IEICE TRANS. INF. & SYST., VOL.E88–D, NO.6 JUNE 2005

LETTER

A Simple Predictive Method for Discriminating Costly Classes Using Class Size Metric*

Hirohisa AMAN[†a)], ***Member***, **Naomi MOCHIDUKI**[†], ***Nonmember***, **Hiroyuki YAMADA**[†], ***Member***, ***and*** **Matu-Tarow NODA**[†], ***Nonmember***

SUMMARY Larger object classes often become more costly classes in the maintenance phase of object-oriented software. Consequently class would have to be constructed in a medium or small size. In order to discuss such desirable size, this paper proposes a simple method for predictively discriminating costly classes in version-upgrades, using a class size metric, Stmts. Concretely, a threshold value of class size (in Stmts) is provided through empirical studies using many Java classes. The threshold value succeeded as a predictive discriminator for about 73% of the sample Java classes.

key words: *object oriented software, modification effort, prediction, class size, metrics*

1. Introduction

Class size is an important attribute of object-oriented software. Larger classes often become more costly classes in the maintenance phase, since larger ones would need more time and effort to review, upgrade and/or reconstruct. Consequently class would have to be constructed in a medium or small size. Eman et al.[1] have discussed the optimal size of class, and empirically showed faulty classes tend to be larger than not-faulty classes. We focus on version-upgrades as well as the number of faults, especially the lines of codes modified through the upgrades. Version-upgrades can include not only fault fixings but also software refactorings and evolutions. Consequently version-upgrades would be important to more software developers and managers. In order to discuss an optimal class size in terms of version-upgrades, we propose a simple method for predictively discriminating the costly classes from the others, using a class size metric. Concretely, we empirically derive a threshold value of class size from many version-upgrade cases in Java classes, and validate it as a predictive discriminator for the costly classes.

Manuscript received October 8, 2004.
Manuscript revised January 25, 2005.

[†]The authors are with the Department of Computer Science, Faculty of Engineering, Ehime University, Matsuyama-shi, 790–8577 Japan.

*This paper was presented at the 6th Joint Conf. on Knowledge-Based Software Eng. (JCKBSE2004), Moscow, Russian Federation, Aug. 2004.

a) E-mail: aman@cs.ehime-u.ac.jp

DOI: 10.1093/ietisy/e88–d.6.1284

2. Predictive Discrimination of Costly Classes by Class Size Metric

2.1 Costly Class

To discuss costly classes in not only closed software projects but also open source software projects, this paper focuses on the number of source code lines modified through the version-upgrade, and considers it to be an important factor of the version-upgrade cost. Formally, let c be a class, and it has upgraded from the version v_i to the next version v_{i+1}; we represent the older version as c_{v_i}, and the newer one as $c_{v_{i+1}}$. Then let $d(c_{v_i}, c_{v_{i+1}})$ be the number of different source code lines between c_{v_i} and $c_{v_{i+1}}$, i.e., the modified lines through the version-upgrade. The different lines contain the changed lines, added lines and deleted lines, excepting the comment statements and empty lines. After filtering out the comment statements in source program, those different lines can be counted up by the Unix command "`diff`" with the option "`-cbwB`"**. $d(c_{v_i}, c_{v_{i+1}})$ would be an important measure of the version-upgrade cost that went into c_{v_i}.

When a class c_{v_i} has a large number of different lines with the next version $c_{v_{i+1}}$, we consider c_{v_i} to be a costly class. Formally, we define it as follows.

Definition 1 (costly class; decision parameter α):
Given n version-upgrade cases for many classes, say $\{c_{v_i}, c_{v_{i+1}}\}$. Consider a real number α $(0 \leq \alpha \leq 1)$, and if $d(c_{v_i}, c_{v_{i+1}})$ is in the largest $\lceil \alpha \cdot n \rceil$ cases of all version-upgrade cases***, then we define c_{v_i} to be a *costly class*. We will say such $\{c_{v_i}, c_{v_{i+1}}\}$'s as "the worst α (%) cases." □

The parameter α would be empirically decided by software managers, to determine their "wrong" version-upgrade cases. For example, when we have 1234 $(= n)$ version-upgrade cases and $\alpha = 0.05$ (5%), we select the largest 62 $(= \lceil 0.05 \times 1234 \rceil = \lceil 61.7 \rceil)$ cases corresponding to the worst 5% cases.

2.2 Class Size Metric

Now we introduce a class size metric for our predictive discrimination of costly classes. Class size is one of the most

**GNU diffutils version 2.8.1.

***The ceiling function $\lceil \cdot \rceil$ is used for only making $\alpha \cdot n$ into an approximate natural number.

```
while ( i < 10 ){
   if ( i % 2 == 0 ) x += i;
   i++;
}
```

```
while ( i < 10 )
{
   if ( i % 2 == 0 )
     x += i;
   i++;
}
```

Fig. 1 Example of two code fragments whose coding styles are differed.

basic object-oriented software attribute, and understandable for a lot of developers and managers. We believe that software quality criteria should be simple for becoming widely used, so we focus on class size in this paper.

We use the metric "Statements (Stmts)" [2] for measuring class size. Stmts value is the number of executable and declaration statements in a class. The metric is sensitive to coding effort, and can provide a coding-style independent measurement.

We might used other size metrics such as "lines of codes (LOC) [3]," "number of methods (NM) [4]," "weighted methods per class (WMC) [5]," and so on. LOC is also sensitive to coding effort but performs a coding-style dependent measurement: e.g., two code fragments shown in Fig. 1 have different coding-styles, and their LOC values are also differed (the left fragment's LOC value is 4 and the right one's LOC value is 6) but their Stmts values are the same (both of their Stmts values are 7; these 7 statements contain 1 while-statement, 3 binary-expressions, 1 if-statement, 1 assignment-expression and 1 unary-expression[†]). Although NM would perform a coding-style independent measurement, it is weakly sensitive to coding effort since this metric uses only the number of methods. WMC might also achieve a coding-style independent measurement and be sensitive to coding effort. However WMC values depend on weight values assigned to methods, and there are various ways for assigning weight values [7]. It is difficult to decide the way for assigning weight values to be good at any type of software. For the above reasons, this study will use Stmts as the size metric.

We represent the Stmts value of class c_{v_i} as $Stmts(c_{v_i})$. Stmts values can be easily counted up using JavaML [8] which is a XML representation of Java source program: we can get $Stmts(c_{v_i})$ value by counting the number of appropriate tags, such as "<if>" and "<loop>," in JavaML corresponding to c_{v_i}. For example, Fig. 2 shows a JavaML which corresponds to both of the code fragments shown in Fig. 1; those two code fragments are identical in JavaML. We can find 7 appropriate tags for executable/declaration statements in Fig. 2: "<loop>," three(3) "<binary-expr>"s, "<if>," "<assignment-expr>" and "<unary-expr>." In such way, we can get Stmts values. JavaML bring us an ease of measurement tool development. We developed a Stmts measurement tool based on JavaML.

2.3 Predictive Discrimination

We propose a concept of threshold value for predictively discriminating the costly classes from the others by Stmts values.

```
<loop kind="while">
  <test>
    <binary-expr op="&lt;">
      <var-ref name="i" idref="var-1"/>
      <literal-number kind="integer" value="10"/>
    </binary-expr>
  </test>
  <block>
    <if>
      <test>
        <binary-expr op="==">
          <binary-expr op="%">
            <var-ref name="i" idref="var-1"/>
            <literal-number kind="integer" value="2"/>
          </binary-expr>
          <literal-number kind="integer" value="0"/>
        </binary-expr>
      </test>
      <true-case>
        <assignment-expr op="+=">
          <lvalue>
            <var-set name="x"/>
          </lvalue>
          <var-ref name="i" idref="var-1"/>
        </assignment-expr>
      </true-case>
    </if>
    <unary-expr op="++" post="true">
      <var-ref name="i" idref="var-1"/>
    </unary-expr>
  </block>
</loop>
```

Fig. 2 JavaML corresponding to both of the code fragments in Fig. 1.

Definition 2 (threshold value τ):

Consider a natural number τ, and predict as follows:

- If $Stmts(c_{v_i}) \geq \tau$, c_{v_i} will be a costly class, i.e., the version-upgrade $\{c_{v_i}, c_{v_{i+1}}\}$ will be in the worst α cases;
- otherwise, c_{v_i} will not be costly class.

□

All version-upgrade cases are classified into four categories as the contingency table [9] shown in Table 1.

In Table 1, E_1 and E_2 correspond to the numbers of cases failed in the prediction by τ. In order to normalize impact levels of prediction errors E_1 and E_2, we formulate the sum of error rates as follows:

$$\varepsilon(\tau) = \frac{E_1}{N_1} + \frac{E_2}{N_2} . \quad (1)$$

Now we find τ such that $\varepsilon(\tau)$ has the least value, i.e., the sum of error rates becomes the minimum. Such τ would be a useful criterion of class size for discriminating costly classes in object-oriented software development.

The above notations are summarized in Table 2.

[†]According to the Java language specification [6], these expressions are called "Expression Statements" which are of executable statements.

Table 1 Contingency table for version-upgrade cases.

	Stmts value		
	$< \tau$	$\geq \tau$	total
worst α cases	E_1	S_2	N_1
others	S_1	E_2	N_2

Table 2 Notations.

notation	description
c_{v_i}	version v_i of class c
$d(c_{v_i}, c_{v_{i+1}})$	number of different lines between c_{v_i} and $c_{v_{i+1}}$
α	decision parameter selecting the worst $\alpha(\%)$ cases; costly classes
$Stmts(c_{v_i})$	Stmts value of c_{v_i}
τ	threshold value for predictively discriminating costly classes by Stmts values
$\varepsilon(\tau)$	sum of error rates in the prediction by τ

3. Empirical Study

3.1 Experimental Data Collection

We collected 3514 version-upgrade cases from three open source software projects: "Relaxer," [10] "JBoss" and "JEdit" [11]. Table 3 shows brief descriptions of these software.

For each case (class pair), we counted the number of different lines between the classes, and calculated Stmts value of the older version; formally, for each version-upgrade case $\{c_{v_i}, c_{v_{i+1}}\}$, we computed $d(c_{v_i}, c_{v_{i+1}})$ and $Stmts(c_{v_i})$.

In this empirical study, we assumed $\alpha = 0.05$, i.e., we focused on the worst 5% cases, and predicted their older versions would be the costly classes. Software managers can use any α value as remarked above. Since many statistical tests use "5%" as their significance level, "5%" would be a reasonable boundary for determining some worst cases. Therefore this study used $\alpha = 0.05$ as a reasonable example. In our set of samples, the worst 5% cases corresponded to the class pairs in which 154 or more lines were modified through their version-upgrades.

Notice that the testing activity will also have a large impact on the version-upgrades, though this study focuses on the class size. The higher testing density will decrease faults in the software, and reduce the version-upgrade cost. Unfortunately it may be difficult to obtain the testing data in the open source software development, unlike in the cases of closed projects which are managed by software vendors. In open source software development, we are often able to collect bug reports from the development site instead of testing data. The bug density[†] would also be an important factor for the code quality and the stability of upgrade.

Automatically generated codes and/or reused codes might be included in our empirical data. Since such codes are unlikely to be upgraded through application releases, they would be noise for computing the threshold value. Thus it is better to exclude those codes from the analysis if we can detect those codes[††]; The study of code clone [12] might be helpful in detecting reused codes.

Table 3 Open source software used as our experimental data.

software	description	version
Relaxer	Schema compiler using RELAX as schema, which can generates Java programs for processing XML according to RELAX. Relaxer is written in Java.	0.1 0.101 0.11 0.12 0.13 0.14 0.141 0.142
JBoss	J2EE based application server implemented in 100% Pure Java.	2.2.2 2.4.0 2.4.1 2.4.3 2.4.4 2.4.5
JEdit	A programmer's text editor written in Java.	3.0 3.0.1 3.0.2 3.1 3.2

3.2 Result and Validation

Through the computation for possible τ with $\alpha = 0.05$, we found that $\varepsilon(\tau)$ has the least value when $\tau = 113$ (see Fig. 3). We could obtain a threshold value 113 in Stmts, which might be a criterion of class size for controlling the upgrade cost. Notice that the value of τ is computed using only the version-upgrade cases. However there were many classes which did not have any changes with the next release, and they were excluded from the computation since they were not regarded as upgrade cases. In order to see an effect of τ on not only upgraded classes but also the above excluded classes, we conducted a follow-up. The initial versions of the application software shown in Table 3 had 154 classes whose sizes were exceeded the threshold value. These 154 classes include the classes which did not have any changes with the next release. We investigated whether those 154 classes were changed in later releases. As the result, we found that 118 classes of them (ca. 77%) were upgraded in a later release. Therefore $\tau(= 113)$ would have a certain level of ability in predictive discrimination.

Since the above three software sets are in different application domains, this experimental result does not de-

[†]We could get bug reports on "JBoss" and "JEdit" from the development sites. In consequence, the bug densities (= Bugs / 1000 Stmts) were not much difference between "JBoss" and "JEdit": the bug density of "JBoss" and "JEdit" were 2.35 and 2.56, respectively. Thus the code quality of "JBoss" would be at about the same level as "JEdit." Unfortunately we could not examine "Relaxer" in terms of the bug density, since we could not find any repositories of bug reports on "Relaxer."

[††]In our empirical data collection, it was difficult to find which fragment of the code is an automatically generated or reused code, so we used all classes as our regular data.

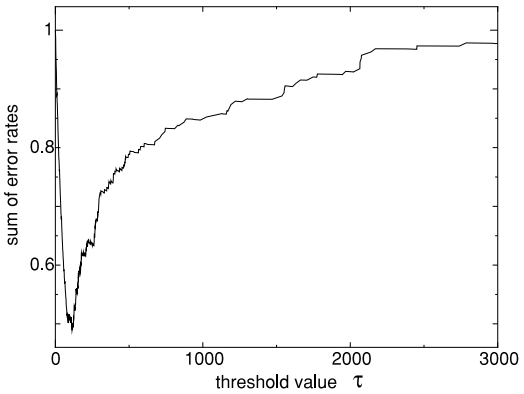

Fig. 3 $\varepsilon(\tau)$ values for possible τ.

pend on specific domains, and would have a generality for discriminating costly classes written in Java. We performed an additional empirical study in order to validate the generality of the above threshold value. We collected another set of Java classes containing 556 version-upgrade cases from two open source software projects "iReport" and "Azureus" [11]†, and computed Stmts value for the older version in each case. As the result, 408 upgrade-cases (ca. 73%) of them could be successfully discriminated by the above threshold value. The threshold value seems to have a certain level of generality for discriminating the costly classes. While this empirical study uses some different applications together in order to see the generality, application specific threshold values can be computed in similar ways. The application specific threshold values are also of interest, and would be more useful in some cases.

4. Conclusion and Future Work

Using a class size metric, Stmts, we have proposed a simple method for predictively detecting costly classes that are likely to need large modifications through their version-upgrades. The method provides a threshold value of class size (in Stmts) for predictively discriminating those costly classes. In our empirical study, we have derived the threshold value 113 (in Stmts) from many Java samples of version-upgrades (ca. 3500 pairs of classes). Further, using another set of Java samples (ca. 500 pairs of classes), we have validated that the derived threshold value has a certain level of generality for predictively discriminating the costly classes: it succeeded as a predictive discriminator for about 73% of our sample classes. The derived threshold value would aid the development of Java classes as a simple criterion of class size. This method is simple but would contribute to controlling software maintenance- and/or testing-efforts.

Our future work includes the followings: (1) enhancement of predictive-accuracy with extended multivariate methods, (2) analysis of application specific threshold values, (3) study of the impact of bug reports (bug density) on the version-upgrades, and (4) investigation of generated/reused code effect on the version-upgrades.

An analysis of source code changes themselves is also our important future work; Fischer et al.[13], [14] have studied relationships among software release data and bug report data, and Ying et al.[15] have proposed a method for predicting source code changes with mining change history. For more precise predictive discrimination of costly classes, our future work would consider the source code changes with their release data and change histories.

†iReport is a visual reporting tool based on JasperReports written in 100% pure java, and Azureus is a powerful, full-featured, cross-platform java BitTorrent client. The sample version-upgrade cases are randomly collected from their CVS repositories; those evaluated classes are the revisions 1.2 and 1.3 of iReport, and the revisions 1.1, 1.2 and 1.3 of Azureus.

Acknowledgments

The authors would like to thank the developers of open source software used in the empirical work. The authors also wish to thank the anonymous reviewer for his/her thoughtful and helpful comments.

This research was partially supported by the Inamori Foundation, and the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 2004, 16700037.

References

[1] K.E. Eman, S. Benlarbi, N. Goel, W. Melo, J. Lounis, and S.N. Rai, "The optimal class size for object-oriented software," IEEE Trans. Softw. Eng., vol.28, no.5, pp.494–509, May 2002.
[2] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," J. Syst. Softw., vol.51, pp.245–273, 2000.
[3] S.D. Conte, V.Y. Shenm, and H.E. Dunsmore, Software Engineering Metrics and Models, Benjamin Cummings Publishing Inc., Calif., 1986.
[4] L.C. Briand, J. Wust, J.W. Daly, and D.V. Porter, " Exploring the relationships between design measures and software quality in object-oriented systems," J. Syst. Softw., vol.51, pp.245–273, 2000.
[5] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design," IEEE Trans. Softw. Eng., vol.20, no.6, pp.476–493, June 1994.
[6] J. Gosling, B. Joy, G. Steele, G. Bracha, and James Java Language Specification Gosling, The Java Language Specification, Second ed., Addison-Wesley, Boston, 2000.
[7] M. Takehara, T. Kamiya, S. Kusumoto, and K. Inoue, "Empirical evaluation of method complexity for C++ program," IEICE Trans. Inf. & Syst., vol.E83-D, no.8, pp.1698–1700, Aug.2000.
[8] G.J. Badros, "JavaML: A markup language for Java source code," Proc. 9th Int'l World Wide Web Conf., 2000.
[9] E.L. Crow, F.A. Davis, and M.W. Maxfield, Statistics Manual, Dover, New York, 1955.
[10] http://www.asahi-net.or.jp/˜dp8t-asm/java/tools/Relaxer/
[11] http://sourceforge.net/
[12] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," IEEE Trans. Softw. Eng., vol.28, no.7, pp.654–670, July 2002.
[13] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," Proc. 10th Working Conf. on Reverse Engineering (WCSE'03), pp.90–99, 2003.
[14] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," Proc.

of the International Conf. on Software Maintenance (ICSM'03), pp.23–33, 2003.

[15] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting source code changes by mining change history," IEEE Trans. Softw. Eng., vol.30, no.9, pp.574–586, Sept. 2004.