International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:1, 2014

Empirical Analysis of the Reusability of Object-Oriented Program Code in Open-Source Software

Fathi Taibi

Abstract**—**Measuring the reusability of Object-Oriented (OO) program code is important to ensure a successful and timely adaptation and integration of the reused code in new software projects. It has become even more relevant with the availability of huge amounts of open-source projects. Reuse saves cost, increases the speed of development and improves software reliability. Measuring this reusability is not s straight forward process due to the variety of metrics and qualities linked to software reuse and the lack of comprehensive empirical studies to support the proposed metrics or models. In this paper, a conceptual model is proposed to measure the reusability of OO program code. A comprehensive set of metrics is used to compute the most significant factors of reusability and an empirical investigation is conducted to measure the reusability of the classes of randomly selected open-source Java projects. Additionally, the impact of using inner and anonymous classes on the reusability of their enclosing classes is assessed. The results obtained are thoroughly analyzed to identify the factors behind lack of reusability in open-source OO program code and the impact of nesting on it.

Keywords**—**Code reuse, Low Complexity, Empirical Analysis, Modularity, Software Metrics, Understandability.

I. Introduction

SOFTWARE reuse as a specific area of study in software engineering was first introduced in the late sixties by Douglas McIlroy [23] who proposed the usage of reusable components in industrial software development. However, since the early days of programming, some forms of improvised code reuse has been practiced. Software reuse can be seen in various forms. Code reuse is one of them and is widely practiced today in the software industry [16]. Reusing software design skeletons and even the design processes are also practiced as well. Software reuse saves cost, increases the speed of development and improves reliability [10], [13].

Structuring program code using modules is a software quality called modularity, which has a positive impact on reusability and maintainability. Code reuse involves reusing modules, which are small and highly-independent programs. In OO software, a module comprises one or several classes whose reusability potential determines the reusability of the module. The concept of package in Java allows organizing classes and interfaces into namespaces similar to the modules in other programming languages. However, the classes belonging to a package do not need to be in a single file. Measuring the reusability of individual classes should enable assessing the reusability of modules or packages in OO software projects.

Studies [20], [27] have shown a strong correlation between complexity and software defects. Methods and functions that have the highest complexity tend to also contain most defects. Hence, low complexity is an important factor to consider for code reuse together with modularity. High complexity does not only hinder reuse but increases the maintenance cost of the code as well. Many other qualities have been associated with code reusability. However, there is a lack of precise metrics that can measure this reusability accurately.

Structuring OO programs is achieved by encapsulating their features in classes and defining relationships between them. In a 'clean' code [22], classes should be primitive and small in size. Because classes are primitive they should not contain a large number of methods. For example, the value 7 has been proposed as an upper bound for the number of methods in a class and 150-200 Lines of Code (LOC) for its size. However, size is not a very significant factor to complexity since very often a small class may be more complex than a larger one. Furthermore, code smells [9] provide a different way for dealing with 'unclean' code such as the presence of too many parameters in a method.

OO classes could be further structured using inner and anonymous classes [25]. These classes have been found to be useful in quite a number of cases such as when developing graphical user interfaces, handling events and encapsulating functionality. However, classes that contain inner or anonymous classes tend to look more complex than those that don't include them. Additionally, anonymous classes could be replaced by a method since they will never be used anywhere else in the software. This raises an important question on the real impact inner and anonymous classes have on the reusability of their inclosing classes.

Code reuse is widely practiced today either by people in academia (especially students) or in software industry. This was made possible by the availability of huge amounts of open-source projects over the Internet. A considerable amount of this code is OO. The overall quality of most of these projects is unknown. Reusing them blindly can cause my problems. Students can accumulate bad design and coding habits as a result of that. Worst, student cannot attain a certain level of problem solving and critical thinking skills as a result of excessive reuse of poor quality code. Professional developers on the other hand may waste valuable time trying

Dr. Taibi is with UNITAR International University, Petaling Jaya, 47301 Selangor, Malaysia (phone: +603-76277200; fax: +603-76277447; e-mail: taibi@unitar.my).

to reuse code with poor reusability to reach their standard of quality. Hence it would be very useful if there is a way to measure the reusability of the classes contained in this program code accurately in order to discard open-source projects with poor reusability from being reused, which should benefit various people who rely on code reuse.

A conceptual model to measure the reusability of OO classes is proposed in this paper. A set of well-established metrics is used to compute the most significant factors of reusability and an empirical investigation is conducted to measure the reusability of the classes of randomly selected open-source Java projects. Additionally, the impact of using inner and anonymous classes on the reusability of their enclosing classes is assessed. The results obtained are thoroughly analyzed to identify the factors behind lack of reusability in open-source OO program code and the impact of nesting on it.

II. Related work

Code reusability depends on several factors. Complexity is one of these factors and has a negative impact on it (i.e. low complexity is desired). Modularity, on the other hand, has a positive impact on it. In fact, the latter is considered almost a pre-requisite for reusability. Many other factors have been found to enhance the quality of source code and potentially improve its reusability. One of these factors is understandability or readability [4]. It is a crucial factor for code reuse since it is associated with the way the code is written. Using naming conventions [18], writing useful code comments and making sure the layout of code makes it readable are example of techniques that can enhance understandability and the overall software quality. Using domain names in code can help understandability. It was highlighted in [14] that in almost half of several studied open-source systems, domain names were used in the source code. Moreover, the usage of naming conventions has been found to be reliable if the names used are related to the concepts implemented [2]. Furthermore, in [7] an approach was proposed to help developers in maintaining consistency between source code identifiers, comments and high-level artifacts. The approach included an identifier suggestion feature, among others.

Reusability of code is also associated with classes that possess two important qualities namely high cohesion [1] and low coupling [6]. High cohesion means the elements of a class have a strong correlation. Low coupling means limiting the number of dependencies between classes as much as possible. Lack of Cohesion of Methods (LCOM) and Coupling between Objects (CBO) [5] are examples of reliable metrics that can measure cohesion of classes and coupling between them. Even Cyclomatic Complexity (CC) [21] has a proven relationship with cohesion and coupling. This was shown in many studies such as in [26]. Moreover, excessive coupling between classes was found to be a very reliable predictor of faults in OO systems as indicated in [12] and [29]. It was found in [12] that CBO is more reliable than LCOM and several other OO design metrics in predicting faults. In [8], a strong correlation between CBO, the Depth of Inheritance Tree (DIT) and fault-proneness was established. Furthermore, LCOM was found to have some limitations when most of the methods of a given class access more the fields and methods of its super class(es) than the fields defined locally. This was verified empirically in [19].

Measuring code reusability is not a straight forward process since it depends on several factors. Most of them are hard to compute and are not representative when considered alone. Moreover, some free tools are available (e.g. Sonar [11], C and C++ Code Counter (CCCC) [30]); they allow measuring a wide range of software metrics, some of which may be related to reusability. However, most of these tools simply display the results without any interpretation of their significance, especially in case of reusability. Hence, they fail to answer the basic question in this situation, which is what is the exact reusability potential of a class or a module.

Assessing code reusability could enable some indirect discoveries to be made. The detection of software defects is one of them. For example, during development while assessing code quality and performing refactoring to make it reusable, several types of defects can be detected. Defects are very costly to detect and correct after software is released. Detecting them at an early stage can lead to huge amounts of savings. Some estimate these savings to be in the order of tens of billions of dollars in the United States alone [3], [15].

All OO programming languages allow inheritance while some of them allow inner classes. Java is an example of a language that supports both features. Inner classes could be named, for example a class ‘B’ could be defined inside a class ‘A’. The results of compiling the class ‘A’ is two classes namely A.class and A$B.class. Moreover, inner classes could be anonymous. The result of compiling them produces classes whose names are derived from the outer class with an additional numerical sequence. For example, A$1.class and A$2.class are produced when compiling the class ‘A’ if it contains two anonymous class blocks. Inner and anonymous classes have been found to be useful in quite a number of cases such as when developing graphical user interfaces, handling events and encapsulating functionality. However, classes that contain inner or anonymous classes tend to look more complex than those that don’t include them. Moreover, they could create ambiguity over direct super-classes since there could be several classes with the same name in the program, which is made possible through nesting classes. This was investigated in [17] and a non-deterministic algorithm was proposed to address this problem. All these problems associated with inner and anonymous classes make their associated code less readable and probably more complex. Hence, it could affect their reusability. Investigating the impact of inner and anonymous classes on the reusability potential of their outer classes is worthy. However, this problem has somehow not been given much attention in the literature.

In [28], a small-scale empirical investigation was conducted to measure the reusability of open-source program code. The projects considered were small in size, they incorporated a

small number of modules and their size in terms of LOC was small as well. The results obtained showed that the studied modules have an acceptable overall reusability. However, they have a relatively poor modularity and understandability. Excessive coupling and low cohesion was observed in more than third of the modules. Additionally, the ratio and quality of code comments was relatively poor and has poor correlation with program entities and domain names. Only a small number of metrics were used to measure the degree to which the factors used to assess the reusability of a module are satisfied. Using more metrics should enable a better precision in the calculation. However, this could make reusability assessment less effective. Hence, it is crucial to identify and use only the metrics that have the highest impact on the factors that affect code reusability directly or any other quality that helps indirectly achieving it.

III. The Proposed Model

In order to come out with a model that is as accurate as possible, several code metrics representing the factors that have some direct or indirect relationship with code reusability were investigated. A new metric was proposed to measure the reusability of a class precisely and effectively. It consists of a weighted balance among three of the most important factors that have been proven to have a significant impact on the reusability of classes. Several attributes are used to assess each factor since one attribute in isolation cannot guarantee that a factor is achieved. For example, writing useful code comments is one of the most significant techniques that can enhance understandability. However, having a high percentage of comments alone cannot guarantee understandability unless they are related to code statements. The calculation uses both textual and structural information extracted from source code. The proposed reusability metric (R) is a function of three factors:

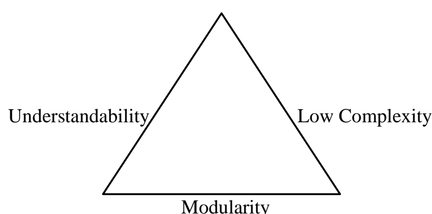

Fig. 1 Elements of the proposed model

- Modularity (M): the degree of modularity is a value between 0 and 1 that is assessed through measuring the cohesion and coupling of the classes of a project. LCOM and CBO are used to perform the assessment of cohesion and coupling respectively. Since CBO is a good predictor to fault proneness, the modularity metric contributes in discarding fault-prone classes from being reused. Similarly, it eliminates classes with poor maintainability since poor cohesion lowers maintainability.
- Understandability (U): the degree of understandability of a class is a value between 0 and 1 that is assessed through the signification or relevance of names used for its classes, fields and methods (Relevance Of Identifiers - ROI), the rate of code comments and their correlation with the names used (Correlation Identifiers Comments - CIC).
- Low Complexity (LC): the average CC of the methods of a class is used as an indicator of complexity; the value 10 is used as threshold. CC alone is very significant. However, it was recommended in [22] that a class should have no more than 7 methods or else it becomes gradually too complex as the number of methods increases. Hence, the Number of Methods (NM) per class is considered as well. Furthermore, DIT is also used in the assessment with the threshold 5 being the upper bound for an acceptable complexity. The class size in terms of LOC was not included in the calculation of LC because it is not very significant to complexity. The metric LC is also a value between 0 and 1.

Hence, the reusability of a class is calculated as follows:

$$R = \sum_{i=1}^{n} \lambda_i \times F_i \qquad (1)$$

where F_i are the factors used to assess reusability, λ_i represent tuning parameters and $\Sigma\lambda_i = 1$.

For a given class, M is calculated based on its LCOM and CBO. A highly cohesive class should have an LCOM equal to zero. Even though there is no best value for CBO, many tools such as Sonar assume that an acceptable value is less than or equal to five. This sounds logical since a class in a module must have at least some relationships with the other classes, or else it should be moved out of the module. Moreover, since many studies have shown that CBO is a reliable metric for fault-proneness (more reliable than LCOM) as indicated earlier, in the proposed formula to compute M, CBO is given more weight than LCOM. Furthermore, the calculation of U is based on two metrics (ROI and CIC) using the information extracted from the source code (i.e. names and comments). ROI is calculated as the ratio of the number of relevant names used in a class by their total number. Relevance in this case means that a name is meaningful and is related to domain information or the requirements. CIC is calculated using a similarity metric based on N-Grams [24] (N=2 is used in this study) since the latter is good in situations where there may be a change of word order. This happens often when comparing names of program elements and comments. U is calculated as the average of ROI and CIC. Finally, LC is calculated based on the NM in a class, its DIT and the average CC of its methods. The average CC and DIT are given more weight because of their significance to complexity and fault proneness. LC is calculated as weighted average of NM, DIT and CC. The calculation of the factors M and LC is made using the following formula:

$$F = \frac{\sum_{j=1}^{m} \alpha_j \times W_j}{\sum_{j=1}^{m} \alpha_j} \qquad (2)$$

W_j ($j \in [1, m]$) are the weights of the metrics used to calculate

the factor F and α_j ($j\in[1, m]$) are the tuning parameters. The weights of the metrics used to calculate the factor M and LC are shown below.

TABLE I
WEIGHTS OF THE METRICS USED TO CALCULATE THE FACTORS M AND LC

W_j	1	0.75	0.5	0.25	0
Condition	LCOM=0	0<LCOM<3	3≤LCOM<5	5≤LCOM≤10	LCOM>10
	CBO≤5	5<CBO≤7	7<CBO≤9	9<CBO≤10	CBO>10
	CC≤10	10<CC≤20	20<CC≤35	35<CC≤50	CC>50
	NM≤7	7<NM≤10	10<NM≤13	13<NM≤16	NM>16
	DIT≤5	5<DIT≤7	7<DIT≤9	9<DIT≤10	DIT>10

An extensive set of experiments were conducted and a heuristic method was used to finalize the values of α_j as well as to find the best balance among the tuning parameters (i.e. λ_i). The weight of U was found to be slightly less significant than the weights of LC and M. A large number of the classes used in the experiments have high scores in LC and M but are not properly commented and/or the names used for their elements are not ideal. The best results (i.e. the most representative of reusability) were obtained for α_j=1.5 for W_{CC}, W_{DIT} and W_{CBO}, α_j=1 for W_{NM} and W_{LCOM} (i.e. CC, DIT and CBO were given more weight than NM and LCOM respectively), λ_i=0.3 is used for U and λ_i=0.35 is used for LC and M.

IV. EVALUATION

In order to test the proposed model with some OO classes, 22 Java projects were randomly selected from 'sourceforge' website [31]. These projects represent various types such as application, utility, tool, game and animation. They incorporated a total of 497 files comprising 908 classes with a total of 80769 LOC. The following table shows the details of the selected projects.

TABLE II
DETAILS OF THE SELECTED PROJECTS

	Max	Min	Median	Mean	Std
#Modules	140	1	19	22.59	37.87
#Classes	303	6	28	41.27	82.49
#Inner Classes	49	0	1	4.55	14.4
#Anonymous Classes	72	0	5	12.14	22.05
Size (LOC)	16766	234	2098	3671.32	4917.59
%Comments	12.82%	1.16%	5.63%	5.68%	3.59%

Since the projects were randomly selected, there was a large variation in their size translated into a standard deviation (Std) that is larger than the mean for the size, the number of modules, the number of classes and the number of inner and anonymous classes. One of the projects was considerably larger than the rest; it included 140 modules and 303 classes. Another 4 projects incorporated between 41 and 86 classes while all the other projects had between 6 and 39 classes.

The metrics U, LC and M were computed individually for each class in the selected projects. In order to achieve that, NM, LCOM, CBO and DIT were calculated using Chidamber and Kemerer Java Metrics (CKJM) tool [32] while CC was calculated using CCCC [30]. CIC was calculated using a developed prototype tool that extracts the names of classes, fields and methods from source code and calculates the similarity between these names and code comments. ROI was assessed manually by two different developers and the average value was taken. The results were then thoroughly analyzed. Fig. 2 shows the reusability of each class in the studied projects where the results are sorted for a better analysis.

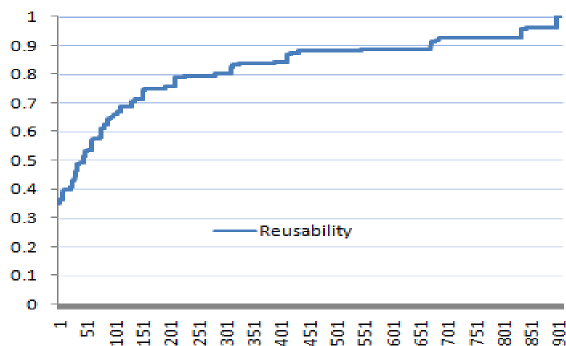

Fig. 2 Reusability of the studied classes

The overall reusability of the classes in each project was acceptable. Only 129 classes had reusability below 0.7 (14.21%). Moreover, out of these classes, only 44 classes had a reusability below 0.5, which represents only 4.84% of all the studied classes. The reusability of all the studied classes was between 0.35 and 1 with an average of 0.82. Furthermore, the classes with reusability problems (i.e. score below 0.7) were further analyzed. They were grouped into two categories; the first one (Cat1) incorporates those with a reusability below 0.7 but greater or equal to 0.5. The second one (Cat2) included those with reusability below 0.5. The aim was to find out the reason (s) why these classes had a lower reusability in comparison with the other classes in their respective projects. It was explained earlier that the impact of inner and anonymous classes on the reusability of their inclosing classes was an important factor to consider since it could make them more complex or less modular. Hence, the ratio of classes that included inner and anonymous classes in Cat1 and Cat2 was analyzed; the results obtained are shown in Fig. 4. Fig. 3 shows the distribution of the classes in Cat1 and Cat2 across the studied projects. The projects with no inner or anonymous classes are not shown for the sake of space.

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:8, No:1, 2014

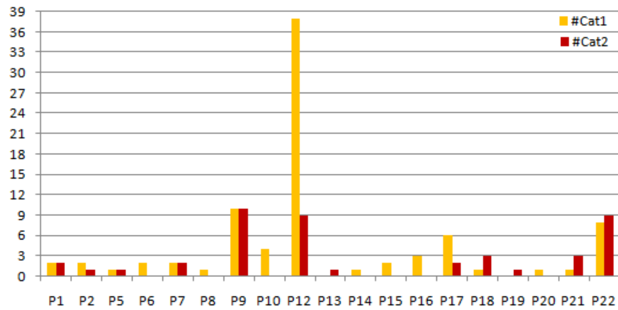

Fig. 3 Distribution of classes with poor reusability

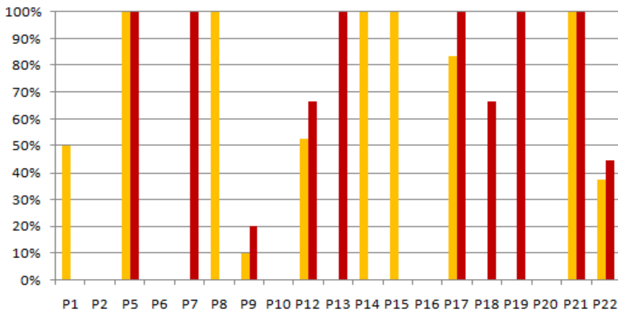

Fig. 4 Percentage of classes containing inner and anonymous classes in Cat1 and Cat2

The results showed that the incorporation of inner and anonymous classes has a negative impact on the reusability of their inclosing classes. In eight projects, all the classes with poor reusability had inner or anonymous classes. Moreover, in another four projects, more than half of the classes with poor reusability had inner or anonymous classes. Out of the 908 classes studied, 129 classes (i.e. Cat1+Cat2) had poor reusability. Out of these classes, 60 classes had an inner or anonymous class, which translates into a rate of 46.51%. This is a significant indication that these types of classes have a negative effect on the reusability of their inclosing classes. This is also shown by the ratio of classes with inner and anonymous classes in Cat2, which was more than half (54.55%).

A further investigation was needed to identify the performance of the classes in respect to the three factors used to calculate the reusability metric (i.e. M, U and LC). Fig. 5 shows the percentage of classes with poor M, U and LC (i.e. metric < 0.7) in the studied projects.

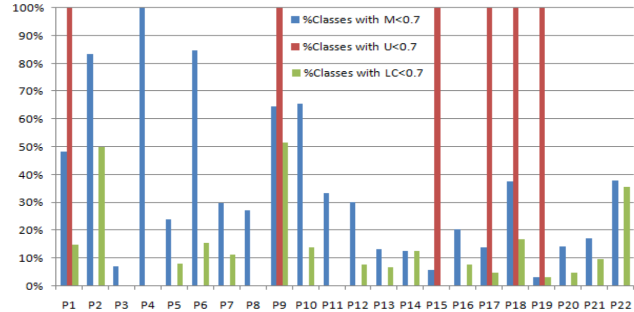

Fig. 5 Percentage of classes with poor M, U and LC in each project

The results obtained showed that poor reusability in the classes of the studied projects was caused by under performance in regards to the metrics M and U. For the latter metric (i.e. U), the underperformance was associated with only six projects whose entire classes had scores below 0.7 (i.e. P1, P9, P15, P17, P18 and P19). This means that all the classes of the remaining projects had scored above 0.7. This leaves M as the only metric where underperformance was quite consistent. The rate of classes with M below 0.7 was between 3.22% and 100% in the studied projects. Each one of these projects has at least one class with an M below 0.7 with a maximum of 91 classes. All these indicators show clearly that the main reason behind poor reusability can be attributed to a consistent poor modularity across all the studied projects.

In order to further investigate this discovery, the performance in terms of modularity of the 44 classes in Cat2 (i.e. those with a reusability below 0.5) was analyzed. The objective was to discover any correlation between a very low reusability and a lack in modularity; a threshold of 0.5 was used for M as well. The results obtained were astonishing. All the 44 classes with poor reusability in the selected projects had modularity below 0.5 (100% correlation). This confirms the findings discussed earlier and provides a strong link between lack of modularity, which is measured through cohesion and coupling in the proposed model, and poor reusability. Hence, lack of modularity is the contributing factor behind poor reusability in most of the classes of the selected projects.

A final analysis was needed to study the correlation between the average reusability of classes and the online rating of their associated projects. The number of votes itself constitutes an important factor because it gives some indication on the popularity of the project. Project P6 is the most popular according to this factor since 353 people rated it with an overall rating of 4.8 out of 5. Project P2 was the least popular with only 2 people rating it with a perfect score (i.e. 5). In order to perform this final analysis, a relative number of votes value was calculated by dividing each number of votes by the maximum number of votes (i.e. the 353 value obtained for P2). Similarly, the overall rating for each project was pro-rated to a value between 0 and 1 by dividing each rating by 5. The results obtained are shown in Fig. 6 where five projects are omitted since they did not receive any votes.

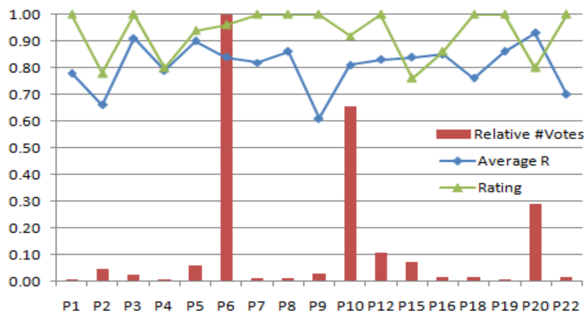

Fig. 6 Correlation between project reusability and rating

The results obtained showed that the rating given to the studied projects exceeded their measured reusability in most cases except in two projects (P15 and P20) where the ratings were slightly below the average reusability. Moreover, for two particular projects (P9 and P22), the difference between the rating given and the measured reusability was quite significant. This could be an indication of overrating. However, since the number of votes was below 100 votes in most projects except in three cases (P6, P10, and P20), no definitive conclusion could be obtained regarding the correlation between the reusability and the popularity of a project.

V. Conclusion and Future Work

An empirical analysis of the reusability of OO program code in open-source software projects was conducted in this paper. Reusability was calculated based on a new proposed metric that incorporates three factors that have a significant effect on reusability. These factors are modularity, understandability and low complexity. A set of well-established metrics was used to compute the degree to which these factors are achieved by a class and a heuristic method was used to identify the best possible weights. Moreover, a set of randomly selected Java projects were assessed using the proposed metric. The overall results obtained showed that the classes of the studied projects have an acceptable reusability. However, they showed limitations in modularity. Almost a third of all the studied classes had modularity below 0.7. Additionally, all the classes with low modularity had also a poor reusability, which makes lack of modularity the most significant factor behind poor reusability in the studied classes. Furthermore, the understandability factor was very poor in six projects where all their classes had understandability below 0.7. This was due to a near absence of code comments in these projects. However, an overall acceptable adherence to naming conventions was observed in a large number of projects combined with a manageable complexity in most of the methods included in these projects.

The impact of inner and anonymous classes on the reusability of their inclosing classes was found to be negative and significant. Almost half of the classes with poor reusability had inner and/or anonymous classes. This constitutes quite a significant negative relationship between nesting and poor reusability. Larger empirical studies must be conducted to confirm this relationship. Finally, additional factors and metrics need to be considered to strengthen the proposed model while maintaining efficiency.

References

[1] Al-Dallal, J. and Briand, L. C.“A Precise method-method interaction-based cohesion metric for object-oriented classes,”*ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, 8:1-8:34, 2012.

[2] Anquetil, N. and Lethbdige, T. “Assessing the Relevance of Identifier Names in Legacy System,” *InProc of the Centre for Advanced Studies on Collaborative Research Conference*, 1998.

[3] Boem, B. and Basili, V. “Software Defect Reduction Top 10 List,” *Software Management*, vol. 34, no. 1, pp. 135-137, 2001.

[4] Buse, R. and Weimer, W. “Learning a metric for code readability,” *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546 – 558, 2010.

[5] 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.

[6] Darcy, D. and Kemerer, C. “OO Metrics in Practice,”*IEEE Software*, vol. 22, no. 6, pp. 17-19, 2005.

[7] De-Lucia, A., Di Penta, M. and Oliveto, R. “Improving Source Code Lexicon via Traceability and Information Retrieval,” *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205-227, 2011.

[8] El-Emam, K., Melo, W. L. and Machado. J. C. “The Prediction of Faulty Classes using Object-Oriented Design Metrics,” *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[9] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] Frakes, W. and Kang, K. “Software Reuse Research: Status and Future,” *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 529-536, 2005.

[11] Gaudin, O. and Mallet, F. “Sonar,” *Methods and Tools*, vol. 18, no. 1, pp. 40-46, 2010.

[12] Gyimothy, T., Ferenc, R., and Siket, I. “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897-910, 2005.

[13] Haefliger, S., Von-Krogh, G. and Spaeth, S. “Code Reuse in Open Source Software,” *Management Science*, vol. 54, no. 1, pp. 180-193, 2008.

[14] Haiduc, S. and Marcus, A. “On the Use of Domain Terms in Source Code,” *InProc of the 16th IEEE International Conference on Program Comprehension*, pp. 113-122, 2008.

[15] Jalbert, N. and Weimer, W. “Automated Duplicate Detection for Bug Tracking Systems,” *InProc of the International Conference on Dependable Systems and Networks*, pp. 34-27, 2008.

[16] Land, R., Sundmark, D., Luders, F., Krasteva, I. and Causevic, A. “Reuse with Software Components - A Survey of Industrial State of Practice,” *Formal Foundations of Reuse and Domain Engineering, Lecture Notes in Computer Science*, vol. 5791, pp. 150-159, 2009.

[17] Langmaacka, H., Salwicki A. and Warpechowski, M. “On an algorithm determining direct superclasses in Java and similar languages with inner classes—Its correctness, completeness and uniqueness of solutions,” *Information and Computation,* vol. 207, pp. 389-410, 2009.

[18] Lawrie, D., Morrell, C. Field, H. and Binkley, D. “Effective Identifier Names for Comprehension and Memory,” *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303-318, 2007.

[19] Makela, S. and Leppanen, V. “Observations on Lack of Cohesion Metrics,” *In Proc of the International Conference on Computer Systems and Technologies*, 2006.

[20] Marcus, D. Poshyvanyk and R. Ferenc. “Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems,” *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287 – 300, 2008.

[21] McCabe, T. “A Complexity Measure,” *IEEE Transactions on Software Engineering*, pp. 308-320, 1976.

[22] McConnell, S. *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition. Microsoft Press, 2004.

[23] McIlroy, D. “Mass-produced software components,” *In Proc 1968 NATO Conference on Software Engineering, Buxton, J.M., Naur, P., Randell, B. (eds.)*, pp. 138-155, Petroceli/Charter, New York, 1969.

[24] Navarro, G. “A Guided Tour to Approximate String Matching,” *ACM Computing Surveys*, vol. 33, no. 1, pp. 31 – 88, 2001.
[25] Schildt H. *Java - The Complete Reference*, 8th Edition. McGraw-Hill Osborne Media, 2011.
[26] Stein, C., Cox, G. and Etzkorn, L. “Exploring the Relationship between Cohesion and Complexity,” *Journal of Computer Science,* vol.1, no. 2, pp. 137–144, 2005.
[27] Subramanyam, R. and Krishnan, M. “Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects,” *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297 – 310, 2003.
[28] Taibi, F. “Reusability of Open-Source Program Code: A Conceptual Model and Empirical Investigation,” *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 4, pp. 1-5, 2013.
[29] Yu, P., Systa, T. and Muller, H. “Predicting Fault-Proneness Using OO Metrics: An Industrial Case Study,” *In Proc of the* 6th*European Conference on Software Maintenance and Reengineering*, pp. 99-107, 2002.
[30] http://cccc.sourceforge.net/
[31] http://sourceforge.net/
[32] http://www.spinellis.gr/sw/ckjm/