2014 40th Euromicro Conference on Software Engineering and Advanced Applications

An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors

Giuseppe Scanniello
Universitá della Basilicata,
Dipartimento di Matematica Informatica ed Economia,
Potenza, Italy
Email: giuseppe.scanniello@unibas.it

***Abstract*—In case of biological organisms, death is unambiguous. It is not the same in software engineering, where death assumes a different meaning. For example, this term can be used to indicate source code that can be removed being it unnecessary for a given software system. In this paper, the early achievements of an empirical investigation on software metrics as predictors for dead code methods in object-oriented (OO) source code have been presented. OO design metrics and traditional code-size metrics have been considered. Data on one open source software system implemented in Java have been collected. The results suggest that five of these metrics can be used as indicators for dead code methods in OO software systems.**

Keywords - ***Code-size metric; Dead code; O-O design metrics; empirical study; software maintenance***

I. Introduction

Survival analysis deals with the death or the survival of biological organisms or with the failure of mechanical systems [1]. In case of biological organisms, death is unambiguous, while it is not the same for other kinds of biological events or physiological events (e.g., heart attack). Failures may not be well-defined and ambiguous also for mechanical systems. In software engineering, death/failure assumes different meanings. For example, Canfora *et al.* [2] reported the results of an empirical investigation aimed at characterizing the survival of bugs in open source systems, while an approach to study how dead code[1] affects evolving software systems is presented by Scanniello [3].

In commercial and open source software, dead code is a common phenomenon. It increases the file size, so excessively using memory and possibly augmenting execution time. The presence of dead code also means more code to read, thus leading to higher maintenance costs mostly due to the comprehension of dead source code [4]. The presence of dead code may also affect system reliability: source code that contains bugs is carried around and could inadvertently become operative later [5].

In an investigation on the economics of software maintenance [6], it is shown that more than 50% of the global software population is engaged in modifying existing applications rather than writing new applications. Among the kinds of work performed under the term maintenance, there is dead code removal that represents a relevant and costly activity. Although it seems important to deal with dead code, only a few studies and approaches have been proposed in the past on this topic (e.g., [3]–[5]).

In this paper, I present an explorative investigation in which a suite of design metrics is empirically investigated with regard to dead code. The object-oriented (OO) metrics proposed by Chidamber and Kemerer [7] are included in that suite as well as some "traditional" code-size metric (e.g., LOC). The goal here was to assess these metrics as predictors of dead code methods within OO software systems implemented in Java. To this end, data from an open source software have been collected. In particular, JEdit was considered. It is a well known and widely investigated system in the software engineering field. A supporting tool has been also implemented to collect metrics and to analyze the data.

Paper Structure In Section II, I shortly introduce the metrics studied in the presented investigation. The design of the investigation and its results are presented in Section III and Section IV, respectively. Related work is discussed in Section V. Final remarks and future work conclude.

II. Studied Metrics

In software engineering, the concept of software metrics is well established [8], [9]. An increasing interest in OO metrics [10]–[12] has been showed thanks to the increasing popularity of OO analysis and design methodologies.

In this paper, I used OO metrics firstly presented by Chidamber and Kemerer [7] and then slightly modified by Basili *et al.* in [10]. I considered these modified metrics and the following ones: Number of Message Sends (NMS), fan-in, fan-out, FI (Flow Info), NM (Number of Methods), Number of Methods Inherited (NMI), Lines of Code (LOC), and Number of Comments (NC). The added metrics are "traditional" code-size metrics. Many of these metrics have been suggested to estimate software quality attributes (e.g., [12]). This is one of the reasons for their selection. In addition,

[1]Unnecessary and inoperative source code that can be removed without affecting the behavior of the software.

978-1-4799-5795-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SEAA.2014.67

the presented investigation can be considered explorative in that no theory was used to choose these metrics. In fact, this investigation is the former that explores the use of OO and size metrics as predictors for dead code methods.

In the following, I shortly introduce the chosen metrics and their associated quality attributes:

- **Coupling Between Object classes (CBO)**. It represents the number of classes to which a given class is coupled. CBO evaluates efficiency and re-usability.
- **Depth of Inheritance Tree (DIT)**. It measures the number of ancestors of a class. DIT primarily evaluates efficiency and re-usability.
- **Fan-in - multiple inheritance**. It indicates the number of methods called in a software component. A class is considered here as a component. This metric is primarily used to measure maintainability.
- **Fan-out**. It is defined as the number of local flows out of a component. As for Fan-in, a class is seen as a component. Fan-out is used to measure maintainability.
- **Flow Info (FI)**. It is computed on fan-in and fan-out as follows: $L * (fan - in * fan - out)^2$. L is a code-size metric. In this respect, the LOC metric is employed. FI is employed to measure testability.
- **Lines of Code (LOC)**. It indicates the number of all nonempty and non-comment lines of the body of the class and its methods. LOC is useful to estimate maintainability.
- **Number Of Children (NOC)**. It gives the number of direct descendants for a class. NOC is used to assess efficiency, re-usability, and testability.
- **Number of Comments (NC)**. It represents the number of comment lines of the body of a class and its methods. NC might be used to evaluate understandability, re-usability, and maintainability attributes.
- **Number of Message Sends (NMS)**. It measures the number of messages from a method. This metric is designed to be an unbiased measure of the size of a method. The value for a class is achieved by summing the NMS values of all the methods in that class. This metric is used to evaluate re-usability and testability.
- **Number of Methods (NM)**. It represents the number of methods of a class and it is used to determine the complexity of a class.
- **Number of Methods Inherited (NMI)**. It gives the percentage of the number of non-redefined operations with regard to the number of inherited operations. This metric is employed to evaluate efficiency and re-usability, but it is also related to understandability and testability.
- **Response For a Class (RFC)**. It measures the number of methods that can be potentially executed in response to a message received by an object of that class. This metric evaluates understandability, maintainability, and testability.
- **Weighted Methods per Class (WMC)**. It is defined as being the number of all member functions and operators defined in each class. WMC measures the complexity of individual classes.

III. Design of the Empirical Study

The design of the investigation is presented following the guidelines suggested by Wohlin *et al.* [13]. For replication purposes, raw data are available on the web[2].

A. Definition

The goal of the investigation was to empirically analyze the design metrics selected for the purpose of evaluating whether or not these metrics are useful estimators for dead code in source classes. Assuming that dead code was properly identified, it should be a good indicator and a relevant measure of source code proneness to become dead. Several granularity levels for dead code can be considered (e.g., variable, block, method, or branch). I focused here on methods, namely unnecessary and inoperative methods.

The presented investigation aims to help researchers in assessing whether the selected design metrics are useful predictors and project managers in using these metrics in the software projects within his/her company.

B. Planning

1) Context: My investigation was conducted on the source code of an open source software system implemented in Java. This system is JEdit[3]. It is a programmer's text editor with an extensible plug-in architecture. JEdit has been chosen because of its availability on the web and its non-trivial size. This software system has been also previously used in empirical studies conducted in software engineering (e.g., [14]) and software maintenance (e.g., [3]), in particular. I randomly selected a version among the ones available on the web when this research started. This decision was taken to reduce possible biases related to the selection of the systems to be studied. The version 4.2 has been considered in the investigation. This version had: 700 classes, 84587 LOC, 41669 number of methods (including the inherited ones), 11669 NC, and 211 unused methods.

2) Hypotheses Formulation and Variable Selection: Similarly to Basili *et al.* [10] and Gyimothy *et al.* [11], the tested null hypothesis for the metric WMC was:

- H_{n_WMC} - A class with significantly more member functions than its peers is more complex and then does not tend to be more prone to include dead methods.

The alternative hypothesis is:

- H_{a_WMC} - A class with significantly more member functions than its peers is more complex and then tends to be more prone to include dead methods.

[2] www.scienzemfn.unisa.it/scanniello/MetricsDeadCode/
[3] www.jedit.org/

For all the other considered metrics, a null hypothesis has been defined and tested in a similar fashion. For space reason and scant relevance, these null hypotheses are not reported here. However, they can be easily derived following the definition of the null hypothesis H_{n_WMC}.

The considered dependent variable is: *DeadM*. It measures the number of dead methods for a class. The independent variables were thirteen, one for each metric. The acronym of a metric indicates the name of the corresponding independent variable.

3) Instrumentation: To get the number of dead methods in a subject system, the best solution should be the analysis of all its methods. For medium/large sized systems, the identification of dead methods is practically impossible because static and dynamic analyses on the source code are needed. The results of a dynamic analysis could be strongly affected by the execution of the system and by the use of execution traces [15]. In this regard, a subset (or sample) of the dead methods among those returned by JTombstone[4] (the population) has been analyzed. This subset was selected employing the random sampling method without replacement [16]. I analyzed the methods in this subset to verify whether they were or they were not actual dead methods. The size of this sample was computed by applying the formula proposed by Daniel [17]:

$$n = \frac{Z^2 * P * (1 - P)}{d^2}$$

The Z value was 1.96. It conventionally corresponds to a level of confidence to 95%. To be conservative, I used 50% (in proportion of one) as the expected prevalence P and 5% (in proportion of one) as the d value (the expected desired precision). The achieved value for n (i.e., 384) was larger than the number of the dead methods returned by the tool ($N = 211$). To compute the size of the sample, the formula $\left\lceil \frac{N*n}{N+n} \right\rceil$ was applied. The size of the sample was then 136. The rationale for using a subset of the population relies on the time needed to assess whether a method is actually dead. The use of a sample statistically significant provides a good estimate of the correctness of the results provided by JTombstone.

To collect the metrics from the source code of JEdit, the implemented tool integrates Moose[5].

4) Execution and Data Analysis: To analyze the relationships between the metrics and the proneness of classes to contain dead methods, I employed univariate and multivariate regression analysis methods. I performed neither univariate nor multivariate logistic regression analyses, because they will only predict the presence or the absence of dead methods. It is worth noting that the number of dead methods in the considered classes was in between 0 to 34. Therefore, estimating the presence or the absence of dead methods could be useless.

[4]jtombstone.sourceforge.net
[5]www.moosetechnology.org/

Table I
RESULTS OF UNIVARIATE REGRESSION ANALYSES

Metric	Coefficient	R^2	p-value
LOC	0.14	0.21	$<$ **0.001**
RFC	0.099	0.10	**0.009**
WMC	0.09	0.08	**0.017**
NMS	0.084	0.007	**0.026**
NM	0.084	0.007	**0.026**
DIT	-0.73	0.05	0.053
NOC	-0.12	0.001	0.75
CBO	-0.068	0.05	0.74
Fan-in	-0.18	0.003	0.633
Fan-out	-0.023	0.001	0.544
FI	-0.009	5.69E-04	0.818
NMI	-0.51	0.03	0.18
NC	0.019	0.003	0.610

Regression analyses are widely used to predict an unknown variable based on one or more known variables (e.g., [11]). I applied univariate analyses to examine the effect of each metric separately. To obtain an optimal model, I included the metrics into a multivariate regression model. The multivariate regression analysis allows studying the combined predictive power of all the metrics. Only the metrics that significantly improve the predictive power of the multivariate model were included through a stepwise selection process. I used a standard stepwise process with a conventional classification threshold: $\pi \leq 0.5$. I applied linear regression analysis without testing its assumptions, e.g., linearity (a linear relationship between independent and dependent variables exists), homoscedasticity (the variance of error is constant), and normality (the distribution of error is normal). This was because actual data rarely satisfies these assumptions. Furthermore, in studies similar to the one presented here (e.g., [11]) assumptions are not tested.

To measure the goodness of models obtained with regression analyses, R^2 was used. The higher the value, the higher the effect of the model's explanatory variables is and the more accurate the model is. To estimate the models, I also considered regression coefficient. The larger the coefficient in absolute value, the stronger the impact (positive or negative) of the variable is. The statistical significance (i.e., p-value) provides an insight into the accuracy of the estimates. As customary, I used $\alpha = 0.05$ as the significance threshold value here.

IV. RESULTS

Table I shows the results of the univariate analyses. We can see that five metrics (i.e., WMC, RFC, NMS,NM, and LOC) are statistically significant (p-value < 0.05), so rejecting the corresponding null hypotheses and then accepting the

Table II
RESULTS OF MULTIVARIATE REGRESSION ANALYSIS

Metric	Coefficient	p-value
Intercept	0.674	< 0.001
LOC	0.023	< 0.001
WMC	-0.068	< 0.001
NC	-0.044	< 0.001
Fan-out	-0.087	< 0.001
RFC	0.355	< 0.001
FI	-2.46E-9	0.003
CBO	-0.408	0.032

alternative ones. Among the statistically significant metrics, LOC had the largest R^2 value.

The results of the multivariate analysis are summarized in Table II. The metrics in the model were selected with the following order: LOC, WMC, NC, Fan-out, RFC, FI, and CBO. The intercept and all the metrics have a p-value less than 0.005 (i.e., these metrics are very significant). The R^2 value of the model is 0.274. It is higher than the highest R^2 value achieved in the univariate analyses (see Table I). This result suggests that the prediction model built with the multivariate analysis is more accurate than each model that has been obtained by applying the univariate analysis on each considered metric.

LOC was found to be significant in both the kinds of regression analyses (i.e., univariate and multivariate). This result suggests that the larger the class, the higher the number of dead methods is. Similarly, WMC is statistically significant, so indicating that the class complexity is a useful predictor for the presence of dead code. Also, RFC can be considered a useful predictor because it is significant in both regression analyses. The number of methods (NM) and their size (NMS) are somewhat poorer predictors as the built model shows.

A. Threats to Validity

The use of an open source software system may threaten the validity of the results. A such a kind of software is usually developed outside companies mostly by volunteers and the development methodologies used are different from those commonly applied in the software industry. Although many large companies are using open source software in their own work or as a part of their marketed software, it will be worth replicating the investigation on commercial software implemented also with programming languages different from Java.

Other threats are related to what was the best way to measure the size of classes given the large number of alternatives. To reduce possible biases, well known code-size and OO metrics [7], [9] were used. These metrics were collected by using the Moose system. It is developed by others and it is widely used in the software engineering field.

The reliability of JTombstone may also affect the obtained results. It could be possible that JTombstone was not able to detect all the actual dead code. Due to the preliminary nature of this study, this represents an acceptable threat to the construct validity. However, future work is needed to build and to share a benchmark with the research community even if it could be practically impossible to manually identify of dead methods in medium- to large-sized software systems.

V. RELATED WORK

This section discusses approaches based on survival analysis or that use this kind of analysis to estimate relevant aspects of software projects. For example, Sentas *et al.* [18] propose statistical tools for studying and modeling the distribution of the duration of software projects. One of the most important advantages of the approach is that data from completed and uncompleted projects can be used for the estimation. Similarly, Sentas *et al.* [19] define a method to predict the duration of a software project. The authors used this statistical method since it allows to utilize information not only from the completed projects in a dataset but also from ongoing projects.

The survivability of software projects is investigated by Samoladas *et al.* [20]. The main feature of the method is the construction of probabilistic models based on data of complete and ongoing projects. The authors considered open source projects to validate their proposal. In particular, the goals of the study consisted in knowing the chances for the survival of the project, how viable a project to either participate or invest in it, and how volunteers want to contribute to the project. To this end, survival analysis is employed to predict the survivability of the projects. In particular, Samoladas *et al.* studied the probability of continuation in the future, by examining their duration combined with other project characteristics such as their application domain and the number of committers.

Mockus [21] a method for empirically evaluating the availability of deployed software systems is presented. The method is based on survival analysis and uses information from operational customer support and inventory systems. The results of a study on a large communications system to investigate the factors that could affect software reliability are also presented. Differently, Wilson and Samaniego [22] propose a software reliability model based on the order-statistic paradigm. The objective of this model is to estimate an unknown number of bugs in the software and to predict failures. Recently, Scanniello *et al.* [23] proposed a defect prediction approaches based on the use of software metrics and fault data to learn which software properties associate with faults in classes. This approach can be considered intra-release and learns from clusters of related classes, rather than from the entire system. Clusters were grouped on the basis of their structural information. Fault prediction models

were built using the properties of the classes in each cluster and multivariate linear regression was used. 29 releases of eight open source software systems from the PROMISE repository were used to empirically assess the approach. The results indicate that the prediction models built on clusters outperform those built on all the classes of the system.

In the software maintenance field, the results of an interesting empirical study are presented by Hutton and Welland [24]. The main objective of this study is to investigate whether programmers unfamiliar with a system to maintain start by actively working on code or spend time to passively explore the system before performing changes. Since some of the participants to the study do not complete the maintenance tasks, the authors use Kaplan Meier and the Cox Proportional Hazard model to analyze the data.

Differently from the work presented here, the approaches described above do not consider dead code and many of them do not use the Kaplan Meier estimator to study this phenomenon in evolving software systems. The empirical work presented in this paper represents the first one in such a direction.

VI. Conclusion and Future Work

A number of works have been conducted to empirically investigate whether OO design metrics can be used as early quality indicators (e.g., [10], [11], [25]). For example, Basili *et al.* [10] observed that some Chidamber and Kemerers metrics are useful to predict class fault-proneness. The authors also reported that those metrics are better predictors than "traditional" code-size metrics. Similar results were achieved on open source software [11]. No studies have been conducted to understand whether those metrics are good predictors of dead code methods in OO software. Indeed, such a kind of dead code has been scarcely studied in the past. For example, an analysis on dead code in evolving open source software is reported in [3]. Differently from that paper, I have shown here the results of an empirical investigation on the proneness of OO source code to become dead. To this end, a predictive model based on software metrics has been built. The elimination of dead code requires a huge cost and effort [6], [26]. Therefore, it seems reasonable to support a software engineer in the decision of removing or not dead code by refactoring operations. The results obtained have suggested that two Chidamber and Kemerer's metrics and three code-size metrics appear to be useful to predict dead code. The results also suggest that dead code might affect some quality attributes of source code (i.e, testability/maintainability, re-usability, and readability). Future work is however needed on this point.

Due to the preliminary nature of this investigation, there are several future directions for the presented research. For example, it will be worth investigating whether the obtained results also hold for different systems possibly implemented in programming languages different from Java. The management of different granularity levels for the dead code (e.g., at statement block level) is also the subject of future work. Furthermore, the tool prototypes could be extended to access information in software configuration management tools (e.g., CVS and SVN). Finally, it could be advisable to study the effect of applying different estimation methods and techniques to predict dead code.

Acknowledgment

I would like to thank the Carmine Gravino for his precious help and for his valuable comments and suggestions. I also thank the Angela Angiolillo, who developed some of the software modules of the tool prototype mentioned in this paper.

References

[1] R. C. Elandt-Johnson and N. L. Johnson, *Survival models and data analysis / Regina C. Elandt-Johnson, Norman L. Johnson*. John Wiley & Sons, New York, 1980.

[2] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *Proceedings of Working Conference on Reverse Engineering*. IEEE Computer Society, 2011, pp. 191–200.

[3] G. Scanniello, "Source code survival with the kaplan meier estimator," in *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2011, pp. 524–527.

[4] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, "A c++ data model supporting reachability analysis and dead code detection," *IEEE Trans. on Softw. Eng.*, vol. 24, pp. 682–694, 1998.

[5] Z. Huang and L. Carter, "Automated solutions: Improving the efficiency of software testing," 2003.

[6] C. Jones, "The economics of software maintenance in the twenty-first century," 2006.

[7] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, June 1994.

[8] T. DeMarco, *Controlling Software Projects: Management, Measurement and Estimation*. New York, NY: Yourdon, 1982.

[9] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer Verlag, 2010.

[10] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, pp. 751–761, October 1996.

[11] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 897–910, October 2005.

[12] L. H. Rosenberg and L. E. Hyatt, "Software Quality Metrics for Object-Oriented Environments," http://satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.PDF, May 1997.

[13] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000.

[14] S. Di Martino, F. Ferrucci, C. Gravino, and F. Sarro, "A genetic algorithm to configure support vector machines for predicting fault-prone components," in *Proceedings of International Conference on Product-focused Software Process Improvement*. Springer-Verlag, 2011, pp. 247–261.

[15] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Proceedings of International Conference on Program Comprehension*. IEEE Computer Society, 2011, pp. 1–10.

[16] M. Triola, *Elementary Statistics*. Pearson College Div, 1998.

[17] W. Daniel, *Biostatistics: a foundation for analysis in the health sciences*, ser. Wiley series in probability and mathematical statistics. Wiley, 1995.

[18] P. Sentas, L. Angelis, and I. Stamelos, "A statistical framework for analyzing the duration of software projects," *Emp. Softw. Eng.*, vol. 13, no. 2, pp. 147–184, 2008.

[19] P. Sentas and L. Angelis, "Survival analysis for the duration of software projects," *In Software Metrics*, vol. 0, p. 5, 2005.

[20] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Inf. and Softw. Tech.*, vol. 52, pp. 902–922, September 2010.

[21] A. Mockus, "Empirical estimates of software availability of deployed systems," in *Proceedings of Empirical Software Engineering and Measurements*. ACM Press, 2006, pp. 222–231.

[22] S. P. Wilson and F. J. Samaniego, "Nonparametric analysis of the order-statistic model in software reliability," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 198–208, March 2007.

[23] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies, "Class level fault prediction using software clustering," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society Press, 2013, pp. 640–645.

[24] A. Hutton and R. Welland, "An experiment measuring the effects of maintenance tasks on program knowledge," in *Proceedings of Evaluation and Assessment in Software Engineering*. BCS Press, 2006.

[25] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 297–310, April 2003.

[26] B. Andreopoulos, "Satisficing the conflicting software qualities of maintainability and performance at the source code level," in *Workshop em Engenharia de Requisitos*, 2004, pp. 176–188.