International Journal of Computing and Business Research (IJCBR)
ISSN (Online) : 2229-6166
International Manuscript ID : 22296166V8I1201803
Volume 8 Issue 1 January - February 2018

Deep Appraisal of Understandability and Reusability using Comments Analysis in Source Code

Jitender Singh Brar [1], Prof. S. S. Sarangdevot [2], Dr. Vishal Goar [3]

[1] Research Scholar
Janardan Rai Nagar Rajasthan Vidyapeeth University
Udaipur, Rajasthan, India

[2] Professor
Janardan Rai Nagar Rajasthan Vidyapeeth University
Udaipur, Rajasthan, India

[3] Assistant Professor
Department of Computer Applications
Government Engineering College, Bikaner, Rajasthan, India

Abstract

The deep assessment and multidimensional evaluation of software quality is one of the key domains of research in software engineering and project management to achieve the higher degree of performance and overall integrity of the software project. The software modules are overall projects are developed to achieve the higher optimization of cohesion and coupling which makes the reusability of the software to higher extent. In this research work, the soft computing based approaches are making the overall scenario of reusability quite effectual with the greater accuracy. The projected approach is making use of fuzzy as well as nature inspired approach to maintain the reusability. The software quality of the project is highly dependent on the source code which is written by the developers with assorted testing approaches. This work is having

multiple scenarios to evaluate the accuracy and quality of source code with the software metrics and integration of cohesion and coupling based reusability factors in the source code. The multiple scenarios and cases of source codes in Java, C, C++ and PHP are taken to analyze the performance of software source code for multiple dimensional based effectiveness and reusability factors.

Keywords: Source Code Quality, Software Quality, Software Project Management

Introduction

Software functional quality reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the fitness for purpose of a piece of software or how it compares to competitors in the marketplace as a worthwhile product. It is the degree to which the correct software was produced. Software structural quality refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability. It has a lot more to do with the degree to which the software works as needed. Some structural qualities, such as usability, can be assessed only dynamically (users or others acting in their behalf interact with the software or, at least, some prototype or partial implementation; even the interaction with a mock version made in cardboard represents a dynamic test because such version can be considered a prototype). Other aspects, such as reliability, might involve not only the software but also the underlying hardware, therefore, it can be assessed both statically and dynamically (stress test).

Functional quality is typically assessed dynamically but it is also possible to use static tests (such as software reviews). Historically, the structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the ISO 9126-3 and the subsequent ISO 25000:2005 quality model, also known as SQuaRE. Based on these models, the Consortium for IT Software Quality (CISQ) has defined five major desirable structural characteristics needed for a piece of software to provide business value: Reliability, Efficiency, Security, Maintainability and (adequate) Size.

Software quality measurement quantifies to what extent a software program or system rates along each of these five dimensions. An

aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. This view of software quality being positioned on a linear continuum is supplemented by the analysis of critical programming errors that under specific circumstances can lead to catastrophic outages or performance degradations that make a given system unsuitable for use regardless of rating based on aggregated measurements. Such programming errors found at the system level represent up to 90% of production issues, whilst at the unit-level, even if far more numerous, programming errors account for less than 10% of production issues. As a consequence, code quality without the context of the whole system, as W. Edwards Deming described it, has limited value. To view, explore, analyze, and communicate software quality measurements, concepts and techniques of information visualization provide visual, interactive means useful, in particular, if several software quality measures have to be related to each other or to components of a software or system. For example, software maps represent a specialized approach that can express and combine information about software development, software quality, and system dynamics.

Halstead Complexity Measures

Such measures are software metrics introduced by Maurice Howard Halstead in 1977 as part of the treatise on establishing an empirical science of software development. Halstead makes the observation that metrics of the software should reflect the implementation or expression of algorithms in different languages, but be independent of their execution on a specific platform. These metrics are therefore computed statically from the code. Halsteads goal was to identify measurable properties of software, and the relations between them. This is similar to the identification of measurable properties of matter (like the volume, mass, and pressure of a gas) and the relationships between them (analogous to the gas equation). Thus his metrics are actually not just complexity metrics.

Halstead complexity metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program modules complexity directly from source

code. Among the earliest software metrics, they are strong indicators of code complexity. Because they are applied to code, they are most often used as maintenance metric. There is evidence that Halstead measures are also useful during development, to assess code quality in computationally-dense applications. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. Halstead measures were introduced in 1977 and have been used and experimented with extensively since that time. They are one of the oldest measures of program complexity.
Halsteads metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand.

Then is counted

- number of unique (distinct) operators (n1)
- number of unique (distinct) operands (n2)
- total number of operators (N1)
- total number of operands (N2).

The number of unique operators and operands (n1 and n2) as well as the total number of operators and operands (N1 and N2) are calculated by collecting the frequencies of each operator and operand token of the source program." Other Halstead measures are derived from these four quantities with certain fixed formulas as described later. The classification rules of CMT++ are determined so that frequent language constructs give intuitively sensible operator and operand counts. All other Halsteads measures are derived from these four quantities using the following set of formulas.

Program length (N): The program length (N) is the sum of the total number of operators and operands in the program:

$$N = N1 + N2$$

Vocabulary size (n): The vocabulary size (n) is the sum of the number of unique operators and operands:

$$n = n1 + n2$$

Program volume (V): The program volume (V) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size (n) :

$$V = N * \log2(n)$$

Halsteads volume (V) describes the size of the implementation of an algorithm. The

computation of V is based on the number of operations performed and operands handled in the algorithm. Therefore V is less sensitive to code layout than the lines-of-code measures. The volume of a function should be at least 20 and at most 1000. The volume of a parameterless one-line function that is not empty; is about 20. A volume greater than 1000 tells that the function probably does too many things.

The volume of a file should be at least 100 and at most 8000. These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits. The limits of volume can be used for double-checking.

Difficulty level (D): The difficulty level or error proneness (D) of the program is proportional to the number of unique operators in the program.

D is also proportional to the ration between the total number of operands and the number of unique operands (i.e. if the same operands are used many times in the program, it is more prone to errors).

D = (n1 / 2) * (N2 / n2)

Program level (L): The program level (L) is the inverse of the error proneness of the program i.e. a low level program is more prone to errors than a high level program.

L = 1 / D

Effort to implement (E): The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.

E = V * D

Time to implement (T): The time to implement or understand a program (T) is proportional to the effort. Empirical experiments can be used for calibrating this quantity. Halstead has found that dividing the effort by 18 give an approximation for the time in seconds.

T = E / 18

Number of delivered bugs (B): The number of delivered bugs (B) correlates with the overall complexity of the software.

Halstead gives the following formula for B:

B = (E ** (2/3)) / 3000 ** stands for to the exponent

Halsteads delivered bugs (B) is an estimate for the number of errors in the implementation. Delivered bugs in a file should be less than 2. Experiences have shown that, when

programming with C or C++, a source file almost always contains more errors than B suggests. The number of defects tends to grow more rapidly than B. When dynamic testing is concerned, the most important Halstead metric is the number of delivered bugs. The number of delivered bugs approximates the number of errors in a module. As a goal at least that many errors should be found from the module in its testing.

Existing Theories for Software Defects Analysis and Predictions for Multiple Instances integrates the following

- The base work associated with software defect predictions is done
- The classical research is based on the analysis and evaluation of parameters based on the real time live projects and a dataset is prepared
- The classical fuzzy approach is used for software defects predictions on the perspective of multiple dimensions.

Analysis of Threshold and Acceptability Score

- The base algorithmic approach of fuzzy mathematical modelling is relying on the decision factor of a software defect
- The fuzzy logic is deciding the parameters which determining the threshold and limits of software defects using multiple parameters including MMRE and BMMRE
- Using acceptability score or threshold, the perspectives are under investigation for final predictive measures

Selection of the Simulation Tool / TestBed for Simulation and Testing of the Results

- To perform or implement any research work, the identification of suitable tool is important task
- The deep analysis and learning of the existing work determines that the MATLAB should be used. The key reason behind this decision is that MATLAB is having full compatibility and support for fuzzy logic as well as artificial neural network using its inherent toolboxes.

Generation of Dataset and Implementation of Data Cleaning

- The same dataset is used which is integrated in the classical approach and finally cleaned.

- Data cleaning means to find out and extract the meaningful attributes from the dataset so that effective and optimal results can be obtained.

Activation of Fuzzy and ANN Based Approach

- Using fuzzy logic toolbox (fuzzy) and artificial neural network toolbox (nntool), the dataset is initialized and activated. These toolboxes are used to train the model and finally devise the predictive reports.

Analysis of the Threshold and Current Acceptance of the Solutions

- Once the dataset is trained and the model obtained appropriate epochs for learning, the acceptability of the output is done.
- The acceptability of the output is checked with the parameters of gradient and regression analysis
- Minimum value towards zero is effective and giving minimum number of error rate

Research Methodology

- Data Set Formation from Assorted Sources
- Implementation of dataset using fuzzy logic in classical approach
- Features Extraction
- Identification of Key Aspects
- Development of a unique network for training
- Training of ANN Model
- Defects Analysis with the inherent parameters
- Deep Investigation and Predictive Analysis

The software defect prediction is one of the prominent domains in software engineering which is required to be processed using new and effective algorithms. The classical approaches related to fuzzy modelling can be improved using proposed artificial neural networks.

Research Goals

1. To perform the detailed comparative analysis of software defect prediction approaches and algorithms
2. To evaluate the performance factors of assorted techniques
3. Development of a real and dynamic code for the software defect prediction so that the real implemented results and output can be obtained

4. Evaluation of Real Time Datasets for software defect prediction with related parameters
5. Implementation of dynamic code with the integration of neural network and fuzzy approach on assorted scenarios

Simulation Scenario

LOC	Classical Approach	Proposed Approach
50	1.234235	0.892324

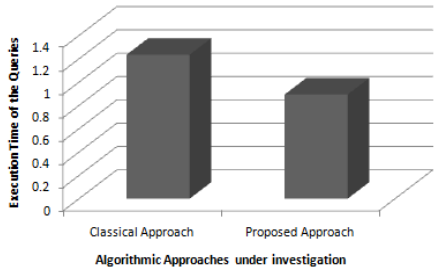

Figure 1: Average Execution Time Analysis of 50 LOC from the Classical and Proposed Approach

In the above drawn representation of the figure, it is very clear from the bar chart that the proposed approach is taking very less time as compared to the corresponding classical approach in every query execution attempt or implementation scenario or execution iteration. In this scenario, we have taken the average of 50 LOC executed on the live server fetching results.

Conclusion

Source Code Quality, Software defect prediction and related testing approaches are not new but in use from a long time since the inception of software development. The software applications are classically vulnerable to different types of bugs and defects because of programming, execution and logic based tasks. There is need to develop the effective mechanisms to evaluate the existing limitations and drawbacks in the current technologies so that further defects cannot happen. The work is having focus on multiple perspectives including source code quality and overall quality check of the software with the defects prediction. In this research work, a novel, effective and performance aware algorithmic implementation is done with the integration of artificial neural networks and found that the ANN based approach is giving better results than fuzzy logic based approach. The fuzzy logic based implementation is done in the existing work in which a set of rules are mentioned and based on these rules, the software defect prediction metrics can be evaluated. The proposed ANN based approach is effective than the classical fuzzy based approach and giving more accurate and

precision based values which are more effective and performance aware.

References

[1] D. Spinellis, “Code Quality: The Open Source Perspective”, Addison-Wesley, Boston - MA,2003.

[2] B. N. Corwin, R. L. Braddock, "Operational performance metrics in a distributed system", Symposium on Applied Computing, Missouri - USA, 1992, pp. 867-872.

[3] R.Numbers, "Building Productivity Through Measurement", Software Testing and Quality Engineering Magazine, vol 1, 1999, pp. 42-47

[4] IFPUG - International Function Point Users Group, online, last update: 03/2008, available: http://www.ifpug.org/

[5] B. Boehm, “Cost Models for Future Software Life Cycle Processes: COCOMO 2.0”, U.S.Center for Software Engineering, Amsterdam, 1995, pp. 57-94.

[6] N. E. Fenton, M. Neil, “Software Metrics: Roadmap”, International Conference on Software Engineering, Limerick - Ireland, 2000, pp. 357–370.

[7] M. K. Daskalantonakis, “A Pratical View of Software Measurement and Implementation Experiences Within Motorola”, IEEE Transactions on Software Engineering, vol 18, 1992, pp. 998–1010.

[8] R. S. Pressman, "Software engineering a practitioner's approach", 4th.ed, McGraw-Hill, New York - USA, 1997, pp. 852.

[9] I. Sommerville, “Engenharia de Software”, Addison-Wesley, 6º Edição, São Paulo – SP, 2004.

[10] D. C. Ince, M. J. Sheppard, "System design metrics: a review and perspective", Second IEE/BCS Conference, Liverpool - UK, 1988, pp. 23-27.

[11] L. C. Briand, S. Morasca, V. R. Basili, “An Operational Process for Goal-Driven Definition of Measures”, Software Engineering - IEEE Transactions, vol 28, 2002, pp. 1106-1125.

[12] Refactorit tool, online, last update: 01/2008, available: http://www.aqris.com/display/ap/RefactorIt

[13] O. Burn, CheckStyle, online, last update: 12/2007, available: http://eclipse-cs.sourceforge.net/index.shtml

[14] M. G. Bocco, M. Piattini, C. Calero, "A Survey of Metrics for UML Class Diagrams", Journal of Object Technology 4, 2005,pp. 59-92.

[15] JDepend tool, online, last update: 03/2006

[16] Metrics Eclipse Plugin, online, last update: 07/2005

[17] Coverlipse tool, online, last update: 07/2006

[18] JHawk Eclipse Plugin, online, last update: 03/2007

[19] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, M. V. Zelkowitz, B. Kitchenham, S. Lawrence Pfleeger, N. Fenton, "Towards a framework for software measurementvalidation", Software Engineering, IEEE Transactions, vol 23, 1995, pp. 187-189.

[20] H. F. Li, W. K. Cheung, "An Empirical Study of Software Metrics", IEEE Transactions on Software Engineering, vol 13, 1987, pp. 697-708.

[21] H. Zuse, "History of Software Measurement", online, last update: 09/1995,

[22] N. E. Fenton, M. Neil, "Software Metrics: Roadmap", International Conference on Software Engineering, Limerick - Ireland, 2005, pp. 357–370.

[23] T. J. McCabe, "A Complexity Measure". IEEE Transactions of Software Engineering, vol SE-2, 1976, pp. 308-320.

[24] D. Kafura, G. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", IEEE Transactions on Software Engineering archive, vol 13 , New Jersey - USA, 1987, pp. 335-343.

[25] B. Ramamurty, A. Melton, "A Syntheses of Software Science Measure and The Cyclomatic Number", IEEE Transactions on Software Engineering, vol 14, New Jersey - USA, 1988, pp. 1116-1121.

[26] J. K. Navlakha, "A Survey of System Complexity Metrics", The Computer Journal, vol 30, Oxford - UK, 1987, pp. 233-238.

[27] E. VanDoren, K. Sciences, C. Springs, "Cyclomatic Complexity", online, last update: 01/2007

[28] R. K. Lind, K. Vairavan, "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort", IEEE Transactions on Software Engineering, New Jersey - USA, 1989, pp. 649-653.

[29] G. K. Gill, C. F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity", IEEE Transactions on Software Engineering, 1981, pp. 1284-1288.

[30] M. H. Halstead, Elements of Software Science, Operating, and Programming Systems, vol 7, New York - USA, 1977, page(s): 128.

[31] B. H. Yin, J. W. Winchester, "The establishment and use of measures to evaluate the quality of software designs", Software quality assurance workshop on Functional and

performance, New York - USA, 1978, pp. 45-52.

[32] R. R. Willis, "DAS - an automated system to support design analysis", 3rd international conference on Software engineering, Georgia - USA, 1978, pp. 109-115.

[33] C. L. McClure, “A Model for Program Complexity Analysis”, 3rd International Conference on Software Engineering, New Jersey - USA, 1978, pp. 149-157.

[34] N. Woodfield, "Enhanced effort estimation by extending basic programming models to include modularity factors", West-Lafayette, USA, 1980.

[35] S. Henry, D. Kafura, "Software Structure Metrics Based on Information Flow", Software Engineering, IEEE Transactions, 1981, pp. 510-518.

[36] S. R. Chidamber, C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, vol 20, Piscataway - USA, 1994, pp. 476-493.

[37] M. Alshayeb, M. Li, "An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes", IEEE Transactions on Software Engineering archive, vol 29, 2003, pp. 1043–1049.

[38] R. Subramanya, M. S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implication for Software Defects”, IEEE Transactions on Software Engineering, vol 29, 2003, pp. 297-310.

[39] L. C. Briand, S. Morasca, V. R. Basili, "Property-based software engineering measurement", Software Engineering, IEEE Transactions, vol 22, 1996, pp. 68 - 86.

[40] S. R. Chidamber, D. P. Darcy, C. F. Kemerer, "Managerial use of metrics for object-oriented software: anexploratory analysis", Software Engineering, IEEE Transactions, vol 24, 1998, pp. :629–639.

[41] Mei-Huei Tang, Ming-Hung Kao, Mei-Hwa Chen, "An empirical study on object-oriented metrics", Software Metrics Symposium, 1999, pp. 242-249.

[42] M. Lorenz, J. Kidd, “Object-Oriented Software Metrics: A Practical Guide”, Englewood Cliffs, New Jersey - USA, 1994.

[43] A. F. Brito, R. Carapuça, "Object-Oriented Software Engineering: Measuring and controlling the development process", 4th Interntional Conference on Software Quality, USA, 1994.

[44] L. Briand, W. Devanbu, W. Melo, "An investigation into coupling measures for C++", 19th International Conference on Software Engineering, Boston - USA, 1997, pp. 412-421.

[45] R. Harrison, S Counsell, R. Nithi, "Coupling Metrics for Object-Oriented Design", 5th International Software Metrics Symposium Metrics, 1998, pp. 150-156.

[46] M. Marchesi, "OOA metrics for the Unified Modeling Language", Second Euromicro Conference, 1998, pp. 67-73.

[47] T. Mayer, T. Hall, "A Critical Analysis of Current OO Design Metrics", Software Quality Journal, vol 8, 1999, pp. 97-110.

[48] N. F. Schneidewind, "Measuring and evaluating maintenance process using reliability, risk, and test metrics", Software Engineering, IEEE Transactions, vol 25, 1999, pp. 769-781.

[49] V. R. Basili, L. C. Briand, W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, vol 22, New Jersey - USA, 1996, pp. 51-761.

[50] L. C. Briand, S. Morasca, V. R. Basili, "Defining and validating measures for object-based high-level design", Software Engineering, IEEE Transactions, vol 25, 1999, pp. 722-743.

[51] K. E. Emam, S. Benlarbi, N. Goel, S. N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics", IEEE Transaction on Software Engineering, vol 27, 2001, pp. 630-650.

[52] A. Chatzigeorgiou, "Mathematical Assessment of Object-Oriented Design Quality", IEEE Transactions on Software Engineering, vol 29, 2003, pp. 1050-1053.