Don’t take anything for granted when using LOC Related Metrics

Steve Kan, STSM Technical Manager, iSeries Software Quality

April 2005

The **lines of code (LOC)** related metrics are anything but simple. The major problem comes from the ambiguity of the **operational definitions**, i.e., the actual counting methods. In the early days of Assembler programming, in which one physical line was the same as one instruction, the **LOC definition** was clearly understood.

With the availability of high-level languages the one-to-one correspondence has broken down. Differences between **physical lines** and **instruction statements (or logical lines of code)** and **differences among languages** contribute to the huge variations existing with regard to LOC counting.

Even within the same language, the methods and algorithms used by different counting tools can cause significant differences in the final counts. Jones (1986) has described several **variations in counting lines of code:**

- Count only executable lines.
- Count executable lines plus data definitions.
- Count executable lines, data definitions, and comments.
- Count executable lines, data definitions, comments, and job control language.
- Count lines as physical lines on an input screen.
- Count lines as terminated by logical delimiters.

According to Capers Jones, the difference between **physical lines** and **instruction statements** can be as large as 500%; and the average difference is about 200%. Given the availability of high level programming languages, if a software product is comprised of different languages, the **variations in LOC counting** can be even larger.

Given such variation and the lack of industrial standards for LOC counting, the **measurement reliability** of LOC data is clearly unacceptable for any meaningful use. When any LOC data is presented, the method for LOC counting should be described. Ideally, industry-wide standards should include the **conversion ratios** from high-level language to assembler. So far, very little research on this topic has been published. The conversion ratios published by Jones in 1986 are still the only set that is well known in the industry.

Not only **LOC metrics** receive a poor grade in **measurement reliability**, it is also problematic with respect to **metric validity**. What do we intend the **LOC metric** to measure? Size, functionality, or productivity? For productivity studies, the problems with using LOC are more severe. A basic problem is that the amount of LOC in software is usually negatively correlated with **design efficiency**.

The purpose of software is to provide certain functionality for solving some business or scientific problems. Efficiency design means providing the functionality with lower implementation effort and fewer LOC. Therefore using LOC (size) to measure functionality is only a surrogate measure.

Recently a software development executive in STG talked about his experience directly related to the **size versus functionality** issue. When he started his IBM career, he improved the design of a complex software module and as a result the size of the module reduced from 5,000 LOC to 2,500 LOC, and defects were all eliminated. However his productivity (in the status report) was -2500 LOC! Hence, using LOC data to measure **software productivity** is like using the weight of an airplane to measure its speed and capability.

The LOC results are so misleading in productivity studies that Jones has stated that using lines of code for productivity studies involving multiple languages and full life cycle activities should be viewed as professional malpractice (2000).

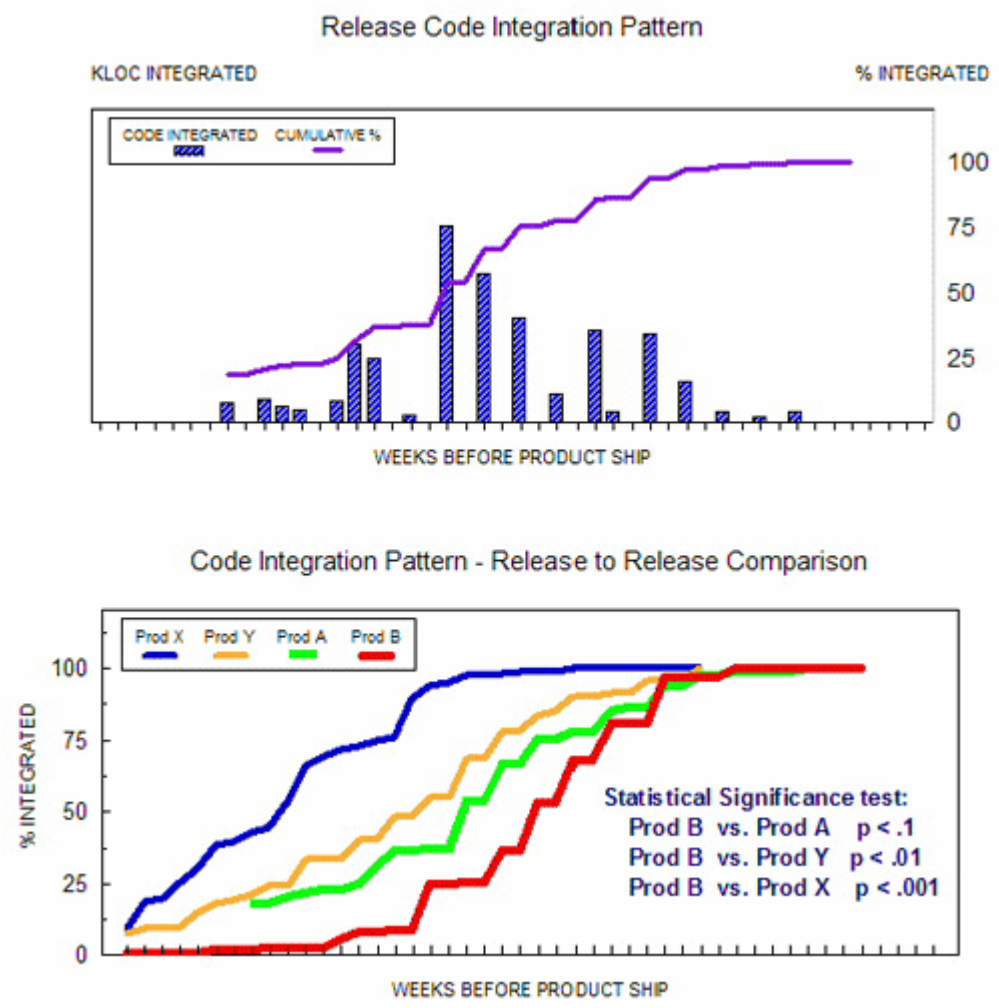
When used for defect rate calculations, LOC related metrics can only provide an internal view; it does not provide a customer view of quality. From the customer's perspective, the amount of LOC is irrelevant, what matters is the total impact of problems.

Given the many problems with LOC related metrics, does it mean they are totally useless? If use with care – i.e., with clear **operational definitions** in place, with comparison to one's history only, understanding the caveats when interpreting the results, etc – we can still make LOC metrics work for us.

An example of a **useful LOC metric** for in-process quality and project management is the **code integration pattern** (into system library) over time. Among the major phases of any development process, code development is perhaps the most fundamental activity. Completion of coding and unit test, and therefore the integration of code into the system library (to be ready for formal testing) is perhaps the most concrete intermediate deliverable of a project.

Code completion by no means implies that the project is near completion. The earlier the code is complete and integrated into the system library relative to the project completion date, the better chance it will be adequately tested. In most software development projects, code completion usually spreads over time. The **pattern of code integration over time**, relative to the product delivery date, therefore, is a crucial variable for schedule and quality management.

The Figure below shows the **code integration pattern** of a systems-software relative to the product ship date. The vertical bars are the amount of code completed and integrated into the system library over time. Their values are expressed via the first y-axis. The S curve is the cumulative percentage of code integrated over time, and is referenced by the second y-axis.



There are at least three ways of meaningful use of this **code integration metric**:

- Establish a **planned code integration curve** as early as possible in the development cycle. Normally at the beginning of the project, team leaders are able to put down target dates for key activities of the line items they are responsible for, such as design complete, design review complete, code complete and integration, and testing start and completion. Use the planned curve to track the actual progress of code completion and integration.
- As early as a plan curve is available, study its pattern and take early actions to improve the pattern. For example, the S curve with a concave pattern at the top half of the curve, such as the example shown, is a normal and healthy pattern. If the S curve shows some steep step increases at the right side (see examples in the bottom chart in the figure), then it is a back-end loaded integration pattern, which may pose risks to testing schedules, defect arrival pattern during testing, and even schedules and quality outcome of the project.
- Perform **project to project or release to release comparison** when baselines are available. Assess the feasibility of the current project based on the comparison results and take early actions. For example, break out large LOC features into smaller ones and integrate the small chunks when ready; risk

mitigation plans for features that are integrated late; use of control limits to determine the process capability of the development team, and so forth.

Based on our experiences, **code integration pattern** is a simple and yet very powerful metric.