3

Measuring Software Size

To make a software development plan, first produce a conceptual design and then estimate the size of the product to be built to that design. With the size estimate, you can then estimate how long it will take to develop the product. Before you can estimate a program's size, however, you must have a way to measure size. This chapter describes the principal ways to measure the sizes of various products and how to use these measures for project planning and quality management. Later chapters show how to estimate program size and development time. They also describe how to produce a schedule and how to measure, track, and report development status. The planning process, however, starts with size estimates, which require size measures.

3.1 Size Measures

The first question to address in selecting a software size measure is whether that measure will be useful for estimating development time. For a size measure to be useful, it must be precise, specific, and automatically countable. For example, although *number of pages* might seem to be a very precise and simple measure, even it must be properly specified. Are these single-spaced or double-spaced pages and are they in single or multiple columns? How will you count empty or partially

filled pages? And what size font should you use? These are the kinds of questions you must answer for every size measure you plan to use.

Using Size Measures in Planning

Not surprisingly, more resources are required to develop large programs than small ones. If the size of a program is approximately proportional to the time required to develop it, the size and time data are said to be correlated. **Correlation** is the degree to which two sets of data are related. The correlation value, *r*, varies from –1.0 to 1.0. When *r* is close to one, then the two sets of data, say *x* and *y*, are highly correlated. That means that increases in the value of *x* imply proportionate increases in the value of *y*. When *r* is close to –1.0, increases in *x* would imply proportionate decreases in *y*. For the PSP, we are principally interested in positive correlations. To be useful for estimating and planning purposes, the value of *r* should be greater than 0.7. Although many PSP tools calculate the correlation, the mathematical formula for the calculation is shown in Box 3.1.

BOX 3.1 CALCULATING THE CORRELATION COEFFICIENT

1. To determine if two sets of data, *x* and *y*, are sufficiently related for planning purposes, calculate their correlation coefficient and significance.
2. Start with *n* pairs of numbers x_i and y_i where *x* is the set of estimates and *y* is the actual data for those estimates.
3. The correlation *r* of the two sets of data is given by the following equation:

$$r(x,y) = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{\left[n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right]\left[n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right]}}$$

4. The values of *r* vary from –1.0 to +1.0. If |*r*| >= 0.7, the relationship is considered good enough for estimation purposes.
5. The closer |*r*| is to 1.0, the better the predictive value of the relationship. However, because small values of *n* can result in high correlations, the significance of the correlation should also be considered. The significance calculation is shown in Box 3.2.

The degree to which a correlation measure is meaningful is also important. It is indicated by what is called **significance**. The significance measures the probability that this relationship could have occurred by chance. Thus, a significance of 0.25 indicates that one-quarter of the time, this result could have occurred as the result of random fluctuations in the data. Conversely, a significance of 0.005 indicates that this result would likely occur at random only about once in 200 times. That is a very high significance. A significance of 0.05 is generally considered good; one of 0.20 is considered poor. The significance calculation is described in Box 3.2. By finding the correlation and significance for the size and time data for a number of programs, you can determine whether that size measure would be a useful predictor of development time.

For database work, for example, a count of database elements is a potentially useful size measure. Figure 3.1 shows how the count of database elements (fields, GUI controls, etc.) compares with the development time for several database development jobs. In this case, the correlation value is $r = 0.84$ with a significance of 0.01. Because r is greater than 0.7, this relationship is good enough for estimating purposes. In addition, with a 0.01 significance, the chance that this relationship occurred by chance is less than 1%.

BOX 3.2 CALCULATING SIGNIFICANCE

1. The significance value indicates whether the correlation of two sets of data, x and y, is sufficiently strong that the data can be used for estimating purposes or the relationship likely occurred by chance.
2. Start with n pairs of numbers x_i and y_i where x is the set of estimates and y is the actual measured values for the estimated items.
3. If the correlation $|r|$ of these data is greater than 0.7, calculate its significance.
4. Calculate the t value for this relationship as follows:

$$t = \frac{|r(x, y)|\sqrt{n-2}}{\sqrt{1-r(x, y)^2}}$$

5. Since r can be plus or minus, the absolute value, $|r(x, y)|$, is used.
6. Look up the value of the t-distribution in Table 3.1. Use the 95% column and, for $n = 5$, use 3 degrees of freedom ($d = 3$).
7. A t value of 3.182 or greater indicates a significance of less than (better than) 0.05, or less than a 5% chance of having occurred by coincidence. Values less than 0.05 are considered adequate.

TABLE 3.1 VALUES OF THE *t*-DISTRIBUTION FOR VARIOUS AMOUNTS OF DATA

Degrees of Freedom *d*	70% Prediction Interval	90% Predication Interval	95% Prediction Interval
1	1.963	6.314	12.706
2	1.386	2.920	4.303
3	1.250	2.353	3.182
4	1.190	2.132	2.776
5	1.156	2.015	2.571
6	1.134	1.943	2.447
7	1.119	1.895	2.385
8	1.108	1.860	2.306
9	1.100	1.833	2.262
10	1.093	1.812	2.228
15	1.074	1.753	2.131
20	1.064	1.725	2.086
30	1.055	1.697	2.042
∞	1.036	1.645	1.960

Figure 3.2 shows a counterexample for which size and time are not highly correlated. These data are for the time it took me to write 25 chapters of a book. Although the times were relatively balanced around the average of 27.5 hours, they varied from 8.8 to 42 hours and the correlation was a very low 0.26. With an *r* of less than 0.7, these page counts were not very useful as a size measure for estimating the writing time for these chapters. You would think that writing time would be closely related to the number of pages written, but the content of these chapters was so different that page count did not accurately predict writing time. However, because page count is generally a useful measure for predicting the average writing time of large documents, I use it, particularly as I haven't found anything that works better.

On large projects, developers usually produce many kinds of products. In addition to code, there may be manuals, databases, support packages, or other items. Because these products all require development effort, you should plan their development. Some of the size measures that development projects typically use are document pages, test scenarios, requirements pages, and database fields or records. Again, to be useful, these measures must meet the planning criteria for usefulness: the size values must correlate with the time needed to develop the products, and

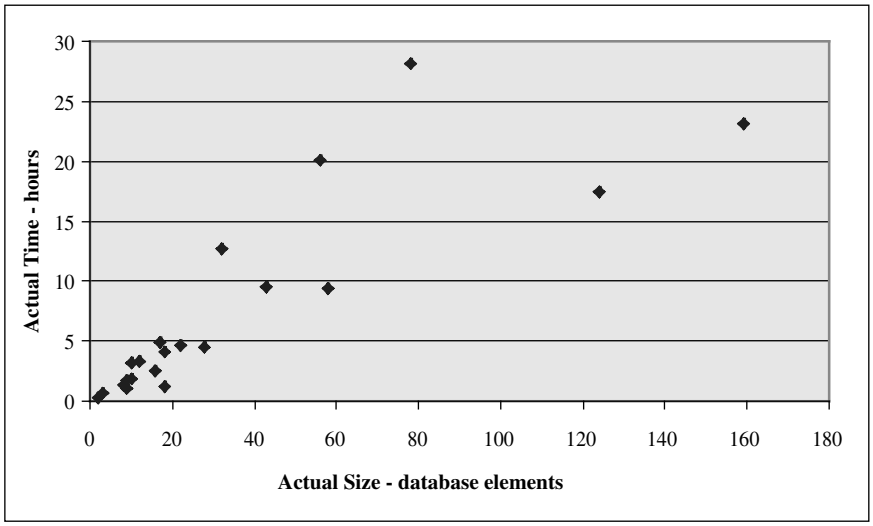

FIGURE 3.1 DATABASE ELEMENTS VERSUS DEVELOPMENT TIME

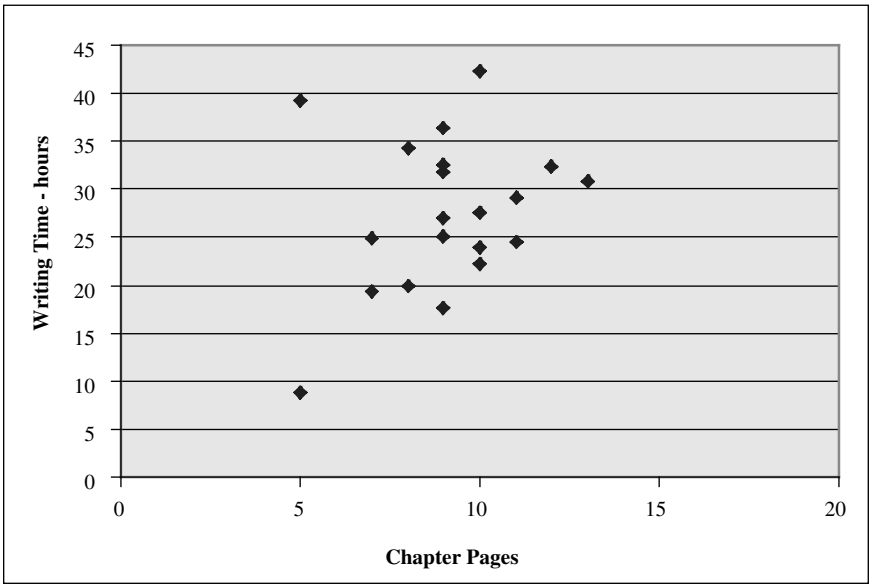

FIGURE 3.2 BOOK CHAPTER PAGES VERSUS WRITING TIME

the measures must be precise, specific, and automatically (or otherwise easily) countable:

- **Precise Size Measures:** A precise size measure gives an exact value for the size of a product. To be precise, the measure must be exactly defined by the product's content. Then, every time the size is measured, the result must be the same, with no room for judgment or opinion.
- **Specific Measures:** A specific measure is one that is based on defined product properties. For example, a C++ size measure would be defined differently from one for VB, SQL, Pascal, and so forth.
- **Countable Measures:** Once you have selected a precise and specific size measure, you need an automated means to count it. Manually counting almost any size measure is tedious, time-consuming, and inaccurate. For large products, it is practically impossible.

3.2 Establishing a Database Counting Standard

For database work, there are many ways to count product elements. For example, if you were producing a database, you might count fields or tables. When developing programs to use a database, you might count queries, GUI elements, tables, or code lines. In addition, when using a GUI tool, you could count the buttons, boxes, labels, and so on. The key, however, is to determine which of these elements is likely to be most significant in determining your development effort. In the example shown in Figure 3.1 (see p. 39), the developer counted all of the database tables and forms and all of the buttons and controls in the GUI interface. Because the correlation had an r of 0.84 with a significance greater than 0.01, this was a useful size measure. However, the developer found that some large database queries also required a significant amount of VB code. Because he had not included a line-of-code (LOC) measure to cover the code development work, his estimates for a couple of larger jobs were too low.

3.3 Establishing a Line-of-Code Counting Standard

A measure of the number of source lines of code (sometimes called **SLOC**) meets all the criteria for a useful size measure for many kinds of programming. Because LOC correlates well with development effort for the exercise programs in this book, this measure is used in many of the examples in this and later chapters.

However, the PSP estimating methods are equally usable with other size measures such as database elements, document pages, forms, or any other measure that meets the size-measurement criteria and correlates with development effort.

The first step in defining a LOC counting standard is to establish a counting strategy. You could take a large-scale view of logic statements and treat each construct as a single countable line. Then, using a simple Pascal-like language, for example, the statement

```
if A>B
    then begin A := A–B end
    else begin A := A+B end;
```

would be one statement. Similarly, it would still be one statement if you wrote it as

```
if A>B
    then
        begin
            A := A–B
        end
    else
        begin
            A := A+B
        end;
```

On the other hand, you could count selected keywords and functions such as **if**, **then**, **else**, **begin**, **end**, and **assignment** (:=). Your count in this case would be nine. Although the choice is arbitrary, you must be explicit and consistent.

For my work, I have adopted the following simple guideline:

- Count logical statements such as, for example, every semicolon and selected keywords.
- Write every countable statement on a separate line.
- Count all statements except blank and comment lines.
- Count each language separately.

These guidelines produce open and clean-looking code that looks like the preceding nine-line example. It also simplifies automatic LOC counting. Blanks and comment lines are not counted because the number of logical program statements generally determines the development effort. If, however, you find that the time required to write comments is significant, then you could decide to count

them. The way to decide this would be to measure size with and without counting comment lines to determine which provides a higher correlation with development time.

3.4 Size Accounting

Even after selecting the size-measurement definition, there are many ways to use the measures. This is particularly true when you work on a development team that produces multiple versions of large legacy products. To track all of the program changes and additions and to gather the data you need for estimating, you must use a size accounting system.

Although size seems like a simple concept, size measures can be hard to track if they are not properly defined and used. For example, if you started with a 1,000-LOC program and added 100 LOC, the result would be 1,100 LOC. If you merely wrote a new 100-LOC program, however, the result would be a 100-LOC program. Although this result seems obvious and the development effort would probably be roughly comparable, the sizes of the resulting products are vastly different. Size accounting will help you to understand and properly use size data. The basic program elements in the size accounting system are as follows:

- **Base:** This is the measure of the initial unmodified base program to which subsequent enhancements are added. For a new program, the base size would be zero.
- **Added:** This is the new code that is added to the base. Again, for a new program, the base size would be zero and the added code would be all the code.
- **Modified:** The modified code is that part of the base code that is changed.
- **Deleted:** The deleted code is that part of the base code that is removed and not used in the next program or program version.
- **Reused:** When an existing program or program routine is copied from some library or otherwise included in a new program, it is counted as reused code only if it is unmodified. If it is even slightly modified, the size of this included program is included in the base size and the changes are counted as modifications.
- **Added and Modified:** For many kinds of software development work, the effort required to write a new LOC is roughly comparable to the effort required to modify an existing LOC. In these cases, it is convenient to combine the Added size measures and Modified size measures into an "Added and Modified" size measure. This is the measure generally used in the PSP exercises for estimating development effort and for measuring and managing quality.

- **New Reusable:** A common development strategy is to build a library of commonly used functions and to reuse them whenever they are subsequently needed. Because this strategy can save a great deal of effort, the PSP includes a New Reusable size measure for the newly developed code that is planned for inclusion in the reuse library. If you rework base code to be reusable, also mark it with an asterisk.
- **Total:** Total program size measures the entire program, regardless of how its parts were produced.

The reason to track program additions, deletions, and modifications is illustrated in Figure 3.3. Starting with version 0, which has a counted size of 350 LOC, you add or modify 125 LOC. Thus, you expect to have a finished version 1 of 475 LOC. When you measure version 1, however, its total size is only 450 LOC. Where did the 25 LOC go?

This can best be explained by using the size accounting format shown in Table 3.2. Here, starting with a new program base of zero LOC, you develop 350 new LOC, for a total of 350 LOC in version 0. This 350 LOC is the base on which you develop version 1. In developing version 1, you add 100 LOC and modify 25 LOC of the V1 base. The 125 LOC is thus made up of 100 LOC new and 25 LOC modified. Notice, however, that the 100 LOC are shown only in the *Added* column on the left of Table 3.2, but the 25 LOC are shown in both the *Added* and *Subtracted* columns. The reason is that a modified LOC must be counted as a one-line addition and a one-line deletion. This is because that line is already counted in the base. If you don't subtract the 25, you will double-count it. Version 1 (or version 2 base) now is shown to contain 450 LOC, just as the counter indicated.

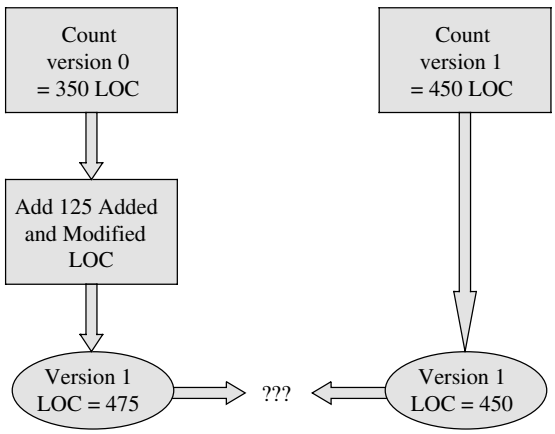

FIGURE 3.3 THE SIZE ACCOUNTING PROBLEM

TABLE 3.2 SIZE ACCOUNTING EXAMPLE

	Added		Subtracted		Net Change		Base
Base V0							0
Added	350						
Deleted			0				
Modified	0		0				
Reused	0						
Totals V0	350	–	0	=	350	+	= 350
Base V1							350
Added	100						
Deleted			0				
Modified	25		25				
Reused	0						
Totals V1	125	–	25	=	100	+	= 450
Base V2							450
Added	60						
Deleted			200				
Modified	75		75				
Reused	600						
Totals V2	735	–	275	=	460	+	= 910
Final Product							910

Another way to describe the relationship of these various size measures is shown in Figure 3.4. Here, the base for version 2 is shown as the 450 LOC from version 1. In developing version 2, we delete 200 LOC from the base, modify 75 LOC, and leave 175 LOC unmodified. Of the original 450-LOC circle, 200 LOC is outside the version 2 circle and both the 75 modified and 175 unmodified LOC are within the version 2 circle. To this we add the 60 lines of added code and the 600 lines of reused code to arrive at a final version 2 total size of 910 LOC (175 + 75 + 60 + 600).

All of the items in Table 3.2 must be counted for every version or the totals will not likely be correct. One nice feature of the LOC accounting structure is that it is additive. This is particularly important when many groups contribute code to a large system. Then, after developing several versions, you can add the numbers of added, deleted, modified, and reused code to get the combined effect of the total development effort for all of the versions. For example, in the case shown in Table 3.2, the initial base code is 0, the added code is 350 + 100 + 60 = 510 LOC, the deleted code is 0 + 0 + 200 LOC, and the modified code is 0 + 25 + 75 = 100 LOC. If we

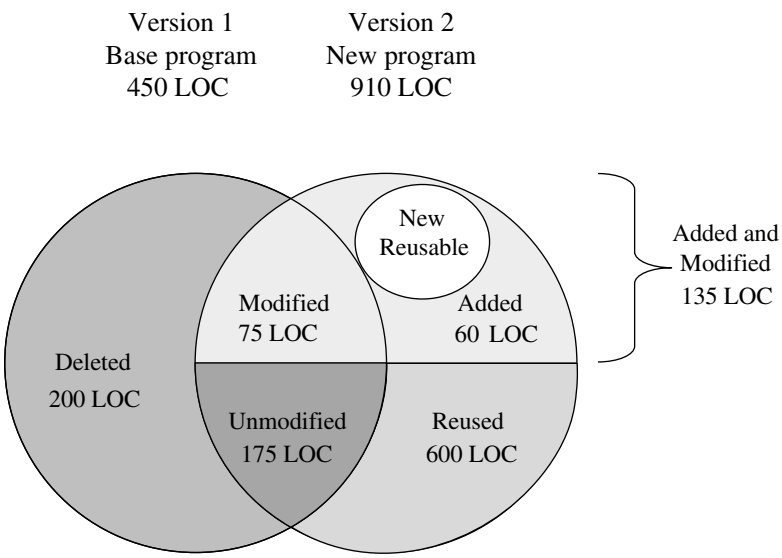

FIGURE 3.4 THE VERSION 2 SIZE ACCOUNTING EXAMPLE

count the modifications as additions and deletions, the total is the additions (510 + 100 = 610) minus the deletions (200 + 100 = 300) plus the reused code (600), giving 610 – 300 + 600 = 910 LOC as the correct final total LOC.

Although size accounting may seem unnecessarily complex, after several releases of a large program you may need to know where all of the code came from and the effort required to develop it. This accounting scheme permits you to find this out. It also gives you a way to precisely account for all of the size changes in a large number of releases. This exact approach can be used to track the sizes of documents, databases, or any other product for which you have defined size measures.

3.5 Using Size Data

The principal ways to use size data are in planning, quality management, and process analysis. The following paragraphs describe how to use size measures in planning, their value in assessing program quality, and how they help in evaluating your personal work.

Using Size Measures for Planning

In planning a development job, if you have a defined size measure and historical size and time data, you can use these data to accurately estimate the size of the new product. With these same data, you can also accurately estimate the development effort. For the PSP exercises in this book, I suggest that you use the added and modified code (or database elements) and not the deletions or other unmodified inclusions in estimating development effort. The effort to add or modify code in these exercise programs will probably be roughly the same, while the effort required to delete or include a previously developed line will generally be much less. For other kinds of work, however, you might make different choices. In maintenance work, for example, the effort required to delete a line could take as much or more time than the effort required to add or modify a line. Examine each situation and be guided by your data. Some methods for making such choices are discussed in Chapter 5.

Assessing Program Quality

In evaluating program quality, it is often helpful to make cross-product comparisons. For example, dividing the number of defects found in a phase by that program's size gives the program's **defect density** for that phase. This is important in planning projects because testing, maintenance, and service costs are generally closely related to a program's defect content at various points in the process. With data on similar programs, you can use estimates of defect density and size to estimate testing, maintenance, and service costs. The defect-density measure can partially compensate for size differences among programs and take advantage of the historical data for a larger number of prior projects.

In calculating defect density, count only the code added and modified during development. When considering the relative quality of several finished programs, however, consider their total size. Although I know of no published data on this point, my experiences at IBM suggest that total program size correlates most closely with product service costs. Although this may seem strange, it is logical when you realize that, for all but very poor-quality products, defect-related costs are only a small part of total service costs. At IBM, we found that the total number of support calls was more closely related to total program size than to the added and modified size of each release. Once you have sufficient data, you can decide what method works best for you and your products.

For program quality, defect density is generally measured in defects per 1,000 LOC, or KLOC. For finished products, however, the most suitable measure is defects per 1,000,000 LOC, or defects per MLOC. For measures of database work, document writing, or other product development, defects are generally counted per 100 or 1,000 elements, depending on which measure gives numbers between about 10 and 100.

Every counting choice has advantages and disadvantages. Using total program size as a quality measure is most appropriate when small modifications are not significant. However, an unpublished IBM study found that small code modifications were 39 times as error-prone as new development when measured in defects per modified LOC (Humphrey 1989). Conversely, counting only added and modified code would ignore any quality problems with the large inventory of existing code. If this existing code had any significant defect content, maintenance cost estimates would then likely be too low. In determining the relative quality of several releases of a single program, one is generally interested in both the defect density for the added and modified code and the defect density of the total product. Again, when in doubt, gather the data and see what works best in your particular situation.

Evaluating Your Personal Work

For the PSP, the principal focus is on the quality of your *personal* process. Here, you should probably use data on added and modified database objects, lines of code, or other suitable measures to make the quality analyses in this book. However, you should also keep track of the size of the reused, added, deleted, and modified code. The PSP process shows you how to do this.

3.6 Calculating Productivity

Productivity is generally measured as the labor hours required to do a unit of work. It is such a simple concept that people tend to think of productivity as a simple calculation. Although the calculation is simple, using the right data and properly interpreting these data is not so easy. Proper productivity calculations must consider, for example, the unique conditions of each job and the quality of the resulting product. If you estimate the size of a new job and then use some global productivity factor to determine the total hours required, you will generally get a misleading result. It would be analogous to always allowing for the average driving time when you go to the airport. With that method, you will miss a lot of flights around rush hour and do a lot of waiting the rest of the time.

When calculating productivity, divide the size of the product produced by the hours spent producing it. This gives the volume of product produced per hour. For programming, for example, if you developed a 600-LOC program in 60 hours, your productivity would be 10 LOC per hour. Although this may seem simple enough, you face many choices for picking these size and hours measures. For example, various combinations of the numbers used in the LOC accounting example in Table 3.2 (see p. 44) are shown in Table 3.3. There is some basis for

TABLE 3.3 VARIATIONS IN LOC PRODUCTIVITY

Productivity Option	Size LOC	Productivity LOC/Hour
Added (350+100+50)	500	8.33
Added + Modified (500+100)	600	10.00
Added + Modified + Deleted (600+200)	800	13.33
Added + Modified + Reused (600+600)	1200	20.00
Added + Modified + Deleted + Reused (1,200+200)	1400	23.33
Total Finished Product	900	15.00

arguing that any one of these six numbers should be used to measure development productivity. Assuming that you spent a total of 60 hours developing this program, the various productivity options could differ by factors of two or more times. This variation is caused solely by the choice you make for using the size measures. That is why comparisons of productivity numbers between organizations or even among individuals are rarely useful and why each developer must choose his or her own method and use it consistently.

For most new or major enhancement projects, added plus modified product is probably most appropriate for calculating productivity. For small modifications to large programs, however, the issue is much more complex. The reason is that the productivity for software modification is often much lower than for new development (Flaherty 1985). By gathering data on the time spent, the sizes of the changes, the size of the base program, and the fraction of the base program modified, you will likely find that small changes in large programs take much more time than either new development or large-percentage modifications of existing programs. Study your own data and see what categories provide the best correlation with development effort. Again, the methods discussed in this book will help you to arrive at appropriate productivity figures.

3.7 Size Counters

Although it is relatively easy to count the sizes of small programs, manually counting even small programs can be time-consuming and error-prone. For large programs, however, manual counting is impractical, making automated size counters essential.

Counters can be designed to count the number of almost any product element as long as the definition for that element is precise and specific. For example, a

LOC counter could count physical lines or logical lines. Although automatically counting database elements can be a bit more complex, if the object model is accessible, you can usually count selected sets of database elements.

Physical LOC Counters

The simplest type of size counter counts physical LOC. Here, you count all text lines except comments and blank lines. A text line that has both comments and source code is counted as one LOC.

Logical LOC Counters

Logical LOC counters work much like physical counters except that line-counter stepping is more complex. My Object Pascal logical line counter had 939 LOC and took me about 50 hours to develop. Don't try to develop a logical line counter unless you have plenty of time to do it. Although people can endlessly debate the best way to count LOC, the details are not that important. The key is to be precise and consistent. In my Object Pascal and C++ programs, I counted every logical program statement. Although some may argue that counting **begin** and **end** statements is not a good idea, this is a matter of personal preference. If you could show, however, that the time required to develop a program had a higher correlation with LOC when **begin-end** pairs were counted than when they were not, that would be a good reason to use them. This would also be the proper way to resolve almost any size counting question.

Presently, there is no compelling evidence to support any one counting method over any other. With good data, however, a study of the various alternatives could quickly show which, if any, would be best for your work. It may show, as is likely, that many different approaches produce roughly equivalent results. In that case, the selection is arbitrary. The fact that it is arbitrary, however, does not mean that everyone should do it differently. If all of the projects in an organization used the same counting standard, they could afford to develop a single high-quality automated counter for everyone to use. They would also have a much larger volume of data to use in planning and quality analysis.

Counting Program Elements

It is often desirable to measure the sizes of various parts of large programs. For example, if you intended to reuse some newly developed classes or procedures, you would like to know how large each one is. As you will see in Chapter 5, such counts can also help with size estimating.

To maintain separate counts for each class or procedure, you must determine where each one starts and ends. Because the way to do this depends on the programming language, there is no general guideline. The counter I built for Object Pascal recognized the keywords **procedure**, **function**, **constructor**, and **destructor** as the beginning of procedures. I found the endpoints by counting **begin-end** pairs. With a little study, you can determine the indicators for the beginning and ending points of the methods, procedures, classes, functions, or other elements for the language you use. You can then use this definition to scan the program text and start and stop the size counter at the appropriate points.

Using Coding Standards and Physical LOC Counters

One simple way to count logical LOC is to use a coding standard and count physical LOC, omitting comments and blanks. The idea is to write the source code with a separate physical text line for every countable line. Then, when you count physical lines, you are also counting logical lines. To use this method, you must carefully follow a coding standard like the example shown in Table 3.4. If, at the same

TABLE 3.4 EXAMPLE C++ CODING STANDARD

Purpose:	To guide the development of C++ programs
Program Headers	Begin all programs with a descriptive header.
Header Format	/***/ /* Program Assignment: the program number */ /* Name: your name */ /* Date: the date program development started */ /* Description: a short description of the program */ /* function */ /***/
Listing Contents	Provide a summary of the listing contents.
Contents Example	/***/ /* Listing Contents: */ /* Reuse instructions */ /* Modification instructions */ /* Compilation instructions */ /* Includes */ /* Class declarations: */ /* CData */ /* ASet */ /* Source code in c:\classes\CData.cpp: */ /* CData */ /* CData() */ /* Empty() */ /***/

TABLE 3.4 (continued)

Reuse Instructions	• Describe how the program is used. Provide the declaration format, parameter values and types, and parameter limits. • Provide warnings of illegal values, overflow conditions, or other conditions that could potentially result in improper operation.
Reuse Example	/********** /* Reuse Instructions */ /* int PrintLine(char *line_of_character) */ /* Purpose: to print string, 'line_of_character', on one print line */ /* Limitations: the maximum line length is LINE_LENGTH */ /* Return: 0 if printer not ready to print, else 1 */ **********/
Identifiers	Use descriptive names for all variables, function names, constants, and other identifiers. Avoid abbreviations or single letter variables.
Identifier Example	int number_of_students; /* This is GOOD */ float x4, j, ftave; /* These are BAD */
Comments	• Document the code so that the reader can understand its operation. • Comments should explain both the purpose and behavior of the code. • Comment variable declarations to indicate their purpose.
Good Comment	If (record_count > limit) /* have all the records been processed? */
Bad Comment	if(record_count > limit) /* check if record_count is greater than limit */
Major Sections	Precede major program sections by a block comment that describes the processing that is done in the next section.
Example	/********** /* This program section will examine the contents of the array */ /* "grades" and will calculate the average grade for the class. */ **********/
Blank Spaces	• Write programs with sufficient spacing so that they do not appear crowded. • Separate every program construct with at least one space.
Indenting	• Indent every level of brace from the previous one. • Open and close braces should be on lines by themselves and aligned with each other.

Indenting Example	while (miss_distance > threshold)
	{
	success_code = move_robot (target_location);
	if (success_code == MOVE_FAILED)
	{
	printf("The robot move has failed.\n");
	}
	} (*continued*)

TABLE 3.4 (continued)

Capitalization	• Capitalized all defines. • Lowercase all other identifiers and reserved words. • Messages being output to the user can be mixed-case so as to make a clean user presentation.
Capitalization Example	#define DEFAULT-NUMBER-OF-STUDENTS 15 int class-size = DEFAULT-NUMBER-OF-STUDENTS;

time, you also follow consistent formatting standards, you will get highly readable code. Note, however, that when you count LOC this way, you cannot reformat the code without changing the LOC count.

One problem with this approach is caused by instructions that span multiple text lines. Because the LOC counter will count each line, you will get too large a count. In these cases, either modify the counter to recognize multi-line instructions, write a true logical LOC counting program, or count these cases by hand. If multi-line instructions are rare in your programs, the easiest approach is to ignore the problem and trust your estimating method to allow for those cases. If such instructions are common, however, you should either count the cases by hand or adjust your counter to properly count multi-line instructions.

Counting Deletions and Modifications

One of the more difficult counting problems is keeping track of the additions, deletions, and changes in multiversion programs. For small programs, you can manually count the added and modified elements, but for large programs this is impractical. Even for small programs, such manual counts are almost impossible unless you do the counting immediately after completing each modification.

One approach is to flag each program line separately with a special comment. A special counter could then recognize these flags and count the added and deleted product elements. Although nice in theory, this approach is impractical. While modifying a program, you will find it almost impossible to remember to comment every addition and deletion. It is even more difficult to remember to do it when you are fixing defects in testing. A second alternative is both practical and convenient, but it involves a bit of programming. Here, you use a special program counter to compare each new program version with its immediate predecessor and insert appropriate flags.

Because this comparator could not easily recognize whether a new line was a modification or an addition, all modifications would be counted as deletions and

additions. This is exactly equivalent to the LOC accounting rules. You could even run the comparator counter after every modification and defect fix.

Another approach is to build such a comparator counter into the source-code control system. That way, every program update would be analyzed and every line automatically flagged with the date and version when it was added or deleted. You could also assign a number to every program fix, release, or version update and affix that number and date to every deleted, added, or modified line in each update. You could also include the developer's name, the fix number, the project name, or any other potentially useful information. With such a support system, you could determine when every change was made, when a defect was injected, or how much code was added and deleted with each update. These data are essential for effective quality management of large programs. Modern source-code control systems generally provide many of these capabilities.

Overall, the problem of counting deletions and modifications can be partially addressed with tool support, but it is still a difficult process that is both error-prone and apt to produce misleading results. The best approach is to make an accurate count for each small program and then use whatever automated tools you can obtain to identify and track all subsequent changes.

3.8 Other Size Measures

Although this chapter concentrates almost entirely on the line-of-code (LOC) measure, that is only one size-measurement option. As noted earlier, depending on the type of work you do, database elements, defect fixes, or pages might be more appropriate. Another useful and much more general size measure is called **function points**. The function-point measure was developed in the late 1970s and has become widely used. There is now a standards body, the International Function Point Users Group (IFPUG), that supports function points, and a considerable amount of literature is available (Garmus and Herron 2001; Jones 2000).

The principal point to remember is that no size measure is best for all situations. Whether you prefer LOC, database elements, or function points, the best strategy is to gather data on your estimated and actual size and time measures. Then analyze these data to see which measures and methods are best for you in each situation. Regardless of what I or anyone else says, if your data indicate that some measure and method is best for you, it probably is.

3.9 Summary

There are many reasons to measure software size. One of the most important is to help in project planning. The size measure should correlate to development effort, be precise, and be automatically countable. Typical size measures are database elements, source program lines of code, pages of design or text, and function points. Because there are many ways to define software size data, you must use precise definitions and carefully note the types of elements to be included in each count.

Size counters can be designed to count physical or logical product elements or lines. A third method combines these two: counting logical lines by using a coding standard and a physical counter. This is the approach the PSP uses to count LOC. For database work, count a selected set of database elements.

Properly designed tools can produce many useful size statistics. For each program version, for example, you need to know the number of added and deleted elements or lines. A practical way to get such data is with a program that compares each new program version with its immediate predecessor. This comparator then identifies and counts the added and deleted items.

Well-defined and consistently used size measures can provide the data needed to make accurate size and development time estimates. Size data are also useful for comparing the defect rates among programs. Here, you use the defect rate per hundred new and modified database elements or per thousand lines of new and modified code. When estimating maintenance workload for a program, however, consider total product size.

An effective software size measure must fit your and your team's needs. First, define the reason you want to use the measure and then gather historical data on both your work and the proposed measure. Then make sure that your measurement practices satisfy your intended purpose for the measure.

3.10 Exercises

The standard assignment for this chapter includes exercises R1 and R2 and uses the PSP0.1 process to write one or more programs. For the R1, R2, and program assignment kits, see your instructor or get them at **www.sei.cmu.edu/tsp/psp**. These kits contain the assignment specifications and the PSP0.1 process scripts. In completing this assignment, faithfully follow the PSP0.1 process, record all required data, and produce the R1 and R2 reports and the program report according to the specifications in the assignment kit.

References

Flaherty, M. J. "Programming Process Productivity Measurement System for System/370." *IBM Systems Journal* 24, no. 2 (1985): 168–175.

Garmus, D., and D. Herron. *Function Point Analysis.* Boston: Addison-Wesley, 2001.

Humphrey, W. S. *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

Jones, C. *Software Assessments, Benchmarks, and Best Practices.* Boston: Addison-Wesley, 2000.