Estimating With Objects - Part II

Contents

This is the second of a series of columns on software project estimating. The first column last month gave an overview of the subject. If you have not read that column, I suggest you do so. Otherwise, you may wonder why I am covering some of the topics and you will almost certainly have trouble seeing how these columns fit into an overall framework.

This column discusses program size and it provides a general background for all the columns to follow. Before discussing size estimating, it is important to cover size measurement. The discussion of size estimating starts next month. To repeat what I said last month, if you are in a hurry to get to the punch line, this set of columns describes an estimating method called PROBE which is described in my book, *A Discipline for Software Engineering*, from Addison Wesley.

Why are we interested in program size?

The principal reason software engineers are interested in the size of a program is to help estimate the time required to develop that program. We all know that it takes longer to develop larger programs than smaller ones. It thus seems obvious that a good size estimate would help in making accurate estimates of development time.

While the correlation between size and development time is not always high, it is a well established fact that program size and development time are directly related. Thus, if you knew the size of a planned program, if you had historical productivity data, and if you used the right methods, you should be able to make a pretty good estimate of the time to develop the program. All of this, of course assumes that you are using an appropriate size measure.

The criteria for a size measure

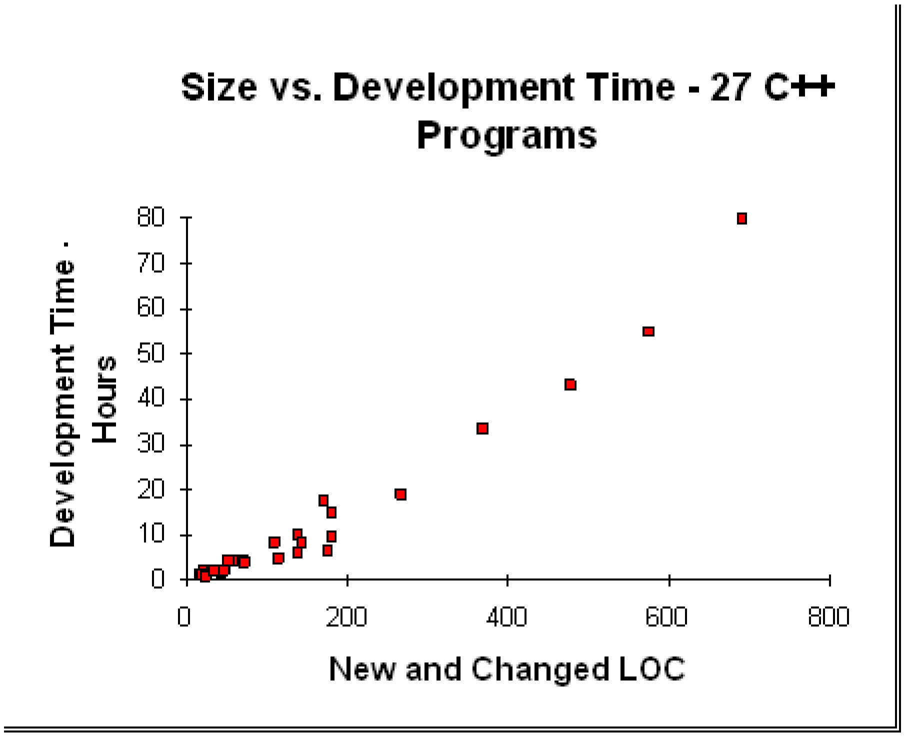
For estimating purposes, a good measure of a program's size should correlate closely to the time required to develop that program. That, however, is not the only criterion. Other criteria are:

- The size measure must be precise. That is, several people measuring the same thing should get precisely the same result. Note that while precision implies accuracy, they are not the same. Very precise measures can be inaccurate. For example, a statement that 30.8% of programmers used object oriented methods sounds impressive largely because of its precision. If, however, you found that this number was based on the finding that four of 13 people in one organization used object oriented methods, you would not be as impressed. It is important not to mistake the precision of a result for its accuracy.
- Good size measures should be automatically countable. This is immediately obvious when you have a large body of programs to measure. At IBM, we had about 30,000,000 lines of code (LOC) in the MVS operating system. The only size counts we would accept were those generated by the XREF tool. This tool automatically counted source programs and provided consistent data for all products.
- Good size measures should also reflect any environmental conditions that affect development cost. For example, if the language used had a significant impact on development cost, the size measure should be specific to the language used. If it wasn't, the size measure could introduce estimating errors. Similarly, if the application domain significantly impacted development costs, that should also be reflected in the size measure. In one case, we found that the productivity for application development was about six times higher than for communications or system control programs. Again, wherever possible, you should use size and development time data for the specific language and application domain of the new development.
- The mix of new, reused, deleted, and modified code should also be reflected in the size measures and counts. Engineering productivity varies significantly depending on the mix of new, reused, and modified code. We found that productivity when developing new code was three to six time higher than for small program modifications.
- Another important requirement is that the measure be usable throughout the development process. You obviously want to estimate program size during planning and you also want to measure and track size during development. Finally, to get data for future estimates, you must measure size after the product has been completed.
- While the measure should also be usable with every type of product element, this is a much more difficult requirement. Examples of some of the more common element types are program source code, documentation pages, screens, reports, menus, and files.

Size versus development time

In my work, I have found LOC to be the most useful size measure. While LOC do not completely meet all the above criteria, they are precise, machine countable, environment specific, and they can reflect the mix of reused, modified, and new code. The principal disadvantage is that LOC are hard to visualize early in a project. The PROBE method, however, is designed to address this problem. With correct estimating methods, LOC are thus generally suitable for estimating almost all types of programming. There are exceptions, however, which I mention later in this column.

One of the reasons that I am so positive about LOC size measures is that the size and development time for the C++ programs I have developed are highly correlated. The data for 27 module-sized C++ programs are shown in the figure. Here, the correlation is 0.979 with a significance of better than 0.005. This means that once I have estimated the size of a new program, I can readily calculate the time it will take me to develop it. Estimating the size, of course, is a non-trivial job. That, however, is the subject of subsequent columns.



Types of LOC

In using LOC to measure size, it is important to recognize the various types of LOC. For example, new LOC concern those LOC you develop from scratch for a specific program. Many programs, however, are modifications or enhancements of prior programs, so the modified LOC must be counted separately. There is also something I call base LOC. This refers to those LOC in the base program that you modify when developing a new program version. For example, if you had an existing version 1 of a program and were going to develop version 2, version 1 would be the base for version 2. All the version 1 LOC would then be counted as base LOC for version 2.

There are also reused LOC. Here I distinguish between reused and base LOC because reuse and version enhancement are substantially different activities and take different amounts of time. The typical intent of a reuse strategy is to establish a library of standard parts to use when building new programs. The objective is to save the time it would take to redevelop these standard functions every time they were needed. This is quite a different strategy from reusing the base LOC from the previous program version when developing a new version. While they are both forms of reuse, the productivities are apt to be quite different Productivity for reusing standard library routines is generally quite high while the productivity for reusing a base program can involve considerable study, particularly if the design is not well documented.

Measuring program size

The most general approach to size measurement is to count the number of text lines in a source program. In doing this, we typically ignore blank lines and lines with only comments. All other text lines are counted. This approach has the advantage of being simple and easy to automate. In fact, the simple LOC counters my students write are typically around 100 LOC.

This LOC counting approach has the disadvantage of being sensitive to formatting. Those programmers who write very open code will get more LOC for the same program than would their peers who used more condensed formats. As a consequence, I suggest you establish coding and counting standards to cover program formatting. In doing this, you should also establish the practice of putting a logical LOC on each physical line of the source program. This, of course, means you must define what you mean by a logical LOC.

The SEI size measurement framework

To address this problem, the SEI has established a size measurement framework. It is described in the technical report Bob Park and others at the SEI developed called: *Software Size Measurement: A framework for Counting Source Statements,* CMU/SEI-92-TR-020. This SEI method both defines what a logical LOC is and provides a consistent way to define size measures. A consistent size measurement and counting system will permit you to gather the data you need to support an effective size estimating program and it will enable different people in your organization to compare their measures. If they used different size measurement criteria, they would obviously get different size measures for the same thing.

Some years ago, I ran into an extreme case of this size definition problem. Shortly before I retired from IBM, corporate management was very concerned about the much higher programming productivities reported by several Japanese computer manufacturers. Several of us went to Japan to find out what they were doing. We found that they counted LOC in much the same way we did. In using these LOC, however, they had entirely different practices. Instead of only counting the new and changed LOC, they counted all LOC, even those that were reused. In one 100,000 program, for example, that only had 10,000 LOC of new and changed code, we would only count 10,000 LOC while they counted 100,000 LOC. Also, they added a LOC factor for every page of documentation. We added nothing. They even adjusted the LOC counts to get equivalent assembly language LOC. In doing this they multiplied by a factor of, as I recall, 6. They did not share their reasons for selecting this factor. Our data suggested it should be closer to 3. In IBM, however, we made no such adjustment.

Just on the size measure alone, we could account for differences of nearly 100 to 1 in productivity. This did not count the differences due to the way the labor content were counted, which were also significant. After making all these corrections, our productivity appeared to be roughly equal or even slightly better than the Japanese numbers we had heard.

Some other size measures

LOC is not usable for counting many other aspects of program size. For example, documentation is a major expense and should be estimated and counted. Similarly, files, reports, screens, and many other aspects of 4-GL programming are not readily countable by LOC. The key point to remember is that the size measure should correlate with the development effort for that same type product or product element. In examining a number of 4GL programs, it is clear that the labor to develop these programs is not related to the LOC in the finished program. The reason is that much of the finished code is automatically generated. Some programs that take relatively little labor can thus generate a great many LOC. Similarly, some relatively high-labor developments can generate few LOC. The key is to determine what countable elements correlate with the labor required. There may be such elements, and there may not. When you cannot find suitable countable elements, you must count the entire entity, such as the number of reports, screens, or files. You would then need historical productivity data for these elements. Get as much historical data as you can, arrange these size data in the order of their development labor, and then see which measure gives the highest correlation. The PROBE method explains how to do this for any kind of size measure.

Function points

Other than LOC, the only other major program size measure is function points. Actually, function points do not measure program size. They provide a way to judge the relative magnitude of a program development job by analyzing the functions the program is to contain. Unfortunately, however, there are no direct ways to precisely count the actual function point content of an existing program. This means that organizations have difficulty in determining the relationship between the actual number of function points in a program and the time it took to develop that program. Even with this shortcoming, many people have found function points helpful in making programming estimates. Function points can also be used with the PROBE method.

If you are interested in further exploring function points, I suggest you check the publications from the International Function Point Users Group or look at Capers Jones' excellent book, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, 1991.

Subsequent columns

In subsequent columns of this series, I will address the elements of size estimating and discuss how object oriented methods can help in making size estimates. I will also describe how to make resource and schedule estimates, and present some data on how the PROBE method has worked in practice.

An invitation to readers

In these columns, I discuss software process issues and the impact of processes on engineers. I am, however, most interested in addressing the issues you feel are important. So please drop me a note with your comments and suggestions. Depending on the mail volume, I may not be able to answer you directly, but I will read your notes and consider them when I plan future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu