Chapter

3

Six Forms of Software Cost Estimation

Among our clients about 80 percent of large corporations utilize automated software-estimation tools. About 30 percent utilize two or more automated estimation tools, sometimes for the same project. About 15 percent employ cost-estimating specialists. In large companies, manual estimates are used primarily for small projects below 500 function points in size or for "quick and dirty" estimates where high precision is not required.

However, for small companies with less than 100 software personnel, only about 25 percent utilize automated software-estimation tools. The primary reason for this is that small companies only build small software projects, where high-precision estimates are not as important as for large systems.

Software cost estimates can be created in a number of different fashions. In order of increasing rigor and sophistication, the following six methods of estimating software costs are used by corporations and government groups that produce software.

- Manual software-estimating methods
  1. Manual project-level estimates using rules of thumb
  2. Manual phase-level estimates using ratios and percentages
  3. Manual activity-level estimates using work-breakdown structures
- Automated software-estimating methods
  1. Automated project-level estimates (macro-estimation)
  2. Automated phase-level estimates (macro-estimation)
  3. Automated activity-level or task-level estimates (micro-estimation)

The most accurate forms of software cost estimation are the last ones in each set: cost estimating at either the activity or the task level. Only the very granular forms of software cost estimation are usually rigorous enough to support contracts and serious business activities. Let us consider the pros and cons of each of these six estimating methods.

Overview of Manual Software-Estimating Methods

Manual estimates for software projects using simple rules of thumb constitute the oldest form of software cost estimation, and this method is still the most widely used, even though it is far from the most accurate.

An example of an estimating rule of thumb would be "Raising the function point total of an application to the 0.4 power will predict the schedule of the project in calendar months from requirements until delivery." Another and more recent example would be "for a story that contains five story points, it can be coded in 30 hours of ideal time."

Examples of rules of thumb using the lines-of-code-metrics might be "JAVA applications average 500 non-commentary code statements per staff month" or "JAVA applications cost an average of $10 per line of code to develop."

About the only virtue of this simplistic kind of estimation is that it is easy to do. However, simplistic estimates using rules of thumb should not serve as the basis of contracts or formal budgets for software projects.

Manual phase-level estimates using ratios and percentages are another common and long-lived form of software estimation. Usually, the number of phases will run from five to eight, and will include such general kinds of software work as: (1) requirements gathering, (2) analysis and design, (3) coding, (4) testing, and (5) installation and training.

Manual phase-level estimates usually start with an overall project-level estimate and then assign ratios and percentages to the various phases. For example, suppose you were building an application of 100 function points, or roughly 10,000 COBOL source code statements in size. Using the rules of thumb from the previous example, you might assume that if this project will average 500 source code statements per month, then the total effort will take 20 months.

Applying typical percentages for the five phases previously shown, you might next assume that requirements would comprise 10 percent of the effort, analysis and design 20 percent, coding 30 percent, testing 35 percent, and installation and training 5 percent.

Converting these percentages into actual effort, you would arrive at an estimate for the project that showed the following:

Requirements	2 staff months
Analysis and design	4 staff months
Coding	6 staff months
Testing	7 staff months
Installation	1 staff month
TOTAL	20 staff months

The problems with simple phase-level estimates using ratios and percentages are threefold:

- The real-life percentages vary widely for every activity.
- Many kinds of software work span multiple phases or run the entire length of the project.
- Activities that are not phases may accidentally be omitted from the estimate.

As an example of the first problem, for small projects of less than 1000 lines of code or 10 function points, coding can total about 60 percent of the total effort. However, for large systems in excess of 1 million lines of code or 10,000 function points, coding is often less than 15 percent of the total effort. You cannot use a fixed percentage across all sizes of software projects.

As an example of the second problem, the phase-level estimating methodology is also weak for activities that span multiple phases or run continuously. For example, preparation of user manuals often starts during the coding phase and is completed during the testing phase. Project management starts early, at the beginning of the requirements phase, and runs throughout the entire development cycle.

As an example of the third problem, neither *quality assurance* nor *technical writing* nor *integration* are usually identified as phases. But the total amount of effort devoted to these three kinds of work can sometimes top 25 percent of the total effort for software projects. There is a common tendency to ignore or to underestimate activities that are not phases, and this explains why most manual estimates tend toward excessive optimism for both costs and schedules.

The most that can be said about manual phase-level estimates is that they are slightly more useful than overall project estimates and are just about as easy to prepare. However, they are far from sufficient for contracts, budgets, or serious business purposes.

The third form of manual estimation, which is to estimate each activity or task using a formal work-breakdown structure, is far and away the most accurate of the manual methods.

This rigorous estimating approach originated in the 1960s for large military software projects and has proven to be a powerful and effective method that supports other forms of project management, such as critical path analysis. (Indeed, the best commercial estimating tools operate by automating software estimates to the level of activities and tasks derived from a work-breakdown structure.)

The downside of manual estimating via a detailed work-breakdown structure of perhaps 50 activities, or 250 or so tasks, is that it is very time consuming to create the estimate initially, and it is even more difficult to make modifications when the requirements change or the scope of the project needs to be adjusted.

Overview of Automated Software-Estimating Methods

The first two forms of automated estimating methods are very similar to the equivalent manual forms of estimation, only faster and easier to use. The forms of automated estimation that start with general equations for the staffing, effort, and schedule requirements of a complete software project are termed *macro-estimation*.

These macro-estimation tools usually support two levels of granularity: (1) estimates to the level of complete projects, and (2) estimates to the level of phases, using built-in assumptions for the ratios and percentages assigned to each phase.

Although these macro-estimation tools replicate the features of manual estimates, many of them provide some valuable extra features that go beyond the capabilities of manual methods.

Recall that automated software-estimation tools are built on a knowledge base of hundreds, or even thousands, of software projects. This knowledge base allows the automated estimation tools to make adjustments to the basic estimating equations in response to the major factors that affect software project outcomes, such as the following:

- Adjustments for levels of staff experience
- Adjustments for software development processes
- Adjustments for specific programming languages used
- Adjustments for the size of the software application
- Adjustments for work habits and overtime

The downside of macro-estimation tools is that they do not usually produce estimates that are granular enough to support all of the important software-development activities. For example, many specialized activities tend to be omitted from macro-estimation tools, such as the

production of user manuals, the effort by quality-assurance personnel, the effort by database administrators, and sometimes even the effort of project managers.

The automated estimating tools that are built upon a detailed work-breakdown structure are termed *micro-estimating* tools. The method of operation of micro-estimation is the reverse of that of macro-estimation.

The macro-estimation tools begin with general equations for complete projects, and then use ratios and percentages to assign resources and time to specific phases.

The micro-estimation tools work in the opposite direction. They first create a detailed work-breakdown structure for the project being estimated, and then estimate each activity separately. When all of the activity-level or task-level estimates are complete, the estimating tool then sums the partial results to reach an overall estimate for staffing, effort, schedule, and cost requirements. The advantages of activity-based micro-estimation are the following:

- The granularity of the data makes the estimates suitable for contracts and budgets.
- Errors, if any, tend to be local within an activity, rather than global.
- New or unusual activities can be added as the need arises.
- Activities not performed for specific projects can be backed out.
- The impact of specialists, such as technical writers, can be seen.
- Validation of the estimate is straightforward, because nothing is hidden.
- Micro-estimation is best suited for Agile projects.

A critical aspect of software estimation is the chart of accounts used, or the set of activities for which resource and cost data are estimated. The topic of selecting the activities to be included in software project estimates is a difficult issue and cannot be taken lightly. There are four main contenders:

- Project-level measurements
- Phase-level measurements
- Activity-level measurements
- Task-level measurements

Before illustrating these four concepts, it is well to begin by defining what each one means in a software context, with some common examples.

A *project* is defined as the implementation of software that satisfies a cohesive set of business and technical requirements. Under this definition, a project can be either a standalone program, such as an accounting application or a compiler, or a component of a large software system, such as the supervisor component of an operating system. The manager responsible for developing the application, or one of the components of larger applications, is termed the *project manager*.

Software projects can be of any size, but those where software cost-estimating and project management tools are utilized are most commonly those of perhaps 1000 function points, or 100,000 source code statements, and larger. Looking at the project situation from another view, in a cost-estimating and project management context, formal project estimates and formal project plans are usually required for projects that will require more than five staff members and will run for more than about six calendar months.

A *phase* is a chronological time period during which much of the effort of the project team is devoted to completing a major milestone or constructing a key deliverable item. There is no exact number of phases, and their time intervals vary. However, the phase concept for software projects implies a chronological sequence starting with requirements and ending with installation or deployment.

An example of a typical phase structure for a software project might include the following:

1. The requirements phase
2. The risk analysis phase
3. The design and specification phase
4. The coding phase
5. The integration and testing phase
6. The installation phase
7. The maintenance phase

Of course, some kinds of work, such as project management, quality assurance, and the production of user documents, span multiple phases. Within a phase, multiple kinds of activities might be performed. For example, the testing phase might have as few as one kind of testing or as many as a dozen discrete forms of testing.

The phase structure is only a rough approximation that shows general information. Phases are not sufficient or precise enough for cost estimates that will be used in contracts or will have serious business implications.

An *activity* is defined as the sum of the effort needed to complete a key milestone or a key deliverable item. For example, one key activity is gathering user requirements. Other activities for software projects would be completion of external design, completion of design reviews on the external design, completion of internal or logical design, completion of design reviews on the logical design, completion of database design, completion of a test plan, completion of a user's guide, and almost any number of others.

There are no limits on the activities utilized for software projects, but from about 15 to 50 key deliverables constitute a normal range for software cost-estimating purposes. Activities differ from phases in that they do not assume a chronological sequence; also, multiple activities are found within any given phase. For example, during a typical software project's testing phase it would be common to find the following six discrete testing activities:

1. New function testing
2. Regression testing
3. Component testing
4. Integration testing
5. Stress testing
6. System testing

A *task* is defined as the set of steps or the kinds of work necessary to complete a given activity. Using the activity of unit testing as an example, four tasks normally included in that activity might comprise the following:

1. Test case construction
2. Test case running or execution
3. Defect repairs for any problems found
4. Repair validation and retesting

There is no fixed ratio of the number of tasks that constitute activities, but from 4 to perhaps 12 tasks for each activity are very common patterns.

Of these four levels of granularity, only activity and task estimates will allow estimates with a precision of better than 10 percent in repeated trials. Further, neither project-level nor phase-level estimates will be useful in modeling process improvement strategies, or in carrying out "what if" alternative analysis to discover the impact of various tools,

methods, and approaches. This kind of modeling of alternative scenarios is a key feature of automated software-estimating approaches, and a very valuable tool for software project managers.

Estimating only at the level of full projects or at phase levels correlates strongly with cost and schedule overruns, and even with litigation for breach of contract.

This is not to say that phase-level or even project-level estimates have no value. These concise estimating modes are very often used for early sizing and estimating long before enough solid information is available to tune or adjust a full activity-level estimate.

However, for projects that may involve large teams of people, have expenses of more than $1 million, or have any kind of legal liabilities associated with missed schedules, cost overruns, or poor quality, then a much more rigorous kind of estimating and planning will be necessary.

A fundamental problem with the coarse estimating approaches at the project and phase levels is that there is no way of being sure what activities are present and what activities (such as user manual preparation) might have been accidentally left out.

Also, data estimated to the levels of activities and tasks can easily be rolled up to provide phase-level and project-level views. The reverse is not true: You cannot explode project-level data or phase-level data down to the lower levels with acceptable accuracy and precision. If you start an estimate with data that is too coarse, you will not be able to do very much with it.

Table 3.1 gives an illustration that can clarify the differences. Assume you are thinking of estimating a project such as the construction of a small switching system. Shown are the activities that might be included at the levels of the project, phases, and activities for the chart of accounts used to build the final cost estimate.

Even more granular than activity-based cost estimates would be the next level, or task-based cost estimates. Each activity in Table 3.1 can be expanded down a level (or even more). For example, activity 16 in Table 3.1 is identified as unit testing. Expanding the activity of *unit testing* down to the task level might show six major tasks:

Activity	Tasks
Unit testing	1. Test case creation
	2. Test case validation
	3. Test case execution
	4. Defect analysis
	5. Defect repairs
	6. Repair validation

Assuming that each of the 25 activities in Table 3.1 could be expanded to a similar degree, then the total number of tasks would be 150. This level

TABLE 3.1 Project-, Phase-, and Activity-Level Estimating Charts of Accounts

Project level	Phase level	Activity level
Project	1. Requirements	1. Requirements
	2. Analysis	2. Prototyping
	3. Design	3. Architecture
	4. Coding	4. Planning
	5. Testing	5. Initial design
	6. Installation	6. Detail design
		7. Design review
		8. Coding
		9. Reused code acquisition
		10. Package acquisition
		11. Code inspection
		12. Independent verification and validation
		13. Configuration control
		14. Integration
		15. User documentation
		16. Unit testing
		17. Function testing
		18. Integration testing
		19. System testing
		20. Field testing
		21. Acceptance testing
		22. Independent testing
		23. Quality assurance
		24. Installation
		25. Management

of granularity would lead to maximum precision for a software project estimate, but it is far too complex for manual estimating approaches, at least for ease and convenience of use.

Although some large software systems consisting of multiple components may actually reach the level of more than 3000 tasks, this is a misleading situation. In reality, most large software systems are really comprised of somewhere between half a dozen and 50 discrete components that are built more or less in parallel and are constructed using very similar sets of activities. The absolute number of tasks, once duplications are removed, seldom exceeds 100, even for enormous systems that may top 10 million source code statements or 100,000 function points.

Only in situations where hybrid projects are being constructed so that hardware, software, microcode, and purchased parts are being simultaneously planned and estimated will the number of activities and tasks top 1000, and these hybrid projects are outside the scope of software cost-estimating tools. Indeed, really massive and complex hybrid projects will stress any kind of management tool.

For day-to-day software estimation, somewhere between 10 and 30 activities and perhaps 30 to 150 tasks will accommodate almost any software application in the modern world and will allow estimates with sufficient precision for use in contracts and business documents.

Estimating to the activity level is the first level suitable for contracts, budgets, outsourcing agreements, and other serious business purposes. Indeed, the use of simplistic project or phase-level estimates for software contracts is very hazardous and may well lead to some kind of litigation for breach of contract.

Estimating at the activity level does not imply that every project performs every activity. For example, small MIS projects and client/server applications normally perform only 10 or so of the 25 activities that are shown previously. Systems software such as operating systems and large switching systems will typically perform about 20 of the 25 activities. Only large military and defense systems will routinely perform all 25.

However, it is better to start with a full chart of accounts and eliminate activities that will not be used. That way you will be sure that significant cost drivers, such as user documentation, are not left out accidentally because they are not part of just one phase.

Table 3.2 illustrates some of the activity patterns associated with six general kinds of software projects:

- Web-based applications
- Management information systems (MIS)
- Contract or outsourced projects
- Systems software projects
- Commercial software projects
- Military software projects

As can be seen from Table 3.2, activity-based costing makes visible some important differences in software-development practices. This level of granularity is highly advantageous in software contracts and is also very useful for preparing detailed schedules that are not likely to be exceeded for such trivial reasons as accidentally omitting an activity.

Now that the topic of activity-based estimating has been discussed, it is of interest to illustrate some of the typical outputs that are available from commercial software-estimating tools. Table 3.3 illustrates a hypothetical 1000–function point systems software project written in the C programming language.

The granularity of the estimate is set at the activity level, and the project is assumed to have started on January 6, 1997. In this example,

TABLE 3.2 Typical Activity Patterns for Six Software Domains

Activities performed	Web-based	MIS	Outsource	Commercial	Systems	Military
01 Requirements		X	X	X	X	X
02 Prototyping	X	X	X	X	X	X
03 Architecture		X	X	X	X	X
04 Project plans		X	X	X	X	X
05 Initial design		X	X	X	X	X
06 Detail design		X	X	X	X	X
07 Design reviews			X	X	X	X
08 Coding	X	X	X	X	X	X
09 Reuse acquisition	X		X	X	X	X
10 Package purchase		X	X		X	X
11 Code inspections				X	X	X
12 Independent verification and validation				X		
13 Configuration management		X	X	X	X	X
14 Formal integration		X	X	X	X	X
15 Documentation	X	X	X	X	X	X
16 Unit testing	X	X	X	X	X	X
17 Function testing		X	X	X	X	X
18 Integration testing		X	X	X	X	X
19 System testing		X	X	X	X	X
20 Field testing				X	X	X
21 Acceptability testing		X	X		X	X
22 Independent testing						X
23 Quality assurance			X	X	X	X
24 Installation and training		X	X		X	X
25 Project management		X	X	X	X	X
Activities	5	16	20	21	22	25

the average burdened salary level for all project personnel is set at $5000 per month.

Although most of the outputs from this illustrative example are straightforward, several aspects might benefit from a discussion. First, note that 20 out of the 25 activities are shown, which is not uncommon for systems software in this size range.

Second, note that the overlapped schedule and the waterfall schedule are quite different. The waterfall schedule of roughly 120 calendar months is simply the arithmetic sum of the schedules of the various activities used. This schedule would probably never occur in real life, because software projects always start activities before the previous activities are completed. As a simple example, design usually starts when the requirements are only about 75 percent complete. Coding usually starts when the design is less than 50 percent complete, and so on.

The overlapped schedule of just over 18 months reflects a much more common scenario, and assumes that nothing is really finished when the next activity begins.

The third aspect of this example that merits discussion is the fact that unpaid overtime amounts to almost 42 staff months, which is about 14 percent of the total effort devoted to the project. This much unpaid overtime is a sign of three important factors:

- The software personnel are exempt, and don't receive overtime payments.
- Schedule pressure is probably intense for so much unpaid overtime to accrue.
- There are major differences between *real* and *apparen*t productivity rates.

If the unpaid overtime is left out (which is a common practice), then the apparent productivity rate for this project is 3.78 function points per staff month, or 473 source code statements per staff month.

If the unpaid overtime is included, then the real productivity rate for this project is 3.27 function points per staff month, or 409 source code statements per staff month. It can easily be seen that the omission or inclusion of unpaid overtime can exert a major influence on overall productivity rates.

Although coding is the most expensive single activity for this project, and costs almost $322,200 out of the total cost of just over $1,320,000, that is still only a little over 24 percent of the total cost for the project.

By contrast, the nine activities associated with defect removal (quality assurance, reviews, inspections, and testing) total to about $383,000 or roughly 29 percent of the overall development cost.

The activities associated with producing paper documents (plans, requirements, design, and user manuals) total to more than $394,000 or about 30 percent of the development cost.

Without the granularity of going down at least to the level of activity-based costs, the heavy proportion of non-coding costs might very well

be underestimated, and it would be difficult to ascertain if these costs were even present in the estimate. With activity-based costs, at least errors tend to be visible and, hence, can be corrected.

The overlap in project schedules is difficult to see from just a list of start and stop dates. This is why calendar intervals are usually shown visually in the form of Gantt charts, critical path networks, or PERT charts.

Figure 3.1 illustrates a Gantt chart that would accompany an activity-based cost estimate such as the one shown in Table 3.3. The provision of graphs and charts is a standard feature of a number of software cost-estimating tools, because graphical outputs make the visualization of key information easier to comprehend.

Many estimating tools allow users to switch back and forth between numerical and graphical output, to print out either or both kinds, and in some cases, to actually make adjustments to the estimate by manipulating the graphs themselves.

Analyzing the Gantt chart, it is easy to see why the waterfall schedule and the overlapped schedule differ by a ratio of almost 8 to 1. The sum of the schedules for the individual software activities is never equal to the elapsed time, because most activities are performed in parallel and overlap both their predecessor and their successor activities.

Incidentally, the kinds of Gantt chart information shown in both Table 3.3 and Fig. 3.1 are standard output from such software-estimating tools as CHECKPOINT, KnowledgePlan, SLIM, and a number of others.

However, if schedule information were needed down to the level of tasks, or even below that to the level of individual employees, then the data would usually be exported from a cost-estimating tool and imported into a project-planning tool, such as Microsoft Project.

The kinds of information shown in Table 3.3 and Figure 3.1 are only a few of the kinds of data that modern software cost-estimating tools can provide. Many of the other capabilities will be illustrated later in this book, as will some of the many other kinds of reports and analyses.

For example, software cost-estimating output reports also include quality and reliability estimates, maintenance and enhancement estimates, analyses of risks, and sometimes even evaluations of the strengths and weaknesses of the methods and tools being applied to the software project being estimated.

Strength and weakness analysis is also a useful capability for other purposes, such as moving up the Software Engineering Institute (SEI) capability maturity model. Since modern software cost-estimation tools, many of which include measurement capabilities, can include as many as a hundred or more influential factors, their ability to focus on topics

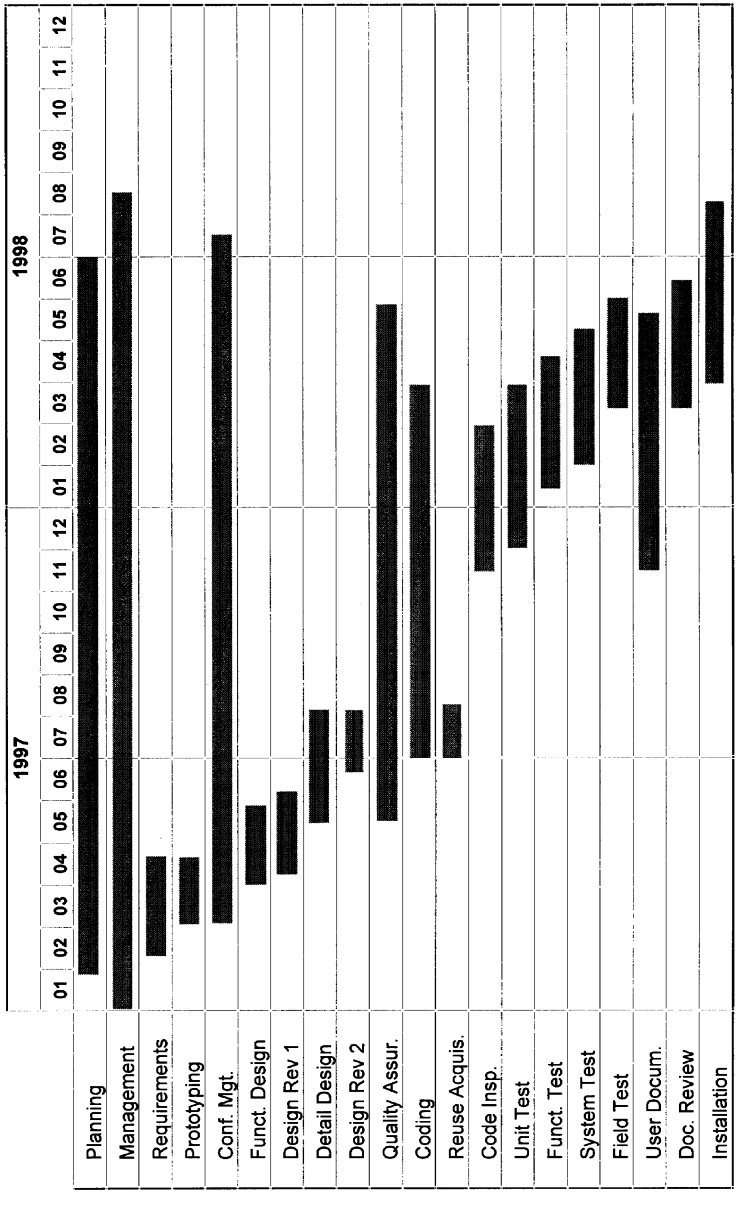

Figure 3.1 Sample Gantt chart output from a software cost-estimating tool.

TABLE 3.3 Example of Activity-Based Software Cost Estimating
Project type: Systems software of 1000 function points (125,000 C statements)
Project start: January 6, 1997
First delivery: July 15, 1998

Activity	Start date	End date	Schedule, months	Effort, months	Staffing	Cost, $
Planning	1/6/97	6/29/98	17.71	13.54	0.76	67,700
Management	1/6/97	8/15/98	17.40	19.45	1.12	97,250
Requirements	2/14/97	4/4/97	1.61	6.86	4.26	34,300
Prototyping	2/27/97	3/30/97	1.02	2.39	2.34	11,950
Configuration management	3/15/97	7/20/98	16.50	8.50	0.52	42,500
Functional design	3/6/97	5/12/97	2.20	15.72	7.15	78,600
Design reviews 1	3/24/97	5/12/97	1.61	3.92	2.43	19,600
Detail design	4/29/97	8/7/97	2.27	16.07	7.08	80,350
Design reviews 2	6/9/97	7/7/97	0.92	4.20	4.57	21,000
Quality assurance	4/3/97	7/28/98	15.80	5.50	0.35	27,500
Coding	5/15/97	4/15/98	9.01	64.44	7.15	322,200
Reuse acquisition	7/1/97	7/13/97	0.39	0.29	0.74	1,450
Code inspections	11/1/97	3/25/98	3.37	11.60	3.44	58,000
Unit test	11/13/97	4/8/98	4.89	5.19	1.06	25,950
Function test	1/30/98	5/5/98	5.01	16.08	3.21	80,400
System test	2/13/98	4/25/98	4.07	20.57	5.05	102,850
Field test	4/20/98	6/1/98	1.45	4.28	2.95	21,400
User documents	11/15/97	5/25/97	6.20	26.70	4.31	133,500
Document reviews	2/1/98	5/5/98	5.65	5.27	0.93	26,350
Installation	4/15/98	7/20/98	3.10	13.43	4.33	67,150
Average staff level					14.47	
Overlapped schedule			18.24			
Waterfall schedule			120.18			
Paid effort and costs				264.00		1,320,000
Unpaid overtime				41.64		
Total effort				305.64		
Cost per function point						1,320.00
Cost per SLOC					10.56	

where the project is either better or worse than industry norms is a great asset for process-improvement work.

In spite of the advantages derived from using software cost-estimation tools, surveys by the author at software project management and metrics conferences indicate that the most accurate forms of software cost estimation are not the most widely used.

The frequency of use among a sample of approximately 500 project managers interviewed during 2004 and 2005 is as shown in Table 3.4.

TABLE 3.4 Frequency of Usage of Software Cost-Estimating Methods

Estimating methodology	Project management usage
Manual software estimating	42%
Automated software estimating	58%
Total	100%

The fact that manual estimating methods, which are known to be inaccurate, are still rather widely utilized is one of the more troubling problems of the software project management domain.

Comparison of Manual and Automated Estimates for Large Software Projects

A comparison by the author of 50 manual estimates with 50 automated estimates for projects in the 5000–function point range showed interesting results. The manual estimates were created by project managers who used calculators and spreadsheets. The automated estimates were also created by project managers or their staff estimating assistants using several different commercial estimating tools. The comparisons were made between the original estimates submitted to clients and corporate executives, and the final accrued results when the applications were deployed.

Only four of the manual estimates were within 10 percent of actual results. Some 17 estimates were optimistic by between 10 percent and 30 percent. A dismaying 29 projects were optimistic by more than 30 percent. That is to say, manual estimates yielded lower costs and shorter schedules than actually occurred, sometimes by significant amounts. (Of course several revised estimates were created along the way. But the comparison was between the initial estimate and the final results.)

By contrast 22 of the estimates generated by commercial software-estimating tools were within 10 percent of actual results. Some 24 were conservative by between 10 percent and 25 percent. Three were conservative by more than 25 percent. Only one automated estimate was optimistic, by about 15 percent.

(One of the problems with performing studies such as this is the fact that many large projects with inaccurate estimates are cancelled without completion. Thus for projects to be included at all, they had to be finished. This criterion eliminated many projects that used both manual and automated estimation.)

Interestingly, the manual estimates and the automated estimates were fairly close in terms of predicting coding or programming effort.

But the manual estimates were very optimistic when predicting requirements growth, design effort, documentation effort, management effort, testing effort, and repair and rework effort. The conclusion of the comparison was that both manual and automated estimates were equivalent for actual programming, but the automated estimates were better for predicting noncoding activities.

This is an important issue for estimating large software applications. For software projects below about 1000 function points in size (equivalent to 125,000 C statements), programming is the major cost driver, so estimating accuracy for coding is a key element. But for projects above 10,000 function points in size (equivalent to 1,250,000 C statements), both defect removal and production of paper documents are more expensive than the code itself. Thus accuracy in estimating these topics is a key factor.

Software cost and schedule estimates should be accurate, of course. But if they do differ from actual results, it is safer to be slightly conservative than it is to be optimistic. One of the major complaints about software projects is their distressing tendency to overrun costs and planned schedules. Unfortunately, both clients and top executives tend to exert considerable pressures on managers and estimating personnel in the direction of optimistic estimates. Therefore a hidden corollary of successful estimation is that the estimates must be defensible. The best defense is a good collection of historical data from similar projects.

References

Barrow, Dean, Susan Nilson, and Dawn Timberlake: *Software Estimation Technology Report*, Air Force Software Technology Support Center, Hill Air Force Base, Utah, 1993.

Boehm, Barry: *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

———: *Software Cost Estimation with COCOMO II*, Prentice-Hall, Englewood Cliffs, N.J.; 2000.

Brown, Norm (ed.): *The Program Manager's Guide to Software Acquisition Best Practices*, Version 1.0, U.S. Department of Defense, Washington, D.C., July 1995.

Charette, Robert N. *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, 1989.

———: *Application Strategies for Risk Analysis*, McGraw-Hill, New York, 1990.

Cohn, Mike: *Agile Estimating and Planning* (Robert C. Martin Series), Prentice-Hall PTR, Englewood Cliffs, N.J., 2005.

Coombs, Paul: *IT Project Estimation: A Practical Guide to the Costing of Software*, Cambridge University Press, Melbourne, Australia, 2003.

DeMarco, Tom: *Controlling Software Projects*, Yourdon Press, New York, 1982.

———: *Deadline*, Dorset House Press, New York, 1997.

Department of the Air Force: *Guidelines for Successful Acquisition and Management of Software Intensive Systems*; vols. 1 and 2, Software Technology Support Center, Hill Air Force Base, Utah, 1994.

Dreger, Brian: *Function Point Analysis*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

Galorath, Daniel D. and Michael W. Evans: *Software Sizing*, *Estimation*, *and Risk Management*, Auerbach, Philadelphia PA, 2006.

Garmus, David, and David Herron: *Measuring the Software Process: A Practical Guide to Functional Measurement*, Prentice-Hall, Englewood Cliffs, N.J., 1995.
———: *Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesely, Boston, Mass., 2001.
Grady, Robert B.: *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
——— and Deborah L. Caswell: *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, Englewood Cliffs, N.J., 1987.
Gulledge, Thomas R., William P. Hutzler, and Joan S. Lovelace (eds.): *Cost Estimating and Analysis—Balancing Technology with Declining Budgets*, Springer-Verlag, New York, 1992.
Howard, Alan (ed.): *Software Metrics and Project Management Tools*, Applied Computer Research (ACR), Phoenix, Ariz., 1997.
IFPUG Counting Practices Manual, Release 4, International Function Point Users Group, Westerville, Ohio, April 1995.
Jones, Capers: *Critical Problems in Software Measurement*, Information Systems Management Group, 1993a.
———: *Software Productivity and Quality Today—The Worldwide Perspective*, Information Systems Management Group, 1993b.
———: *Assessment and Control of Software Risks*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
———: *New Directions in Software Management*, Information Systems Management Group, ISBN 1-56909-009-2, 1994.
———: *Patterns of Software System Failure and Success*, International Thomson Computer Press, Boston, 1995.
———: *Applied Software Measurement*, 2d ed., McGraw-Hill, New York, 1996.
———: *The Economics of Object-Oriented Software*, Software Productivity Research, Burlington, Mass., April 1997a.
———: *Software Quality—Analysis and Guidelines for Success*, International Thomson Computer Press, Boston, 1997b.
———: *The Year 2000 Software Problem—Quantifying the Costs and Assessing the Consequences*, Addison-Wesley, Reading, Mass., 1998.
———: *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, Boston, Mass, 2000.
Kan, Stephen H.: *Metrics and Models in Software Quality Engineering, 2nd edition*, Addison-Wesley, Boston, Mass., 2003.
Kemerer, C. F.: "Reliability of Function Point Measurement—A Field Experiment," *Communications of the ACM*, **36:** 85–97 (1993).
Keys, Jessica: *Software Engineering Productivity Handbook*, McGraw-Hill, New York, 1993.
Laird, Linda M. and Carol M. Brennan: *Software Measurement and Estimation: A practical Approach*; John Wiley & Sons, New York, 2006.
Lewis, James P.: *Project Planning, Scheduling & Control*, McGraw-Hill, New York, 2005.
Marciniak, John J. (ed.): *Encyclopedia of Software Engineering*, vols. 1 and 2, John Wiley & Sons, New York, 1994.
McConnell, Steve: *Software Estimation: Demystifying the Black Art*, Microsoft Press, Redmond, WA, 2006.
Mertes, Karen R.: *Calibration of the CHECKPOINT Model to the Space and Missile Systems Center (SMC) Software Database (SWDB)*, Thesis AFIT/GCA/LAS/96S-11, Air Force Institute of Technology (AFIT), Wright-Patterson AFB, Ohio, September 1996.
Ourada, Gerald, and Daniel V. Ferens: "Software Cost Estimating Models: A Calibration, Validation, and Comparison," in *Cost Estimating and Analysis: Balancing Technology and Declining Budgets*, Springer-Verlag, New York, 1992, pp. 83–101.
Perry, William E.: *Handbook of Diagnosing and Solving Computer Problems*, TAB Books, Blue Ridge Summit, Pa., 1989.
Pressman, Roger: *Software Engineering: A Practitioner's Approach with Bonus Chapter on Agile Development*, McGraw-Hill, New York, 2003.

Putnam, Lawrence H.: *Measures for Excellence—Reliable Software on Time, Within Budget:* Yourdon Press/Prentice-Hall, Englewood Cliffs, N.J., 1992.

———, and Ware Myers: *Industrial Strength Software—Effective Management Using Measurement*, IEEE Press, Los Alamitos, Calif., 1997.

Reifer, Donald (ed.): *Software Management*, 4th ed., IEEE Press, Los Alamitos, Calif., 1993.

Rethinking the Software Process, CD-ROM, Miller Freeman, Lawrence, Kans., 1996. (This CD-ROM is a book collection jointly produced by the book publisher, Prentice-Hall, and the journal publisher, Miller Freeman. It contains the full text and illustrations of five Prentice-Hall books: *Assessment and Control of Software Risks* by Capers Jones; *Controlling Software Projects* by Tom DeMarco; *Function Point Analysis* by Brian Dreger; *Measures for Excellence* by Larry Putnam and Ware Myers; and *Object-Oriented Software Metrics* by Mark Lorenz and Jeff Kidd.)

Rubin, Howard: *Software Benchmark Studies for 1997*, Howard Rubin Associates, Pound Ridge, N.Y., 1997.

Roetzheim, William H., and Reyna A. Beasley: *Best Practices in Software Cost and Schedule Estimation*, Prentice-Hall PTR, Upper Saddle River, N.J., 1998.

Stukes, Sherry, Jason Deshoretz, Henry Apgar, and Ilona Macias: *Air Force Cost Analysis Agency Software Estimating Model Analysis—Final Report*, TR-9545/008-2, Contract F04701-95-D-0003, Task 008, Management Consulting & Research, Inc., Thousand Oaks, Calif., September 1996.

Stutzke, Richard D.: *Estimating Software-Intensive Systems: Projects, Products, and Processes*; Addison-Wesley, Boston, Mass, 2005.

Symons, Charles R.: *Software Sizing and Estimating—Mk II FPA (Function Point Analysis)*, John Wiley & Sons, Chichester, U.K., 1991.

Wellman, Frank: *Software Costing: An Objective Approach to Estimating and Controlling the Cost of Computer Software*, Prentice-Hall, Englewood Cliffs, N.J., 1992.

Yourdon, Ed: *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*, Prentice-Hall PTR, Upper Saddle River, N.J., 1997.

Zells, Lois: *Managing Software Projects—Selecting and Using PC-Based Project Management Systems*, QED Information Sciences, Wellesley, Mass., 1990.

Zvegintzov, Nicholas: *Software Management Technology Reference Guide, Dorset House Press, New York, 1994.*