

PD05

Metrics Unleashed: Measuring Productivity or Inspiring Artisans

Greg Nelson - ThotWave Technologies, Cary, North Carolina
Neil Howard – Independent Consultant, Thousand Oaks, California

Abstract

To successfully impose metrics in a statistical programming environment where “faster, better, cheaper” is often the motto, we need to understand the problems we’re trying to solve and goals we need to meet with these numbers. Among the questions this paper will address: Who is asking for metrics – sponsor, CRO, staff or executive management, internal or external source? What should we measure? How do we measure and report? Will the measurement be meaningful? What about meeting deadlines, coming in under budget, satisfying requirements, and/or error-free deliverables? Should we count the number of tables, listings, derived datasets and figures? How do we account for level of complexity, re-use code, validation, ad hoc requests, re-work, and changes? Do we risk sacrificing quality, de-motivating employees, or promoting the wrong kind of competition? How do we factor in training, mentoring, teamwork, and customer satisfaction? We are striving for a mechanism that will promote the right kinds of behaviors.

In Dilbert's words: *"We've got to be more action-oriented. From now on, measure once, cut twice!"*

Introduction

In the fast paced world of clinical trials research, we are inundated with numbers. P-values, coefficients, eigenvalues and in some cases, these are accompanied by Greek symbols to give us the illusion that they are correct and indisputable – statistical evidence to support our theories. So when we are presented “numbers” that represent level of effort, percent complete, billability estimates or metrics which measure productivity of a human workforce, we tend to believe them. Or do we?

Metrics have been used to help us improve clinical trial performance, establish operational benchmarks for performance, create a balanced view (or scorecard) that monitors trial performance, financial metrics, customer satisfaction, and organizational growth and performance. We can, of course, can use these metrics to identify sub-standard internal performance and industry-relative performance.

In this paper, we will talk about metrics – those numbers that we use to set expectations for future behavior based on past performance. Numbers that define how someone is doing or should be doing. On a grander scale, we can estimate how long a clinical trial should take by therapeutic area or perhaps estimate patient recruitment costs. But the focus of this paper is really about understanding “how do we measure programmer productivity?”

Metrics: a retrospective

Measuring programmers has long been discussed since the art of crafting 1's and 0's into beautifully orchestrated symphonies of functional relevance started a half a century ago. Managers use metrics as a way of forecasting work and capability, evaluating programmer's productivity and often as a performance

appraisal tool. In fact, we often see managers looking at lines of code (or number of semicolons!), number of bugs or defects and even number of function points. Software measure researchers are split into two camps: those who claim software can be measured, and those who say that software cannot be analyzed by measurement. In any case, the majority of us are concerned about software quality and the need to quantify it. During the past, more than one thousand software measures have been proposed by researchers and practitioners, and still today more than 5000 papers about software measurement are published. Measurement has a long tradition in natural sciences. At the end of the last century the physicist, Lord Kelvin (1824-1904), formulated the following about measurement (Kelvin, 1891):

“When you can measure what you are speaking about, and express it into numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.”

If you really want to understand software metrics and how to measure programmers, go an excellent source is “Software Metrics: A Rigorous & Practical Approach” (Fenton & Pfleeger, 1998). They provide an excellent overview of measurement theory and measurement in practice (in software development) and there are almost 60 pages of references in the annotated bibliography for the motivated reader.

Taken together, measuring progress is a good thing. In fact, there is evidence to support that simply just measuring something makes it better. Measurement can certainly provide motivation which is one of the central theories about why metrics work as a tool to measure productivity.

Triplett (1898) observed that bicycle racers have faster times when they race against others than when they race alone. Based on this observation, he hypothesized that the presence of competitors causes people to perform tasks faster. The general phenomenon demonstrated by Triplett is called *social facilitation*, which refers to the faster (or better) performance of a task when others are present. Triplett's study was one of the first to demonstrate this claim experimentally.

More recently, back in the 1960's when Don Clifton took over the Gallup organization; he noted huge improvements in a keypunch operator's performance, just by telling them how many keystrokes they could produce and those of their fellow employees.

Measurement and Motivation

A wise person once said “tell me how someone is measured, and I will tell you how they behave.” We have to be very careful to measure the right things in order to elicit the correct behavior. We all know, for example, that most used car salesmen get paid a commission on the total cost of the car they sell you. They are rewarded to sell the car at the highest price possible. It is only the exceptional personalities can resist the temptation to charge a higher-than-value price on that car.

Motivation, talent, drive, dexterity, and experience are all factors in determining how successful a programmer can be given a task. With enough motivation any ordinary person can become a world class athlete. Without it the same person could end up begging for change downtown. Even a tremendously talented programmer with little motivation will go nowhere without motivation. How do some people always seem to be so motivated? What are the sources of their motivation? This has been a central theme when Triplett studied the effects of audience and competition on performance in the late nineteenth century. Though a great deal has been written on motivation since then it is still an individual construct. As a

programmer you need to identify what motivates you and cultivate the sources of your motivation. Obviously, individual differences will no doubt play an important role in individual performance. Some people just think faster; type faster; can process complex business rules easier.

In this paper, we aren't so much interested in individual performance differences; it is important to examine metrics more at a macro level. Here we will lay out the discussion in 4 logical sections:

- **Metrics Defined** – what is it that we measure, who do we measure and why is this important.
- **Measuring statistical programming** - here we will talk about traditional methods to measure software engineers and then translate that to statistical programming. In addition, we will discuss the role of the instrument in motivation as well as how we “sell” it to the masses and what metrics might be appropriate
- **Quality versus Quantity** – here we will outline our perspectives on quality and the importance of measuring the right thing in the right way.
- **Creating the complete picture** – sometimes the metric alone doesn't convey the whole story. In this section, we will also describe the intangibles that have to be considered with the metric.

Metrics Defined

When we talk about metrics, we are talking about quantifying things that give us a sense of productivity or accomplishment. Productivity is the amount or value of output per unit of input. So, theoretically, a programmer who makes \$70,000 per year and received 2 weeks of training, 100 square feet of cubicle space and 10 gallons of coffee a year, should be more productive than one receiving only \$60,000 in compensation and drinks less coffee. Ludicrous. We know that people can't be measured that way.

So why measure, as Horst Zuse outlined for us (Zuse, 1995):

“We use software measures to derive:

- A basis for estimates,
- To track project progress,
- To determine (relative) complexity,
- To help us to understand when we have archived a desired state of quality,
- To analyze our defects,
- and to experimentally validate best practices.

...in short: they help us to make better decisions.”

We often think about our work in certain quantities and their various complexities. If we have to produce a listing for example, we know that the inputs will be the requirements (Statistical Analysis Plan), codebook for the Case Report Forms, and the dataset that we will use as our source. We know typically how long it takes to produce that and we can feel confident that our more experienced programmers (that is, someone familiar with our system, taxonomy of projects, standard macros, familiarity with clinical data and validation requirements) can do this type of thing fairly quickly. Is it worth estimating the time exactly or with a confidence interval?

PhUSE 2006

So, let's explore what form measuring programmer productivity has taken over the years and outline pros and cons. (Note: for a much more complete perspective on the history of software measurement, see Zuse, 1995). In the following table, we outline the two most common ways of measuring programmer productivity.

Metric Method	Description	Characteristics
Lines of Code (KLOC, SLOC, LOC) ¹	Source lines of code (SLOC) is a software metric used to measure the amount of code in a software program. SLOC is typically used to estimate the amount of effort that will be required to develop a program, as well as to estimate productivity or effort once the software is produced.	Lines of code can be measured by looking at either the physical lines of code or by measuring the “statements” which are specific to a language. Different approaches for measuring lines includes those that count comment lines and blank lines versus those that don't count those. Useful for looking at the “order of magnitude” for a project rather than specifically how productive a programmer has been. The metric is also not terribly useful for measuring code generation techniques such as macros since the macro would likely be much smaller than the resulting code had a macro not been used (but more efficient).
Function Point Analysis ²	A technique used to determine the size and complexity of a development task (programming problem/ system), based on the number of function points. It is a way to decomposing a problem into chunks that can be reasonably achieved and described in business terms.	Function Points can be used to size software applications accurately. Sizing is an important component in determining productivity (outputs/inputs). They can be counted by different people, at different times, to obtain the same measure within a reasonable margin of error. Function Points are easily understood by the non technical user. This helps communicate sizing information to a user or customer. Function Points can be used to determine whether a tool, a language, an environment, is more productive when compared with others.

¹ Note: for more information, see http://en.wikipedia.org/wiki/Source_lines_of_code

² See for example: <http://www.ifpug.com/fpafund.htm>

In the table, we outlined the classic approaches for measuring programmer productivity. Obviously each of these can have its own set of challenges. For example, in counting lines of code, we all know that sometimes a much more efficient programmer is one who is more concise. Knowing how many line breaks in their program tells us little to nothing about their skill, whether it met the requirement or how much white space they have in the programs.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight." ~ Bill Gates.

Function point analysis may be challenging to learn and develop a standardized process for counting function points. Bias may also be a factor. Project personnel may overstate counts because they will be judged on the size of the system they deliver. Project customers may understate the size to push for quicker and cheaper delivery. There is a need for someone who is being judged solely on the accuracy of the count and any associated estimates not on the magnitude of the number.

Measuring Statistical Programming

Translating these concepts into something we can use in the SAS ecosystem is certainly challenging. Most of the researchers that discuss software measurement speak generally about compiled languages such as C, C++ and semi-compiled languages like Java. However, the idea of figuring out what to measure is still the most critical thing to keep in mind.

If you create a metric and expect people to be measured by that metric, be prepared for unnatural behaviors. For example, if someone got “rewarded” (meaning compensated or evaluated) based on the number of modules they created; you can likely expect to see lots of modules in the end result. If lines of code (LOC) is what is measured, then you’ll see long programs with lots of carriage returns, comments, blank spaces and text copied over and over again. So let’s think about what behavior we are really trying to promote. We want high quality software that is simple to maintain, highly reusable and demonstrates the required functionality as outlined in the functional specifications. The key in our minds is to create metrics for the following:

- Accuracy and completeness of the requirements
- Some measure as to the reusability of this code after its intended purpose (and a corollary to this, how well documented and structured is this code)
- Some built in way to prove that it meets the requirements
- low dependence on other programs, data structures or knowledge of other modules

In addition, we realize that not all tasks are the same, so we need some sense of the complexity of the program that will be required to satisfy the requirement. Further, the complexity can be thought of in terms of data management complexity (how difficult will it be to cajole the data into the form that is required), analytics complexity (is the statistical analysis well known or perhaps risky?) and finally, presentation complexity – that is, how complex are the requirements for look and feel.

Quality versus Quantity

In 2002, Allan Glaser put forth eleven criteria to consider when approaching metrics for programming:

- Correctness
- Maintainability
- Modularity
- Reliability
- Understandability
- Compliance
- Completeness
- Adaptability
- Generality
- Portability
- Efficiency

Correctness assumes dependence on a well-ingrained software development life cycle (SDLC) that ensures specifications and code can be compared and that includes a well-defined validation process. Glaser states that maintainability “is inversely related to complexity” – the implication being to keep things simple and straightforward. Inside every complex program is a simple program screaming to get out. Modularity isolates components or sections of code that accomplish a single function and represent limited inputs and outputs. And there’s no question that modularity is directly related to maintainability. In its simplest form, reliability looks at a program’s performance over time with respect to producing the desired results and at what error rate. Understandability is dependent on programming style to ensure clarity, and we are supportive of Glaser’s mention of peer review as being a significant element of achieving understandability.

In a highly regulated environment, compliance is the degree of adherence to industry standards and conventions – as defined by an organization’s current SOPs, guideline, checklist, and working practices. Completeness is usually determined in the user acceptance phase of the SDLC and is another check against the original requirements.

The idea of adaptability feeds into the model of developing reuse code and generalized code, and is critical to understandability and maintainability. In most CRO, pharmaceutical and biotech environments, the programmers strive to develop macro libraries and standardized programs that can be used across studies. Programmers look for functions that are generalizeable. The SAS System lends itself to our ability to implement portable solutions in our industry.

And lastly, Glaser defines an efficient program as one that “consumes a minimum of resources”. We contend, too, that depending on the environment, there are many trade-offs to consider, whether they be issues of human capital, CPU, storage, performance or other elements.

Unofficial Survey on Metrics

One of authors, in the process of writing a paper on hiring statistical programmers, conducted a non-scientific survey on the hiring process that included questions on metrics: programming metrics and measures of productivity. Those queried were directors, managers, recruiters and programmers. Most respondents were in agreement that it is hard to quantify and qualify what programmers and analysts do to the satisfaction of 'bean counters' (accountants/ number crunchers). It is also clear from the lack of response that this is a touchy, often untouched, subject. The following responses are all things to consider when you try to predict how imposing metrics on an organization will be received.

The managers among the respondents are often asked to devise metrics for their group. Non-technical senior management need to see numbers. "Happy clients" won't satisfy this mindset; but it is equally difficult to apply metrics. Productivity measures most often mentioned as making the most sense are things like: timeliness of deliverables, performance against deadlines, end-user feedback, accuracy of status reporting, ability to solve problems. But these are hard to quantify.

Respondents also referred to: the ability to take feedback, success in teamwork, adherence to specifications, self-motivation, knowing when and how to ask for help, having generated repeat business, follow-through, willingness to share and mentor, and being innovative in attacking tasks.

Several mentioned adherence to SOPs and programming standards, begging the issue of how many organizations use these tools. Some use strict code standards for readability and maintainability, checklists, guidelines, and keep complete validation folders. Questions on QC and validation revealed that accuracy of results was important, yet many did not use program checkers or QC reviewers. The measure was client acceptance.

Number of lines of code written was dismissed almost entirely as a useful metric for programmers. Comments indicate it could promote "hackerism". Programmers would not be striving for the most efficient solutions and they would not be motivated to document their programs or maintain progress reports. Programmers often claim the number of hours worked is an effective measure. Managers, however, were looking for programmers who work smarter - not longer.

Creating the Complete Picture

But it isn't just about counting. You have to consider an entire context when introducing the collection of metrics. Linda Westfall advocates a 12 step approach to implementing and using meaningful software metrics that goes a long way in completing the picture.

1. identifying metrics customers
2. setting target goals
3. determine questions to ask to decide if a goal has been met
4. select the metrics

PhUSE 2006

5. standardize
6. decide how you're going to calculate the metric, direct or derived
7. establish the measurement method for the functions in 6.
8. define what you're going to do with the results
9. define the reporting mechanism(s)
10. determine any additional qualifiers (e.g., the demographic information for views of the metric results)
11. collect the data
12. consider the people:
 - a. don't measure individuals
 - b. don't use metrics as a "stick" against an individual or group
 - c. don't ignore the data – use it to support actions
 - d. never use only one metric
 - e. select metrics based on goals
 - f. provide feedback
 - g. obtain buy-in.

To those 12 steps, we would add:

- Continuously evaluate the metrics (get feedback, study the utility of your measurement results),
- Continuously improve the process,
- and Communicate.

As author Jim Clemen's states in his article, "Don't Wait to See the Blood!": "Improving...performance without constant feedback is like trying to pin the tail on the donkey when we're blindfolded; only through knowing where we are, can we change where we are going."

Summary

There is no question that measuring quality and productivity in the context of statistical programming is a key differentiator for any company spending its share of the multi-billion dollar drug market. We have to produce higher quality software in shorter time periods. We are faced with off-shoring and out-sourcing decisions all of the time. Justifying team composition and head count is part of our everyday world – as is balancing work load. It is not a question of if we should measure, but how.

Attempts by organizations and researchers alike to define a correct path for software measurement have been wrought with complexities. There is still a lack of maturity in software measurement. There is still no standardization of software measures. The proposed software measures in (IEEE, 1989) are not widely accepted. However, we need to spend the right amount of energy in developing a metrics program that can

improve time to delivery, improve quality of software and continue to motivate and excite the programmer community.

Afterwords:

- WARNING: You'll Get What You Measure
- Abraham Lincoln: "If I had eight hours to chop down a tree, I'd spend six hours sharpening my ax."
- Werner Karl Heisenberg: "Since the measuring device has to be constructed by the observer ... we have to remember that what we observe is not nature itself, but nature exposed to our method of questioning."
- Lord Kelvin: If you can not measure it, you can not improve it.
- Lord Kelvin: To measure is to know.
- The Metricator's Maxim: Not all that counts can be counted; not all that can be counted, counts.
- Alexander's 1st Law of Metrics: Metrics are hard to get on projects which don't keep records.
- Simplicity is prerequisite for reliability. (Edsger Dijkstra)
- Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (Edsger Dijkstra)
- The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague. (Edsger Dijkstra)

Acknowledgments

The authors would like to thank Jack Shostak, Mark Matthews, and Kyle McBride for their contributions to the panel discussion on metrics at PharmaSUG 2006 in Bonita Springs, FL.

SAS is a registered trademark of SAS Institute Inc., Cary, NC.

References:

- Fenton, Norman E. and Lawrence Pfleeger, Shari (1998). *Software Metrics: A Rigorous & Practical Approach*. Course Technology; 2 edition (February 24, 1998)
- Garmus, David and Herron, David (2000). *Function Point Analysis: Measurement Practices for Successful Software Projects*. Published by Addison Wesley Professional.
- IEEE: *Standard Dictionary of Measures to Produce Reliable Software*. The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017-2394, USA IEEE Standard Board, 1989.
- Jones, Capers (1991) *Applied Software Measurement: Assuring Productivity and Quality* (Software Engineering Series).
- Jones, Capers (2000) *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley.
- Kelvin, W.T. . (1891-1894): *Popular Lectures and Addresses*.

PhUSE 2006

Triplett, Norman (1898). *The Dynamogenic Factors in Pacemaking and Competition*. American Journal of Psychology, 9, 507-533.

Zuse, Horst (1995). *History of Software Measurement*. Available at http://irb.cs.tu-berlin.de/~zuse/metrics/History_00.html

Author Contact Information:

Greg Nelson, President and CEO of ThotWave Technologies, LLC.

ThotWave Technologies, LLC.
2054 Kildaire Farm Rd #322
Cary, NC 27511
800-584-2819
greg@thotwave.com

Neil Howard

Amgen Inc.
One Amgen Center Drive, B24-2-C
Thousand Oaks, CA
805-447-5713
neil.howard@amgen.com