

Manual Software-Estimating Methods

This chapter includes a variety of rules of thumb that have been derived primarily from several thousand projects that utilized the traditional “waterfall” model of software development. The next chapter includes some rules of thumb derived from projects that use the Agile approaches, object-oriented methods, and some special situations such as deploying enterprise resource planning (ERP) applications.

The phrase *manual software cost estimation* refers to estimating methods that are simple enough that they can be performed mentally or, at least, using nothing more sophisticated than a pocket calculator.

Manual estimating methods remain widely used approaches to software cost estimating as the second edition of this book is being written. Manual estimating methods are useful for the following purposes:

- Early estimates before requirements are known
- Small projects needing only one or two programmers
- Low-value projects with no critical business impacts

However, there are a number of situations where manual estimates are not very useful and indeed may be hazardous:

- Contract purposes for software development or maintenance
- Projects larger than 100 function points or 10,000 source code statements
- Projects with significant business impact

Manual methods are quick and easy, but not very accurate. Accurate estimating methods are complex and require integration of many

kinds of information. When accuracy is needed, avoid manual estimating methods.

As pointed out in Chapter 4, a comparative study of 50 estimates produced manually and 50 estimates produced by commercial estimating tools noted two significant results:

- Manual estimates were wrong by more than 35 percent more than 75 percent of the time. The maximum errors exceeded 50 percent for both costs and schedules. The errors with manual estimates were always on the side of excessive optimism, or underestimating true costs and actual schedules.
- Automated estimates came within 5 percent of actual costs about 45 percent of the time. The maximum errors were about 30 percent. When errors occurred, they were usually on the side of conservatism, or estimating higher costs and longer schedules than actually occurred.

Of course, automated estimating tools can achieve optimistic results, too, if users do things such as exaggerate staff experience, understate application size or complexity, minimize paperwork and quality control, and ignore learning curves. These are common failures with manual estimates, and they can be carried over into the automated estimation domain, too.

In spite of the plentiful availability of commercial software-estimating tools, the author receives dozens of e-mail and phone messages containing requests for simple rules of thumb that can be used manually or with pocket calculators.

This chapter provides a number of rules of thumb that are interesting and sometimes useful. Readers are cautioned that rules of thumb are not suitable for formal estimates, major projects, or software with important schedule, cost, or business implications.

Rules of Thumb Based on Lines-of-Code-Metrics

For many years manual estimating methods were based on the lines-of-code (LOC) metrics, and a number of workable rules of thumb were developed for common procedural programming languages, such as Assembly, COBOL, FORTRAN, PASCAL, PL/I, and the like.

These LOC rules of thumb usually start with basic assumptions about coding productivity rates for various sizes of software projects, and then use ratios or percentages for other kinds of work, such as testing, design, quality assurance, project management, and the like.

Tables 6.1 and 6.2 illustrate samples of the LOC-based rules of thumb for procedural languages in two forms: Table 6.1 uses *months* as the unit

TABLE 6.1 Rules of Thumb Based on LOC-Metrics for Procedural Languages
(Assumes 1 work month = 132 work hours)

Size of program, LOC	Coding, LOC per month	Coding effort, months	Testing effort, %	Non-code effort, %	Total effort, months	Net LOC per months
1	2500	0.0004	10.00	10.00	0.0005	2083
10	2250	0.0044	20.00	20.00	0.0062	1607
100	2000	0.0500	40.00	40.00	0.0900	1111
1,000	1750	0.5714	50.00	60.00	1.2000	833
10,000	1500	6.6667	75.00	80.00	17.0000	588
100,000	1200	83.3333	100.00	100.00	250.0000	400
1,000,000	1000	1000.0000	125.00	150.00	3750.0000	267

for work, while Table 6.2 uses *hours* as the unit for work. Both hourly and monthly work metrics are common in the software literature, with the hourly form being common for small programs and the monthly form being common for large systems.

Both tables show seven size ranges for software, with each one being an order of magnitude larger than its predecessor.

The column labeled “Testing Effort, %” denotes the relative amounts of time for testing versus coding. As can be seen, the larger the project, the greater the amount of testing required.

The column labeled “Non-code Effort, %” denotes the host of activities other than pure programming that are associated with software projects and need to be included in the estimate:

- Requirements definition
- External design
- Internal design
- Change management

TABLE 6.2 Rules of Thumb Based on LOC-Metrics for Procedural Languages
(Assumes 1 work month = 132 work hours)

Size of program, LOC	Coding, LOC per hour	Coding effort, hours	Testing effort, %	Non-code effort, %	Total effort, hours	Net LOC per hour
1	18.94	0.05	10.00	10.00	0.06	15.78
10	17.05	0.59	20.00	20.00	0.82	12.18
100	15.15	6.60	40.00	40.00	11.88	8.42
1,000	13.26	75.43	50.00	80.00	173.49	5.76
10,000	11.36	880.00	75.00	100.00	2,420.00	4.13
100,000	9.09	11,000.00	100.00	150.00	38,500.00	2.60
1,000,000	7.58	132,000.00	125.00	150.00	495,000.00	2.02

- User documentation
- Project management

As software applications grow in size, a larger and larger proportion of total effort must be devoted to paperwork and other non-coding activities.

As can be seen, the *monthly* form of normalizing effort is fine for large systems but is not very convenient for the smaller end of the spectrum. Conversely, the *hourly* form is inconvenient at the large end.

Also, the assumption that a work month comprises 132 hours is a tricky one, since the observed number of hours actually worked in a month can run from less than 120 to more than 170. Because the actual number of hours varies from project to project, company to company, and country to country, it is best to replace the generic rate of 132 with an actual or specific rate derived from local conditions and work patterns.

While LOC-based estimating rules of thumb served for many years, they are rapidly dropping from use because software technologies have changed so much that it is difficult, and even dangerous, to apply them under some conditions.

Usage of the LOC-metrics obviously assumes the existence of some kind of procedural programming language where programmers develop code using some combination of alphanumeric information, which is the way such languages as COBOL, C, FORTRAN, and hundreds of others operate.

However, the development of Visual Basic and its many competitors, such as Realizer, has changed the way many modern programs are developed. Although these visual languages do have a procedural source code portion, quite a bit of the more complex kinds of “programming” are done using button controls, pull-down menus, visual worksheets, and reusable components. In other words, programming is being done without anything that can be identified as a *line of code* for measurement or estimation purposes.

Also, the object-oriented programming languages and methods, such as Objective C, Smalltalk, Eiffel, and the like, with their class libraries, inheritance, and polymorphism, have also entered a domain where attempting to do estimates using conventional lines of code is not a very effective approach.

As the 21st century unfolds, the volume of programming done using languages where the LOC-metrics no longer works well for estimating purposes is rising rapidly. By perhaps 2025 more than 70 percent of the new software applications will be developed using either object-oriented languages or visual languages, or both.

Over and above the fact that the LOC-metrics is difficult to apply for many modern programming languages, there are deep and serious

TABLE 6.3 Rank Order of Large System Software Cost Elements

1. Defect removal (inspections, testing, and finding and fixing bugs)
2. Producing paper documents (plans, specifications, and user manuals)
3. Meetings and communication (clients, team members, and managers)
4. Programming or coding
5. Project management
6. Change management

economic problems with attempting to use LOC-metrics for measurement and estimation purposes.

Most of the important kinds of software, such as operating systems, billing systems, aircraft navigation systems, word processors, spreadsheets, and the like, are quite large compared to the sizes of applications built 20 years ago.

Twenty years ago, a programming application of 100,000 LOC was viewed as rather large. Even IBM's main operating system, OS/360, was only about 1 million LOC in size during its first year of release, although the modern incarnation, MVS, now tops 10 million LOC.

Today, a size of 100,000 LOC is more or less an entry-level size for a modern Windows XP application. Many software packages can top 1,000,000 LOC, while things like major operating systems are in the 20-million-LOC domain and up.

For large systems in the 1 million–source code statement size range, programming itself is only the fourth most expensive activity. The three higher-cost activities cannot really be measured or estimated effectively using the LOC-metrics. Also, the fifth and sixth major cost elements, project management and change management, cannot easily be estimated or measured using the LOC-metrics either. (See Table 6.3.)

If these non-coding activities are not measured separately, the overall project can be measured using LOC, but such measures have a disturbing property: They will penalize high-level languages and make low-level languages seem to have higher productivity rates than high-level languages.

As can easily be seen, the usefulness of a metric such as lines of code, which can only measure and estimate one out of the six major software cost elements, is a significant barrier to economic understanding.

Rules of Thumb Based on Ratios and Percentages

Software projects obviously include a lot more kinds of work than just coding. A fairly large family of manual estimating methods has developed based on the use of ratios and percentages.

These rules assume a *cascade* form of estimation. First, the overall project is estimated in its entirety. Second, ratios and percentages are applied to apportion the overall effort into the desired sets of phases or activities.

The basic problem with ratio-based estimation is the false assumption that there are constant ratios between coding and other key activities, such as testing, project management, integration, and the like. In fact, the ratios vary significantly, based on five sets of independent variables:

- The class of the application (systems software, information systems, web-based, etc.)
- The size of the application
- The programming language or languages utilized
- The presence or absence of reusable materials
- The methodology used such as Agile methods, waterfall methods, spiral methods etc.

The complexity of the interactions of these five sets of variables is why commercial software cost-estimating tools contain hundreds of rules and adjustment factors.

There are too many combinations of factors to illustrate all of them, but it is instructive to view how the percentages applied to key software activities vary in response to the class and size of the application.

Table 6.4 uses only seven major activities in order to highlight the percentage of coding effort against the background of non-coding activities.

Table 6.4 shows some of the changes in ratios among five different classes of applications. Note that for end-user applications there is very little work except coding and testing, because when users are building their own software they obviously know what their own requirements are, and they don't need personal user's guides.

TABLE 6.4 Percentages of Development Effort by Software Class
(Assumes application of 1000 function points or 100,000 source code statements)

Activity	End-user projects	MIS projects	Systems projects	Commercial projects	Military projects
Requirements definition	0	7	8	4	10
Design	10	12	15	10	15
Coding	60	25	18	25	18
Testing	30	30	30	36	22
Change management	0	6	10	5	12
Documentation	0	8	7	10	10
Project management	0	12	12	10	13
Total	100	100	100	100	100

By contrast, for military software projects, the activities associated with requirements, design, and documentation total roughly twice the effort devoted to coding itself: 35 percent versus 18 percent.

When we consider ratios based on various size ranges of applications, we also see huge differences that prevent any single ratio or percentage from being a safe general estimating method.

Table 6.5 shows five size plateaus an order of magnitude apart, starting with small applications of 10 function points (roughly 1000 source code statements) and running up to enormous systems of 100,000 function points (roughly 10 million source code statements).

At the small end of the spectrum, coding is the dominant activity in terms of effort, but at the large end of the spectrum, coding itself is only 15 percent of the total effort. The main cost drivers at the large end of the spectrum are the activities associated with finding and fixing bugs and the activities associated with producing various kinds of paper documents, such as requirements, plans, specifications, and user manuals.

At the intersection between rules of thumb based on ratios and those based on lines of code is an important topic that is not well covered in the software literature: the amount of reusable source code taken from similar applications or from other sources.

Table 6.6 gives reuse percentages noted among SPR's clients for a variety of programming languages. It is interesting that in spite of the emphasis that the object-oriented community places on software reuse, the language that typically has the largest volume of reusable code for common applications is Visual Basic.

When reusable material is present in substantial volumes during a software development project, it will obviously affect the schedule, effort, and cost results from an estimating standpoint.

Software reusability is an important topic with many poorly understood aspects. For example, software reuse is only valuable if the reused material approaches zero-defect quality levels. Reusing poor-quality source code is dangerous and expensive.

TABLE 6.5 Percentages of Development Effort by Software Size
(Assumes procedural languages, such as COBOL or C)

Activity	10-FP projects	100-FP projects	1000-FP projects	10,000-FP projects	100,000 FP projects
Requirements definition	5	5	7	8	9
Design	5	6	10	12	13
Coding	50	40	30	20	15
Testing	27	29	30	33	34
Change management	1	4	6	7	8
Documentation	4	6	7	8	9
Project management	8	10	10	12	12
Total	100	100	100	100	100

TABLE 6.6 Approximate Amount of Reusable Code by Language

Language	Average reuse, %
Visual Basic	60.00
Eiffel	55.00
Smalltalk	50.00
JAVA	50.00
Objective C	45.00
Ada 95	35.00
C++	27.50
SQL	25.00
Ada	25.00
COBOL	17.50
FORTRAN	15.00
Macro assembly	15.00
C	12.50
Pascal	12.50
Jovial	10.00
CMS2	10.00
PL/I	7.50
Basic assembly	5.00
Average	26.15

The ratios for software reuse shown in Table 6.6 are only approximate and can vary widely. For example, there can be Visual Basic applications with less than 10 percent reused code, while some basic assembly applications can top 40 percent in terms of code reuse.

In general, the languages that endorse and facilitate reuse, such as the object-oriented family of languages and the Visual Basic family, have far more sources of reusable code than do such languages as COBOL, where reuse is merely accidental.

The overall conclusion is that the use of simplistic ratios and percentages for software cost estimating is a very hazardous practice unless the estimator has experience-based ratios derived from projects of the same size, same class, same programming language, and that utilize the same volume of reusable materials.

There is no single set of ratios or percentages that can be universally applied to software projects. This lack of universal constants is true in every other human activity. Knowing the average cost of buying an automobile in the United States is obviously irrelevant to buying your own personal automobile. You start with the kind of automobile and set of features that you want, check the averages for that combination, and then try to negotiate with dealers for a better cost.

It is astonishing that project managers would use simple ratios and percentages for software projects that cost millions of dollars, and yet

use much more sophisticated costing techniques for buying automobiles, home appliances, houses, clothes, food, or any other personal purchases.

Rules of Thumb Based on Function Point Metrics

Since function point metrics were first published in the late 1970s, their use has swept through the software world, and function points are now among the most widely used of any metric in all countries that develop significant volumes of software, such as the United States, Japan, South Korea, China, Germany, Australia, and Canada.

Function point metrics solve some of the more difficult problems for software cost estimation and cost measurement, and it is interesting to explore the origin and evolution of functional metrics.

In the middle 1970s, the IBM corporation was the world's largest and most successful producer of mainframe software. However, IBM's success and innovation led to some new and unexpected problems.

By the middle 1970s, IBM's software community was topping 25,000 members, and the costs of building and maintaining software were becoming a significant portion of the costs and budgets for new products.

Programming languages were exploding in number, and within IBM applications were being developed in assembly language, APL, COBOL, FORTRAN, RPG, PL/I, PL/S (a derivative of PL/I), and perhaps a dozen others. Indeed, many software projects at IBM and elsewhere used several languages concurrently, such as COBOL, RPG, and SQL, as part of the same system.

Also, the sizes of IBM's average software applications were growing from the 10,000 lines of code typical for the old IBM 1401 computers to well over 100,000 lines of code for the IBM 360 and IBM 370 computers.

The combination of larger applications and dozens of programming languages meant that manual estimates based on the lines-of-code-metrics were causing cost and schedule overruns of notable proportions. The problems of IBM in this era are captured very well in Dr. Fred Brooks's classic book *The Mythical Man-Month* (Brooks 1974), which was revised and reissued to commemorate the 20th anniversary of its first publication.

Allan J. Albrecht and his colleagues at IBM White Plains were tasked with attempting to develop an improved methodology for sizing, estimating, and measuring software projects. The method they developed is now known as *function point analysis*, and the basic metric they developed is termed a *function point*.

Although the actual rules for counting function points are rather complex, the essential concepts behind the function point metric are

simple: Base the size of applications on external characteristics that do not change because of the programming language or languages used.

In essence, a *function point* consists of the weighted totals of five external aspects of software applications:

- The types of *inputs* to the application
- The types of *outputs* that leave the application
- The types of *inquiries* that users can make
- The types of *logical files* that the application maintains
- The types of *interfaces* to other applications

In October 1979, Allan Albrecht presented the function point metric at a conference in Monterey, California, sponsored jointly by IBM and two IBM user groups, SHARE and GUIDE. Concurrently, IBM placed the basic function point metric into the public domain.

What Albrecht and his colleagues at IBM were attempting to do was to create a software metric that had the following ten important design goals:

- The metric can be used to measure software productivity.
- The metric can be used to measure software quality.
- The metric can be used to measure software in any known programming language.
- The metric can be used to measure software in any combination of languages.
- The metric can be used to measure all classes of software (real-time, MIS, systems, etc.)
- The metric can be used to measure any software task or activity and not just coding.
- The metric can be used in discussions with clients.
- The metric can be used for software contracts.
- The metric can be used for large-scale statistical analysis.
- The metric can be used for value analysis.

On the whole, the function point metric meets all ten goals fairly well. This is not to say that function points have no problems of their own, but they meet the ten criteria far better than the older lines-of-code-metrics. Indeed, given the advent of visual languages and object-oriented software, the lines-of-code-metrics does not currently meet any of the ten criteria at all.

As the usage of function points expanded, a nonprofit organization, the International Function Point Users Group (IFPUG), was formed. IFPUG has now become one of the largest measurement associations in the world, and there are affiliated organizations in at least 20 other countries. (For additional information on this group, readers can visit their web site, www.IFPUG.org.)

IFPUG took over responsibility from IBM for modernizing and updating the basic counting rules for function points, which have gone through a number of major and minor sets of revisions. On average, IFPUG issues a major revision about every three years, and minor revisions once or twice a year. The most common reason for a revision is to include new kinds of software or new environments, such as web-based applications or embedded applications.

This chapter contains a number of simple rules of thumb that cover various aspects of software development and maintenance. The rules assume the Version 4.5 function point counting rules published by IFPUG. Adjustments would be needed for the British Mark II function point rules, for COSMIC function point rules, or the older IFPUG rules. Also, these rules would need adjustments for some of the many function point variations, such as the Boeing 3D function point, object points, use-case points, feature points, or the DeMarco *bang* function point.

However, rules of thumb are far too limited to cover every aspect of software development. So this section also discusses the usage of simple table-driven *templates* that allow somewhat finer levels of estimation. Here, too, the table-driven method is not as accurate as a commercial estimating tool, but can be performed quickly and can provide a rough check on more formal and rigorous estimating approaches.

Strong cautions must be given yet again:

- Simple rules of thumb are *not* accurate.
- Simple rules of thumb should *not* be used for contracts, bids, or serious business purposes.
- No manual software-estimating methodology can give rapid response when assumptions change and requirements creep.

The following rules of thumb are known to have a high margin of error. They are being published in response to many requests to the author for simple methods that can be used manually or with pocket calculators. Also, an understanding of the limitations of manual estimating methods can give a greater appreciation for the need for more formal automated methods.

The best that can be said is that the rules of thumb are easy to use, and can provide a “sanity check” for estimates produced by other and hopefully more rigorous methods.

Function Point Sizing Rules of Thumb

Predicting the sizes of various deliverables is the usual starting point for software cost estimating. The function point metric has transformed sizing from a very difficult task into one that is now both easy to perform and comparatively accurate, although the accuracy of early sizing methods applied prior to the availability of information that can lead to function point analysis is not great.

Sizing Function Point Totals Prior to Completion of Requirements

It often happens that software cost estimates are required long before there is enough solid information to actually create an accurate estimate. Since the function point metric is the basis for so many subsequent estimating stages, it is very difficult to produce a reasonable software cost estimate prior to the completion of the requirements, which comprise the first software document with enough information to derive function point totals.

However, there are function point sizing methods that can be used to create a rough approximation of function point totals long before requirements are complete, although this method has a high margin of error.

Software Productivity Research (SPR) uses a multipart taxonomy for defining software projects in terms of scope, class, and type, in order to identify a project when entering information into the software cost-estimating tools.

These project-identification checklists are organized using more or less the same principle as the Richter scale for earthquakes; that is, the larger numbers have more significance than the smaller numbers.

This property can be utilized to produce very early size approximations of function points before almost any other facts are known about the software projects in question. Admittedly, this crude form of sizing is far too inaccurate for serious cost-estimating purposes, but it has the virtue of being usable before any other known form of sizing is possible.

To use the scope, class, and type of taxonomy for guessing at the approximate size of the software, it is only necessary to sum the list values for the scope, class, and type and raise the total to the 2.35 power. This calculation sequence will yield a rough approximation of function points, assuming IFPUG Version 4 is the counting method. Table 6.7 summarizes the scope, class, and type lists in numeric order.

The kind of information needed to use this sizing approximation method is usually known on the very first day that requirements begin, so very early sizing is possible even if the size approximation

TABLE 6.7 Examples of Scope, Class, and Type Values

Scope	Class	Type
1 Subroutine	1 Individual software	1 Nonprocedural
2 Module	2 Shareware	2 Web applet
3 Reusable module	3 Academic software	3 Batch
4 Disposable prototype	4 Single location—internal	4 Interactive
5 Evolutionary prototype	5 Multilocation—internal	5 Interactive GUI or web-based
6 Standalone program	6 Contract project—civilian	6 Batch database
7 Component of system	7 Time sharing	7 Interactive database
8 Release of system	8 Military services	8 Client/server
9 New system	9 Internet	9 Mathematical
10 Compound system	10 Leased software	10 Systems
	11 Bundled software	11 Communications
	12 Marketed commercially	12 Process control
	13 Outsource contract	13 Trusted system
	14 Government contract	14 Embedded
	15 Military contract	15 Image processing
		16 Multimedia
		17 Robotics
		18 Artificial intelligence
		19 Neural net
		20 Hybrid: mixed

is rather imprecise. To utilize this rough sizing method, it is only necessary to do three things:

- Apply the numeric list values to the project to be sized in terms of the scope, class, and type factors.
- Sum the numeric values from the three lists.
- Raise the total to the 2.35 power.

For example, assume you are building an application with the following three attributes:

Scope = 6 (standalone program)
 Class = 4 (internal—single site)
 Type = 8 (client/server)
 Sum = 18

Raising 18 to the 2.35 power yields a value of 891, which can be utilized as a very rough approximation of function points, assuming the IFPUG Version 4.1 rules. Incidentally, client/server applications are often in the 1000–function point range, so this is not a bad starting point.

Let us try this method again on a different form of application. Suppose you were building a small personal application with the following properties:

Scope = 4 (disposable prototype)
 Class = 1 (individual program)
 Type = 1 (nonprocedural)
 Sum = 6

Raising 6 to the 2.35 power gives a value of 67, which can serve as a rough approximation of the application's function point total. Here, too, since most personal applications are less than 100 function points in size, this is not a bad way of beginning even if the true size will vary.

Applying this same method to a more significant military software project, the results might be the following:

Scope = 9 (new system)
 Class = 15 (military contract)
 Type = 13 (trusted system)
 Sum = 37

Raising 37 to the 2.35 power gives a value of 4844 function points, which again can serve as a rough approximation of the application's function point total until enough information is available to perform a proper function point analysis. Here, too, military software projects at the system level are often in the 5000–function point range in size (or larger), so this approximation method is enough to see that the application will not be a trivial one.

This very crude function point size approximation method is not recommended for any purpose other than estimating size prior to full requirements definition, when almost nothing is known about a software project and some form of sizing is needed to complete an early estimate.

Incidentally, it would be possible to use this taxonomy with other metrics such as object points, Cosmic function points, or engineering function points. The only change would be that the combination would have to use different powers to match the other metrics. However, those powers are not known by the author, so the groups that control these other metrics would have to provide such information.

It should be noted that the SPR scope, class, and type lists are changed from time to time to add new kinds of software or to adjust the rank placements as empirical evidence suggests that changes are needed. These changes mean that users of the methods should feel free to experiment with other power settings besides 2.35, or even to develop their own list sequences. This kind of early sizing is essentially a form of pattern matching. Now that thousands of software projects exist, this method uses only a basic taxonomy to attempt to find applications with similar patterns. It is theoretically possible that this pattern matching approach

could be extended to bring in historical data from such projects, and even to provide links to reusable components from similar projects.

The fundamental operating principle of this early sizing method is that useful information about software projects can be derived from a taxonomy that can rank a software application in a fashion that allows rough size information to be derived merely from the ranking itself.

Even the first column, or scope portion, of this taxonomy can be used for rough sizing purposes. Table 6.8 shows the “average” sizes of various kinds of projects, assuming a rough expansion factor of 100 logical source code statements for every function point.

In Table 6.8 most of the terms are self-explanatory. However, item 10, *compound system*, is a large system that is actually comprised of several systems linked together. An example of a compound system would be the SAP R3 integrated business suite, although that is much larger than 15,000 function points (closer to 250,000 function points). It is comprised of a number of linked *systems*, which taken together constitute the entire SAP product.

A modern example of a compound system would be Microsoft Office, which comprises the separate applications of word processing, spreadsheet, database, graphics, and personal scheduler, all integrated and working together. In fact, Microsoft Office is in the 25,000–function point size range, with each of the individual applications running about 3000 to 5000 function points, plus the OLE capabilities to share information making up the total.

Because triangulating a software project in terms of its scope, class, and type is possible as early as the first day that requirements definition begins, this method can be utilized very early in a development cycle, long before any other kind of information is available.

Users are urged to experiment with their own ranking systems and taxonomies, and also with varying the power used to achieve the function point approximations.

TABLE 6.8 Size Approximations Using Only the Scope Factor

Application scope	Size, function points	Size, source lines of code
1 Subroutine	1	100
2 Module	3	300
3 Reusable module	5	500
4 Disposable prototype	7	700
5 Evolutionary prototype	10	1,000
6 Standalone program	25	2,500
7 Component of system	100	10,000
8 Release of system	5,000	500,000
9 New system	10,000	1,000,000
10 Compound system	25,000	2,500,000

Sizing by Analogy

Another form of rapid sizing is simply browsing a list of the sizes of applications that have been measured and selecting one or more similar projects to serve as an approximate size basis for the new project that is about to be estimated, as shown by Table 6.9.

TABLE 6.9 Approximate Sizes of Selected Software Applications
(Sizes based on IFPUG Version 4 and SPR logical statement rules)

Application	Type	Purpose	Primary language	Size, KLOC	Size, FP	LOC per FP
Graphics Design	Commercial	CAD	Objective C	54	2,700	20.00
IEF	Commercial	CASE	C	2,500	20,000	125.00
Visual Basic	Commercial	Compiler	C	375	3,000	125.00
IMS	Commercial	Database	Assembly	750	3,500	214.29
CICS	Commercial	Database	Assembly	420	2,000	210.00
Lotus Notes	Commercial	Groupware	Mixed	350	3,500	100.00
MS Office Proj.	Commercial	Office tools	C	2,000	16,000	125.00
SmartSuite	Commercial	Office tools	Mixed	2,000	16,000	125.00
MS Office SB	Commercial	Office tools	C	1,250	10,000	125.00
Word 7.0	Commercial	Office tools	C	315	2,500	126.00
Excel 6.0	Commercial	Office tools	C	375	2,500	150.00
MS Project	Commercial	Project management	C	375	3,000	125.00
KnowledgePlan	Commercial	Project management	C++	134	2,500	56.67
CHECKPOINT	Commercial	Project management	Mixed	225	2,100	107.14
Function Point Control	Commercial	Project management	C	56	450	125.00
SPQR/20	Commercial	Project management	Quick Basic	25	350	71.43
WMCCS	Military	Defense	Jovial	18,000	175,000	102.86
Aircraft Radar	Military	Defense	Ada 83	213	3,000	71.00
Gun Control	Military	Defense	CMS2	250	2,336	107.00
Airline Reservation	MIS	Business	Mixed	2,750	25,000	110.00
Insurance Claims	MIS	Business	COBOL	1,605	15,000	107.00
Telephone billing	MIS	Business	C	1,375	11,000	125.00
Tax Preparation (Personal)	MIS	Business	Mixed	180	2,000	90.00
General Ledger	MIS	Business	COBOL	161	1,500	107.00
Order Entry	MIS	Business	COBOL/ SQL	106	1,250	85.00
Human Resource	MIS	Business	COBOL	128	1,200	107.00
Sales Support	MIS	Business	COBOL/ SQL	83	975	85.00
Budget Preparation	MIS	Business	COBOL/ SQL	64	750	85.00

TABLE 6.9 Approximate Sizes of Selected Software Applications (Continued)
(Sizes based on IFPUG Version 4 and SPR logical statement rules)

Application	Type	Purpose	Primary language	Size, KLOC	Size, FP	LOC per FP
Windows XP	Systems	Operating system	C	25,000	85,000	129.41
MVS	Systems	Operating system	Assembly	12,000	55,000	218.18
UNIX V5	Systems	Operating system	C	6,250	50,000	125.00
DOS 5	Systems	Operating system	C	1,000	4,000	250.00
5ESS	Systems	Telecommunication	C	1,500	12,000	125.00
System/12	Systems	Telecommunication	CHILL	800	7,700	103.90
Total				68,669	542,811	126.51
Average				2,020	15,965	126.51

Sizing by analogy is a feature of automated cost-estimating models, but as more and more projects are measured, it will be increasingly common to see published tables of applications sizes using both function points and source code statements. Table 6.9 gives an example of such a table.

Sizing Source Code Volumes

Now that thousands of software projects have been measured using both function points and lines of code (LOC), empirical ratios have been developed for converting LOC data into function points, and vice versa. The following rules of thumb are based on *logical statements* rather than *physical lines*.

(The physical LOC-metrics has such wide and random variations from language to language and programmer to programmer that it is not suited for sizing, for estimating, or for any other serious purpose.)

Rule 1: Sizing Source Code Volumes

- One function point = 320 statements for basic assembly language
- One function point = 213 statements for macro assembly language
- One function point = 128 statements for the C programming language
- One function point = 107 statements for the COBOL language
- One function point = 107 statements for the FORTRAN language
- One function point = 80 statements for the PL/I language
- One function point = 71 statements for the ADA 83 language
- One function point = 53 statements for the C++ language
- One function point = 50 statements for the JAVA language
- One function point = 15 statements for the Smalltalk language

The overall range of noncommentary logical source code statements to function points ranges from more than 300 statements per function

point for basic assembly language to less than 15 statements per function point for object-oriented languages and many program generators.

However, since many procedural languages, such as C, Cobol, Fortran, and Pascal, are close to the 100-to-1 mark, that value can serve as a rough conversion factor for the general family of procedural source code languages.

For object-oriented programming languages with full class libraries, such as Actor, Eiffel, Objective-C, C++, JAVA, and Smalltalk, the range is from perhaps 14 to about 50 statements per function point, and a value such as 20 statements per function point can serve as a rough approximation.

These code-sizing rules of thumb have a high margin of error, and need specific adjustments based on individual *dialects* of programming languages. Also, individual programming styles can vary significantly. Indeed, in a controlled study within IBM where eight programmers implemented the same specification using the same language, a 5-to-1 variation in the number of source code statements was noted, based on individual interpretations of the specification.

Note that these rules for sizing source code can also be reversed. It often happens when dealing with aging legacy applications, and sometimes for small enhancement and maintenance projects, that source code volumes are known earlier in the development cycle than is possible to calculate function points.

Direct conversion from source code volumes to an equivalent count of function points is termed *backfiring*. Although the accuracy of backfiring is not great, because individual programming styles can cause wide variations in source code counts, it is easy and popular.

Indeed, for some aging legacy applications where the specifications are missing and the source code is the only remaining artifact, backfiring provides the only effective method for arriving at function point values.

There is another situation where backfiring is very popular, too: small enhancements and maintenance projects. The normal method of calculating function points has trouble with small projects below about 15 function points in size because the weighting factors have lower limits; this creates a floor at about 15 function points, below which the normal method ceases to work effectively.

However, the backfiring method of conversion between logical statements has no lower limit and can even be used for sizes as small as a fraction of a function point. For example, making a 1-source code statement change to a COBOL application has a size of about 0.001 function points. There is no way to calculate such a tiny fraction of a function point using normal methods.

Because maintenance and enhancement projects are very common, and because aging legacy applications are also common, the backfiring method for approximating function points is actually the most widely used method in the world for deriving function point totals.

Backfiring is a common feature of software cost-estimating tools, and is found in at least 30 commercial software cost-estimating tools in the United States, as well as in those developed in Europe and elsewhere.

Note that for backfiring to work well, the starting point should be *logical statements* rather than physical lines of source code. However, it is possible to cascade from physical lines to logical statements, although the accuracy is reduced somewhat.

Sizing Paper Deliverables

Software is a very paper-intensive industry. More than 50 kinds of planning, requirements, specification, and user-related document types can be created for large software projects. For many large systems, and especially for large military projects, producing paper documents costs far more than producing the source code.

The more common kinds of paper documents associated with software applications include:

- Requirements
- Cost estimates
- Development plans
- Functional specifications
- Change requests
- Logic or internal specifications
- Project status reports
- User manuals
- Test plans
- Bug or defect reports

Document types that occur from time to time, but not for every application, include:

- Contracts between clients and outsource vendors
- Use cases
- Design inspection reports
- Code inspection reports

- Quality assurance reports
- Translations of documents into other languages
- Training and tutorial materials
- Sales and marketing plans for commercial software

Paperwork volumes correlate fairly closely to the size of the application, using either function points or source code metrics. Small projects create comparatively few paper documents, and they themselves are not very large.

Also, some development methods tend to create more or less documentation than other methods. For example, the Agile development methods typically produce a very sparse or lean document set. At the other end of the spectrum, projects that are created using military standards or that require ISO certification tend to produce much larger document sets than average.

Overall, large systems in the 10,000–function point or 1 million–source code statement range often produce large numbers of documents, and some of these are very large indeed. For example, some sets of specifications have topped 60,000 pages.

For a few really large systems in the 100,000–function point range, the specifications can actually exceed the lifetime reading speed of a single person, and could not be finished even by reading eight hours a day for an entire career!

The following rule of thumb encompasses the sum of the pages that will be created in requirements, internal and external specifications, development and quality plans, test plans, user manuals, and other business-related software documents.

However, if you are following ISO 9000–9004 standards, you should use 1.17 as the power rather than 1.15. If you are following older military standards, such as DoD 2167, you should use 1.18 as the power rather than 1.15, because military projects produce more paperwork than any other kind of software application.

Rule 2: Sizing Software Plans, Specifications, and Manuals

Function points raised to the 1.15 power predict approximate page counts for paper documents associated with software projects.

A simple corollary rule can be used to predict the approximate volume of text that the pages will contain: Pages multiplied by 400 words per page predict the approximate number of English words created for the

normal page size used in the United States (i.e., 8.5- by 11-in paper stock) with single-spaced type, using a 12-point type such as Times Roman. The actual capacity is greater than 400 words, but the inclusion of graphics and tables must be factored in.

Obviously, adjustments must be made for European A4 paper, where the capacity is closer to 500 words, once again assuming that graphic elements and tables will also be present, and that a 12-point type is selected.

Paperwork is such a major element of software costs and schedules that it cannot safely be ignored. Indeed, one of the major problems with the LOC-metrics is that it tends to conceal both the volumes of paper deliverables and the high costs of software paperwork.

Sizing Creeping User Requirements

One of the most severe problems of the software world is that of dealing with the emergence of new and changing requirements after the completion of the initial requirements phase.

The function point metric is extremely useful in measuring the rate at which requirements creep. In fact, the usage of *cost per function point* is now starting to occur in software contracts and outsourcing agreements. For contract purposes, cost per function point is used with a sliding scale that becomes more expensive for features added later in the development cycle.

Rule 3: Sizing Creeping User Requirements

Creeping user requirements will grow at an average rate of 2 percent per month from the design through coding phases. On an average, software applications will grow by at least 15 percent during development.

Assume that you and your clients agree during the requirements definition to develop an application of exactly 1000 function points. This rule of thumb implies that every month thereafter during the subsequent design and coding phases the original requirements will grow by a rate of 20 function points.

Since the overall schedule for a generic 1000–function point project would be about 16 calendar months, and the design and coding phases would be about half that, or 8 calendar months, this rule implies that about 16 percent new features would be added due to creeping requirements. The final total for the application would be 1160 function points rather than the initial value of 1000 function points.

In real life, the observed range of creeping requirements ranges from close to 0 to more than 5 percent per month. The better requirements methods, such as joint application design (JAD), prototypes, and requirements inspections, can reduce the rate of creep to well below 1 percent per month.

However, quick requirements methods, such as those often found with rapid application development (RAD), the Agile approach, or client/server projects, can cause the rate of creep to top 5 percent per month and throw the project into turmoil.

Because internal information systems have no visible penalties associated with creeping requirements, they tend to grow almost uncontrollably. For contract software and outsourcing arrangements, there may be financial penalties for adding requirements late in the development cycle.

For software development contracts, perhaps the most effective way of dealing with changing user requirements is to include a sliding scale of costs in the contract itself. For example, suppose a hypothetical contract is based on an initial agreement of \$500 per function point to develop an application of 1000 function points in size, so that the total value of the agreement is \$500,000.

The contract might contain the following kind of escalating cost scale for new requirements added downstream:

Initial 1000 function points	\$500 per function point
Features added more than 3 months after contract signing	\$600 per function point
Features added more than 6 months after contract signing	\$700 per function point
Features added more than 9 months after contract signing	\$900 per function point
Features added more than 12 months after contract signing	\$1200 per function point
Features deleted or delayed at user request	\$150 per function point

Similar clauses can be utilized with maintenance and enhancement outsource agreements, on an annual or specific basis such as the following:

Normal maintenance and defect repairs	\$125 per function point per year
Mainframe to client/server conversion	\$200 per function point per system

The advantage of the use of function point metrics for development and maintenance contracts is that they are determined from the user requirements and cannot be unilaterally added or deleted by the contractor.

One of the many problems with the older LOC-metrics is that there is no objective way of determining the minimum volume of code needed to implement any given feature. This means that contracts based on cost per LOC could expand without any effective way for the client to determine whether the expansions were technically necessary.

Function points, on the other hand, cannot be unilaterally determined by the vendor and must be derived from explicit user requirements. Also, function points can easily be understood by clients, while the LOC-

metrics is difficult to understand in terms of why so much code is needed for any given contract.

Sizing Test-Case Volumes

The function point metric is extremely useful for test-case sizing, since the structure of function point analysis closely parallels the items that need to be validated by testing.

Rule 4: Sizing Test-Case Volumes

Function points raised to the 1.2 power predict the approximate number of test cases created.

Commercial software-estimating tools can predict the number of test cases for more than 15 discrete forms of testing, and can deal with the specifics of unit testing, new function testing, system testing, and all of the other varieties. This simple rule of thumb encompasses the sum of all test cases in all forms of testing, which is why rules of thumb should be used with caution.

A simple corollary rule can predict the number of times each test case will be run or executed during development: Assume that each test case would be executed approximately four times during software development.

Of course, there are at least 18 separate forms of testing used for software projects. In addition, companies are split between those where developers perform testing, where professional test personnel perform testing, and even where independent test organizations perform testing. The true complexity of testing requires much more work than a simple rule of thumb for acceptable precision in estimating results.

Sizing Software Defect Potentials

The *defect potential* of an application is the sum of bugs or errors that will occur in five major deliverables:

- Requirements errors
- Design errors
- Coding errors
- User documentation errors
- Bad fixes, or secondary errors introduced in the act of fixing a prior error

One of the many problems with LOC-metrics is the fact that more than half of all software defects are found in requirements and design, and hence the LOC-metrics is not capable of either predicting or measuring their volume with acceptable accuracy.

Because the cost and effort for finding and fixing bugs is usually the largest identifiable software cost element, ignoring defects can throw off estimates, schedules, and costs by massive amounts.

Rule 5: Sizing Software Defect Potentials

Function points raised to the 1.25 power predict the approximate defect potential for new software projects.

A similar corollary rule can predict the defect potentials for enhancements. In this case, the rule applies to the size of the enhancement rather than to the base that is being updated: Function points raised to the 1.27 power predict the approximate defect potential for enhancement software projects, using the enhancement (and not the base system) as the basis for applying the rule. The higher power used in the enhancement rule is because of the latent defects lurking in the base product that will be encountered during the enhancement process. The same rule can be used for maintenance projects such as defect repairs, although these may be so small (less than one function point) that the rule becomes ineffective.

The function point metric has been very useful in clarifying the overall distribution of software defects. Indeed, almost the only literature on the volume of non-code defects uses function point metrics for expressing the results. The approximate U.S. average for software defects during development for new projects is shown in Table 6.10.

As can easily be seen from Table 6.10, there are more software defects found outside the code than within it. Data expressed using the older

TABLE 6.10 U.S. Averages for Software Defect Levels During Development

Defect origin	Defects per function point
Requirements	1.00
Design	1.25
Code	1.75
User documents	0.60
Bad fixes	0.40
Total	5.00

LOC-metrics almost never deals with non-coding defects, such as those in requirements, specifications, and user manuals.

There are two other important categories of defects where empirical evidence has not yet accumulated enough data to derive useful rules of thumb: (1) errors in databases, and (2) errors in test cases themselves.

Research on database errors is handicapped by the fact that there is no *data point* metric for normalizing database sizes or defect rates.

This same problem is true for test cases. However, assessments and interviews with commercial software companies indicate that errors in test cases are sometimes more plentiful than errors or bugs in the software itself!

A very preliminary rule of thumb for test-case errors would be about 1.8 defects per function point. This rule is derived by simply dividing the function point total of the application by the number of errors fixed in test cases.

Hopefully, more reliable data will become available for test-case errors in the future, and data quality will be able to be measured if a data point size metric can be created.

Sizing Software Defect-Removal Efficiency

The defect potential is the life-cycle total of errors that must be eliminated. The defect potential will be reduced by somewhere between 85 percent (approximate industry norms) and 99 percent (best-in-class results) prior to actual delivery of the software to clients. Thus, the number of delivered defects is only a small fraction of the overall defect potential.

Rule 6: Sizing Defect-Removal Efficiency for Test Steps

Each software test step will find and remove 30 percent of the bugs that are present.

An interesting set of rules of thumb can size the number of defects that might be found and can approximate the defect-removal efficiency of various reviews, inspections, and tests.

Testing has a surprisingly low efficiency in actually finding bugs. Most forms of testing will find less than one bug or defect out of every three that are present. The implication of this fact means that a series of between 6 and 12 consecutive defect-removal operations must be utilized to achieve very high-quality levels.

A typical set of test steps for medium-sized software projects in the 5,000–function point size range would include

1. Unit testing by developers
2. New function testing
3. Regression testing
4. Performance testing
5. System testing
6. Acceptance testing

If each of these six test stages removed 30 percent of the latent defects, the cumulative efficiency of the whole series would be about 88 percent. Assume you start with 1000 defects and remove 30 percent with each test step. The results would be 1000, 700, 490, 343, 240, 168, and 117 defects still latent at the end.

This is why major software producers normally use a multistage series of design reviews, code inspections, and various levels of testing from unit test through system test.

The rather low defect-removal efficiency level of most common forms of testing explains why the U.S. average for defect-removal efficiency is only about 85 percent unless formal design and code inspections are utilized.

Rule 7: Sizing Formal Inspection Defect-Removal Efficiency

Each formal design inspection will find and remove 65 percent of the bugs present.
Each formal code inspection will find and remove 60 percent of the bugs present.

In fact, the defect-removal efficiency of formal design and code inspections is so much higher than testing that it is useful to show a separate rule for these activities.

It is easy to see from Rule 7 why most of the software organizations that produce really high quality software with more than 95 percent cumulative defect-removal efficiency utilize formal pretest inspections at the design and code levels.

Incidentally, although formal inspections are not inexpensive, they have such a strong effect on defect removal that the increase in testing speed and reduction in testing and maintenance costs that they trigger give them one of the best returns on investment of any known software technology.

As an added benefit, if you do use formal design and code inspections before testing begins, you will find that the overall cumulative efficiency of the inspections and test steps together may well top 95 percent on

average, and achieve 99 percent defect removal efficiency levels from time to time.

Rule 8: Post-Release Defect-Repair Rates

Maintenance programmers can repair 10 bugs per staff month.

To complete the set of quality-related rules of thumb, Rule 8 deals with the rate at which bugs can be repaired after software applications are released.

Rule 8, or maintenance repair rates, has been around the software industry for more than 30 years and still seems to work. However, some of the state-of-the-art maintenance organizations that utilize complexity analyzers and code-restructuring tools and have sophisticated maintenance work benches available can double the rate and may even top 20 bugs per month.

Rules of Thumb for Schedules, Resources, and Costs

After the sizes of various deliverable items and potential defects have been quantified, the next stage in an estimate is to predict schedules, resources, costs, and other useful results.

Some of the rules of thumb for dealing with schedules, resources, and costs are compound, and require joining several individual rules together. However, these combinations are simple enough to be done with a pocket calculator, spreadsheet, or even in your head if you are good with mental calculations.

Assignment Scope and Production-Rate Logic

Software rules of thumb and both manual and automated estimating methods utilize algorithms and relationships based on two key concepts:

- The *assignment scope* (A scope) is the amount of work for which one person will be responsible on a software project.
- The *production rate* (P rate) is the amount of work that one person can perform in a standard time period, such as a work hour, work week, work month, or work year.

Both assignment scopes and production rates can be expressed using any convenient metric, such as function points, source code statements, or “natural” deliverables, such as words or pages.

For example, assume that you are responsible for estimating an application of 50,000 source code statements in size using the C programming language. If an average assignment for programmers in your organization is 10,000 source code statements, then this project will require five programmers.

If an average programmer in your organization can code at a rate of 2000 C statements each month, then the project will require 25 months of effort.

By combining the results of the assignment-scope and production-rate rules, you can derive useful estimates with a pocket calculator.

Let us look at the kinds of information that can be derived using A-scope and P-rate logic. We will also include an average monthly salary rate in order to derive a complete cost estimate.

Monthly pay	\$6,000
Project size	50,000 C statements
A scope	10,000 C statements
P rate	2,000 C statements per month
Staff	5 programmers (50,000 divided by the A scope of 10,000)
Effort	25 months (50,000 divided by the P rate of 2,000)
Schedule	5 months (25 months of effort divided by 5 programmers)
Cost	\$150,000 (25 months of effort at \$6000 per month)

This logic is not foolproof, but is often useful for creating quick and rough estimates. However, to use assignment scopes and production rates well, you need to know the averages and ranges of these factors within your own organization. This is one of the reasons why software measurements and software cost estimating are so closely aligned. Measurements supply the raw materials needed to construct accurate cost estimates.

Indeed, companies that measure software projects well can create local rules of thumb based on their own data that will be much more useful than generic rules of thumb based on overall industry experience.

Estimating Software Schedules

Schedule estimation is usually the highest priority topic for clients, project managers, and software executives. Rule 9 calculates the approximate interval from the start of requirements until the first delivery to a client.

<p>Rule 9: Estimating Software Schedules</p> <p>Function points raised to the 0.4 power predict the approximate development schedule in calendar months.</p>

TABLE 6.11 Software Schedules in Calendar Months from Start of Requirements to Delivery
(Assumes 1000 function points from requirements to delivery)

Power	Schedule in calendar months	Projects within range
0.32	9.12	Agile projects
0.33	9.77	Scrum, Crystal, etc.
0.34	10.47	
0.35	11.22	
0.36	12.02	O-O software
0.37	12.88	Client/server software
0.38	13.80	Outsourced software
0.39	14.79	MIS software
0.40	15.85	Commercial software
0.41	16.98	Systems and embedded software
0.42	18.20	Civilian government software
0.43	19.50	Military software
0.44	20.89	
0.45	22.39	

Note that Rule 9 is a generic rule that would need adjustment between civilian and military projects. Because military software usually takes more time, raising function point totals to about the 0.45 power gives a better result.

Among our clients, the range of observed schedules in calendar months varies from a low of about 0.32 to a high of more than 0.45. In general, smaller, simpler projects would match the lower power levels, while larger, more complex projects would match the higher power levels. Also, standards that add complexity and extra paperwork to software development, such DoD 2167 or ISO 9001, tend to push the schedule power level above the 0.4 average value.

Table 6.11 illustrates the kinds of projects whose schedules are typically found at various power levels, assuming a project of 1000 function points in size.

The use of function points for schedule estimation is one of the more useful by-products of function points developed in recent years. As with all rules of thumb, the results are only approximate and should not be used for serious business purposes such as contracts. However, as a sanity check these rules of thumb are fairly useful. Readers are urged to explore historical data within their own organizations and to develop schedule power tables based on their own local results, rather than on industry averages.

Estimating Software Staffing Levels

The next rule of thumb is concerned with how many personnel will be needed to build the application. Rule 10 is based on the concept of

assignment scope or the amount of work for which one person will normally be responsible.

Rule 10: Estimating Software Development Staffing Levels

Function points divided by 150 predict the approximate number of personnel required for the application.

Rule 10 includes software developers, quality assurance, testers, technical writers, database administrators, and project managers.

The rule of one technical staff member per 150 function points obviously varies widely based on the skill and experience of the team and the size and complexity of the application. This rule simply provides an approximate starting point for more detailed staffing analysis. If you are only interested in programming and not in analysis, design, technical writing, etc. then use a value of 500 function points. Thus if you are concerned with building a 10,000–function point application (1,000,000 LOC), then some 20 programmers will be needed. However, if you use pair programming, the assignment scope will be cut in half.

Rule 11: Estimating Software Maintenance Staffing Levels

Function points divided by 750 predict the approximate number of maintenance personnel required to keep the application updated.

A corollary rule can estimate the number of personnel required to maintain the project during the maintenance period.

The implication of Rule 11 is that one person can perform minor updates and keep about 750 function points of software operational. (Another interesting maintenance rule of thumb is: Raising the function point total to the 0.25 power will yield the approximate number of years that the application will stay in use.)

Among our clients, the best-in-class organizations are achieving ratios of up to 3500 function points per staff member during maintenance. These larger values usually indicate a well-formed geriatric program, including the use of complexity analysis tools, code-restructuring tools, reengineering and reverse-engineering tools, and full configuration control and defect tracking of aging legacy applications.

Rule 11 will vary with programming languages. For low-level languages such as Assembler, an assignment scope of about 250 function points would be used. For higher-level languages such as JAVA, 750

would be used. For really experienced maintenance personnel, the assignment scope will be about 1500 function points.

Rule 12: Estimating Software Development Effort

Multiply software development schedules by the number of personnel to predict the approximate number of staff months of effort.

Estimating Software Development Effort

The last development rule of thumb in this chapter is a hybrid rule that is based on the combination of Rules 9 and 10.

Since this is a hybrid rule, an example can clarify how it operates. Assume you are concerned with a project of 1000 function points in size.

- Using Rule 9, or raising 1000 function points to the 0.4 power, indicates a schedule of about 16 calendar months.
- Using Rule 10, or dividing 1000 function points by 150, indicates a staff of about 6.6 full-time personnel.
- Multiplying 16 calendar months by 6.6 personnel indicates a total of about 106 staff months to build this particular project.

(Incidentally, another common but rough rule of thumb defines a *staff month* as consisting of 22 working days with 6 productive hours each day, or 132 work hours per month.)

Hybrid rules are more complex than single rules, but at least this hybrid rule includes the two critical factors of staffing and schedules.

Rules of Thumb Using Activity-Based Cost Analysis

There are manual software cost-estimating methodologies that are more accurate than simple rules of thumb, but they require work sheets and more extensive calculations in order to be used. Of these more complex manual methods, the approaches that lead to activity-based cost estimates are often the most useful and also the most accurate in the hands of experienced project managers.

It has been known for many years that military software projects have much lower productivity rates than civilian software projects. It has also been known for many years that large systems usually have much lower productivity rates than small projects. A basic question that our measurement methods should be able to answer is, “Why do these productivity differences occur?”

It is the ability to explore some of the fundamental reasons why productivity rates vary that gives activity-based measurements a strong advantage that can lead to significant process improvements.

Data measured only to the project level is inadequate for any kind of in-depth economic analysis or process comparison. This is also true for data based on rudimentary phase structures, such as “requirements, design, coding, integration, and testing.”

However, when activity-based cost analysis is used, it becomes fairly easy to answer questions such as the one posed at the beginning of this section. For example, military software projects have lower productivity rates than civilian software projects because military software projects perform many more activities than do civilian projects of the same size.

SPR has analyzed many software development methodologies from around the world, and has constructed a generic checklist of 25 activities that occur with high frequency. This list of activities is used for baseline cost collection, for schedule measurement, and as the basis for exploring the effectiveness of various kinds of tools and practices.

One of the interesting by-products of exploring software projects down to the activity level is the set of *patterns* that are often associated with various sizes and kinds of software projects.

To illustrate some of the variations at the activity level, Table 6.12 gives examples of activity pattern differences noted during SPR assessment and baseline studies for various classes of software based on the checklist of 25 activities that SPR utilizes for data collection.

Table 6.12 illustrates the patterns noted for six general kinds of software: (1) end-user software, (2) management information systems (MIS), (3) outsource projects, (4) commercial software vendors, (5) systems software, and (6) military software.

It is very interesting to realize that much of the observed difference in productivity rates among various industries and kinds of software is due to the fact that they do not all build software using the same sets or patterns of activities. It takes much more work to build U.S. military software than any other kind of software in the world. This is because Department of Defense standards mandate activities such as *independent verification and validation* and *independent testing* that civilian software projects almost never use.

Over and above the average values shown, there can be other significant variations. For example, small client/server projects may only perform 8 of the 16 activities listed under the MIS domain, which simultaneously explains why client/server productivity can be high and client/server quality low, since many of the quality-related activities are conspicuously absent in the client/server domain.

From activity-based analysis such as this, it becomes easy to understand a number of otherwise ambiguous topics. For example, it can

TABLE 6.12 Typical Activity Patterns for Six Software Domains

Activities performed	End user	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 Acceptance 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	18	21	20	23	25

easily be seen why U.S. military software projects are more expensive than any other kind of software, since they perform more activities.

Variation in activity patterns is not the only factor that causes productivity differences, of course. The experience and skill of the team, the available tools, programming languages, methods, processes, and a host of other factors are also important. However, the impact of these

other factors cannot be properly understood unless the cost and effort data for the project is accurate and is also granular down to the level of the activities performed.

If your software cost-tracking system “leaks” large but unknown amounts of effort, and if you collect data only to the level of complete projects, you will not have enough information to perform the kind of multiple regression analysis necessary to evaluate the impact of the other factors that influence software productivity and cost results.

To illustrate the approximate amount of effort and costs required for specific activities, Table 6.13 shows the average amount of work hours per function point for each of the 25 activities in the standard SPR chart of accounts for software projects, although there are, of course, large variations for each activity.

The information shown in Table 6.13 illustrates the basic concept of activity-based costing. It is not a substitute for one of the commercial software cost-estimating tools that support activity-based costs in a much more sophisticated way, such as allowing each activity to have its own unique cost structure and varying the nominal hours expended based on experience, methods, tools, and so forth.

In this table, the “Staff FP Assignment” column represents the average number of function points assigned to one staff member.

The “Monthly FP Production” column represents the typical amount of a particular activity that one person can accomplish in one month.

The “Work Hours per FP” column represents the number of hours for each function point, assuming in this case that 132 hours are worked each month. Obviously, this column will change if the number of available monthly hours changes.

To use the data in Table 6.13, you need to know at least the approximate function point size of the application in question. Then, select the set of activities that you believe will be performed for the application. After that, you can add up the work hours per function point for each activity.

You can do the same kind of selection and aggregation with costs, of course, but you should replace the default compensation level of \$5000 per staff month and the default burden or overhead rate of 100 percent with the appropriate values from your own company or organization.

Note that Table 6.13 uses a generic chart of accounts that includes both civilian and military activities. For example, activity 12 is *independent verification and validation*, which is required by many military contracts but is seldom or never used for civilian projects.

Once activity-based costing is started, it can be extended to include many other activities in a similar fashion. For example, the set of activities shown here is common for development projects. If you are concerned with the maintenance of aging legacy applications, with porting software

TABLE 6.13 Example of Activity-Based Chart of Accounts
(Assumes new development projects)

Assumptions					
Work hours per month		132			
Unpaid overtime per month		0			
Average monthly salary		\$5,000			
Burden rate		100%			
Burdened monthly rate		\$10,000			
Burdened hourly rate		\$76			
Activities	Staff FP assignment	Monthly FP production	Work hours per FP	Burdened cost per FP	Staffing per 1000 FP
01 Requirements	250.00	175.00	0.75	\$57.14	4.00
02 Prototyping	350.00	150.00	0.88	66.67	2.86
03 Architecture	2000.00	300.00	0.44	33.33	0.50
04 Project plans	1000.00	500.00	0.26	20.00	1.00
05 Initial design	250.00	175.00	0.75	57.14	4.00
06 Detail design	250.00	150.00	0.88	66.67	4.00
07 Design reviews	200.00	225.00	0.59	44.44	5.00
08 Coding	150.00	50.00	2.64	200.00	6.67
09 Reuse acquisition	250.00	600.00	0.22	16.67	4.00
10 Package purchase	5000.00	400.00	0.33	25.00	0.20
11 Code inspections	150.00	150.00	0.88	66.67	6.67
12 Independent verification and validation	2000.00	125.00	1.06	80.00	0.50
13 Configuration management	1000.00	1750.00	0.08	5.71	1.00
14 Integration	2000.00	250.00	0.53	40.00	0.50
15 User documentation	750.00	70.00	1.89	142.86	1.33
16 Unit testing	150.00	150.00	0.88	66.67	6.67
17 Function testing	350.00	150.00	0.88	66.67	2.86
18 Integration testing	700.00	175.00	0.75	57.14	1.43
19 System testing	2500.00	200.00	0.66	50.00	0.40
20 Field (beta) testing	1500.00	225.00	0.59	44.44	0.67
21 Acceptance testing	750.00	350.00	0.38	28.57	1.33
22 Independent testing	2500.00	200.00	0.66	50.00	0.40
23 Quality assurance	2000.00	150.00	0.88	66.67	0.50
24 Installation and training	5000.00	350.00	0.38	28.57	0.20
25 Project management	750.00	100.00	1.32	100.00	1.33
Cumulative results	203.39	6.75	19.55	1481.03	4.92

from one platform to another, or with bringing out a new release of a commercial software package, then you will need to deal with other activities outside of those shown in the table.

Table 6.14 illustrates a similar chart of accounts as might be used for an enhancement project in the civilian software domain. While some

of the activities are identical between new projects and enhancements, enhancements often have some unique activities, such as analyzing the base system, restructuring the base system (if necessary), regression testing, and several others. These *enhancement-only* activities are shown in boldface type in the table to set them off from common activities.

Neither table should be regarded as anything more than an example to illustrate the concept of activity-based cost estimating. What are actually

TABLE 6.14 Example of Activity-Based Chart of Accounts
(Assumes enhancement projects)

Assumptions					
Work hours per month		132			
Unpaid overtime per month		0			
Average monthly salary		\$5,000			
Burden rate		100%			
Burdened monthly rate		\$10,000			
Burdened hourly rate		\$76			
Activities	Staff FP assignment	Monthly FP production	Work hours per FP	Burdened cost per FP	Staffing per 1000 FP
01 Requirements	250.00	175.00	0.75	\$57.14	4.00
02 Base analysis	1000.00	300.00	0.44	33.33	1.00
03 Restructuring base	3000.00	1000.00	0.13	10.00	0.33
04 Project plans	1000.00	500.00	0.26	20.00	1.00
05 Initial design	300.00	200.00	0.66	50.00	3.33
06 Detail design	300.00	175.00	0.75	57.14	3.33
07 Design reviews	200.00	225.00	0.59	44.44	5.00
08 Coding	150.00	50.00	2.64	200.00	6.67
09 Reuse acquisition	250.00	600.00	0.22	16.67	4.00
10 New inspections	150.00	150.00	0.88	66.67	6.67
11 Base inspections	500.00	150.00	0.88	66.67	2.00
12 Configuration management	1000.00	1750.00	0.08	5.71	1.00
13 Integration	2000.00	250.00	0.53	40.00	0.50
14 User documentation	750.00	70.00	1.89	142.86	1.33
15 Unit testing	150.00	150.00	0.88	66.67	6.67
16 New function testing	350.00	150.00	0.88	66.67	2.86
17 Regression testing	1000.00	150.00	0.88	66.67	1.00
18 Integration testing	700.00	175.00	0.75	57.14	1.43
19 System testing	2500.00	200.00	0.66	50.00	0.40
20 Repackaging	2500.00	300.00	0.44	33.33	0.40
21 Field (beta) testing	3000.00	225.00	0.59	44.44	0.33
22 Acceptance testing	2500.00	200.00	0.66	50.00	0.40
23 Quality assurance	2000.00	150.00	0.88	66.67	0.50
24 Installation and training	5000.00	350.00	0.38	28.57	0.20
25 Project management	750.00	100.00	1.32	100.00	1.33
Cumulative results	218.18	6.94	19.02	1496.48	4.58

needed are similar tables that match your organization's charts of accounts and substitute your organization's data for generic industry data.

However, both tables do illustrate the levels of granularity that are needed to really come to grips with the economic picture of software development. Data collected only to the level of projects or a few phases is too coarse to really understand software economics with enough precision to plan significant process improvements.

Tables 6.13 and 6.14 are generic in nature and do not assume any particular methodology or formal process.

However, there are also specific activity patterns associated with object-oriented (OO) development, information engineering (IE), rapid application development (RAD), and a host of methodologies and software processes. The fundamental concept of activity-based costing can still be used, even though the activities and their patterns will vary, as will the specific values for the activities in question.

Summary and Conclusions

Simple rules of thumb are never very accurate, but continue to be very popular. The main sizing and estimating rules of thumb and the corollary rules presented here are all derived from the use of the function point metric. Although function point metrics are more versatile than the older lines-of-code-metrics, the fact remains that simple rules of thumb are not a substitute for formal estimating methods.

These rules of thumb and their corollaries have a high margin of error and are presented primarily to show examples of the kinds of new project management information that function point metrics have been able to create. At least another dozen or so rules of thumb also exist for other predicting phenomena, such as annual software enhancements, optimal enhancement sizes, software growth rates during maintenance, and many other interesting topics.

Software estimating using rules of thumb is not accurate enough for serious business purposes. Even so, rules of thumb continue to be the most widely used estimating mode for software projects. Hopefully, the limits and errors of these simplistic rules will provide a motive for readers to explore more accurate and powerful estimation methods, such as commercial estimating tools.

Manual estimating methods using work sheets that drop down to the level of activities, or even tasks, are more accurate than simple rules of thumb, but they require a lot more work to use. They also require a lot more work when assumptions change.

Replacing work sheets with automated spreadsheets eliminates some of the drudgery, but neither worksheets nor spreadsheets can deal with some of the dynamic estimating situations, such as creeping user

requirements or improvements in tools and processes during project development. This is why automated estimating tools usually outperform manual estimating methods.

Software measurement and estimation are becoming mainstream issues as software becomes a major cost element in corporations and government organizations. In order to use collected data for process improvements or industry benchmark comparisons, it is important to address and solve three problems:

- Leakage from cost-tracking systems must be minimized or eliminated.
- Accurate normalization metrics, such as function points, are needed for benchmarks to be economically valid.
- Cost and effort data needs to be collected down to the level of specific activities to understand the reasons for software cost and productivity-rate variances.

Activity-based software costing is not yet a common phenomenon in the software industry, but the need for this kind of precision is already becoming critical.

The literature on manual estimating methods is the largest of almost any project management topic, with a number of books offering algorithms, rules of thumb, or empirical results taken from historical data.

References

- Albrecht, A. J.: "Measuring Application Development Productivity," *Proceedings of the Joint IBM/SHARE/GUIDE Application Development Symposium*, October 1979, reprinted in Capers Jones, *Programming Productivity—Issues for the Eighties*, IEEE Computer Society Press, New York, 1981, pp. 34–43.
- 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.
- Brooks, Fred: *The Mythical Man-Month*, Addison-Wesley, Reading, Mass., 1974, rev. 1995.
- 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-Wesley, 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; 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.)
- Roetzheim, William H., and Reyna A. Beasley: *Best Practices in Software Cost and Schedule Estimation*, Prentice-Hall PTR, Upper Saddle River, N.J., 1998.
- Rubin, Howard: *Software Benchmark Studies for 1997*, Howard Rubin Associates, Pound Ridge, N.Y., 1997.
- 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.