A UML Approximation of Three Chidamber-Kemerer Metrics and Their Ability to Predict Faulty Code across Software Projects

Ana Erika CAMARGO CRUZ†‡, Member and Koichiro OCHIMIZU†, Nonmember

SUMMARY Design-complexity metrics, while measured from the code, have shown to be good predictors of fault-prone object-oriented programs. Some of the most often used metrics are the Chidamber and Kemerer metrics (CK). This paper discusses how to make early predictions of fault-prone object-oriented classes, using a UML approximation of three CK metrics. First, we present a simple approach to approximate Weighted Methods per Class (WMC), Response For Class (RFC) and Coupling Between Objects (CBO) CK metrics using UML collaboration diagrams. Then, we study the application of two data normalization techniques. Such study has a twofold purpose: to decrease the error approximation in measuring the mentioned CK metrics from UML diagrams, and to obtain a more similar data distribution of these metrics among software projects so that better prediction results are obtained when using the same prediction model across different software projects. Finally, we construct three prediction models with the source code of a package of an open source software project (Mylyn from Eclipse), and we test them with several other packages and three different small size software projects, using their UML and code metrics for comparison. The results of our empirical study lead us to conclude that the proposed UML RFC and UML CBO metrics can predict fault-proneness of code almost with the same accuracy as their respective code metrics do. The elimination of outliers and the normalization procedure used were of great utility, not only for enabling our UML metrics to predict fault-proneness of code using a code-based prediction model but also for improving the prediction results of our models across different software packages and projects.

key words: UML metrics, CK metrics, fault-proneness of code, logistic regression

1. Introduction

The Chidamber and Kemerer metrics (CK) are design complexity metrics that have traditionally been measured from the code and then related to managerial factors such as productivity, re-work effort for reusing classes and design effort and maintenance effort [1], [2]. In addition, they have been widely used in prediction models to determine how faulty classes will be during the testing phase, showing their ability to predict fault-proneness. [3]–[8] (see Fig. 1-A).

This paper discusses two main topics, first how to approximate a subset of the CK metrics using UML diagrams, which previously has found some difficulty (see Fig. 1-B). And second, how such approximations can be used to detect those classes which would result most faulty after their implementation (see Fig. 1-C).

If the CK metrics are per se good predictors of fault-proneness and our UML approximations are close enough to their code values, then we expect to predict fault-proneness of code during the design time as well as if we were using code CK metrics.

To test the ability of our UML metrics to predict faulty classes, we use Logistic Regression to build three univariate prediction models using the code CK metrics of a large-size-open-source software project. Thereafter, we test the resultant models using on the one hand the code CK metrics of three small-size projects developed by students of JAIST, and on the other hand using their UML CK metrics, which are approximated according to our proposal.

The above approach comes with two challenges. First, we need to ensure that our prediction models work properly across different software projects. Second, because of the difference in obtained values between UML CK metrics and code CK metrics, our code-based prediction models might not work with UML CK metrics. Therefore, we should find an appropriate manner to decrease such a difference as much as possible. To address these two challenges, we propose to perform data normalization.

This paper is divided into eight sections: In Sect. 2, we present a brief background on the CK metrics and their relation with the fault-proneness of code. In Sect. 3, we present some work related to our research. In Sect. 4, we describe how to approximate the CK metrics using UML diagrams, we focus on the analysis of three metrics (RFC, CBO and WMC) and evaluate the proposed UML metrics using the design and implementation of three small-size software projects. In Sect. 5, we compare the results of two data normalization techniques. In Sect. 6, we detail the construction of three code-based prediction models using Univariate Logistic Regression. Then, we test these models in different projects.
projects and packages, using both UML and code CK metrics. In Sect. 7, we provide a set of simple guidelines to apply our models and methodology. In Sect. 8, we discuss the correlation among design-complexity metrics. Finally, in Sect. 9, we draw our conclusion and plans for future work.

2. Background

Nowadays, in the field of software development, the measurement of quality is performed after the software has been implemented. Once its implementation is concluded, the software is tested and defects are detected. Such defects cause, in the majority of the cases, the software to be the subject of further analysis, correction and/or extra development, incrementing time and human resources for repairing these low-quality elements of the software. Therefore, there is a need to detect these defects before they are implemented.

In order to detect and evaluate problems beforehand, and to predict future trends within the elaboration process of a certain product, whether is software, the use of efficient metrics can serve as a baseline to perform such important tasks. In the field of software measurement, we can find several metrics for object-oriented systems. Some of the most used are: Weighted methods per class (WMC), Response for class (RFC), Lack of cohesion in methods (LCOM), Coupling between objects (CBO), Depth of inheritance tree (DIT), and Number of children (NOC). These metrics are known as the Chidamber and Kemerer (CK) metrics.

The CK metrics were defined to measure design complexity, and are traditionally measured from the implementation of the classes. Furthermore, they have been related to managerial factors such as productivity, re-work effort for reusing classes and design effort and maintenance effort [1], [2]. In addition, the CK metrics, when measured from the code, have shown their ability to predict fault-prone code in several empirical studies [3]–[8]. Most of the results of these studies have found that coupling measures, such as RFC, are strongly related to fault-proneness code. The accuracy of their models ranges from 62.9% to 93%.

Although the author of the CK metrics established that they can be obtained in early stages of the life cycle of the software, prior to program execution [9], they cannot be measured accurately unless there exists a very well detailed formal documentation of the design, which is rarely found during the development of software projects. On the other hand, one should remember that a metric should be objective, avoiding inconsistency and preventing individual bias, which means that if different people perform the measurement, they will give similar values [10]. Therefore, written documents, although specified in detail, might be subject of different interpretations when measuring the same metric. Because UML diagrams are part of a formal modeling language of the design of object-oriented software, they seem to be the best source of information, in the framework of elaboration of the software, for measuring the CK metrics accurately, rather than simply written documentation.

3. Related Work

3.1 CK and UML

The few works, that have tried to approximate the CK metrics out of UML diagrams, focus mainly on UML class diagrams, and their observations are in general that the calculation of three of the six metrics from UML class diagrams is not straightforward. These three metrics are the CBO, RFC and LCOM. Moreover, their methods either lack accuracy (in terms of proximity to the measures obtained directly from the source code) or need very well detailed design information.

In [11], Twan van Enckevort tried to verify modifications to UML models using quality metrics, among them some CK metrics measured from UML class diagrams. He indicates that RFC requires behavioral diagrams, LCOM cannot be measured from UML class diagrams and that CBO is difficult to calculate, but it can be estimated by calculating the sum of all class dependencies.

Baroni et al. have established a formal definition of the CK metrics using UML class diagrams [12]. However, CBO, RFC and LCOM CK metrics, measured from UML class diagrams, do not capture all the information, which normally the code provides.

Tang et al. [13] used a combination of UML class, collaboration and activity diagrams to obtain approximations of the CBO, RFC and LCOM metrics. Strong assumptions are made, in order to obtain accurate measures of these metrics, such as that every non-abstraction operation must be represented in one activity diagram and instance variable usages. They claim that these assumptions are not required in order to use their algorithms and obtain different levels of precision of the CK metrics at various stages. Unfortunately, they do not provide any further information on which these various stages are and what the quantitative precision level of their algorithms is.

McQuillan et al. in [14] indicate that RFC can be derived by inspecting various behavioral diagrams, but they do not say how, as for the CBO metric, they indicate that it can be approximated from UML class diagrams, but to obtain more precise measures, behavioral diagrams are needed. Later in [15], a formal definition of the CK metrics using UML 2.0 is presented by the same authors. Their work is an improved version of Baroni’s previous work using UML 1.3.

Because of the importance of coupling measures for fault-proneness detection, we studied an alternative easy approach to measure CBO and RFC metrics using only collaboration diagrams. Further details are given in a later section.

3.2 Early Prediction of Fault Proneness

We found few works that use metrics from design artifacts and predict faulty code before coding. Only one uses a subset of the CK metrics, which were partially obtained from
design documents [16]. Further details of these works are given in the following paragraphs.

In [17], the relationship between design metrics and the number of function test failure reports of software packages is investigated. From this work, a tool called ERIMET was developed to analyze design documents automatically. Then, packages are represented by one or more graphs. Finally, some metrics, which are believed to contribute to the complexity of the code, are calculated. This calculation was possible due to the usage of a formal language named FDL, which is related to SDLs' process diagrams. An example of these metrics is a modified McCabe’s Cyclomatic Complexity metric. Although this method allows to predict before coding, it is unknown to what extent the proposed prediction model can be used by different organizations. Therefore, it is recommend that organizations build their own prediction models, using their own specific data.

In [16], Ramanath et al. provide empirical evidence supporting the role of a subset of the CK metrics in determining software defects. Complexity measures were computed from design documents and source code. While WMC and DIT were computed from both sources, CBO and size were computed only from source code.

In [18], Schroter et al. used relationships between components, which are typically defined in the design phase, to predict fault-proneness of code. They carried out an empirical study of 52 eclipse plug-ins, where all possible import packages in ECLIPSE were used as independent variables of their model. From their experiments, they conclude that prediction on a package level yields better results than prediction on a file/class level. They also found that models trained in one version can be used to predict failure-prone components in later versions. Following the same investigation line, Holz et al. used the same import packages of a component to predict its size and complexity metrics. Since the import relationships are available at design time, the predicted complexity metrics can be used to predict faulty code. Their predictions were reported “outstandingly good” for CBO and CBOJDK. Moreover, SLOC, DIT and RFC were reported as predicted with “reasonable accuracy” [19].

In the same year, Yue Jiang et al. [20] compared the performance of predictive models which use design-level metrics with those which use code-level metrics, and with those which use both. They concluded that models built from code metrics typically outperform design metrics-based models, and that models that utilize a combination of design and code-level metrics outperform models which use either one, or the other metrics set. What is of interest to us is that, by performing reverse engineering, they recovered some design artifacts from code, and they could derive software quality metrics from these artifacts, metrics such as Cyclomatic complexity, number of decisions, etc. (no CK metrics). For our research, reverse engineering is not an option to obtain UML diagrams, and then measure CK metrics from them to predict fault-prone code. This is mainly because our findings suggest that measures taken from the design (at a certain stage) are different from those taken from the code, as it happens with our UML WMC metric. Further details can be found in the following section.

Recently, Nugroho et al. [21] proposed five UML metrics measured from class and sequence diagrams to predict faulty classes. They classify their UML metrics into Level of detail and Coupling metrics. A significant industrial Java system was used to build and test three logistic regression models, one using solely UML metrics, a second one using code metrics, and a third hybrid model using 2 of their UML metrics and KSLOC. They concluded that their UML metrics ‘SDmsg’ and ‘ImpCoupling’, yields a slightly better prediction accuracy than using SLOC metric. Moreover, their hybrid model resulted in the prediction model with the best accuracy, 78%, which is 16% above the accuracy percentage of their UML model. We should indicate that their two UML Coupling metrics taken from UML sequence diagrams are similar, but not equal, to our proposed UML metrics to approximate WMC and CBO. The difference lies in the fact that they are measuring received and sent messages by the objects and to the objects of a given class indistinctly, please refer to Sect. 4.1.2 for comparison.

4. UML Metrics Approximations of the CK Metrics

In this section, we first describe how the CK metrics can be approximated using UML diagrams. Then, we present the evaluation of our proposed UML metrics.

4.1 UML Approximation

In the next following paragraphs, we will restate the definition of each of the CK metrics given by their authors [9], with the simple purpose of analyzing how they can be obtained from UML diagrams.

4.1.1 Measuring from Class Diagrams

- Depth of inheritance tree (DIT). It is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. Deeper trees constitute greater design complexity, but also promote the re-usability of inherited methods. DIT can be obtained straightforwardly from the UML class diagram.

- Number of children (NOC). It is the number of immediate subclasses subordinate to a class in the hierarchy. The greater the number of children, the greater the reuse of the class might be, but the more complex testing and maintainability become. NOC can be obtained also straightforwardly from the UML class diagrams.

††Specification and Description Language: A modeling language used to describe real-time systems.

*Java specific CBO.
4.1.2 Measuring from Collaboration Diagrams

- Weighted methods per class (WMC). It is a count of the methods implemented in a class, or the sum of the complexities of every method of the class. A class with high-weight methods per class is likely to be more specific, limiting its re-usability in other applications. It also complicates its maintenance.

  Using UML class diagrams, we can obtain the number of methods of a class straightforwardly. A first approximation of the WMC can be obtained using collaboration diagrams as well, according to Eq. (1).

  \[ UWMC_i = R_i \]  \hspace{2cm} (1)

  Where \( R_i \) represents a count of all different messages received by all objects of Class \( i \).

- Coupling between objects (CBO). It is the number of other classes to which a class is coupled. If a method within a class uses a method or instance of a variable of a different class, it is said that this pair of classes is coupled.

  The higher the number of couples a class has, the more complex it is to change or to correct. Low coupling between objects improves modularity and promotes encapsulation.

  We proposed to obtain an approximation of this metric using UML collaboration diagrams in [22], according to Eq. (2)

  \[ UCBO_i = S_i - Sr_i \]  \hspace{2cm} (2)

  Where \( S_i \) represents a count of all messages sent to the different objects by all objects of class \( i \), and \( Sr_i \) are those messages sent by all objects of class \( i \) returning a value.

  The precedent equation can be explained through an example. Figure 2-a shows a UML collaboration diagram where the object ‘OperationsFundsServer’ is returning a value of ‘FundsCommited’ to the ‘RequisitionAgent’ object, represented by message R2. If we count the R2 message as part of CBO for ‘OperationsFundsServer’, we would be assuming that ‘OperationsFundsServer’ is instantiating a method of ‘RequisitionAgent’, when it is only returning a value. Therefore, to avoid any misunderstanding, we propose using a different representation of this interchange of messages, as shown in Fig. 2-b.

- Response for class (RFC). It is a set of all methods that can be invoked in response to a message to an object of the class. Please see Fig. 3. From the code, RFC is measured as the number of methods of a given class (M), plus the number of methods of other classes directly called by any of the methods of the given class (R). A method is counted only once in R even, if it is executed by several methods M.

  The higher RFC in a class, the more complex the class is. If a large number of methods can be invoked in response to a message, the testing and debugging becomes more difficult and complex.

  We proposed to obtain an approximation of this metric using UML collaboration diagrams, according to Eq. (3)

  \[ URFC_i = R_i + (S_i - Sr_i) \]  \hspace{2cm} (3)

  Where the first term \( R_i \) stands for the total number of different messages received by all objects of class \( i \), which approximates the number of methods of the class (M). The second term \( (S_i - Sr_i) \) is the total number of messages sent to different objects by all objects of class \( i \) but those which return a value. This term approximates the number of methods of other classes directly called by any of the methods of the class \( i \) (R). This new approach is slightly different from the one we proposed in [22]. This time, we eliminate the messages which return a value, since they do not call any method of the class.

4.1.3 Other

- Lack of cohesion of methods (LCOM). It measures the dissimilarity of methods in a class. High cohesion indicates
a good subdivision class and thus promotes re-usability. A method is more similar to another method if they instantiate the same variables. Lack of Cohesion is not desirable. We concluded that Lack of Cohesion in Methods cannot be approximated using only UML diagrams. It might be approximated using other resources different from UML diagrams, which gives us the details of the behavior of every method, such as the class diagram documentation specifying which attributes of the class are being used in every method, or a flowchart diagram per method of every class in the system, as it was suggested in [13]. To summarize, we have three CK metrics that can be obtained straightforwardly from the UML class diagrams: DIT, NOC and WMC; the last one also can be obtained from collaboration diagrams. Two other metrics whose values can be approximated using collaboration diagrams: CBO and RFC. And one metric, LCOM, which is not possible to obtain accurately either from UML class or collaboration diagrams.

4.2 Evaluation

We present the evaluation of the three UML metrics that can be obtained from collaboration diagrams: WMC, CBO and RFC. In [22], we evaluated two of these metrics using only one small-size software project. In this paper, we present 2 more small-size software projects, which were developed by students of JAIST during a 3-month lecture. These projects were developed by different groups of 2 or 3 students in different years, the three projects being written in JAVA. Every group of students based its implementation on the analysis and basic design of the projects provided in Gomaa’s book [23]:
- An e-commerce system (ECS).
- A banking system (BNS).
- A cruise control and monitoring system (CRS).

In order to evaluate the goodness of our UML approximations of the WMC, CBO and RFC metrics, we first measured them for every class of the projects using the UML collaboration diagrams provided in [23] and following the procedure described in the previous section. Then, we measured the same metrics directly from the source code. Finally, we compared both resultant sets of measures using a simple line graph per project and measured the correlation coefficients between them. The numbers of classes analyzed of the three projects were 10, 11 and 16 from the ECS, BNS and CRS respectively.

To determine whether the correlation coefficients can be regarded as statistically significant, and unlikely to occur by mere chance, we used the conventional 5% level (p-value \( < 0.05 \)) considering a non-directional hypothesis.\(^1\)

Notice that we are capturing design information at a very early stage of the life cycle of the software projects, since we are measuring the UML metrics from the collaboration diagrams given by Gomaa’s book, which was used by the students to elaborate a more complete design to finally implement the systems.

1. RFC

The graph in Fig. 4 shows the raw RFC measures of the ECS, BNS and CRS software projects. The line with squares represents RFC measured from UML diagrams, and the line with triangles represents RFC measured from the code. As we can observe, there is a gap between the UML measures and the code measures, which is larger in the BNS and ECS projects. However, the correlation coefficients of these measures tell us that there is a strong linear relationship between the UML and code measures. The correlation coefficients are 0.7, 0.57 and 0.86 for the BNS, CRS and ECS projects respectively. The obtained p-values are 0.01, 0.02 and 0.0014 for the BNS, CRS and ECS projects respectively, suggesting that the three correlation coefficients are statistically significant.

2. CBO

Likewise, the graph in Fig. 5 shows the raw CBO measures of the ECS, BNS and CRS software projects. As we can observe, the gap between the UML and the code measures is not as large as the one found between RFC measures. Also, the correlation coefficients between the UML and code measures tell us that they have a strong linear relationship. The correlation coefficients are: 0.92, 0.56 and 0.92 for the BNS, CRS, and ECS projects respectively. The obtained p-values are 0.00006, 0.024 and 0.0001 for the BNS, CRS and ECS projects respectively, suggesting that the three correlation coefficients are statistically significant.

3. WMC

The graph in Fig. 6 shows the raw WMC measures of the ECS, BNS and CRS software projects. The line with squares represents WMC measured from UML diagrams, and the line with triangles represents WMC measures.

\(^1\)A non-directional hypothesis considers a non-zero correlation between two variables, either positive or negative.
sured from the code. As we can observe, there is a gap between the UML and code measures, which is larger in the BNS and ECS projects. The correlation coefficients between the UML and code measures of the BNS, CRS and ECS projects are 0.16, 0.51 and -0.17 respectively. The obtained p-values are 0.63, 0.04 and 0.63 for the BNS, CRS and ECS projects respectively, suggesting that only the correlation coefficient between the WMC measures of the CRS project is statistically significant. The numbers of classes used for this analysis are 8 from the BNS, 16 from the CRS and 11 from the ECS.

Because only the measures of CRS show some lineal relationship between code and UML measures, and because of the existent difference of values found between code WMC and UML WMC measures in all classes analyzed, we conclude that at the stage of finishing the collaboration diagrams, the number of methods per class is still far from that developed in the implementation phase.

The average relative errors of our approximations are 0.68, 0.51 and 0.64 for RFC, CBO and WMC respectively. More details can be found in Table 1, where URFC, UCBO and UWMC are our UML approximations.

### Normalization

Because our final target is to use a code-based prediction model with different projects using UML metrics, we need to find a way to decrease the difference of values (error approximation) between UML metrics and code metrics. Furthermore, the challenge is also to find a possible common scale that allows us to use the same prediction model across different projects, regarding the domain of the project or the source of the metric (either UML artifacts or code).

In [24], we proposed the use of simple log data transformations for making distributions of some design complexity metrics (from different projects) more comparable among them. Thus, we can improve the prediction ability of a logistic regression model across different software projects. The results found in that work led us to conclude that log-data transformations can be useful, especially when code measures of the project are not as spread as the code measures used to build the prediction model.

In this occasion, apart from log transformations, we are presenting an additional approach: linear scaling to unit variance.

Due to space limitations, we are presenting only the resultant graphs of applying the above mentioned normalization approaches to the code and UML RFC measures of our 3 projects. We chose to present only the RFC measures because these are the measures which have the largest difference of values between code and UML measures, and

<table>
<thead>
<tr>
<th>Statistic</th>
<th>URFC</th>
<th>URFC&lt;sub&gt;LOG&lt;/sub&gt;</th>
<th>URFC&lt;sub&gt;LS&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.68</td>
<td>0.42</td>
<td>0.21</td>
</tr>
<tr>
<td>DevSt</td>
<td>0.25</td>
<td>0.22</td>
<td>0.15</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0.02</td>
</tr>
<tr>
<td>Max</td>
<td>0.96</td>
<td>0.79</td>
<td>0.72</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>UCBO</th>
<th>UCBO&lt;sub&gt;LOG&lt;/sub&gt;</th>
<th>UCBO&lt;sub&gt;LS&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.51</td>
<td>0.34</td>
<td>0.19</td>
</tr>
<tr>
<td>DevSt</td>
<td>0.4</td>
<td>0.31</td>
<td>0.13</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>1</td>
<td>1</td>
<td>0.38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>UWMC</th>
<th>UWMC&lt;sub&gt;LOG&lt;/sub&gt;</th>
<th>UWMC&lt;sub&gt;LS&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>0.64</td>
<td>0.39</td>
<td>0.36</td>
</tr>
<tr>
<td>DevSt</td>
<td>0.5</td>
<td>0.23</td>
<td>0.42</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>Max</td>
<td>3</td>
<td>1</td>
<td>2.21</td>
</tr>
</tbody>
</table>

To ease the comparison of the raw WMC measures with their respective normalized measures in a later step, we only show those classes resulted from the normalization and elimination of outlying classes procedure, which are different for the RFC and CBO analysis.
RFC has shown continuously to be a good predictor of fault-proneness among other metrics [3]–[7].

5.1 Logarithm Data Transformations

Transformations in general enable us to express data in a new scale that offers several improvements over the original scale of data collection.

When we have several batches of data, and a relationship between spread and level is found, one can reduce or eliminate the dependence of spread on level by using power transformations [25]. In our previous study, mainly CBO measures of the analyzed data showed this tendency [24].

Power transformations are defined as the transformation that replaces $x$ by $x^p$. If $p = 0$ then $\log x$ is used instead.

Some of the extra benefits that these transformations can offer are: easier data comparison, symmetry and normality improvements, and fewer outliers.

Figure 7 shows the RFC measures of the BNS, CRS and ECS projects using the transformation of Eq. (4) (we added 1 in order to avoid $\log(0)$ values).

$$x' = \log(x + 1) \quad (4)$$

Where:
- $x'$ is the new normalized value.
- $x$ is the raw value of the metric.

The line with squares corresponds to the UML measures and the line with triangles corresponds to the code measures. As we can observe, the log transformation failed to reduce the gap between UML and code measures for the BNS and ECS projects and for some of the classes of the CRS project.

The average relative errors of our approximations using log data transformations are: 0.42, 0.32 and 0.39 for RFC, CBO and WMC respectively. In general, we could improve our approximations using simple log transformations. More details can be found in Table 1, where the subscript $LOG$ refers to our results obtained using Log data transformations.

5.2 Linear Scaling to Unit Variance

Linear scaling to unit variance is often used in applications of image processing. According to [26], if we assume that our data is normally distributed, a transformation of the kind of Eq. (5) guarantees that 99% of the data lies in the range $[0,1]$. Values outside this range can be truncated either to 0 or 1.

$$x' = \frac{x - \mu}{3\sigma} + 1 \quad (5)$$

Where:
- $x'$ is the new normalized value,
- $x$ is the raw value of the metric,
- $\mu$ is the mean of the distribution of $x$,
- $\sigma$ the standard deviation of the distribution of $x$.

We applied Linear scaling to unit variance to the BNS, CRS and ECS projects. Figure 8 shows the resultant values of RFC after the transformation. The line with squares corresponds to the UML measures, and the line with triangles corresponds to the code measures. As we can observe, linear scaling to unit variance has successfully decreased the gap between UML and code measures for all of the three projects, reducing the relative errors of our approximations by 47%. The average relative errors for RFC, CBO and WMC are 0.21, 0.19 and 0.36 respectively. More details can be found in Table 1. The subscript $LS$ denotes the results obtained using Linear Scaling.
6. Code-Based Prediction Models Using Logistic Regression and Linear Scaling to Unit Variance

In this section, we provide the details of the construction of three code-based prediction models using Univariate Logistic Regression and linear scaling to unit variance normalization.

We used Logistic Regression mainly because this technique does not require normally distributed variables as other statistical techniques such as linear regression and discriminant analysis. Another reason is that Logistic Regression parameters and the relationships among them are easier to interpret in contrast with other techniques such as Artificial Neural Networks (ANNs), which also have been used for fault-prediction. ANNs are often referred as black boxes because their derived parameters and coefficients cannot be easily analyzed or interpreted.

Independent variables

For the construction of the models, we used 13 packages of the first-released version of the Mylyn software (MYL), which is a task-focused interface for Eclipse, and is an open-source project written in Java by six developers. From every package of the MYL, we obtained three univariate logistic (UL) models using CK-RFC, CK-CBO and CK-WMC as independent variables, which can be obtained from UML diagrams. Since the small-size projects we are using to test our prediction models do not have significant values of DIT and NOC, we did not consider them for this study.

Dependent Variable

Since in Logistic Regression the dependent variable is dichotomous, we chose to classify classes as Most Faulty (MF) and Least Faulty (LF). Classes with the number of post-release faults greater than or equal to the second quartile of the dataset were classified as MF, and as LF otherwise.

6.1 Construction of UL Models

To construct our three UL models using RFC, WMC and CBO independently, we used the following procedure:

1. For every package of the MYL, we created a dataset.
2. We detected and eliminated outlier data points (values equal to or greater than the third quartile of the entire dataset) for every dataset to improve normality.
3. We normalized every dataset using Linear Scaling to Unit Variance.
4. We built 13 UL models and selected the best resultant model.

Table 2 shows the coefficients of the three best UL models found, of the kind of Eq. (6). The $x$ column contains the independent variable of the model, the $A$ column refers to the intercept values and the $B$ column to the regression coefficients of the independent variable. While coefficients

\[
\begin{array}{c|c|c}
\text{Model} & -2\log L & \chi^2 \\
\hline
\text{RFC} & 7.06 | 7.17 & 10.31 |
\chi^2_{0.05} = 10.16 \\
\text{WMC} & 14.68 | 14.93 & 3.56 |
\chi^2_{0.05} = 3.59 \\
\text{CBO} & 33.39 | 33.91 & 8.51 |
\chi^2_{0.05} = 8.5 \\
\end{array}
\]

on the left side of columns $A$ and $B$ are those for the models using data transformations, the coefficients on the right are for the models using the raw data.

\[
P = \frac{1}{1 + e^{-(Ax + B)}}
\]  

6.2 Validation of UL Models

In the following paragraphs, we present the main steps to assess the appropriateness and usefulness of every model [27].

a) Importance of Each Independent Variable

For this, we use the $-2\log L$ statistic, which has a $\chi^2$ (chi square) distribution with 1 degree of freedom ($df$). If we choose a significance level $\alpha = 0.05$ then $-2\log L > \chi^2_{1,0.05} = 3.8$ is desirable. The resultant $-2\log L$ values of our models suggest that the three variables are significantly important for its correspondent model to fit the data, using transformed and raw values. These values are shown in Table 3.

b) Overall Goodness of Fit

To test the goodness of fit, we use the Hosmer-Lemeshow (HL) test. This statistic has a $\chi^2$ distribution. Large values of HL (small values of $P$), and/or less than 4 $df$ indicate that the overall goodness of fit may not be good [28]. Thus, considering $\alpha = 0.05$, we obtain the result that all of our models pass this test. The values of this statistic of our models can be found in Table 3; values on the left side of the column correspond to the models using data transformations, while those in the right to the models using the raw data.

c) Discrimination

We need to measure the ability of our models to discriminate between the MF and the LF classes. For this, we used Hubert’s statistical procedure [29]. The following statistics are required: $e$, which is the expected number of correct classifications due to chance, $O$, which is the number of correct classifications, and the $Z$ statistic, which follows a

$\chi^2$ distribution.

It is defined as the probability of making a decision to REJECT the null hypothesis. If a test of significance gives a $p$-value lower than the $\alpha$-level, the null hypothesis is rejected. The null hypothesis of this test is: The inclusion of the independent variable does not improve the model fit.
normal distribution. If its value is significant at $\alpha = 0.05$, generally suggests that the number of correct classifications is significantly greater than that obtained by chance.

For all our models the resultant values of $O$ are greater than those of $e$, which indicates a good discrimination between the MF and the LF classes. Also, the resultant $Z$ values suggest that the number of correct classifications is significantly greater than that obtained by chance. The values of these statistics can be found in Table 4; values on the left side of the column correspond to the models using data transformations, while those in the right to the models using the raw data.

For the three models, the cutoff probability to discriminate between the MF and the LF classes is 0.5.

d) Validation Using Different Data

Finally, we present the most important test of this study. We test our models with different data from different packages of the same MYL system and from the 3 other small-size software projects used in Sect. 3.1, using code and UML metrics. Such a test has two purposes: Firstly, to determine if the elimination of outliers and normalization procedure described in 5.1 improve the results obtained using the same prediction model with other datasets (packages and projects); secondly, to determine if this same procedure can be used with UML metrics, so that we can predict before coding.

For the sake of simplicity, we introduce three indicators of a good classification:

- Correctness: Number of classes (LF and MF) correctly classified.
- Specificity: Number of LF classes correctly classified.
- Sensitivity: Number of MF classes correctly classified.

Table 5 shows the percentages of Correctness, Specificity and Sensitivity obtained with our models in other packages and projects using only code measures. NRFC, NCBO and NWMC refer to the results obtained with the models using normalized values, while RFC, CBO and WMC are the percentages obtained with the models using the raw data.

Furthermore, in Table 6, we show the results obtained with our NRFC, NCBO and NWMC models in the projects: ECS, CRS and BNS, using not only their code metrics, but also their UML metrics.

Effect of Normalization on Code Measures

To summarize, comparing the 16 datasets (packages

---

**Table 4** Discrimination test.

<table>
<thead>
<tr>
<th>Group</th>
<th>CBO</th>
<th>RFC</th>
<th>WMC</th>
<th>O</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>MF</td>
<td>16.33</td>
<td>3</td>
<td>8.83</td>
<td>23</td>
<td>2.56</td>
</tr>
<tr>
<td>RFC</td>
<td>1</td>
<td>3</td>
<td>11</td>
<td>12</td>
<td>1.33</td>
</tr>
<tr>
<td>WMC</td>
<td>5.83</td>
<td>9</td>
<td>10</td>
<td>1.77</td>
<td>2.33</td>
</tr>
<tr>
<td>LF</td>
<td>24.67</td>
<td>42</td>
<td>14</td>
<td>15</td>
<td>2.07</td>
</tr>
<tr>
<td>Both</td>
<td>14.66</td>
<td>22</td>
<td>23</td>
<td>2.73</td>
<td>3.1</td>
</tr>
</tbody>
</table>

---

**Table 5** Normalized vs raw code measures.

<table>
<thead>
<tr>
<th>Project</th>
<th>Correctness</th>
<th>Specification</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFC</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>NRFC</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>CSR</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>BNS</td>
<td>87%</td>
<td>91%</td>
<td>100%</td>
</tr>
<tr>
<td>MYL-P01</td>
<td>67%</td>
<td>59%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P02</td>
<td>63%</td>
<td>54%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P03</td>
<td>57%</td>
<td>71%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P04</td>
<td>51%</td>
<td>48%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P05</td>
<td>51%</td>
<td>50%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P06</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P07</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P08</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P09</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P10</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P11</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P12</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>MYL-P13</td>
<td>75%</td>
<td>75%</td>
<td>80%</td>
</tr>
<tr>
<td>CBO</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>NCBO</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
<tr>
<td>NWMC</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
</tr>
</tbody>
</table>

---

**Table 6** Normalized code measures vs UML measures.

<table>
<thead>
<tr>
<th>Model</th>
<th>Project</th>
<th>Correctness</th>
<th>Specification</th>
<th>Sensitivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>Code</td>
<td>Code</td>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>ECS</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>CRS</td>
<td>57%</td>
<td>64%</td>
<td>80%</td>
<td>0%</td>
</tr>
<tr>
<td>BNS</td>
<td>33%</td>
<td>67%</td>
<td>50%</td>
<td>0%</td>
</tr>
<tr>
<td>NRFC</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>NCBO</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
<td>67%</td>
</tr>
<tr>
<td>NWMC</td>
<td>80%</td>
<td>80%</td>
<td>100%</td>
<td>67%</td>
</tr>
</tbody>
</table>

---
and projects) using only code measures, we can say that the normalization procedure greatly improves the results of the RFC model, it benefits less to the CBO model, and almost no improvement is obtained with the WMC model, except for the fact that the NWMC model can discriminate MF classes better than the WMC model. Note that, in most of the cases presented here, the WMC model yields 0% of Sensitivity and high percentages of Specificity, which means that the WMC model is classifying most or all classes as LF.

**Normalized UML VS Normalized Code Measures**

In general, the normalized UML RFC measures yield equal, and in some cases better results, than the normalized code RFC measures. The normalized UML CBO measures yield nearly equal results to those obtained with the normalized code CBO measures. Some of the percentages are better, some others equal and some others worse. The normalized UML WMC measures did not perform better than the code WMC measures.

7. **Guidelines for Further Use**

In this Section, we provide a set of simple guidelines to apply the models and methodology exposed in this paper, see Fig. 9:

1. Define the scope of your datasets. The size of a dataset can be either of a package or of a whole project. The reader must have in mind that the models will detect the MF classes within the chosen dataset.
2. Measure the UML metrics from the data of interest as detailed in Sect. 4.1.2.
3. Form as many datasets as needed. If you decided to work with packages, group your UML measures in as many datasets as number of packages your project has.
4. For every dataset eliminate outlier data points. Since our UML metrics are approximation of the CK metrics and outlier data points of these metrics are, by themselves, focus of attention indicating high complexity and/or possible design violations, such data points should be subjected to further analysis [1].
5. Normalize the datasets according to Sect. 5.2.
6. Using the resultant normalized datasets, apply the normalized models proposed in Sect. 6.1.
7. The models will yield the probability of having a MF class within the chosen dataset.

8. **Discussion**

Is it worth investing efforts to find which other design-complexity metrics, obtained from UML artifacts, can be included in our prediction models? For years, research has shown that a single metric such as CBO and RFC can yield prediction accuracy of sometimes nearly 80%–90%. In order to increase this accuracy percentage, we could integrate other variables in to our model. However, several studies have found a high correlation among design-complexity metrics, not only among the CK metrics [1], [30], but also among other sets of metrics, which are also related to design complexity, such as the QMOOD and the MOOD metrics [5], [6]. If we wish to use multiple regression analysis, either linear or logistic, to describe fault-proneness in terms of more than one metric, uncorrelated metrics are required. A common solution for this problem is to use Principal Component Analysis, which is a technique that transforms a set of correlated variables to a new set of uncorrelated variables, named principal components [31]. Unfortunately, when using these new variables, the resultant new coefficients in the regression model are difficult to interpret, since the dependent variables are not longer describing purely the initial metrics, but a combination of these.

Furthermore, let us say that we wish to investigate if simple UML metrics (not an approximation of those traditionally measured from the code) have some relationship to fault-proneness, we would face the same correlation problem. Previous studies found that UML class metrics are also highly correlated [32], [33]. These metrics have been associated with the maintenance and understandability of UML class diagrams [33]–[35].

A second question to be answered is: how much better does the inclusion of a correlated design-complexity metric, in a prediction model, perform? For example, Olague, in [5], found a good model to predict faulty code across different versions of a software. Such a model was a combination of: CBO, DIT, LCOM, NOC and WMC-McCabe metrics, which were highly correlated. The reported results were that the only significant regressor in the model was WMC, which made the authors wonder whether they should simply have used the univariate WMC model.

Trying to find a prediction model that explains fault-proneness of code using only design-complexity metrics assumes that fault-proneness of code is generated only due to complexity in the design. Therefore, there is a need to identify other metrics, representative of other contributors of faulty code, such as the developer’s experience, team com-

---

Fig. 9 Flow chart for guidelines.
munication, etc. We do agree with Fenton’s criticism [36], since the point of view that fault-proneness of code cannot be only due to the complexity of the design, but also to other factors. All research work done for years has helped us to identify design complexity as one of the major contributors of fault-prone code. Moreover, very good predictors have been clearly detected from their and our own results. However, they are not good enough yet, at least, to use the same model across software projects. Therefore, we are encouraged to study other factors, which might be related to fault-proneness of code.

9. Conclusion and Future Work

Design-complexity metrics, such as the CK metrics, have shown themselves to be good predictors of faulty object-oriented classes when measured from the code. In order to predict before coding, we proposed a simple approach to approximate CBO, RFC and WMC CK metrics using UML collaboration diagrams. Moreover, we carried out an empirical study to evaluate the fault-proneness prediction ability of our UML metrics.

Our results lead us to conclude that the proposed UML RFC and CBO metrics can predict faulty code almost with the same accuracy as their respective code metrics. The elimination of outliers and the normalization procedure used were significantly efficient, not just for enabling our UML metrics to predict faulty code using a code-based prediction model, but also for improving the prediction results of our models across different packages and projects, which was found to be difficult, even across packages of the same project. On the other hand, both the UML WMC and the code WMC metrics showed a poor fault-proneness prediction ability.

We expect that a UML-based prediction model will improve upon the results found in this study. Our univariate logistic regression analysis yields correct classifications of about 80% when using UML metrics. Yet, such rate is not sufficient, if we consider that it can vary from project to project or from package to package. Including other metrics in the models might improve these results, therefore we need to look for some other metrics that are easily obtainable before the implementation of the system, with low or zero correlation among them, and that are strongly related to the fault-proneness of code.

Moreover, other preprocessing techniques for datasets, such as normalization and over/under sampling, can also be helpful. The latter technique is a preprocessing procedure for balancing datasets with a large difference between the number of faulty and non-faulty classes, which has been found to cause performance degradation of fault-proneness models [37]. As we can observe in Tables 5 and 6, the percentages of specificity and sensitivity, and in consequence correctness, generally were improved using normalization, nonetheless specificity and sensitivity remain unbalanced in some cases. We observed that such percentages vary according to the chosen cutoff probability for to discriminate between MF and LF classes, and that a chosen cutoff probability for a specific dataset (package or project) does not always obtain the same results in others. Likewise, the chosen threshold of number of faults to classify classes as MF or LF also affects such percentages. In our experiments, apart of using the number of faults greater than or equal to the second quartile of a dataset as classification threshold, we also used the third quartile, however the former always yields better results when predicting across datasets with the same model. Such phenomenon is another specific topic of further investigation, and perhaps the work previously mentioned on over/under sampling techniques might be a good starting point of research.

Furthermore, other suitable techniques for the construction of prediction models should be investigated. Recalling the main reason we chose Logistic Regression is because this technique, different to other statistical techniques, does not require normally distributed variables, and the resultant models are easy to interpret and to understand. Therefore, we are looking for prediction techniques with similar characteristics and, if possible (considering our previous discussion), that address the problem of correlation among independent variables efficiently. Some statistical data mining techniques have been recently adopted by fault-proneness prediction researchers, such as bayesian networks and decision trees, however we still do not know the extent of usefulness of these techniques for our specific purposes.

Finally, a larger number of empirical studies are needed, mainly using data from the industry, to ensure that the resultant prediction model is robust to various real-life factors, possible contributors of fault-prone code, such as team members’ stress and experience.

Our intentions are, first, to encourage practitioners of fault-proneness prediction to research more on data normalization techniques, in order to make the distribution of metrics used in prediction models more similar to each other. Thus, we can have major possibilities of predicting well across software projects using the same model; second, to direct others’ researchers attention towards our UML metrics as early predictors of fault-prone code; and finally, to attract the attention of those who have access to real-life software projects to consider the results presented in this paper, and to study other factors, different from design complexity, which might contribute to the production of faulty code.

References


Ana Erika Camargo Cruz was born in 1976. She graduated from the Faculty of Engineering of The National Autonoumous University of Mexico in 2000. Later, she obtained a Master degree from the School of Information Science of the Japan Institute of Science and Technology in 2008. Research interests are on early detection of fault-prone object-oriented classes.
Koichiro Ochimizu was born in 1946. He graduated from the Faculty of Engineering Science of Osaka University in 1969. He obtained a Ph. of Eng. degree from the Graduate School of Engineering Science in 1974. He was Lecturer (1974), Assoc. Professor (1975), Professor (1988) of the Dep. of Comp. Sci. at Shizuoka University, Professor of JAIST (1992). His specialties are SW Process Models, OO SW Development Methodologies, Distributed Coop. SW Development Support Env., Test Env. for Embedded Sys., SW Accountability, and PM. ICSE Member of PC (1987–1988, 1992–1993), JSSST Member of the board of trustees (1990–1992). Member of IEEE, IPSJ, JSSST and JSISE.