J. Basic. Appl. Sci. Res., 3(2s)66-71, 2013

ISSN 2090-4304
Journal of Basic and Applied Scientific Research
www.textroad.com

Definition of Fault-proneness at Early Phase in Object-Oriented Development

Mansoure Ashourion[1]

Faculty Members of Payame Noor University, Computer Engineering, Iran, Esfahan

Received: June 10 2013
Accepted: July 10 2013

ABSTRACT

To analyze the complexity of object-oriented software, several metrics have been proposed. Among them, Chi-damber and Kemerer's metrics are well-known ones as object-oriented metrics. Also, the effectiveness has been empirically evaluated from the viewpoints of estimating the fault-proneness of object-oriented software. In the evaluations, their metrics were applied to not design specification but the source code because some of them measure an inner complexity of a class, and such information cannot be obtained until the algorithm and structure of the class are determined at the end of design phase. However, the estimation of the fault-proneness should be done in the early phase to effectively allocate effort for fixing the faults. This paper proposes a new method to estimate the fault-proneness of the class in the early phase, using several complexity metrics for object-oriented soft- ware. In the proposed method, we introduce four check- points into the analysis / design / implementation phase, and estimate the fault-prone classes using the applicable metrics at each checkpoint.
KEYWORDS: Fault-proneness, metric design complexity ,object oriented design

1. INTRODUCTION

In attempt to reduce the number of delivered faults, it was reported that most companies spend between 50-80% of their software development effort on testing[7]. There- fore, reducing the effort of testing is a key to high productivity in software development. Software review is one of the most effective techniques to reduce the testing effort. In order to effectively review and test the software products, it is needed to identify the fault-prone modules so that review and testing effort can be concentrated on the modules[1]. Several complexity metrics have already been proposed to localize such the fault-prone modules. Then, Chidamber and Kemerer proposed six complexity metrics for OO software [5]. However, the metrics suite was applied to the source code because some of their metrics measure an inner complexity of a class, and such information cannot be obtained until the algorithm and structure of the class are determined at the end of design phase. The estimation of the fault-proneness should be done in the early phase to effectively allocate the effort for fixing the faults.

This paper proposes a new method to estimate the fault-proneness of the class in the early design phase, using several complexity metrics for OO software. In the proposed method, we introduce four checkpoints into the development process based on OMT[14]. According to the data obtained from the products (design specification and source code) at each checkpoint, we use only the applicable metrics among the complexity metrics and estimate the fault-proneness by using the multivariate logistic regression analysis[18]. Then, we have applied the proposed method to an experimental project held at a computer company. The analysis result of the experiment shows the validity and usefulness of the proposed method in the estimation of the fault-proneness.

2. Preliminary

2.1. Object-Oriented Design Method

There exist several object-oriented design methods [2][6][14][15]. Among them, OMT[14] is a broad methodology that addresses most aspects of the OO analysis and design technology and incorporates the lessons learned from real projects as well as from other methodologies.

OMT consists of three phases: Analysis, System Design and Object Design. Analysis is concerned with under-standing and modeling the application and the domain within which it operates. System Design determines the overall architecture of the system. The system is organized into subsystems. Concurrency is organized by grouping objects into concurrent tasks. Object Design elaborates, refines and then optimizes the analysis models to produce a practical design. Algorithms and data structures used in the classes are determined.

From the viewpoint of the structure of the class, elabo- ration is carried out as the following six steps:

1. Identify classes in the target system.
2. Identify the reference-relationships between the classes.
3. Identify the attributes of the classes.

This article is extracted from a plan that has been approved to support payamenoor university with subject" Definition of Fault-proneness at Early Phase in Object-Oriented Development"

Corresponding Author: Mansoureh Ashourion, Faculty members of Payame Noor University, Computer Engineering, Iran, Esfahan,
E_mail : ashourion@pnu.ac.ir

4. Determine the derivation-relationships between the classes.
5. Define the operations based on the functional models.
6. Design the algorithms to implement the operations.

2.2. Complexity Metrics for Object-Oriented Software

Software metrics are quantitative measures of software products and process[8]. Especially, the complexity metric for a program code is the most well-known metric, and is usually used in the implementation phase and test phase, since it is a good indicator of whether the product is well- designed, understandable, and easy to modify. For example, Software Science by Halstead[9] and Cyclomatic Number by McCabe have been used in many software development organizations. Then, Chidamber and Kemerer[5] have defined a suite of metrics for OO design. The metrics suite includes six metrics and mainly evaluate the relationship between the classes on the design specification. Each metric evaluates the complexity of the target class and the definitions of the metrics are as follows[5]:

WMC(weighted methods per class):

Consider the target class C_1, with methods M_1, ..., M_n, that are related in the class. Let c_1, ..., c_n be the complexity of the methods. Then $WMC = \sum c_i$. You ought to choose an adequate interval scale metric f that gives $c_i = f(M_i)$.

In both [1] and [5], they made an assumption that all method complexities are considered to be unity, then WMC is the number of methods. We also use this as- assumption, so we plainly employ a term NIM(Number of Instance Methods)[12] as WMC.

DIT(depth of inheritance tree):

Depth of inheritance of the target class is the DIT metric for the class.

NOC(number of children):

NOC means the number of immediate subclass subordinated to the target class in the class hierarchy.

CBO(coupling between object classes):

CBO for the target class is a count of the number of other classes to which it is coupled. A class is coupled to another if it uses is methods or instance variables of an- other.

RFC(response for a class):

$RFC = |RS|$ where RS is the response set for the class. $RS = M\ (\text{all } i\ R_i)$ where R_i = set of methods called by method i and M = set of all methods in the class.

LCOM(lack of cohesion in methods):

Consider the target class C_1 with n methods M_1, ..., M_n. Let I_i = set of instance variables used by methods M_i. There are n such sets I_1, ..., I_n.

Let $P = \{ (I_i, I_j) \mid I_i \quad l_j = \}$ and $Q = \{ (I_i, I_j) \mid I_i\ I_j \quad \}$. If all n sets I_1, ..., I_n are then let $P = \ . LCOM = \max(|P| - |Q|, 0)$.

Besides the above metrics, NIV (Number of Instance Variables) is frequently used as a design metric of object- oriented software[12].

NIV(number of instance variables):

NIV is a number of instance variables of the target class.

2.3. Evaluation of Complexity Metrics

Chidamber and Kemerer have presented empirical data on their metrics from actual commercial systems. They have adopted the Weyuker's properties to evaluate their metrics (Weyuker has developed a list of desiderata for software metrics and has evaluated a number of existing software metrics using these properties[16]), and concluded that the metrics are generally satisfied the Weyuker's properties. Basili et al. empirically evaluated that Chi- damber and Kemerer's OO metrics show to be better predictors than the best set of traditional code metrics[1].

Table 1. Checkpoint and available metrics.

Available Metrics	Added Information	Checkpoint
NIV, CBON	Reference-relationship among classes and attributes of classes	(CP1) Entity and Relation
NIV, CBON, CBOR, CBO, NIM, DIT, NOC	Class hierarchy, methods and reused library	(CP2) Structure and inheritance
NIV, CBON, CBOR, CBO, NIM, DIT, NOC, RFC, LCOM	Algorithm of the methods	(CP3) Algorithm
NIV, CBON, CBOR, CBO, NIM, DIT, NOC, RFC, LCOM, SLOC	Source code	(CP4) Implementation

In both [1] and [5], they collected the metrics values from source codes. But the design specification is constructed in the earlier development phase. Thus, by applying the metrics to the design specification, it is possible to estimate the fault proneness more effectively. It would produce efficient fault detection to do the resource- allocation and coordinate the

scheduling based on the result of fault-estimation. However, it is very difficult to apply the metrics to the design specification. For example, with respect to calculating the RFC and LCOM, we need the detail information about the algorithms in the method and call-relationship between the methods. These information are usually described on the design specification at the later design phase (just before the implementation phase).

3. Proposed Method

3.1. Key Idea

We regard the analysis / design / implementation phase as a series process in which the information about software product gradually increases as the process progresses. As described in Section 2, some of the metrics can be applied to the design specification at the earlier design phase, and some of them can be applied at the later design phase. Based on the fact, as design phase progress, we estimate the fault-proneness only using the applicable metrics to the design specification.

At first, we introduce four checkpoints in the development process and identify which information has been added to the design specification at each checkpoint. Then, we define the subset of the conventional metrics applicable to the design specification developed at each checkpoint. Finally, at each checkpoint, we estimate the fault-proneness of the class using the multivariate logistic regression analysis with the applicable metrics.

3.2. Checkpoints and Applicable Metrics

In Section 2, we divided the development process of OMT into the following six steps. Based on the division, we introduce the four checkpoints in the analysis / design / implementation phase of OMT.

(CP1) Entity and relation

CP1 is the checkpoint where Steps 1, 2 and 3 have been completed. That is, reference-relationship between classes (coupling) and attributes (instance variable) of the classes have been determined. NIV can be calculated from the attribute information. The derivation-relationships have not been described. CBO can be calculated from the reference-relationship between classes, but reference to the reused class is not clearly described so that the value of CBO is not correct.

(CP2) Structure and inheritance

CP2 is the checkpoint where Steps 4 and 5 have been completed. That is, derivation between classes and the methods in the classes have been determined and class hierarchy tree is clearly described. Thus, DIT can be calculated. NIM can be calculated from the information of the methods. The reused classes are determined so that CBO can be calculated correctly.

(CP3) Algorithm

CP3 is the checkpoint where Step 6 has been completed. That is, algorithms in each method and call-relationship between the methods are determined. Based on the information, LCOM and RFC can be calculated.

(CP4) Implementation

CP4 is the checkpoint where source code has been implemented. For each class, SLOC (source lines of code) can be calculated.

CBO at CP1 (hereafter: *CBON*, CBO newly developed) differs from original definition. However, our previous research result showed that CBON have highly correlated with the fault-proneness[10]. We also introduce *CBOR*(CBO reused) that is defined as *CBOR* = *CBO* – *CBON*. Table 1 summarizes the checkpoints and the met- rics which can be calculated at each checkpoint.

3.3. Multivariate Logistic Regression Analysis

Recently, the multivariate logistic regression analysis, whose inputs are several metrics, has been frequently used to evaluate the fault-proneness of the program[1][3].

A multivariate logistic regression model is based on following relationship equation:

$$P(X_1, ..., X_n) = 1 / \{ 1 + \exp(-(C_0 + C_1 X_1 + ... + C_n X_n))\}$$

where P is the probability that errors is found in a class, and X_is are metrics of the class. If given metrics values make P greater than 0.5, the class is predicted to have faults (fault prone).

To obtain particular prediction equation $P(X_1, ..., X_n)$, we need to determine the coefficients $C_0, ..., C_n$ by *step-wise variable selection process*[18], using observed metrics and fault data.

4. Empirical Evaluation

4.1. Outline of the experiment

The experimental project was performed at a computer company for five days in August 1997. The main characteristics of the project can be summarized as follows:

(1) The Developers were new employees of the computer company and had just graduated from college in March 1997. All the developers studied object-oriented design and programming in C++ Language.

(2) There are sixteen developer teams of four or five developers. Each team built a mail delivery system of an identical

requirement. The predetermined system requirements, the division to subsystems and subsystem's interface designs were handed over to each team.

(3) After a team declared that its program was finished, the instructors executed an acceptance test.

4.2. Empirical Data

We collected complexity metrics and fault data from each developer. Unfortunately, we could not collect the design specification based on OMT in this experiment. We assume that all the information of the design specification

Table 2. Descriptive statistics of the 141 C++ classes.

Std. Dev.	Mean	Me-dian	Max.	Min.	Metrics
2.67	4.00	3	14	0	NIV
1.59	1.39	1	5	0	CBO
0.99	0.53	0	3	0	CBON
0.99	0.86	1	4	0	CBOR
4.86	5.73	3	22	0	NIM
1.41	3.44	4	6	0	DIT
0.00	0.00	0	0	0	NOC
6.81	8.23	7	27	0	RFC
36.84	22.42	3	190	0	LCOM
81.01	96.43	71	420	0	SLOC

Table 3. Coefficients at Each Checkpoint.

Coefficients		
CP4	CP3	CP2
-2.69	-1.31	-1.23
EL	EL	EL
EL	EL	EL
EL	0.890	0.934
EL	EL	EL
EL	EL	0.336

Ec is the number of faults detected in the class. Et is the time spent for fixing the faults.

The 'EL' means that the metric was eliminated form the prediction equation by a backward variable elimi- nation process. The '-'s are not applicable at the checkpoint.

Table 4. Fault Prediction at CP1.

	Fault	No fault	Prediction
Actual	2	112	No fault
	9(37)	18(43)	Fault

The figures before parentheses are the number of classes. The figures within parentheses are the num- ber of faults in those classes.

Table 5. Fault Prediction at CP2.

	Fault	No fault	Prediction
Actual	5	109	No fault
	16(60)	11(20)	Fault

Table 6. Fault Prediction at CP3.

	Fault	No fault	Prediction
	3	111	No fault
Actual	18(62)	9(18)	Fault

Table 7. Fault Prediction at CP4.

Fault	No fault	Prediction	
3	111	No fault	Actual
19(66)	8(14)	Fault	

Table 8. Fault Predict Precision at Checkpoint.

CP4	CP3	CP2	CP1	Checkpoint
86	85	76	82	Correctness(%)
70	63	59	33	Completeness(%)
83	71	75	46	Error-based Completeness (%)

were included in the source code since it was implemented based on the design specification. So, we collected the metrics values, which can be calculated at each checkpoint, from source code by using a metrics tool that extracts nine metrics from C++ source code[11].

We prepared a documentation tool to collect fault data. We have eliminated the data of developers who did not report data or irresponsible data. As a result, seventeen members' data (141 classes and 80 faults) are available. Table 2 shows common descriptive statistics of the metrics distributions.

4.3. Analysis

Table 3 shows the prediction model's coefficients cal- culated by multivariate logistic regression analysis. DIT appeared as a negative factor to complexity. This can be explained by the fact that there were a lot of 'dialog' classes developed and the functionalities were very simple but the their DIT values were relatively large. LCOM is eliminated from a equation at CP4, and this result agrees with the results in [1].

Tables 4, 5, 6 and 7 show the obtained prediction model by the multivariate logistic regression with the data col- lected at each checkpoint. For example, in Table 4, 112classes are predicted to be no-faulty and actually no-faulty. Eighteen classes are predicted to be no-faulty but actually faulty (include 43 faults).

Here, we introduce the following three criterions to evaluate the result of the prediction:

$$Correctness = C_{PF\ AF} / (C_{PF\ AF} + C_{PF\ AN}) \quad Completeness = C_{PF\ AF} / (C_{PF\ AF} + C_{PN\ AF})$$
$$Completeness_{\text{error based}} = E_{PF\ AF} / (E_{PF\ AF} + E_{PN\ AF})$$

where $C_{PF\ AF}$ is the number of predicted faulty and actu- ally faulty classes, $C_{PF\ AN}$ is number of predicted faulty but actually no-faulty classes, $C_{PN\ AF}$ is number of predicted no-faulty but actually faulty classes, E_X is number of faults found in corresponding C_X classes.

Table 8 shows the estimation accuracy of the fault proneness at each checkpoint. According to the progress of the checkpoints, the estimation accuracy is improved. CP4 is the last phase of the development process and the estimation accuracy at CP4 is the upper limit in this ex- periment.

Completeness at CP1 is relatively low (33%). On the other hand, correctness is high (82%). Thus, the estimati- on can be used to *seed* the fault-prone classes. The seeded classes become the candidates that should be reviewed and tested selectively. Also, the location of the seeded classes might be the criterion of the judgment for review. For example, if the seeded classes are concentrated on the important section of the design specification and the sec- tion is difficult to test, we should redesign it.

Though the metrics for algorithms of the method can- not be used at CP2, completeness at CP2 is excellent with respect to the upper limit at CP4. It suggests that it would be possible to estimate the fault-proneness from the design specification in the design phase where the algorithms are not determined, without source code.

The result of estimation at CP3 fell short of our expec- tations. We consider that the accuracy would be improved by using 'fine-grained' C++ design metrics[3] together at CP3. The accuracy would be improved by cyclomatic number as WMC.

5. Conclusion

In this paper, we proposed a new method to estimate the fault-proneness of the class in the early design phase, using several complexity metrics for OO software. We have introduced four checkpoints into the analysis/design/ implementation phase, in which particular subsets of metrics are applicable. We have applied the proposed method to an experimental project. The analysis result shows the validity and usefulness of the proposed method.

REFERENCES

[1] V. R. Basili, L. C. Briand, W. L. Mélo: A validation of object-oriented design metrics as quality indicators, *IEEE Trans. on Software Eng.*, Vol. 20, No. 22, pp. 751-761 (2005).

[2] G. Booch: *Object Oriented Analysis and Design with Applications*, The Benjamin / Cummings (2010).

[3] L. C. Briand, P. Devanbu, and W. Mélo: An Investigation into Coupling Measures for C++, *Proc. of the19th Int'l Conference on Software Eng.*, Boston, USA, pp.412-421 (2001).

[4] L. C. Briand, J. Daly, V. Porter and J. W st: Pre- dicting Fault-Prone Classes with Design Measures in Object-Oriented Systems, *Proc. of the 9th Int'l Sym- posium on Software Reliability Eng.*, Paderborn, Germany, pp.334-343 (2003).

[5] S. R. Chidamber and C. F. Kemerer: A metrics suite for object-oriented design, *IEEE Trans. on Software Eng.*, Vol. 20, No.6, pp.476-493 (1994).

[6] P. Coad and E. Yourdon: *Object Oriented Analysis, 2nd ed.*, Yourdon Press (2008).

[7] J. S. Collofello and S. N. Woodfield: Evaluating the effectiveness of reliability-assurance techniques, *Journal of Systems & Software*, Vol.9, No.3, pp.191-195 (1989).

[8] N. E. Fenton, and S. L. Pfleeger, *Software Metrics, A Rigorous & Practical Approach. 2nd ed.*, Thomson, London (1996).

[9] M. H. Halstead: *Element of software science*, New York, Elsevier North-Holland (1977).

[10] T. Kamiya, S. Kusumoto, K. Inoue and Y. Mohri: Empirical evaluation of reuse sensitiveness of com- plexity metrics, *Information and Software Technol- ogy* (to appear).

[11] T. Kamiya, S. Takayabashi, S. Kusumoto and K. Inoue: Measurement tool for complexity of C++ pro- gram, Objecto-Oriented '98 Symposium, Tokyo, Ja- pan (2008) (in Japanese).

[12] M. Lorenz and J. Kidd: *Object-Oriented software metrics*, New Jersey, Prentice Hall (1994).

[13] T. J. McCabe: A complexity measure, *IEEE Trans. on Software Eng.*, Vol. SE-2, No.4, pp.308-320 (2000).

[14] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen: *Object Oriented Modeling and Design*, Prentice Hall (2006).

[15] S. Schlaer and S. Mellor: *Object Oriented System Analysis*, Prentice Hall (2004).

[16] E. J. Weyuker: Evaluating software complexity measures, *IEEE Trans. on Software Eng.*, Vol.14, No.9, pp.1357-1365 (2005).

[17] *UML Modeling Language, Standard Software Nota- tion*. Available at <http://www.rational.com/>.

[18] *SPSS Base 8.0 Applications Guide/Professional Statistics*, SPSS (2010).