The changing perception of software metrics

M. Bush

Centre for Systems & Software Engineering,
School of Computing, Information Systems & Mathematics, South Bank University, Borough Road, London, SE1 0AA, UK

ABSTRACT

This paper discusses software metrics as a subject area, and in particular the way it has been growing in popularity and also expanding in scope over recent years. Talk of measurement in various guises is now common-place in mainstream software engineering conferences and periodicals. However, the growth in scope is leading to a lack of focus. There is a danger of losing sight of the original motivation for measurement, which was the desire to move software production nearer towards being a true engineering activity.

1. INTRODUCTION

In recent years software metrics have been attracting increasing amounts of attention and money from various funding sources. There has been a concerted push in particular by various ESPRIT projects aimed at encouraging the European software industry to adopt a more quantitative approach to software development, in the belief that this will lead to better software. Expenditure in phases I and II of ESPRIT alone have amounted to roughly £60 million since 1984, funding more than 600 person-years of effort [1]. Considering that ESPRIT funding represents roughly 5% of all the R&D funds spent on IT in Europe, the total amount of money spent on software metrics “research” in Europe must be in the region of twenty times this figure.

The subject is also becoming the focus of attention for a growing number of seminars, workshops and international conferences. Recent examples are the IEEE International Software Metrics Symposium held in Baltimore in May 1993 (due to repeated in 1994), and “Eurometrics”, the Annual European Conference on Quantitative Evaluation of Software & Systems, which was first held in 1991.

In spite of this, there also remains a good deal of healthy scepticism about software metrics. For example, in the latest edition of Sommerville’s comprehensive and influential textbook [2] (the previous edition being identified in [3] as the most popular undergraduate textbook in the UK on software engineering), discussion of software metrics is left to the very last section - excluding the appendix. The discussion closes with the following statement:

“Metrics clearly have a role to play in the QA process. Unfortunately we do not yet know enough about software metrics to arrive at any general conclusions of what that role should be. Measurements are seductive and it is easy to fall into the trap of accepting figures which appear to confirm prejudices and beliefs. Individual experimentation is essential if metrics are to be useful. QA must be applied to suggested metrics as well as the software process and products!”
418 Software Quality Management

Metrics are clearly not trusted by everyone. However, opinions depend on perceptions. Part of the problem lies in the uncertainty attached to the term “software metrics” itself. The fact that it is generating more and more interest is partly due to the expansion in topics which the term has come to embrace. No doubt Sommerville has a rather more restricted view of the term than some other authors. Actually, the software engineering literature is full of examples of measurements although these are often not recognised as having anything to do with the subject of software metrics.

Indeed, the discussion of many of the fundamental issues of software engineering on which Sommerville’s and other texts are rightly based actually rely heavily upon past quantitative experience and the results of experiments. Two examples of this are the typical proportions quoted for effort per life-cycle phase for different types of software development, and the severity of bugs (in terms of cost to fix) versus the point of their introduction in the life-cycle. Very little can be said about such matters without recourse to measurement in some form.

Thanks to initiatives such as the ESPRIT projects METKIT, AMI, PYRAMID and SCOPE, and the recent QUANTUM study funded by the UK Government’s Department of Trade and Industry [4], software metrics have certainly had a good deal of publicity. However, the message being pushed by these initiatives has still tended to be rather evangelical - a criticism often made of earlier work. At least now there is a growing body of industrial experience in the form of “pilot” studies which give the message some degree of credibility.

In the words of the PYRAMID project [5], the message is:

“Quantitative Management: Get a Grip on Software!”

2. EXPANSION OF THE SUBJECT AREA

The scope of software metrics has been expanding over the years. This section discusses past and present perceptions of the subject, and the problems associated with the term “software metrics” itself due to this expansion.

2.1 Historical Context

The history of software metrics can be traced back as far as the history of software engineering. Inevitably, the impression that most practitioners have of this as of any other evolving discipline lags somewhat behind latest (that is, “current”) thinking. The two names which still come most readily to many people’s minds when the term “software metrics” is mentioned are Halstead and McCabe. This is a pity, since the metrics which they invented have largely been discredited, and things have moved on a long way since then. Their metrics have been kept alive through constant citations and referrals in the literature (this paper being no exception!), and because they are implemented in nearly every software tool which purports to measure software.

McCabe in particular is associated with attempts to measure the “complexity” of software, and indeed this is the area which has traditionally been seen as lying at the very heart of software metrics. This is a sad state of affairs because software “complexity” measurement is in reality only a small, though intriguing, niche area. No doubt there is still a good deal of worthwhile research to be done there, but it would be a pity if this continues to be seen as the heart of the subject. There are
many far more straight-forward and useful areas of measurement in software engineering, as we shall see later.

Another rather unfortunate hangover from the past is the evidence of failed metrics programmes in industry. It is not possible to quantify this, but it seems that failure of metrics programmes, and other initiatives in related areas such as Total Quality Management and Process Improvement, is at least as common as success [6]. No doubt the reasons for it are many and varied, but some obvious and often-used explanations are:

*Insufficient thought given to what to measure*

Measurement for its own sake is unhelpful, and leads to the collection of data which serves no useful end. This is to be avoided, even if there is pressure to “measure something”.

*Imprecise definition of the data to be collected*

Even simple sounding measures like Lines-Of-Code have to be very carefully defined if they are not to be open to interpretation.

*Measurement seen as a low priority activity*

Understandably, the low priority often attached to measurement means that it is often ignored. It is important to strike a balance; it would also be unwise to have very high expectations of measurement, since it is not itself a cure for problems. Measurement can only enable problems to be seen more clearly.

*Lack of co-operation from all staff*

Measurement of the work of individuals must be undertaken with extreme caution since it can easily create ill-feeling. It is also notoriously difficult to use measurement in such a way that it fairly reflects whether or not a person is doing a good job. (See [7] for a comprehensive discussion of this point.)

2.2 Influence of the GQM Paradigm

The so-called “Goal-Question-Metric paradigm”, referred to as GQM, originates from the ideas of Professor Basili and his colleagues at the University of Maryland in the USA [8]. Although the philosophy underlying GQM is so simple as to be almost trivial, it has been very influential in shaping the outcome of initiatives such as METKIT and AMI. Its principal benefit has been to lead researchers and practitioners away from the bottom-up search for interesting and elegant measures towards a top-down approach, starting with a consideration of the goals of measurement.

The idea is that the goals of measurement should be made absolutely clear before any measurement data is actually collected. Goals may be general or specific; often a goal will embody the desire of software engineers to produce better software, or to be more productive. In order to relate individual goals to specific measures it is necessary to identify intermediate questions which identify the information required as the basis for making the decisions which will (hopefully) enable the goal to be achieved. The relationship between a goal and its related questions and measures is normally depicted by a tree diagram, where the root corresponds to the goal and the leaves to particular measures.
Collecting measurement data in software engineering is much like collecting data in any other domain. Consider, for example, a market research exercise with the goal of assessing the effectiveness of a particular advertising campaign. This might involve drawing up a questionnaire to be answered by, say, a thousand individuals selected at random. Questions related to the goal might include the following:

- what does the individual read/listen to/watch, and how frequently?
- has the individual noticed particular adverts in the past?
- is the individual aware of particular brand names?

The questions have to be formulated precisely, and their relationship to the goal must be clear. Only then does it make sense to identify the actual data which needs to be collected in order to address the overall goal. The eventual analysis of the data should of course also be considered before the start, since this may well affect the choice of data to be collected. For example, multiple choice questions will normally yield data that is more amenable to analysis than open-ended questions.

Questions inevitably lead either to indirect measurement of something or to prediction of something which can then be measured once it has occurred. The problem of establishing measures which go towards answering a particular question is exactly the same problem as that of constructing a model relating to an indirect measurement or to a prediction system (as defined in [9]).

In fairness to Professor Basili it should be added that there is considerably more to GQM than just the philosophy that has been described here; the rest consists of guidelines and heuristics to help organisations in defining appropriate GQM trees for their particular purposes. Indeed Professor Basili and his colleagues do good business as consultants by guiding organisations in this process.

2.3 Problems with the Term “Software Metrics”

It is this author’s opinion that the term “software metrics” is misleading and out of date. It is misleading, because:

- the meaning of the term “metric” is unclear, unless we regard it as being synonymous with the term “measure” which is much better understood, in which case it would be clearer to use the latter
- it implies that the subject is about measures of software alone, when the general agreement is that in fact the subject covers measures of software development activities as well, and a good deal besides.

It is out of date because, by general consensus - borne out by the wide scope of content of recent textbooks, workshops and conferences on the subject - the subject now actually incorporates so many diverse topics that a new term is surely warranted (see next section). The IEEE International Software Metrics Symposium held in May 1993, for example, covered software process maturity and its impact on productivity and quality, return on investment for re-used software, evaluation of the success of metrics programmes, and measurement of the effectiveness of testing, in addition to theoretical and practical aspects of the measurement of software products which would have been the focus of such a symposium several years ago.

It seems that the distinctions between “software metrics”, “software engineering” and “software quality assurance” (or perhaps “software quality management”) are getting increasingly blurred.
3. THE “SOFTWARE ENGINEERING MEASUREMENT” UMBRELLA

In an effort to bypass the uncertainty and misconceptions often associated with the term “software metrics”, the METKIT project introduced the new term “software engineering measurement” as an alternative. This section briefly reviews the topics which are covered by this term, and ends with a discussion of the concepts of entities and attributes which have been used as part of the conceptual basis for software engineering measurement. By way of an aside, it also points out that there are still some minor problems of interpretation in this area.

3.1 Topics Covered

It is proposed that the term “software engineering measurement” more accurately reflects the range of topics which the subject now covers. This is regarded as an umbrella term, taking in the role of measurement in the following areas:

- Cost Estimation
- Productivity Models
- Quality Control and Assurance
- Data Collection
- Quality Models
- Reliability Models
- Performance Analysis
- Structural and Complexity Analysis.

The term suggests, literally, the application of measurement in software engineering, which is precisely what the subject is about. It is intended to lead to a perception of the subject as being an integral part of software engineering, which has not always been the case with the term “software metrics”.

The eight topic areas listed above were identified in [10] as being included within the domain of software metrics. The educational material produced by the METKIT project tackles all of these topics, dividing up the treatment of the subject matter into the following areas (see [11] for a fuller explanation):

- Motivation to Measure
- Measurement Theory
- Setting Goals
- Measures & Models
- Data Collection
- Data Analysis & Presentation
- Current Industrial Practice
- Implementing a Measurement Programme.

These areas also cover related topics such as the role of experiments in software engineering, and the issue of certification.

The motivation to measure comes essentially from the desire to ensure that software engineering is truly an engineering discipline. This is further discussed in section 4. The crucial role of measurement theory emerged during the 1980’s,
through the work of such authors as Zuse [12], and Fenton and Whitty [13]. The importance of **setting goals** before starting to make measurements was emphasised by Professor Basili amongst others, as we have seen.

The range of **measures and models** which come under the umbrella of software engineering measurement is vast. Essentially, any measure or model of any attribute of any sensible product, process or resource counts. Therefore cost estimation models, productivity models, quality models, reliability models, performance models and structural models are all included.

**Data collection, analysis and presentation** all play a crucial supporting role. Collection of accurate measurement data often relies on correct understanding and prior consideration of pertinent human issues, and subsequent analysis of the measurement data requires a grasp of the principles of scale types and appropriate statistical analysis techniques.

Measurement is not often thought of as being particularly evident in typical current industrial practice, however if we consider that any form of quantitative analysis counts then we can see that it does play many key roles (for example in product testing and in tracking process costs). Finally, the problem of how to go about **implementing a measurement programme** is one of growing concern, as more and more organisations are getting serious about measurement. Probably the most influential work in this area has been that of Hewlett-Packard, as described in [14]. It seems that the leading Japanese software producers have had active measurement programmes for some time, as they are able to quote figures showing trends in productivity, degree of re-use and product reliability stretching back many years. In particular, their “software factory” ideals involve the consistent long-term use of product, process and resource measurement [15].

### 3.2 The Role of Experiments

It is important to realise the very foundation of software engineering rests in a number of hypotheses which have been confirmed experimentally using measurement. In the past, controlled experiments (of sorts) have been carried out - often on students it must be said - to test specific hypotheses such as that:

- structured programs are easier to understand than non-structured programs
- productivity can vary by more than 10:1 between different individuals
- the use of structured programming techniques leads to fewer bugs
- high level languages are more understandable than low level languages.

In other cases so-called “quasi-experiments” have been used to confirm more general hypotheses such as that:

- low module coupling leads to improved maintainability
- very large modules have higher defect densities than smaller modules.

The goal of an experiment is always to test a given hypothesis by taking a sample of the population to which the hypothesis applies. The hypothesis is ultimately accepted or rejected with an associated degree of certainty. Controlled experiments are those in which the experimenter has a high degree of control over the experiment, and is able to vary one factor alone (the “independent variable”) and observe the outcome (ie. the effect on the “dependent variable”). True controlled experiments are rare in software engineering. “Quasi-experiments”, which are much
more common, are those over which the experimenter has less control. This might be because the sample is not truly representative of the whole population, or because the effects of other variables cannot entirely be eliminated. Quasi-experiments can even be retrospective; in this case previously gathered data is analysed to test a hypothesis for which it was not originally intended.

Indeed, software engineering would have no credibility whatsoever were it not for the “wisdom” contained in such hypotheses, and for the fact that numerous experiments of one sort or another have been carried out over the years which confirm them to greater or lesser degrees.

3.3 Certification

The issue of certification has been attracting growing attention in recent years, and is one of the factors leading to greater emphasis on measurement. A recent study by the DTI [16], based on 200 returned questionnaires, indicated that 70% of UK companies were aware of BS5750 while 40% were aware of ISO9001. Although 43% had implemented company-wide quality assurance (QA) programmes, only a third of these were seen as successful, while a quarter were regarded as failures. BS5750 and ISO9001 represent process certification, whereby an organisation’s working practices are certified as conforming to certain requirements. The assessment is carried out by qualified assessors who visit the company in question to inspect working practices, after studying the documentation relating to the company’s QA system. The assessment is essentially a subjective exercise; measurement as such is hardly involved.

Perhaps more attractive than process certification, although certainly less well advanced, is the notion of product certification. This was one of the main aims of the ESPRIT project SCOPE [17]. The best existing example of such a scheme is the German Gütegemeinschaft programme, in which on the one hand the documentation accompanying a piece of software must adhere to certain requirements (for example it must describe the functionality of the software in some detail), and on the other hand the software itself must conform to the documentation. Hence, the software can be certified with respect to its accompanying documentation.

3.4 The Conceptual Basis for Software Engineering Measurement (and Some Associated Problems)

The METKIT project developed a new conceptual basis to underlie the subject of software engineering measurement. This section briefly reviews two aspects of this framework, highlighting minor problems of interpretation in each case.

The conceptual basis rests partly on a classification of the entities or things in software engineering that can be measured into three distinct groups:

- products
- processes
- resources.

The idea is that products are the artefacts produced by processes which are themselves activities, and that resources are the things which are required in order for these processes to take place (such as personnel and computer hardware). This simple classification scheme, applied to a variety of typical entities occurring in the domain of software engineering, was first proposed in [10] and also forms the basis of the treatment of software metrics in the influential book by Fenton [9].
The essential point about measurement is that it assigns numbers to entities in such a way as to characterise particular attributes of the entities. Examples of attributes of entities are:

- the size of a product
- the duration of a process
- the cost of a resource.

This treatment of the subject is described as a “model of thoroughness” in [18]. However, although it does constitute a simple and apparently elegant way of viewing the subject and of classifying measures accordingly, it is not without its problems. For example, faults may be regarded as attributes of products, and failures as attributes of processes. Alternatively, faults and failures could (perhaps) instead be considered to be entities in their own right having their own attributes. Faults can have locations and potential failures for example, while failures can have times of occurrence and potential effects (on the system).

Failures could conceivably be thought of as instantaneous processes in their own right; perhaps as sub-processes of other processes. Strictly speaking we are not able to regard failures as the products of processes since products must be artefacts of some sort. For this reason faults are also not products in their own right; nor are they processes or resources. Faults are probably best dealt with by sticking to the notion that they are attributes of the products in which they reside, so that if product A has three known faults X, Y and Z, say, then we can speak about the following product attributes:

- the number of known faults in product A
- the location of fault X in product A
- the potential effect of fault Y in product A.

The important point about all this is that it is not always entirely clear whether something should be regarded as a product, a process, a resource, or an attribute of something else. A good example of this problem is whether a methodology should be regarded as an entity in its own right or as an attribute of some other entity. Clearly, methodologies do themselves have a variety of characteristics, such as area of applicability and ease of use. Fenton chooses to regard methodologies as resources in [9].

The distinction between two different kinds of attributes - internal and external - is another fundamental concept as far as METKIT is concerned. The idea is that the internal attributes of an entity can be measured by analysing that entity alone, while the external attributes can only be measured indirectly, based on other measures not only of that entity but also of others. So, for example, the size of a given subroutine is regarded as internal (since it is independent of any other entity) but its speed of execution is external since this depends upon the machine which will execute it. Likewise, its understandability or maintainability are considered external since these depend to some extent on who is doing the understanding or maintaining.

Whether an attribute is internal or external depends on how it is defined. Some attributes are fairly clear-cut in the sense that they are not open to interpretation to any significant degree. Examples include length, mass and temperature. (These also happen to be internal attributes.) Others, however, are much less clear-cut. Examples include beauty, elegance and quality. These tend to be subjective, and are
therefore unmeasurable. To be made measurable they must be unambiguously defined in terms of a set of more basic attributes. Whether or not such an attribute turns out to be internal or external depends upon the basic attributes used in its definition; it can only be internal if all the basic attributes relate to the same entity.

However, like the distinction between different types of entities and their attributes, the distinction between the notions of internal and external attributes is also not always straight-forward. For example, Fenton [9] regards process attributes such as effort and number of failures found during testing as being internal, whereas they could easily be regarded as external, since the former must depend on how many people were involved and the latter must depend on what counts as a failure. If a failure is a deviation from a specification then clearly the number of failures is dependent upon the specification, and therefore must be external.

It is important to realise of course that these are just ways of looking at things. The concept of measurement is essentially very simple. The only motivation for trying to classify the things being measured, or the measures themselves, is the desire to impose some kind of order to make it all easier to explain. The distinctions discussed in this section are relatively simple ones and do make a lot of sense, although we have seen that they can give rise to uncertainties in particular cases.

By way of contrast, the PYRAMID initiative has adopted a different “software quality metrics framework” which divides metrics into the following three distinct classes [5]:

• size metrics
• product quality metrics
• process quality metrics.

The explanations of these are as follows:

“Size
Size can be measured in function points, lines of code, .... It is also often used as a “normalising” factor for the quality categories described below (e.g., faults found per number of thousand lines of code).

Product quality
Metrics such as program or functional specification complexity may give interim indications of the likely quality of the final result. The number of faults detected and counted during system test, the number of system change requests, are examples of product quality metrics.

Process quality
This category of metrics categorises the maturity of the process within which the software product is developed. These metrics can be measured during the product development in order to provide real-time feedback to software project management. Productivity is a process quality metric; an example of this type of metric would be the management of development team productivity.”

This appears to give rise to several areas of confusion. For example, it is not clear why functional specification complexity should be considered to be a quality metric whereas function points should be considered to be a size metric. More generally, it is not clear that a metric is always either to do with size or quality but never both. Also, there seems to be no distinction between an attribute and a
measure of that attribute. For example, if program complexity is considered to be a
quality metric then are all proposed measures of it also quality metrics? Some studies
have shown that size is actually as good an indicator of “complexity” as any other
metric so far proposed! There are other questions, too, such as whether cost
estimation models fit into this scheme, and if so where.

4. MEASUREMENT IN TRADITIONAL ENGINEERING
DISCIPLINES

It has been suggested that the use of measurement is the key distinction between a
craft discipline and an engineering discipline. With this in mind, Capers Jones, for
instance, asserts [7] that:

“The measurement of programming has been the weakest link in the whole
science of software engineering.”

Measurement is something that the software development community - proud of
their status as professional engineers (now official in the UK, by way of the
“Chartered Engineer” status symbol) - are keen to latch on to. Since it is difficult to
say exactly what “engineering” constitutes, measurement seems like a good place to
start.

It is no doubt true that measurement does play a central role in traditional
engineering disciplines. There are currently 42 professional institutions in the UK
recognised by the Engineering Council as engineering institutions [19]. The British
Computer Society is one of the newest of these. By contrast, the Institution of
Electrical Engineers (IEE) is one of the oldest, reflecting the longevity and relative
maturity of the electrical engineering profession. The important role of measurement
is certainly recognised by the IEE, being the focus of interest of one of the
professional groups within the IEE - Group S4: Fundamental Aspects of
Measurement.

This group organised a half-day colloquium in 1992 entitled “Why Aren’t We
Training Measurement Engineers?” reflecting the view, shared by most of the
attendees as well as the colloquium organisers, that measurement should be given
more attention in the electrical engineering industry [20]. In his opening speech, the
chairman declared that “the value of measurement to the economy has been estimated
at approximately 5% of the Gross National Product”. This statement was also used
as a starting point for one of the speakers (Dr. Cunningham from Manchester
University), who suggested that 5% of university courses ought, therefore, to be
measurement-oriented. For example, students reading subjects such as electrical
engineering ought to be spending 5% (or more) of their time studying aspects of
measurement. He used various arguments to show that neither of these is the case.
(After his presentation, another delegate reminisced that he had attended another
colloquium at the IEE, with the same or a similar title, 20 years ago, and that the
situation seemed to him no better or worse now that at any time that he could
remember!)

Another of the speakers (Mr. Cronin from the National Measurement
Accreditation Service - NAMAS) discussed the requirements for so-called
“measurement laboratories” to be NAMAS accredited. Such laboratories exist in
order to periodically test and calibrate sensitive electrical equipment. With over 1500
NAMAS accredited laboratories in the UK, this is something that is clearly taken very seriously within the electrical engineering industry. Interestingly, NAMAS accreditation focuses primarily on the personnel who work in the laboratory being assessed. Specified levels of academic and/or professional qualifications are required, together with relevant experience. If a key member of staff leaves, NAMAS accreditation is often frozen until a suitably qualified replacement is appointed. It is interesting to reflect on the contrast between this and BS5750/ISO9001 accreditation (as described in section 3.1). No doubt the British Computer Society would be happy to see BS5750/ISO9001 accreditation attaching more significance to membership of the Society.

However, while measurement laboratories are commonplace within organisations concerned with electrical engineering, there is nothing whatsoever like it in the field of software engineering. Indeed, whereas measurement laboratories of one sort or another lie at the very heart of many (all?) other engineering disciplines, suggestions that they be set up for the measurement of software are understandably viewed as being rather radical. Nevertheless, such suggestions have been made; see for example the DTI’s QUANTUM report [4].

An important point to note here is that the only kind of measurement envisaged in a measurement laboratory is product measurement. In the case of software, sadly, apart from measures associated with software testing and reliability assessment (which arguably are process measures in any case), and software sizing of one form or another, the useful measurement of software products still remains little more than a promising area of research, even after many years. There are simply no known equivalents of instruments such as voltmeters, ammeters, oscilloscopes, spectrum analysers, thermometers or strain gauges for measuring the performance of software.

Of course, process measurement is important as well. Two areas of particular concern are the prediction and tracking of project costs and time scales. In the software industry today these are now regarded as falling within the domain of the subject of software metrics, as we have seen. However, it must be said that such issues are applicable to any kind of project whatsoever, and it is rather unlikely that engineers in other disciplines would agree that this has much to do with the subject of measurement as they understand it, except in a rather pedantic sense.

5. CONCLUSIONS

This paper has attempted to argue three main points. Firstly, that the scope of what continues to be called software metrics has been steadily widening to the point where the term itself has become misleading (and should be replaced by a new term such as “software engineering measurement”). Secondly, that there are still minor problems of interpretation associated with the conceptual basis for the subject proposed by METKIT. Thirdly, that the subject is now so diverse as to have lost some of its original raison d’être; namely that measurement of software should play a central role in software engineering if it is truly to be an engineering discipline.

What this last point implies is that more emphasis should be placed on software product measurement if we are serious about software engineering being based on measurement. This should be seen in the wider context of software engineering measurement. We must be careful not to let activity in other areas such as process
measurement delude us into believing that we are in fact utilising measurement in software engineering in the true spirit of engineering. Despite its interest and importance, software product measurement still has only a marginal profile within software engineering.

REFERENCES


