Measurement of Software Quality

Thorbjörn Andersson * Aimo Törn †

Åbo Akademi University,
Department of Computer Science,
DataCity, SF–20520 Turku, Finland

March 8, 1992

Abstract

Maintenance costs of Information Systems (ISs) are high today and are growing. Part of the costs come from low software quality. The software quality can be measured when it is maintained. By improving the quality of maintenance prone modules future maintenance costs may be curtailed. In this article we will discuss how to measure software quality and we will also present a prototype that has been built to especially focus on the software quality factor maintainability. The fundamentals of measurement is discussed and also different software quality constructs found in the litterature.

Keywords

Software quality, measurement tool for IS management,
traditional software metrics, data collection

1 Introduction

Most of the yearly software budget is normally spent on maintenance of old information systems [Art88,Boe81,LS80]. Systems of poor quality demand more maintenance resources than systems of good quality. Development of new systems without strict requirement for quality control supported by quality measuring will

*e-mail: taanders@ra.abo.fi, fax:(+358) 21 654 732, tel: (+358) 21 654 475
†e-mail: atorn@finabo.abo.fi, fax:(+358) 21 654 732, tel: (+358) 21 654 220

lead to even bigger needs of maintenance resources in the organizations in the future.

The question of the complexity of software and maintenance is a vicious circle. Modifications of software are difficult because of the complexity of the software. Due to the complexity more modifications, i.e. maintenance, are often required [LB85].

In this article we will concentrate on how to measure software quality. The fundamentals of measurement together with different software quality constructs found in the litterature will be discussed.

We will present a prototype for measuring software quality. The prototype has been built especially focusing on the software quality factor maintainability.

The structure of the article follows. In section 2 we will discuss the notion of measurement. In section 3 we analyse different software quality constructs. In section 4.1 the discussion concerns how to measure quality, i.e. how to collect and store the data in a database relevant for this measure. Section 5 contains a discussion and a summary.

2 Measurement

Measurement is the key to management and control. By measurement the state of an attribute of an real world entity is classified by assigning a number of symbol to is. If the state is considered critical, apropriate actions may then be taken.

In our view the measures you use has to agree with your own conception of what you are measuring. Fenton [Fen91] asks how much you need to know about the attributes you want to measure, how do you know you have measured the right attributes, what can you conclude from the measurements, and what can you do with the measures. Fenton's answer is twofold. He introduces the term representation conditions as an answer to the two first questions and scales to the other two questions.

The representational theory of measurement is based on a relation system. Fenton uses the mathematical axiom of transitivity to be able to order the empirical observations of measurements. If for example liquid A is hotter than B, and B is hotter than C, then A should be hotter than C. This representation have then to be mapped to some numerical system. In everyday life we measure length, time, and temperature, but when it comes to software we often lack the necessary understanding and sophisticated tools to perform the measurements.

2.1 Scales

Statisticians recognize four basic types of scales: the nominal, the ordinal, the interval and the ratio scale. A problem is how to determine the type of scale the metrics belong to. For example the number of errors in software does not necessarily belong to the ratio scale. Conte, Dunsmore and Shen's [CDS86] counter example is whether a module with eight errors is "twice as error-prone" as a module with four. Similarly the number of errors may not be from an interval scale because the types of errors may be of a different kind. The ranking of erroneous software is not encouraged by Conte, Dunsmore and Shen either, because there will perhaps be many "ties", i.e. modules with no reported errors. Fenton adds the scale type of absolute to the list mentioned earlier.

3 Software quality

'Software quality' has been defined as follows:

ISO/DIS 9126 [ISO90]:

> The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.

Different approaches for modelling quality exists: hierarchical models, e.g. Boehm et al. [BBK+78], quantitative models, e.g. Gilb [Gil87], and product/process based models, e.g. Kitchenham [Kit87].

The hierarchical model by Boehm et al. is based on the relative importance of different aspects of software. The quantitative model by Gilb is based on the notion that quality and resource requirements should be measured directly and specified individually for each system. The quality concepts should be broken down to directly measurable attributes. The product/process based model is classified according to interest groups.

The same idea is present in the model by Eriksson and Törn [ET91]. This model for IS quality forms the bases for the SOLE-project[1]. The model "aims at a division of quality concepts consistent with the different decision makers and decisions made during the software life cycle". The main division is: IS Cost Effectiveness, IS Use Quality and IS Work Quality.

A high level classification of software into factors and criteria was first introduced by Gilb [Gil76] and Boehm et al. [BBK+78]. An often cited classification

[1]Software Library Evolution project (SOLE) started in 1988 and is led by Prof. A. Törn at Åbo Akademi University. The project is mainly financed by the Academy of Finland and the Foundation of Economic Education.

was done by McCall et al. [MRW77]. According to McCall et al. the software quality criteria are attributes of the software product or the software production process by which the factors can be judged. They describe the factors as functions of the criteria.

Kitchenham and Walker [KW86] writes about the classical principles of dividing quality into a number of quality factors. The quality factors themselves are broken further down into lower level quality criteria. The underlying quality metrics are the quantitative measures of the degree to which software possesses a given attribute that affects its quality. For a discussion on the quality factor maintainability see [AEA92].

3.1 Software metrics

In software, the term *metrics* is used to refer to the discipline of collecting and analyzing software data and to the actual measurement of software attributes, as well as the measurement scales. According to Kitchenham [Kit90] "metrics are scales or units in which a quantitative attribute can be measured". Attributes are features of the software product or process which can be observed. They provide some useful information about the status of the product or the progress of the project.

An attempt to formally define the terms measures and metrics can also be found in the joint report by GMD and NCC [CGI+84]. The formal definitions are based on set theory. Their aim by giving the formal definitions is to try to avoid the misinterpretations between metrics and measures.

Fenton tries to distinguish between measures and software metrics in the following way. A metric is a simple attribute like length, number of decisions, number of bugs found, a.s.o. whereas a measure is a function of metrics which can be used to predict complex attributes like cost or quality.

The different scale types mentioned in section 2.1 determine what kind of mathematical operations we can perform on these measures. The meaningfulness of these measurements are thus essential. We can compute mean values on ratio scale measures, but only medians for ordinal scale measures.

There are three different classes of metrics to measure from the management viewpoint of quality.

- Product,
- Process and
- Resources.

Products are the results, i.e. the deliverables like the software code, and documentation. The process consists of the tasks that delivers the products. The resources include personnel, materials, tools, and methods among other things that are used during the process. For a discussion about different views on software quality, see [ET91,ISO90].

4 A data collection system for measurement

In the following subsections we present the concept of data collection, our selection of maintainability metrics, and our prototype for measurement data collection.

4.1 Data collection

Large companies often collect data to produce daily or weekly progress reports of their projects. Small and medium size organizations should also be encouraged to collect data on their software development processes and the products produced. The data collected could be used as project management information, and thus be useful on a short-term basis. Most product oriented metrics discussed in the previous section are useful only on a long-term basis in estimating future project costs, effort and time. These could be used in steering and controlling the work in IS developing organizations.

There is no point in collecting data before the goals are specified. Kitchenham [Kit90] writes:

> "If you can't use the numbers don't collect them."

The software quality factors and criteria of interest have to be determined. The application portfolio varies among organizations, as do the factors and criteria of interest. Therefore the decisions are always site specific.

Humphrey [Hum89] has proposed a "Data Gathering Plan" where one of the principles deals with issues such as "what data is needed, by whom, and for what purpose". Fenton adds the idea of selling measurement to the personnel in the organization as one of the most crucial success factors.

After these actions the process of data collection can start. The metric values will be stored in a metric database [Hum89,Kit84,Ros90]. Ross [Ros87] has the following steps for data collection, where you ensure that data is captured as it occurs (during phase breakpoints), store the data in a format that permits easy analysis, use a database in which to keep software metrics, integrate a graphical statistics package with the database for routine reports, do some data interpretation recording, and give feedback of the results to all participants (works as a motivator).

Kitchenham [Kit90] has proposed that after data collection, analysis of the collected data can take place when you compare the reported values with the quantitative targets, define acceptable metric value ranges, verify the collected data, identify the components that are outside the specified ranges, investigate the unusual measured results, let a software development expert interpret the results, and start rework or repair of the components and possibly reschedule the project.

The interpretation of the measurement results is very crucial. Extreme values are not necessarily indicators of problems or possible error-prone modules. Large effort values correlate normally positive with large size values, and large size values correspond normally positive with large complexity values. Small values on documentation or some design metrics may indicate insufficient testing or inexperience of the developer. On the other hand simple functions or reused code might result in extreme values, but these values will not necessitate any actions. Thus expert interpretation is necessary.

4.2 Quality criteria

We combine the traditional hierarchical approach to model software quality with Tom Gilb's direct approach and choose a set of traditional criteria for our purposes. The set consists of the following criteria:

1 the structure of the system,

2 readability of source code and the documentation,

3 the size of source code and the documentation,

4 the complexity of the system,

5 number of errors,

6 where applicable, the same criteria as above in earlier phases of the process.

Our selection is based on critical analysis of what has been proposed in the literature in this field. We find support from a survey reporting, among other things, ways of improving maintainability [Kur90]. The main results are; a need for documentation, comments and clear descriptions (86%) and for clear program structure, code and coding technique and modularity (19%), while other aspects reported have received minor percentage of the answers (5–10%).

It is important to know the structure of the system and to identify the central routines, since often used modules are critical for the whole system. A system is easier to maintain if the readability of source code is high, this is true also

regarding the documentation. Most time spent in maintenance is spent reading the code and trying to understand it. This could amount to as much as 60 %. The size measures bring knowledge about the modularization of the whole system. Large size measures might also indicate complexity. Modules with large complexity measures are potential trouble spots in the future, and require forthcoming preventive maintenance. Even small concentrations of reported errors indicate a need for probable maintenance actions. The reports may also indicate a need for better education or support for the users or a need of better quality of the handbooks, online help screens, and so forth. These questions are, however, not the scope of this article.

Corresponding metrics, where applicable earlier in the process, have the following features. The proposed structure is possible to measure in the design phase. The readability of documentation could be useful in measuring the design proposals. The size values of code are naturally not measurable before the implementation phase, but the amount of documentation is. The lack of documentation will have serious consequences in the future. The complexity is measurable in some sense from the pseudo-code. Reported errors are a measure of the product quality. The earlier the errors are detected during the different stages in the assessment procedure the cheaper their correction will be.

4.3 Maintainability metrics

Existing findings of software metrics studies are applied in developing our model. For our study we have derived the following metrics as most relevant for maintainablity:

- Structure — Henry and Kafura's Information Flow metric,
- Readability — Gunning's Fog Index,
- Size — LOC, Halstead's volume and length, and size of documentation,
- Complexity — McCabe's $v(G)$ as a complexity density metric,
- Number of errors reported,
- The corresponding attributes metrics earlier in the process.

We have chosen the metrics above because they are well known and commonly accepted. The fan-in and fan-out values in Henry and Kafura's Information Flow metric [HK81] show how often a module calls other modules and how often a specific module is called from other modules. In this way critical modules can be identified. A system is easier to maintain when the documentation and

the source code are readable. Gunning's Fog Index [Gun68] is used for such measuring. The size measures are easy to collect and many surveys have shown the strengths of a simplistic measure like LOC and one of Halstead's Software Science metrics [Hal77]. Also when it comes to the complexity measures we use the classic well tested metrics by McCabe [McC76] and also as a density metric [GK91]. The earlier in the development process the errors are detected in walk throughs, unit tests or integration test, the better for the end result and the total costs. The process attribute metrics are monitored at different phase breakpoints in the development life cycle. The breakpoints are sometimes referred to as milestones in a project. We use the same kind of metrics in principle both for process and product measuring, but collect the metric values using slightly different objectives.

Not only support, but also criticism has been presented against metrics and measuring. A brief critical summary can be found in [Sør89].

4.4 The prototype

In this section we will describe the data collection part in the measurement process of the software quality. We will shortly discuss how the measurement of the metrics mentioned in section 3.1 are performed.

The prototype was built and is running on a IBM ES/9000 series host system. The prototype was built using the procedural language REXX and runs under IBM's VM/SP operating system. The data is collected from the software libraries of a production environment running IBM's MVS/ESA environment with online and batch routines and CA's IDMS network database.

The collection instrument is twofold. The automatic part analyses COBOL and ADS/Online code in a software library and also collects necessary administrative data. The error reporting part is still manual.

Henry and Kafura's Information Flow metric describes the structure. The metric consists of the $fan{-}in$ and $fan{-}out$ values. Keywords, i.e. reserved words in COBOL and ADS/Online, trigger the measurement counts in our system. In COBOL the $fan{-}in$ value will increase when words like READ, for reading files, and GET, OBTAIN, and so on for database reads. The $fan-out$ value will increase when words like DELETE and WRITE are encountered.

The size metrics are calculated by using the definition of noncomment source lines of code in Conte, Dunsmore and Shen [CDS86, p. 35]. Document lines of code are also calculated. We also use Halstead's volume and length as size metrics.

The cyclomatic complexity metric $v(G)$ by MaCabe has to be calculated in order to get the complexity density measure. We use the simpler formula and

calculate the branching instructions instead of using the graf-theory method.

Some administrative data has to be collected also. By administrative data we mean the name of the routine that is examined, to name of the IS it is part of, language used, type of routine, average monthly usage, date created, date last modified, creator and last modifier. Most of these are straight forward things except monthly usage. Depending on the environment you either have to extract this from logs or you get it from the system tables. An extract of the data collected can be found in Appendix A.

The incidents, or error reports are still collected manually, and are stored using a simple data entry routine that does elementary data validity checking. The incidents are classified according to the model presented by Lyytinen and Hirschheim [LH87].

5 Discussion

To summarize, in our model we use existing findings of software metric studies. Our goal is to use the measured values to help steer the maintenance work in IS-developing organizations. We combine structure, readability, number of errors, size and complexity metrics to measure the quality factor maintainability.

The collected measurement data is stored in a relational database. Periodic reports will be retrieved from this metric database. The "top ten" worst modules in a system of the application portfolio will be listed according to agreed-upon criteria. We will concentrate on creating standard reports (views). A few examples of standard reports are shown in Appendix B. The example reports feature complexity density measures and incident occurrences.

We have discussed how to measure software quality and discussed why it is important. A brief discussion about the fundamentals of measurement and different software quality constructs has also been presented. A prototype for measuring software quality has been presented. Future research will focus on an expert tool to guide IT-managers, project managers, systems analysts, and programmers in dealing with measurement results.

Acknowledgements

A special thanks to Jussi Hirvonen for his valuable work in building the prototype of the data collection system, to Inger Eriksson for her continuous support, and to Kaisa Sere for her constructive critique.

Appendix A. Collected Data

The automatically collected measurement data in our model classified according to the quality metrics:

Identifiers:

$System$	is the name of the project/system,
$Program$	is the name of the main routine,
$Module$	is the name of the subroutine,

Structure:

$fan-in$	and
$fan-out$	are elements in Henry and Kafura's Information Flow,

Size:

$SLOC$	is the number of source lines of code,
$DLOC$	is the number of comment lines of code,
$n1$, $n2$, $N1$, and $N2$	are elements in Halstead's Software Science,

Complexity:

$v(G)$	is the cyclomatic complexity number,

Administrative information:

$language$	is the programming language or generator used,
$type$	is the type of routine, i.e. batch or online,
$avg.$	is the average number of times the routine is used per month,
$cdate$	is the creation date of the subroutine,
$mdate$	is the modification date of the subroutine,
$creator$	contains the name of the creator, and
$modifier$	contains the name of the modifier.

The manually collected incident report data in our model is:

$Program$	and
$Module$	as above,
$edate$	is the date of the incident,
$etype$	is the type of the incident,
$reason$	is the reason for the incident,
$conseq$	is the consequence of the incident, and
rem	is a remark.

Appendix B. Model reports

Program	*Module*	*density*	$v(G)$	*SLOC*
DIHL0101	PRHL0101	0.207	12	58
TKIP007	TKIP006	0.202	37	183
DIHL0103	PRHL0103	0.191	9	47
TMAK022	TMAK023	0.085	12	142

Table 1: Complexity density

Legend: *Program*, *Module*, $v(G)$, and *SLOC* as in Appendix A,
density is the cyclomatic complexity density.
The rows are ordered by descending values on *density*.

System	*Program*	*Module*	*language*	*type*	*avg.*
BIG1	DIHL0101	PRHL0101	ADSO	Online	486.90

Table 2: Background information

Legend: *System*, *Program*, *Module*, *language*, *type*, and *avg.* as in Appendix A.

System	*Program*	*Module*	$\#ofinc.$	*sdate*	*edate*
BIG1	DITAL111	PRTAL111	3	03051991	31051991

Table 3: Incidents of type 1

Legend: *System*, *Program*, and *Module* as in Appendix A,
$\#ofinc.$ is the number of times the incident has occurred,
sdate is the start time of the measurement interval, and
edate is the end time of the measurement interval.

References

[AEA92] Thorbjörn Andersson, Inger Eriksson, and Donald L. Amoroso. Steering the maintenance costs: An exploration of the maintenance construct. In Jay F. Nunamaker Jr. and Ralph H. Sprague Jr., editors, *Proceedings of the Hawaii International Conference on System Sciences*, pages 348–358, CA, January 1992. IEEE Computer Society Press. Vol IV.

[Art88] L. J. Arthur. *Software Evolution, the software maintenance challenge*. John Wiley & Sons, New York, 1988.

[BBK+78] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, and M.J. Merrit. *Characteristics of Software Quality*, volume 1 of *TRW Series of Software Technology*. North-Holland, 1978.

[Boe81] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, New Jersey, 1981.

[CDS86] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publ. Comp., CA, USA, 1986.

[CGI+84] M-L Christ, H. Geist, W.D. Itzfeldt, D. Law, M. Schmidt, M. Timm, and R. Watts. Software Quality Measurement and Evaluation. Project MQ Final Report, GMD and NCC, St. Augustin, FRG and Manchester, UK, March 1984. SDSS-project: Measuring Quality of Software Products and Software Production Aids (MQ), Volume II.

[ET91] I. Eriksson and A. Törn. A Model for IS Quality. *Software Engineering Journal*, pages 152–158, August 1991.

[Fen91] Norman E. Fenton. *Software metrics: A rigorous approach*. Chapman & Hall, London, 1991.

[Gil76] T. Gilb. *Software Metrics*. Studentlitteratur, Lund, Sweden, 1976.

[Gil87] Tom Gilb. *Principles of software engineering management*. Addison-Wesley, 1987.

[GK91] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complecity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, December 1991.

[Gun68] R. Gunning. *Technique of Clear Writing*. McGraw-Hill, New York, revised edition, 1968.

[Hal77] M.H. Halstead. *Elements of Software Science*. North-Holland, New York, 1977.

[HK81] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. The SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, 1989.

[ISO90] ISO/IEC. Information technology – software product evaluation – quality characteristics and guidelines for their use. Technical Report 9126, International Organization for Standardization, 1990. Draft.

[Kit84] B.A. Kitchenham. Program history records: a system of software data collection and analysis. Technical journal, ICL, May 1984.

[Kit87] B.A. Kitchenham. Towards a constructive quality model. *Software Engineering Journal*, 2(4):105–113, 1987.

[Kit90] B.A. Kitchenham. Software metrics — lecture notes. Part I & II, April 1990.

[Kur90] Seppo Kurkinen. Atk-systeemien ylläpidon apuvälineeet (maintenance tools for computer systems). Research report A:17, Tietotekniikan kehittämiskeskus ry, Helsinki, May 1990. (in Finnish).

[KW86] B.A. Kitchenham and J.G. Walker. The meaning of quality. In D. Barnes and Brown P., editors, *Software Engineering 86*, pages 393–406, London, UK, September 1986. Peter Peregrinus Ltd.

[LB85] M.M. Lehman and L.A. Belady. *Program Evolution – Processes of Software Change*, volume 27 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 1985.

[LH87] Kalle Lyytinen and Rudy Hirschheim. Information systems failures – a survey and classification of the empirical litterature. In *Oxford Surveys in Information Technology, Vol 4*. Oxford University Press, 1987.

[LS80] B.P. Lientz and E.B. Swanson. *Software Maintenace Management, A Study of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, 1980.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[MRW77] J.A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality. Technical report CDRL A003, US Rome Air Development Center, November 1977. Volume I.

[Ros87] N. Ross. Collection and use of software data. In B.A. Kitchenham and B. Littlewood, editors, *Proceedings of the Centre for Software Reliability Conference entitled: Measurement for Software Control and Assurance*, pages 125–154, London, September 1987. Elsevier Applied Science.

[Ros90] N. Ross. Metricating a software production process. In *Conference Papers of Software Quality Workshop*, pages 119–133, Dundee, Scotland, June 1990.

[Sør89] Pål Sørgaard. An overview of research in maintenance. Reports on Computer Science and Mathematics 94, Department of Computer Science, Åbo Akademi University, Åbo, Finland, 1989.