International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 1 Issue 5, July - 2012

Implementing Source Code Metrics for Software quality analysis

Mandeep K. Chawla
Research Scholar
Department of Computer Science
Panjab University, Chandigarh

Indu Chhabra
Associate Professor and Head
Department of Computer Science
Panjab University, Chandigarh

ABSTRACT

Developing a high quality software product in an economical way is one of the fundamental goals of any software engineering activity. As computers are being used in almost every conceivable area in the contemporary world, quality of software becomes a key factor in the strategic success of a business and human security in general. Finding determinants of software quality and mapping them into quantitative measures is a crucial factor in sustainable success of an end product. Software metrics as means of quality analysis has attracted a lot of attention among researchers and practitioners in last one decade. Mapping of program characteristics into these metric values indicate structural complexity and behavior of an information system. In this case study, the five software metrics- lines of code (LOC), cyclomatic complexity (MVG), Halstead volume (HV), number of modules (NOM) and lines of comment (COM) have been utilized to analyze a set of three java based sorting programs. Three software measurement tools have been applied on them to judge their performance with respect to the metrics mentioned therein. Also a derived metric maintainability index has been calculated from the base metrics to indicate relative maintainability of the source code. Comparative analysis of the chosen tools have also been undertaken to reveal how they differ in delivering results for the same programs. Further, some other quality factors which can be derived from the constituent metrics are mentioned in a later sub-section.

1. INTRODUCTION

Software engineering is fairly intellectual and crucial design process because of today's dynamic environment which is quite unpredictable and in principle, not fully specifiable in advance. Effective software quality evaluation requires determinants that describe what quality is and how it can be traced back to the development process or the end product itself. Software industry is gradually progressing towards a period of high maturity; where informal approaches to quality analysis can no longer work. Due to the revolutionary growth, customers are also recognizing its value and they are not willing to compromise on the qualitative aspects. Despite of all this, internal quality of a product may go unchecked or be deliberately compromised at times. Software metrics are primitive indicators to code quality which provide us with the means to take pro-active actions at the earliest stage possible, whenever project is moving off-track.

Quality has different interpretation for different people. Various quality standards exist which are applicable for the organizations involved in software development. ISO and IEEE are the most widely used standards in this field. ISO/IEC 9126 [1] defines functionality, reliability, usability, efficiency, maintainability and portability as quality characteristics for software products. IEEE has published a standard for the software quality metrics methodology [2]. IEEE defines Software Quality as - the degree to which a system, component, or process meets specified requirements and/or customer expectations. Further, Software metrics are instruments applied to a piece of software or its design specifications with the goal to achieve reproducible quantitative measurements, which may be further applied in cost estimation, project scheduling, debugging, quality assurance and alike.

2. BACKGROUND

Measurement is essential in any engineering domain, and there is no exemption to software engineering. Several researchers in the past have applied software metrics as key inputs to guide quality predictors. Henrike Barkmann [3] identifies correlation between several metrics from well-known object oriented metrics suites such as CK metrics, McCabe Cyclomatic Complexity and various size metrics, besides presenting possible thresholds. Yasunari Takai et al. [4] propose new software metrics based on coding standards violations to capture latent faults in a development. PA Judas [5] identifies a linear growth trend in software size for crewed space and aircraft, which can reasonably predict software size in similar future programs, using SLOC based data. Zhou Yuming and XU Baowen [6] investigate the relationships of size and complexity metrics with maintainability of open source software. S. Pradeep et al. [7] utilizes CK metrics, SLOC, COM metrics etc. to investigate the relationship between software metrics and defects. Domenico Cotroneo [8] demonstrates the relation between software aging and several static features of the software. Cesar Couto, Christofer Silva [9] discover evidences towards causality between software metrics (as predictors) and the occurrence of bugs. Yuming Zhou [10] re-examines the ability of complexity metrics to predict fault-proneness. Daniela Glasberg [11] validates OO design metrics on a commercial Java application. AK Pandey [12] has made use of LOC, MVG and Halstead metrics to classify the software module as fault prone or not. Zhou et al [13] concludes that LOC and WMC (weighted method McCabe complexity) are indeed better fault-proneness predictors than other lesser known complexity metrics SDMC, AMC. In his large empirical study of five Microsoft software systems, Nagappan [14] found that failure prone software entities are statistically correlated with code complexity measures.

3. METHODOLOGY FOLLOWED

There are two approaches to software measurement. One is focused on direct evaluation of the quality of end product produced during various processes; and in the second one, processes themselves are measured to inform on duration, cost, effectiveness and efficiency of software development activities. In this study, we intend to evaluate source code as end product for metric based analysis. To begin with, programs are selected for which metrics shall be empirically validated for. We have opted for three java based sorting programs from well established algorithms of Bubble sort, Selection sort and Quick sort. Then a suitable set of metrics of interest are chosen. This in order requires determination and pre-testing of tools which

ISSN: 2278-0181
Vol. 1 Issue 5, July - 2012

are language compatible, support given metrics and on the basis of availability. After implementing the tools and capturing metric values, a derived metric Maintainability Index (MI) is calculated from base metrics; results are compared and interpreted eventually. Figure 1 illustrates the methodology followed in this paper:

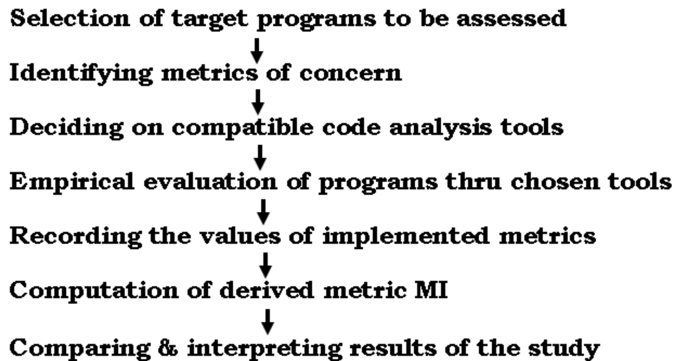

Figure 1: Methodology followed

There exist many open-source and commercial measurement tools to choose from depending upon the analyst preferences and other compatibility issues. In this paper, tools supporting the analysis of java programs were required. After some preliminary investigation, three tools have been selected – C and C++ Code Counter, Source Monitor and JHawk. For the sake of concision, they are identified as CCCC, SM and JHK respectively from this point. One reason behind opting for multiple tools is to put across the differences and similarities prevailing among them in delivering results. Tools produce many metrics values out of which results of five metrics of interest are recorded, and these are: Line of count (LOC), cyclomatic complexity (MVG), number of modules (NOM), Line of comments (COM), Halstead Volume (HV). Out of these 5 constituent metrics, three have further been utilized to calculate MI as function of LOC, MVG and HV. A few other derivable quality factors are also summed-up along with.

4. METRICS UNDER CONSIDERATION

A clear understanding of the characterization of code attributes and potential of their application in improving outcome of prospective projects led to a body of research principally converging on validation of these metrics. Further these are capable of reducing subjectivity during quality assurance and helps in decision making due to their nature of reproducibility. There exist several direct and indirect measures, out of which five metrics have been opted for the tools to be examined. Ahead is a brief description of them.

4.1 Line of Count (LOC) – Physical Size

This one of the most popular size-oriented metric represents total number of non-blank, non-comment lines. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input to evaluate other aspects of cost and quality [15].

4.2 McCabe's Cyclomatic Complexity (MVG)

Originally developed by Thomas McCabe, this widely used measure counts linearly independent paths through a flow of control graph. This can be found by counting language keywords and operators which affect on source code complexity [16]. Cyclomatic complexity [15] has a foundation in graph theory and provides us with extremely useful logical metric. Cyclomatic complexity, V(G), for a flow graph, G, is defined as

$V(G) = E - N + 2$

where E is the number of flow graph edges, N is the number of flow graph nodes.

It provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements. Hence it offers a quantitative measure of testing difficulty and an indication of ultimate reliability. Experimental studies indicate distinct relationships between the McCabe metric and the number of errors existing in source code, as well as time required to find and correct such errors.

4.3 NOM (Number of modules) –Code Distribution

All the functions, procedures or subroutines are counted under this physical as well as logical metric. As compared to LOC, it is more meaningful a size-metric because to some extent, it is independent of the programming language opted for. It is easy to calculate and serves best as an interface metric. The more modules a class has, more complex its interface is assumed to be [17].

4.4 COM (Lines of Comments)- Documentation

A well documented software aids developers and maintainers equally well. COM represents the total source comment count and further as an attribute to the measures - reusability, maintainability and understandability. Another useful metric called 'Code to Comment Ratio' (CCR) can be derived from this measure to have an estimate of how much the source code is well documented.

4.5 Halstead Volume (HV)

Halstead Volume, a measure from the family of Halstead metrics, is a composite metric based on the number of (distinct) operators and operands in source code [18]. According to Halstead, Volume is the count of number of mental comparisons needed to generate a program [Menzies et al.2002]. It is calculated as the program length times the 2-base logarithm of the vocabulary size. It represents the volume of information (in bits) required to specify a program. HV depicts textual code complexity and is one of the key parameter in computing maintainability index.

5. TOOLS DESCRIPTION

5.1 C and C++ Code Counter (CCCC)

CCCC [19] was developed in 2001 by Tim Littlefair as a part of his doctorate research project. It is free-ware open source command line interface originally meant for Linux, but also build-able on the Win32 platform. Originally implemented to process C++ and ANSI C programs, subsequent versions are able to process Java source files as well. It is easy to run on the command line by mentioning the names of one or more source files to be analyzed. CCCC will first check the extension of the filename, and if the extension is acknowledged as indicating a supported language, the appropriate parser will run on the file. As each file is parsed, identification of certain constructs will cause records to be written into an internal database. Final output will be generated in HTML format and XML files. CCCC produces various measures such as size metrics,

complexity metrics, object oriented metrics from CK and few others.

5.2 SourceMonitor (SM)

Developed by Campwood software with graphical-interface, SourceMonitor [20] is a free-ware closed-source software measurement tool. It is capable to be operated on ASCII text files created on other systems but runs only on Windows. Checkpointing is one of its distinct features to keep the results around so that project managers can see how the project code changes over time. There are five different views available to display the results such as Checkpoint view, Charts view, Project view, Details view and Method view. The languages supported are - VB6, HTML, VB.NET, C, C++, Java, C# and a few others. One can export resultant metrics data from SourceMonitor to text files, XML or CSV format. Metrics support vary somewhat with programming language chosen, however most commonly captured ones are- LOC, Methods per Class, Classes and Interfaces, Maximum Method Complexity, Percent Branch Statements and Percent Lines with Comments.

5.3 JHawk (JHK)

Primarily a Java metric tool, JHawk [21] has evolved from a stand-alone GUI application to include a command line version and an Eclipse plugin. It offers to produce IDE integration (for Visual Age for Java) and provides the CSV, XML and HTML export formats. Apart from letting the users create their own new metrics, it provides a dashboard tab which gives a quick overview of the metrics at System, Package and Class level. Also, the JHawk Dataviewer allows a user to view changes in core metrics over time – for example over a project lifecycle.

6. EXPERIMENTAL EVALUATION

Code analyses were performed after this preliminary study and pre-preparations. Three java programs based on three sorting techniques - Bubble sort, Selection sort, and Quick sort were analyzed through the tools undertaken. Brief description of the source programs is in Table I.

Table I: Source Programs description

Symbolic names of programs	Description
ProgA	Bubble sort
ProgB	Selection sort
ProgC	Quick sort

Each program is evaluated through all the three tools so that results can be compared across distinct tools. According to the individual tool's metric support, numerous metrics values were calculated and delivered automatically as part of results. However only the metrics of interest were captured and recorded in Table II for further investigation.

Table II: Results of tools' implementation

Tools	Prog	LOC	MVG	NOM	COM	HV
CCCC	ProgA	57	5	3*	1	
	ProgB	30	5	2*	2	-
	ProgC	45	11	2*	3	
Source Monitor (SM)	ProgA	44	4	4	1#	
	ProgB	32	4	2	2#	-
	ProgC	40	9	2	3#	
JHawk (JHK)	ProgA	47	5	4	1	318.0
	ProgB	36	4	2	2	519.7
	ProgC	42	8	2	3	727.3

- indicates metric is not supported by corresponding tool
 # indicates normalized values according to Table III (row 4, col2)
** indicates different granularity level according to Table III (row3, col2)*

……………………………………………………………

It is apparent in Table II that for the same program, identical metrics produce different results. This is because of the fact that all tools hold varying assumptions about their metric definitions and accordingly, outcomes moderately differ across each other. In spite of this, we can discern interesting similarities between them as mentioned in Table III. Note that HV is supported by only one of the tool, so is excluded from the comparative analysis in the following table.

Table III: Comparative analysis of tools against metrics calculated

Metric	Concluding observations w.r.t CCCC, SM and JHK
LOC	Out of three, SM provides most optimistic LOC value. CCCC counts all non-blank lines and curly brackets {, } as part of LOC while SM counts non-blank lines only and does not consider curly brackets under the label 'Statements'. JHK counts the same under the label LLOC as SM does. JHK differs from SM in the way it counts the 'for' statement.
MVG	CCCC measures it class-wise and picks the maximum as final value. SM measures module-wise and reports the result as 'maximum complexity'. JHK calculates the metrics quite nearer to SM. Since no two tools agreed to a common value for MVG, we tested the programs with one anonymous well established quality analysis tool. It validates the result of SM's analysis.
NOM	CCCC measure for NOM is not comparable to its counterparts because it counts number of classes as against others two which count number of functions and procedures spanning over all the classes in a program. Since a method undoubtedly is at a finer granularity level than a class, we confirm the result of SM and/or JHK analysis in this case.
COM	SM reports this metric in percentage form, it has been converted into fixed value before entering into table by taking two other metrics 'Lines (including comments)' and 'Percent line with comments' as input parameters. CCCC and JHK directly returns the result in absolute figures and convenient to counter-check. Among all, this metric remains the most stable of all.

7. RESULTS AND INTERPRETATION

The results of the experimental evaluation are briefed in Table IV.

Table IV: Program characteristics

Programs	Size	Logical Complexity	Documentation	Volume
ProgA	Largest	less complex	Poor documented	Small
ProgB	Smallest	less complex	Few comments	Medium
ProgC	middle-sized	most complex	well documented	High

Metrics characterize various program features objectively. They may be classified by their volume or size, interdependence among the modules or intricacy of flow control in each program module and a lot more. These measurements become more meaningful if some significant quality attributes could be further derived from the base metrics. In next sub-section, we attempt to compute one such composite metric to indicate relative maintainability which is one of most sought-after quality factor for the project managers.

7a. MAINTAINABILITY

Maintainability Index (MI) [22] is a composite metric that incorporates a number of traditional source code metrics into a single number that indicates relative maintainability. It is calculated with certain formulae from LOC, MVG and Halstead volume (HV). The metric originally is calculated as follows:

*MI = 171 - 5.2 * ln(aveV) - 0.23 * (aveMVG) – 16.2 * ln(aveLOC)*

where 'ave' is average of the measure per module. To reset this measure to lie between 0 and 100, it has been normalized as-

MI = MAX(0,(171 - 5.2 * ln(aveV) - 0.23 * (aveMVG) - 16.2 * ln(aveLOC)))*100/171*

It calculates an index value between 0 and 100 that represents the relative ease of maintaining the source code. A higher value suggests better maintainability. Table V records MI values calculated for all programs.

Table V: MI computation

Programs	Calculating MI	Result
ProgA	MAX(0,(171 - 5.2 * LN(317.98)-0.23 * 2.75 - 16.2 * LN(10.5))) * 100/171	59.83
ProgB	MAX(0,(171 - 5.2 * LN(519.69)-0.23 * 3.5 - 16.2 * LN(18))) * 100/171	53.13
ProgC	MAX(0,(171 - 5.2 * LN(727.36)-0.23 * 5.5 - 16.2 * LN(21))) * 100/171	50.38

According to Table V, ProgA (Bubble sort) has highest degree of maintainability among the threesome and ProgC (Quick sort) is most difficult to maintain. ProgB (Selection sort) comes in between the line. One can notice that these observations come quite in concordance with the program characteristics in Table IV. Quick Sort carries highest complexity in source code, largest volume and hence results in lowest maintainability index in Table V. Bubble sort is easiest to program, less complex and of least volume and scores highest MI. Trade-off remains similar for Selection sort too. Therefore the algorithmic traits and resultant attributes reveal that our programs are successfully tested for the said measures.

7b. OTHER DERIVED PARAMETERS

Though the attributes measured in Section 7 may not directly define the quality however they can be utilized to derive parameters signifying the potential changes to be carried out in the final product. Some of the quality factors which can be determined by these code attributes are summed up as under:

i) Correctness: Once LOC is calculated, it can prove useful to derive other code attributes such as Defects per KLOC. It estimates the defect density and eventually the ‘correctness’ which is one of the main quality metrics.

ii) Programming effort and Cost: Project cost per KLOC is another useful cost metrics derivable from LOC. For instance, assuming $2.00 per LOC, the pure coding cost can be evaluated for ProgB as $88 (presuming SM’s LOC measure).

Also based on degree of complexity (MVG), programmer’s effort and subsequently cost estimation can be developed. MVG indicates the breadth of functional coverage of the software.

iii) Fault prone-ness: One of the key aims of complexity metrics is to predict modules that are fault-prone. Based on MVG, residual defect prediction can be made. The more complex a system is, more challenging it is to test it fully and more error-prone it is.

iv) Modularity: NOM reveals the logical design intricacy of the system. It quantitatively describes how well modularized the software system is.

v) Usability: A program augmented with appropriate comments increases usability and readability during development process. A derived metrics ‘Comment density’ from COM is assumed to be a good predictor of maintainability and hence survival of a software project [23].

8. DIFFICULTIES EXPERIENCED WHILE COMPARING TOOLS

- Same metrics carry distinct names in different tools. So efforts took to identify the similar ones.
- Each tool bears different assumptions while measuring metrics. Some of these were cited in the tools’ guidebook while for others, manual code inspections had to be done to ensure their legitimacy.
- Tools either quantify the attributes at different granularity level or report figures in different forms, which makes them difficult to compare without some normalizations. Such as CCCC counts the number of classes for ‘NOM’ while SM and JHK counts number of methods; SM provides COM in percent form while others result in fixed value.

9. THREATS TO VALIDITY

Software quality is a multi-facet concept. Like any experimental study, our findings may be biased according to what primitive data was used to produce them. Possible traces of bias include, programming language selected, source code representative-ness, choice of tools and their measurement precision. There are numerous other tools available which could have been taken into account. Nonetheless there is little chance that choice of different data set or tools would altogether change the inferences drawn because most tools do not vary significantly in their estimation accuracy. However, we encourage readers to test

programs for different metrics, on different programming languages and other promising tools.

10. CONCLUSION AND PERSPECTIVES

This paper entails the evaluation of five software metrics on a set of three well-known sorting techniques through three automated analysis tools. It is followed by derivation of Maintainability index from component metrics and a brief determination of other quality factors which can be inferred. Undoubtedly software metrics are economical instruments available to management for decision making purposes and making them capable of taking pro-active action in case of prospective software crisis by stating early indicators to risk-prone issues. Yet project managers should formulate their own tailor-made metrics program to address company's unique strategic goals, priorities, clients' custom needs and expectations to fully utilize their enormous worth. Our study enhances prior empirical literature on software metrics validating the association between software metrics and quality attributes derived thereon, presenting the pros and cons on choosing automated tools which are available in large number. Like most other research in this stream, our study has several limitations. Our analysis covers only a subset of metrics and tools. This research needs to be further extended with large number and size of software sets to evaluate many more measures of performance.

11. REFERENCES

[1] ISO/IEC 9126-1 Software engineering – Product Quality - Part 1: Quality model", 2001.
[2] IEEE Std 1061-1998 "IEEE standard for a software quality metrics methodology, IEEE publications.
[3] H. Barkmann, R. Lincke, and W. Löwe. "Quantitative Evaluation of Software Quality Metrics in Open-Source Projects". In Proceedings of The 2009 IEEE International Workshop on Quantitative Evaluation of large-scale Systems and Technologies (QuEST09), Bradford, UK, 26-29th May, 2009.
[4] Yasunari Takai, Takashi Kobayashi, Kiyoshi Agusa. "Software Metrics based on Coding Standards Violations", In Proc. the Joint Conference of the 21th International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement (IWSM/MENSURA2011) pp.273-278, Nara, Japan, 3-4 Nov. 2011
[5] Paul A. Judas, Lorraine E. Prokop. "A historical compilation of software metrics with applicability to NASA's Orion spacecraft flight software sizing". ISSE 7(3): 161-170 (2011)
[6] Yuming Z., Baowen X. "Predicting the Maintainability of Open Source Software Using Design Metrics". Wuhan University Journal of Natural Sciences, Vol. 13 No.1, PP 14-20. 2008
[7] S Pradeep, Chaudhary K D and V Shrish. "An Investigation of the Relationships between Software Metrics and Defects". International Journal of Computer Applications 28(8):13-17, August 2011. Foundation of Computer Science, New York, USA.
[8] D. Cotroneo, R. Natella, R. Pietrantuono. "Is Software Aging Related to Software Metrics?" In proc. of the 2st IEEE International Workshop on Software Aging and Rejuvenation (WoSAR 2010) Ð in conj. with International Symposium on Software Reliability Engineering (ISSRE) 2010. San Jose CA, USA, Novembre 2010.
[9] Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto da Silva Bigonha, Nicolas Anquetil. "Uncovering Causal Relationships between Software Metrics and Bugs". CSMR 2012: 223-232
[10] Yuming Zhou, Baowen Xu, Hareton Leung. "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems". Journal of Systems and Software, 83(4), 2010: 660-674
[11] Glasberg, D., Emam, K. E., Melo, W., and Madhavji, N., "Validating Object-Oriented Design Metrics on a Commercial Java Application," National Research Council 44146, September 2000.
[12] AK Pandey. "Predicting Fault-prone Software Module Using Data Mining Technique and Fuzzy Logic". Special Issue of IJCCT Vol. 2, 2010
[13] Yuming Zhou, Baowen Xu, Hareton Leung. 2010. "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems". Journal of Systems and Software, 83(4), 2010: 660-674.
[14] Nagappan, N., Ball, T., Zeller, A. 2006.: Mining metrics to predict component failures. In ICSE(2006) 452-461.
[15] Roger S. Pressman, "Software Engineering – A Practitioner's Approach", 5th Ed., McGraw Hill International Edition.
[16] Yuhanis Yusof and Qusai Hussein Ramadan, 2010. Automation of Software Artifacts Classification. International Journal of Soft Computing, 5: 109-115.
[17] Wei Li, Sallie M. Henry. "Object-oriented metrics that predict maintainability". Journal of Systems and Software 23(2): 111-122 (1993)
[18] Tobias Kuipers and Joost Visser. "Maintainability Index Revisited - position paper", System Quality and Maintainability (SQM 2007), satellite of CSMR 2007.
[19] http://cccc.sourceforge.net.
[20] www.campwoodsw.com
[21] http://www.virtualmachinery.com/jhawkprod.htm
[22] Kurt D. Welker 2001. "The Software Maintainability Index Revisited". Crosstalk The Journal of Defense Software Engineering.
[23] Beat Fluri, Michael Wursch, and Harall Gall. 2007. "Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes." In Proceedings of the 14th Working Conference on Reverse Engineering (WCRE 2007). Page 70-79.