I.J. Intelligent Systems and Applications, **2016, 1, 60-66**
Published Online January 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2016.01.07

A Model for Object-Oriented Software Maintainability Measurement

Morteza Asadi
Faculty of Computer and Information Technology Engineering, Qazvin Branch, Islamic Azad, University, Qazvin, Iran
Email: info@asadiweb.ir

Hassan Rashidi *
Department of Mathematics and Computer Science, Allameh Tabataba'i University,Tehran,Iran
Email: hrashi@atu.ac.ir, hrashi@gmail.com

Abstract—Software maintenance is one of the main quality characteristics of the software product. The maintainability of a system is a measure of the ability of the system to undergo maintenance or to return to normal operation after a failure. In this paper, a new model to improve the maintainability of object-oriented software has been proposed. The proposed model is based on newer versions of software quality standard and it is according to the measurement of several new metric. This model has been evaluated on famous PHP framework and the results showed that the proposed model is effective compared with the previous models.

Index Terms—Software maintenance, maintainability, object-oriented software, PHP framework, PHP.

I. Introduction

Traditionally, software maintenance includes all changes of a software system after delivery. Software maintenance is one of the main quality characteristics of the software product. According to IEEE 1219 Standard, software maintenance involves some modifications to a software product such as correcting faults, improving performance or other attributes, adapting the product to a modified environment after its delivery to customers [1]. The maintenance phase of the Software Development Life Cycle is the most costly of all the phases in terms of budget and programmer's effort, these costs can be included 67% of the total cost of software life cycle [2]. Software maintenance activities are categorized into four classes: Corrective maintenance, Adaptive maintenance, Perfective maintenance and Preventive maintenance [1]. Corrective maintenance is reactive modification of a software product performed after delivery to correct discovered faults. Adaptive maintenance is reactive modification of a software product performed after delivery to make a computer program usable in a changed environment. Perfective maintenance is Modification of a software product after delivery to improve performance or maintainability. Preventive maintenance is maintenance performed for the purpose of preventing problems before they occur.

The maintainability of a system is a measure of the ability of the system to undergo maintenance or to return to normal operation after a failure, in other words, Maintainability is the degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications [3]. According to old versions of ISO/IEC, sub-characteristics of software maintainability were analyzability, changeability, stability, and testability; but in new version of ISO/IEC, modularity and reusability are added to sub-characteristics [3].

Analyzability is the degree to which the software product can be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. Changeability is the degree to which the software product enables a specified modification to be implemented or the ease with which a software product can be modified. Stability is the degree to which the software product can avoid unexpected effects from modifications of the software. Testability is the degree to which the software product enables modified software to be validated. Modularity is the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. Reusability is the degree to which an asset can be used in more than one software system, or in building other assets.

Maintenance problems are not low, from the external perspective, the cost of maintenance is too high, the speed of maintenance service is too slow, and difficulty in managing the priority of change requests [4]. From the internal perspective, the work environment forces maintainers to work on poorly designed and coded software. Also, software maintainers encounter three categories of problems: perceived organization alignment problems, process problems, and technical problems. To further exacerbate these problems, much less research has been performed for software maintenance than for development. There are also fewer books and research papers on the subject, and many that are commonly cited may be twenty or more years old. Moreover, a large number of the more recent software engineering books only refer to software maintenance marginally, as they focus on a developers' point of view.

In this paper, a new model to improve the maintainability of object-oriented software has been proposed. In the next section related works for Object-Oriented software maintainability measurement are studied. In section 3, our proposed model is presented and in section 4 this model is evaluated. Finally the conclusion of this article is in section 5, and then section 6 includes paper's references.

II. Related Works

Antonellis et al. in [5] presented ongoing work on using data mining to evaluate a software system's maintainability according to the ISO/IEC-9126 quality standard. More specifically their work proposes a methodology for knowledge acquisition by integrating data from source code with the expertise of a software system's evaluators a process for the extraction of elements from source code and Analytical Hierarchical Processing for assigning weights to these data are provided; K-Means clustering is then applied on these data, in order to produce system overviews and deductions. Their methodology is evaluated on Apache Geronimo (a large Open Source Application Server). The resulted clusters proved to be representative of the code artifacts, helping the domain expert to identify relations between specific metrics and global maintainability as well as spot individual outlier classes that may need reconsideration.

Lincke et al. in [6] Tried to answer this question: Do the differences between general software quality prediction models matter? The goal of their study is to answer this question for a selection of quality models that have previously been published in empirical studies. They compare these quality models statistically by applying them to the same set of software systems. Finally, they calculate a quality trend and compare these conclusions statistically and they identify significant differences among the quality models. Hence, the selection of the quality model has influence on the quality assessment of software based on software metrics.

The main concern of Losavio et al. in [7] is measuring the quality of the architectural design. The goal of their work is to use the architectural design process proposed in the unified process framework, adapting and detailing it to include the quality requirements specification at architectural level. There is general agreement on the fact that in modern applications the selection of the architecture must be addressed early in the development process, to mitigate risks. Moreover, the integration of enterprise applications is a component-based development requiring quality values associated to the services offered by the components. The services depend mostly on the architecture. In consequence, methods arise for guiding the selection or for constructing software architectures. Their approach allows associating the quality requirements (nonfunctional properties) for the architecture expressed using the ISO 9126-1 standard quality model, with the use cases, to facilitate the selection of the key use cases. Measures for the architecture's quality characteristics are specified in details, précising attributes, units, numerical systems and scale types. A case study of a real-time application for monitoring stock exchanges illustrates their approach.

Heitlager et al. in [8] express that the amount of effort needed to maintain a software system is related to the technical quality of the source code of that system. Also, they express the ISO 9126 model for software product quality does not provide a consensual set of measures for estimating maintainability on the basis of a system's source code. On the other hand, the Maintainability Index has been proposed to calculate a single number that expresses the maintainability of a system. They discuss several problems with the MI, and they identify a number of requirements to be fulfilled by a maintainability model to be usable in practice. They sketch anew maintainability model that alleviates most of these problems, and they discuss their experiences with using such as system for IT management consultancy activities.

Chen & Huang in their study in [9] focused on those software development problem factors which may possibly affect software maintainability. They classified twenty-five problem factors into five dimensions; a questionnaire was designed and 137 software projects were surveyed. A K-means cluster analysis was performed to classify the projects into three groups of low, medium and high maintainability projects. For projects which had a higher level of severity of problem factors, the influence on software maintainability becomes more obvious. The influence of software process improvement (SPI) on project problems and the associated software maintainability was also examined in this study. Results of Their paper suggest that SPI can help reduce the level of severity of the documentation quality and process management problems, and is only likely to enhance software maintainability to a medium level. Finally, they identified the top 10 list of higher-severity software development problem factors, and implications were discussed.

Kanellopoulos et al. in [10] proposed a methodology for source code quality and static behavior evaluation of a software system, based on the standard ISO/IEC-9126. Their methodology uses elements automatically derived from source code enhanced with expert knowledge in the form of quality characteristic rankings, allowing software engineers to assign weights to source code attributes. Also, it is flexible in terms of the set of metrics and source code attributes employed, even in terms of the ISO/IEC-9126 characteristics to be assessed. They applied the methodology to two case studies, involving five open source and one proprietary system. Results demonstrated that the methodology can capture software quality trends and express expert perceptions concerning system quality in a quantitative and systematic manner.

Ping in [11] expressed that software maintainability is one important aspect in the evaluation of software evolution of a software product. Due to the complexity of tracking maintenance behaviors, it is difficult to accurately predict the cost and risk of maintenance after delivery of software products. In an attempt to address

this issue quantitatively, he viewed software maintainability as an inevitable evolution process driven by maintenance behaviors, given a health index at the time when a software product are delivered. He used a Hidden Markov Model (HMM) to simulate the maintenance behaviors shown as their possible occurrence probabilities. And software metrics is the measurement of the quality of a software product and its measurement results of a product being delivered are combined to form the health index of the product. The health index works as a weight on the process of maintenance behavior over time. When the occurrence probabilities of maintenance behaviors reach certain number which is reckoned as the indication of the deterioration status of a software product, the product can be regarded as being obsolete. Longer the time, better the maintainability would be

Orenyi et al. reviewed the proposed models and approaches for Object-Oriented Software Maintainability Measurement in the past Decade [12]. Their review shows Software Quality Models such as ISO/IEC is scarcely used in the development of maintainability models. Majority of reviewed models used the existing object-oriented metrics without a critical review and adaptation of these metrics before they are used to develop the models. This makes the developed models to inherit the inconsistencies and ambiguities observed in the object-oriented metrics [12].

The object-oriented metrics used in the models/methods are mostly measured objectively, but the methods used for aggregating metrics or predicting maintainability from the metrics have some element of subjectivity. Though the models developed using the different types of regression analysis have their base/derived metric measured objectively, they are subjective to the peculiar characteristics of the empirical/historical data used to develop the regression equations.

Also, different software products have different structural properties and complexities; thus using a model developed from a set of software "A" to evaluated another software "B"(that is not used in the development of the model) will yield result that will be partly determined by the properties of "A". Hence, the value of "B" will be subjective to "A" [12]. This form of subjectivity has not been identified in Software metrics. This could be one of the fundamental reasons why several forms of regression models yield different results when applied to the same software. Also, this is a potential threat to the wide applicability and acceptability of the various regression models. Thus, there is the need to develop maintainability measurement model that will use objective measurement method to yield consistent result anytime anywhere and by anybody (in this case software developer).

III. Introducing the Proposed Method

A software quality model includes the measurement of the properties of sub-characteristics of a software product. Each sub-characteristic can be measured properly by many methods of metrics and each method of metrics can be applied to more than one sub-characteristic. Ping in [11] provided some metrics for measuring sub-characteristics maintainability for a software product and presented a constant for evolution process of a software product by calculating the summation of ratio of metrics. But his proposed model has some defects, such that All new sub-characteristics of maintainability is Not included in his model and number of implement metrics in his model are too low; Therefore, in this paper, we try to review, complete and simulate this model and suggest an improved model for object-oriented software maintainability measurement.

Table 1 contains the appropriate metrics to measure each of the sub-characteristics of software maintainability that we use for our suggested model.

According to Table 1 for compare Implementation metrics, we need to threshold values for each metric to ratio of each metric with respect to the threshold value is obtained. The threshold value for some metrics in object-oriented programming has been proposed, such as the threshold value for Cyclomatic Complexity that recommended by McCab [13], the threshold value for DIT that recommended by Cais and P ćha in [14], the threshold value for WMC that recommended by Chandra and Linda [15] and the threshold value for Ca that recommended by Ferreira et al. in [16]. But in most papers, the threshold values achieved by analysis software that written in Java or C++ and they not according to programming language such as PHP that support object-oriented programming. So, with respect to the most widely used and most popular open source PHP frameworks have been compared in this paper, it is better that compare relative done between them; Because software maintenance is one of quality characteristics of software produce and calculating maintainability by using relative methods instead of quantitative methods, seems more appropriate. For calculate ratio of each metric the below equation is used:

$$Ratio\ of\ Metric_i\ of\ framework_j = \frac{Metric_i}{\sum_{j=1}^{n} Metric_i\ of\ framework_j} \quad (1)$$

In equation (1), i represent the number of measured metrics for each framework and n is the total number of frameworks that have been reviewed in this paper. If a software product has better analyzability, changeability, stability, testability, modularity and reusability, it certainly will cost less for its maintenance after its

delivery. These sub-characteristics can compose a perfect weight on the effect of maintenance behaviors. Therefore, the method is to forge the measurements of sub-characteristics into a constant C as a weight on the evolution process of a software product. The constant represents the health status of a software product when delivered. The smaller C represents a better health. This constant value was calculated from below equation:

$$C = Ratio\ of\ BKLOC + Ratio\ of\ CC + Ratio\ of\ NOM + Ratio\ of\ WMC + Ratio\ of\ DIT + Ratio\ of\ Ca + Ratio\ of\ CBO \quad (2)$$

Table 1. Implementation metrics for measuring sub-characteristics of Maintainability.

Maintainability sub-characteristics	Metrics Implemented	Result Analysis
Analyzability	1. Line of Code (LOC) 2. Cyclomatic Complexity (CC) 3. Number of Method (NOM) 4. Weighted Methods per Class (WMC)	1. LOC directly has impact on the time and effort required to diagnose errors or faults, and the modules related to them and needed to be modified. 2. Analyzability declines when Cyclomatic Complexity increases, which means the higher complexity of the control flow. 3. Increasing the number of methods in a class indicates that the class does not have a high cohesion, it may indicate the need for further object-oriented decomposition and it causes reducing analyzability. 4. If number of weighted methods increases, classes or modules becomes more complex and and it causes reducing analyzability.
Changeability	1. Line of Code (LOC) 2. Cyclomatic Complexity (CC) 3. Depth of Inheritance Tree (DIT)	1. Changing requires understanding of an entire software entity. The difficulty increases naturally when LOC increases. 2. Cyclomatic Complexity computes the number of the linearly independent paths and each modification must be correct for all execution paths. Therefore, changeability declines when Cyclomatic Complexity increases. 3. Increasing depth of inheritance tree cases compromise encapsulation and increase complexity and it cases reducing changeability.
Stability	1. Coupling Between Object (CBO)	1. Modules with a high coupling can affect the stability. So stability decreases when the coupling between objects increases.
Testability	1. Line of Code (LOC) 2. Cyclomatic Complexity (CC)	1. Complete testing requires coverage of all possible codes. The difficulty increases when LOC increases. 2. Complete testing requires coverage of all execution paths. So testability declines when Cyclomatic Complexity increases.
Modularity	1. Coupling Between Object (CBO) 2. Depth of Inheritance Tree (DIT)	1. High coupling cases more dependencies between modules, So modularity decreases when coupling between objects increases. 2. Increasing depth of inheritance tree causes increasing dependencies and complexity and reducing modularity.
Reusability	1. Coupling Between Object (CBO) 2. Afferent Coupling (Ca) 3.Weighted Methods per Class (WMC)	1. High coupling cases more dependencies between modules, So reusability decreases. 2. Afferent coupling For a module increases By increasing the number of modules that's associated to it and it cases reducing reusability. 3. If number of weighted methods increases, classes or modules becomes more complex and and it causes reducing reusability.

IV. Definition of Hidden Markov Model

An atomic event is an assignment to every random variable in the domain. For example, "it is raining today" and "it is not raining today" are two atomic events. We can use a binary variable *raining* to describe these two events. If it is raining, we assign *raining* to 1. Apparently for *n* random variables, there are 2^n possible atomic events.

States are atomic events that can transfer from one to another [17]. Suppose a model has *n* states $\{s_1, s_2, \ldots, s_n\}$, we can describe how a system behaves with a state-transition diagram.

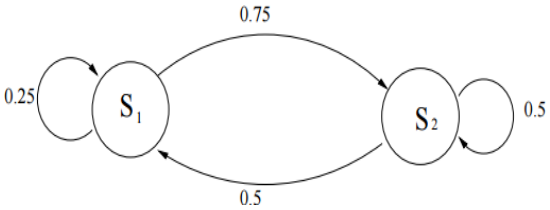

Fig.1. State-transition diagram [17]

In Fig. 1, $P(S_i|S_j), (1 \le i, j \le n)$ are called transition probabilities. Transitions among the states are governed by these transition probabilities. If we consider that time moves in uniform, discrete increments, $P(S_i|S_j)$ represent the probability that in time *t+1*, the system is in state S_i, given that in time *t*, the system is in state S_j. For example in the Fig. 1, in a time interval *t* if the system is in stat S_1, then in time *t+1*, there is a ¾ probability that the system is in state S_2, and a ¼ probability that the system is still in state S_1. Notice that transition probabilities should also satisfy the normal stochastic constraints.

$$0 \le P(S_i|S_j) \le 1 \quad , (1 \le i, j \le n) \quad (3)$$

And

$$\sum_{i=1}^{n} P(S_i|S_j) = 1, \ (1 \le j \le n) \quad (4)$$

In the state-transition diagram (Fig. 1), there are three assumptions: First, Transition probabilities are stationary and they do not change over times (the stationary assumption). Second, the event space does not change

over time and we will not get a new state as time goes on. Third, probability distribution over next states depends only on the current state (Markov assumption) [17].

Table 2. Results of calculated ratio of each metric and C constant for each framework

	codeigniter	cakephp	Yii framework	Symfony	laravel	zend
CBO	0.510	1.867	2.538	3.182	2.270	3.247
NOM	10.812	3.521	3.596	5.829	7.296	5.931
DIT	1.299	1.224	2.148	1.298	1.098	1.205
WMC	40.352	10.294	12.396	11.021	10.621	17.336
Ca	0.119	0.422	0.772	1.645	0.886	1.958
CC	3.393	2.615	2.664	2.037	1.345	2.521
KLOC	64.104	14.316	18.013	367.789	42.456	293.617
BKLOC	8.065	28.150	42.358	3.276	8.573	2.448
C	**1.222**	**1.036**	**1.436**	**1.114**	**0.938**	**1.251**

Actually, Markov assumption is a special kind of conditional independence. It shows that given the current state, future state is independent of all past states. It seems that this assumption is very limited, but actually most cases of the real world can satisfy this assumption given our states are well defined. Target tracking, patient monitoring and speech recognition are all this kind of applications.

A system with states that obey the Markov assumption is called a Markov Model. A sequence of states resulting from such a model is called a Markov Chain. Markov model has a very nice property that its description can be maintained within quadratic space (as to the number of states in the model). Potentially we can get an infinite time sequence.

In the Markov Model we introduce E_t as the outcome or observation at time *t*. Observations are generated according to the associated probability distribution. Given the current state *S*, the probability we have the observation *E* is defined as emission probability *P(E|S)*. Here we also make the stationary assumption that emission probabilities do not change over time. Besides, very similar to the Markov assumption, we assume that the current observation is only depended upon the current state. Or in another word, observations are conditionally independent of other variables given the current state. Mathematically this assumption is represented as

$$P(E_t|S_t, S_{t-1}, \dots, S_0) = P(E_t|S_t) \quad (5)$$

V. Evaluation Proposed Model

In the first part of this section, our proposed model is evaluated on some PHP frameworks and by equation (2) constant value for each framework is calculated. In the next part, the Hidden Markov Model is used to show the probability of maintenance behaviors.

5.1. Metrics Measurement

The metrics that were reviewed in the previous section are commonly used for object-oriented systems and we choose PHP frameworks for evaluation our proposed model, because these PHP frameworks are object-oriented and PHP language always has received less attention than Java, C++, etc. in evaluation of object-oriented programming language; so we choose PHP frameworks to evaluation our proposed model. We select the most widely used and popular PHP frameworks on GitHub [1] website that a repository for open source projects in different programming languages. By searching for PHP frameworks with most participants and high popularity, finally we find the six well-known PHP which include: Zend framework, Symfony, Yii framework, cakephp, codeigniter and laravel.

After downloading source code of frameworks from GitHub and import them in Eclipse IDE, we used PHP Depend[2] as a plugin for Eclipse for metric measurement. PHP Depend after analysis source code of each framework, returns measured metrics as an xml file, we have write a program in c # to process these xml files. For calculating the number of Bugs in Kilo Lines of Code (BKLOC), first we wanted to use change log file of each framework, but in some frameworks (such as cakephp) this file is not released; so we decided to use issue's page for each project on GitHub website and we consider all issues with bug's label as the number of bugs for each framework. Also, since the equation (2) is in general form and it more suitable for one class or method, so for use it on PHP frameworks that contain many package, class and method, first, we calculate average of each metric per module and then calculate average for total of each framework. Table 2 shows result of calculate ratio of each metric for each PHP frameworks.

5.2. Create a Hidden Markov Model

A Hidden Markov Model is a matrix with cells representing the states of a matter in different timestamps displaying a process of a matter's status evolution. Ping in [11] categories user's request after delivery software product in four type of maintenance in percentage: Corrective maintenance 12.4%, Adaptive maintenance 65.4%, Perfective maintenance 9.5% and Preventive

[1] www.github.com
[2] www.pdepend.org

maintenance 9.3%, also small percentage of requests from users outside of the four type and not considered. To display the status evolution of software maintainability, a Hidden Markov Model is set up and in this matrix, the row items indicate the probabilities of each kind of maintenance behaviors occurring individually and the column items are the probabilities of a software product switching from one kind of maintenance behavior to another. Since four type of maintenance behavior is available, the matrix has four rows and four columns.

For initialization of the matrix, mentioned percentages is set to each column, because each row indicate the probabilities of each kind of maintenance behaviors occurring individually. As required by a HMM, the sum of each row should be 1, so the model is normalized by:

$$a_{ij} = s_i \times s_j / \sum_{j=0}^{4} s_i \times s_j \quad , \qquad 1 \le i \le 4 \qquad (6)$$

And the model becomes as below,

i		$P(p_{t+1}=s_1 \mid p_t= s_i)$	$P(p_{t+1}=s_2 \mid p_t= s_i)$	$P(p_{t+1}=s_3 \mid p_t=s_i)$	$P(p_{t+1}=s_4 \mid p_t= s_i)$
1	s_1	a_{11}=0.015	a_{12}=0.012	a_{13}=0.081	a_{14}=0.012
2	s_2	a_{21}=0.012	a_{22}=0.009	a_{23}=0.061	a_{24}=0.009
3	s_3	a_{31}=0.081	a_{32}=0.061	a_{33}=0.428	a_{34}=0.062
4	s_4	a_{41}=0.012	a_{42}=0.009	a_{43}=0.062	a_{44}=0.009

For calculate the time period for threshold of each maintenance type is reached, Ping in [5] proposed below algorithm. In this algorithm C is the constant value that obtained from the ratio of metric according to equation (1) and using this constant value caused that each software product have own evolution rate. In other words, the constant C as a weight on the evolution process of a software product indicates the quality of software product and it can be applied to influence the process of software maintainability.

Algorithm 1.Calculate the time period for reached threshold

1. $\boldsymbol{p_0(1) = P(q_0 = s_1) = 1 \times C}$

2. $\boldsymbol{p_{t+1}(j) = P(q_{t+1} = s_j) = \left(\sum_{i=0}^{4} a_{ij} \times p_t(i)\right) \times C}$

3. If the threshold is reached, the time t shows the time period. Otherwise, go to step 2.

By applying the algorithm 1 on the PHP frameworks, table 3 shows that results were obtained for each type of maintenance.

According to Table 3, time period to reach the threshold for each type of maintenance behavior in each of the PHP framework is obtained. For example, the time period of cakephp and laravel, for reaching the threshold for all four types of maintenance behavior is 2 and that this value is very appropriate. Also maximum value for time period for all PHP frameworks is 3, which means reaching the all type of maintenance behavior after the third time period, which is a good value.

Table 3. Results of applying the algorithm 1 on the PHP frameworks

Frameworks / Time period	Codeigniter	cakephp	Yii framework	Symfony	laravel	zend
Corrective maintenance	2	2	2	2	2	2
Adaptive maintenance	3	2	3	3	2	3
Perfective maintenance	2	2	3	3	2	2
Preventive maintenance	3	2	3	3	2	3

VI. Conclusion

In this paper, a new model to improve the maintainability of object-oriented software has been proposed and this model has been evaluated on famous PHP frameworks. Our proposed model is more accurate than other existing models due to using of more metrics and for calculating ratio of metric use relative methods instead of quantitative methods. Also, proposed model for criticisms of the regression analysis, use summation for aggregating metrics. To show the possibility of maintenance behaviors Hidden Markov Model was used. The results show the appropriate time period to reach threshold values for maintenance behaviors and that means maintainability of PHP framework is desirable in terms of cost and effort.

References

[1] IEEE P14764, http://ieeexplore.ieee.org/servlet/opac?punumber=4040502

[2] Lee R., Tepfenhart M., 2005, UML and C++: A Practical Guide to Object-Oriented Development, Pearson Prentice Hall, second edition.

[3] ISO/IEC 25010:2011, http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733

[4] April, A., Huffman Hayes, J., Abran, A., Dumke, R.: Software Maintenance Maturity Model (SMmm): the software maintenance process model. Journal of Software Maintenance and Evolution: Research and Practice, 2005, 17, 13, 197–223.

[5] Antonellis P., Antoniou D., Kanellopoulos Y., Makris C., Theodoridis E., Tjortjis C. and Tsirakis N., A data mining methodology for evaluating maintainability according to ISO/IEC-9126 software engineering–product quality standard, Special Session on System Quality and Maintainability-SQM2007, 2007.

[6] Lincke R., Gutzmann T. and W. Lo we, Software quality prediction models compared, Quality Software (QSIC), 2010, 10th International Conference, pp. 82-91.

[7] Losavio F., Chirinos L., Matteo A., Lévy N. and Ramdane-Cherif A., ISO quality standards for measuring architectures, J. Syst. Software, 2004, vol. 72, pp. 209-223.

[8] Heitlager I., Kuipers T. and Visser J., A practical model for measuring maintainability, Quality of Information and Communications Technology, 2007, 6 th International Conference, pp. 30-39.

[9] Chen J. C. and Huang S. J., An empirical analysis of the impact of software development problem factors on

software maintainability, J. Syst. Software, 2009, vol. 82, pp. 981-992.
[10] Kanellopoulos Y., Antonellis P., Antoniou D., Makris C., Theodoridis E., Tjortjis C. and Tsirakis N., Code Quality Evaluation Methodology Using The ISO/IEC 9126 Standard, Arxiv Preprint arXiv:1007.5117, 2010.
[11] Ping L., A quantitative approach to software maintainability prediction, Information Technology and Applications (IFITA), 2010 International Forum, pp.105-108.
[12] Orenyi A., Basri S., Tan Jung L., Object-Oriented Software Maintainability Measurement in the past Decade, International Conference on Advanced Computer Science Applications and Technologies, 2012, 257-262.
[13] Thomas McCabe, "A Complicity Measure", IEEE Transaction on Software Engineering, VOL. SE-2, No. 4, 1976.
[14] Š. Cais & P. Pícha, Identifying Software Metrics Thresholds for Safety Critical System, SDIWC, 2014.
[15] E. Chandra, P. Edith Linda, Class Break Point Determination Using CK Metrics Thresholds, Global Journal of Computer Science and Technology, Vol.10 Issue 14, November 2010.
[16] K. Ferreira, M. Bigonha, R. Bigonha, L. Mendes, H. Almeida, Identifying thresholds for object-oriented software metrics, The Journal of Systems and Software, Elsevier, 2011.
[17] Ron Parr, "Hidden markov models," Duke University, tutorial, October 2004, https://www.cs.duke.edu/courses/fall03/cps260/notes/lecture14.pdf [Accessed on: 2015-08-20].

Authors' Profiles

Morteza Asadi is a student of M.Sc in Computer Software Engineering at Islamic Azad University of Qazvin. He received his BSc degrees in Computer Software Engineering from University of Zanjan. His research interests include software engineering, software maintenance, computer science and object-oriented programming. His website is http://asadiweb.ir.

Hassan Rashidi is an Associate Professor in Department of Mathematics and Computer Science of Allameh Tabataba'i University. He received an M.Sc degree in Systems Engineering and Planning from the Isfahan University of Technology and a PhD from Computer Science and Engineering from University of Essex. His research interests include software engineering, software testing, and Scheduling algorithms.

How to cite this paper: Morteza Asadi, Hassan Rashidi, "A Model for Object-Oriented Software Maintainability Measurement", International Journal of Intelligent Systems and Applications (IJISA), Vol.8, No.1, pp.60-66, 2016. DOI: 10.5815/ijisa.2016.01.07

