

IT METRICS STRATEGIES

Helping Management Measure Software and Processes and their Business Value



Using “Backfiring” to Accurately Size Software: More Wishful Thinking Than Science?

by Carol Dekkers and Ian Gunter

Functional size measurement is a fairly recent concept to be embraced by the IT industry. But, increasingly, the method of function point (FP) analysis, as maintained by the International Function Point Users Group (IFPUG), is establishing a position within the field of software measurement.

At the same time, in an industry dominated by engineers, computer scientists, and math majors, it is easy to understand why physical measures of software size, such as source lines of code (SLOC), continue in common usage. To make the transition between SLOC and FPs easier, a method called “backfiring” was developed to calculate FPs by taking the SLOC count and dividing it by a static factor based on the dominant software programming language.

In this article we will discuss the basis for the two measures, FPs and SLOC, highlighting their differences and distinct advantages. We will also highlight why “backfiring” can lead to gross inaccuracies when sizing software.

Continued on page 2.

Determining Your Own Function Point and Lines of Code Proportions — Three Things to Watch Out For

by Michael Mah

Understanding the inventory of IT applications is considered to be of high strategic value for organizations interested in IT benchmarking. For some, the purpose might be to better estimate future projects. Others might be embarking on a process improvement strategy and want to know how various IT groups perform relative to one another. A third group could include IT organizations involved in outsourcing or partnering, and they might need this information to quantify the respective roles and productivity commitments.

In any case, you might find yourself wanting to understand the relationship in lines of code per function point (FP) for your applications. Some might want to know these proportions for other

Continued on page 8.

executive summary

In recent issues of *ITMS*, we looked at managing the size of IT applications. This is critical whether you're building IT applications inhouse or contracting an outsourcer or IT partner for applications development and maintenance. In both cases, mismanaging the scope and size of commitments can have dire results. Overpromising can result in the project overruns and slippages that are at epidemic levels in our industry.

Two common metrics for software size are source lines of code and function points. But by no means are these the only units of size; others include objects, modules, programs, components, and frames. Obviously there are scaling relationships between these abstractions. How can you translate one from the other to understand the proportional aspects of one metric to the next? If you knew the size of a major application in source lines of code, is there a way to equate the size using a metric like function points for people who speak that language?

For example, if you knew the square footage of a house, and the amount of building materials used, could you approximate the number of rooms (if you knew the scaling relationship between types of rooms and typical square feet per room for that design)?

Addressing this topic are Carol Dekkers and Ian Gunter. Their article tackles the subject of deriving function points from code, which is known as backfiring. Why is backfiring used by some organizations? In addition to the examples cited above, some advocates of function points desire a “common currency” of size, irrespective of the code size, which can vary due to different software development languages. But backfiring has its problems, and you should read this article to understand some of the issues you'll need to consider.

Also included in this issue is a review of an important new IT book, *Measuring the Software Process*, by Anita Carleton and William Florac. Jim Heires, a front-lines practitioner of metrics, benchmarking, and process improvement, takes an in-depth look at the advice and knowledge contained in this work.

Michael Mah, Editor

november 2000 vol. VI, no. 11

Using “Backfiring” to Accurately Size Software: More Wishful Thinking Than Science?	1
Determining Your Own Function Point and Lines of Code Proportions — Three Things to Watch Out For.	1
<i>Measuring the Software Process</i>	9

Continued from page 1.

The Requirement for Different Software Size Measures

Functional size measurement, the act of measuring the size of software based on its logical user functions, is a somewhat new concept in the IT industry. Although it was first introduced more than 20 years ago, only in the past 5 years has the method stabilized to such an extent that more than 50 commercial estimating tools and multiple industry databases now include FPs as an input parameter. Additionally, the December 1998 *Scientific American* featured an article on software sizing that prominently profiled FP-based metrics.

Today, the FP analysis method as maintained by IFPUG is making inroads for software sizing within the US Department of Defense and the Software Engineering Institute's Capability Maturity Model Integration projects. With the inclusion of FP-based metrics in major outsourcing contracts, FPs are becoming acknowledged as a useful size measure alongside the traditional physical measures of size, such as SLOC.

Why is software size important to the IT industry? Size underpins many key project decisions — from work effort and cost estimating to scheduling. As one of the key input measures for predicting project costs, it is vitally important that the anticipated project size be reliable and accurate. However, in our haste to arrive at “quick and dirty” estimates, the importance of an accurate size measure often goes unrecognized and may even be overlooked by estimators demanding fast answers. This has created interest in techniques for generating derived measures of functional software size from other measures using a shortcut approach. In particular, to make the transition between SLOC and FPs easier, a method called backfiring was developed that derives FPs by simply dividing SLOC by a static factor based on the

dominant software development language. The promised benefits of such backfiring techniques are speed and ease of derivation over the manual process of FP counting. But using shortcuts without a thorough understanding of the limitations often leads to inferior results.

In addition, when a “conversion” is attempted between two measures such as SLOC and FPs, the results can appear to be sound, yet must be challenged in terms of their accuracy and applicability. This is especially true when backfired FPs are used as the basis of corporate decisionmaking.

What Are Source Lines of Code and Function Points?

To understand the problem and the associated issues involved with deriving one software size measure (FP) from another (SLOC), it is important to understand the differences between the measures themselves. SLOC and FPs measure two distinct and different dimensions of software size: SLOC measure the physical, implemented size of source instructions for software; FPs measure the functional size based on logical software functions. As such, the two measures are not directly interchangeable — they measure distinct dimensions of software size. Consequently, each has its own special features, advantages, and disadvantages as a measure of software size. Does this mean that one cannot approximate or derive one measure from another? This subject is the essence of this article, particularly in relation to the backfiring of SLOC into FPs.

A construction analogy can be useful for understanding the differences: FPs represent the functional count of the logical software requirements at a certain level of abstraction in a way similar to the square feet of a building's floor plan. SLOC, on the other hand, represent the physical size of software code components similar to how the number of

Editorial Office: Clocktower Business Park, 75 South Church Street, Suite 600, Pittsfield, MA 01201, USA. Tel: +1 413 499 0988; Fax: +1 413 447 7322; E-mail: michaelm@qsm.com.

Circulation Office: *IT Metrics Strategies*® is published 12 times a year by Cutter Information Corp., 37 Broadway, Suite 1, Arlington, MA 02474-5552, USA. For information, contact Megan Niels, Tel: +1 781 641 5118 or, within North America, +1 800 964 5118, Fax: +1 781 648 1950 or, within North America, +1 800 888 1816, E-mail: info@cutter.com, Web site: www.cutter.com/consortium/.

Editor: Michael Mah. Publisher: Karen Fine Coburn. Group Publisher: Anne Mullaney, Tel: +1 781 641 5101, E-mail: amullaney@cutter.com. Production Editor: Lori Goldstein, +1 781 641 5112. Subscriptions: \$485 per year; \$545 outside North America. ©2000 by Cutter Information Corp. ISSN 1080-8647. All rights reserved. Unauthorized reproduction in any form, including photocopying, faxing, and image scanning, is against the law. Reprints, bulk purchases, past issues, and multiple subscription and site license rates are available on request.

wood two by fours used in construction quantifies a building's physical size. Some size dimensions can be translated into other dimensions, such as square feet per room, however, problems can occur if care is not taken to understand what the translated measures mean.

A further examination of SLOC and FPs follows, which will lead into our discussion of backfiring.

Key Features of SLOC

SLOC are a measure of the physical size of software. To measure SLOC, you count the number of noncomment, self-contained lines of code contained in the software, regardless of whether it is batch or online code. Often the total SLOC for a piece of software is subdivided by programming language for ease in potentially backfiring the figure into FPs. Compiled code and other variations of the source code are not usually counted, but rules surrounding SLOC counting do not usually address whether job control language, hardware-specific, and other variations of batch submission code are to be counted. In short, SLOC:

- Are a physical size measure of software based on a count of its source code implementation
- Are simple and easily understood measures in common usage
- Provide a "builder" perspective on the software size, based on how programmers view software (similar to how a carpenter would view the size of a house based on the number of boards used in construction)
- Are easy, inexpensive to count, and automatable, but there are no industry-wide standard counting rules in place¹
- Can be estimated at the coding phase or later (however, actual SLOC figures are not available until later in the development lifecycle)

- Are meaningful only for comparisons of software developed in the same language and using similar coding conventions
- Are a primary sizing input for many conventional software costing models, including COCOMO, COCOMOII, SLIM-Estimate, Price/S, Estimate Professional, and KnowledgePlan
- Are appropriate for software sizing when a physical measure is needed, when large amounts of SLOC data are available, or when measuring maintenance of applications that use the same or similar programming languages
- Are dependent on the programming language and physical implementation
- Vary with the skill and programming style of the individual programmers
- Treat all lines of code equally (i.e., given the same weight)

Key Features of FPs

The term function points refers to the unit of measure that is used to quantify the logical, functional size of software, independent of its development or implementation technology. The FP measure is backed by a rigorous method of counting rules maintained by IFPUG, which produces the *Function Point Counting Practices Manual* (CPM), currently in release 4.1.² In short, FPs:

- Measure the functional size of the software, from a user perspective (the size of what is going to be built)
- Are conceptually less easy to understand than SLOC, especially from a programmer's or developer's point of view
- Provide a software customer or user perspective on the functional software size (like the square feet of rooms in a house)
- Are relatively expensive to measure compared to SLOC; to count FPs, you need a skilled counter trained in the FP counting rules

¹Editor's note: However, there are guidelines from the Carnegie Mellon Software Engineering Institute (SEI) under definitions of the Four Core Metrics, known as the SEI Minimum Data Set.

²The *IFPUG Counting Practices Manual*, CPM 4.1 (1999), can be obtained directly from IFPUG at www.ifpug.org or by calling the administrative office at +1 609 799 4900.

- Are subject to standard, well-defined counting rules as published by IFPUG
- Can be estimated early in the project/development lifecycle and counted once the requirements are articulated
- Are a primary input for many software costing and work-effort estimating models
- Are independent of the programming language and physical implementation — a common currency of software measurement
- Give different weight to different types of logical user functions

The key things to remember are that the two measures represent different dimensional attributes of the software, are used for different purposes, and are not interchangeable — much like height and weight. When we talk about FPs and SLOC, we're not talking about feet and meters; we're talking about number of boards and square feet.

Why Derive One Software Sizing Measure from Another?

There are many situations where only one directly counted measure of software size, typically SLOC, may be available. The most common reasons for this include:

- SLOC counts are readily available using automated SLOC counting software
- Certified FP specialists are not readily available and/or are too expensive
- Lack of time and budget to hand count the FPs
- Lack of understanding about SLOC counts
- No perceived, compelling reason to hand count FPs
- Little or no user or requirements documentation from which to generate FP counts
- Out-of-date user manuals or lack of knowledgeable resources from which to glean functional requirements

However, the existence of SLOC alone may not satisfy estimators requiring additional size representation. In such circumstances there may be compelling reasons for wanting to derive FPs from SLOC. To make such a transition between SLOC and FPs easier, backfiring was developed. Backfiring calculates FPs by dividing SLOC by a static factor (or multiplying by a fraction) based on the dominant software development language.

For instance, backfiring is often used when only the SLOC figures are available and FP analysis is not possible or practical. Because the FP measure is independent of the language used to implement the software, FPs allow comparisons across diverse systems and offer a normalized currency of size. Backfiring promises a quick and inexpensive means of deriving a FP figure.

Conversely, the backfiring technique is also used for those times when SLOC figures are needed at the start of a project for input into those cost-estimating models that require lines-of-code counts. At the time of estimating, only SLOC estimates are available, and these are often based on questionable, or even unstated, assumptions. The attraction of an SLOC figure derived from an available FP figure is that its derivation appears more methodical and traceable, thus providing figures more credible than other estimates.

The Mathematical Principles of the Backfiring Method

Establishing a mathematical form of the relationship between the FPs' measure of software size and the SLOC measure for software implemented in the *same* language is not difficult. The Software Productivity Research, Inc. (SPR) Web site mentions that the constants in its programming languages table must be based on a sample size of at least 10 similar language projects, and that the data is continually being refreshed.³

The major concept behind backfiring is the assumption that there is a directly proportional (linear) relationship that can be established for a given programming language. Such a linear relationship is given by: $FP = k * SLOC$ (where SLOC is a count

³SPR's Web site is www.spr.com. Refer to SPR's Resources page for further details about programming languages translation table.

of the logical lines of code in the subject software, and k is a constant based on the programming language).

With a set of several data pairs of corresponding FP and SLOC measures for software developed in the same language, the value of the constant translation factor, k , can be calculated via a “least squares best fit” approach. Once the value of k is established, the equation can be applied to derive an estimate of software size measured in FPs from a known SLOC figure; that is, to backfire. However, the calculation of k is not sufficient. Some measure of the goodness of fit to the data needs to be considered, and, in the case of backfiring, confidence limits for the value of k should always be stated.

Similarly it is possible to calculate a corresponding equation to derive SLOC from a known FPs figure: $SLOC = k' * FP$.

Capers Jones, chairman and founder of SPR, is an early advocate of the backfiring method and has produced a programming languages table that contains the k values as determined by SPR research over the years. In the preamble about how to use backfiring and the cautions associated with it, Jones discusses how the SLOC count is intended to be the count of the *logical* lines of code, (i.e., non-commented, noncompiled program code), not the physical lines of code. (Also see “Sizing Your Promises and Expectations” by Michael Mah in the September 2000 *ITMS* for a table of k values.)

Inherent Uncertainties with the Backfiring Method

The uncertainties attached to the relationship between FPs and SLOC must be understood if backfiring is to be applied effectively and meet the intention for which it was designed. Ultimately, the resultant size (in FP if using the constant to derive FP from SLOC; or in SLOC if using the inverse of the constant, k , to derive SLOC given FP) depends on the data that has been used to calculate the value of the translation factor, k . The questions that arise in relation to the constant k surround the following issues:

- Software that the data relates to (type and size of application, development environment). For example, is the software to

which the backfiring method is being applied of the same size/type and being developed in a similar environment? If the answer is no, then the constant k may result in flawed data.

- Errors in the data. Even with counting rules at +/- approximately 10% for FPs (accuracy according to Chris Kemerer of MIT in one of his FP analysis studies circa 1993) and with SLOC figures subject to no counting rules, is all of the data used to derive the k constant consistent?
- The goodness of fit to the data. What are the confidence limits for FP and code size? Is there any flexibility to adjust the constant k for changes in project conditions besides purely programming language level?

Of course, the relationships and usefulness of backfiring all depend on the homogeneity of the data. But in practice, the correlation between software size measured in SLOC versus FPs is far from perfect. So, given that the SLOC to FP translation is flawed on an individual project-by-project basis, are there, perhaps, some usage situations where backfiring can be used for success? Number of boards translated into square feet of building can work well only with good data and a translation constant derived from several projects with similar attributes.

Acquiring a Suitable Model/Language Translation Factor

There are two options for acquiring the translation factor: calculate your own or use a reliable factor that someone else has derived. Conveniently, tables of language translation factors, such as those from SPR and QSM, are available. Alternatively, an organization may have sufficient software size data of its own to consider calculating its own value for a translation factor. Both approaches have their limitations and risks.

Using a published table of translation factors is attractive; after all, all of the statistical analysis has been done, hasn't it? Well, maybe. Usually there are important omissions from some published tables. Some sources don't tell you where the translation factor has come from — what data was used to calculate it — and you don't know the confidence limits that apply to it.

If a sufficient supply of relevant corresponding pairs of SLOC and FP measures is available, then it may be better for an organization to use its own data to calculate an appropriate translation factor. This has the advantage that the relevance of the model is clear, and the goodness of fit of the model can be known.

Another benefit of using local data is that it can be used very effectively to test the validity of translation factors from published language tables.

Software Project Attributes Ignored by Backfiring

For all its mathematical rigor, backfiring can often disregard the nature of software projects. The method takes no account of the process that translates the functional requirement into the software implementation, other than the language used for coding the software. Backfiring relies on a linear model of SLOC to FP (or vice versa), where the constant is based solely on programming language. The relationship between SLOC (a physical measure) and FPs (a logical, functional measure) is much more complex than backfiring gives credit. As such, a single dimensional model such as backfiring can be grossly inaccurate.

Also, there is a wide variation in implementation styles that can affect the SLOC to FP ratio significantly. For example, backfiring will make a product design that is implemented via verbose coding practices appear to be a functionally rich product. What's more, it will conceal instances where the application's product design extends beyond its functional requirements. The larger the number of SLOC program statements, the larger the resultant FP count — and this may not actually be the situation. While it is easily understood that the larger the project, the larger the source code usually is, the relationship between FPs and SLOC is not as simple as it first appears.

A frequent complication when choosing to use the backfiring technique is that many applications are developed using more than one language. Unless there is a well-understood boundary between the separate language elements in the software implementation, the use of backfiring

in these situations must involve further approximations.

The simple truth about FPs backfired from SLOC is that the technique produces a FP figure in name only, reflecting a particular programming language only, and represents the software implementation rather than the functional requirement.

Misconceptions and Dangers

We have explored the fact that the relationship between SLOC and FPs is less than perfect, and when disparate sets of data are involved, the figures can be even more inaccurate. However, despite this, some factions of our software estimating industry tend to talk up the validity of backfiring.

Most software cost-estimating tools require SLOC figures as a primary input. But many of these tools have been adapted to accept an FP figure and derive the necessary SLOC figure from it. While the vendors of cost-estimating tools are keen to promote this feature as a selling point for their applications, the accuracy of the approach is usually played down.

In environments with an immature metrics program, the inclination to use backfiring to derive FP figures is strong. And often such backfired figures are used for comparisons with industry averages, as in benchmarking. However, the results are likely to be misleading and unfair. In this situation, it is sometimes the vendors of such services who are most likely to play down the risks in backfiring.

The use of backfired FP figures in productivity calculations is contentious, especially at the client/vendor interface. For example, in benchmarking an outsourced development and support service, the use of backfired figures is likely to be the basis of many arguments between client and vendor.

One question remains: if the backfiring method to derive FPs from SLOC (or vice versa) is so flawed, why are major, multimillion-dollar outsourcing contracts using it to establish the size of their outsourced portfolio? The answer lies in the fact that some data (even imperfect) is better than no data. Also, there are some

portfolio-wide situations where backfiring can be used to degrees of success.

Such instances include corporate-wide measurement initiatives where the overall portfolio SLOC counts are available, but the organizations involved do not have the time, energy, or budget to properly fund a portfolio FP sizing effort. This may apply in the same way that the number of boards in a house can likely be related to the square footage size of the house, where the relationship of square feet per room is fairly static. As such, a linear relationship between square feet per room can work, given a large enough sample size and a relatively homogeneous environment.

This is much the same situation when a portfolio is brought forward to be sized using the backfiring technique. But in the same way that a luxurious family room might skew the relationship constant because it is so different from the other rooms in a house, software applications that are outside the “norm” of the types of applications developed in a single language will skew the backfiring constant.

For a large home, the relationship between the number of boards and the square footage can provide a fairly consistent estimate of square feet. For a large number of similar software applications, the relationship between FPs and SLOC by language also works fairly well. In both cases, the law of large numbers (the more data points in your sample size, the better the average results) usually provides a good overall portfolio approximation of size.

The problems emerge when you try to use a single data point with the translation constant to derive FP or SLOC from each other. This is similar to saying that the square footage in a house derived from the number of boards will be accurate based on limited data relating the two measures. When a house (or a software application for that matter) does not fit the norm of the average in the data sample, there will obviously be variations in the accuracy of the result. A data entry system with fewer lines of code and a high language level will result in a lower number of FP than one with many lines of code used to derive mathematical functions for use in reporting, even considering the

effects of the language level. In these situations, the SLOC to FP ratio does not reflect the differences in logical functionality and will arrive at an inaccurate backfired FP count.⁴

Conclusion

At best, published tables of language translation factors should be viewed as no more than indicative, particularly when they are not qualified with details of their source/applicability and claimed accuracy. Like buying food without an expiration date, it may be good, but the risks are unknown.

Software measurements are needed as the basis for project decisionmaking. And although the quality of the measurement needs to be no better than the decision that is to be based on it, it has to be acknowledged that bad information leads to bad decisions. For every measurement activity there needs to be a cost/benefit tradeoff. However, it is important to understand the uncertainties and risks associated with any measurement, and in the case of software size measures derived from backfiring, there are significant uncertainties and risks. When backfiring, you may get a quick, cheap measure — but also a crude, risky measure.

The bottom line is that the use of backfiring is about on par with estimates produced on the back of an envelope, especially for single projects. The technique may be suitable for rough-and-ready calculations when the law of large numbers (and the large sample size) will even out the discrepancies between types of applications on a portfolio-wide basis. However, backfiring, due to its lack of rigor in many cases is simply not good enough as a basis for important project decisions that require an accurate functional size measurement.

About the Authors

Carol Dekkers is president of Quality Plus Technologies, Inc., a US-based, leading

⁴A. Lubeshevsky of Lucent Technologies reported in a paper several years ago that the AT&T actual field tests of the backfiring technique resulted in a variation of up to 400% between the hand-counted FP and the backfired FP number. Similar results have been experienced by the authors who have seen ratios of backfired FP figures to actual FP counts that have exceeded several hundred percent.

software measurement and process improvement consultancy. Dekkers has been involved in FP-based measurement and outsourcing since 1994, and she is a frequent writer, presenter, and trainer at major software quality conferences worldwide. Dekkers was recently named one of the “21 New Faces of Quality for the 21st Century” by the American Society for Quality. She can be reached by e-mail at dekkers@qualityplustech.com.

Ian Gunter is president of Numerical Science, Inc., a London-based consultancy specializing in software process quality, outsourcing, and process improvement initiatives. He is often retained by major businesses who have outsourced their systems development and/or maintenance activities and who are challenged to make the measurement provisions of their agreements work. Gunter can be reached by e-mail at igg@numerical-science.com.

Determining Your Own Function Point and Lines of Code Proportions

Continued from page 1.

components (or levels of abstraction), such as the typical amount of code per C++ object, modules, elements, etc. The concept behind finding these proportions (or “gearing factors” in some circles) is similar, but there are certain aspects of FPs that warrant special attention.

Here are three factors to pay special attention to when determining your numbers.

1. Watch Out for the Wrong Fit.

Some organizations build applications that do not fit the classic FP paradigm of create, read, update, and delete against an underlying database. Examples include Web-based applications, e-commerce applications (transaction-intensive), algorithm-heavy applications, or those with lots of error checking, security, and/or system diagnostic attributes.

If you’re building systems like these, and not your classic legacy COBOL mainframe applications, your FP counts might be low since the five function-providing elements don’t fit as well in your world. (Nothing against FPs — just an inadequate fit.)

Meanwhile, there might be a great deal of complex code required to accomplish these tasks. You might then be scratching your head when you find high lines of code per FP. It may not be that you’ve got programmers writing “verbose” code in all their spare time. It may be that this code is doing a task not defined by the five function-providing elements.

In these cases, viewing your system inventory from other levels of abstraction (instead of just FPs) might be valuable to you. Number of use cases, requirements, business processes, work requests, and others might be practical units of measure from an end-user perspective in your organization.

2. Be Consistent in Code and Function Point Counting Rules Where Possible.

Some organizations get unusual results when their counting conventions are inconsistent, thus introducing components for error. Make sure that for code, you at least use consistent rules from the Carnegie Mellon Software Engineering Institute. Don’t count physical lines in one application, then logical source statements in another. If you do, try and correct the counts by an adjustment factor. For example, one organization found that it was common for physical files to have about 10% additional lines for comments and blanks. So they adjusted their source count by that amount when they had to.

Count new and changed code, as well as new and changed FPs. If you include all the reused or unmodified code and FPs from previous releases, you’ll get odd results.

Also, watch out for counts of deleted FPs. One organization in recent memory counted buckets full of deleted FPs as part of its delivered FPs. When it also counted the code that existed in its configuration management libraries, things looked strange indeed as its source libraries didn’t have

phantom code counts from programs that were vaporized (when FPs were deleted).

3. Changed Function Points Are Not All Alike.

Even when you count just the new and changed FPs, there is another factor to consider. You can make a simple change to an FP by rewriting a small amount of the program where that FP resides. At the other end of the complexity spectrum, you might have to rewrite almost the whole thing.

Rolling this up, you might have a project with a lot of changed FPs that were simple code changes. Low lines of code per FP. Another project might have been an ugly bear with high amounts of rewritten code to make difficult changes. Higher lines of code per FP. That's life.

You should be careful about falsely concluding that designers and software engineers on the latter project were once again inefficient programmers writing fatty, verbose code in their spare time.

Understand that the least squares best fit when determining your proportionality or gearing factors (constant k in the Dekkers and Gunters article; see page 1) might have logical underlying components that explain the inherent variability. Not all backfiring constants are different for bad reasons. They might be different for perfectly logical reasons. You have to understand the "whys".

Summing It Up

Being able to transition from one size metric to another is very useful, as long as you understand the purposefulness of such an endeavor. Speaking languages other than English for example, helps you communicate with others in a meaningful way that bridges gaps in understanding between the diverse groups of people involved in IT. If sharing a common language and using a normalized currency (like the euro) helps get you there, then go for it.

But the last thing you want to do is introduce yet another source of misunderstanding among your colleagues and business partners. If organizations don't avoid potentially harmful source lines of code to FP translation mistakes, then you might wind up creating exactly what you sought to avoid — conflict over substantive issues that are difficult to explain or understand.

To make these translations meaningful, you can start with the "currency converters" found from metrics researchers like Software Productivity Research, QSM, and others. But by no means should you depend entirely on other people's metrics.

It is always a good idea to find out what's going on in your world and then use the scaling or gearing tables to validate or cross-check the results from your data to help you determine their legitimacy. And if your numbers are not an exact match, that's okay. There is a normal variation in these things.



Measuring the Software Process

by James T. Heires

"Measurement is a sampling process designed to tell us something about the universe in which we live that will enable us to predict the future in terms of the past through the establishment of principles or natural laws."

— Walter Shewhart

Overview

Measuring the Software Process by William Florac and Anita Carleton (Addison-Wesley, 1999) is a self-contained statistical process control (SPC) foundation in the context of software process improvement (SPI). SPC was pioneered by Walter Shewhart in a telephone-manufacturing environment in the 1930s. Florac and Carleton apply this

early industrial wisdom and some previous work at the Software Engineering Institute (SEI) to a modern software development environment.

The emphasis is primarily on the use of analytical studies (predicting future outcomes) using the control chart as the primary instrument. Only brief treatment of the use of enumerative studies (evaluating current situations), however, is presented in this problem domain. Time-honored tools, such as the Pareto chart, cause-and-effect diagram, and histogram, are given even less attention.

The first half of the book is a slow read but directs attention to critical topics such as planning, managing, and measuring. The subject matter is somewhat to blame here, but the authors could have added more case studies and examples to make their message more interesting and useful. The second half of the book is much livelier, due to the generous use of examples from industry and explanations about how to benefit from the results.

The authors adequately cover the material as it applies to software development, but the reader is cautioned that many statistical fundamentals are omitted from this work. In order to put these ideas into practice, one should seek further instruction or consult a statistician.

A few annoying typographical and redundancy errors are present as well. Most bothersome about the book, however, is that the authors do not seem to be convinced that statistical process control for process improvement actually works! This is somewhat alarming, given the long, successful history of SPC in other industries. All in all, though, this book desperately needs to be read by anyone wishing to improve a software development process.

Motivation for Software Process Management

The book opens with a discussion about the motivation for software process management. Software process management is made up of the following activities:

- Defining the process

- Measuring the process
- Controlling the process
- Improving the process

This chapter details each of these activities and lays a strong foundation upon which the remainder of the book is built. The flow chart shown in Figure 1 is used several times throughout the book to illustrate the activities and decisions prescribed by the authors to carry out quantitative SPI.

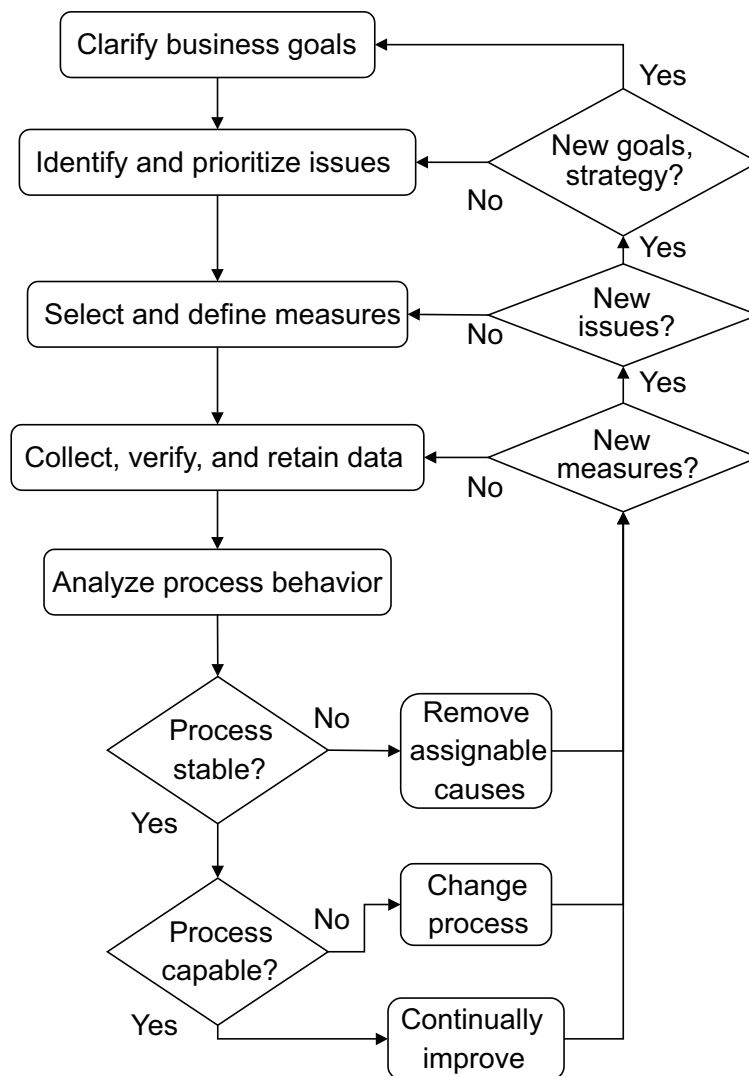
Using this flow chart encourages the reader to implement the activities associated with SPI in the correct sequence. Many measurement programs fatally attempt to get to third base without stepping on first. That is to say, they start collecting measurements before clarifying business goals or identifying issues. This mistake can cause tremendous delays in deriving any value from the measurement program.

To provide the best guidance to the measurement program, business goals should be identified in terms of project characteristics (e.g., schedule, cost, quality, and functionality). Goals expressed in terms of sales volume or return on assets, for example, hold little meaning to software developers. In large and small organizations alike, it is sometimes difficult for developers to see the effect their work has upon sales volume. It is for this reason that business goals need to be translated into “project speak” before an SPI program can contribute to business goals.

“There’s a monster living under my bed, whispering in my ear.”¹

The aggressive senior manager might say that faster, better, and cheaper are simultaneous business goals. All too often, however, it is the same manager who will forsake proven tools and processes to favor the political priorities for pet projects. The “I want it all” approach is a noble long-term strategy, however, because it implies efficiency gains. Nevertheless, practitioners need to be able to prioritize the goals of their projects today, which have hard schedule constraints and resource limitations. This is where the

¹“Put Your Lights On.” Carlos Santana, from the album *Supernatural*. Arista Records, 1999.



©1999 Addison-Wesley. Reprinted with permission.

Figure 1 — Process measurement improvement.

customer can help. If the customer is willing to pay more for a more reliable system, the development team can use this information to plan and manage its project to an appropriate set of measures.

Planning for Measurement

Failing to plan is like planning to fail.

Software engineering processes are designed to help meet business goals. Likewise, improvements to the software engineering process should be designed to address business goals. Proper planning ensures that

business goals are addressed. For example, a process-oriented business goal might be to reduce the average cycle time by 15%, as compared to their own measured 1999 average of 8.3 months.

The authors suggest using a five-step approach to identify process-related issues, which threaten the achievement of business goals. Some of the more common process issues involve quality, schedule, and cost. Next, questions about these issues should be formulated to focus attention on each critical process in order to maintain alignment with business goals. An example question

relating to the goal above might be: what is the current average cycle time since 1999?

The third and final planning activity is to select and define measures. First select measures, then define them. Selection of appropriate measures is based on the goals and questions formulated above. Reasonable measures might be cycle time and project completion date. By measuring cycle time of projects since 1999, the achievement of the business goal can be determined.

This three-step process forms the essence of Victor Basili's Goal/Question/Metric method for establishing a measurement program to achieve business goals.²

Measuring the Software Process offers a guideline to help select and check the appropriateness of process-oriented measures. Process measures are different from product measures in that they describe how the process performs. Examples of process measures include effort expended, task duration, or developer experience level. Product measures, on the other hand, describe the software under construction and include the code size, system reliability, or execution speed. Defining process measures is often overlooked but is vital to the success of any measurement program. Clear definitions facilitate communication and provide consistency when collecting, analyzing, and reporting measurements. Operational definitions are those that address communication, repeatability, and traceability. Operational measures must be expressed in terms of sampling, test, and criteria for judgment in order to have communicable meaning.³

A definition checklist framework developed by SEI is described as one way to explain operational definitions of measures. The checklist format is flexible and easy to use and understand. Each checklist describes exactly what is included and what is excluded from a particular measure. Each checklist is also classified using various characteristics of the measurement. For example, a defect measurement might include unique software defects, against

code, but exclude duplicates, hardware defects, and defects against test cases.

Once defined, measures are integrated with the development process. This step involves ensuring that any existing measures are not duplicated and producing an action plan for implementing the measurement program. Existing measures may need to be defined, modified, or eliminated to ensure alignment with business goals. The action plan identifies sources of data; tools utilized; and details about collecting, storing, and reporting measures.

Since process performance data is used to judge process stability and capability, consider issues such as the time sequence (the order of observations) and data precision (rounding of numerical values appropriate to the measuring instrument). With action plan in hand, it is time to start collecting data (i.e., the plan is put into practice). This means that people involved should be fit for the task, tools are pressed into service, and processes are ready for use.

During data collection, the data needs to be validated to ensure integrity. Data should be of the correct type (alpha, numeric, boolean), format (count, currency, date), within range constraints, and complete. As I stated in the April 2000 issue of *ITMS*, if you are not willing or able to validate the data, be sure you are on a short-term contract.⁴ This way, by the time anyone figures out what a mess you've made, your contract will be finished and you'll be sunning yourself in the French Caribbean.

Consistency of the collected data is also crucial to any analysis that includes a time series or comparative study. Consistency means both within a group and between groups. If multiple data collectors, tools, or measurement definitions are in use, this is of particular concern. This is where precise definitions of measures become very important. If a measurement definition is unclear, the analysis of that data for the purpose intended is questionable at best.

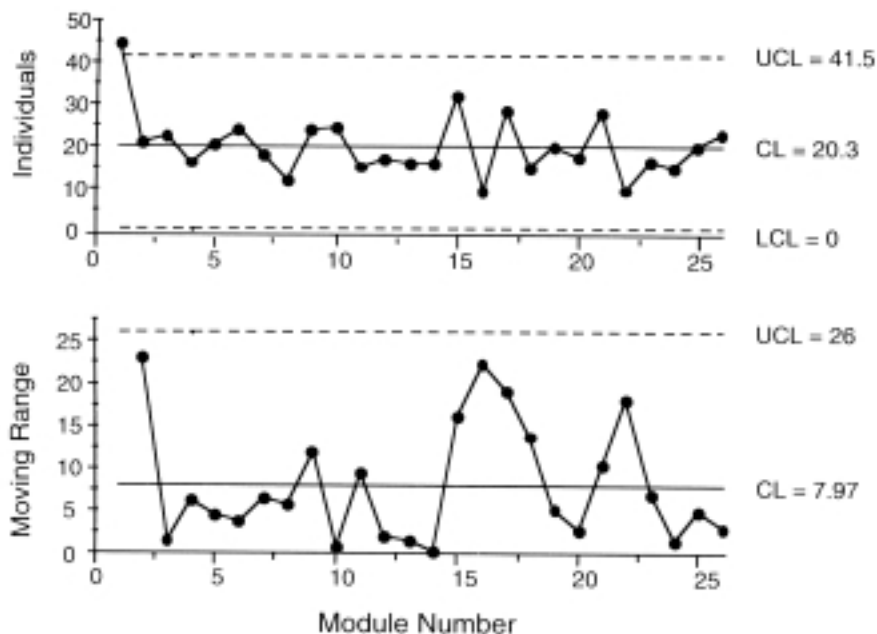
Magnificent Five?

The authors present an abbreviated treatment of a portion of what Kaoru Ishikawa called

²Van Solingen, Rini, and Egon Berghout. *The Goal/Question/Metric Method*. McGraw-Hill, 1999.

³Deming, W. Edwards. *Out of the Crisis*. MIT Press, 2000.

⁴Heires, James T. "The High-Technology Detective." *ITMS*, April 2000.



CL = control limit
UCL = upper control limit
LCL = lower control limit

©1999 Addison-Wesley. Reprinted with permission.

Figure 2 — Individuals and moving range charts for a software inspection process.

the magnificent seven tools of quality control.⁵ These useful tools are presented in order to help identify and analyze trends, relationships, and problems. The explanation is a bit too brief for anyone to begin using them, however. In fact, less than 4% of the book's pages are dedicated to explaining these valuable analysis tools. (If only William Shatner's singing in a Priceline.com commercial would be so brief!) Although several references are cited, the authors admit that none utilize examples from the software industry. This is unfortunate, since the book is written for software industry professionals.

The magnificent seven tools are the check sheet, Pareto chart, cause-and-effect diagram, histogram, scatter diagram, flow chart, and control chart. This book leaves out an introduction to check sheets and flow charts. Flow charts are particularly useful for defining, documenting, and analyzing processes, and therefore should have been explained.

"There's an angel with a hand on my head, she says I've got nothing to fear."⁶

⁵Ishikawa, Kaoru. *Guide to Quality Control*. Quality Resources, 1982.

⁶"Put Your Lights On." Carlos Santana, from the album *Supernatural*. Arista Records, 1999.

Control Charts to the Rescue

The remainder of the book illustrates the various ways control charts — and, to a lesser degree, histograms — are leveraged to assist with the task of reasoning about the data. Reasoning about data, after all, is what communication master Edward Tufte cites as the primary purpose of any technical illustration.⁷ Proper use of control charts leads to processes that are stable and capable of measurable performance in the future.

This section constitutes a large portion of the book and itself is worth the price of the book. The importance of appreciating and reducing variation is presented, followed by an overview of control chart construction.

Next is a more in-depth discussion of how to construct and use the various types of control charts in a software development setting. Many fitting examples are explained within a problem-solving context, such as quality improvement, cycle time reduction, and system availability. Figure 2 shows one example of the use of control charts to measure and improve one aspect of an inspection process.

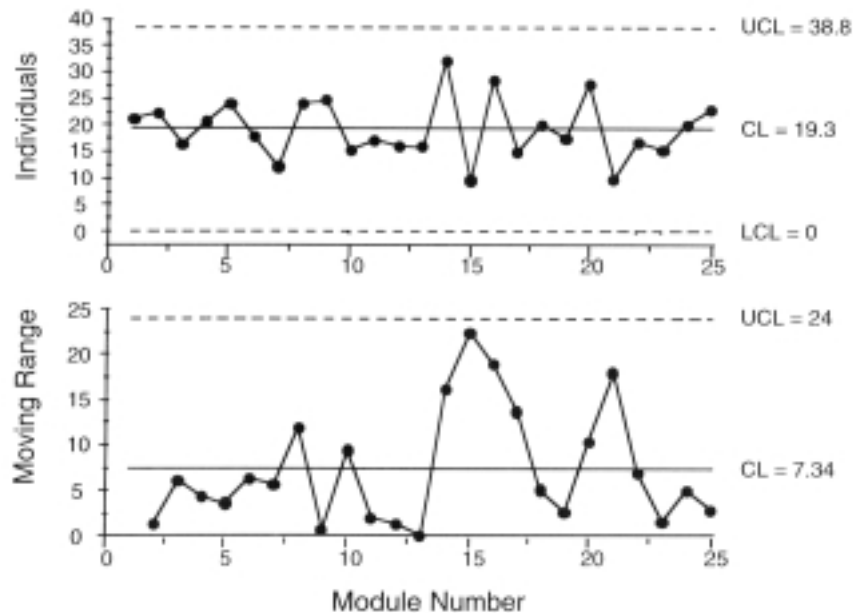
⁷Tufte, Edward. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

This is an individuals and moving range (XmR) type control chart showing measurements of defect density from 26 different software modules in the upper chart and the moving range (difference between individual measures) in the lower chart. The upper control limit (UCL), lower control limit (LCL), and center line (CL) are all calculated from the observed data. The message this chart communicates is that the first individual measurement represents an assignable cause (because its value is greater than the UCL). Any value lower than the LCL would also indicate an out-of-control condition. In fact, there are numerous other rules which signal a process that is considered out of statistical control. The presence of an assignable cause conveys the system is out of statistical control and that something needs to be done to bring the system into control. This leads the analyst to conclude that the process that produced this data is not in a state of statistical control. This means that, at this time, predictions of defect density cannot be made about software modules created by this process. Although the moving range chart does not indicate that an

out-of-control condition exists, the conclusion is the same because of the individuals chart. The individuals chart and the moving range chart pair is analyzed jointly to determine the stability of a process.

Figure 3 shows the same XmR chart less the first data point. Suppose the inspection process produced this data set. Note the effect this has on the calculated values of UCL, CL and LCL. Since both the individual and range values exhibit less variation, the control limits and center line values are reduced. More importantly, since no values fall outside the control limits, this process can be described as under statistical control. Further investigation may be warranted in the search for assignable causes, but so far this process appears to be stable.

An entire chapter is devoted to finding and correcting assignable causes, the presence of which defines the absence of statistical control. Achieving statistical control is very important, for skipping this stage prevents improving the performance level (see sidebar, "Stability Before Accuracy" on page 15). In other words, from a statistical point



CL = control limit
 UCL = upper control limit
 LCL = lower control limit

©1999 Addison-Wesley. Reprinted with permission.

Figure 3 — Individuals and moving range charts in statistical control.

Stability Before Accuracy

“Stability before accuracy” is a principle taught in traditional statistics, which, to the untrained, seems counterintuitive. It states that reducing the variation of a characteristic (e.g., defects) must occur prior to moving the mean (e.g., reducing defects). The premise is that if one tries to improve the level of performance without first stabilizing variation, the results become worse, not better. This has been experimentally proven and is discussed in W. Edward Deming’s book *Out of the Crisis* (MIT Press, 2000). This phenomenon is due to the fact that assignable causes of variation affect the ability of a process to consistently meet target values; they must be removed before attempting to change the level of performance.

For example, let’s say you’re an archer wishing to land more arrows in the target’s bull’s eye (see Figure 4). After studying where the first arrow lands, you adjust your aim to compensate for error in the previous attempt. This technique causes your performance to get worse (Figure 4D), because you don’t know why the error is present in the first place. Instead, first work to reduce variation (Figure 4B); then it is much easier to improve performance (Figure 4C) by adjusting the central tendency.

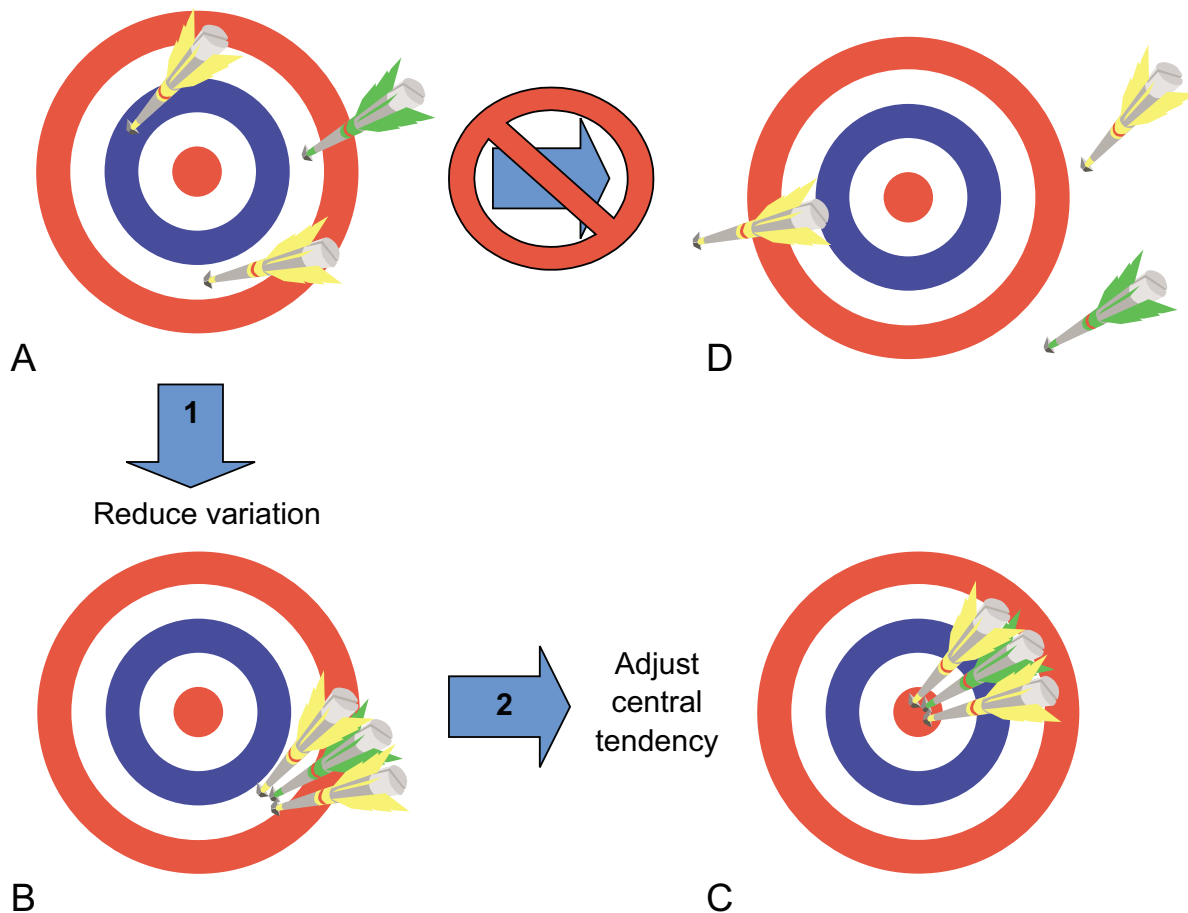


Figure 4 — Statistical improvement sequence.

of view, it makes no sense to try to improve the process before first bringing the process into a state of statistical control.

The final section of the book includes a brief discussion of process capability. A capable process is one that can be relied upon to meet business or customer requirements now and into the future.

Process capability is the Holy Grail of process improvement — it's too bad the authors chose to give the topic the short shrift. With a capable process, one can state with a specific degree of certainty, for example, that the defect density of products produced with that process will meet customer expectations of less than 10 defects per 1,000 delivered source lines of code. This is an enviable position to be in from the standpoint of predictably meeting customer demands. The organization that achieves capable software development processes is an organization with few competitors.

Conclusion

Any good SPC book will show you how to construct control charts — as does *Measuring the Software Process*. This book is different though, because it shows through software examples how to use control charts to solve real product- and process-oriented problems.

The book is an excellent introduction to SPC for software process measurement, but it lacks adequate detail to carry out a complete measurement-based improvement program.

The good news is that several other sources are available to assist where this reference falls short. This book could have made a more lasting impression by demonstrating, with more complete examples, how SPC techniques are used to improve the software process.

By all means, read this book and use it to assist with your SPI initiative. Then, read some of the references cited in the book (and in this article) to round out your understanding of the subject.

Finally, do try this at home! It is only by using these techniques on your own projects that you will truly appreciate their importance and usefulness.

About the Author

James T. Heires is a 14-year veteran of the software industry, spending the majority of his time with Rockwell Collins, Inc. His professional experiences include design of electronic flight instrumentation, consumer electronics, and software process improvement. Additionally, Heires's work in software process improvement featured the achievement of SEI Capability Maturity Model (CMM) Level 3 in two Rockwell Collins business units. Most recently, Heires has been improving the state-of-the-practice in IT using CMM, parametric estimating, and quantitative project management. Heires can be reached via e-mail at jtheires@netins.net.

Please start my subscription to *IT Metrics Strategies*® for one year at \$485, or US \$545 outside North America. Phone Megan Nields at +1 781 641 5118, or fax +1 781 648 1950, or e-mail info@cutter.com.

Please renew my subscription.

Name _____

Title _____

Organization _____

Dept. _____

Address _____

City _____ State/Province _____

Zip/Postal Code _____ Country _____

Tel. _____ Fax _____

E-mail _____

Payment or purchase order enclosed

Please bill my organization

Charge my Mastercard, Visa, American Express, Diners Club, or Carte Blanche

220*5ITS

Card no. _____

Expiration Date _____

Signature _____

**Web site: www.cutter.com/itms/
Cutter Information Corp.
Suite 1, 37 Broadway
Arlington, MA 02474-5552 USA**