C. Wohlin, "Revisiting Measurements of Software Complexity", Proceedings Asian Pacific Software Engineering Conference, pp. 35-43, Seoul, South Korea, 1996.

Revisiting Measurement of Software Complexity

Claes Wohlin
Dept. of Communication Systems
Lund Institute of Technology, Lund University
S - 221 00 Lund, Sweden
e-mail: claesw@tts.lth.se

Abstract

Software complexity measures are often proposed as suitable indicators of different software quality attributes. This paper presents a study of some complexity measures and their correlation with the number of failure reports, and a number of problems are identified. The measures are so poor predictors that we might as well use a very simple measure, such a measure is proposed. The proposal is supposed to be ironic to stress the need for a more scientific approach to software measurement. The objective is primarily to encourage discussions concerning methods to estimate different quality attributes. It is concluded that either completely new methods are needed to predict software quality attributes or a new view on predictions from complexity measures is needed. This is particular crucial if the software industry should be able to use software metrics successfully on a broad basis.

1. Introduction

In Oxford Advanced Learner´s Dictionary of Current English, we can, for example, read: Engineer: 1) “skilled and trained person in control of …”, and 2) “arrange or bring about skilfully”. The foundation to turn something into an engineering discipline must be measurement, otherwise we cannot be in control. This means that in software engineering, we need measurement otherwise we are not working with software engineering, but rather with software creation. Therefore, it is essential to use measurements in software development. It is, however, not enough to measure; the measures must be well-defined, goal-oriented and validated. One type of measures which has been discussed since the middle of the 70´s is complexity measures. Although they have been around for 20 years, they are not widely used in the software industry. This paper focuses on different aspects of complexity measurements and some of the intrinsic problems with them.

The complexity of software is often used as an indicator of software quality. The objective is to define a measure which then can be used as a indirect measure of one or several important software quality attributes. The major problem is that complexity of software is not a single measure, and hence it is extremely difficult to relate it directly to software quality. It is, however, a quite generally accepted method to use complexity metrics to try to estimate and predict software quality attributes, even though the task is difficult and evidence of success still is lacking for most complexity measures proposed in the literature.

Complexity measures of software have been debated since the 70´s, when McCabe presented his article on cyclomatic complexity, [1] and Halstead published his ideas concerning software science, [2]. A major problem has since been present, i.e. the inability of these complexity measures to predict different software quality attributes accurately, as for example reliability and maintainability. A large number of measures have been proposed during the last two decades, but still there is no common agreement on a suitable measure.

The maturity of the software process has been very much in focus during the 90´s. A similar reasoning can be made for the use of software metrics. During the 80´s, much research was focused on metrics and correlation between different metrics and, for example, fault content. These studies were often discouraging and the researchers from the 80´s have often moved on to identification of outliers or fault-prone modules [3, 4] and measurement for process improvement [5].

Unfortunately, the growing maturity in part of the research community is not widely spread. The research continues and new measures are proposed regularly [6, 7] and investigations on correlations between measures and, for example, fault content are published every now and then [7]. The interest for software metrics in industry is spreading, but it is too often believed that simple relationships can be found, for example, through correlation stud-

ies. The overall objective here is to indicate that this belief is not well-founded.

More precisely the objectives of this paper are:

- to present a prior unpublished study of the correlation between a number of complexity measures and the number of failure reports,
- to discuss a number of unsolved problems with complexity measures,
- to argue that we might as well go back to a very basic measure, which is understandable to a majority of the people, not only in the software community, since the existing measures of complexity are not really complexity measures at all. Unfortunately, this means arguing yet another measure, but the simplicity and the well-known interpretation of the measure are, however, outweighing the disadvantages with introducing a new measure.
- to promote debate and discussion about the interpretation, usability and predictability of software quality attributes from complexity measures. The measure mentioned in the previous item is primarily suggested to encourage discussions, but still it shows the level that software complexity measures is on.

The latter is actually the main objective of this paper otherwise complexity measures will continue to be researched within the research community and not being used within the software development community. Therefore, intensive discussions are needed to overcome the problems with complexity measures or to invent other means to estimate and predict software quality attributes during software development.

2. Predicting software failures from complexity measures

2.1 Introduction

The objective of most studies of complexity measures is to correlate the measures with the number of failure reports or similar attributes. Based on the correlation, a model is derived which is intended to be used for prediction purposes for future software systems. This section presents a study of several complexity measures and the objective is to identify one or several measures which can be used to predict the number of failure reports as new software modules are developed.

The study was conducted in 1983, but the results have not been published before and unfortunately the results are still quite typical for many investigations, where we are trying to correlate some complexity measures with for example the number of failure reports. A more thorough presentation of the study can be found in [8].

The results are based on an investigation of 28 software modules for a large real time system. The size of the modules is between 270 to 1900 lines of code. For each of these modules the number of failure reports were counted and several measures intending to be complexity measures were derived. In total 27 different measures were collected, with 12 measures being collected from the design documentation and 15 measures being collected from the resulting code.

2.2 Data collection

The design data collected were primarily a counting of the number of symbols of different types. The language used during design is a predecessor to SDL, (Specification and Description Language) [9], and it is basically a finite-state-machine with states and signals. A modified version of McCabe´s Cyclomatic Complexity, [1] was used as well. The modification was simply due to being able to handle signal sending and reception respectively.

The measures from the code are also primarily a matter of counting the number of constructs of different types, for example number of variables and number of if-statements. It must be noted that the programming language was very much based on goto-statements, and in particular it was observed that two different types of goto-statements were used. The difference is:

- goto 11; (unconditional goto)
- if … then goto 12; (conditional goto)

These two types were calculated separately since the hypothesis was that the first type of goto contributed more to the fault content than the second type of goto. One more sophisticated measure, than just calculating the different constructs in the language, was tested, i.e. an information theory based measure [10].

It is infeasible to present all data in a paper like this. Therefore the three best design measures (1-3) and the three best code measures (4-6) are only shown in Table 1, together with the number of failure reports for each module and the length of each module in terms of lines of code, including comments. The best refers to the best measures when minimizing the sum of the quadratic deviations from the best line, where the line was identified through linear regression. The data was also analysed with multiple linear regression and with non-linear regression. Both of these types of analysis improved the correlation and also the predictive ability. The improvements were, however, minor and no significant breakthrough could be observed. Therefore, only the results from the linear regression are presented as the results in themselves are not the main concern.

The complete data set can be found in [8]. The data have not been analysed with, for example, neural networks

[11] and principal component analysis [12]. This will not be done either as the problem of complexity measures are not primarily in the statistical method being used. Of course, the methods can be improved, but the underlying problem is the measures themselves. This is further discussed in Section 3 and it is also discussed in [13], where a rigorous approach to software measurement is emphasized. An interesting observation from Table 1, is that the two more complicated measures, that is McCabe´s Cyclomatic Complexity and the information theory based measure presented by Berlinger [10], were worse than crude counting measures of different symbol types in the design and different language constructs on the code in predicting the number of failure reports.

The three design measures are: number of input signals to the module (measure no. 1), number of internal signals in the module (measure no. 2) and number of output signals from the module (measure no. 3). An interesting observation is to note that the three best predictors of failure reports from the design are all related to signalling.

TABLE 1. Data collected.

Module	Length	Failure reports	1 Input signals	2 Internal signals	3 Output signals	4 Uncon. goto	5 Enter clause	6 No. of var.
1	732	0	24	1	20	53	24	72
2	936	0	9	1	12	107	11	58
3	1338	0	14	1	13	132	17	72
4	1492	0	33	1	35	129	32	93
5	748	0	11	0	10	43	11	64
6	611	0	14	3	13	45	17	69
7	618	1	12	1	14	66	14	62
8	271	1	4	2	4	13	6	29
9	697	1	15	1	16	59	15	70
10	1633	1	41	1	41	114	40	82
11	613	2	8	1	8	64	12	50
12	1351	2	10	1	16	149	11	65
13	1144	2	25	0	26	125	24	76
14	1316	3	17	2	16	176	22	77
15	416	4	11	0	12	33	10	26
16	736	4	15	2	15	38	17	53
17	1171	4	29	1	32	106	29	111
18	1885	4	19	2	21	172	15	144
19	1464	5	19	3	19	163	22	78
20	1092	5	11	2	14	80	17	78
21	1002	5	19	1	18	103	20	95
22	1233	5	15	1	18	114	16	65
23	1205	5	18	1	18	183	19	87
24	1306	6	24	1	25	153	24	89
25	1491	8	22	6	18	128	22	133
26	1777	11	71	6	59	211	74	147
27	1377	12	31	1	31	161	33	86
28	639	17	20	0	20	59	20	42

These three measures are, however, quite highly correlated, which will be highlighted in Section 2.3, as many inputs normally also means a lot of internal signalling and then responses are sent. The three code measures are not as directly related. The code measures are: number of unconditional goto (measure no. 4), number of Enter clauses in the code (measure no. 5) and finally number of declared variables in the module (measure no. 6).

It may also be noted that even if more sophisticated statistical methods are applied in [12] and [11], the measures collected are still about the same as collected in this study presented here, which was conducted 13 years ago. The objective is not to criticize these studies in particular, but to point out the general status in the field. The work reported in [12] and [11] is on the contrary some of the best recent work within the field of complexity measurement.

2.3 Data analysis

The analysis includes both inter-correlation between the complexity measures and correlation of these to the number of failure reports, as well as correlation between the different measures and the program length[1]. It is of particular interest to correlate the design measures to the program length as that actually is a prediction. Based on the correlation study, linear regression is applied for the best design and code measures respectively, that is we try to identify a prediction formula to determine the number of failure reports from the collected measures. A formula for predicting the program length from the best design measures is also presented. The prediction is based on the best measures for predicting the number of failure reports. The correlations are shown in Table 2 and the prediction formulas are presented below.

1. Both the correlation study and the linear regression are performed with Matlab.

The correlations are unfortunately quite low in most interesting cases. In particular, it is worth noting that no measure has a high correlation with the number of failure reports, and that the lowest correlation with the number of failure reports is for the program length. Thus indicating that the failures must be explained by other means. The correlation figures mean that the best prediction formulas are:

- Number of failure reports (from design) =
  = 0.124 * Input + 1.38
- Number of failure reports (from code) =
  = 0.127 * Enter + 1.16
- Program length (from design) = 24.6 * Output + 586

The predictive accuracy is not very good as can be guessed from the correlation figures. It is, of course, possible to perform a thorough investigation of the accuracy. The comparison can either be made with the data used in the formulation of the formulas or by letting new modules be evaluated towards the formulated prediction formulas. In this particular case, we have used all the available data in the derivation of the formulas, but it would have been possible to, by random, select a number of modules which were excluded from the derivation of formulas a solely used for evaluation purpose.

The mean estimation error for the modules included in the study are, for the three cases, 72%, 78% and 32% respectively, where the mean error is calculated as:

$$MeanError = \left(\sum_{i=1}^{28} ABS\left(\frac{(Estimation(i) - RealValue(i))}{RealValue(i)} \right) \right) / 28$$

since the number of modules is 28, but if the real value is zero it is skipped from the calculation. This is the case for some of the modules as no failures have been reported for them. It is, of course, necessary to take the absolute value for each error, which is shown through the function ABS. It should also be observed that all figures have been rounded off to integers.

The mean error is quite high, at least when predicting the number of failure reports hence indicating that there

TABLE 2. Correlations between the different measures.

Corr.	Var.	Enter	Goto	Output	Internal	Input	Failure
Length	0.81	0.58	0.89	0.66	0.39	0.60	0.22
Failure	0.25	0.40	0.30	0.37	0.25	0.39	-
Input	0.65	0.99	0.54	0.98	0.44	-	-
Internal	0.60	0.49	0.35	0.33	-	-	-
Output	0.64	0.95	0.58	-	-	-	-
Goto	0.70	0.55	-	-	-	-	-
Enter	0.62	-	-	-	-	-	-

are more important factors to explain the number of failure reports than a single and too simple complexity measure. The best mean error is obtained when estimating the length of the program from the design. This is hardly surprising as that relationship is much clearer. The mean error can hardly be expected to be better for modules which have not been included in the formulation of the formulas, hence this type of estimation procedure is quite inaccurate.

2.4 Conclusions

The correlations reported in the previous section as well as the mean errors are discouraging. Therefore, it is not possible to recommend the use of single measures to capture software complexity and it is very doubtful if one single measure ever can be used to predict the number of faults to be found in specific modules. One problem arising is the actual granularity. Modules may be too fine grain to allow for prediction, a potential problem is the dependency of the individual who has developed the module. This issue must be further investigated, to see if, for example, a grouping of modules into subsystems is a feasible approach. The usefulness of single measures is the ability to identify outliers, that is modules which have an exceptional high value of one or several parameters and hence is likely to be fault-prone. This is for example discussed in [4, 14].

The results from the study presented above have not been presented as they did not give any valuable (prediction) result. Unfortunately, the decision not to publish the results means that the same type of problems are encountered by other people trying a similar approach. Software complexity measures have a number of problems, which are discussed in Section 3.

3. Problems with complexity measures

3.1 Identification of problems

The rationale behind software complexity measures is simple and easy to understand. The hypotheses are that a complex piece of software is more likely to contain faults and also that it is probably harder to maintain a complex program than a simple one. Complexity measures are introduced to measure directly measurable aspects of software, hence working as indirect measures of the attributes of actual interest, i.e. reliability and so forth. The basic assumption is, of course, that a relationship between the measurable aspects and the attributes of interest can be identified. This is the difficult part. It is quite apparent that it is easier to measure a specific complexity measure on, for example, a computer program, than to relate this measure to the program´s actual reliability. This problem ought to be possible to overcome, if it was not for all the other problems with this type of measures. The problems are as follows:

- Metrics definition
  In many cases, there exists no common definition of a complexity measure or the definitions are interpreted differently by different people applying the measures. The most well-known example is, of course, the problem of defining lines of code. At first, it seems simple enough, but when starting to discuss the issue the problem becomes apparent. This means that it is not possible to interpret and understand experiences from others when applying complexity measures, hence being an immense problem as it implies that everybody has to build their own experience base, without being able to learn anything from others.
- Aspects of complexity
  Most complexity measures focus on one aspect of the software, i.e. the measures can often be related to one or perhaps two of the following classes: size, control, data or communication. It is, for example, easily realized that lines of code is a size measure and it does not reveal any information about the structure of the program or about the complexity of the data handling. Therefore, complexity measures are too restricted in their judgement of actual complexity and consequently they are not good indicators of the software quality attributes. Moreover, they should not really be referred to as being complexity measures as they do not measure complexity. The proposed complexity measures usually represent one or perhaps two aspects of complexity, but complexity consists of too many aspects to, with our current level of understanding, represent it with a single figure.
- Applied late in the life cycle
  Another common problem of complexity measures, is that they are often not applicable until late in the software life cycle. Most measures cannot be used until the coding phase, which seems ridiculously late if being able to use the measures to improve the software. Although, it is better to at least get some indications during coding, than being surprised in testing. So in some sense, coding is early, but in another sense it is late, since we are not able to do much about it in the ongoing project. The prediction of software quality attributes is often not available until the software is going to be shipped to the customer, which is clearly too late. It is no use standing there with the hat in our hand saying: yes, we know the software is unreliable we measured it the same day we delivered the software. Some measures that try to address this problem have been proposed, for example, function points [15].

- Static measures vs. dynamic behaviour
  Furthermore, even if succeeding in using complexity measures to estimate static aspects during development, there has been little work done to relate these estimates to the dynamic behaviour encountered when executing the software. An example is that complexity measures may be related to fault content, but then the problem of mapping fault content to failures and hence to reliability remains. Some work has been done to overcome these problems, but the problem is not solved.
- On-line measurement
  Complexity measurements in an on-going project is difficult, even though it is preferable to the situation described in the next item. The major problem with on-line measurement is that as soon as people become aware of what you are measuring, then they are able to influence it. This is a similar phenomena as Heisenberg's principle in physics, i.e. you influence your measurement object and hence it is doubtful if any conclusions can be drawn from the study.
  A simple example of Heisenberg's principle in software can be formulated as follows: Assume that you have decided to measure the number of unique operands, cf. Halstead´s software science, as soon as this becomes well-known in the project, it is likely that the programmers try to keep the number of operands down to a minimum. This means that they reuse operands for different purposes, hence making the program really complicated and, of course, error-prone. Reuse may be argued as an important issue to raise software productivity, but hardly if it comes to reuse of operands.
- Unexpected correlation
  The collection of a large number of metrics increases the risk that a correlation is found by coincidence, i.e. two parameters may be correlated by chance. This means that there is a risk, that it is believed that a certain relationship exists although it does not in reality. This means that erroneous conclusions may be drawn.
- Retrospect analysis
  Last, but not least, one problem is retrospect analysis. In many cases the analysis presented in the literature is based on investigations being done in retrospect, which means that the complexity measure is based on software which has been changed due the faults found, and then it is clearly wrong to try to correlate the complexity measure with the faults found, see Figure 1. Assume, for example, that the original software refers to the software prior to system testing, then systematic testing is performed and several faults are identified. These faults are corrected and the software can be released.

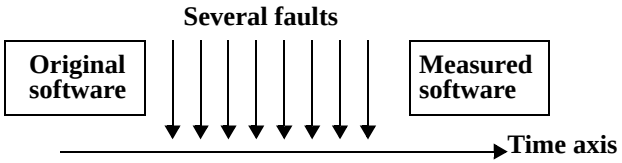

FIGURE 1. The retrospect problem.

The persons performing complexity measurements comes in and measure the complexity of the delivered software and then try to correlate the measures with the number of faults found during system testing. This results in, at least, one big mistake, i.e. the measures are based on software which has been changed due to the faults. It may be the case that a simple module prior to system testing is quite complicated after system testing due to that a large number of faults were found in it, but on the other hand it is correct now. In this particular case, the complexity comes from correcting the faults and hopefully obtaining a better module after the corrections, even if it seems as it may be problematic due to high complexity and a large number of faults. Many scenarios of this type can be formulated, hence retrospect analysis is bound to yield incorrect results and it must be abandoned.

To these problems of complexity measures themselves, we may add the poor predictive ability and accuracy. All in all, when adding up the problems it is no surprise that very few companies use complexity measures. New methods must evolve, as, for example, recent ideas of applying capture-recapture techniques to software inspections. The difficulty with complexity measures means that we might as well go back to a very basic definition, instead of inventing new complexity measures which have all the above disadvantages.

The possible problems listed above should be recognized, although some of the issues listed may be more or less relevant or important in different cases. For example, it may be positive if software developers strive to minimize certain metrics given that it actually means that it improves one or several software product quality attributes.

3.2 Problems of our study

Unfortunately, the study described in Section 2 suffers from most of the problems of software complexity measures presented in Section 3.1. The points in the previous section are here treated one by one and compared with the study described above.

- Metrics definition
  In the study, a common definition existed since all the measures were defined and collected by one person, but the definitions were not formal hence making it difficult for anybody else to use the definitions with the same interpretation.
- Aspects of complexity
  All the measures in the study only measured one aspect at the time, hence no measure was able to estimate "true complexity".
- Applied late in the life cycle
  Most of the measures were collected from the code, which clearly most be considered as being too late. Some measures were, however, collected from the design descriptions which improves the situation. Unfortunately, there are no measures from the requirements specification or other specifications.
- Static measures vs. dynamic behaviour
  The measures are correlated to failure reports from the field, but as they are not related to the usage of the software and the time between failures, it is not possible to estimate the reliability. The failure reports reflect problems, but only as a part of the total fault content.
- On-line measurement
  The study was not performed on-line and hence the problem with on-line measurement is irrelevant for the study.
- Unexpected correlation
  The study did not reveal any unexpected correlation. The problem of the study is primarily the lack of correlation.
- Retrospect analysis
  This problem is critical in the study reported. The analysis was made without being in control of the actual connection between the descriptions from which the measures were collected, and the descriptions from which the failures were reported. Thus, the study is dubious as it is not possible to determine if we actually are correlating measures which have a relation or not.

In summary, we can conclude that study suffers from four of the problems and partly from one of them and finally two of the problems are irrelevant. This makes it quite clear that the result from the study are not reliable and hence not particular useful, when trying to turn software creation into software engineering. This is clearly supported by the low correlation figures and relatively high mean errors reported above. Although the objective is to improve software measurement with this type of studies, it is more probably that they harm the field of software measurement more than they improve the field.

4. Software weight

In order, to emphasize the lack of scientific foundation in many measures proposed for software, a metrics of our own is proposed. The metric is supposed to be ironic and the objective is to stimulate discussion and hopefully an increased awareness about software metrics and in particular concerning the need for a rigorous approach, which also is stressed elsewhere see for example [12].

This section introduces a simple and understandable complexity measure, and the objectives with the introduction of a new measure are: to provide a basis for discussion and to indicate the level of finesse in many of the existing complexity measures. The measure to introduce is: Software Weight! Would it not be wonderful to ask questions like: How much does your program weigh?

Software weight is a relevant complexity measure. The measure can be applied on the requirement specification by just putting the requirements document on the balance. The actual code can also be weighed, either by putting the code listings on the balance or even better to go back to the good old days when the code was on punch cards. The latter can be used in the following way:

1. Weigh the cards prior to punching the cards,
2. Punch the program into the cards,
3. Weigh the cards after the program has been put on the cards,
4. The software weight is now equal to the difference between the weight before and after the punching.

This procedure can be summarized in the following formula:

$$S_W = PC_{W1} - PC_{W2}$$

where, S_W is the software weight and PC_{W1} and PC_{W2} are the weight of the punch cards before and after having put the program on to the cards respectively. This measure ought to capture code complexity, at least as good as most existing measures, as the number of holes in the cards mirrors the implemented code.

Therefore, it is proposed to start using this basic complexity measure for software. The measure may even be described in a standardized unit, i.e. grams. The unit is also understandable to people outside the computer society. This measure would have helped in the following anecdote, which probably is not true, but all the same quite charming:

"Back in the 70´s, when one of the Apollos where being sent out into space, there was one person being responsible for the weight of the rocket. His responsibility was to make sure that the weight of the rocket was below a given value, which would enable the rocket to take off. He knew that software was going to be loaded on the comput-

ers, and hence he went to the computer department to inquire about the weight of the software. He asked about the weight of the software, but was told that it did not weigh anything. This answer was not accepted, and he told the computer department that either they found out the weight or the software did not go aboard. He went away, but promised to come back the next day and then he expected a more accurate answer. When he came back, the showed him the punch cards and he said, with some satisfaction, I was right these clearly weigh something. The computer expert then told him; we do not send the cards, we send the holes!".

This minor anecdote clearly shows that the notion of software weight is needed. Therefore, it is proposed that software weight ought to be used as a measure of complexity of software, as it is at least as relevant as most existing complexity measures of today!

The challenge for the future, as unfortunately seen by too many software metrics researchers, is then to start using this measure and try to correlate it to the important software quality attributes, as for example reliability and maintainability. These are commonly reported as being more or less correlated with all these obscure existing complexity measures of software, and hence ought to be correlated with the software weight as well.

This is certainly not the appropriate road ahead in software measurements. Other ways must be sought, which is further discussed below.

5. Discussion

Software measurement must be taken seriously as it is the basis to turn software development into an engineering practice. Therefore, the approach to complexity measurement of software must be improved. It is not enough to try to correlate any measure with anything of interest, usually external software attributes as for example reliability. Software measurement must be based on a well-founded hypothesis and then the measurements conducted must be carried out in an orderly manner. Too many investigations of software complexity are bound to fail as they suffer from many of the problems presented in Section 3. Measurement must be based on scientific principles and they must be goal-oriented [15]. In other words, we must have goals and determine what to measure based on these goals, rather than start measuring and then see if the data collected is useful.

Furthermore, if measurement is the basis for turning software development into an engineering discipline, then the maturity of software measurement must grow beyond the search for a silver bullet in the form of the ultimate software complexity measure or correlation studies. It is the responsibility of the software measurement community to lead the way by abandoning this search and turn software measurement into a rigorous engineering discipline on which software engineering can be based. We cannot expect industry to embrace the concept of software measurement as long as most papers being published focus on finding the silver bullet for prediction, when other articles are arguing that we cannot expect a silver bullet to solve the software crisis, [17].

More specifically, complexity measures do have several problems as pointed out in this paper. The measures have been around for 20 years and the problems are still the same. To overcome the prevailing problems of measuring complexity to predict external product attributes as for example software reliability, two things are needed:

- New methods and techniques to estimate and predict software quality attributes throughout the software life cycle. Some research is in progress to come up with other techniques than relying on unreliable complexity measures, for example capture-recapture techniques as discussed in [18, 19].
- A vector of complexity measures is needed, that is measures which captures different aspects of software complexity must be used. The vector can then be used to capture extreme behaviour of some modules and hopefully this can lead to the identification of outliers.

The first item has been outside the scope of this paper, but the second item has been emphasized by showing the inability of different complexity measures to predict the number of faults in different modules.

Software weight is probably not a good complexity measure, but it shows clearly the level of finesse that many of the existing, so called, complexity measures is on. The simple measure proposed has its benefits though; the measure can be applied throughout the software life cycle and in particular the real logical weight of a piece of software can be determined by using the software weight as defined in this paper. The latter implying that we would get a measure which is understandable outside the software community. Unfortunately, the software weight is probably as good predictor of software quality attributes as most existing measures.

The objective here is, however, not to argue for this measure, on the contrary, the objective here is to promote discussion and emphasize that measurement of software complexity must be treated differently. It is not possible to find one general measurement describing software complexity.

To summarize, software measurement is an important activity, but it must be carried out rigorously. The status of complexity measures and prediction from them cannot be considered to fulfil this criterion. Thus, software measurement of today will not form the necessary platform for turning software development into an engineering discipline.

Acknowledgement

I would like to express my sincere thanks to Prof. Min Xie, National University of Singapore for presenting this paper for me at the Asian-Pacific Software Engineering Conference in Seoul, South Korea in December 1996.

References

[1] T.J. McCabe, “A Complexity Measure”, IEEE Transactions on Software Engineering, Vol. 4, No. 2, pp. 308-320, 1976.

[2] M. H. Halstead, “Elements of Software Science”, Elsevier North-Holland, 1975.

[3] J. Munson and T. Khoshgoftaar, “The Detection of Fault-Prone Programs”, IEEE Transactions on Software Engineering, Vol. 18, No. 5, 1992.

[4] N. Ohlsson, M. Helander and C. Wohlin, “Quality Improvement by Identification of Fault-Prone Modules Using Software Design Metrics”, Proceedings 6th International Conference on Software Quality, Ottawa, Canada, October 1996.

[5] V. Basili, G. Caldiera and H. D. Rombach, “Experience Factory” in Encyclopedia of Software Engineering, Vol. 1, editor: J. J. Marciniak, John Wiley and Sons, pp. 469-476, 1994.

[6] Y. R. Pant, J. M. Verner and B. Henderson-Sellers “S/C: A Software Size/Complexity Measure” in Software Quality & Productivity: Theory, Practice, Education and Training, editors: M. Lee, B-Z. Barta and P. Juliff, Chapman & Hall, pp. 320-327, 1995.

[7] P. S. Grover and N. S. Gill, “Composite Complexity Measures’ in Software Quality & Productivity: Theory, Practice, Education and Training, editors: M. Lee, B-Z. Barta and P. Juliff, Chapman & Hall, pp. 279-283, 1995.

[8] C. Wohlin, “A Study of Software Complexity”, Master thesis, Dept. of Communication Systems, Lund University, Lund, Sweden, 1983 (in Swedish).

[9] ITU, “Recommendation Z.100: SDL - Specification and Description Language”, 1988.

[10] E. Berlinger, “An Information Theory Based Complexity Measure”, Proceedings National Computer Conference, pp. 773-779, 1980.

[11] T. Khoshgoftaar and D. L. Lanning, “A Neural Network Modeling Methodology for the Detection of High-Risk Programs”, Proceedings 4th IEEE International Symposium on Software Reliability Engineering, Denver, Colorado, USA, November 1993.

[12] J. Munson and R. H. Rawenel, “Designing Reliable Software’, Proceedings 4th IEEE International Symposium on Software Reliability Engineering, Denver, Colorado, USA, November 1993.

[13] N. Fenton, “Software Metrics: A Rigorous Approach”, Chapman & Hall, 1991.

[14] D. Kafura and J. Canning, “A Validation of Software Metrics Using Many Metrics and Two Resources”, Proceedings 8th IEEE International Conference on Software Engineering, London, UK, August 1985.

[15] A. J. Albrecht and J. E. Gaffney, “Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation”, *I*EEE Transactions on Software Engineering, Vol. 9. No. 6, pp. 639-648, 1983.

[16] Basili, V., Caldiera, G. and Rombach, H.D., “The Goal Question Metric Approach”, in Encyclopedia of Software Engineering, Vol. 1, editor: J. J. Marciniak, pp. 528-532, John Wiley and Sons, New York, 1994.

[17] F. J. Brooks, “No Silver Bullet: Essence and Accidents in Software Engineering”, IEEE Computer, pp. 10-19, April 1987.

[18] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta and S. Vander-Wiel, “Estimating Software Fault Content before Coding’, Proceedings 14th IEEE International Conference on Software Engineering, Melbourne, Australia, 1992.

[19] C. Wohlin, P. Runeson and J. Brantestam, “An Experimental Evaluation of Capture-Recapture in Software Inspections”, Software Testing, Verification and Reliability, Vol. 5, No. 4, pp. 213-232, 1995.