SOFTWARE ESTIMATING RULES OF THUMB

Version 1 - April 6, 1997
Version 2 – June 13, 2003
Version 3 – March 20, 2007

Abstract

Accurate software estimating is too difficult for simple rules of thumb. Yet in spite of the availability of more than 50 commercial software estimating tools, simple rules of thumb remain the most common approach. Rules based on the function point metric are now replacing the older LOC rules. This article assumes IFPUG counting rules version 4.1.

Capers Jones, Chief Scientist Emeritus
Software Productivity Research, Inc.
Email cjonesiii@CS.com

INTRODUCTION

For many years manual estimating methods were based on the "lines of code" (LOC) metric 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.

Tables 1 and 2 illustrate samples of the LOC based rules of thumb for procedural languages in two forms: Table 1 uses "months" as the unit for work, while table 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.

Table 1: Rules of Thumb Based on LOC Metrics for Procedural Languages (Assumes 1 work month = 132 work hours)

Size of Program in LOC	Coding LOC per Month	Coding Effort (Months)	Testing Effort Percent	Noncode Effort Percent	Total Effort (Months)	Net LOC per Month
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

As can be seen, the "monthly" form of this table is not very convenient for the smaller end of the spectrum but the "hourly" form is inconvenient at the large end.

Table 2: Rules of Thumb Based on LOC Metrics for Procedural Languages (Assumes 1 work month = 132 work hours)

Size of Program in LOC	Coding LOC per Hour	Coding Effort (Hours)	Testing Effort Percent	Noncode Effort Percent	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

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, it is best to

replace the generic rate of "132" with an actual or specific rate derived from local conditions and work patterns.

The development of Visual Basic and its many competitors such as Realizer have 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. By the end of the century perhaps 30% of the new software applications will be developed using either object-oriented languages or visual languages (or both).

For large systems, programming itself is only the fourth most expensive activity. The three higher-cost activities can not really be measured or estimated effectively using the lines of code metric. Also, the fifth major cost element, project management, cannot easily be estimated or measured using the LOC metric either:

Table 3: Rank Order of Large System Software Cost Elements

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

The usefulness of a metric such as lines of code which can only measure and estimate one out of the five major software cost elements is a significant barrier to economic understanding.

The Development of Function Point Metrics

By the middle 1970's IBM's software community was topping 25,000 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 numbers, 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 in IBM and elsewhere used several languages concurrently, such as COBOL, RPG, and SQL as part of the same system.

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."

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

1. The types of inputs to the application
2. The types of outputs which leave the application
3. The types of inquiries which users can make
4. The types of logical files which the application maintains
5. The types of interfaces to other applications

In October of 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.

Now that the function point metric has been in use for almost 20 years on many thousands of software projects, a new family of simple rules of thumb has been derived. These new rules are based on function points, and encompass software sizing algorithms, schedule algorithms, quality algorithms, and other interesting topics.

This article contains a set of ten simple rules of thumb that cover various aspects of software development and maintenance. The rules assume the version 4.1 function point counting rules published by the International Function Point Users Group (IFPUG).

The International Function Point Users Group (IFPUG) is a non-profit organization which has become the largest software metrics association in history. Between IFPUG in the United States and other function point affiliates in about 20 countries, more than 2000 corporations and 15,000 individuals now comprise the function point community. Membership in function point user groups has been growing at more than 50% per year, while usage of lines of code has been declining for more than 10 years.

Users of other kinds of function points such as Mark II, COSMIC, web-object points, story points, engineering function points, etc. should seek out similar rules from the appropriate sponsoring organization. However, most of the function point variants have the interesting property of creating function point totals about 15% larger than IFPUG function points.

The following set of rules of thumb are known to have a high margin of error. They are being published in response to many requests for simple methods that can be used manually or with pocket calculators or spreadsheets. 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.

SIZING RULES OF THUMB

The function point metric has transformed sizing from a very difficult task into one that is now both easy to perform and comparatively accurate.

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."

For similar information on almost 500 programming languages refer to my book Applied Software Measurement (McGraw Hill 1996) or to our web site (http://www.spr.com).

Rule 1: Sizing Source Code Volumes

One function point = 320 statements for basic assembly language
One function point = 125 statements for the C programming language
One function point = 107 statements for the COBOL language
One function point = 71 statements for the ADA83 language
One function point = 15 statements for the SMALLTALK language

The overall range of non-commentary 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 with full class libraries 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.

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, the costs of producing paper documents costs far more than source code.

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

Rule 2: Sizing Software Plans, Specifications, and Manuals

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

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 "lines of code" (LOC) metric was that it tended to conceal both the volumes of paper deliverables and the high costs of software paperwork.

Sizing Creeping User Requirements

The function point metric is extremely useful in measuring the rate at which requirements creep.

Rule 3: Sizing Creeping User Requirements

Creeping user requirements will grow at an average rate of 2% per month from the design through coding phases.

Assume that you and your clients agree during the requirements to develop an application of exactly 100 function points. This rule of thumb implies that every month thereafter, the original requirements will grow by a rate of 2 function points. Since the design and coding phases of a 100 function point project are usually about 6 months, this rule would imply that about 12% new features would be added and the final total for the application would be 112 function points rather than the initial value of 100 function points.

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. Commercial software estimating tools can predict the number of test cases for more than a dozen discrete forms of testing. This simple rule of thumb encompasses the sum of all test cases:

Rule 4: Sizing Test Case Volumes

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

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.

Sizing Software Defect Potentials

The "defect potential" of an application is the sum of bugs or errors that will occur in five major deliverables: 1) requirements errors; 2) design errors; 3) coding errors; 4) user documentation errors; 5) bad fixes, or secondary errors introduced in the act of fixing a prior error.

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

Because the costs 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 predicts 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 the base that is being updated: Function points raised to the 1.27 power predicts the approximate defect potential for enhancement software projects.

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.

Incidentally, if you are doing complex client-server applications the 1.27 power actually matches the defect potentials somewhat better than the 1.25 power. Client-server applications are often very buggy, and the higher power indicates that fact.

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% (approximate industry norms) and 99% (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

Each software review, inspection, or test step will find and remove 30% of the bugs that are present.

The implication of this rule means that a series of between six and 12 consecutive defect removal operations must be utilized to achieve very high quality levels. This is why major software producers normally use a multi-stage series of design reviews, code inspections, and various levels of testing from unit test through system test.

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.

Estimating Software Schedules

Rule 7 calculates the approximate interval from the start of requirements until the first delivery to a client:

Rule 7: Estimating Software Schedules

Function points raised to the 0.4 power predicts the approximate development schedule in calendar months.

Among our clients, the range of observed schedules in calendar months varies from a low of about 0.32 to a high or more than 0.45. Table 4 illustrates the kinds of projects whose schedules are typically found at various power levels, assuming a project of 1000 function points in size:

Table 4: Software Schedules in Calendar Months
(Assumes 1000 function points from requirements to delivery)

Power	Schedule in Calendar Months	Projects Within Range
0.32	9.12	
0.33	9.77	Agile
0.34	10.47	Extreme
0.35	11.22	Web
0.36	12.02	OO 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 software
0.42	18.20	
0.43	19.50	Military software
0.44	20.89	
0.45	22.39	

The use of function points for schedule estimation is one of the more useful byproducts of function points that has been developed in recent years.

Estimating Software Staffing Levels

Rule 8 is based on the concept of "assignment scope" or the amount of work for which one person will normally be responsible. Rule 8 includes software developers, quality assurance, testers, technical writers, data base administrators, and project managers.

Rule 8: Estimating Software Development Staffing Levels

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

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.

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

Rule 9: Estimating Software Maintenance Staffing Levels

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

The implication of rule 9 is that one person can perform minor updates and keep about 750 function points of software operational. (Anther 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 3,500 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.

Estimating Software Development Effort

The last rule of thumb in this article is a hybrid rule that is based on the combination of rule 7 and rule 8:

Rule 10: Estimating Software Development Effort

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

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 7, or raising 1000 function points to the 0.4 power, indicates a schedule of about 16 calendar months.
- Using rule 8, 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.

Summary and Conclusions

Simple rules of thumb are never very accurate, but continue to be very popular. The 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 points metrics are more versatile than the former "lines of code" metric the fact remains that simple rules of thumbs are not a substitute for formal estimating methods.

SUGGESTED READINGS

Dreger, Brian; Function Point Analysis; Prentice Hall, Englewood Cliffs, NJ; 1989; ISBN 0-13-332321-8; 195 pages.

This is an introductory college primer to function point analysis. It assumes no prior knowledge of function point metrics and is often used for training those new to function point analysis.

Jones, Capers; Applied Software Measurement; McGraw Hill, New York; 1996; ISBN 0-07-032826-9; 818 pages.

This book uses function point metrics to calculate U.S. national averages for software productivity and quality from a sample of some 6,700 projects derived from 600 organizations.

Jones, Capers; Estimating Software Costs, McGraw Hill, New York, 2007.

This new addition includes information on 22 methodologies that were not common when the first edition was published in 1998. These new methods include Agile development, Extreme programming, Scrum, and many others.

International Function Point Users Group; IT Measurement; Addison Wesley, Boston MA; 2002; ISBN 0-201-74158-X; 759 pages.

See also several web sites for current information on function points and software estimating: www.spr.com and www.ifpug.org.