EXHIBIT 12

Software
Productivity
Research, LLC

SPR Programming Languages Table

Version PLT2007c

28 December 2007

Table of Contents

Software Productivity Research LLC

Programming Languages Table

Version 2007c – December 28, 2007

Introduction

In the 1970s Allan Albrecht and his colleagues at IBM measured a number of projects using both logical source code statements and function point metrics. These pioneering studies found some interesting (but not perfect) correlations between the volume of source code and function points for many programming languages.

In 1985, Capers Jones (founder of Software Productivity Research, Inc.) developed a commercial sizing methodology for the SPQR/20 software estimating tool called "backfiring" that allowed direct conversion between logical source code statements and International Function Point Users Group (IFPUG) function point metrics. Jones published the results of research into real-world examples of these correlations in the original Programming Languages Table (PLT). SPR developed the PLT to illustrate empirically-derived standard rates for common source code programming languages. The latest version of the PLT, published in 2006, now includes over 600 entries.

Backfiring Function Points

The term "backfiring" has been used for many years by software measurement and metrics practitioners to describe the process of converting between physical metrics (i.e., lines-of-code) and logical (i.e., functional) metrics. Backfiring function points is often used as a surrogate for logical sizing, despite the fact that it is (on average) significantly less accurate than normal function point counting.

When using backfiring to estimate the number of function points represented by an application code base, a few key facts should be considered.

- The correlation between the level of a language and development productivity is non-linear. For most large software projects, coding amounts to only about 30 percent of the effort. Assume a program is written in a language that is twice the level of a similar program, for instance level 6 versus level 3. In this example, the coding effort might be reduced by 50 percent. But the total project might be improved by only 15 percent, since coding only comprised 30 percent of the original effort. Double the level of the language again to a level 12. That will only give an additional 7.5 percent net savings. Once again, coding is halved. But coding is not a major factor for very high level languages.
- More accurate economic productivity rates can be gained by examining the average monthly Function Point production rates associated with various language levels.
- The accuracy of backfiring is not claimed to be as high as standard function point counts. Inaccuracy stems from a number of sources (see Table 1 for details):

- The starting point is physical (not logical) lines of code (see Table 2)
- Mixed-language applications
- Variances resulting from individual programming styles
- Variances in cyclomatic complexity[1]
- Ambiguity in interpretation of logical statements in non-linear (i.e., graphical) code environments

The accuracy of backfiring from logical statements is generally plus or minus 25% or more[2], while normal counting by certified function point counters has been measured to range by about plus or minus 10% in a study commissioned by IFPUG and performed by Dr. Chris Kemerer[3] when he was at MIT.[4]

Table 1: Impact of Possible Source Code Size Counting Variations[5]

Code Counting Measurement Variation	Range of Possible Impact
Including or excluding existing code from prior releases	1000%
Including or excluding meta-language statements.	300%
Including or excluding copybook statements	200%
Including or excluding macro expansions.	150%
Including or excluding code from packages	125%
Including or excluding reused code.	125%
Including or excluding data definition statements.	35%
Including or excluding "dead code" that no longer executes	30%
Including or excluding scaffold or temporary code.	30%
Including or excluding code in test cases.	20%
Including or excluding commentary statements.	15%
Including or excluding blank lines between paragraphs.	12%
Including or excluding job control language (JCL).	5%

[1] Introduced by Thomas McCabe in 1976, *cyclomatic complexity* measures the number of linearly-independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs.

[2] The accuracy level depends on several factors, including: a) the level of cyclomatic structure of the language itself (both nominal and as practiced specifically by the code writer); b) the nominal "generation" of the language; c) whether the counted statements include comments or not; d) whether the quantity counted includes physical lines or logical/functional statements; e) for graphical languages, the method of interpretation of a "logical statement."

[3] http://www.pitt.edu/~ckemerer/

[4] Kemerer, Chris F.: Reliability of Function Point Measurement: A Field Experiment; MIT Sloan School Working Paper 3192-90-MSA; January 1991.

[5] Jones, C. (1997, July 23). Can there be a truce in the war of the metrics?