Are Software Metrics Really Any Use?

Benjamin Stopford

Introduction

The famous British physicist Lord Kelvin (1824-1904) once commented:

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

This statement, when applied to software engineering, reflects harshly upon the software engineer that believes themselves to really be a computer scientist. The fundamentals of any science lie in its ability to prove or refute theory through observation. Software engineering is no exception to this yet, to date, we have failed to provide satisfactory empirical evaluations of many of the theories we hold as truths.

I take the view that comprehensibility should be the main driver behind software design, other than satisfying business and functional requirements, and that the route to this goal lies in minimization of code complexity. Software comprehension is an activity performed early in the software development lifecycle and throughout the lifetime of the product and hence it should be monitored and improved during all phases. In this paper I will reflect specifically on methods through which software metrics can aid the software development lifecycle through their ability to measure, and allow us to reason about, software complexity.

Kelvin says that if you cannot measure something then your knowledge is of an unsatisfactory kind. What he is most likely alluding to in this statement is that any understanding that is based on theory but lacks qualitative support is inherently subjective. This is a problem prevalent within our field. Software Engineering contains a plethora of self-appointed experts promoting their own, often unsubstantiated, views. Any scientific discipline requires an infrastructure that can prove or refute such claims in an objective manner. Metrics lie at the essence of observation within computer science and are therefore pivotal in this aim.

In the conclusion to this paper I reflect on the proposition that metrics are more than just a way of optimizing system construction, they provide the means for measuring, reasoning about and validating a whole science.

Measuring Software

Software measurement since its conception in the late 1960's has striven to provide measures on which engineers may develop the subject of Software Engineering. One of the earliest papers on software metrics was published by Akiyama in 1971 [8].

Akiyama attempted to use metrics for software quality prediction through a crude regression based model that measured module defect density (number of defects per thousand lines of code). In doing this he was one of the first to attempt the extraction of an objective measure of software quality through the analysis of observables of the system. To date defect counts form one of the fundamental measurements of a software system (although a general distinction between pre and post release defects is usually made).

In the following years there was an explosion of interest in software metrics as a means for measuring software from a scientific standpoint. Developments such as Function Point measures pioneered in 1979 by Albrecht [17] are a good example. The new field of software complexity also gained a lot of interest, largely pioneered by Halstead and McCabe.

Halstead proposed a series of metrics based on studies of human performance during programming tasks [11]. They represent composite, statistical measures of software complexity using basic features such as number of operands and operators. Halstead performed experiments on programmers that measured their comprehension of various code modules. He validated his metrics based on their performance.

McCabe presented a measure of the number of linearly independent circuits through the program [10]. This measure aims specifically to gauge the complexity within the software resulting from the number of distinct routes through a program.

The advent of Object Orientation in the 1990's saw a resurgence of interest as researches attempted to measure and understand the issues of this new programming paradigm. This was most notably pioneered by Chidamber and Kemerer [2] who wrapped the basic principals of Object Orientated software construction in a suite

of metrics that aim to measure the different dimensions of software.

This metrics suite was investigated further by Basili and Briand [25] who provided empirical data that supported their applicability as measures of software quality. In particular they note that the metrics proposed [2] are largely complementary (see later section on metrics suites).

These metrics not only facilitate the measurement of Object Orientated systems but also lead to the development of a conceptual understanding of how these systems act. This is particularly notable with metrics like Cohesion and Coupling which a wider audience now considers as basic design concepts rather than just software metrics. However questions have been raised over their correctness from a measurement theory perspective [26,27,30] and as a result optimizations have been suggested [31☺].

A second complimentary set of OO metrics was proposed by Abreu in 1995 [32]. This suite, denoted the Mood Metrics Set, encompasses similar concepts to Chidamber and Kemerer but from a slightly different, more system wide, viewpoint on the system.

To date there are over 200+ documented software metrics designed to measure and assess different aspects of a software system. Fenton [12] states that the rationale of almost all individual metrics for measuring software has been motivated by one of the two activities: -

1. The desire to assess or predict the effort/cost of development processes.
2. The desire to assess or predict quality of software products.

When considering the development of proper systems, systems that are fit for purpose, the quality aspects in Fenton's second criteria, in my opinion, outweigh those of cost or effort prediction. Software quality is a multivariate quantity and its assessment cannot be made by any single metric [12]. However one concept that undoubtedly contributes to software quality is the notion of System Complexity. Code complexity and its ensuing impact on comprehensibility are paramount to software development due to its iterative nature. The software development process is cyclical with code often being revisited frequently for maintenance and extension. There is therefore a clear relationship between the costs of these cycles and the complexity and comprehensibility of the code.

There are a number of attributes that drive the complexity of a system. In Software Development these include system design, functional content and clarity. To determine whether metrics can help us improve the systems that we build we must look more closely at Software Complexity and what metrics can or cannot tell us about its underlying nature.

Software complexity

The term "Complexity" is used frequently within software engineering but often when alluding to quite disparate concepts. Software complexity is defined in IEEE Standard 729-1983 as: -

"The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics."

This definition has widely been recognized as a good start but lacking in a few respects. In particular it takes no account of the psychological factors associated with the comprehension of physical constructs.

Most software engineers have a feeling for what makes software complex. This tends to arise from conglomerate of different concepts such as coupling, cohesion, comprehensibility and personal preferences. Dr. Kevin Englehart [19] divides the subject into three sections: -

- Logical Complexity e.g. McCabes Complexity Metric
- Structural Complexity e.g. Coupling, Cohesion etc..
- Psychological/Cognitive/Contextual Complexity e.g. comments, complexity of control flow.

Examples of logical and structural metrics were discussed in the previous section. Psychological/Cognitive metrics have been more of a recent phenomenon driven by the recognition that many problems in software development and maintenance stem from issues of software comprehension. They tend to take the form of analysis techniques that facilitate improvement of comprehension rather than actual physical measures.

The Kinds of Lines of Code metric proposed in [28] attempts a measure cognitive complexity through the categorization of code comprehension at its lowest level. Analysis with this metrics gives a measure of the relative difficulty associated with comprehending a code module. This idea was developed further by Rilling et al [33] with a metric called Identifier

Density. This metric was then combined with static and dynamic program slicing to provide a complementary method for code inspection.

Consideration of the more objective, logical and structural aspects of complexity is still a hugely challenging task, due to the number of factors that contribute to the overall complexity of a software system. In this paper I consider complexity to comprise all three of the aspects listed above but note that there is a base level associated with any application at any point in time. The complexity level can be optimized to refractor sections that are redundant or accidentally complex but a certain level of functional content will always have a corresponding *base* level of complexity.

Within research there has been, for some, a desire to identify a single metric that encapsulates software complexity. Such a consolidated view would indeed be hugely beneficial, but many researchers feel that such a solution is unlikely to be forthcoming due to the overwhelming number of, as yet undefined, variables involved. There are existing metrics that measure certain dimensions of software complexity but they do so often only under limited conditions and there are almost always exceptions to each. The complex relationships between the dimensions, and the lack of conceptual understanding of them, adds additional complication. George Statks illustrates this point well when he likens Software Complexity to the study of the weather.

"Everyone knows that today's weather is better or worse than yesterdays. However, if an observer were pressed to quantify the weather the questioner would receive a list of atmospheric observations such as temperature, wind speed, cloud cover, precipitation: in short metrics. It is anyone's guess as to how best to build a single index of weather from these metrics."

So the question then follows: If we want to measure and analyze complexity but cannot find direct methods of doing so, what alternative approaches are likely to be most fruitful for fulfilling this objective?

To answer this question we must fist delve deeper into the different means by which complex systems can be analyzed.

Approaches to Understanding Complex Systems

There are a variety of methods for gathering understanding about complex systems that are employed in different scientific fields. In the physical sciences systems are usually analyzed by breaking them into their elemental constituent parts. This powerful approach, known as Reductionism, attempts to understand each level in terms on the next lower level in a deterministic manner.

However such approaches become difficult as the dimensionality of the problem increases. Increased dimensionality promotes dynamics that are dominated by non-linear interactions that can make overall behaviour appear random [20].

Management science and economics are familiar with problems of a complex, dynamic, non-linear and adaptive nature. Analysis in these fields tends to take an alternative approach in which rule sets are derived that describes particular behavioural aspects of the system under analysis. This method, known as Generalization, involves modelling trends from an observational perspective rather than a Reductional one.

Which approach should be taken, Reductionism or Generalization, is decided by whether the problem under consideration is deterministic. Determinism implies that the output is uniquely determined by the input. Thus a prerequisite for a deterministic approach is that all inputs can be quantified directly and that all outputs can be objectively measured.

The main problem in measuring the complexity of software through deterministic approaches comes from difficulty in quantifying inputs due to the sheer dimensionality of the system under analysis.

As a final complication, software construction is a product of human endeavour and as such contains sociological dependencies that prevent truly objective measurement.

Using metrics to create multivariate models

To measure the width of this page you might use a tape measure. The tape measure might read 0.2m and this would give you an objective statement which you could use to determine whether it might fit it in a certain envelope. In addition the measurement gives you a conceptual understanding of the page size.
Determining whether it is going to rain is a little trickier. Barometric pressure will give you an indicator with which you make an educated guess but it will not provide a precise measure. Moreover it is difficult to link the concept of pressure with it raining. This is because the relationship between the two is not defining.

What is really happening of course is that pressure is one of the many variables that together contribute to rainfall. Thus any model

that predicts weather will be flawed if other variables, such as temperature, wind speed or ground topologies are ignored.

The analysis of Software Complexity is comparable to this pressure analogy in that there is disparity between the attributes that we can currently measure, the concepts that are involved and the questions we wish answered. The relationships between multiple metrics and the, often more encompassing underlying physical attributes are shown in fig 2.

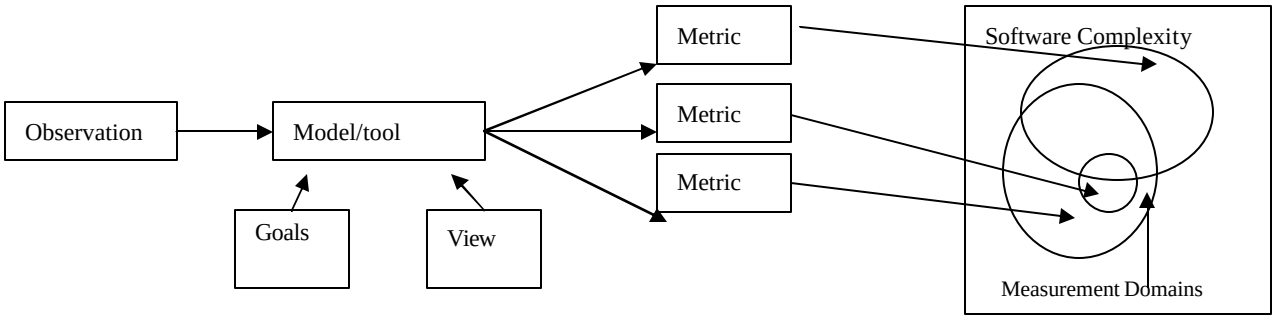

Fig (2): Demonstrates the measurement process of a metrics tool and how it samples from complexity domains.

Multivariate models attempt to combine as many metrics as are available in a way that maximizes the dimension coverage within the model. They also can examine the dependencies between variables. Complex systems are characterized by the complex interactions between these variables. A good example is the duel pendulum which, although being only comprised of two single pendulums, quickly falls into a chaotic pattern of motion. Various multivariate techniques are documented that tackle such interdependent relationships within software measurement. They can be broadly split into two categories:

1. The first approach notes that it is the dependencies between metrics that form the basis for complexity. Thus examination of these relationships provides analysis that is deeper than that created with singular metrics as it describes the relationship between metrics. Halstead's theory of *software science* [2] is probably the best-known and most thoroughly studied example of this.
2. The second set is more pragmatic about the issue. They accept that there is a limit to what we can measure in terms of physical metrics and they suggest methods by which those metrics available can be combined in a way that maximizes benefit. Fenton's Bayesian Nets [4] are a good example of this although their motivation is more heavily focused on the prediction of software cost than the evaluation of its quality.

Metrics suites

One of the popular methods for dealing with the multi dimensionality of complexity is by associating different metrics within a metrics suite. Methods such those discussed in [13], [14] follow this approach. The concept is to select metrics that are complementary and together give a more accurate overview of the systems complexity that each individual metric would alone.

Regression Based and stochastic models

The idea of combining metrics can be extended further with regression-based models. These models use statistical techniques such as factor analysis over a set of metrics to identify a small number of unobservable facets that give rise to complexity.

Such models have had some success. In 1992 Borcklehurst and Littlewood [21] demonstrated that a stochastic reliability growth model could produce accurate predictions of the reliability of a software system providing that a reasonable amount of failure data can be collected.

Models like that produced by Stark and Lacovara [15] use factor analysis with standard metrics as observables. The drawback of these methods is that the resulting models can be difficult to interpret due to their "black box" analysis methodologies. Put another way; the methods by which they analyze cannot be attributed to a causal relationship and hence their interpretation is more difficult.

Halstead [23] presented a statistical approach that looks at total number of operators and operands. The foundation of this measure is rooted in information theory - Zipf's laws of natural languages, and Shannon's information theory. Good agreement has been found between analytic predictions using Halstead's model and experimental results. However, it ignores the issues of variable names, comments,

choice of algorithms or data structures. It also ignores the general issues of portability, flexibility and efficiency.

Causal Models

Fenton [12] suggests an alternative that a uses a causal structure of software development which makes the results much easier to interpret. His proposal utilizes Bayesian Belief Networks. These allow those metrics that are available within a project to be combined in a probabilistic network that maps the causal relationships within the system.

These Bayesian Belief Nets also have the added benefit that they include estimates of the uncertainly of each measurement. Any analytical technique that attempts to provide approximate analysis must also provide information on the accuracy of the results and this is a strong benefit with this technique.

Successes and Failures in Software Measurement

In spite of the advances in measurement presented by the various methods discussed above there are still problems evident in the field. The disparity between research into new measurement methods and their uptake in industrial applications highlight these problems.

There are 30+ years of research into software metrics and far in excess of 200 different software metrics available yet these have barely penetrated the mainstream software industry. What has been taken up also tends to be based on the many of the older metrics such as Lines of code, Cyclometric Complexity and Function points which where all developed in or before the 1970's.

The problem is that prospective users tend to prefer the simpler, more intuitive metrics such as lines of code as they involve none of the rigmarole of the more esoteric measures [12]. Many metrics and consolidation processes lack strong empirical backing or theoretical frameworks. This leaves users with few compelling motivations for adopting them. As a result these new metrics rarely appear any more reliable than their predecessors and are often difficult to digest. These factors have contributed to their lack of popularity.

However metrics implemented in industry are often motivated by different drivers to those of academia. Their utilization is often motivated by a desire to increase certification levels (such as CMM [22]). They are sometimes seen as something used as a last resort for projects that are failing to hit quality or cost targets. This is quite different from the academic aim of producing software of better quality or rendering more effective management.

So can metrics help us build better systems?

Time and cost being equal and business drivers aside, the goal of any designer is to make their system easy to understand, alter and extend. By maximizing comprehensibility and ease of extension the designer ensures that the major burden in any software project, the maintenance and extension phases are reduced as much as possible.

In a perfect word this would be easy to achieve. You would simply take your "complexity ruler" and measure the complexity of your system. If it was too complex you might spend some time improving the design.

However, as I have shown, there is no easily achievable "complexity ruler". As we have seen software complexity extends into far more dimensions that we can currently model with theory, not to mention accurately measure.

But nonetheless, the metrics we have discussed give useful indicators for software complexity and as such are a valuable tool within the development and refactoring process. Like the barometer example they give an indicator of the state of the system.

Their shortcomings arise from the fact that they must be used retrospectively when determining software quality. This fact arises as metrics can only provide information after the code has been physically put in place. This is of use if you are a manager in a large team trying to gauge the quality of the software coming from the many developers you may oversee. It is less useful when you are trying to prevent the onset of excessive or accidental complexity when designing a system from scratch. Reducing complexity through refactoring retrospectively is known to be far more expensive that a pre-emptive design. Thus a pre-emptive measure of software complexity that could be integrated at design time would be far more attractive.

So my conclusion must be that current complexity metrics provide a useful, if somewhat limited, tool for analysis of the system attributes but are, as yet, not really applicable to earlier phases of the development process.

The role of Metrics in the Validation of Software Engineering

There is another view, that the success of metrics for aiding the construction of proper software lies

not in their ability to measure software entities specifically. Instead it is to provide a facility that lets us reason objectively about the process of software development. Metrics provide a unique facility through which we can observe software. This in turn allows us to validate the various processes. Possibly the best method for reducing complexity from the start of a project lies not in measurement of the project itself but in the use of metrics to validate the designs that we wish to employ.

Through the history of metrics development there has been a constant oscillation between the development of understanding of the software environment and its measurement. There are few better examples of this than the measurement of object orientated methods where the research by figures like Chidamber, Kemerer, Basili, Abreu and Briand lead not only to the development of new means of measurement but to new understanding of the concepts that drive these systems.

Fred S Roberts said, in a similar vein to the quote that I opened with:

“A major difference between a “well developed” science such as physics and some other less “well developed” sciences such as psychology or sociology is the degree to which they are measured.”

Software metrics provide one of the few tools available that allow the measurement of software. The ability to observer and measure something allows you to reason about it. It allows you to make conjectures that can be proven. In doing so something of substance is added to the field of research and that knowledge in turn can provide the basis for future theories and conjectures. This is the process of scientific development.

So as a final response to the question posed, software metrics have application within development but I feel that their real benefit lies not in the measurement of software but in the validation of engineering concepts. Only by substantiating the theories that we employ within software development can we attain a level of scientific maturity that facilitates true understanding.

References

[1] Startk and Lacovara; On the calculation of relative complexity measurement.
[2] S. R. Chidamber , C. F. Kemerer : A Metrics Suite for Object Oriented Design
[3] The Goal Question Metric Approach: V. Basili, G Caldiera, H Rombach
[4] Fenton NE, Software Metrics, A Rigorous Approach, 1991
[5[Briand, Morasca, Basili: Property-Based software engineering measurement, IEEE Transactions on Software Engineering 1996.
[6] Zuse H: Software Complexity, Measures and Methods 1991
[7] Bache, Neil: Introducing metrics into industry: a perspective on GQM, 1995
[8] Akiyama F: An example of software system debugging 1971
[9] History of Software Measurement by Horst Zuse (<http://irb.cs.tu-berlin.de/~zuse/metrics/History_02.html>)
[10] T McCabe: A Complexity Measure, IEEE Transactions in Soft Engineering Dec 1976
[11] M.H. Halstead: On Software Physics and GM's PL.I Programs, General Motors Publications 1976
[12] Fenton NE, Software Metrics, A Roadmap, 1991
[13] Nagapan, Williams, Vouk, Osborne: Using In Process Testing Metrics to Estimate Software Reliability.
[14] Valerdi, Chen, and Yang: System Level Metircs for Software development
[15] G. Stark, L Robert on the Calculation of Relative Complexity Measurement
[16] Fenton NE, A critique of software defect prediction models 1999
[17] Albrecht: Measuring application development 1979
[18] David Garland - Why it is hard to build systems out of existing parts.
[19] CMPE 3213 - Advanced Software Engineering (http://www.ee.unb.ca/kengleha/courses/CMPE3213/Complexity.htm)
[20] Ben Goertzel - The Faces of Psychological Complexity
[21] Littlewood B, Brocklehurst S, "New ways to get accurate reliability measures", IEEE Software, vol. 9(4), pp. 34-42, 1992.
[22] Capability Maturity Model for Software - <http://www.sei.cmu.edu/cmm/>
[23] Halstead, M., *Elements of Software Science*, North Holland, 1977.
[24] Klemola, Rilling: CA Cognitive Complexity Metric Based On Category Learning
[25] Victor R. Basili, Lionel C. Briand, Walcelio L. Melo: A Validation of Object-Oriented Design Metrics as Quality Indicators
[26] Neville I. Churcher, Martin J. Shepperd: Comments on 'A Metrics Suite for Object Oriented Design
[27] Graham, I: Making Progress in Metrics
[28] Klemola, Rilling: A Cognitive Complexity Metric Based on Category Learning
[29] Bandi, Vaishnave, Turk: Predicting Maintenance Performance Using Object-Orientated Design Complexity Metrics.
[30] Rachel Harrison, Steve J. Counsell, Reuben V. Nithi: An Evaluation of the MOOD Set of Object-Oriented Software Metrics
[31] S Counsell, E Mendes, S Swift: Comprehension of Object-Oriented Software Cohesion: The Empirical Quagmire
[32] Abreu: The MOOD Metrics Set.
[33] Rilling, Klemola: Identifying Comprehension Bottlenecks Using Program Slicing and Cognitive Complexity Metrics 2003