

An Entropy-Based Approach to Assessing Object-Oriented Software Maintainability and Degradation – A Method and Case Study

Hector M. Olague¹, Letha H. Etzkorn², Glenn Cox³
Computer Science Department
University of Alabama in Huntsville, Huntsville, AL U.S.A.

Abstract –The term ‘software entropy’ has been anecdotally defined to mean that software declines in quality, maintainability and understandability though its lifetime. While there are numerous software metrics that assess “snapshots” of software maintainability, few assess software degradation at multiple, discrete points in the life cycle. Assessing object-oriented (OO) software degradation is more art than science. Recently studies have shown that OO software degradation may be assessed by measuring the increase in the number of “links”, or coupling, within an abstraction model and between abstraction models of the software. We believe that software degradation may also be measured using cyclomatic complexity since it has been shown to be highly correlated with fault-proneness of OO classes. We take the approach of defining software decay in terms of Shannon entropy and McCabe cyclomatic complexity using industry-established complexity threshold criteria. We use the Rosenberg WMC risk threshold criteria and the McCabe risk interpretation threshold criteria in our experiment. We applied this metric retrospectively to Mozilla Rhino, an open-source implementation of JavaScript written in Java. Our initial findings were inconclusive since the number of software revisions was limited. However, we conducted further analyses and showed that components with high cyclomatic complexities were associated with more maintenance activities than those components with lower cyclomatic complexities. Entropy scores showed the collection of OO classes requiring changes between software versions had a higher composite entropy score than those classes that did not undergo changes between software versions. Additionally, a pattern of repeated component modification was detected in our secondary analysis, indicating that possibly decision tree analysis may be more effective in analyzing software degradation.

Keywords: Information theory, entropy software metrics, software decay measurement, software entropy.

1.0 Introduction

Bianchi, et al. [5] proposed the measurement of software degradation through entropy. Their basis is that software degradation can be assessed by the increase in the number of “links” within an abstraction model and between abstraction models. The direct measures used to validate the entropy metrics are number of defects discovered, slipped defects, and maintenance effort. This concept shows positive results. Bianchi applies this metric to the software design as well as the source code. In environments where all of the artifacts of the software development process are present, this is an excellent approach. Of concern is the ease of computing results using this metric and the effect of not having regularly updated models at all levels of abstraction.

The only artifact that is guaranteed to survive the repeated process of software modification in most post deployment scenarios is the source code. For reasons primarily associated with cost, post deployment software modifications are typically poorly documented. Therefore, the Bianchi approach might be modified to consider only linkage increases within the source code. In the case of object-oriented (OO)

¹ holague@cs.uah.edu, (256) 955-1801

² letzkorn@cs.uah.edu, (256) 824-629

³ gcox@cs.uah.edu, (256) 824-6433

software, this is interpreted as an increase in the coupling between objects. Coupling metrics have been shown to be good predictors of software quality. However, to our knowledge, threshold values for interpreting the results have not been established.

Complexity metrics may be a better approach to assessing individual component and system-wide software degradation over several sequential software iterations. They have been studied for a longer period of time and empirical threshold values have been established by at least two groups. For all the contentious discussion about OO software complexity, it is generally agreed that complexity is the best predictor of maintenance and testing effort [1,3,7,28,29].

In addition, the assessment of software degradation may be more relevant in the context of agile software development processes. Examples are *extreme programming* and *timeboxing*. These methods of developing software are less rigorous than traditional processes and may result in fewer robust final products if care is not taken to keep the incremental products robustly designed and at a high level of quality.

In this paper, we propose examining software degradation using two widely accepted OO complexity measures in the context of empirical complexity threshold criteria to assess system-wide software degradation.

2.0 Goal Statement and Research Hypotheses

Two research hypotheses will be tested under the common goal:

Goal: *To measure OO software degradation using class and method complexity.*

Hypothesis 1 (H1): *There is a correlation between system-wide object-oriented software degradation and the Weighted Methods per Class (WMC) metric, expressed in terms of Shannon entropy, and error-proneness.*

Hypothesis 2 (H2): *There is a correlation between system-wide object-oriented software degradation and the WMC-McCabe metric, expressed in terms of Shannon entropy, and error-proneness.*

The Weighted Methods per Class metric is defined to be the summation of the complexities of the methods in a class [10]. It is intended to be a first-level approximation of the complexity of a class. Chidamber and Kemerer (C&K) [10] left undefined the exact meaning of “complexity,” leaving it up to the implementer to choose an appropriate definition. If complexity is taken to be unity (1) for each method, WMC for a class is equal to the number of methods in the class. This measure is known as WMC. If the complexity for each method is taken to be the McCabe Cyclomatic Complexity, WMC for a class is equal to the sum of the McCabe Cyclomatic Complexities of all the methods in the class.

System Category	McCabe CC Threshold (x)	Risk Interpretation
1	$1 \leq x \leq 10$	Usually simple procedures, little risk.
2	$11 \leq x \leq 20$	Moderately complex, moderate risk.
3	$21 \leq x \leq 50$	Complex, high risk.
4	>50	Not testable, very high risk.

System Category	C&K WMC Threshold (x)	Risk Interpretation
1	$1 \leq x \leq 20$	Good value of class complexity.
2	$20 \leq x \leq 100$	Moderately high value of class complexity.
3	$X > 100$	High class complexity, cause for investigation.

3.0 Developing an Entropy-based Measure of OO Software Degradation

Our goal is to establish a measure of OO software degradation that is easy to use and interpret. We attempt to do this using Shannon's entropy equation.

In order to test our first research hypotheses, we will use well-established McCabe thresholds as a basis for our metric. The interpretation of McCabe cyclomatic complexity in industry is contained in Table I. These criteria were originally developed for non-OO software and were applied to software modules. We will attempt to apply these criteria to classes in OO software using WMC-McCabe. WMC-McCabe is similar to an application of cyclomatic complexity in a traditional sense, since the complexities of individual functions are aggregated into the module's complexity value. In terms of OO software, category 1 of Table I, indicates that the class is comprised of simple methods (e.g. simple class constructors, simple class destructors, and other simple procedures). Since class constructors and destructors are part of every class, and would be expected to remain part of every class for its lifetime, this means that in almost every application constructors and destructors will guarantee that category 1 classes will not have an aggregate WMC score of zero.

To test the second hypothesis, we used the threshold criteria for WMC published by Rosenberg, et al. Software Assurance Technology Center (SATC), NASA Goddard Space Flight Center, in 1998 [26]. These thresholds were based on their experiences at NASA with OO projects. These thresholds are contained in Table II and will be used without modification in this study.

Categories defined in Tables I and II assign threshold criteria, and give an interpretation of *risk* for each category. The use of these thresholds in industry allows software managers to make judgments about the condition of their software in terms of level of effort to test and confidence in software deployment.

The Shannon’s entropy relationship we will examine for Tables I and II threshold criteria is expressed as follows:

$$-H_{SystemComplexity} = \sum_{i=1}^j \frac{S_i}{N} \log_2 \frac{S_i}{N} \quad (\text{eq.1})$$

Where

H_{System} = System Complexity Entropy.

i = Integer value 1,2,...,j, representing each of the categories considered.

S_i = Total number of classes that are in category S_i .

N = Total number of system classes (equal to the sum of all the S_i s).

The maximum entropy is achieved when $S_i = S_{i+1} = S_{i+2} = S_{i+3}$, or when all the classes are evenly distributed.

Shannon’s equation may be better than complexity averaging. Shannon’s equation “dampens” the effect of a few very highly complex methods to skew the overall complexity of the system. This is because the equation limits the contribution of the entropy score from each category to the overall (system) entropy score.

4.0 Experiment

We chose the Mozilla Rhino project for this study. Rhino is an open-source implementation of JavaScript written in Java. It is typically embedded in Java applications to provide scripting to end users. Rhino was started in 1997 by Netscape, to produce a Java version of the popular Navigator Internet browser. More history about Netscape can be found at [31]. Rhino versions 1.4R3, 1.5R1, 1.5R2, 1.5R3, 1.5R4 and 1.5R5 were analyzed and used in this study.

Table III. Rhino version statistics for study

Rhino Version:	1.4R3	1.5R1	1.5R2	1.5R3	1.5R4	1.5R5
Files:	94	149	218	110	113	112
Lines:	30940	48047	58742	47774	50241	52791
Blank Lines:	3048	4573	5537	4545	4781	4838
Code Lines:	20420	31819	39999	32459	33752	36097
Comment Lines:	8620	12981	14537	12014	12194	12306
Statements executable:	8890	13538	18600	15819	16962	17930
Statements declarative:	3857	5483	7148	5697	6138	6507
Ratio Comment/Code:	0.42	0.41	0.36	0.37	0.36	0.34
Number of Classes:	100	137	210	130	143	148
Number of Methods:	1155	1536	2085	1662	1760	1817
Defects Reported:	21	29	10	41	153	61
Enhancements Made:	1	3	0	0	76	37

In the Mozilla Rhino software we analyzed for this study, the software development strategy can be described at times as *agile* due to the incorporation of software extensions and improvements by generous third parties. Software additions to Rhino by third parties make the Rhino software development process resemble an agile process. New software enhancements usually are followed by new software defects due to the code integration and testing philosophy employed by agile systems (and Rhino) in general.

Table IV. RHINO complexity metrics correlation –vs- defects Pearson Correlation (P-value in parenthesis)		
Version	WMC	WMC-McCabe
1.4R3	0.331 (0.001)	0.497 (0.000)
1.5R1	0.544 (0.000)	0.725 (0.000)
1.5R2	0.306 (0.000)	0.588 (0.000)
1.5R3	0.550 (0.000)	0.680 (0.000)
1.5R4	0.772 (0.000)	0.731 (0.000)
1.5R5	0.545 (0.000)	0.424 (0.006)

Error data exists for Rhino in the online Bugzilla repository located at <https://bugzilla.mozilla.org/>. The change logs for each version of Rhino were examined, which lists the bugs that were resolved for that version of Rhino. Examining the change logs was much easier than trying to search Bugzilla initially since the change log contains query links back to the Bugzilla database. The change log allows us to search for searches for bugs targeted against each version of Rhino. Bug fixes were cross referenced with classes affected by each bug/fix. The Pearson correlation coefficient between defects and enhancements was 0.962, with p-value 0.002, indicating the software may be exhibiting signs of brittleness, or code decay, as enhancements seem to coincide with the emergence a large number of defects.

After error data was collected and analyzed for six versions of Rhino, we downloaded the source code for each version and used Understand for Java® to collect complexity metrics for each version. Some parametric summary data for each Version of Rhino is contained in Table III.

We used MiniTab® to conduct preliminary correlation analysis using the most common OO complexity-related metrics available to us through our commercial metrics package.

First, we used Pearson’s correlation to determine if WMC and WMC-McCabe are closely correlated with OO class defects. Our class sample sizes in this study are large enough to justify the use of parametric statistics. The results are contained in Table IV. WMC-McCabe appears to be a more sensitive metric than WMC and confirms the findings of previous studies that class complexity metrics are a good predictor of OO class quality [1,3,7,29]. The p-values associated with the correlation values indicate the truth of the null hypothesis that the correlation coefficient is “different from zero” at the 10 percent- and 5 percent-levels of significance. P-values less than 0.10 indicate that we cannot reject the hypothesis that there is a linear relationship between the two correlated variables at the 10 percent significance level (see Table IV).

Version	Total Classes	WMC Entropy	N*(WMC Entropy)	Defects	Enhancements	S ₁	S ₂	S ₃	S ₄
14R3	100	1.69378	169.378	21	1	55	12	14	19
15R1	137	1.70589	233.707	29	3	74	21	15	27
15R2	210	1.67034	350.771	10	0	119	28	28	35
15R3	130	1.72864	224.723	41	0	66	17	14	33
15R4	143	1.68644	241.161	153	76	75	17	14	37
15R5	148	1.65728	245.277	61	37	79	17	13	39

Version	Total Classes	WMC Entropy	N*(WMC Entropy)	Defects	Enhancements	S ₁	S ₂	S ₃
14R3	100	0.745704	74.570	21	1	84	13	3
15R1	137	0.683760	93.675	29	3	118	15	4
15R2	210	0.656768	137.921	10	0	182	23	5
15R3	130	0.859166	111.692	41	0	105	19	6
15R4	143	0.860231	123.013	153	76	115	22	6
15R5	148	0.853257	126.282	61	37	120	21	7

Having verified the utility of complexity metrics for predicting initial OO class, we applied equation 1 to Rhino WMC and WMC-McCabe class data. The results are contained in Tables V and VI. A Spearman Rank Correlation on the entropy values versus the number of defects for the corresponding software version was undertaken. The results are contained in Table VII.

	WMC	WMC-McCabe
Spearman Rank Correlation (P-Value)	0.670 (0.145)	-0.111 (0.834)

Surprisingly, neither risk threshold criteria proved statistically conclusive at identifying system-wide decay in Rhino. The p-value associated with the Spearman Rank Correlation test exceeded 0.10 in both cases.

5.0 Analysis

Our research hypotheses are predicated on the notion that once a software system exceeds our complexity thresholds, it will always be error-prone because it will not be fully testable. This implies that there will be a higher proportion of classes that exceed our threshold criteria than in a system that is not degrading. In addition, we could expect to see error-prone (decaying) components reappear during consecutive software releases.

We used our metrics tool, Understand for Java®, to identify classes that changed from one version of Rhino to the next. The results are contained in Table VIII and IX. In each case, the proportion of classes

with WMC-McCabe values greater than 50 was larger in the subset of classes affected by revisions to the software (Table VIII). The subset of classes unaffected by defects was also examined using the same

Version Differences	Methods with Complexity (X)				Entropy
	$0 \leq X \leq 10$	$11 \leq X \leq 20$	$21 \leq X \leq 50$	$X > 50$	
1.4R3 → 1.5R1	36	7	6	1	1.218285
1.5R1 → 1.5R2	37	8	4	7	1.443047
1.5R2 → 1.5R3	44	8	5	3	1.230565
1.5R3 → 1.5R4	28	3	2	2	1.033259
1.5R4 → 1.5R5	27	1	0	2	0.560825
Mean					1.097196

Version Differences	Methods with Complexity (X)				Entropy
	$0 \leq X \leq 10$	$11 \leq X \leq 20$	$21 \leq X \leq 50$	$X > 50$	
1.4R3 → 1.5R1	16	5	7	21	1.788184
1.5R1 → 1.5R2	20	12	14	22	1.957043
1.5R2 → 1.5R3	15	9	9	30	1.804760
1.5R3 → 1.5R4	27	13	12	33	1.868541
1.5R4 → 1.5R5	38	11	11	22	1.801013
Mean					1.843908

Version Differences	Methods with Complexity (X)			Entropy
	$0 \leq X \leq 20$	$20 \leq X \leq 100$	$X > 100$	
1.4R3 → 1.5R1	46	40	0	0.996486
1.5R1 → 1.5R2	51	5	0	0.434081
1.5R2 → 1.5R3	49	6	5	0.869554
1.5R3 → 1.5R4	32	3	0	0.422001
1.5R4 → 1.5R5	29	1	0	0.210842
Mean				0.586592

Version Differences	Methods with Complexity (X)			Entropy
	$0 \leq X \leq 20$	$20 \leq X \leq 100$	$X > 100$	
1.4R3 → 1.5R1	36	13	0	0.834648
1.5R1 → 1.5R2	53	13	2	0.886205
1.5R2 → 1.5R3	48	18	2	1.011921
1.5R3 → 1.5R4	62	21	2	0.957640
1.5R4 → 1.5R5	68	22	2	0.936008
Mean				0.925284

Class Version Differences Compared	Number of Classes			
	That Do Not Appear Again in This Version from Previous	That Appear for First Time	That Appear in Last Version and Also This Version	Net Total Affected in This Version
1.4R3 → 1.5R1 vs 1.5R1 → 1.5R2	14	33	35	68
1.5R1 → 1.5R2 vs 1.5R2 → 1.5R3	24	19	44	63
1.5R2 → 1.5R3 vs 1.5R3 → 1.5R4	5	27	58	85
1.5R3 → 1.5R4 vs 1.5R4 → 1.5R5	18	25	67	92

criteria and was found to have less classes with WMC-McCabe values greater than 50 (Table IX). This indicates the complexity of classes affected by changes due to corrective maintenance is, in general, greater than those of the unchanged classes, confirming the validity of the research hypothesis (H1).

Repeating the same experiment using the NASA SATC WMC threshold criteria showed a similar trend between classes affected or unaffected by defects. The results are contained in Tables X and XI.

In addition, we were interested to know how many classes were affected in two or more consecutive versions of Rhino. This knowledge would help us characterize the maintainability of the Rhino software. Therefore, we analyzed the classes that were affected in the transition from one release of Rhino to the next. Since the Rhino project is an active and progressing project, new classes are being added due to enhancements, and existing classes are being modified and removed mainly due to corrective maintenance and refactoring.

The affected classes were compared to the classes affected in the previous release with the intention of determining the reoccurrence of affected classes in successive releases. These are the same class subsets that were previously identified in Tables VIII through XI. The results are contained in Table XII. Many classes that were modified in the previous release due to bug fixes or enhancements were also affected/modified in the subsequent version of Rhino.

6.0 Conclusion and Future Work

The benefits of object-oriented programming are the resulting simplicity and understandability of the problem through the use of abstraction. However, even OO software is not immune to the effects of brittleness, or degradation, as its original design is extended from one release to the next.

We revalidated previous study findings that showed a moderate- to high-correlation to between WMC and WMC-McCabe complexity metrics and defects. WMC-McCabe appeared to be the more sensitive measure of complexity. We subsequently developed two models based on Shannon's entropy equation, using the two variations of the WMC metric and widely accepted threshold values for interpreting complexity. Neither metric was statistically conclusive. However the measure based on the Chidamber and Kemerer version of WMC, where a complexity score of '1' assigned to each method in a class

showed the most promise at being a good indicator of system degradation. However, due to the lack of availability of data points for the Rhino project, we cannot assert with confidence that this metric will perform well on all software projects.

To find more conclusive proof about the feasible use of this metric, we identified intra-version components that had undergone change or were added to Rhino. We identified OO classes that were not affected between versions and compared them to those that were modified between two versions of the Rhino software. We noted that the group of classes that did undergo change tended to have higher entropy scores than those that did not. Also, we noted that the NASA/Rosenberg threshold risk criteria provided the best correlation to system degradation, possibly because of the grouping of the classes into three categories versus four for the McCabe industry risk criteria we examined.

We believe that this entropy degradation metric may be useful in evaluating OO software, specifically large Java and C++ systems, because of the applicability of the threshold criteria to both languages. This metric may be of most value in programming environments where legacy code is being reengineered into object-oriented programs. During reengineering efforts, especially in reengineering C programs to C++, there is a temptation to redesign the top layers in C++ and to keep the functions that do the real work intact as methods of the resulting classes. In these cases, *reengineering* only fixes a portion of the problem. This occurs frequently in DoD and industry, to quickly rejuvenate programs and extend their life (and their funding). It may also be helpful in evaluating the progress and quality of Java and C++ systems being developed using agile software processes.

In the future, we plan to continue evaluating the feasibility and benefit of this metric on large OO software systems. In addition, based on our analysis we plan to investigate the use of decision trees and complexity metrics in the evaluation of software systems for degradation.

7.0 Acknowledgements

We want to thank Norris Boyd, module owner of the Mozilla Rhino Project, for providing invaluable support and information about the Rhino software, and Stacy Lukins of the University of Alabama in Huntsville, Huntsville, AL for her invaluable assistance in analyzing the Rhino software and collecting and organizing the Rhino class fault data. We also wish to thank Thomas Struebi of XL Consulting GmbH, Switzerland, for providing evaluation versions of Synchronizer®, a MS Excel spreadsheet comparison tool used to analyze Rhino versions for changes. This work was funded in part by the National Aeronautics and Space Administration under Grants NAG5-12725 and NCC8-200.

8.0 References

- [1] Alshayeb, M., Li, W., “An Empirical Validation Of Object-Oriented Metrics In Two Different Iterative Software Processes”, IEEE Trans. Software Eng., vol. 29, no. 11, November 2003, pp. 1043–1049.
- [2] Bansiya, J., Davis, C. G., Etzkorn, L. H., “An Entropy-Based Complexity Measure for Object-Oriented Designs,” Theory and Practice of Object Systems, Vol. 5(2), 1999.
- [3] Basili, V. R., Briand, L.C., Melo, W.L., “A Validation Of Object-Oriented Design Metrics As Quality Indicators,” IEEE Trans. Software Eng., vol. 22, no. 10, October 1996, pp. 751–761.

- [4] Berlinger, E., "An Information Theory-Based Complexity Measure," Proceedings of the National Computer Conference, 1980.
- [5] Bianchi, A., Caivano, D., Lanubile, F., Visaggio, G., "Evaluating Software Degradation through Entropy," Dipartimento di Informatica - Università di Bari, Italy
- [6] Baldessarrei, M.T., Bianchi, A., Caivano, D., Visaggio, C.A., "Full Reuse Maintenance Process for Reducing Software Degradation," Dipartimento di Informatica - Università di Bari, Italy
- [7] Binkley, A., Schach, S., "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," Proc. 20th Int'l Conf. Software Eng., pp. 452–455, 1998.
- [8] Chapin, N., "An Entropy Metric for Software Maintainability," IEEE, 1989.
- [9] Chen, E., "Program Complexity and Programmer Productivity," IEEE, 1978.
- [10] Chidamber, S. and Kemerer, C. "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476–493, 1994.
- [11] Davis, J.S., and LeBlanc, R.J., "A Study of the Applicability of Complexity Measures," IEEE Transactions on Software Engineering, Vol. 14, No. 9, Sept. 1988.
- [12] El Emam, K., Benlarbi, S. Goel, N. Rai, S.N., "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics", IEEE Trans. Software Eng., vol. 27, no. 7, July 2001, pp. 630–650.
- [13] Etzkorn, L. H., "A Metrics-based Approach to the Automated Identification of Object-Oriented Reusable Software Components," A Dissertation, The University of Alabama in Huntsville, Huntsville, AL, 1997.
- [14] Fasolino, A., Visaggio, G., "Improving Software Comprehension through an Automated Dependency Tracker," Dipartimento di Informatica, University of Bari
Via Orabona, 4, 70126, Bari, Italy
- [15] Gill, G.K., Kemerer, C.F., "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Trans. Software Eng.*, vol. 17, no. 12, Dec 1991, pp. 1284–1288.
- [16] Gottipati, S., "Empirical Validation of the Usefulness of Information Theory-Based Software Metrics," Master's Thesis, Department of Computer Science and Engineering, Mississippi State University, MS, May 2003.
- [17] Harrison, W., "An Entropy-Based Measure of Software Complexity," IEEE Transactions on Software Engineering," vol. 18, no. 11, Nov. 1992.
- [18] Kafura, D., Canning, J., "Validation of Software Metrics Using Many Metrics and Two Resources," Proceedings - International Conference on Software Engineering, 1985, p 378–385.
- [19] Lorenz, M., Kidd, J., "Object-Oriented Software Metrics," Pearson Education POD, July 1994.
- [20] McCabe, T.J., "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, Dec 1976, pp. 308–320.

- [21] Michura, J., Capretz, M.A.M., "Metrics Suite for Class Complexity," International Conference on Information Technology: Coding and Computing, pp. 404–409, Vol. 2, Apr 2005.
- [22] Mohagheghi, P., Conradi, R., Killi, O., Scheaz, H., "An Empirical Study of Software Reuse vs. Defect-density Stability," Proceedings - International Conference on Software Engineering, v 26, Proceedings - 26th International Conference on Software Engineering, ICSE 2004, 2004, p 282–291.
- [23] Mohanty, S., "Entropy Metrics for Software Design Evaluation," The Journal of Systems and Software, No. 2, pp. 39–46, 1981.
- [24] Pressman R., Software Engineering: A Practitioner's Approach, McGraw-Hill, 1994.
- [25] Roca, J. L., "An Entropy-Based Method for Computing Software Structural Complexity," Elsevier Science, Microelectronics Reliability, vol. 36, issue 5, May 1996.
- [26] Rosenberg, L. H., "Applying and Interpreting Object Oriented Metrics," April 1998.
- [27] Shao, J., Wang, Y., "A New Measure of Software Complexity Based on Cognitive Weights", *IEEE CCECE 2003*, vol. 2, May 2003, pp. 1333–1338.
- [28] Subramanyam, R., Krishnan, M.S., "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *Trans. Software Eng.*, vol. 29, no. 4, Apr 2003, pp. 297–309.
- [29] Tang, M-H., Kao, M-H., Chen MH., "An Empirical Study on Object Oriented Metrics," Proc. Sixth Int'l Software Metrics Symp., pp. 242–249, 1999.
- [30] Zweben, S., Halstead, M., "The Frequency distribution of Operators in PL/I Programs," IEEE Transactions on Software Engineering, Vol. 5, pp. 91–95.
- [31] <http://www.mozilla.org/rhino/history.html>