4

Measuring the Size of Software

How big is it? Well . . . it depends on how you count.

Size is one of the most basic attributes of software. Some of the key questions people want to know about a project involve size. For example:

- How big is the software? How big is it compared to other projects?
- How much effort is it going to take to build the software?
- How does our quality compare to other projects?
- How productive are we compared to other projects?

Size is fundamental to all of these metrics. In order to answer the questions, we need to be able to measure size in a standardized way that allows us to compare ourselves against other projects and external benchmarks.

In this chapter, we examine size in both the physical and functional sense. Both views are important in effort prediction and in normalization of other important metrics such as productivity and quality.

4.1 PHYSICAL MEASUREMENTS OF SOFTWARE

The traditional measure for software size is the number of lines of code (LOC). It is a simple measure, but it easily encourages undesirable behavior when used

Software Measurement and Estimation, by Linda M. Laird and M. Carol Brennan

inappropriately. If productivity is measured based on LOC/programmer day, the more lines you write, whether or not they do something useful, the more productive you are. It also seems overly simplistic. Although "a rose is a rose is a rose," a line of code can be a blank, a comment, or a ";", or it can be calculating a missile trajectory.

So how should we measure size? The LOC measure is frequently rejected because it does not seem to adequately address the functionality, complexity, and technology involved and may cause the wrong organizational behavior. However, consider an analogy to buildings. A building has a size, typically in square feet. It is an extremely useful measurement. Two thousand square feet means something to us all. However, size is not the only way to describe a building. It could be a home or an office building. It could be an executive seaside retreat or a cabin in the woods. Size is a major cost factor in buildings, but it is just one factor. One would expect the cost per square foot of a self-storage building to be less than a state-of-the-art data center, with raised floors, dual power supplies, and optical networking. However, one also would expect that the construction cost per square foot of a data center built in Montana and one built in New Hampshire would be relatively similar. In software, it is the same. The LOC measurement, just like square feet, serves a purpose. It is *one* of the cost drivers and normalization factors. We can use it to normalize metrics and compare different projects. We just need to be judicious in how we use it.

LOC measures the physical length of the software itself. When well specified, it is a reliable measurement. However, with visual and nonprocedural languages, such as Visual Basic, it frequently misses the mark. Moreover, it is always lacking in representing the size of the functionality of a system. Consequently, we need some other size measurements as well.

4.1.1 Measuring Lines of Code

Counting source lines of code is simple and reliable. Tools are available on the Internet that count well. The issue with counting code is determining which rules to use so that comparisons are valid.

The Software Engineering Institute (SEI) of Carnegie-Mellon University has published a framework for counting source code [1]. Included is a code counting checklist that allows you to determine explicitly how you count the code, which allows both internal and external standardization and benchmarking of code counting. Figure 4.1 is an example of the first page of a filled-in checklist.

Different organizations and studies use different rules. The most popular is NKLOC (non-commented thousand LOC) and LLOC (logical lines of code).

Your organization should be consistent in how it counts LOC. You can use the SEI checklist to decide consciously how you will count code.

4.1.2 Language Productivity Factor

Consider the problem of counting all the occurrences of "amazing" in a file. How many lines of code would this take? We would guess that in Assembler, it would

Definition name: *Physical Source Lines of Code (basic definition)* Date: *8/7/92*
Originator: *SEI*

Measurement unit: Physical source lines ☑ / Logical source statements ☐

Statement type	Definition	Data array	Includes	Excludes
When a line or statement contains more than one type, classify it as the type with the highest precedence.				
1 Executable	Order of precedence -> 1		✓	
2 Nonexecutable				
3 Declarations	2		✓	
4 Compiler directives	3		✓	
5 Comments				
6 On their own lines	4			✓
7 On lines with source code	5			✓
8 Banners and nonblank spacers	6			✓
9 Blank (empty) comments	7			✓
10 Blank lines	8			✓
11				
12				

How produced	Definition	Data array	Includes	Excludes
1 Programmed			✓	
2 Generated with source code generators			✓	
3 Converted with automated translators			✓	
4 Copied or reused without change			✓	
5 Modified			✓	
6 Removed				✓
7				
8				

Origin	Definition	Data array	Includes	Excludes
1 New work: no prior existence			✓	
2 Prior work: taken or adapted from				
3 A previous version, build, or release			✓	
4 Commercial, off-the-shelf software (COTS), other than libraries			✓	
5 Government furnished software (GFS), other than reuse libraries			✓	
6 Another product			✓	
7 A vendor-supplied language support library (unmodified)				✓
8 A vendor-supplied operating system or utility (unmodified)				✓
9 A local or modified language support library or operating system			✓	
10 Other commercial library			✓	
11 A reuse library (software designed for reuse)			✓	
12 Other software component or library			✓	
13				

Figure 4.1. *Code counting checklist [1].*

take about 300; in FORTRAN, around 100; in C, probably 10 to 15. In Unix Shell, it takes only one line with a "grep" in it.

Studies have shown that a proficient programmer can program approximately the same number of debugged lines of code per day regardless of the language. That is, if you are proficient at both Assembler and Java, your productivity, measured as lines of code per day, is approximately the same in both languages.

Consequently, we have the concept of a "language gearing factor," which compares the expressiveness of languages and takes into account the differences in "productivity" in languages. The more productive a language is, the fewer lines of code you should need to write, and hence, the more productive you should be.

Early gearing factors[1] were published by T. Capers Jones in 1991 [2], some of which are shown in Table 4.1.

[1]The gearing factor tables were created initially to show the relationship between LOC and function points. Function points are discussed in detail later in this book.

TABLE 4.1 1991 Gearing Factors

Language	Gearing Factor	Language	Gearing Factor
Accesss	38	Cobol (ANSI 85)	91
Ada 83	71	FORTRAN 95	71
Ada 95	49	High Level Language	64
AI Shell	49	HTML 3.0	15
APL	32	Java	53
Assembly—Basic	320	Powerbuilder	16
Assembly—Macro	213	Report Generator	80
Basic—ANSI	64	Spreadsheet	6
Basic—Visual	32	Shell Scripts	107
C	128	Visual C++	34
C++	55		

This table says that, on average, a function that took 38 LOC to program in Access would take 91 LOC to program in Cobol (assuming both languages were appropriate). If you can select the language you use, you want to pick the one with the lowest gearing factor, since you should end up writing the fewest LOC. In other words, if you can use a spreadsheet, do it, rather than writing code.

The concept of gearing factors is powerful: it allows us to normalize across languages. Recognize though that these numbers are only initial guidelines or averages and that they will vary based on a number of factors such as problem domain and programmer skill.

Other companies such as QSM, The David Consulting Group, and SPR have later gearing factors based on their clients and benchmarking research. Table 4.2 shows results from the 2005 QSM report, based on 2597 QSM function point projects [3] and The David Consulting Group's data.

4.1.3 Counting Reused and Refactored Code

You need to indicate the extent of the reuse in order to validly compare measures such as productivity and defect density. Refactored[2] code offers some special challenges due to the predominance of changed and deleted code.

For reused code, NASA uses a classification scheme that has an ordinal scale with four points: reused verbatim, slightly modified (<25%), extensively modified (≥25% modified), and new [4]. This classification scheme can be simplified to reused (<25% changed) and new, since, typically, if you change more than 25%, the effort is the same as if all of the code were new. Informal industrial surveys show 25% count a verbatim module each time it is used, 20% only count it once, 5% never count it, and 50% never count code at all [2].

[2]In case you are unfamiliar with the term refactoring, it is a disciplined technique for restructuring an existing body of code by altering its internal structure without changing its external behavior. Refactoring is frequently used in object-oriented development, in which class and object evolution is a natural and preferred method of development. There are tools that refactor code for some of the more popular OO languages such as Java and C^{++} and automatically produce many code structure metrics.

TABLE 4.2 2005 Gearing Factors

Language	QSM SLOC/FP Data				David Consulting Group Data
	Average	Median	Low	High	
Access	35	38	15	47	—
Ada	154	—	104	205	—
Advantage	38	38	38	38	—
APS	86	83	20	184	—
ASP	69	62	32	127	—
Assembler	172	157	86	320	575 Basic/ 400 Macro
C	148	104	9	704	225
C++	60	53	29	178	80
C#	59	59	51	66	—
Clipper	38	39	27	70	60
Cobol	73	77	8	400	175
Cool:Gen/IEF	38	31	10	180	—
Culprit	51	—	—	—	—
DBase III	—	—	—	—	60
DBase IV	52	—	—	—	55
Easytrieve +	33	34	25	41	—
Excel	47	46	31	63	—
Focus	43	42	32	56	60
FORTRAN	—	—	—	—	210
FoxPro	32	35	25	35	—
HTML	43	42	35	53	
Ideal	66	52	34	203	—
IEF/Cool:Gen	38	31	10	180	—
Informix	42	31	24	57	—
J2EE	61	50	40	60	—
Java	60	59	14	97	80
JavaScript	56	54	44	65	50
JCL	60	48	21	115	400
JSP	59	—	—	—	—
Lotus Notes	21	22	15	25	—
Mantis	71	27	22	250	—
Mapper	118	81	16	245	—
Natural	60	52	22	141	100
Oracle	38	29	4	122	60
Oracle Dev 2K/FORMS	41/42	30	21/23	100	—
Pacbase	44	48	26	60	—
PeopleSoft	33	32	30	40	—
Perl	60	—	—	—	50
PL/1	59	58	22	92	126
PL/SQL	46	31	14	110	—
Powerbuilder	30	24	7	105	—
REXX	67	—	—	—	—
RPG II/III	61	49	24	155	120
Sabretalk	80	89	54	99	—
SAS	40	41	33	49	50

(*continued*)

Table 4.2 *Continued*

Language	QSM SLOC/FP Data				David Consulting Group Data
	Average	Median	Low	High	
Siebel Tools	13	13	5	20	—
Slogan	81	82	66	100	—
Smalltalk	35	32	17	55	—
SQL	39	35	15	143	—
VBScript	45	34	27	50	50
Visual Basic	50	42	14	276	—
VPF	96	95	92	101	—
Web Scripts	44	15	9	114	—

For counting reused code, you need to decide what makes the most sense in your environment. Our personal preference is to count reused code as new if greater than 25% has changed, and only count it once. Again, it is important to be consistent and to understand what behaviors you want to encourage and discourage.

4.1.4 Counting Nonprocedural Code Length

Until the early 1990s, most code was text. With visual languages, such as Visual Basic, the size of the text is irrelevant, as "selections" and "clicks" became the "code." For these types of language and for generated languages, the typical length measurements do not work well. Functional size measurements, discussed later in this chapter, are typically used instead.

For object-oriented code, text size is still used, although it may not be as meaningful as with procedural languages.

4.1.5 Measuring the Length of Specifications and Design

It is enticing to consider that the length of the specifications and/or design of a system might predict actual code size and hence effort. Many researchers have tried to find generic correlations without great success. Fenton and Pfleeger [5] suggest that the success of the correlation depends greatly on the organization. We agree that these simple techniques can be effective in a disciplined environment with a consistent style and level of detail in the documents. They do need to be tuned and calibrated for the specific environment and tested to understand their validity.

Two length measurements for specifications and design are number of pages and number of "shalls."

The simplest technique is to use the number of pages of specifications and look for a correlation with either LOC or effort. If you use LOC, this is called the *specification-to-code expansion ratio*. Similarly, you can measure the length of the design specification and determine the *design-to-code expansion ratio*.

For example, from previous projects, you know that your design-to-code expansion ratio is 300, that is, 300 NLOC for every page of design. Your current project has 12 pages of design. You estimate that you will write 3600 NLOC.

Many organizations that are ISO-9000 certified and/or CMM Level 3 or higher use "shalls" within their requirements documents for mandatory requirements. For example, "the system shall record the time and date of every update" is a mandatory requirement. These mandatory requirements can be traced to design, to code, and to test cases, allowing an organization to certify that they have indeed developed and tested every mandatory requirement. If you have a consistent style and level of detail in your requirements, you may find a strong correlation between the number of shalls and the overall effort or LOC in your project.

EXERCISE: Within your organization, you have found that the average number of staff months per "shall" in a requirements document is 1.05 with a standard deviation of 10%. You have a new project that has 100 "shalls" in the requirements. What do you estimate the number of staff months to be? This estimate is for your boss, and you need to be as accurate as possible.

Answer: First, verify that this project and team is similar to the other projects. Next, estimate the percentage of "shall" changes that you expect to receive as the requirements "evolve" (e.g., change after they are finalized). Assume it is 10. Then you would estimate the number of staff months to be $110 * 1.05 = 115.5$ with two standard deviations of 20%. (Use 2 SD rather than 1 SD for the increase in accuracy.) Twenty percent is $\sim$23 staff months. In other words, you expect the effort to be between 92 and 138 staff months $\sim$95% of the time. If you do not assume any requirements creep, then it is $100 * 1.05 = 105$ staff months $\pm$ 21 staff months.

Although there are no benchmarks nor standards to use in predicting code length from requirements or design documents, if you have a well-disciplined organization with a consistent style, you may want to try using the length of the documents or the number of mandatory requirements. You need to gather sufficient data to create a historical trend. If your organization is not well-disciplined or does not have a consistent style, these techniques will not work.

4.2 MEASURING FUNCTIONALITY

One issue with using LOC, or any physical size measurement, as a measure of productivity is that it has little to do with the problem being solved and more to do with the software development technology itself. Customers want solutions and care little for the language or technology used to create them.

Consider what strategy you might use to size a system, based on what it needs to do rather than how it does it internally. It is a little difficult, isn't it? One possible way would be to count the inputs, outputs, interfaces, and databases in a system. That is the function point (FP) approach, once you add inquiries as well.

Function Point Analysis (FPA) [6] was invented as an indirect measure for the functional size of a system. Although many other measures have evolved from function points, it is arguably still the most widely used Functional Size Measurement (FSM).

Alan Albrecht originally developed function points at IBM in 1979, after he determined that you could estimate the size of a system from its external transactions and databases. Function Point Analysis has grown to become a standardized measure of size, with an international standards group (International Function Point Users Group, IFPUG), certification procedures, and over 1400 members. The IFPUG's website, www.ifpug.org, is a comprehensive source for FP information, including a function point counting manual. In addition, there are many excellent texts [7] and training courses if you desire a deeper understanding or to become certified in Function Point Analysis.

The IFPUG version of Function Point Analysis is the predominant function point methodology within the United States, but there are many other extensions and evolutions. Feature Points is a simple extension of function points for scientific applications. MKII Function Points was developed in the 1980s in the United Kingdom by Charles Symons with the objective of improving accuracy and being more compatible with structured analysis and design concepts. It was used extensively throughout Europe and Asia. COSMIC Full Function Points is a new international standard, which is gaining popularity in Europe and Canada. In addition, there are other FSM methodologies such as the Lorenz–Kidd Method, Object points, WebMO [8], and Use Case Points [9]. We will examine a few of the function point counting methodologies in this chapter, and a few more of the other functional sizing methodologies in the estimation chapter, since they are used for effort estimation.

FSMs are also called "proxy points." With proxy points, you count "something else," that is, a proxy, as a measure of the "functional size" of a system. You usually can count proxies early in the development process—some of them are effective during the high-level specification phase. From the proxy points, you extrapolate to estimate either LOC or effort. Most of the methodologies start with a "size" based on the "proxy" measurements and adjust it based on "difficulty" factors, such as complexity and environment.

From a user viewpoint, proxy points (including function points) are a better measure of productivity than lines of code because they measure the external rather than the internal behavior of the system. Increases in productivity as measured by function points means that the user is getting more functionality for the effort applied. Increases in productivity as measured by LOC means the user is getting more code for the effort applied, whether or not it is useful code.

4.2.1 Function Points

4.2.1.1 Counting Function Points

In the IFPUG Function Point Analysis,

- the system's functionality is decomposed into components of inputs, outputs, external interface files (maintained by another system), internal data files (maintained by this system), and inquiries;

- each component's difficulty is rated as simple, average, or complex;
- a complexity rating is given to each component based on its type and difficulty, as shown in Table 4.3;
- the unadjusted function points (UFPs) are counted by summing all of the complexity ratings;
- a value adjustment factor (VAF) is calculated based on the complexity of the overall system; and
- the adjusted function point (AFP) count is calculated by multiplying VAF by the UFPs.

TABLE 4.3 Function Point Complexity Ratings

Component	Simple	Average	Complex
Inputs (I)	3	4	6
Outputs (O)	4	5	7
Data Files (F)	7	10	15
Interfaces (N)	5	7	10
Inquiries (Q)	3	4	6

The complexity ratings represent the relative implementation effort. For example, from the FPA viewpoint, an average interface to an external file is harder to implement than an average inquiry; hence, the weighting for the average interface is 7 versus 4 for the average inquiry. That is, the external interface file requires 1.75 times more effort than the inquiry.

The specific IFPUG rules such as how to determine whether a component is simple, average, or complex, and how to count graphical user interfaces (GUIs) can be found on the IFPUG website. Unfortunately, the inherent number of rules and details in the Function Point Analysis process can be daunting and may deter some from using FPA. David Herron, a leading FP consultant, recommends a getting-started strategy of rating everything as average complexity. You can then evolve into a more precise FP counting method with experience.

EXERCISE: You have a small program with four simple inputs, one data file (trivial), and two outputs, both of average complexity. How many unadjusted function points would this be?

Answer:

$$\text{Inputs} = 4 * 3 = 12$$
$$\text{Data File} = 1 * 7 = 7$$
$$\text{Ouputs} = 2 * 5 = 10$$
$$\text{Total UFPs} = 12 + 7 + 10 = 29$$

An additional adjustment is made based on the expected implementation difficulty of the system. Using the IFPUG standards, you calculate the VAF based on

14 General Systems Characteristics (GSCs), each of which is rated on a scale of 0 to 5, with 0 meaning no influence (or not present) and 5 meaning that characteristic has an extensive influence throughout the project. The 14 GSCs are described in Table 4.4. These characteristics are really "implementation difficulty" factors. Tremendous transaction rates (GSC #5) are more difficult to support than low ones. Automated backup and recovery (GSC #12) requires more implementation effort than manual.

The formula for adjusted function points is $\text{AFPs} = \text{UFPs} * (0.65 + 0.01 * \text{VAF})$.

TABLE 4.4 General System Characteristics [10]

General System Characteristic (GSC)	Brief Description
1. Data Communications	How many communication facilities are there to aid in the transfer or exchange of information with the application of the system?
2. Distributed Data/Processing	How are distributed data and processing functions handled?
3. Performance Objectives	Are response time and throughput performance critical?
4. Heavily Used Configuration	How heavily used is the current hardware platform where the application will be executed?
5. Transaction Rate	Is the transaction rate high?
6. Online Data Entry	What percentage of the information is entered online?
7. End-User Efficiency	Is the application designed for end-user efficiency?
8. Online Update	How many data files are updated online?
9. Complex Processing	Is the internal processing complex?
10. Reusability	Is the application designed and developed to be reusable?
11. Conversion/Installation Ease	Are automated conversion and installation included in the system?
12. Operational Ease	How automated are operations such as backup, startup, and recovery?
13. Multiple Site Use	Is the application specifically designed, developed, and supported for multiple sites with multiple organizations?
14. Facilitate Change	Is the application specifically designed, developed, and supported to facilitate change and ease of use by the user?

The VAF adjusts the function point count based on the "implementation difficulty" (i.e., the general system characteristics) of the system. Observe that for an "average" system, with all average values of 2.5 for the 14 GSCs, the $\text{VAF} = 14 * 2.5 = 35$, such that the $\text{AFP} = \text{UFP} * (0.65 + 0.35) = \text{UFP}$, that is, the AFP equals 100% of the UFP. The VAF can adjust the function point count by $\pm 35\%$ (if all of the GSCs are five or all are zero).

EXERCISE: You have a small project that has 25 UFPs. You rate the GSCs of your system as trivial in all categories except complex processing and data

communications, which have a high influence. All trivial factors are scored as 1. Complex processing and data communications both get scores of 5. How many AFPs do you have?

Answer:

$$\text{VAF} = 12 * 1 + 2 * 5 = 22$$
$$\text{AFPs} = \text{UFPs} * (0.65 + 0.01 * \text{VAF}) = 25 * (0.65 + 0.22) = 21.75$$

Software Productivity Research (SPR) has an alternative calculation for the VAF, which "achieves results that are very close to the IFPUG method, but it goes about it in quite a different manner" [11]. The SPR method uses only two factors—problem complexity and code complexity—to adjust the function point count. These are each rated on a scale of 1 to 5, as shown in Table 4.5. In this method, the VAF = 0.4 + 0.1 (Program Complexity Score + Data Complexity Score). Notice that if both factors are average, the VAF = 1 and the adjustment range is ±40% in this method versus ±35% in the IFPUG method. The SPR method can be calculated manually but is automated in some of the SPR estimation tools.

TABLE 4.5 SPR Function Point Complexity Adjustment Factors [11]

Program Complexity Rating
1. All simple algorithms and simple calculations
2. Majority of simple algorithms and simple calculations
3. Algorithms and calculations of average complexity
4. Some difficult or complex algorithms or calculations
5. Many difficult algorithms and complex calculations
Data Complexity Rating
1. Simple data with few elements and relationships
2. Numerous variables and constant data items, but simple relationships
3. Average complexity with multiple files, fields, and data relationships
4. Complex file structures and complex data relationships
5. Very complex file structures and very complex data relationships

EXERCISE: Assume you have a program with a UFP count of 35. You determine that the program has many difficult algorithms but simple data with few elements and relationships. What is the AFP count using the SPR FP complexity adjustment method?

Answer:

$$\text{AFPs} = \text{UFPs} * (0.4 + 0.1 * (\text{PC} + \text{DC})) = 35(0.4 + 0.1 * (5 + 1)) = 35 * 1 = 35$$

4.2.1.2 Function Point Example Assume you work for a company that currently makes home safety and security monitoring devices and controllers. Now your company wants to sell home safety *systems*.

You need to design and build the software part of the system. The available components are:

- Controller (with associated monitoring devices) for door and window alarms
- Controller (with associated monitoring devices) for motion detectors
- Controller (with associated monitoring devices) for panic buttons
- Controller (with associated monitoring devices) for fire detector
- Controller (with associated devices) for light activator and deactivator
- Controller/monitor for key device (to turn system on and off)
- Wireless dial-out device with controller

(a) Estimate the effort to build the system. Assume a productivity of 10 FPs per staff month, with a standard deviation of ± 1 staff month.

(b) Now you are told that your company can only afford 75% of the estimated staff months to build the system. What do you do now? Be specific. Answers such as work 20% unpaid overtime will not get much credit, although your boss may be impressed. *Hint*: Figure out multiple ways to reduce the AFPs, showing the FP change and the impact on the estimate.

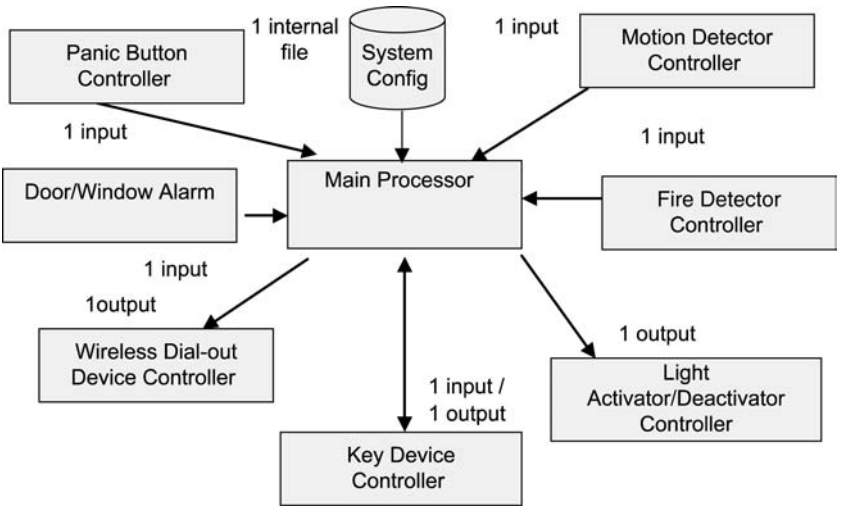

Figure 4.2. *Home security architecture.*

Solution for (*a*). You specify and design the system using block diagrams as shown in Figure 4.2.[3]

The component count is:

Simple Input = 5

Simple Output = 2

Simple Internal Database = 1

Medium Output (to Dial-out Controller) = 1

$$\text{UFPs} = 3 * 5 + 4 * 2 + 1 * 7 + 1 * 5 = 15 + 8 + 7 + 5 = 35$$

The values for the GSCs are[4]:

Data Communications	3
Distributed Functions	2
Performance	3
Heavily Used Configuration	1
Transaction Rate	0
Online Data Entry	1
End-User Efficiency	1
Online Update	1
Complex Processing	1
Resuability	3
Installation Ease	5
Operational ease	5
Multiple Sites	5
Facilitation of Change	2
VAF	33

$$\text{AFPs} = 35 * (0.65 + 0.33) = 35 * 0.98 = 34.3$$

The estimated number of staff months = $34.3/10 = 3.4$. Since the standard deviation is ± 1 staff month, 1 SD is $34.3/9 = 3.81$ and $43.3/11 = 3.12$. Therefore, you expect the effort on this project to be between $\sim$3.1 and $\sim$3.8 staff months $\sim$68% of the time.

Solution for (*b*).You need to reduce the expected staff months by at least 25%. There are many ways to reduce the staff months by reducing the number of AFPs.

The first step is to look at the UFPs. How can these be reduced? One answer is to reduce the functionality, but that might hurt the product. The next idea is to simplify the design. You can change the interface to the monitoring devices to be one generic monitoring input rather than 4 separate inputs. So now, instead of having 4 simple

[3]There are many possible designs. This is just one.

[4]The values for the GSCs are based on the assumption made about the system requirements and specifications. Other values are reasonable as well.

inputs, which total 12 AFPs, there is 1 average input for a total of 4 AFPs. The design change actually results in a better product, because it allows additional monitoring devices (such as pressure monitors) to be added easily. You can also reduce the VAF by changing some of the assumptions that you have made concerning the GSCs. For example, you could choose to use a faster processor, which would allow you to adjust the performance factor.

With only the design change, the AFPs are reduced from 34.3 to $27 * 0.98 = 26.46$, for a reduction of $\sim$7.8 AFPs, which results in a reduction of $\sim$23%. Also, if we select a faster processor, so that the GSC for performance is 1 instead of 3, then the VAF becomes 0.31 which reduces the AFPs to $27 * 0.96 = 25.92$, which is a reduction of $\sim$8.4 AFPs or nearly 25%.

This example illustrates an important point and the value of proxy point analysis. Budgets and staffing will always be inadequate. The question that must be answered is how to reduce the required effort. There are hosts of factors that generate effort. The best software engineers understand these factors and figure out how to simplify the solution, which reduces the proxy points, rather than working more overtime or cutting features.

4.2.1.3 Converting Function Points to Physical Size

The primary purpose of the gearing factors in Tables 4.1 and 4.2 is to convert function points to lines of code, based on the implementation language.

EXERCISE: In Section 4.2.1.2, we ended up with a FP count of $\sim$26. How many lines of code would you estimate if you were writing this system in Java? In Ada? Which would you prefer to use if you knew both well? Use the QSM data (Table 4.2) and the average SLOC/FP.

Answer: For Java, the average SLOC/FP is 60. For Ada, it is 154. There are 26 AFPs. Therefore, the estimated LOC is 1560 for Java and 4004 for Ada. Rounding, it is 1.6 KLOC for Java and 4.04 KLOC for Ada. We would choose to use Java and spend the extra time at the gym.

The only issue with converting FPs to LOC is which gearing factor to pick. You could choose the average, the maximum, the minimum, the median, The David Consulting Group (DSG) number, or the Jones number. We prefer the QSM or DSG data because they are newer and have more data points. After that, hopefully we would have some experience with past projects to guide us. If not, and unless we had more insight into the ease of using the language for this specific system, then we would start with the average, use a range to demonstrate the potential variance, and factor it into our risk management plan.

4.2.1.4 Converting Function Points to Effort

Once you count function points, how do you use them to estimate effort? The primary method is to convert the function points to KLOC and use the methods and tools discussed in later chapters to estimate schedule and effort.

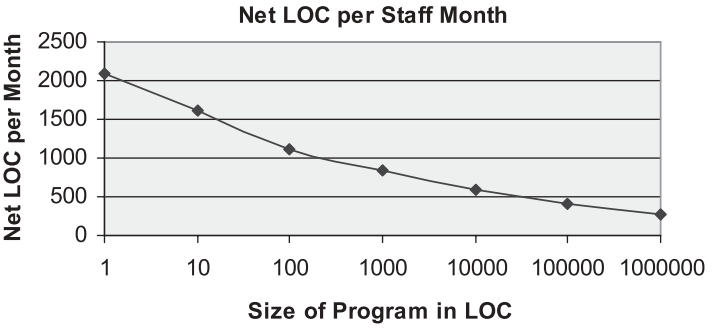

Figure 4.3. *Productivity engineering rule in LOC.*

Our second method, assuming you are using a procedural language, is to use the Jones engineering rules[5] for procedural languages [11]. He suggests using these engineering rules in situations where you are looking for only rough estimates or as a sanity check against other estimation methodologies. The essence of his data for productivity in LOC versus project size is captured in Figure 4.3. This data assumes 132 hours per month and includes all project efforts from requirements through testing.

The productivity decreases logarithmically as the size of the program increases. This decrease is due in large part to the increase in nonprogramming activity in larger projects.

EXERCISE: Estimate the effort for the system in Section 4.2.1.2.

Answer: For 1.6 KLOC, we would estimate the productivity to be $\sim$2 KLOC per month, based on Figure 4.3. Therefore, we would estimate the total effort to be $\sim$1 staff month. For 4 KLOC, the productivity is 1.8 KLOC per month, so we would estimate it to be $4/1.8$ or $\sim$2.2 staff months.

4.2.1.5 Other Function Point Engineering Rules Jones also has other engineering rules for function points. Some of the more interesting ones are:

- Schedule $= FP^{0.4}$, where schedule is in calendar months.
- Staff $= FP/150$, where staff includes developers, testers, quality assurance personnel, technical writers, project managers, and database administrators.
- Effort $=$ Staff $*$ Schedule.

For the schedule equation, the power factor ranges from $\sim$0.32 to $\sim$0.45 with 0.32 for the smaller, simpler projects, and 0.45 for the larger, complex military projects.

[5]We use the terminology of "engineering rules" throughout this text. It is similar in meaning to "rules of thumb" without the potential negative connotations.

The schedule covers the period from requirements through delivery. In his text, Jones remarks that the schedule engineering rule has been one of the most useful ones to come out of the FP data. He also urges readers to collect their own historical data and tune these rules to their own environment, which we thoroughly endorse.

EXERCISE: Estimate the effort for a system with 25.93 FPs, using the Jones engineering rules for function points.

Answer: First, we would round the number of function points to 26. The precision of this methodology certainly is worse than 0.1 FP. To estimate schedule, you need to decide on the power factor: 0.4 is the average for 1000 FP projects. We know that smaller projects have greater efficiencies, so we would use the smallest power factor, which is 0.32. Therefore,

$$\text{Schedule} = 26^{0.32} = 2.84 \text{ calendar months}$$
$$\text{Staff} = 26/150 = 0.173 \text{ (!!!)}$$
$$\text{Effort} = 0.173 * 2.84 = 0.5 \text{ staff month}$$

Hmm. . . The schedule seems to be a reasonable number, but the staffing is basically 1/5 of a person. This does not seem reasonable, or at least it does not match up that well with the LOC engineering rules.

Actually, this is an excellent example of using the engineering rules and estimation methodologies. The FP engineering rules result is 50% less than the Java estimate and ~75% less than the Ada estimate.[6] Our experience is that the staffing rule does not hold up well for very small projects, although the schedule rule might. The accuracies may seem low, but the point in estimation, especially using engineering rules, is to get in the right ballpark, not in the third row of the upper deck. These rules give a starting point and a sanity check and are extremely useful when you understand both their strengths and limitations.

4.2.1.6 Function Point Pros and Cons

Function points have been a tremendous addition to software estimation and counting methodologies. Function points are:

- technology independent,
- effective early (requirements phase) in the software life cycle,
- well-documented and specified,
- supported by standards and an international users group,
- backed by substantial data that supports the methodology,
- reasonably reliable and accurate,

[6]Note that the language chosen can have a huge impact on the FP productivity, which is not taken into account in the FP engineering rules. Consequently, since the range of gearing factors is large, it is reasonable to expect the FP engineering rules for effort to be even more imprecise.

- useful in negotiations with users and management, and
- insightful for understanding the sources of effort.

However, function points do have their issues, which include:

- They are semantically difficult. Many of the terms and factors are from the 1970s and 1980s and are difficult to understand in the context of today's systems. In this chapter, we have actually simplified the components for ease in understanding [12].
- They include many steps.
- Significant effort is required to become a certified function point counter.
- There are no tools that automatically count function points.
- There can be significant subjectivity in adjustment factors.

Nevertheless, function points are an important size metric and are still used successfully today. Albrecht's function point concepts are the foundation for all of the other FSMs.

4.2.2 Feature Points

Function points were originally designed for use with management information systems (MISs). The perception was that FPs were not optimal for heavily algorithmic systems, such as military, communications, and systems software. Feature points were developed by SPR [13] as an extension to function points to give additional weight to algorithmic complexity. SPR also reduced the relative weight of the average interface from 10 to 7. You can use the information in Table 4.6 to calculate the number of feature points. You use the VAF to convert UFPs to AFPs.

TABLE 4.6 Feature Point Component Weights

Component	Empirical Weight
Algorithms	3
Inputs	4
Outputs	5
Inquiries	4
Data Files	7
Interfaces	7

EXERCISE: You have a program that converts N, the number of days past A.D. 0, into the calendar date D. For example, if N = 10, then D would be January 10, 0001. How many total unadjusted feature points are there?

Answer:

Algorithms = 1
Inputs = 1
Outputs = 1
Inquiries = 0
Data files = 0
Interface Files = 0

Therefore, UFPs = 3 + 4 + 5 = 12

Feature points never reached widespread acceptance. They are interesting in that they show an evolution of FPs to better match another problem domain space.

4.3 SUMMARY

Size is one of the fundamental measures of software. We use it to normalize effort, quality, and cost. Although lines of code may have a poor reputation as a measure of size, since it does not indicate the complexity, functionality, or technology, it is a fundamental and useful metric, just as square feet is a fundamental and useful metric for buildings. It is important to have specific counting rules for lines of code—what should be included and what should not—so that you are consistent across projects and so that you understand how your counting compares with other projects.

We also need to measure the functionality, not just physical length, of software. Customers care about features and functionality, not how many lines of code it took to write it. Function Point Analysis was the first significant functional size measurement (FSM) and, although not perfect, a huge step forward for software estimation. It is the foundation for later FSM metrics.

The important concept of language gearing factors comes from function points. Gearing factors indicate the comparative number, on average, of statements required within a language to implement a function point. They allow us to both predict the number of lines of code required from a function point count and compare projects normalized by functionality rather than lines of code.

For both function points and lines of code, there are many engineering rules and algorithms to predict effort and schedule. These rules are extremely useful, especially for quick, rough estimates.

PROBLEMS

4.1 You have a system that has 3 inputs, 1 output, and 1 database file. All are of average complexity. The Technical Complexity Factors are all 2. You are writing this in Java, using The David Consulting Group's gearing factors.

(a) How many UFPs are there?

(b) What is the AFP count?

(c) What is the expected LOC?

4.2 You are the development manager and your team comes up with an estimate of 6 people for 9 months (e.g., 54 staff months). You know that the job needs to be done in 36 months, and you cannot add more people. (You also think that more people may actually slow you down anyway.) You find out that your team estimated by taking FPs and converting them into LOC, and then using Jones' productivity guideline.

(a) How many lines of code did they initially estimate?

(b) You know they intend to write the project in Java. How many function points did they calculate?

(c) You know you can reduce the functionality to cut the schedule, by working with the customer. How many function points do you need to reduce to have the estimate be 6 months instead of 9?

4.3 If you could program a system in Java or C equally well, which language would you expect would lead to less effort? Why?

PROJECT

The *Theater Tickets Project* is used throughout this book. The first step is to create a high-level design. In later chapters, you will continue to use it.

4.1 Theater Tickets Reservation System. This system reserves and allows you to purchase theater tickets. It has a database for tickets, a system administration interface, a user interface, and an interface to an online credit card validation system. The credit card validation system is a different system, which you do not need to build.

(a) Create a high-level design for this system (preferably, a block diagram).

(b) Without using any of the estimation tools you have learned so far, guess at the LOC and the number of staff months of effort it would take to build the system. Java is the programming language.

(c) Now count the number of function points.

(d) How many LOC do you estimate?

(e) Using Jones' table, what is the estimated effort?

(f) Compare your estimates. How do they differ?

(g) Now your boss listens nicely to your estimate and tells you it is 20% too large. How do you cut the effort by 20%? *Hint*: Change your design/plan so that there really is 20% less effort rather than by working overtime or just saying a component is simpler.

REFERENCES

[1] R. Park. "Software size measurement: a framework for counting source statements," CMU/SEI-92-TR-020. Available from http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr20.92.pdf. Accessed Jan. 20, 2005.

[2] T. C. Jones. *Applied Software Measurement*, McGraw-Hill, New York, 1991.

[3] QSM website, QSM Function Point Languages Table, Version 2.0 July 2002, from http://www.qsm.com/FPGearing.html#MoreInfo. Accessed Feb. 7, 2005.

[4] Software Productivity Consortium, *Software Measurement Guidebook*, International Thomson Computer Press, London, 1995.

[5] N. Fenton and S. Pfleeger, *Software Metrics*, PWS Publishing Company, Boston, 1997.

[6] A. J. Albrecht. "Measuring application development productivity," in *Proceeding IBM Applications Development Symposium.* GUIDE Int and Share Inc., IBM Corp., Monterey, CA, Oct. 14–17, 1979, p. 83.

[7] D. Gamus and D. Herron, *Function Point Analysis: Measurement Practices for Successful Software Projects* (Addison-Wesley Information Technology Series), Addison-Wesley, 2000.

[8] D. Reifer, "Estimating web development costs: there are differences," *CrossTalk*, June 2002.

[9] G. Karner, *Metrics for Objectory.* Diploma thesis, University of Linköping, Sweden. No. LiTH-IDA-Ex-9344:21. December 1993.

[10] D. Longstreet, "Fundamentals of function point analysis," www.softwaremetrics.com. Accessed Jan. 31, 2005.

[11] T. C. Jones, *Estimating Software Costs*, McGraw-Hill, New York, 1998.

[12] L. Fischman, "Evolving function points," www.stsc.hill.af.mil/crosstalk/2001/02/fischman.html. Accessed Feb. 14, 2005.

[13] "Feature points," http://www.spr.com/products/feature.shtm. Accessed Feb. 4, 2005.