Software Engineering Metrics: What Do They Measure and How Do We Know?

Cem Kaner, *Senior Member, IEEE*, and Walter P. Bond

Abstract—Construct validity is about the question, how we know that we're measuring the attribute that we think we're measuring? This is discussed in formal, theoretical ways in the computing literature (in terms of the representational theory of measurement) but rarely in simpler ways that foster application by practitioners. Construct validity starts with a thorough analysis of the construct, the attribute we are attempting to measure. In the IEEE Standard 1061, direct measures need not be validated. "Direct" measurement of an attribute involves a metric that depends only on the value of the attribute, but few or no software engineering attributes or tasks are so simple that measures of them can be direct. Thus, all metrics should be validated. The paper continues with a framework for evaluating proposed metrics, and applies it to two uses of bug counts. Bug counts capture only a small part of the meaning of the attributes they are being used to measure. Multidimensional analyses of attributes appear promising as a means of capturing the quality of the attribute in question. Analysis fragments run throughout the paper, illustrating the breakdown of an attribute or task of interest into sub-attributes for grouped study.

Index Terms—D.2.8 Software Engineering Metrics/Measurement, D.2.19.d Software Engineering Measurement Applied to SQA and V&V.

1 Introduction

We hear too often that few companies establish measurement programs, that fewer succeed with them, or that many of the companies who have established metrics programs have them in order to conform to criteria established in the Capability Maturity Model. [1]

One *could* interpret this as evidence of the immaturity and unprofessionalism of the field or of resistance to the high cost of metrics programs (Fenton [1] estimates a cost of 4% of the development budget). In some cases, these explanations are undoubtedly correct. In other cases, however, metrics programs are resisted or rejected because they do more harm than good.

Robert Austin [2] provided an excellent discussion of the problems of measurement distortion and dysfunction in general. In this paper, we explore one aspect of the problem of dysfunction. We assert that Software Engineering as a field presents an approach to measurement that underemphasizes measurement validity (the condition that the measurement actually measures the attribute in question). This has a likely consequence: if a project or company is managed according to the results of measurements, and those metrics are inadequately validated, insufficiently understood, and not tightly linked to the attributes they are intended to measure, measurement distortions and dysfunctional should be commonplace.

After justifying our basic assertion, we lay out a model for evaluating the validity and risk of a metric, and apply it to a few metrics common in the field. Not surprisingly (given our main assertion), serious problems will show up.

In the final section of this paper, we suggest a different approach: the use of multidimensional evaluation to obtain measurement of an attribute of interest. The idea of multidimensional analysis is far from new [3], but we will provide detailed examples that appear to have been used effectively at the line manager level, in the field. A pattern of usability and utility emerges from these examples that, we hope, could stimulate further practical application.

- *Cem Kaner is Professor of Software Engineering at the Florida Institute of Technology, Melbourne, FL, 32901. E-mail: kaner@kaner.com.*
- *Walter Bond is Associate Professor of Computer Science at Florida Institute of Technology, Melbourne, FL, 32901. E-mail: pbond@cs.fit.edu*

2 What are We Measuring?

2.1 Defining Measurement

To provide context for the next two sections, we need a definition of measurement. To keep the measurement definitions in one place, we present several current definitions here. We'll distinguish between them later.

- "Measurement is the assignment of numbers to objects or events according to rule. [4] The rule of assignment can be any consistent rule. The only rule not allowed would be random assignment, for randomness amounts in effect to a nonrule." [5, p. 47]
- "Measurement is the process of empirical, objective, assignment of numbers to properties of objects or events of the real world in such a way as to describe them." [6, p. 6]
- "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterize them according to clearly defined rules." [7, p.5]
- Measurement is "the act or process of assigning a number or category to an entity to describe an attribute of that entity." [8, p. 2]

- "Fundamental measurement is a means by which numbers can be assigned according to natural laws to represent the property, and yet which does not presuppose measurement of any other variables" than the one being measured. [9, p. 22]

More formal definitions typically present some variation of the representational theory of measurement. [10] [7] [11] [12] [13] Fenton and Pfleeger provide a concise definition:

> Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute. [7, p. 28]

2.2 Developing a Set of Metrics

IEEE Standard 1061 [8] lays out a methodology for developing metrics for software quality attributes. The standard defines an *attribute* as "a measurable physical or abstract property of an entity." A *quality factor* is a type of attribute, "a management-oriented attribute of software that contributes to its quality." A metric is a measurement function, and a software quality metric is "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality."

To develop a set of metrics for a project, one creates a list of quality factors that are important for it:

> Associated with each quality factor is a direct metric that serves as a quantitative representation of a quality factor. For example, a direct metric for the factor reliability could be mean time to failure (MTTF). Identify one or more direct metrics and target values to associate with each factor, such as an execution time of 1 hour, that is set by project management. Otherwise, there is no way to determine whether the factor has been achieved. [8, p. 4]

> For each quality factor, assign one or more direct metrics to represent the quality factor, and assign direct metric values to serve as quantitative requirements for that quality factor. For example, if "high efficiency" was one of the quality requirements from 4.1.2, assign a direct metric (e.g. "actual resource utilization / allocated resource utilization" with a value of 90%). Use direct metrics to verify the achievement of the quality requirements. [8, p. 6]

> Use only validated metrics (i.e. either direct metrics or metrics validated with respect to direct metrics) to assess current and future product and process quality (see 4.5 for a description of the validation methodology). [8, p. 6]

Standard 1061 (section 4.5) lays out several interesting validation criteria, which we summarize as follows:

1) *Correlation.* The metric should be linearly related to the quality factor as measured by the statistical correlation between the metric and the corresponding quality factor.
2) *Consistency.* Let F be the quality factor variable and Y be the output of the metrics function, M: F->Y. M must be a monotonic function. That is, if $f_1 > f_2 > f_3$, then we must obtain $y_1 > y_2 > y_3$.
3) *Tracking*. For metrics function, M: F->Y. As F changes from f_1 to f_2 in real time, M(f) should change promptly from y_1 to y_2.
4) *Predictability.* For metrics function, M: F->Y. If we know the value of Y at some point in time, we should be able to predict the value of F.
5) *Discriminative power.* "A metric shall be able to discriminate between high-quality software components (e.g. high MTTF) and low-quality software components (e.g. low MTTF). The set of metric values associated with the former should be significantly higher (or lower) than those associated with the latter.
6) *Reliability.* "A metric shall demonstrate the correlation, tracking, consistency, predictability, and discriminative power properties for at least P% of the application of the metric."

The validation criteria are expressed in terms of quantitative relationships between the attribute being measured (the quality factor) and the metric. This poses an interesting problem—how do we quantify the attribute in order to compare its values to the proposed metric?

2.3 "Direct" Measurement

The IEEE Standard 1061 answer lies in the use of direct metrics. A *direct metric* is "a metric that does not depend upon a measure of any other attribute." [8, p. 2]

Direct metrics are important under Standard 1061, because a direct metric is presumed valid and other metrics are validated in terms of it ("Use only validated metrics (i.e. either direct metrics or metrics validated with respect to direct metrics)"). "Direct" measurement is often used synonymously with "fundamental" measurement [9] and contrasted with indirect or derived measurement [14].

The contrast between direct measurement and indirect, or derived measurement, is between a (direct) metric function whose domain is only one variable and a (derived) function whose domain is an n-tuple. For example, density is a function of mass and volume. Some common derived metrics in software engineering are [7, p. 40]:

- Programmer productivity (code size/ programming time)
- Module defect density (bugs / module size)
- Requirements stability (number of initial requirements / total number of requirements)
- System spoilage (effort spent fixing faults / total project effort)

Standard 1061 offers MTTF as an example of a direct measure of reliability. But if we look more carefully, we see that this measure is not direct at all. Its values depend on many other variables. As we'll see, this is true of many (perhaps all) software engineering metrics. Analyzing the components of Mean Time To Failure:

- *Mean?* Why calculate *mean* time to failure? Imagine two subpopulations using the same product, such as a professional secretary and an occasional typist using a word processor. The product might fail rarely for the secretary (who knows what she's doing) but frequently for the occasional typist (who uses the product in odd or inefficient ways). These two types of users have different *operational profiles* [15]. They use the product dif-

ferently and they experience it differently (high versus low reliability). The average (mediocre reliability) is not representative of either group's experience. Perhaps MTTF is an indirect measure of reliability, because it is partially a function of the operational profile of the user subpopulation. Similarly, if new users of a product tend to experience more failures until they learn how to avoid or work around the problems, *mean* time to failure is misleading because the failure probability is not stationary. MTTF appears to be a function of the individual's experience with the product, the user subpopulation's operational profile, and the inherent reliability of the product. *What other variables influence the mean of the times to failure?*

- *Time?* What *time* are we counting when we compute mean *time* to failure? Calendar time? Processor time? Suppose that User-1 operates the product for 10 minutes per day and User-2 operates the product for 1440 minutes per day. Mean time to failure of two weeks suggests appalling reliability if it is User-1's experience, but not-so-bad reliability if it is User-2's. Another issue correlated with time is diversity of use. A person who uses the product the same way every time is less likely to experience new failures than one who constantly uses it in new ways, executing new paths and working with new data combinations. So, even if we agree on the temporal unit (calendar time, processor time, user-at-the-keyboard-time, whatever), we will still experience different mean *times* to failure depending on diversity of use. A final example: if the user can recover from failure without restarting the system, residue from a first failure might raise the probability of the next. In a system designed to recover from most failures, the system reliability as estimated by time to next failure might be a declining function of the time the product has been in service since the last reboot.
- *To?* Should we measure mean time *to first failure* or mean time *between failures*? A program that works well once you get used to its quirks might be appallingly unreliable according to MTT(first)F but be but rock solid according to MTBF. Will the *real* measure of reliability please stand up?
- *Failure?* What's a failure? Program crash? Data corruption? Display of an error message so insulting or intimidating that the user refuses to continue working with the system? Display of a help message with a comma missing in the middle of long sentence? Display of a copyright notice that grants the user more rights than intended? Any event that wastes X minutes of the user? Any event that motivates the user to call for support? If we define a failure as a behavior that doesn't conform to a specification, and we ignore the reality of error-ridden and outdated specifications, is there a rational basis for belief that all intended behavior of a program can be captured in a genuinely complete specification? How much would it cost to write that specification? In 1981, Gutenberg Software published *The Gutenberg* word processor for the Apple II computer. This was before mice were in wide use—to designate a target for a command, you used the keyboard. For example, in command mode, the sequence "LL" set up scrolling by lines, "LS" set up scrolling by sentences, "DL" deleted a line, and "DS" deleted the entire screen. Some users would scroll sentence by sentence through the document while editing, and then type DS to delete a sentence. There was no undo, so this cost a screenful of data. Screens might include complex equations that took the user hours to lay out. This was in the user manual and was part of the intentional design of the product. Is this irrecoverable (but specified) data loss a failure? Presumably, the set of events that we accept as "failures" will influence the computed time to failure, and thus our allegedly direct measurement of reliability.

We belabored analysis of MTTF to make a point. Things that appear to be "direct" measurements are rarely as direct as they look.

As soon as we include humans in the context of anything that we measure—and most software is designed by, constructed by, tested by, managed by, and/or used by humans—a wide array of system-affecting variables come with them. We ignore those variables at our peril. But if we take those variables into account, the values of our seemingly simple, "direct" measurements turn out to be values of a challenging, multidimensional function. By definition, they are no longer direct. Certainly, we can hold values of all those other variables constant, and restrict our attention to the marginal relationship between the attribute and the measured result, but that's fundamentally different from the assertion that the value of our metric function depends only on the value of the underlying attribute.

Because direct measurements have the special status of inherent validity, there is an incentive to attribute directness to many proposed measurements. Consider the four examples of direct measurement provided by Fenton & Pfleeger:

- *Length* of source code (measured by lines of code);
- *Duration* of testing process (measured by elapsed time in hours);
- *Number of defects discovered* during the testing process (measured by counting defects);
- *Time* a programmer spends on a project (measured by months worked). [7, p. 40]

One problem with these measures is that, like MTTF, they are intrinsically complex. (Like the MTTF analysis above, try this: *Lines of code?* What's a line? What's code? How do people interact with lines or code, and under what different situations? How do those differences affect the size or meaning of the size of lines of code? Repeat the same analysis for the next three.)

An different problem with these measures is that it is easy to create a metric with a narrow definition that makes it *look* direct but that will never be used as a measure of the defined attribute. (Le Vie [16] makes this point nicely for an applied audience.) For example, consider *time on project*, measured in programmer-months. How often do we really want to know about *time on project* for its own sake? What attribute are we

actually trying to measure (what question are we actually trying to answer) when we count programmer-months? The amount of effort spent on the project? Difficulty of the project? Diligence of the individual? Programmer-months is relevant to all of these, but not a direct measure of any of them, because many factors other than time on the clock play a role in all of them.

Rather than *define* a metric in terms of the operations we can perform (the things we can count) to compute it, we prefer to think about the question we want answered first, the nature of the information (the attributes) that could answer that question, and then define measures that can address those attributes in that context.

In practice, we question the value of distinguishing between direct and indirect metrics. All metrics need validation, even the supposedly direct ones.

3 A Framework for Evaluating Metrics

The term, *construct validity,* refers to one of the most basic issues in validation, the question: *How do you know that you are measuring what you think you are measuring?*

In a check of the ACM Guide to the Computing Literature (online, June 29, 2004), we found only 109 references that included the phrase "construct validity." Of those papers, many mentioned the phrase in passing, or applied it to measurements of human attitudes (survey design) rather than characteristics of a product or its development. In the development of software engineering metrics, the phrase "construct validity" appears not to be at the forefront of theorists' or practitioners' minds.

Fenton and Melton point to a different structure in which these questions are asked:

> We can use measurement theory to answer the following types of questions.
>
> 1) How much do we need to know about an attribute before it is reasonable to consider measuring it? For example, do we know enough about complexity of programs to be able to measure it?
> 2) How do we know if we have really measured the attribute we wanted to measure? For example, how does a count of the number of bugs found in a system during integration testing measure the reliability of the system? If not, what does it measure? . . .
>
> The framework for answering the first two questions is provided by the *representation condition* for measurement. [17, p. 29-30]

The representational theory is laid out generally in [6] [10] [11] and harshly critiqued by Michell [18]. Applied to computing measurement, it is nicely summarized by Fenton and Melton [17] and presented in detail by Fenton and Pfleeger [7], Morasca and Briand [13] and Zuse [12].

We agree with this way of understanding measurement, but our experience with graduate and undergraduate students in our Software Metrics courses, and with practitioners that we have worked with, taught, or consulted to, is that the theory is profound, deep, intimidating, and not widely enough used in practice.

The following approach simplifies and highlights many of what we think are the key issues of practical measurement.

3.1 Defining Measurement

Suppose that while teaching a class, you use the following rule to assign grades to students—the closer the student sits to your lectern, the higher her grade. Students who sit front and center get A's (100); those who hide in the far rear corner flunk (0). Intermediate students get grades proportional to distance.

Does this grading scheme describe a measurement?

If we accept Stevens' definition of measurement ("assignment of numbers to objects or events according to rule") as literally correct, then this grading rule does qualify as a measurement.

Intuitively, however, the rule is unsatisfactory. We assign grades to reflect the quality student performance in the course, but this rule does not systematically tie the grade (the measurement) to the quality of performance. Several definitions of measurement, such as Fenton and Pfleeger's ("process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to characterize them according to clearly defined rules") address this problem by explicitly saying that the measurement is done to describe or characterize an attribute.

What is the nature of the rule(s) that govern the assignments? This question is at the heart of the controversy between representational theory and traditional (physics-oriented) theory of measurement [18]. Under the traditional view, the numbers are "assigned according to natural laws" [9]. That is, the rule is based in a theory or model, and (in the traditionalist case) that model derives from natural laws. The ideal model is causal—a change in the attribute *causes* a change in the value that will result from a measurement.

Many current discussions of metrics exclude or gloss over the notion of an underlying model. IEEE 1061 refers to correlation as a means of validating a measure, but this is a weak and risk-prone substitute for a causal model [7].

For many variables, we don't yet understand causal relationships, and so it would be impossible to discuss measurements of those variables in causal terms. Even for those, however, we have notions that can be clarified and made explicit.

Accordingly, we adopt the following definition of measurement:

> **Measurement is the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them.**

3.2 The Evaluation Framework

To evaluate a proposed metric, including one that we propose, we find it useful to ask the following ten questions:

1) **What is the purpose of this measure?** Examples of purposes include:
   - facilitating private self-assessment and improvement. [19]
   - evaluating project status (to facilitate management of the project or related projects)
   - evaluating staff performance
   - informing others (e.g. potential customers) about the characteristics (such as development status or behavior)

of the product
- informing external authorities (e.g. regulators or litigators) about the characteristics of the product

The higher the stakes associated with a measurement, the more important the validation. A measure used among friends for personal coaching might be valuable even if it is imprecise and indirect.

2) **What is the scope of this measure?** A few examples of scope:

- a single method from one person
- one project done by one workgroup
- a year's work from that workgroup
- the entire company's output (including remote locations) for the last decade

As the scope broadens, more confounding variables can come into play, potentially impacting or invalidating the metric. A metric that works well locally might fail globally.

3) **What attribute are we trying to measure?** If you only have a fuzzy idea of what you are trying to measure, your measure will probably bear only a fuzzy relationship to whatever you had in mind.

> Measurement presupposes something to be measured. Both in the historical development and logical structure of scientific knowledge, the formulation of a theoretical concept or construct, which defines a quality, precedes the development of measurement procedures and scales.
>
> Thus the concept of 'degree of hotness' as a theoretical construct, interpreting the multitude of phenomena involving warmth, is necessary before one can conceive and construct a thermometer. Hardness must, similarly, first be clearly defined as the resistance of solids to local deformation, before we seek to establish a scale for measurement. The search for measuring some such conceptual entity as 'managerial efficiency' must fail until the concept is clarified. . . .
>
> One of the principal problems of scientific method is to ensure that the scale of measurement established for a quality yields measures, which in all contexts describe the entity in a manner which corresponds to the underlying concept of the quality. For example, measures of intelligence must not disagree with our basic qualitative concept of intelligence. It is usual that once a scale of measurement is established for a quality, the concept of the quality, the concept of the quality is altered to coincide with the scale of measurement. The danger is that the adoption in science of a well defined and restricted meaning for a quality like intelligence, may deprive us of useful insight which the common natural language use of the word gives us. [6, p. 10-12]. (For an important additional discussion, see Hempel [20].)

4) **What is the natural scale of the attribute we are trying to measure?** We can measure length on a ratio scale, but what type of scale makes sense for programmer skill, or thoroughness of testing, or size of a program? See [4] and [7] for discussions of scales of measurement.

5) **What is the natural variability of the attribute?** If we measure two supposedly identical tables, their lengths are probably slightly different. Similarly, a person's weight varies a little bit from day to day. What are the inherent sources and degrees of variation of the attribute we are trying to measure?

6) **What is the metric (the function that assigns a value to the attribute)? What measuring instrument do we use to perform the measurement?** For the attribute *length*, we can use a ruler (the instrument) and read the number from it. Here are a few other examples of instruments:

- *Counting* (by a human or by a machine). For example, count bugs, reported hours, branches, and lines of code.
- *Matching* (by a human, an algorithm or some other device). For example, a person might estimate the difficulty or complexity of a product by matching it to one of several products already completed. ("In my judgment, this one is just like that one.")
- *Comparing* (by a human, an algorithm or some other device). For example, a person might say that one specification item is more clearly written than another.
- *Timing* (by computer, by stopwatch, or by some external automated device, or by calculating a difference between two timestamps). For example, measure the time until a specified event (time to first failure), time between events, or time required to complete a task.

A metric might be expressed as a formula involving more than one variable, such as *Defect Removal Efficiency*, (DRE) which is often computed as the ratio of defects found during development to total defects (including ones found in the field). Pfanzagl makes a point about these measures, with which we agree:

> The author doubts whether it is reasonable to consider "derived measurement" as measurement at all. Of course, we can consider any meaningful function as a scale for a property which is defined by this scale [density]. On the other hand, if the property allegedly measured by this derived scale has an empirical meaning by its own, it would also have its own fundamental scale. The function used to define the derived scale then becomes an empirical law stating the relation between fundamental scales. [10, p. 31]

We *can* assign a number to DRE by calculating the ratio, but we could measure it in other ways too. For example, a customer service manager might have enough experience with several workgroups to rank (compare) their defect removal efficiencies, without even thinking about any ratios.

7) **What is the natural scale for this metric?** [7]. The scale of the underlying attribute can differ from the scale of the metric. For example, we're not sure what the natural scale would be for "thoroughness of testing," but suppose we measured thoroughness by giving an expert access to the testing artifacts of several programs and then asked the expert to compare the testing efforts and rank them from least thorough to most thorough. No matter what the attribute's underlying scale, the metric's scale is ordinal.

8) **What is the natural variability of readings from this instrument?** This is normally studied in terms of "measurement error."

9) **What is the relationship of the attribute to the metric value?** This is the *construct validity* question. How do we know that this metric measures that attribute?

   A different way to ask this question is: **What model relates the value of the attribute to the value of the metric?** If the value of the attribute increases by 20%, why should we expect the measured value to increase and by how much?

10) **What are the natural and foreseeable side effects of using this instrument?** If we change our circumstances or behavior in order to improve the measured result, what impact are we going to have on the attribute? Will a 20% increase in our measurement imply a 20% improvement in the underlying attribute? Austin [2] provides several examples in which the work group changed its behavior in a way that optimized a measured result but without improving the underlying attribute at all. Sometimes, the measured result looked better, while the underlying performance that was allegedly being measured was actually worse. Hoffman [21] described several specific side effects that he saw while consulting to software companies.

   - A measurement system yields *distortion* if it creates incentives for the employee to allocate his time so as to make the measurements look better rather than to optimize for achieving the organization's actual goals for his work.
   - The system is *dysfunctional* if optimizing for measurement so distorts the employee's behavior that he provides less value to the organization than he would have provided in the absence of measurement. [2]

3.3 Applying the Evaluation Framework

We have room in this article to illustrate the application of the framework to one metric. We chose bug counts because they are ubiquitous. For example, in Mad About Measurement, Tom DeMarco says: "I can only think of one metric that is worth collecting now and forever: defect count." [19, p. 15] Despite its popularity, there are serious problems with many (not all) of the uses of bug counts. Let's take a look.

1) **What is the purpose of this measure?** Bug counts have been used for a variety of purposes, including:

   - Private, personal discovery by programmers of patterns in the mistakes they make. [22]
   - Evaluation (by managers) of the work of testers (better testers allegedly find more bugs) and programmers (better programmers allegedly make fewer bugs).[23]
   - Evaluation of product status and prediction of release date. [24] [25]
   - Estimation of reliability of the product. [26]

2) **What is the scope of this measure?** Bug statistics have been used within and across projects and workgroups.

3) **What attribute are we trying to measure?** In the field, we've seen bug counts used as surrogates for quality of the product, effectiveness of testing, thoroughness of testing, effectiveness of the tester, skill or diligence of the programmer, reliability of the product, status of the project, readiness for release, effectiveness of a given test technique, customer satisfaction, even (in litigation) the negligence or lack of integrity of the development company.

   In this paper, we narrow the discussion to two attributes, that are popularly "measured" with bug counts.

   - ***Quality (skill, effectiveness, efficiency, productivity, diligence, courage, credibility) of the tester.*** We are trying to measure how "good" this tester is. The notion underlying the bug-count metric is that better testers find more bugs. Some companies attach significant weights to bug counts, awarding bonuses on the basis of them or weighting them heavily in discussions of promotions or raises. However, when we think in terms of defining the attribute, we ignore the proposed metric and keep our focus on what we know about the attribute. One way to think about the attribute is to list adjectives that feel like components or dimensions of it. Some of the aspects of "goodness" of a tester employee are
     - *Skill*—how well she does the tasks that she does. If we think of bug-hunting skill, we might consider whether the bugs found required particularly creative or technically challenging efforts),
     - *Effectiveness*—the extent to which the tester achieves the objective of the work. For example, "The best tester isn't the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed." [27, p. 15]
     - *Efficiency*—how well the tester uses time. Achievement of results with a minimum waste of time and effort.
     - *Productivity*—how much the tester delivers per unit time. The distinction that one can draw between efficiency and productivity is that efficiency refers to the way the person does the job whereas productivity refers to what she gets done. For example, a tester who works on a portion of the code that contains no defects can work through the tests efficiently but produce no bug reports.
     - *Diligence*—how carefully and how hard the tester does her work.
     - *Courage*—willing to attempt difficult and risky tasks; willing to honestly report findings that key stakeholders would prefer to see suppressed.
     - *Credibility*—the extent to which others trust the reports and commitments of this tester.

     A different way to think about the attribute is to consider the services that the tester provides, and then evaluate the quality of performance of each service. Thinking this way, testers provide test automation design and coding, test project planning, test case design and documentation, coaching customer support staff, technical accuracy editing of documentation, status reporting, configuration management (of test artifacts, and often of the entire project's artifacts), laboratory design and workflow management (this is critical if the product must be tested on many configurations), specification analysis, in-

specting code, and, of course, hunting bugs and persuasively reporting the bugs that are found. Some testers provide all of their value to the project by enabling others to find bugs rather than finding bugs themselves.

- ***Status of the project and readiness for release.*** One of the key release criteria for a project is an acceptably low count of significant, unfixed bugs. It is common, over the course of the project, for testers to find a few bugs at the start (while they're getting oriented), then lots of bugs, then fewer and fewer as the program stabilizes. The pattern is common enough that bug curves—graphs showing how many new bugs were found week by week, or how many bugs are unresolved week by week, or some other weekly variant—are in common use in the field.

  As with quality of the tester, however, when we are defining the attribute, the hypotheses about how to measure it are—for the moment—irrelevant. Once we have a better idea of what it is that we are trying to measure, we can look again at the proposed metric to assess the extent to which the metric covers the attribute.

  A project is complete enough to release when it provides enough of the features, delivers enough of the benefits (the features have to work well enough together for the user to actually succeed in using the product to get real work done), is documented well enough for the user, validated well enough for regulators or other stakeholders (e.g. litigators of the future) who have a legitimate interest in the validation, has been sufficiently instrumented, documented, and troubleshot to be ready for field or phone support, is sufficiently ready for maintenance, localization or porting to the next environment (readiness might include having maintainability features in the code as well as effective version control and other maintainability-enhancing development processes in place), is acceptable to the key stakeholders, and has few enough bugs. This list is not exhaustive, but it illustrates the multidimensionality of the release decision. Many companies appraise status and make release decisions in the context of project team meetings, with representatives of all of the different workgroups involved in the project. They wouldn't need these team meetings if the status and release information were one-dimensional (bug counts). We describe these dimensions in the language of "good enough" because projects differ in their fluidity. One organization might insist on coding everything agreed to in a requirements specification but do little or nothing to enable later modification. Another might emphasize high reliability and be willing to release a product with fewer than the desired number of features so long as the ones that are included all work well. Even if we restrict our focus to bugs, the critical question is not how many bugs are in the product, nor how many bugs can be found in testing but is instead how reliable the product will be in the field [15], for example how many bugs will be encountered in the field, how often, by how many people, and how costly they will be.

4) **What is the natural scale of the attribute we are trying to measure?** We have no knowledge of the natural scales of either of these attributes.

5) **What is the natural variability of the attribute?** We have no knowledge of the variability, but there is variability in anything that involves human performance.

6) **What is the metric (the function that assigns a value to the attribute)?** What measuring instrument do we use to perform the measurement?

   - ***Quality (skill, effectiveness, efficiency, productivity) of the tester.*** The proposed metric is some variation of bug count. We might adjust the counts by weighting more serious bugs more heavily. We might report this number as bugs per unit time (such as bugs per week or per month). Whatever the variation, the idea is that more bugs indicate better testing (and fewer bugs indicate worse testing).
   - ***Status of the project and readiness for release.*** The metric is typically expressed as a curve or table that shows bug counts per unit time (typically bugs per week). The "bug counts" might include all open (not-yet-fixed) bugs or only bugs found this week. The counts might be filtered to exclude trivial problems or suggestions that are clearly intended to be confronted in the next release, not this one. One challenging question is whether some bugs are weaker indicators than others. A bug that will take 5 minutes to fix has a very different impact on project status than one that will require a week of troubleshooting and experimentation.

7) ***What is the natural scale for this metric?*** In both cases, we're counting something. That suggests that the scale is interval or ratio. But before we can agree that the scale is one of those, we have to apply some acid tests:

   - ***Ratio scale.*** Bug count is a ratio-scaled measure of tester quality if double the bug count implies that the tester is twice as good.
   - ***Interval scale.*** Suppose that W, X, Y, and Z are four testers, who found N(W) < N(X) < N(Y) < N(Z) bugs. Bug count is an interval-scaled measure of tester quality if the equality: (N(Z)-N(Y) = N(X)-N(W)) implies that Z is as much better a tester than Y as X is better than W, for all bug counts. This if Z found 1000 bugs and Y found 950, Z is as much better than Y as X (who found 51 bugs) is than W (who found 1).

   If neither of these relationships holds, then, *as a measure of tester quality*, bug counts must be ordinal measures.

8) **What is the natural variability of readings from this instrument?** Counting bugs is not perfectly deterministic. Bugs are dropped from the count for many reasons, such as being a duplicate of another report, or reflecting a user error, or not serious enough to pass an agreed threshold. Humans make these decisions, and different humans will sometimes make different decisions. This is an example of a source of variation of the bug counts. There are undoubt-

edly other sources of variation.

9) **What is the relationship of the attribute to the metric value?** Now that we have more clearly described the attributes we're trying to measure, we're in a better position to ask whether or to what degree the metric actually measures the attribute. It seems self-evident that these are surrogate measures.

> "Many of the attributes we wish to study do not have generally agreed methods of measurement. To overcome the lack of a measure for an attribute, some factor which can be measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures." [28]

Surrogate measures provide unambiguous assignments of numbers according to rules, but they don't provide an underlying theory or model that relates the measure to the attribute allegedly being measured.

Interestingly, models have been proposed to tie bug curves to project status. We will focus on one model, recently summarized lucidly by Erik Simmons. [24] Simmons reports successful applications of this model at Intel [24] [29], and references his work back to Lyu. In sum, Simmons plots the time of reporting of medium and high severity bugs, fits the curve to a Weibull distribution and estimates its two parameters, the shape parameter and the characteristic life. From characteristic life, he predicts the total duration of the testing phase of the project.

Even though the curve-fitting and estimation appear successful, it is important to assess the assumptions of the model. An invalid model predicts nothing. According to Simmons, the following assumptions underlie the model:

> 1. The rate of defect detection is proportional to the current defect content of the software.
> 2. The rate of defect detection remains constant over the intervals between defect arrivals.
> 3. Defects are corrected instantaneously, without introducing additional defects.
> 4. Testing occurs in a way that is similar to the way the software will be operated.
> 5. All defects are equally likely to be encountered.
> 6. All defects are independent.
> 7. There is a fixed, finite number of defects in the software at the start of testing.
> 8. The time to arrival of a defect follows a Weibull distribution.
> 9. The number of defects detected in a testing interval is independent of the number detected in other testing intervals for any finite collection of intervals.
>
> These assumptions are often violated in the realm of software testing. Despite such violations, the robustness of the Weibull distribution allows good results to be obtained under most circumstances." [24, p. 4]

These assumptions are not just "often violated." They are blatently incorrect:

- *Detection rate proportional to current defect content:* Some bugs are inherently harder to expose than others. For example, memory leaks, other memory corruption, or timing faults might require long testing sequences to expose. [30] Additionally, it is common practice for test groups to change test techniques as the program gets more stable, moving from simple tests of one variable to complex tests that involve many variables. [31, 32]
- *Rate of defect detection remains constant.* Whenever we change test techniques, introduce new staff, or focus on a new part of the program or a new risk, the defect detection rate is likely to change.
- *Instant, correct defect correction*. If this was true, no one would do regression testing and automated regression test tools wouldn't be so enormously popular.
- *Test similar to use.* This reflects one approach to testing, testing according to the operational profile. [15] However, many test groups reject this philosophy, prefering to test the program harshly, with tests intended to expose defects rather than with tests intended to simulate normal use. The most popular mainstream test technique, domain testing, uses extreme (rather than representative) values. [33] Risk-based testing also hammers the program at anticipated vulnerabilities, without reference to operational profile. [34]
- *All defects equally likely to be encountered.* This is fundamentally implausible. Some bugs crash the program when you boot it or corrupt the display of the opening screen. Other bugs, such as wild pointer errors and race conditions, are often subtle, hard to expose, and hard to replicate.
- *All defects are independent*. Bugs often mask other bugs.
- *Fixed, finite number of defects in the software at the start of testing.* There is a trivial sense in which these words are true. If we fix any point in time and identify all of the code in a product, that codebase must have, for that moment, a fixed total number of bugs. However, the meaning behing the words is the assertion that the totla *stays fixed* after the start of testing. That is, bug fixes could introduce no new bugs. No new code could be added to the product after the start of testing or all of it would be perfect. Requirements would never change after the start of testing and changed external circumstances would never render any previously good code incompatible or incomplete. We have never seen a project for which this was close to true.
- *Time to arrival follows a Weibull distribution.* There is nothing theoretically impossible about this, but the assumptions that provided a rationale for deriving a Weibull process (listed above) have failed, so it might be surprising if the distribution were Weibull.
- *Number of defects detected in one interval independent of number detected in others*. Again, the rate of detection depends on other variables such as selection of test technique or introduction of new testers or the timing of vacations and corporate reorganizations.

These assumptions are not merely sometimes violated. They individually and collectively fail to describe what happens in software testing. The Weibull distribution is right-skewed (more bugs get found early than near the ship

date) and unimodal, and that pattern might be common in testing, but there are plenty of right-skew distributions, and they arise from plenty of different causes. The Weibull distribution is not a plausible model of project status or project testing phase duration.

10) **What are the natural and foreseeable side effects of using this instrument?** People are good at tailoring their behavior to things that they are measured against. [35] If we want more bugs, we can get more bugs. If we want a nice, right-skew curve, we can get that curve. But the pretty new numbers doesn't necessarily imply that we'll get the improvements in the underlying attribute that we're looking for. The less tightly linked a measure is to the underlying attribute, the more we expect to see distortion and disfunction when the measure is used. [2]

- ***Quality (skill, effectiveness, efficiency, productivity) of the tester.*** Measuring testers by their bug count will encourage them to report more bugs.
  - This creates incentives for superficial testing (test cases that are quick and easy to create) and against deep tests for serious underlying errors. Bug counting punishes testers who take the time to look for the harder-to-find but more important bugs.
  - The system creates disincentives for supporting other testers. It takes time to coach another tester, to audit his work, or to help him build a tool that will make him more effective, time that is no longer available for the helper-tester to use to find bugs.
  - More generally, emphasizing bug counts penalizes testers for writing test documentation, researching the bugs they find to make more effective bug reports, or following any process that doesn't yield more bugs quickly.
  - The system also creates political problems. A manager can make a tester look brilliant by assigning a target-rich area for testing. Similarly, a manager can set up a disfavored tester for firing by having him test stable areas or areas that require substantial setup time per test. As another political issue, programmers will know that testers are under pressure to maximize their bug counts, and may respond cynically to bug reports, dismissing them as chaff filed to increase the bug count rather than good faith reports. Hoffman [21] provides further illustrations of political bug count side effects.

  Problems like these have caused several measurement advocates to warn against measurement of attributes of individuals (e.g., [36]) unless the measurement is being done for the benefit of the individual (for genuine coaching or for discovery of trends) and otherwise kept private (e.g. [19] [22]). Often, people advocate using aggregate counts--but any time you count the output of a group and call that "productivity", you are making a personal measurement of the performance of the group's manager.

- ***Status of the project and readiness for release.*** We can expect the following problems (side effects) from reliance on bug curves. Some of these were reported by Hoffman [21]. Kaner has seen most of these at client sites.
  - Early in testing, the pressure is to build up the bug count. If we hit an early peak, the model says we'll finish sooner. One way to build volume is to run every test onhand, even tests of features that are already known to be broken or incomplete. Each seemingly-new way the program fails is good for another bug report. Another way to build volume is to chase variants of bugs—on finding a bug, create several related tests to find more failures. Some follow-up testing is useful, but there's a point at which it's time to pass the reports to the programmers and let them clear out the underlying fault(s) before looking for yet more implications of what is likely the same fault. In general, testers will look for easy bugs in high quantities and will put less emphasis on automation architecture, tool development, test documentation, or other infrastructure development. This has a dual payoff. The testers find lots of bugs over the immediate term, when they are under pressure to find lots of bugs, and they don't build support for a sustained attack on the product, so later, when the easiest bugs are out of the system, the bug find rate will plummet just like the model says it should.
  - Later in testing, the expectation is that the bug find rate will decline. Testers have permission to find fewer bugs, and they may run into a lot of upset if they sustain a solid bug-find rate late in the project. As a result, they're less likely to look for new bugs. Instead, they can rerun lots of already-run regression tests—tests that the program has passed time and again and will probably pass time and again in the future. [37] Later in the project, testers can spent lots of time writing status reports, customer support manuals, and other documents that offer value to the company—but not bugs. Programmers and project managers under pressure to keep up with the bug curve have also aggressively managed the bug database by closing lightly-related bugs as duplicates, rejecting a higher portion of bugs as user errors or design requests, closing hard-to-reproduce bugs as irreproducible rather than making an effort to replicate them, or finding ways to distract the testers (such as sending them to training sessions or even to the movies!) In some companies, the testers and the programmers hold the "quality assurance" metrics-gathering staff in contempt and they collaborate to give the QA outsiders the numbers they want in order to get them to go back to Head Office, far away. This includes slowing down testing before major milestones (so that the milestones, which are defined partially in terms of the predicted bug cure, can be recorded as met) and reporting bugs informally and not entering them into the bug tracking system until the programmer is ready to enter a fix. At one client site, the staff even had a cubicle where they would write

bugs up on Post-It notes, posting them on the inside wall until a bug was fixed or the numbers in the tracking system were low enough to admit more new bugs. This system worked fairly well except when Post-Its fell off the wall at night and were swept away by the janitor.

Rather than accepting the smooth decline in bug find rate, some test managers treat a drop in the bug count as a trigger for change. They adopt new test techniques, re-analyze the product for new risks, focus on less-tested areas, bring on staff with other skills, and try to push the bug count back up. Eventually, the testers run out of good ideas and the new-bugs-found rate drops dramatically. But until then, the testers are fighting against the idea that they should find fewer bugs, rather than collaborating with it.

4 A More Qualitative Approach to Qualitative Attributes

Rather than fighting the complexity of software engineering attributes, it might make sense to embrace them. These notes are based on work done at two meetings of experienced test managers (the Software Test Managers' Roundtables), interviews by Cem Kaner of test managers, and extensive work by Kaner and some of his consulting clients on improving the effectiveness of their bug reporting. The bug reporting notes have also been refined through use in classroom instruction [38] and course assignments based on the notes, and in peer critiques of previous presentations, such as [39]. The test planning notes are more rough, but an earlier version has been published and critiqued.[40] We summarize those notes here.

The notion of measuring several related dimensions to get a more complete and balanced picture is not new. The balanced scorecard approach [41] [42] developed as a reaction to the inherently misleading information and dysfunction resulting from single-dimensional measurement. We also see multidimensional work done in software engineering, such as [3] and [43]. What we add here (in this section and in several of the analyses above) are primarily examples of breakdowns of some software engineering attributes or tasks into a collection of related sub-attributes.

Imagine being a test manager and trying to evaluate the performance of your staff. They do a variety of tasks, such as bug-hunting, bug reporting, test planning, and test tool development. To fully evaluate the work of the tester, you would evaluate the quality of work on each of the tasks.

Consider the bug reporting task. Take a sample of the reports to evaluate them. Start by skimming a report to form a first impression of it.

- Is the summary short (about 50-70 characters) and descriptive?
- Can you understand the report? Do you understand what the reporter did and what the program did in response?
- Do you understand what the failure was?
- Is it obvious where to start (what state to bring the program to) to replicate the bug? What files to use (if any)? What to type?
- Is the replication sequence provided as a numbered set of steps, that state exactly what to do and, when useful, what you will see?
- Does the report include unnecessary information, personal opinions or anecdotes that seem out of place?
- Is the tone of the report insulting? Are any words in the report potentially insulting?
- Does the report seem too long? Too short? Does it seem to have a lot of unnecessary steps?

Next, try to replicate the bug.

- Can you replicate the bug? Did you need additional information or steps? Did you have to guess about what to do next?
- Did you get lost or wonder whether you had done a step correctly? Would additional feedback (like, “the program will respond like this...”) have helped?
- Did you have to change your configuration or environment in any way that wasn’t specified in the report?
- Did some steps appear unnecessary? Were they unnecessary?
- Did the description accurately describe the failure?
- Did the summary accurate describe the failure?
- Does the description include non-factual information (such as the tester’s guesses about the underlying fault) and if so, does this information seem credible and useful or not?

Finally, make a closing evaluation:

- Should the tester have done further troubleshooting to try to narrow the steps in the bug or to determine whether different conditions would yield worse symptoms?
- Does the description include non-factual information (such as the tester’s guesses about the underlying fault). Should it? If it does, does this information seem credible and useful?
- Does the description include statements about why this bug would be important to the customer or to someone else? Should it? If it does, are the statements credible?

Along with using a list like this for your evaluation, you can hand it out to your staff as a guide to your standards.

Evaluating test plans is more challenging, especially in a company that doesn't have detailed test planning standards. Your first task is to figure out what the tester's standards are. For example, what is the tester's theory of the objectives of testing for this project? Once you know that, you can ask whether the specific plan that you're reviewing describes those objectives clearly and achieves them. Similarly, we considered the tester's theory of scope of testing, coverage, risks to manage, data (what data should be covered and in what depth), originality (extent to which this plan should add new tests to an existing collection, and why), communication (who will read the test artifacts and why), usefulness of the test artifacts (who will use each and for what purposes), completeness (how much testing and test documentation is good enough?) and insight (how the plan conveys the underlying ideas). The test planner has to decide for each of these dimensions how much is enough—more is not necessarily better.

In considering these dimensions, we've started experimenting with rubrics. [44] [45] A rubric is a table. There's a row for each dimension (objective, scope, coverage, etc.). There are 3 to 5 columns, running from a column that describes weak performance through a mid-level that describes acceptable but not spectacular work, through a column that describes excellent work. By describing your vision of what constitutes excellent, adequate, and poor work, you give your staff a basis for doing what you want done.

The basic rubric works excellently when you are in full control of the standards. However, it is more subtle when you leave the decisions about standards to the staff and then evaluate their work in terms of their objectives. Opinions vary as to the extent to which staff should be allowed to set their own standards, but there is a severe risk of mediocrity if the tester's (or any skilled professional's) work is micromanaged.

After you have reviewed several bug reports (or test plans) using the bug reporting checklist (or test plan rubric), you will form an opinion of the overall quality of work of this type that a given tester is doing. That will help you rate the work (ordinal scale). For example, you might conclude that the tester is Excellent at test planning but only Acceptable at bug report writing.

The set of ratings, across the different types of tasks that testers do, can provide a clear feedback loop between the tester and the test manager.

To convey an overall impression of the tester's strength, you might draw a Kiveat diagram or some other diagram that conveys the evaluator's reading on each type of task.

We have not seen this type of evaluation fully implemented and don't know of anyone who has fully implemented it. A group of test managers has been developing this approach for their use, and many of them are now experimenting with it, to the extent that they can in their jobs.

Our intuition is that there are some challenging tradeoffs. The goal of this approach is not to micromanage the details of the tester's job, but to help the test manager and the tester understand which tasks the tester is doing well and which not. There are usually many ways to do a task well. If the scoring structure doesn't allow for this diversity, we predict a dysfunction-due-to-measurement result.

5 Conclusion

There are too many simplistic metrics that don't capture the essence of whatever it is that they are supposed to measure. There are too many uses of simplistic measures that don't even recognize what attributes are supposedly being measured. Starting from a detailed analysis of the task or attribute under study might lead to more complex, and more qualitative, metrics, but we believe that it will also leads to more meaningful and therefore more useful data.

Acknowledgment

Some of the material in this paper was presented or developed by the participants of the *Software Test Managers Roundtable* (STMR) and the *Los Altos Workshop on Software Testing* (LAWST).

LAWST 8 (December 4-5, 1999) focused on *Measurement*. Participants included Chris Agruss, James Bach, Jaya Carl, Rochelle Grober, Payson Hall, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Nguyen, Bret Pettichord, Melora Svoboda, and Scott Vernon.

STMR 2 (April 30, May 1, 2000) focused on the topic, *Measuring the extent of testing*. Participants included James Bach, Jim Bampos, Bernie Berger, Jennifer Brock, Dorothy Graham, George Hamblen, Kathy Iberle, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, and Steve Tolman.

STMR 8 (may 11-12, 2003) focused on *measuring the performance of individual testers*. Participants included Bernie Berger, Ross Collard, Kathy Iberle, Cem Kaner, Nancy Landau, Erik Petersen, Dave Rabinek, Jennifer Smith-Brock, Sid Snook, and Neil Thompson.

References

[1] N. E. Fenton, "Software Metrics: Successes, Failures & New Directions," presented at ASM 99: Applications of Software Measurement, San Jose, CA, 1999. http://www.stickyminds.com/s.asp?F=S2624_ART_2

[2] R. D. Austin, *Measuring and Managing Performance in Organizations*. New York: Dorset House Publishing, 1996.

[3] L. Buglione and A. Abran, "Multidimensionality in Software Performance Measurement: the QEST/LIME Models," presented at SSGRR2001 - 2nd International Conference in Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, L'Aquila, Italy, 2001. http://www.lrgl.uqam.ca/publications/pdf/722.pdf

[4] S. S. Stevens, "On the Theory of Scales of Measurement," *Science*, vol. 103, pp. 677-680, 1946.

[5] S. S. Stevens, *Psychophysics: Introduction to its Perceptual, Neural, and Social Prospects*. New York: John Wiley & Sons, 1975.

[6] L. Finkelstein, "Theory and Philosophy of Measurement," in *Theoretical Fundamentals*, vol. 1, *Handbook of Measurement Science*, P. H. Sydenham, Ed. Chichester: John Wiley & Sons, 1982, pp. 1-30.

[7] [7] N. E. Fenton and S. L. Pfleeger, "Software Metrics: A Rigorous and Practical Approach," 2nd Edition Revised ed. Boston: PWS Publishing, 1997.

[8] IEEE, "IEEE Std. 1061-1998, Standard for a Software Quality Metrics Methodology, revision." Piscataway, NJ,: IEEE Standards Dept., 1998.

[9] W. Torgerson, S., *Theory and Methods of Scaling*. New York: John Wiley & Sons, 1958.

[10] J. Pfanzagl, "Theory of Measurement," 2nd Revised ed. Wurzburg: Physica-Verlag, 1971.

[11] D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky, *Foundations of Measurement*, vol. 1. New York: Academic Press, 1971.

[12] H. Zuse, *A Framework of Software Measurement*. Berlin: Walter de Gruyter, 1998.

[13] S. Morasca and L. Briand, "Towards a Theoretical Framework for Measuring Software Attributes," presented at 4th International Software Metrics Symposium (METRICS '97), Albuquerque, NM, 1997.

[14] N. Campbell, *An Account of the Principles of Measurement and Calculation*. London: Longmans, Green, 1928.

[15] J. Musa, *Software Reliability Engineering*. New York: McGraw-Hill, 1999.

[16] D. S. Le Vie, "Documentation Metrics: What do You *Really* Want to Measure?," *STC Intercom*, pp. 7-9, 2000.

[17] N. E. Fenton and A. Melton, "Measurement Theory and Software Measurement," in *Software Measurement*, A. Melton, Ed. London: International Thomson Computer Press, 1996, pp. 27-38.

[18] J. Michell, *Measurement in Psychology: A Critical History of a Methodological Concept*. Cambridge: Cambridge University Press, 1999.

[19] T. DeMarco, "Mad About Measurement," in *Why Does Software Cost So Much?* New York: Dorset House, 1995, pp. 13-44.

[20] C. G. Hempel, *Fundamentals of Concept Formation in Empirical Science: International Encyclopedia of Unified Science*, vol. 2. Chicago: University of Chicago Press, 1952.
[21] D. Hoffman, "The Darker Side of Metrics," presented at Pacific Northwest Software Quality Conference, Portland, Oregon, 2000. http://www.softwarequalitymethods.com/SQM/Papers/DarkerSideMetricsPaper.pdf
[22] W. Humphrey, "Introduction to the Personal Software Process." Boston: Addison-Wesley, 1996.
[23] C. Kaner, "Don't Use Bug Counts to Measure Testers," in *Software Testing & Quality Engineering*, 1999, pp. 79-80. http://www.kaner.com/pdfs/bugcount.pdf.
[24] E. Simmons, "When Will We be Done Testing? Software Defect Arrival Modeling Using the Weibull Distribution," presented at Pacific Northwest Software Quality Conference, Portland, OR, 2000. http://www.pnsqc.org
[25] S. H. Kan, J. Parrish, and D. Manlove, "In-Process Metrics for Software Testing," *IBM Systems Journal*, vol. 40, pp. 220 ff, 2001. http://www.research.ibm.com/journal/sj/401/kan.html.
[26] S. Brocklehurst and B. Littlewood, "New Ways to Get Accurate Reliability Measures," *IEEE Software*, vol. 9, pp. 34-42, 1992.
[27] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, 2 ed. New York: John Wiley & Sons, 1999.
[28] M. A. Johnson, "Effective and Appropriate Use of Controlled Experimentation in Software Development Research," in *Computer Science*. Portland: Portland State University, 1996.
[29] E. Simmons, "Defect Arrival Modeling Using the Weibull Distribution," presented at International Software Quality Week, San Francisco, CA, 2002.
[30] C. Kaner, W. P. Bond, and P. J. McGee, "High Volume Test Automation (Keynote Address)," presented at International Conference for Software Testing Analysis & Review (STAR East), Orlando, FL, 2004. http://www.testingeducation.org/articles/KanerBondMcGeeSTAREAST_HVTA.pdf
[31] Kaner, "What is a Good Test Case?," presented at International Conference for Software Testing Analysis & Review (STAR East), Orlando, FL, 2003. http://www.kaner.com/pdfs/GoodTest.pdf
[32] C. Kaner, "The Power of 'What If ... and Nine Ways to Fuel Your Imagination: Cem Kaner on Scenario Testing," in *Software Testing and Quality Engineering*, vol. 5, 2003, pp. 16-22.
[33] C. Kaner, "Teaching Domain Testing: A Status Report," presented at 17th Conference on Software Engineering Education and Training, Norfolk, VA, 2004.
[34] J. Whittaker, *How to Break Software*. Boston: Addison-Wesley, 2002.
[35] G. Weinberg and E. L. Schulman, "Goals and Performance in Computer Programming," *Human Factors*, vol. 16, pp. 70-77, 1974.
[36] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
[37] C. Kaner, "Avoiding Shelfware: A Manager's View of Automated GUI Testing," presented at International Conference for Software Testing Analysis and Review, Orlando, FL, 1998. http://www.kaner.com/pdfs/shelfwar.pdf
[38] C. Kaner and J. Bach, "Editing Bugs," in *A Course in Black Box Software Testing: 2004 Academic Edition*. Melbourne, FL: Florida Tech Center for Software Testing Education and Research, 2004. http://www.testingeducation.org/k04/bbst28_2004.pdf.
[39] C. Kaner, "Measuring the Effectiveness of Software Testers," presented at International Conference for Software Testing Analysis and Review (STAR East), Orlando, FL, 2003. http://www.testingeducation.org/articles/performance_measurement_star_east_2003_presentation.pdf
[40] B. Berger, "Evaluating Test Plans Using Rubrics," presented at International Conference for Software Testing Analysis and Review (STAR East), Orlando, FL, 2004.
[41] R. S. Kaplan and D. P. Norton, *The Balanced Scorecard*. Boston: Harvard University Press, 1996.
[42] N.-G. Olve, J. Roy, and M. Wetter, *Performance Drivers: A Practical Guide to Using the Balanced Scorecard*. Chichester: John Wiley & Sons, 1999.
[43] A. Abran and L. Buglione, "A Multidimensional Performance Model for Consolidating Balanced Scorecards," *Advances in Engineering Software*, vol. 34, pp. 339-349, 2003. http://www.lrgl.uqam.ca/publications/pdf/740.pdf.
[44] J. Arter and J. McTighe, *Scoring Rubrics in the Classroom*. Thousand Oaks, CA: Corwin Press, 2001.
[45] G. Taggart, L., S. J. Phifer, J. A. Nixon, and M. Wood, *Rubrics: A Handbook for Construction and Use*. Latham, MA: Scarecrow Press, 1998.

Cem Kaner, B.A. (Interdisciplinary, primarily Mathematics & Philosophy, 1974); Ph.D. (Experimental Psychology; dissertation on the measurement of the perception of time, 1984); J.D. (1994). Industry employment (Silicon Valley, 1983-2000) included WordStar, Electronic Arts, Telenova, Power Up Software, Psylomar, and kaner.com (a consulting firm with a wide range of clients). Positions included software engineer, human factors analyst, tester, test manager, documentation group manager, software development manager, development director, and principal consultant. Legal employment included Santa Clara County Office of the District Attorney and the Law Office of Cem Kaner. Currently Professor of Software Engineering and Director of the Center for Software Testing Education at the Florida Institute of Technology. Kaner is the Program Chair of the 2004 Workshop on Website Evolution, Editor of the Journal of the (recently formed) Association for Software Testing, and co-founder of the Los Altos Workshops on Software Testing. His current research and teaching areas include software testing, computer science education, software metrics, and the law of software quality.

Walter Bond, B.A. (Mathematics, 1963), M.S. (Mathematics, 1968), Ph.D. (Mathematical Statistics, dissertation on the use of interative graphics for statistical computation, 1976). Industry employment: over 35 years of industrial experience in software engineering and the application of statistical methodology. Positions include: Manager of the Computer Services Department at the Kennedy Space Center; Senior Manager, Quantitative Analysis in the Engineering Productivity Group of the Quality and New Processes Department of Harris Corporation, responsible for process improvements across all disciplines and divisions of the corporation, including the introduction of Six Sigma concepts to Harris operations.. In his current position as Associate Professor of Computer Sciences at the Florida Institute of Technology, he teaches software engineering, software metrics, software design methods, requirements analysis and engineering lifecycle cost estimation. His research areas include software metrics, software reliability modeling, and the assessment of software architecture. He is past president of the Florida Chapter of the American Statistical Association.