines of code needed to implement a requirement cannot be estimated, decompose the requirement into smaller requirements until the number of lines of code (or function points) needed to fulfill each decomposed requirement can be estimated.
4. Use the size estimates obtained in steps 2 and 3 to estimate the number of lines of code (or function points) needed to meet the individual requirements.
5. Ignore any requirements for which an existing function, procedure, object, configuration file, software component, or COTS product can be reused as is.

If an existing function, procedure, object, configuration file, software component, or COTS product must be changed before it is used, estimate the amount of changes that must be made before it can be reused. Include an estimate of the size of any "filters" or "glueware" that are necessary to interface existing reusable software components with the rest of the software system to be developed.

6. Compute the total of all new lines of code (or function points) needed for the entire project.

Clearly the estimation of the size in step 2 has a great potential for errors. However, the underlying premise of the work breakdown structure approach is that even a large error in the estimation of the size of an implementation of an individual requirement (as measured by lines of code, function points, or similar metric) will be a relatively small factor in the overall estimation of the size of the system. For example, a fifty percent underestimation of size for an implementation of a requirement accounting for five percent of a system can cause at most a 2.5 percent error for the implementation of the overall system, assuming that most other estimates are accurate and the number of underestimates balances the number of overestimates.

The amount of reuse has a considerable effect on the cost of fulfilling a requirement. However, the cost models depend on the amount of change that a potentially reusable component must undergo before it can be reused in a software system. For example, if a component can be reused as is, without any changes, to meet a requirement, software development using the classic waterfall development process will have a cost that can be described as

```
cost = M * (non-reuse costs to develop)
+
integration and maintenance costs of non-
reuse-based system
```

Here the constant M represents a multiplier that is considered to be approximately 12.5% for software components reuse as is, without any changes (4). This description indicates the explicit need to integrate the existing software component into a system, to maintain it, and, most importantly, to locate the potentially reusable component and certify its correctness.

If the component is not reused as is, but has a relatively small number of changes (fewer than 25%), the cost model would be something like

```
cost = M * (non-reuse costs to develop)
+
.25 * (non-reuse costs to develop)
+
integration and maintenance costs of non-
reuse-based system
```

Here the constant M is still 12.5%.

The effect of reuse can be incorporated into other cost models as well. For example, software development based on the rapid prototyping approach would have a cost model of the form

```
cost = cost of requirements for non-reuse
system
+
.125 * (cost to evaluate non-reuse system)
+
cost to maintain system
```

if the requirements can be fulfilled by reusing an existing software component as is, and

```
cost = cost of requirements for non-reuse
system
+
sum (for all prototypes) of

.125 * (cost to evaluate non-reuse system)
+
reuse factor * (cost to develop prototype)
+
integration costs (non-reuse-based system)

+
cost to maintain system
```

if multiple prototypes must be developed with reuse factors other than one.

Establishment of a Baseline

Due to the complexity of software cost estimation and the risks involved with either underestimating or overestimating costs by large amounts, most organizations prefer to use only very experienced software managers to estimate costs for projects. Such managers often know the organization's history of successes and failures and can determine similarities between the project whose cost is being estimated and previous projects that are considered to be "similar." This in turn requires that adequate cost data must be available for these previously-developed "similar" projects. The most systematic approach to formalize the knowledge of these experienced software managers in the area of cost estimation is to develop a "baseline."

A baseline is a database of information about previous software projects. Different software development organizations use different approaches to database structure. Some common fields of a database include:

- Name of project
- Application domain
- Size of the project
- Special requirements, such as real-time or safety-critical systems
- Special interface requirements, such as other systems with which the software must be interoperable
- Programming language(s) used
- Computer hardware used for the final product
- CASE (computer-aided software engineering) or other software tools used for development
- Software development methodology used (classical waterfall, rapid prototyping, etc.)
- Number of project personnel used
- Time for delivery of the project
- Cost of each deliverable item for the project
- Cost of the entire project

- Unusual problems encountered during development
- Unusual problems encountered during maintenance

Note that some of the fields in the database used to create a baseline of project cost and size information are similar to those listed for the COCOMO and COCOMO-2 models.

The deliverable items for a typical software project might include: initial systems engineering analysis, preliminary and detailed requirements, preliminary and detailed design, source code, test plan, test data, integration plan, documentation, training manuals, and maintenance plans. Project management costs would also be included in the baseline database.

Since there are many different methodologies for software development, there is little common ground in the way that the cost of different deliverable products is treated. An organization using the classical waterfall model might have only two design documents as deliverable products: a preliminary, high level design and a more detailed, final design. In some highly complex projects, another, intermediate, design document might be produced. This is different from a prototyping model of software development. In this iterative development methodology, many intermediate design documents will be produced, at least one for each iteration of the software prototype. Comparing costs of different intermediate deliverables for different software development methodologies is meaningless.

Clearly, separate baselines must be developed for different development methodologies. The same is true for most of the other components of the baseline database.

Frequently, the completeness of the information in the database will vary by project, due to differences in accounting or project reporting procedures. Clearly, the quality of the information in the baseline affects the quality of cost and schedule prediction. It should be noted that the number of projects in the baseline database should exceed the number of parameters to be used in the mathematical cost model in order to avoid wide variations in estimation of future projects because there is insufficient data to determine the best values for parameters properly.

Once the database is used to establish a baseline, it is possible to indicate a range of costs for related software projects. For example, suppose that three projects consist of 250 KLOC, 280 KLOC, and 300 KLOC, with corresponding total costs of $1,500,000, $1,680,000, and $1,800,000, respectively. The acronym KLOC means thousand lines of code. If these three projects are in the same application domain, use the same programming language, target hardware, and CASE development tool, with all other factors being the same, then it is reasonable to expect that a similar software project of size 290 KLOC will have a cost somewhere between $1,500,000 and $1,800,000. More complex analyses can be performed, depending on the quality of information in the baseline database and on the project.

On the other hand, if a new project is estimated to have a size of 300 KLOC, but has real-time or safety-critical requirements that were not needed or relevant for other items in the baseline, then the project cost is likely to be considerably more than the $1,800,000 for the previous project in the database, which was of similar size, but which was less complex.

Managerial evaluation of cost estimates

A formal review of cost estimates for a new software development project by a team of senior software managers is often included in an organization's cost estimation process. Once the formal presentation is made and time has been allotted for the review of the project's initial set of requirements for completeness and accuracy, the senior managers are consulted for their own estimates. The goal is to get a consensus estimate of the true cost of the system to be developed.

One way in which consensus is reached in some software development organizations is often called the Delphi method, after the Oracle at Delphi which, according to Greek mythology, was consulted for advice and was noted for the cryptic nature of its answers. The idea is that the managers are given the information from the reviews and then they disperse to develop their own cost estimates for the system to be developed. After completion of their initial estimates, the managers will come together and describe the reasoning used to develop their estimates. After each manager has presented his or her case, they separate again to revise their estimates, incorporating as much of their colleague's analyses as they see fit. They then come together as a group and resume the discussion of their (revised) estimates. The process is repeated until the managers either come to consensus or determine that no consensus can be reached. Lack of consensus is a sign to upper management that the project may be risky.

Return on Investment and Risk Analysis

Once there is agreement on the size and cost of a project, a cost-benefit analysis must be performed. The organization must determine if the perceived value of the proposed software, in terms of its potential to improve market share, competitive advantage, or, in the case or government, security, exceeds its projected cost. Even if there is sufficient perceived advantage, there may be other competing proposals and resources that are too limited to handle two or more software development projects of a certain size and complexity at the same time. It is at this stage that an organization's potential return on investment might be computed. This calculation will involve other estimations of the potential increase in market share or revenue, the likelihood of other opportunities for use of the same resources during the project's development lifetime, and the cost of money if funds must be borrowed. Such decisions clearly involve many non-technical factors and negative decisions to terminate projects can annoy software engineers who championed the canceled projects. Decisions to cancel projects are often an impetus to the development of start-up software companies.

Another factor that can affect the decision of an organization to go forward with a project is the perceived risk. Any software development in the 1980s that focused exclusively on the Commodore-64 or Atari computers would have been risky. Not knowing the likely direction of the

industry in hardware, operating systems, standards, networking, or applications packages could lead to unacceptable amounts of risk for some organizations.

A final, less precisely defined factor that can affect the decision to continue a software project is the confidence that senior management has in the accuracy of the cost estimation process. Without a well-designed baseline database of information on previously completed software projects, an accurate estimate of costs for future projects will be nearly impossible to obtain. Lack of good information about the cost (and quality) of previous projects would make senior management highly skeptical of cost estimates for any proposed software projects. The natural reaction of senior management in the face of limited information would be to assume that cost estimates may be very low; therefore, many otherwise meritorious projects might be canceled because their cost/benefit ratio appears to be high and the return on investment is perceived to be low.

Scheduling

Once a cost estimate is obtained for a project, an assessment of the project's duration can be determined. Frequently, the estimation of the project's duration is a by-product of the cost model used, as is the case for the COCOMO and COCOMO-2 models. The total predicted time can then be broken down further into a schedule of project milestones, including deliverable prototypes, requirements and design reviews, etc.

The database used as a cost estimation baseline can be used to provide guidance in project scheduling. The time needed for each milestone can be read from the baseline database and the historical profile of the percentage of time spent on each project activity can be used to provide an initial estimate of the current project's schedule.

RESEARCH DIRECTIONS IN SOFTWARE COST MODELING

Much of the research in software cost estimation can be classified as falling into one of three categories:

1. Many universities perform small research projects using student programmers. These projects often include experiments that evaluate the efficiency of particular strategies on software development. Strategies include reorganizing sets of requirements to accommodate some preexisting software components or even COTS products. Here the "cost" of a software project is measured indirectly by the number of hours indicated by students in project reports. This type of research can set a direction for researchers in the filed of software cost modeling, but the results obtained often do not scale up to industrial applications.
2. Many organizations, or single sites within an organization, perform comparative studies of moderate-sized projects. The research is usually restricted to consider software projects within a single application domain. These studies generally take the form of a comparative study or effectiveness of particular methods of software cost estimation. Often the study is limited to comparison of features of commercial software that can be used by project managers to aid in software cost estimation. Other studies examine baseline databases to determine patterns that might have predicted costs better than the cost models that were used for the initial cost estimates of the baseline projects (11). The data analysis suffers from the incompleteness of databases. The results may not be applicable to other organizations because of differences in software development methodology or the special nature of the particular application domain.
3. There are a few research efforts to consider large projects across government and several industries. The COCOMO-2 project at the University of Southern California is one of the most prominent examples (12). The advantage of this approach is that the data is wide-ranging. The disadvantage is that there is no controlled experiment and the data obtained might be somewhat flawed because of the lack of a rigidly controlled experimental process.

We note explicitly that the Software Engineering Laboratory at NASA's Goddard Space Flight Center, which is a partnership between NASA, the University of Maryland at College Park, and Computer Sciences Corporation, performs all types of research (small, formal experiments; comparative studies; and case studies) and is an excellent source of cost modeling information (13). The Software Engineering Institute is also an excellent source of information on software cost modeling (14).

Research Directions in Software Reuse and Cost Modeling

Software reuse can be a major factor in reducing software costs and is therefore an important component of cost models. From the perspective of software cost modeling, the most difficult problem occurs when the software component to be reused is either in the form of a complete COTS product with no source code available, or is such a complex system that there are so many interactions with operating system services or other COTS applications that the task of integrating the software becomes much more expensive than might have been allowed for in the cost estimates. Essentially, the "glueware" or "filters" become more of an expense than the reused software component. There are few metrics that appear to be relevant for determining the size of the glueware of filters and even less reliable information on prediction of integration costs.

Characterization of software as matching one of a set of patterns is an important new research area in software engineering, particularly for object-oriented software (15). Few large industrial software projects have been developed using the approach of matching patterns within a framework. However, it is clear that new methods of software cost modeling will be necessary for accurate prediction of any substantial software system developed using this approach. As with cost modeling of software projects developed with COTS products, prediction of the cost of integration is a major stumbling block, due to unforeseen low-level interactions.

The use of COTS may require a change in an organization's software development process, because so much of the cost of COTS selection and analysis may occur before a contract is obtained for software development and systems integration. The paper by Waund indicates some of the issues in the case of a defense contractor developing COTS-based systems for the government (16). Ellis has a related paper (17).

A sophisticated systematic approach to software reuse is known as "product line architectures". In this approach, software components and subsystems considered likely to be reused are developed simultaneously with product schedules, and the cost models are best treated systemwide. An easily accessible source of current best practices for software product line architectures is available from the Software Engineering Institute (18).

An interesting doctoral dissertation by Sassenburg (19) addresses the effects of modeling of cost and quality in determining new releases of software systems. It may be especially useful when combined with product line architecture approaches.

Research Directions in Java, the Internet, and Software Cost Modeling

The influence of the explosive growth of the Java programming language, its associated application programming interfaces (API), and the smooth interface between Java and the Internet has had a profound effect on software development. As yet, there are few careful studies of the effect of Java or the Internet on software costs for real systems. This is due primarily to the lack of data regarding the costs of software maintenance for systems written in Java or the cost of providing configuration management of web sites.

Java allows APIs to applications written in multiple languages. As such, there is the potential for the same type of hidden costs due to unforeseen low-level interactions between software components. It remains to be seen if the popularity of Java will result in a fundamentally different approach to software cost modeling. The same holds true for scripting languages, which are beginning to be popular for Internet applications.

ACKNOWLEDGEMENTS

This research was partially funded by the United States Government under agreement number W911W6-06-2-0008. This research was also partially supported by the National Science Foundation under grant number 0324818.

The U. S. Government is authorized to reproduce and distribute reprints notwithstanding and copyright notation therein.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either express or implied, of the U. S. Government.

BIBLIOGRAPHY

1. W. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Massachusetts, 1989.
2. M. A. Cusumano, R. W. Selby," How Microsoft Builds Software", *Commun. ACM*, Vol.**40**, No. 6,June, 1997.
3. D. Bricklin," Visicalc '79", *Creative Computing*, Vol.**10**, 1984, pp. 122–124.
4. R. J. Leach, *Software Reuse: Methods, Models, Costs*, McGraw-Hill, New York, 1996.
5. B. Stroustrup, *The C++ Programming Language, second edition*, Addison-Wesley, Reading, Massachusetts, 1991.
6. E. Au, D. Makower, *Java Programming Basics*, MIS Press, New York, 1996.
7. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
8. Capers Jones, *Assessment and Control of Software Risks*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994.
9. O. J. Dunn, V. A. Clark,*Applied Statistics: Analysis of Variance and Regression*, John Wiley, New York, 1987.
10. L. Putnam, W. Myers, *Measures for Excellence*, Yourdon Press Computing Series, 1992.
11. M. Shepperd, C. Schofield, B. A. Kitchenham," Effort Estimation Using Analogy," *International Conference on Software Engineering, ICSE-18*, Berlin, 1996.
12. B. Boehm, B. Clark, S. Devnani-Chulani," Calibration Results of COCOMO II" *Proceedings of the Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, Maryland, December 3-4, 1997.
13. Software Engineering Laboratory (SEL), NASA Goddard Space Flight Center, Greenbelt, Maryland.
14. Software Engineering Institute (SEI), Carnegie-Mellon University, Pittsburgh, Pennsylvania.
15. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
16. C. Waund," COTS Integration and Support Model," in Systems Engineering in the Global Marketplace: *NCOSE International Symposium*, St. Louis, Missouri, July 24-26, 1995.
17. T. Ellis," COTS Integration in Software Solutions - a Cost Model," in "Systems Engineering in the Global Marketplace," *NCOSE International Symposium*, St. Louis, Missouri, July 24-26, 1995.*A Framework for Software Product Line Practice, Version 4.2, www.sei.cmu.edu/productlines/framework.html.* Software Engineering Institute, Pittsburgh, Pennsylvania, 2005.
18. D. M. Weiss, C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, Reading, Massachusetts, 1999.
19. H. Sassenburg,"*Design of a Methodology to Support Software Release Decisions: Do the Numbers Really Matter*?," Thesis, University of Groningen, SE-CURE AG(www.se-cure.ch), 2005.

Cross-References

See also: software engineering, software metrics, software selection, software management, software reusability, computer aided software engineering, cost-benefit analysis

RONALD J. LEACH

Department of Systems and
Computer Science, College of
Engineering, Architecture,
and Computer Sciences,
Howard University,
Washington, DC