Overview

| **Applicability of Techniques in This Chapter** | | | |
| --- | --- | --- | --- |
|  | **Function Points** | **Dutch Method** | **GUI Elements** |
| What's estimated | Size, Features | Size, Features | Size, Features |
| Size of project | S M L | S M L | S M L |
| Development stage | Early-Middle | Early | Early |
| Iterative or sequential | Sequential | Sequential | Sequential |
| Accuracy possible | High | Low | Low |

Once you move from directly estimating effort and schedule to computing them from historical data, size becomes the most difficult quantity to estimate. Iterative projects might use a size estimate to help determine how many features can be delivered within an iteration, but they usually focus on techniques designed to estimate features more directly. Estimation in the later stages of sequential projects tends to focus on bottom-up effort estimates created by the people who will be doing the work. Estimating size is thus most applicable to the early and middle stages of sequential projects. The purpose of a size estimate is to support long-range predictability in the wide part of the Cone of Uncertainty.

The common size measures of lines of code and function points have different strengths and weaknesses, as do custom measures defined by organizations for their own use. Creating estimates by using multiple size measures and then looking for convergence or spread tends to produce the most accurate results.

This chapter describes how to create the size estimate. Chapter 19, "Special Issues in Estimating Effort," explains how to convert this chapter's size estimates into an effort estimate, and Chapter 20, "Special Issues in Estimating Schedule," describes how to convert the effort estimate into a schedule estimate.

18.1 Challenges with Estimating Size

Numerous measures of size exist, including the following:

* Features
* User stories
* Story points
* Requirements
* Use cases
* Function points
* Web pages
* GUI components (windows, dialog boxes, reports, and so on)
* Database tables
* Interface definitions
* Classes
* Functions/subroutines
* Lines of code

The lines of code (LOC) measure is the most common size measure used for estimation, so we'll discuss that first.

Role of Lines of Code in Size Estimation

Using lines of code is a mixed blessing for software estimation. On the positive side, lines of code present several advantages:

* Data on lines of code for past projects is easily collected via tools.
* Lots of historical data already exists in terms of lines of code in many organizations.
* Effort per line of code has been found to be roughly constant across programming languages, or close enough for practical purposes. (Effort per line of code is more a function of project size and kind of software than of programming language, as described in Chapter 5, "Estimate Influences." *What you get* for each line of code will vary dramatically, depending on the programming language.)
* Measurements in LOC allow for cross-project comparisons and estimation of future projects based on data from past projects.
* Most commercial estimation tools ultimately base their effort and schedule estimates on lines of code.

On the negative side, LOC measures present several difficulties when used to estimate size:

* Simple models such as "lines of code per staff month" are error-prone because of software's diseconomy of scale and because of vastly different coding rates for different kinds of software.
* LOC can't be used as a basis for estimating an individual's task assignments because of the vast differences in productivity between different programmers.
* A project that requires more code complexity than the projects used to calibrate the productivity assumptions can undermine an estimate's accuracy.
* Using the LOC measure as the basis for estimating requirements work, design work, and other activities that precede the creation of the code seems counterintuitive.
* Lines of code are difficult to estimate directly, and must be estimated by proxy.
* What exactly constitutes a line of code must be defined carefully to avoid the problems described in "Issues Related to Size Measures" in Section 8.2, "Data to Collect."

Some experts have argued against using lines of code as a measure of size because of problems associated with using them to analyze productivity across projects of different sizes, kinds, programming languages, and programmers (Jones 1997). Other experts have pointed out that variations of the same basic issues apply to other size measurements, including function points (Putnam and Myers 2003).

The underlying issue that's common to lines of code, function points, and other simple size measures is that measuring anything as multifaceted as software size using a single-dimensional measure will inevitably give rise to anomalies in at least a few circumstances (Gilb 1988, Gilb 2005).

We don't use single-dimensional measures to describe the economy or other complex entities. We can't even use a single measure to determine who the best hitter in baseball is. We consider batting average, home runs, runs batted in, on-base percentage, and other factors—and then we still argue about what the numbers mean. If we can't measure the best hitter using a simple measure, why would we expect we could measure something as complex as software size using a simple measure?

My personal conclusion about using lines of code for software estimation is similar to Winston's Churchill's conclusion about democracy: The LOC measure is a terrible way to measure software size, except that all the other ways to measure size are worse. For most organizations, despite its problems, the LOC measure is the workhorse technique for measuring size of past projects and for creating early-in-the-project estimates of new projects. The LOC measure is the *lingua franca* of software estimation, and it is normally a good place to start, as long as you keep its limitations in mind.

Your environment might be different enough from the common programming environments that lines of code are not highly correlated with project size. If that's true for you, find something that is more proportional to effort than lines of code, count that, and base your size estimates on that instead, as discussed in Chapter 8, "Calibration and Historical Data." Try to find something that's easy to count, highly correlated with effort, and meaningful for use across multiple projects.

|  |  |  |
| --- | --- | --- |
|  | Tip #80 | Use lines of code to estimate size, but remember both the general limitations of simple measures and the specific hazards of the LOC measure. |

18.2 Function-Point Estimation

One alternative to the LOC measure is function points. A function point is a synthetic measure of program size that can be used to estimate size in a project's early stages (Albrecht 1979). Function points are easier to calculate from a requirements specification than lines of code are, and they provide a basis for computing size in lines of code. Many different methods for counting function points exist. The standard for function-point counting is maintained by the International Function Point Users Group (IFPUG) and can be found on their Web site at *www.ifpug.org*.

The number of function points in a program is based on the number and complexity of each of the following items:

* ***External Inputs*** Screens, forms, dialog boxes, or control signals through which an end user or other program adds, deletes, or changes a program's data. They include any input that has a unique format or unique processing logic.
* ***External Outputs*** Screens, reports, graphs, or control signals that the program generates for use by an end user or other program. They include any output that has a different format or requires a different processing logic than other output types.
* ***External Queries*** Input/output combinations in which an input results in an immediate, simple output. The term originated in the database world and refers to a direct search for specific data, usually using a single key. In modern GUI and Web applications, the line between queries and outputs is blurry, but, generally, queries retrieve data directly from a database and provide only rudimentary formatting, whereas outputs can process, combine, or summarize complex data and can be highly formatted.
* ***Internal Logical Files*** Major logical groups of end-user data or control information that are completely controlled by the program. A logical file might consist of a single flat file or a single table in a relational database.
* ***External Interface Files*** Files controlled by other programs with which the program being counted interacts. This includes each major logical group of data or control information that enters or leaves the program.

Table 18-1 shows how the count of inputs, outputs, and so on gets converted to an Unadjusted Function Point count. You multiply the number of low-complexity inputs by 3, you multiply the number of low-complexity outputs by 4, and so on. The sum of those numbers gives you the Unadjusted Function Point count.

| Table 18-1: Multipliers for Computing an Unadjusted Function Point Count | | | |
| --- | --- | --- | --- |
| **Function Points** | | | |
| **Program Characteristic** | **Low Complexity** | **Medium Complexity** | **High Complexity** |
| External Inputs | \_\_ × 3 | \_\_ × 4 | \_\_ × 6 |
| External Outputs | \_\_ × 4 | \_\_ × 5 | \_\_ × 7 |
| External Queries | \_\_ × 3 | \_\_× 4 | \_\_ × 6 |
| Internal Logical Files | \_\_ × 4 | \_\_ × 10 | \_\_ × 15 |
| External Interface Files | \_\_ × 5 | \_\_ × 7 | \_\_ × 10 |
| Source: Adapted from *Applied Software Measurement, Second Edition* (Jones 1997). | | | |

After you've computed the Unadjusted Function Point total, you compute an Influence Multiplier based on the influence that 14 factors have on the program. These factors include data communications, online data entry, processing complexity, and ease of installation. The influence multiplier ranges from 0.65 to 1.35. When you multiply the unadjusted total by the Influence Multiplier, you get an Adjusted Function Point count.

If you've read my earlier comments about "subjective control knobs," you can probably guess what I think about the Influence Multiplier and its 14 control knobs. Two studies have found that Unadjusted Function Points are more strongly correlated with ultimate size than Adjusted Function Points are (Kemerer 1987, Gaffney and Werling 1991). Some experts also recommend eliminating the "low complexity" and "high complexity" judgments, and classifying all counted items as "medium," which eliminates another source of subjectivity (Jones 1997). The ISO/IEC 20926:2003 standard is based on Unadjusted Function Points.

Table 18-2 provides an example of how you would come up with the final Adjusted Function Point total. The specific number of inputs, outputs, queries, logical internal files, and external interface files shown in the table were chosen solely for purposes of illustration.

| Table 18-2: Example of Computing the Number of Function Points | | | |
| --- | --- | --- | --- |
|  | **Function Points** | | |
| **Program Characteristic** | **Low Complexity** | **Medium Complexity** | **High Complexity** |
| External Inputs | *6* × 3 = 18 | *2* × 4 = 8 | *3* × 6 = 18 |
| External Outputs | *7* × 4 = 28 | *7* × 5 = 35 | *0* × 7 = 0 |
| External Queries | *0* × 3 = 0 | *2* × 4 = 8 | *4* × 6 = 24 |
| Internal Logical Files | *0* × 7 = 0 | *2* × 10 = 20 | *3* × 15 = 45 |
| External Interface Files | *2* × 5 = 10 | *0* × 7 = 0 | *7* × 10 = 70 |
| **Unadjusted Function Point total** |  |  | **284** |
| **Influence multiplier** |  |  | **1.0** |
| **Adjusted Function Point total** |  |  | **284** |

The example illustrated here works out to a size of 284 function points. You can convert that directly to an effort estimate (described in Chapter 19), or you can convert it first to a lines of code estimate, and then convert that to an effort estimate.

The terminology in the function-point approach is fairly database-oriented, but IFPUG has steadily updated the rules for counting function points, and the approach works well for all kinds of software. Studies have found that certified function-point counters will usually produce counts that are within about 10% of each other, so function-point counting presents a real possibility of narrowing the scope-related variability in the Cone of Uncertainty early in a project (Stutzke 2005).

|  |  |  |
| --- | --- | --- |
|  | Tip #81 | Count function points to obtain an accurate early-in-the-project size estimate. |

Converting from Function Points to Lines of Code

If you want to convert to lines of code, Table 18-3 lists the conversion factors between function points and lines of code for several popular languages.

| Table 18-3: Programming Language Statements per Function Point | | | |
| --- | --- | --- | --- |
|  | **Programming Statements per Function Point** | | |
| **Language** | **Minimum (Minus 1 Standard Deviation)** | **Mode (Most Common Value)** | **Maximum (Plus 1 Standard Deviation)** |
| Ada 83 | 45 | 80 | 125 |
| Ada 95 | 30 | 50 | 70 |
| C | 60 | 128 | 170 |
| C# | 40 | 55 | 80 |
| C++ | 40 | 55 | 140 |
| Cobol | 65 | 107 | 150 |
| Fortran 90 | 45 | 80 | 125 |
| Fortran 95 | 30 | 71 | 100 |
| Java | 40 | 55 | 80 |
| Macro Assembly | 130 | 213 | 300 |
| Perl | 10 | 20 | 30 |
| Second generation default (Fortran 77, Cobol, Pascal, etc.) | 65 | 107 | 160 |
| Smalltalk | 10 | 20 | 40 |
| SQL | 7 | 13 | 15 |
| Third generation default (Fortran 90, Ada 83, etc.) | 45 | 80 | 125 |
| Microsoft Visual Basic | 15 | 32 | 41 |
| Source: Adapted from *Estimating Software Costs* (Jones 1998), *Software Cost Estimation with Cocomo II* (Boehm 2000), and *Estimating Software Intensive Systems* (Stutzke 2005). | | | |

If your 284-function-point program were to be implemented in Java, you would take the range of 40 to 80 LOC per function point from the table and multiply that by 284 function points to arrive at a size estimate of 11,360 to 22,720 LOC, with an expected value of 55 times 284, or 15,675 LOC. To avoid conveying a false sense of accuracy, you might simplify these numbers to 11,000 to 23,000 LOC with an expected case of 16,000 LOC.

The conversion factors presented in the table use wide ranges, typically a factor of 2 to 3 between the high and low ends of the ranges. As with many other quantities you estimate, if you can collect historical data about how function points translate into lines of code in your organization, you will be able to estimate more accurately and probably with narrower ranges than if you use industry-average data.

This section's description of function-point counting just skims the surface of a sophisticated technique. While expert function-point counters can produce results that are within 10% of each other, counts of inexperienced function-point counters will vary by 20% to 25% (Kemerer and Porter 1992, Stutzke 2005). For more details on the technique, see the "Additional Resources" section at the end of this chapter.

18.3 Simplified Function-Point Techniques

Function-point counting requires going through a requirements specification line by line and literally counting each input, output, file, and so on. This can be time consuming.

Estimation experts have proposed a handful of simplified approaches to counting function points. Considering the other sources of variability that feed into a software project in the early stages when function points are relevant, a focus on minimizing the effort required to obtain a not-very-accurate estimate seems appropriate.

The Dutch Method

The Netherlands Software Metrics Association (NESMA) suggests an "Indicative" method for early-in-the-project function-point counting (Stutzke 2005). In its method, rather than counting all inputs, outputs, and queries, only Internal Logical Files and External Interface Files are counted. An Indicative Count is then computed using this equation:

|  |  |
| --- | --- |
| (#8) | image from book |

The numbers 35 and 15 have been derived through calibration, and you would ultimately want to come up with your own calibrations for use in your environment.

The function-point counts created using this method will be less accurate than counts created using the full function-point counting technique described in Section 18.2, "Function-Point Estimation." But the effort required is much lower, and so this sort of approximation can be useful for rough estimates.

|  |  |  |
| --- | --- | --- |
|  | Tip #82 | Use the Dutch Method of counting function points to attain a low-cost ballpark estimate early in the project. |

GUI Elements

As an alternative to counting function points directly, you might count GUI elements instead. This is an example of proxy-based estimation, as described in Chapter 12, "Proxy-Based Estimates." The process follows these steps:

1. Count the number of GUI elements according to the categories in Table 18-4.
2. Convert the GUI elements to an approximate function-point count by transferring appropriate entries generated from Table 18-4 to the matrix shown in Table 18-1.
3. Calculate size in lines of code by using the relationships shown in Table 18-3.

| Table 18-4: Substituting GUI Elements for Function Points | |
| --- | --- |
| **GUI Element** | **Function-Point Equivalent** |
| Simple Client Window | 1 Low Complexity External Input for add, change, and delete (if present), plus 1 Low Complexity External Query |
| Average Client Window | 1 Average Complexity External Input for add, change, and delete (if present), plus 1 Average Complexity External Query |
| Complex Client Window | 1 High Complexity External Input for add, change, and delete (if present), plus 1 High Complexity External Query |
| Average Report | 1 Average Complexity External Output |
| Complex Report | 1 High Complexity External Output |
| Any File | 1 Low Complexity Internal Logical File |
| Simple Interface | 1 Low Complexity External Input if coming in; 1 Low Complexity External Output if going out |
| Average Interface | 1 Average Complexity External Input if coming in; 1 Average Complexity External Output if going out |
| Complex Interface | 1 High Complexity External Input if coming in; 1 High Complexity External Output if going out |
| Message or Dialog Box | Not counted; are counted as part of the screen they connect to |

If you use this approach, recognize how much uncertainty is feeding into your estimate. Some uncertainty likely exists in your original counts of the number of GUI elements or your estimates of them. You introduce additional uncertainty when you convert from GUI elements to function points. And you introduce still more uncertainty when you convert from function points to lines of code.

|  |  |  |
| --- | --- | --- |
|  | Tip #83 | Use GUI elements to obtain a low-effort ballpark estimate in the wide part of the Cone of Uncertainty. |

This chapter and other chapters in this book have presented numerous techniques for estimating size, including several techniques that can produce a size estimate in lines of code. Table 18-5 summarizes the techniques that have been presented so far.

| Table 18-5: Techniques for Estimating Size | | |
| --- | --- | --- |
| **Technique** | **Chapter** | **Kind of Size That Can Be Estimated** |
| Analogy | 11 | features, function points, Web pages, GUI components, database tables, interface definitions, lines of code |
| Decomposition | 10 | features, function points, Web pages, GUI components, database tables, interface definitions, lines of code |
| Dutch Method | 18 | function points, lines of code |
| Estimation Tools | 14 | function points, lines of code |
| Function Points | 18 | function points, lines of code |
| Fuzzy Logic | 12 | function points, lines of code |
| Group Reviews | 13 | features, user stories, story points, requirements, use cases, function points, Web pages, GUI components, database tables, interface definitions, classes, functions/subroutines, lines of code |
| GUI Elements | 18 | function points, lines of code |
| Standard Components | 12 | function points, lines of code |
| Story Points | 12 | story points, lines of code |
| Wideband Delphi | 13 | features, user stories, story points, requirements, use cases, function points, Web pages, GUI components, database tables, interface definitions, classes, functions/subroutines, lines of code |

The entries in the table's "Kind of Size That Can be Estimated" column really depend on the calibration data you have. The most common kinds of size data—and the most generally usable—are function points and lines of code.

As Chapter 15, "Use of Multiple Approaches," discussed, the best estimators usually use multiple estimation techniques and then look for convergence or spread among the estimates. The different approaches listed in Table 18-5 provide numerous options for estimating size in different ways and then comparing your estimates.

|  |  |  |
| --- | --- | --- |
|  | Tip #84 | With better estimation methods, the size estimate becomes the foundation of all other estimates. The size of the system you're building is the single largest cost driver. Use multiple size-estimation techniques to make your size estimate accurate. |