Software Measurement: A Necessary Scientific Basis

Norman Fenton

***Abstract*— Software measurement, like measurement in any other discipline, must adhere to the science of measurement if it is to gain widespread acceptance and validity. The observation of some very simple, but fundamental, principles of measurement can have an extremely beneficial effect on the subject. Measurement theory is used to highlight both weaknesses and strengths of software metrics work, including work on metrics validation. We identify a problem with the well-known Weyuker properties, but also show that a criticism of these properties by Cherniavsky and Smith is invalid. We show that the search for general software complexity measures is doomed to failure. However, the theory does help us to define and validate measures of specific complexity attributes. Above all, we are able to view software measurement in a very wide perspective, rationalising and relating its many diverse activities.**

***Index Terms*—Software measurement, empirical studies, metrics, measurement theory, complexity, validation.**

I. Introduction

IT IS over eleven years since DeMillo and Lipton outlined the relevance of measurement theory to software metrics [10]. More recent work by the author and others [4], [11], [46] has taken the measurement theory basis for software metrics considerably further. However, despite the important message in this work, and related material (such as [31], [34], [43], [38]), it has been largely ignored by both practitioners and researchers. The result is that much published work in software metrics is theoretically flawed. This paper therefore provides a timely summary and enhancement of measurement theory approaches, which enables us to expose problems in software metrics work and show how they can be avoided.

In Section II, we provide a concise summary of measurement theory. In Section III, we use the theory to show that the search for general-purpose, real-valued software 'complexity' measures is doomed to failure. The assumption that fundamentally different views of complexity can be characterised by a single number is counter to the fundamental concepts of measurement theory. This leads us to re-examine critically the much cited Weyuker properties [45]. We explain how the most promising approach is to identify specific attributes of complexity and measure these separately. In Section IV, we use basic notions of measurement to describe a framework which enables us to view apparently diverse software measurement activities in a unified way. We look at some well-known approaches to software measurement within this framework, exposing both the good points and bad points.

Manuscript received August 1992; revised September 1993. This work was supported in part by IED project SMARTIE and ESPRIT project PDCS2.

The author is with the Centre for Software Reliability, City University, London, EC1V OHB, UK.

IEEE Log Number 9215569.

II. Measurement Fundamentals

In this section, we provide a summary of the key concepts from the science of measurement which are relevant to software metrics. First, we define the fundamental notions (which are generally not well understood) and then we summarise the representational theory of measurement. Finally, we explain how this leads inevitably to a goal-oriented approach.

A. What is Measurement?

Measurement is defined as the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules [13], [36]. An *entity* may be an object, such as a person or a software specification, or an event, such as a journey or the testing phase of a software project. An *attribute* is a feature or property of the entity, such as the height or blood pressure (of a person), the length or functionality (of a specification), the cost (of a journey), or the duration (of the testing phase).

Just what is meant by the numerical assignment "describing" the attribute is made precise within the representational theory of measurement presented below. Informally, the assignment of numbers or symbols must preserve any intuitive and empirical observations about the attributes and entities. Thus, for example, when measuring the height of humans bigger numbers must be assigned to the taller humans, although the numbers themselves will differ according to whether we use metres, inches, or feet. In most situations an attribute, even one as well understood as height of humans, may have a different intuitive meaning to different people. The normal way to get round this problem is to define a *model* for the entities being measured. The model reflects a specific viewpoint. Thus, for example, our model of a human might specify a particular type of posture and whether or not to include hair height or allow shoes to be worn. Once the model is fixed there is a reasonable consensus about relations which hold for humans with respect to height (these are the empirical relations). The need for good models is particularly relevant in software engineering measurement. For example, even as simple a measure of length of programs as lines of code (LOC) requires a well defined model of programs which enables us to identify unique lines unambiguously. Similarly, to measure the effort spent on, say, the unit testing process we would need an agreed "model" of the process which at least makes clear when the process begins and ends.

There are two broad types of measurement: direct and indirect. *Direct measurement* of an attribute is measurement which does not depend on the measurement of any other attribute. *Indirect measurement* of an attribute is measurement which involves the measurement of one or more other attributes. It turns out that while some attributes can be measured directly, we normally get more sophisticated measurement (meaning a more sophisticated scale, see below) if we measure indirectly. For a good discussion of these issues, see [25], [27].

Uses of Measurement: Assessment and Prediction: There are two broad uses of measurement: for *assessment* and for *prediction.* Predictive measurement of an attribute A will generally depend on a mathematical model relating A to some existing measures of attributes $A_1, \cdots, A_n$. Accurate predictive measurement is inevitably dependent on careful (assessment type) measurement of the attributes $A_1, \cdots, A_n$. For example, accurate estimates of project resources are *not* obtained by simply "applying" a cost estimation model with fixed parameters [26]. However, careful measurement of key attributes of completed projects can lead to accurate resource predictions for future projects [22] Similarly, it is possible to get accurate predictions of the reliability of software in operation, but these are dependent on careful data collection relating to failure times during alpha-testing [5].

For predictive measurement the model alone is not sufficient. Additionally, we need to define the procedures for a) determining model parameters and b) interpreting the results. For example, in the case of software reliability prediction we might use maximum likelihood estimation for a) and Bayesian statistics for b). The model, together with procedures a) and b), is called a *prediction system* [29]. Using the same model will generally yield different results if we use different prediction procedures.

It must be stressed that, for all but the most trivial attributes, proposed predictive measures in software engineering are invariably stochastic rather than deterministic. The same is true of proposed indirect measures [14].

Measurement Activities must have Clear Objectives: The basic definitions of measurement suggest that any measurement activity must proceed with very clear objectives or goals. First you need to know whether you want to measure for assessment or for prediction. Next, you need to know exactly which entities are the subject of interest. Then you need to decide which attributes of the chosen entities are the significant ones. The definition of measurement makes clear the need to specify both an entity and an attribute before any measurement can be undertaken (a simple fact which has been ignored in much software metrics activity). Clearly, there are no definitive measures which can be prescribed for every objective in any application area. Yet for many years software practitioners expected precisely that: 'what software metric should we be using?' was, and still is, a commonly asked question. It says something about the previous ignorance of scientific measurement in software engineering that the Goal/Question/Metric paradigm of Basili and Rombach [7] has been hailed as a revolutionary step forward. GQM spells out the above necessary obligations for setting objectives before embarking on any software measurement activity.

B. Representational Theory of Measurement

The Issues Addressed: Although there is no *universally* agreed theory of measurement, most approaches are devoted to resolving the following issues: what is and what is not measurement; which types of attributes can and cannot be measured and on what kind of scales; how do we know if we have really measured an attribute; how to define measurement scales; when is an error margin acceptable or not; which statements about measurement are meaningful. The texts [13], [25], [27], [36], [40] all deal with these issues. Here we present a brief overview of the *representational* theory of measurement [13], [25].

Empirical Relation Systems: Direct measurement of a particular attribute possessed by a set of entities must be preceded by intuitive understanding of that attribute. This intuitive understanding leads to the identification of empirical relations between entities. The set of entities C, together with the set of empirical relations R, is called an *empirical relation system* (C, R) for the attribute in question. Thus the attribute of "height" of people gives rise to empirical relations like "is tall", "taller than", and "much taller than".

Representation Condition: To measure the attribute that is characterised by an empirical relation system (C, R) requires a mapping M into a *numerical relation system* (N, P). Specifically, M maps entities in C to numbers (or symbols) in N, and empirical relations in R are mapped to numerical relations in P, in such a way that all empirical relations are preserved. This is the so-called *representation condition,* and the mapping M is called a *representation.* The representation condition asserts that the correspondence between empirical and numerical relations is two way. Suppose, for example, that the binary relation $\prec$ is mapped by M to the numerical relation $<$. Then, formally, we have the following instance:

Representation Condition: $x \prec y \iff M(x) < M(y)$

Thus, suppose, C is the set of all people and R contains the relation "taller than". A measure M of height would map C into the set of real numbers $\Re$ and 'taller than' to the relation ">". The representation condition asserts that person A is taller than person B, if and only if $M(A) > M(B)$.

By having to identify empirical relations for an attribute in advance, the representational approach to measurement avoids the temptation to *define* a poorly understood, but intuitively recognisable, attribute in terms of some numerical assignment. This is one of the most common failings in software metrics work. Classic examples are where attributes such as "complexity" or "quality" are equated with proposed numbers; for example, complexity with a "measure" like McCabe's cyclomatic number [30], or Halstead's E [18], and "quality" with Kafura and Henry's fan-in/fan-out equation [23].

Scale Types and Meaningfulness: Suppose that an attribute of some set of entities has been characterised by an empirical relation system (C, R). There may in general be many ways of assigning numbers which satisfy the representation condition. For example, if person A is taller than person B, then $M(A) > M(B)$ irrespective of whether the measure M is in inches, feet, centimetres metres, etc. Thus, there are many different measurement representations for the normal empirical relation system for the attribute of height of people. However, any two representations M and M' are related in a very specific way: there is always some constant $c > 0$ such that $M = cM'$

(so where M is the representation of height in inches and M' in centimetres, $c = 2.54$). This transformation from one valid representation into another is called an *admissible transformation.*

It is the class of admissible transformations which determines the *scale type* for an attribute (with respect to some fixed empirical relation system). For example, where every admissible transformation is a scalar multiplication (as for height) the scale type is called *ratio*. The ratio scale is a sophisticated scale of measurement which reflects a very rich empirical relation system. An attribute is never of ratio type *a priori*; we normally start with a crude understanding of an attribute and a means of measuring it. Accumulating data and analysing the results leads to the clarification and re-evaluation of the attribute. This in turn leads to refined and new empirical relations and improvements in the accuracy of the measurement; specifically this is an improved scale.

For many software attributes we are still at the stage of having very crude empirical relation systems. In the case of an attribute like "criticality" of software failures an empirical relation system would at best only identify different classes of failures and a binary relation "is more critical than". In this case, any two representations are related by a monotonically increasing transformation. With this class of admissible transformations, we have an *ordinal* scale type. In increasing order of sophistication, the best known scale types are: *nominal, ordinal, interval, ratio,* and *absolute*. For full details about the defining classes of admissible transformations, see [36].

This formal definition of scale type based on admissible transformations enables us to determine rigorously what kind of statements about measurement are meaningful. Formally, a statement involving measurement is *meaningful* if its truth or falsity remains unchanged under any admissible transformation of the measures involved. Thus, for example, it is meaningful to say that "Hermann is twice as tall as Peter"; if the statement is true (false) when we measure height in inches, it will remain true (false) when we measure height in any constant multiple of inches. On the other hand the statement "Failure x is twice as critical as failure y" is not meaningful if we only have an ordinal scale empirical relation system for failure criticality. This is because a valid ordinal scale measure M could define $M(x) = 6$, $M(y) = 3$, while another valid ordinal scale measure M' could define $M'(x) = 10$, $M'(y) = 9$. In this case the statement is true under M but false under M'.

The notion of meaningfulness also enables us to determine what kind of operations we can perform on different measures. For example, it is meaningful to use *means* for computing the average of a set of data measured on a ratio scale but not on an ordinal scale. *Medians* are meaningful for an ordinal scale but not for a nominal scale. Again, these basic observations have been ignored in many software measurement studies, where a common mistake is to use the mean (rather than median) as measure of average for data which is only ordinal. Good examples of practical applications of meaningfulness ideas may be found in [3], FW86, [37], [39]. An alternative definition of meaningfulness is given in [16].

Representation Theorems: The serious mathematical aspects of measurement theory are largely concerned with theorems which assert conditions under which certain scales of direct measurement are possible for certain relation systems. A typical example of such a theorem, due to Cantor, gives conditions for real-valued ordinal-scale measurement when we have a countable set of entities C and a binary relation b on C:

Cantor's Theorem: The empirical relation system (C, b) has a representation in $(\Re, <)$ if and only if b is a strict weak order. The scale type is ordinal when such a representation exists.

The relation b being a "strict weak order" means that it is:

1) *asymmetric* (xRy implies that it is not the case yRx), and
2) *negatively transitive* (xRy implies that for every $z \in C$, either xRz or zRy).

III. Measuring Software "Complexity"

The representational theory of measurement is especially relevant to the study of software complexity measures. In this section we show that the search for a general-purpose real-valued complexity measure is doomed to failure, but that there are promising axiomatic approaches which help us to measure specific complexity attributes. However, one well-known axiomatic approach [45] has serious weaknesses because it attempts to characterise incompatible views of complexity.

A. General Complexity Measures: The Impossible Holy Grail

For many years researchers have sought to characterise general notions of "complexity" by a single real number. To simplify matters, we first restrict our attention to those measures which attempt only to characterise control-flow complexity. If we can show that it is impossible to define a general measure of control-flow complexity, then the impossibility of even more general complexity measures is certain.

Zuse cites dozens of proposed control-flow complexity measures in [46]. There seems to be a minimum assumption that the empirical relation system for complexity of programs leads to (at least) an ordinal scale. This is because of the following hypotheses which are implicit in much of the work.

Hypothesis 1: Let C be the class of programs. Then the attribute control-flow "complexity" is characterised by an empirical relation system which includes a binary relation b "less complex than"; specifically $(x, y) \in b$ if there is a consensus that x is less complex than y.

Hypothesis 2: The proposed measure $M: C \to \Re$ is a representation of complexity in which the relation b is mapped to $<$.

Hypothesis 1 seems plausible. It does not state that C is totally ordered with respect to b; only that there is some *general* view of complexity for which there would be a reasonable consensus that certain pairs of programs are in b. For example, in Fig. 1, it seems plausible that $(x, y) \in b$ (from the measurement theory viewpoint it would be good enough if most programmers agreed this). Some pairs appear to be

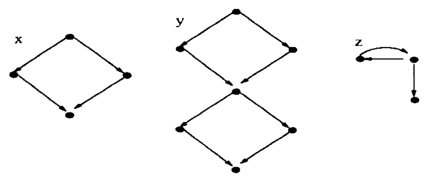


Fig. 1. Complexity relation not negatively transitive?

incomparable, such as x and z or y and z; if people were asked to "rank" these for complexity they would inevitably end up asking questions like "what is meant by complexity" before attempting to answer. Since b is supposed to capture a general view of complexity, this would be enough to deduce that $(x, z) \notin b$ and $(z, x) \notin b$ and also that $(z, y) \notin b$ and $(y, z) \notin b$. The idea of the inevitable incomparability of some programs, even for some specific views of complexity, has also been noted in [43].

Unfortunately, while Hypothesis 1 is plausible, Hypothesis 2 can be dismissed because of the Representation Condition. The problem is the "incomparable" programs. While b is not a total order in C, the relation $<$ *is* a total order in $\Re$. The measurement mapping M might force an order which has to be reflected back in C. Thus, if for example $M(z) < M(y)$ (as in the case of McCabe's cyclomatic complexity measure in Fig. 1 where $M(z) = 2$ and $M(y) = 3$) then, if M is really a measure of complexity, the Representation Condition asserts that we must also have $z < y$ for which there is no consensus.

Formally we can prove the following theorem.

Theorem 1: Assuming Hypothesis 1, there is no general notion of control-flow complexity of programs which can be measured on an ordinal scale in $(\Re, <)$

To prove this, the previous argument is made formal by appealing to Cantor's Theorem. It is enough to show that the relation b is not a strict weak order. This follows since (according to our definition of b) it is reasonable to deduce that $(x, y) \in b$ but $(x, z) \notin b$ and $(z, y) \notin b$ (since it is not clear that any consensus exists about the relative complexities of x and z and y and z).

The theorem should put an end to the search for the holy grail of a general complexity measure. However, it does not rule out the search for measures that characterise specific views of complexity (which is the true measurement theory approach). For example, a specific program complexity attribute is "the number of independent paths." McCabe's cyclomatic complexity is an absolute scale measure of this attribute. It might even be a ratio scale measure of the attribute of 'testability' with respect to independent path testing. Other specific attributes of complexity, such as the maximum depth of nesting, distribution of primes in the decomposition tree, and the number of paths of various types, can all be measured rigorously and automatically [11], [34].

This idea of looking at measures with respect to particular viewpoints of complexity is taken much further by Zuse [46]. Zuse uses measurement theory to analyse the many complexity measures in the literature; he shows which viewpoint and assumptions are necessary to use the measures on different scales. The beauty and relevance of measurement theory is such that it clearly underlies some of the most promising work in software measurement even where the authors have not made the explicit link. Notable in this respect are the innovative approaches of Melton *et al.* [31] and Tian and Zelkowitz [43]. In both of these works, the authors seek to characterise specific views of complexity. In [43], the authors do this by proposing a number of axioms reflecting viewpoints of complexity; in the context of measurement theory, the axioms correspond to particular empirical relations. This means that the representation condition can be used to determine the acceptability of potential measures.

Melton *et al.* [31] characterise a specific view of program complexity by specifying precisely an order relation $\preceq$ on program flowgraphs; in other words they *define* the binary relation b (of Hypothesis 1) as $\preceq$. The benefit of this approach is that the view of complexity is explicit and the search for representations (i.e., measures of this view of complexity) becomes purely analytical. The only weakness in [31] is the assertion that a measure M is "any real-valued mapping for which $M(x) \leq M(y)$ whenever $x \preceq y$." This ignores the sufficiency condition of the Representation Condition. Thus, while McCabe's cyclomatic complexity [30] satisfies necessity, (and is therefore a "measure" according to Melton *et al.* [31]), it is not a measure in the representational sense (since in Fig. 1 $M(z) < M(y)$ but it is not the case that $z \preceq y$). Interestingly, Tian and Zelkowitz also use the same weakened form of representation, but acknowledge that they "would like the relationship" to be necessary and sufficient.

It follows from Cantor's theorem that there is no representation of Melton's $(F, \prec)$ in $(\Re, <)$. However, it is still possible to get ordinal measurement in a number system which is not $(\Re, <)$ (and hence, for which it is not required that $\prec$ is a strict weak order), although the resulting measure is of purely theoretical interest. It is shown in [12] that there is a representation in (*Nat*, |) where *Nat* is the set of natural numbers and | is the divides relation. The construction of the measurement mapping M is based on ensuring incomparable flowgraphs are mapped to mutually prime numbers. For the flowgraphs of Fig. 1, $M(z) = 2$, $M(x)$ is a fairly large multiple of 3, and $M(y)$ is a very large multiple of 3.

B. The Weyuker Properties

Despite the above evidence, researchers have continued to search for single real-valued complexity measures which are *expected* to have the magical properties of being key indicators of such diverse attributes as *comprehensibility, correctness, maintainability, reliability, testability,* and *ease of implementation* [30], [32]. A high value for a "complexity" measure is supposed to be indicative of low comprehensibility, low reliability, etc. Sometimes these measures are also called "quality" measures [23]. In this case, high values of the measure actually indicate low values of the quality attributes.

The danger of attempting to find measures which characterise so many different attributes is that inevitably the measures have to satisfy *conflicting* aims. This is counter to the representational theory of measurement. Nobody would expect a single number M to characterise every notion of "quality" of people, which might include the very different notions of a) physical strength, and b) intelligence. If such a measure M existed it would have to satisfy a) $M(A) > M(B)$ whenever A is stronger than B and b) $M(A) > M(B)$ whenever A is more intelligent than B. The fact that some highly intelligent people are very weak physically ensures that no M can satisfy both these properties. Nevertheless, Weyuker's list of properties [45] seems to suggest the need for analogous software "complexity" measures. For example, two of the properties that Weyuker proposes any complexity measure M should satisfy are the following properties.

Property A: For any program bodies P, Q, $M(P) \leq M(P;Q)$ and $M(Q) \leq M(P;Q)$.

Property B: There exist program bodies P, Q, and R such that $M(P) = M(Q)$ and $M(P;R) \neq M(Q;R)$.

Property A asserts that adding code to a program cannot decrease its complexity. This reflects the view that program *size* is a key factor in its complexity. We can also conclude from Property A that low comprehensibility is *not* a key factor in complexity. This is because it is widely believed that in certain cases we can understand a program *more* easily as we see more of it [43]. Thus, while a "size" type complexity measure M should satisfy property A, a "comprehensability" type complexity measure M cannot satisfy property A.

Property B asserts that we can find two program bodies of equal complexity which when separately concatenated to a same third program yield programs of different complexity. Clearly, this property has much to do with comprehensibility and little to do with size.

Thus, properties A and B are relevant for very different, and incompatible, views of complexity. They cannot both be satisfied by a single measure which captures notions of size *and* low comprehensibility. Although the above argument is not formal, Zuse has recently proved [47] that, within the representational theory of measurement, Weyuker's axioms are contradictory. Formally, he shows that while Property A explicitly requires the ratio scale for M, Property B explicitly excludes the ratio scale.

The general misunderstanding of scientific measurement in software engineering is illustrated further in a recent paper [9], which was itself a critique of the Weyuker's axiom. Cherniavsky and Smith define a code based "metric" which satisfies all of Weyuker's axioms but, which they rightly claim, is not a sensible measure of complexity. They conclude that axiomatic approaches may not work. There is no justification for their conclusion. On the one hand, as they readily accept, there was no suggestion that Weyuker's axioms were complete. More importantly, what they fail to observe, is that Weyuker did not propose that the axioms were *sufficient*; she only proposed that they were necessary. Since the Cherniavsky/Smith "metric" is clearly not a measure (in our sense) of any specific attribute, then showing that it satisfies any set of necessary axioms for any measure is of no interest at all.

These problems would have been avoided by a simple lesson from measurement theory: the definition of a numerical mapping does not in itself constitute measurement. It is popular in software engineering to use the word "metric" for any number extracted from a software entity. Thus while every measure is a "metric", the converse is certainly not true. The confusion in [9], and also in [45], arises from wrongly equating these two concepts, and ignoring the theory of measurement completely.

IV. Unifying Framework for Software Measurement

A. A Classification of Software Measures

In software measurement activity, there are three classes of entities of interest [11].

Processes: are any software related activities which take place over time.

Products: are any artefacts, deliverables or documents which arise out of the processes.

Resources: are the items which are inputs to processes.

We make a distinction between attributes of these which are *internal* and *external.*

Internal attributes of a product, process, or resource are those which can be measured purely in terms of the product, process, or resource itself. For example, length is an internal attribute of any software document, while elapsed time is an internal attribute of any software process.

External attributes of a product, process, or resource are those which can only be measured with respect to how the product, process, or resource relates to other entities in its environment. For example, *reliability* of a program (a product attribute) is dependent not just on the program itself, but on the compiler, machine, and user. *Productivity* is an external attribute of a resource, namely people (either as individuals or groups); it is clearly dependent on many aspects of the process and the quality of products delivered.

Software managers and software users would most like to measure and predict external attributes. Unfortunately, they are necessarily only measurable indirectly. For example, productivity of personnel is most commonly measured as a ratio of: *size* of code delivered (an internal product attribute); and *effort* (an internal process attribute). The problems with this oversimplistic measure of productivity have been well documented. Similarly, "quality" of a software system (a very high level external product attribute) is often defined as the ratio of: *faults discovered during formal testing* (an internal process attribute); and *size measured by KLOC)* [19]. While reasonable for developers, this measure of quality cannot be said to be a valid measure from the viewpoint of the user. Empirical studies have suggested there may be little real correlation between faults and actual failures of the software in operation. For example, Adams [1] made a significant study of a number of large software systems being used on many sites around the world; he discovered that a large proportion of faults almost never lead to failures, while less than 2% of the known faults caused most of the common failures.

It is rare for a genuine consensus to be reached about the contrived definitions of external attributes. An exception is the definition of reliability of code in terms of probability of failure free-operation within a given usage environment [21], [29]. In this case, we need to measure internal process attributes. The processes are each of the periods of software operation between observed failures; the key attribute is the duration of the process.

B. Software Metrics Activities Within the Framework

The many, apparently diverse, topics within "software metrics" all fit easily into the conceptual framework described above [14]. Here, we pick out just three examples.

1) Cost Modeling: is generally concerned with *predicting* the attributes of *effort* or *time* required for the *process* of development (normally from detailed specification through to implementation). Most approaches involve a prediction system in which the underlying model has the form $E = f(S)$ where E is effort in person months and S is a measure of system size. The function f may involve other product attributes (such as complexity or required reliability), as well as process and resource attributes (such as programmer experience). In the case of Boehm's COCOMO [6], size is defined as the number of delivered source statements, which is an attribute of the final implemented system. Since the prediction system is used at the specification phase, we have to *predict* the product attribute size in order to plug it into the model. This means that we are replacing one difficult prediction problem (effort prediction) with another prediction problem which may be no easier (size prediction). This is avoided in Albrecht's approach [2], where system "size" is measured by the number of function points (FP's). This is computed directly from the specification.

2) Software Quality Models and Reliability Models: The popular quality models break down quality into "factors," "criteria," and "metrics" and propose relationships between them. Quality factors generally correspond to *external* product attributes. The criteria generally correspond to *internal* product or process attributes. The metrics generally correspond to proposed measures of the internal attributes. In most cases the proposed relationships are based on purely subjective opinions. Reliability is one of the high-level, external product attributes which appears in all quality models. The only type of products for which this attribute is relevant is executable software. Reliability modelling is concerned with *predicting* reliability of software on the basis of observing times between failures during operation or testing. Thus internal attributes of processes are used to predict an external product attribute. The prediction systems [29] used in reliability modelling typically consist of a probability distribution model together with a statistical inference procedure (such as maximum likelihood estimation) for determining the model parameters, and a prediction procedure for combining the model and the parameter estimates to make statements about future reliability.

3) Halstead's Software Science: Halstead proposed measures of three internal program attributes which reflect different views of size, namely *length, vocabulary,* and *volume* [18] (all are defined in terms of μ_1, the number of operators, μ_2 the number of operands, N_1 the number of operators, and N_2 the number of operands). For example, length $N = N_1 + N_2$. Although these seem reasonable measures of the specific attributes from the measurement theory perspective, they have been interpreted by many as being measures of program *complexity*, a totally different attribute. Other Halstead "measures" such as E effort, and T time for a program P are genuinely problematic from the measurement theory perspective. Specifically, these are given by:

$$E = \frac{\mu_1 N_2 \log \mu}{2\mu_2} \text{ and } T = E/18.$$

E is supposed to represent 'the number of mental discriminations required to understand P', and T represents the actual time in seconds to write P. It should now be clear that these are crude prediction systems. For example, E is a predicted measure of an attribute of the process of *understanding the program.* However, as discussed in Section II-A, a prediction system requires a means of both determining the model parameters and interpreting the results. Neither of these is provided in Halstead's work. More worryingly, it is possible to show that using E leads to contradictions involving meaningful statements about effort (the attribute being measured) [11].

C. The Importance of Internal Attributes

The discussion in Section IV-B confirms that invariably we need to measure internal attributes to support the measurement of external attributes. This point has also been noted in [43]. Even the best understood external product attribute, *reliability* requires inter-failure time data to be collected during testing or operation. In many situations, we may need to make a prediction about an external product attribute before the product even exists. For example, given the detailed designs of a set of untested software modules, which ones, when implemented, are likely to be the most difficult to test or maintain? This is a major motivation for studying measures of *internal* attributes of products, and was the driving force behind much work on complexity measures.

Consider, for example, the product attribute *modularity.* Many modern software engineering methods and techniques are based on the premise that a modular structure for software is a "good thing." What this assumption means formally is that modularity, an internal software product attribute, has a significant impact on external software product attributes such as maintainability and reliability. Although a number of studies such as [42] and [44] have investigated this relationship there is no strong evidence to support the widely held beliefs about the benefits of modularity. While the study in [42] sought evidence that modularity was related to maintainability, it presumed a *linear* relationship, whereas Pressman and others [35] believe that neither excessively high nor excessively low modularity are acceptable. However, the main problem with all the studies is the lack of a previously validated measure of modularity.

Using the representational approach, we need to identify the intuitive notions which lead to a consensus view of modularity before we can measure it. Some empirical relations were identified in [15]. Others can be picked up be reading the general software engineering literature. For example, it is widely believed that the average module size alone does not determine a system's modularity. It is affected by the whole structure of the module calling hierarchy. Thus, the number of levels and the distribution of modules at each level have to be considered; module calling structures with widely varying widths are not considered to be very modular because of ideas of chunking from cognitive psychology.

D. Validating Software Measures

Validating a software measure in the assessment sense is equivalent to demonstrating empirically that the representation condition is satisfied for the attribute being measured. For a measure in the predictive sense, all the components of the prediction system must be clearly specified and a proper hypothesis proposed, before experimental design for validation can begin.

Despite these simple obligations for measurement validation, the software engineering literature abounds with so-called validation studies which have ignored them totally. This phenomenon has been examined thoroughly in [14] and [33], and fortunately there is some recent work addressing the problem [38]. Typically a measure (in the assessment sense) is proposed. For example, this might be a measure of an internal structural attribute of source code. The measure is "validated" by showing that it correlates with some other existing measure. What this really means is that the proposed measure is the main independent variable in a prediction system. Unfortunately, these studies commonly fail to specify the required prediction system and experimental hypothesis. Worse still, they do not specify, in advance, what is the dependent variable being predicted. The result is often an attempt to find fortuitous correlations with any data which happens to be available. In many cases, the only such data happens to be some other structural measure. For example, in [28], structural type measures are "validated" by showing that they correlate with "established" measures like LOC and McCabe's cyclomatic complexity number. In such cases, the validation study tells us nothing of interest. The general dangers of the "shotgun" approach to correlations of software measures have been highlighted in [8].

The search for rigorous software measures has not been helped by a commonly held viewpoint that no measure is "valid" unless it is a good predictor of effort. An analogy would be to reject the usefulness of measuring a person's height on the grounds that it tells us nothing about that person's intelligence. The result is that potentially valid measures of important internal attributes become distorted. Consider, for example, Albrecht's function points [2]. In this approach, the *unadjusted function count* UFC seems to be a reasonable measure of the important attribute of *functionality* in specification documents. However, the intention was to define a single size measure as the main independent variable in prediction systems for effort. Because of this, a *technical complexity factor* (TCF), is applied to UFC to arrive at the number of function points FP which is the model in the prediction system for effort. The TCF takes account of 14 product and process attributes in Albrecht's approach, and even more in Symons' approach [41]. This kind of adjustment (to a measure of system functionality) is analogous to redefining measures of height of people in such a way that the measures correlate more closely with intelligence. Interestingly, Jeffery [20] has shown that the complexity adjustments do not even improve effort predictions; there was no significant differences between UFC and FP as effort predictors in his studies. Similar results have been reported by Kitchenham and Kansala [24].

V. Summary

Contrary to popular opinion, software measurement, like measurement in any other discipline, must adhere to the science of measurement if it is to gain widespread acceptance and validity. The representational theory of measurement asserts that measurement is the process of assigning numbers or symbols to attributes of entities in such a way that all empirical relations are preserved. The entities of interest in software can be classified as processes, products, or resources. Anything we may wish to measure or predict is an identifiable attribute of these. Attributes are either internal or external. Although external attributes like reliability of products, stability of processes, or productivity of resources tend to be the ones we are most interested in measuring, we cannot do so directly. We are generally forced to measure indirectly in terms of internal attributes. Predictive measurement requires a *prediction system*. This means not just a model but also a set of prediction procedures for determining the model parameters and applying the results. These in turn are dependent on accurate measurements in the assessment sense.

We have used measurement theory to highlight both weaknesses and strengths of software metrics work, including work on metrics validation. Invariably, it seems that the most promising theoretical work has been using the key components of measurement theory. We showed that the search for general software complexity measures is doomed to failure. However, the theory does help us to define and validate measures of specific complexity attributes.

Acknowledgment

I would like to thank B. Littlewood and M. Neil for providing comments on an earlier draft of this paper, and P. Mellor, S. Page, and R. Whitty for sharing views and information that have influenced its contents. Finally, I would like to thank four anonymous referees who made suggestions that clearly improved the paper.

References

[1] E. Adams, "Optimizing preventive service of software products." *IBM Res. J.*, vol. 28, no. 1, pp. 2–14, 1984.

[2] A. J. Albrecht, "Measuring application development productivity," in *IBM Applic. Dev. Joint SHARE/GUIDE Symp.*, Monterey, CA, 1979, pp. 83–92.

[3] J. Aczel, F. S. Roberts, and Z. Rosenbaum, "On scientific laws without dimensional constants," *J. Math. Anal. Applicat.*, vol. 119, no. 389–416, 1986.

[4] A. Baker, J. Bieman, N. E. Fenton, D. Gustafson, A. Melton, and R. W. Whitty, "A philosophy for software measurement," *J. Syst. Software*, vol. 12, pp. 277–281, July 1990.

[5] S. Brocklehurst, P. Y. Chan, B. Littlewood, and J. Snell, "Recalibrating software reliability models," *IEEE Trans. Software Eng.*, vol. 16, no. 4, pp. 458–470, Apr. 1990.

[6] B. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice Hall, 1981.

[7] V. Basili and D. Rombach, "The tame project: Towards improvement-orientated software environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 758–773, June 1988.

[8] R. E. Courtney and D. A. Gustafson, "Shotgun correlations in software measures," *IEE Software Eng. J.*, vol. 8, no. 1, pp. 5–13, 1993.

[9] J. C. Cherniavsky and C. H. Smith, "On weyuker's axioms for software complexity measures," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 636–638, June 1991.

[10] R. A. DeMillo and R. J. Lipton, "Software project forecasting," in *Software Metrics,* A. J. Perlis, F. G. Sayward, and M. Shaw, Eds. Cambridge, MA: MIT Press, 1981, pp. 77–89.

[11] N. E. Fenton, *Software Metrics: A Rigorous Approach.* London: Chapman & Hall, 1991.

[12] ——, "When a software measure is not a measure," *IEE Software Eng. J.*, vol. 7, no. 5, pp. 357–362, May 1992.

[13] L. Finkelstein, "A review of the fundamental concepts of measurement," *Measurement*, vol. 2, no. 1, pp. 25–34, 1984.

[14] N. E. Fenton and B. A. Kitchenham, "Validating software measures," *J. Software Testing, Verification & Reliability*, vol. 1, no. 2, pp. 27–42, 1991.

[15] N. E. Fenton and A. Melton, "Deriving structurally based software measures," *J. Syst. Software*, vol. 12, pp. 177–187, July 1990.

[16] J.-C. Falmagne and L. Narens, "Scales and meaningfulness of quantitative laws." *Synthese*, vol. 55, pp. 287–325, 1983.

[17] P. J. Fleming and J. J. Wallace, "How not to lie with statistics," *Commun. ACM*, vol. 29, pp. 218–221, 1986.

[18] M. H. Halstead. *Elements of Software Science.* Amsterdam: Elsevier North Holland, 1975.

[19] J. Inglis, "Standard software quality metrics," *AT&T Tech. J.*, vol. 65, no. 2, pp. 113–118, Feb. 1985.

[20] D. R. Jeffery, G. C. Low, and M. Barnes, "A comparison of function point counting techniques," *IEEE Trans. Software Eng.*, vol. 19, no. 5, pp. 529–532, Mar. 1993.

[21] Z. Jelinski and P. B. Moranda, "Software reliability research," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic Press, 1972, pp. 465–484.

[22] B. A. Kitchenham and B. de Neumann, "Cost modelling and estimation," in *Software Reliability Handbook*, P. Rook, Ed. New York: Elsevier Applied Science, 1990, pp. 333–376.

[23] D. Kafura and S. Henry, "Software quality metrics based on interconnectivity," *J. Syst. & Software*, vol. 2, pp. 121–131, 1981.

[24] B. A. Kitchenham and K. Kansala, "Inter-item correlations among function points," in *IEEE Software Metrics Symp.*, Baltimore, MD, 1993, pp. 11–15.

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

[26] B. A. Kitchenham and N. R. Taylor, "Software project development cost estimation," *J. Syst. Software*, vol. 5, pp. 67–278, 1985.

[27] H. E. Kyburg, *Theory and Measurement.* Cambridge: Cambridge Univ. Press, 1984.

[28] H. F. Li and W. K. Cheung, "An empirical study of software metrics," *IEEE Trans. Software Eng.*, vol. 13, no. 6, June 1987.

[29] B. Littlewood, "Forecasting software reliability," in *Software Reliability, Modelling and Identification*, S. Bittanti, Ed. (*Lecture Notes in Computer Science*, vol. 341). New York: Springer-Verlag, 1988, pp. 141–209.

[30] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308–320, Apr. 1976.

[31] A. C. Melton, D. A. Gustafson, J. M. Bieman, and A. A. Baker, "Mathematical perspective of software measures research," *IEE Software Eng. J.*, vol. 5, no. 5, pp. 246–254, 1990.

[32] J. C. Munson and Khoshgoftaar, "The detection of fault prone modules," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 423–433, May 1992.

[33] M. Neil, "Multivariate assessment of software products," *J. Software Testing Verification and Reliability,* to appear 1992.

[34] R. E. Prather and S. G. Giulieri, "Decomposition of flowchart schemata." *Comput. J.*, vol. 24, no. 3, pp. 258–262, 1981.

[35] R. S. Pressman, *Software Engineering: A Practitioner's Approach.*, 2nd ed. New York: McGraw-Hill Int., 1987.

[36] F. S. Roberts, *Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences.* Reading, MA: Addison Wesley, 1979.

[37] ——, "Applications of the theory of meaningfulness to psychology," *J. Math. Psychol.*, vol. 29, pp. 311–332, 1985.

[38] N. F. Schneidewind, "Methodology for validating software metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410–422, May 1992.

[39] J. E. Smith, "Characterizing computer performance with a single number," *Commun. ACM,* vol. 31, pp. 1202–1206, 1988.

[40] P. H. Sydenham, Ed., *Handbook of Measurement Science*, vol. 1. New York: J. Wiley, 1982.

[41] C. R. Symons, "Function point analysis: Difficulties and improvements," *IEEE Trans. Software Eng.*, vol. 14, no. 1, pp. 2–11, Jan. 1988.

[42] D. A. Troy and S. H. Zweben, "Measuring the quality of structured design," *J. Syst. Software,* vol. 2, pp. 113–120, 1981.

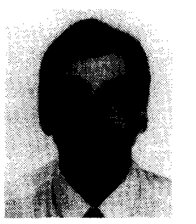
[43] J. Tian and M. V. Zelkowitz, "A formal program complexity model and its applications," *J. Syst. Software*, vol. 17, pp. 253–266, 1992.

[44] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effects of modularisation and comments on program comprehension," in *Proc. 5th Int. Conf. Software Eng.*, 1979, pp. 213–223.

[45] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357–1365, Sept. 1988.

[46] H. Zuse, *Software Complexity: Measures and Methods.* Amsterdam: de Gruyter, 1990.

[47] ——, "Support of experimentation by measurement theory," in *Experimental Software Engineering Issues* (*Lecture Notes in Computer Science*, vol. 706), H. D. Rombach, V. R. Basili, and R. W. Selby, Eds. New York: Springer-Verlag, 1993, pp. 137–140.



Norman Fenton is a Professor of Computing Science in the Centre for Software Reliability at City University, London, UK. He was previously the Director of the Centre for Systems and Software Engineering (CSSE) at South Bank University and a Post-Doctoral Research Fellow at Oxford University (Mathematical Institute).

He has consulted widely to industry about metrics programs, and has also led numerous collaborative projects. One such current project is developing a measurement based framework for the assessment of software engineering standards and methods. His research interests are in software measurement and formal methods of software development. He has written three books on these subjects and published many papers.

Prof. Fenton is Editor of Chapman and Hall's *Computer Science Research and Practice Series* and is on the Editorial Board of the *Software Quality Journal.* He has chaired several international conferences on software metrics. Prof. Fenton is Secretary of the (National) Centre for Software Reliability. He is a Chartered Engineer (Member of the IEE), and Associate Fellow of the Institute of Mathematics and its Applications, and is a member of the IEEE Computer Society.