Chapter 7

UML Metrics for Predicting Fault-prone Java Classes

Identifying and fixing software problems before the implementation is believed to be much cheaper than during or after the implementation. Hence, it follows that predicting fault-proneness of software modules based on early software artifacts like software designs is beneficial as it allows software engineers to perform early predictions to anticipate and avoid faults early. Taking this motivation into consideration, in this chapter we evaluate the usefulness of UML design metrics to predict fault-proneness of Java classes. We use historical data of a significant industrial Java system to build and validate a UML-based prediction model. Based on the case study we have found that level of detail of messages and import coupling—both measured from sequence diagrams, are significant predictors of class fault-proneness. We also learn that the prediction model built exclusively using the UML design metrics demonstrates a better accuracy than the one built exclusively using code metrics.

7.1 Introduction

Identifying and fixing problems in a software system can be expensive activities if performed after its deployment. At this stage of the software life cycle, performing such activities is costly because software engineers face higher risks and complexity imposed by the established interdependencies amongst software components. As such, a small change to a subsystem might lead to repercussions for larger parts of the system. Considering this issue, much work (e.g., [10, 28, 57]) has been devoted to investigate techniques and methods that

This chapter is adapted from the paper entitled "Assessing UML Design Metrics for Predicting Fault-prone Classes in a Java System", published in the proceedings of the 7th International Working Conference on Mining Software Repositories (MSR) 2010.

enable early predictions so that faults in software can be detected and fixed earlier during software development.

Design metrics are measures of static structure of a system that can be calculated from source code or software designs. Many works in the area of early prediction of software quality used design metrics measured from source code (also known as code metrics). As a result, the majority of works that reported the usefulness of design metrics for predicting class fault-proneness were actually based on metrics data that was collected from source code. There are at least two major reasons why measuring design metrics from code is preferable. First, software designs are generally difficult to obtain. Second, even if such designs exist, their level of analysis is often too high to allow the calculation of metrics being investigated. However, in many cases design metrics that are measured from code and are found to be good predictors are not collectible from software designs. In such cases, we need to come up with new metrics or new ways of calculating existing metrics for software designs.

Considering the above issue, in this chapter we define several design metrics that can be collected from software designs and use them to construct prediction models of class fault-proneness. The metrics are extracted from class diagrams and sequence diagrams of a UML model. Using data from a significant industrial case study, we build prediction models based on UML design metrics only, code metrics only, and a combination of both UML design metrics and code metrics. We subsequently compare their performance in terms of prediction accuracy and cost-effectiveness. In essence, we aim to answer the following questions:

1. Are UML design metrics significant predictors of class fault-proneness?

2. What is the accuracy of the prediction model built using UML design metrics (compared to those built using code metrics and a combination of both UML design and code metrics)?

3. What is the cost-effectiveness of the prediction model built using UML design metrics compared to the other models?

7.2 Related Work

Much research has been done in the area of software fault prediction. Generally, research in this area focuses on investigating good predictors that can be obtained early in the development phase, the accuracy of prediction models, and the cost-effectiveness of using certain prediction models [89]. Additionally, some studies investigate the performance of prediction models built using various methods, which include multiple linear regression, logistic regression, principal component analysis, discriminant analysis, and artificial neural networks [70, 72, 103]. Because this chapter focuses on early metrics for fault predictions, in this section we restrict the discussions on prediction studies that use design metrics as predictors.

According to a recent survey reported in [32], most fault prediction studies that used object-oriented systems have been employing object-oriented metrics defined by Chidamber and Kemerer (CK metrics) [35] as predictors. For instance, an early study that used CK metrics as quality indicator is the work of Basili et al. [15]. The authors validated the usefulness of CK metrics to predict class fault proneness in eight identical information systems developed by students. The authors found that five out of six CK metrics (i.e., DIT, RFC, NOC, CBO, and WMC) were useful predictors of class fault-proneness. Furthermore, it was suggested that the CK metrics were better predictors than the code metrics used in the study. The study of El Emam et al. reported in [44] used two CK metrics, namely DIT and NOC, and coupling metrics defined in [22]. Based on a single case study, they concluded that DIT and EC (export coupling) metrics are significant predictors of fault-proneness, and the prediction model built using these two metrics also demonstrated a good accuracy. Briand et al. investigated the usefulness of fault-proneness prediction models to be used across two different projects [24]. The authors employed three sets of object-oriented metrics: coupling, polymorphism, and a subset of CK metrics to build the prediction model. One of the findings discussed in the paper was that a prediction model constructed for a particular system might be useful to predict class fault-proneness in different systems with similar development environments. Other studies that used CK metrics as predictors of class fault-proneness include [57, 68]. These studies confirmed, in particular, about the significance of coupling metrics as predictor of fault proneness. The work of Zhou and Leung assessed CK metrics as predictors of fault proneness across fault severity [134]. The authors found that CBO, WMC, RFC, and LCOM metrics are statistically significant predictors of high and low severity faults. Note that although the aforementioned works used object-oriented design metrics as predictors of fault proneness, the metrics were extracted from source code.

There exist a very few studies that use design metrics collected from design artifacts for class fault-proneness prediction. Early studies in the late 1990's (e.g., [102, 103, 133]) extracted design metrics from FDL (formal description language) graphs to construct prediction models. A study by Jiang et al. [69] compared prediction models built using code and design metrics based on NASA MDP (Metrics Data Program) data set. The authors collected metrics from both FDL graphs and UML diagrams and used them as predictors. The results show that the prediction model built using the combination of design and code metrics has the best performance. Moreover, the authors found that the prediction model built using design metrics demonstrated the lowest performance compared to the ones built using code metrics and combined metrics. This result is somewhat different compared to that of Zhao et al. [133]. Zhao et al. collected design metrics from FDL graphs of a real time system, and compared the performance of prediction models built using design and code metrics for predicting the number of faults. The results suggest that the prediction model built using design metrics is as good as the model built using code metrics. The authors also found that there was no significant improvement when both design and code metrics were combined together as predictors.

While there exists only a few studies that used design metrics collected from design artifacts in the area of fault prediction, many works in the area of program comprehension and software maintenance have been exploring the usefulness of design metrics as explanatory variables. The work of Genero et al., for example, experimentally investigated UML metrics that can be used to predict maintainability of class diagrams [54]. The authors found

that the number of associations and the maximum DIT of a class, which indicate structural complexity, are related to the understandability and modifiability of class diagrams. Our previous study also investigated level of detail (LoD) metrics in class- and sequence diagrams of UML models and their relation to defect density in the implementation [98]. Based on an industrial case study, we have found that level of detail in sequence diagrams significantly correlates with defect density of classes in the implementation.

Considering the importance of early predictions of class fault-proneness and the fact that very little work has been done to investigate the usefulness of design metrics for early fault prediction, we think much research is needed to dismantle and explain the relationships between early design metrics and fault-proneness. In this chapter, we explore the usefulness of UML design metrics that can be collected early in the design phase for predicting class fault-proneness.

7.3 UML Design Metrics

The goal of this chapter is to assess the usefulness of LoD metrics (previously defined in Chapter 6) for predicting class fault-proneness. LoD was proposed as a measure because other measures (e.g., the number of attributes or operations in a class) generally have low values and little variation when applied to industrial UML models that we have obtained. LoD metrics offer an alternative to measure UML model quality by measuring the extent to which UML model elements are specified in UML models (e.g., the proportion of class attributes with signature).

The UML design metrics used in this study measure two aspects, namely level of detail (LoD) and coupling. LoD is defined as the amount of information that is used to specify software models. Our study discussed in Chapter 5 has shown that applying higher LoD in UML models improves model comprehension, which in turn might reduce defects in the implementation (refer to Chapter 4 and 6).

Except for SD_{obj} , all of the LoD metrics defined in Chapter 6 are used as part of the predictors—SD_{obj} was not used because it has zero variance. Note that in this chapter the sequence diagram metrics are measured at object level (in Chapter 6 they were measured at diagram level). The reason of doing so is that the coupling metrics needed to be measured at object level. To keep the measurement approach similar, we measured the sequence diagram LoD metrics also at object level. Below we recapitulate the definitions of the LoD metrics that are used as predictors.

- CD_{aop} is an aggregate metric that measures the level of detail of attributes and operations of classes modeled in class diagrams. As such, for an implementation class x, and corresponding design class x' we define:

$$CD_{aop}(x) = CD_{attrsig}(x') + CD_{opspar}(x') \tag{7.1}$$

  - $CD_{attrsig}$ measures the ratio of the number of attributes with signature to the total number of attributes of a class.

  - CD_{opspar} measures the ratio of the number of operations with parameters to the total number of operations of a class.

- CD_{asc} is an aggregate metric that measures the level of detail of associations of classes modeled in class diagrams. CD_{asc} is defined as follows:

$$CD_{asc}(x) = CD_{asclabel}(x') + CD_{ascrole}(x') \tag{7.2}$$

  - $CD_{asclabel}$ measures the ratio of the number of associations with a label (i.e., association name) to the total number of associations attached to a class.
  - $CD_{ascrole}$ measures the ratio of the number of associations with role name to the total number of associations attached to a class.

- SD_{msg} is an aggregate metric that measures the level of detail of messages of *class instances (objects)* modeled in sequence diagrams. For an object i we define its message LoD score as follows:

$$MsgLoD = SD_{ops(i)} + SD_{ret(i)} + SD_{par(i)} \tag{7.3}$$

  For a given object that appear in a sequence diagram:

  - SD_{ops} measures the ratio of the number of messages that correspond to class methods specified in class diagrams to the total number of messages of the object.
  - SD_{ret} measures the ratio of the number of return messages with label (any text attached to the return messages) to the total number of return messages of the object.
  - SD_{par} measures the ratio of the number of messages with parameters to the total number of messages of the object.

  Because an object may appear in multiple sequence diagrams, SD_{msg} of a class is defined by taking account all sequence diagrams in which that particular object appears. Furthermore, to normalize the effect of the number of occurrences of an object across multiple sequence diagrams on its message LoD score, the score of a given class is defined as the average of the $MsgLoD$ values of the sequence diagrams in which it appears.

  For an implementation class x, a corresponding design class x' and n sequence diagrams in which x' appears, we define message detailedness (SD_{msg}) as follows:

$$SD_{msg}(x) = \frac{1}{n} \sum_{x' \in n}^{n} MsgLoD(x') \tag{7.4}$$

Notice that in the definitions of SD_{ops} and SD_{par} metrics, we do not make a distinction between incoming and outgoing messages. Furthermore, the LoD metrics are measured in ratio. Calculating the metrics in ratio allows us to perform a fair comparison of those metrics across UML models.

In addition to the aforementioned LoD metrics, we use two coupling metrics that are also measured from sequence diagrams. Similar to the LoD metrics, the coupling metrics are also calculated at object level. However, the coupling metrics are not measured in ratio.

- *ExCoupling* measures the total number of *incoming method invocations* to a particular object modeled in sequence diagrams. For a given implementation class x, a corresponding design class x' and n sequence diagrams in which x' appears :

$$ExCoupling(x) = \sum_{x' \in n} MsgIn(x') \tag{7.5}$$

  $MsgIn$ is an incoming method call to a given object.

- *ImpCoupling* measures the total number of *outgoing method invocations* from a particular object modeled in sequence diagrams.

$$ImpCoupling(x) = \sum_{x' \in n} MsgOut(x') \tag{7.6}$$

  $MsgOut$ is an outgoing method call from a given object.

7.4 Design of Study

In this section, we discuss the design of this study, which includes measured variables, data collection, and the analysis procedure.

7.4.1 Measured Variables

The outcome variable is class fault-proneness, which is defined as the likelihood that an implementation class contains at least one defect. We choose fault-proneness over other measures such as fault-count because the variability of fault-count in our data set is low. Using a dependent variable with low variability will affect our ability to identify significant contributions of the predictors.

The predictors in this study are UML design metrics defined in Section 7.3, namely LoD metrics and coupling metrics. In addition to the UML design metrics, we incorporate several metrics that are well-known for their relations with fault-proneness, namely coupling between objects (CBO) [35], McCabe's complexity (MCC) [84], and lines of code (measured in KSLoC). These metrics are measured from source code.

7.4.2 Case Study and Data Collection

The case study used in this chapter is similar to that of Chapter 4. Please refer to Chapter 4 for detailed descriptions of the case study.

Data used in the analysis is obtained from implementation classes that are modeled in both class- and sequence diagrams. Furthermore, the implementation classes will be categorized as *faulty* or *not faulty* depending on whether or not they contain defects. We define faulty classes as classes that contain one or more defects. To determine classes that are faulty, we obtain registered defects from the ClearQuest repository. Subsequently, we obtain change sets (files modified to solve defects) by using a Perl script that recovers change sets associated with every defect automatically. Java classes that are modified *at least* once will be considered as faulty, otherwise not faulty.

To calculate the UML design metrics, we first export the UML files to XMI. Having done that, we use SDMetrics [4] to calculate the metrics from the XMI files. However, because Rational XDE does not export sequence diagrams to XMI, we need to calculate metrics from sequence diagrams manually (i.e., through visual inspection)—in total, there are 341 sequence diagrams. Additionally, there are 34 class diagrams, and we used SDMetrics to automatically calculate metrics of classes modeled in class diagrams. To calculate code metrics from the implementation classes, we use an open source tool called CCCC (C and C++ Code Counter).

The core part of the system (excluding frameworks classes), contains 878 Java classes, of which 85 classes are modeled in both class- and sequence- diagrams. Eighty of these classes pertain to the logic and data layers, and the rest belong to the presentation layer. Furthermore, of the 85 Java classes, 56 are found to be faulty and 29 are not faulty. To obtain balance groups of faulty and non-faulty classes, we randomly select 29 out of the 56 faulty classes. This results in two groups (faulty and not faulty) of 29 Java classes for building the prediction models.

7.4.3 Analysis Method

We build the prediction model using the logistic regression method [64]. Logistic regression is a standard technique for a binary prediction (dichotomous outcome variable) that is based on maximum likelihood estimation. Prior to the analysis, all predictors are transformed using *log transformation*. Log transformation fixes left-skewed distributions by shifting low values closer to the centre of data distribution. In the analysis, we first perform univariate analyses, in which we assess the predictive capability of the individual predictors to predict class fault-proneness.

In the second step of the analysis, we perform multivariate analyses. In this step we construct three prediction models: a model built using UML design metrics only, a model built using code metrics only, and a model built using a combination of UML design metrics and code metrics. We use a stepwise method (the backward elimination method) to select the best set of predictors. The backward elimination method starts with all predictors included, and iteratively removes predictors that have least impact on the predictive capability of the model. Backward elimination is preferred to forward selection, which starts with no predictor and iteratively incorporate a predictor that gives the most significant improvement to the model, because the forward selection method is more likely to exclude a predictor that has a significant effect but only when another variable is held constant (also known as suppressor effects) [49].

Table 7.1: Results of univariate analysis

Variable	**Odds-ratio**	p
CD_{aop}	0.099	0.236
CD_{asc}	1.943	0.738
SD_{msg}	0.197	0.278
ExCoupling	1.215	0.753
ImpCoupling	4.523	**0.027**
MCC	1.340	0.410
CBO	25.214	**0.047**
KSLOC	10562.039	**0.034**

The accuracy of the prediction models will be assessed using several criteria of goodness of fit. We also perform leave-one-out cross validations in order to get a more realistic assessment of the accuracy of the prediction models. Additionally, we perform cost-efficiency assessments to understand the efficiency of each model when used in real projects. This analysis can tell us whether a model delivers a reasonable result if verification effort is taken into account.

To perform the statistical analyses we use the SPSS statistical package [2]. Additionally, we use Weka [59] to perform the cross validation.

7.5 Results

In this section, we provide the results of the analysis. We shall start by discussing the univariate analysis. Subsequently, we present the results of the multivariate analysis and discuss the assessments of the multivariate models from the perspective of accuracy and cost-effectiveness.

7.5.1 Univariate Analysis

In the univariate analysis, we assess the predictive capability of each predictor to predict class fault-proneness. Table 7.1 shows the results of the univariate analysis. We can see in the table that except for ImpCoupling, CBO, and KSLOC, none of the predictors are significantly correlated with class fault-proneness—true significance is considered if $p \leq 0.05$.

Also important to the interpretation is the odds-ratio, which represents the change in odds resulting from a unit change in the predictor. An odds-ratio greater than 1 indicates that the odds of a class being faulty increase as the value of the predictor increases. In the contrary, an odds-ratio less than 1 indicates that the odds of a class being faulty decrease as the value of the predictor increases. We can see in Table 7.1 that the odds-ratios of ImpCoupling, CBO, and KSLOC metrics are all greater than 1. These results suggest that as the values of ImpCoupling, CBO, or KSLOC of a class increase, so do the odds of that class for being faulty.

Table 7.2: Results of multivariate analysis

Model	Variable	Odds-ratio	p
Model-U	SD_{msg}	0.042	0.057
	ImpCoupling	6.753	0.010
Model-C	KSLOC	10562.039	0.034
Model-H	SD_{msg}	0.003	0.009
	ImpCoupling	9.425	0.018
	KSLOC	1.869E7	0.015

7.5.2 Multivariate Analysis

In this section we discuss the results of the multivariate analyses. First, we analyze the model that is built using UML metrics only. Following that, we assess prediction models that are built using the code metrics and the combination of UML metrics and code metrics. Table 7.2 shows the result of the analysis.

The first prediction model is built using UML metrics only (**Model-U**). As shown in Table 7.2, of the five UML metrics, the backward elimination method selects two metrics as significant predictors of class fault-proneness, namely SD_{msg} and ImpCoupling (the p-values are 0.057 and 0.010 respectively). Although the p-value of SD_{msg} is slightly higher than the significance threshold, the backward elimination method still qualifies SD_{msg} as a significant predictor. Having considered that the p-value of SD_{msg} is only slightly higher than the significance threshold, we decided to keep SD_{msg} as a predictor in the model. Furthermore, looking at the odd-ratio of SD_{msg} we learn that the higher the value of SD_{msg} of a class, the lower its odds to contain faults. For ImpCoupling, on the other hand, the higher the ImpCoupling of a class, the higher the odds of that class to contain faults.

The second prediction model is built solely using code metrics (**Model-C**), namely MCC, CBO, and KSLOC. However, the backward elimination method selects only KSLOC as a significant predictor. As shown in Table 7.1, KSLOC has a p-value of 0.034 and an odds-ratio that is greater than 1. This odd-ratio tells us that the bigger the size of a class, the higher the odds of that class to contain faults.

The third prediction model is a hybrid model (**Model-H**), in which we use both UML metrics and code metrics to construct the prediction model. Having all the metrics assessed, the backward elimination method finally selects three significant predictors, namely SD_{msg} , ImpCoupling, and KSLOC. The result shows that in the hybrid model, SD_{msg} has the highest significance level ($p \leq 0.01$). This is particularly interesting if we consider the fact that SD_{msg} has a somewhat lower significance (p-value) than ImpCoupling in the model built using UML metrics only. Additionally, if we look at the odd-ratios of SD_{msg} , ImpCoupling, and KSLOC, which represent the relations of the respective metrics to fault-proneness, they remain consistent (i.e., SD_{msg} is smaller- and ImpCoupling and KSLOC are greater than 1 respectively) in the first and second prediction models.

From the results of the multivariate analysis we learn the following:

Table 7.3: The Confusion Matrix

Observed	Predicted: Negative	Predicted: Positive
Negative	True Negative (TN)	False Positive (FP)
Positive	False Negative (FN)	True Positive (TP)

Table 7.4: Classification Accuracy of the Prediction Models (cross-validated using LOOCV)

Observed faulty		Predicted Faulty: No	Predicted Faulty: Yes
Model-U	No	18	11
	Yes	11	18
Model-C	No	18	11
	Yes	15	14
Model-H	No	23	6
	Yes	10	19

- SD_{msg} metric, which is an indicator of LoD in UML designs, is a significant predictor of class fault-proneness if it is combined with other metrics, namely ImpCoupling and KSLOC (an indicator of coupling and size respectively).
- ImpCoupling and KSLOC are significant predictors of class fault-proneness both in the univariate and multivariate analyses.
- SD_{msg} has a negative correlation with class fault-proneness. On the other hand, both ImpCoupling and KSLOC have positive correlations with class fault-proneness.

7.5.3 Goodness of Fit of the Prediction Models

In Section 7.5.1 and 7.5.2, we have learnt about metrics that are significant predictors of class fault-proneness. Additionally, we have learnt how those metrics correlate with class fault-proneness. However, up to this point we have no knowledge concerning the accuracy of the prediction models in predicting fault-prone classes. Therefore, in this section we compare the performance of the three prediction models in terms of their accuracy in predicting fault-prone classes.

To assess the accuracy of the prediction models, we look at the goodness of fit using several criteria, which include accuracy, precision, and recall. The definitions of the assessment criteria are given as follows:

- **Accuracy:** The percentage of classes correctly predicted as faulty and non-faulty: $\frac{TP+TN}{TP+TN+FP+FN}$
- **Precision:** The percentage of predicted faulty classes that are correctly classified as faulty: $\frac{TP}{TP+FP}$
- **Sensitivity (recall):** The percentage of faulty classes predicted as such: $\frac{TP}{TP+FN}$

Table 7.5: The Goodness of Fit of the Multivariate Models (probability threshold = 0.5)

Criteria	**Prediction Models**					
	Model-U		**Model-C**		**Model-H**	
	No LOOCV	LOOCV	No LOOCV	LOOCV	No LOOCV	LOOCV
Accuracy	65%	62%	58%	55%	72%	72%
Precision	64%	62%	59%	56%	76%	76%
Sensitify (Recall)	69%	62%	55%	48%	65%	65%
Specificity	62%	62%	62%	62%	79%	79%
FP rate	19%	19%	19%	19%	10%	10%
FN rate	15%	19%	22%	26%	17%	17%

- **Specificity:** The percentage of non-faulty classes predicted as such: $\frac{TN}{TN+FP}$
- **False positive rate:** The percentage of classes incorrectly predicted as faulty: $\frac{FP}{TP+TN+FP+FN}$
- **False negative rate:** The percentage of classes incorrectly predicted as non-faulty: $\frac{FN}{TP+TN+FP+FN}$

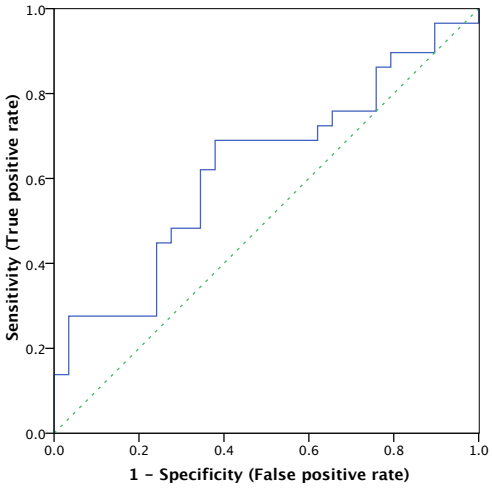
The work of Ostrand and Weyuker reported in [105] discusses the above assessment criteria in detail. For each definition, there is a formula that corresponds to the confusion matrix depicted in Table 7.3. The confusion matrix is a visualization tool that helps us identify correct and wrong classifications easily. For example, Table 7.4 provides classification results of the prediction models visualized using a confusion matrix. For the *precision* of Model-U, for example, $Precision = \frac{TP}{TP+FP}$; thus, we obtain $\frac{18}{18+11} = 0.62$. Hence, of all classes that are predicted as faulty by Model-U, 62% of them are actually faulty.

The results in Table 7.4 are obtained after running the leave-one-out cross validation. Leave-one-out cross validation provides a more realistic assessment by excluding one observation, building the prediction model on the remaining observations, and assess the model by predicting or classifying the excluded observation. These steps are performed iteratively for all observations. We perform multiple runs of cross validation (more than 5 runs), and the results remain consistent.

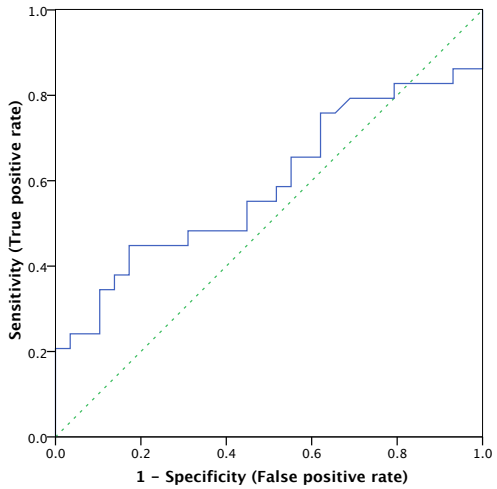
The complete comparison of goodness of fit criteria between the three prediction models is provided in Table 7.5. The table shows the accuracy of the prediction models with and without cross validation—LOOCV columns provide results obtained after running leave-one-out cross validation.

Looking at the results in Table 7.5, we can see that the goodness of fit of the non cross-validated models is generally better than the cross-validated ones. This phenomenon is natural because prediction models generally perform better when built and tested using the same data set. Nonetheless, Model-H does not decrease in accuracy after running the cross validation method.

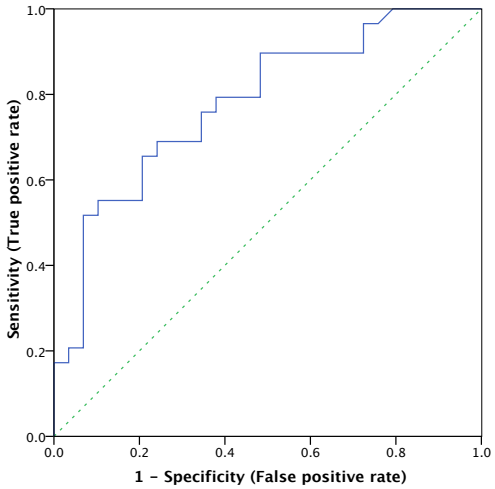
The results in Table 7.5 show that Model-U, which is composed of SD_{msg} and ImpCoupling metrics, has a moderate accuracy. For the cross-validated results, it demonstrates 62% accuracy, precision, recall, and specificity. Furthermore, it has 19% false positive and false negative rate. Model-C, on the other hand, appears to be the most unsatisfactory



(a) Model-U (area = 62%)



(b) Model-C (area = 59%)



(c) Model-H (area = 78%)

Figure 7.1: Accuracy of the prediction models represented using ROC curves

model. Only its specificity and false positive rate are similar to Model-U, but the rest of the assessed criteria are inferior to those of Model-U. Finally, we can see that Model-H has the best accuracy compared to the other prediction models.

Another way to assess accuracy is by looking at the ROC (Receiver Operating Characteristics) curve [60]. The ROC curve shows the trade-off between sensitivity and specificity across classification cut-off points—that is, an increase in sensitivity will be followed by a

Table 7.6: A Comparison of the goodness of fit

Models	Evaluation Criteria					
	Accuracy	Precision	Sensitivity	Specificity	FP Rate	FN Rate
Model-H (LOOCV)	72%	76%	65%	79%	10%	17%
Briand et al. (10-fold CV)	80%	78%	79%	81%	11%	10%
Gyimothy et al.	69%	72%	44%	69%	7%	23%

decrease in specificity. If we plot *sensitivity* against $(1 - specificity)$ in a graph, prediction models with a perfect accuracy (high sensitivity and specificity across all cut-off points) will have area of 1 under the ROC curve.

Figure 7.1 shows the ROC curves of the prediction models. The solid (blue) line represents the accuracy of the prediction models built using leave-one-out cross validation method. The dashed (green) line, on the other hand, represents the accuracy baseline—the accuracy of a prediction model should be above this line to have any value.

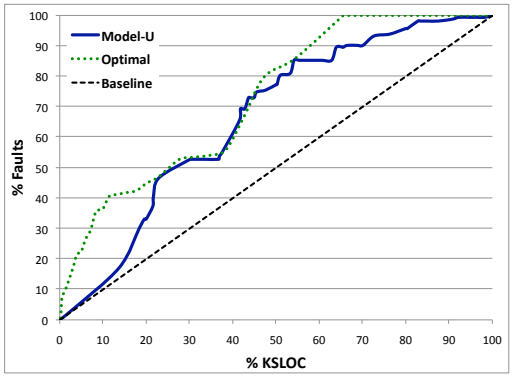
As shown in Figure 7.1, Model-H has the largest area under the ROC curve (78%), followed by Model-U (62%), and finally Model-C (59%). To interpret these numbers, imagine that we have classified all classes into either faulty or non-faulty group. If we randomly draw one class from the faulty and non-faulty groups respectively and subsequently predict the fault-proneness of both classes, the area under the curve represents the percentage of randomly drawn pairs that have been correctly classified—that is, each class in a pair is correctly classified as being faulty or not faulty.

The results discussed above essentially show that developing a prediction model using code metrics alone, i.e., best represented by SLOC, does not give a satisfactory results in terms of prediction accuracy. On the other hand, the use of UML design metrics (i.e., SD_{msg} and ImpCoupling) as predictors yields a slightly better accuracy than using SLOC metric. Furthermore, we see that combining the UML metrics and code metrics as predictors has resulted in a prediction model with the best accuracy.

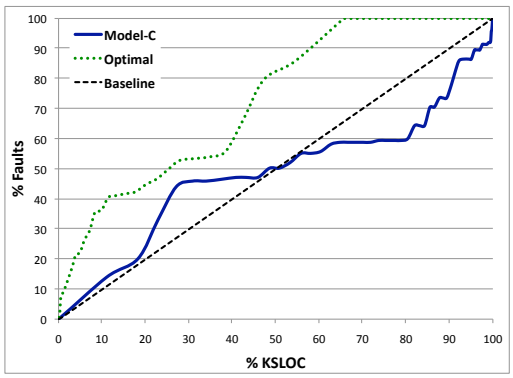
In Table 7.6 we compare the goodness of fit of Model-H with the prediction model proposed by Briand et al. [28] and Gyimothy et al. [57]. These two prediction models were based solely on object-oriented metrics measured from source code such as coupling, cohesion, and inheritance, and were also built using logistic regression. The results show that Model-H generally is more accurate than that of Gyimothy's, but compared to Briand's Model-H has a relatively higher false negative rate, which also explains why its sensitivity is also lower than that of Briand. Further, similar to Gyimothy's model, Model-H is more accurate in predicting non fault-prone classes than in predicting fault-prone classes (specificity is fairly higher than sensitivity).

7.5.4 Assessing Cost-effectiveness

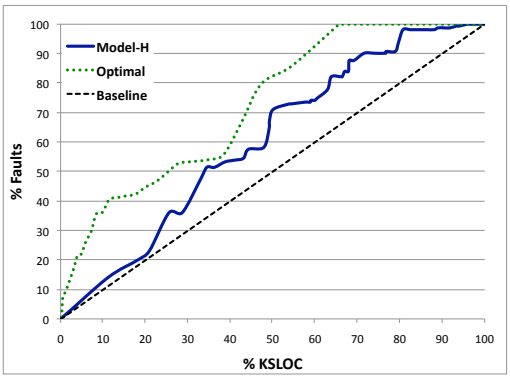
Another important aspect to be measured from a fault prediction models is its cost-effectiveness when applied in real software projects. Arisholm et al. suggests a surrogate measure of cost-effectiveness by considering the trade-off between the verification effort and the actual



(a) Model-U



(b) Model-C



(c) Model-H

Figure 7.2: Cost-effectiveness of the Prediction Models

numbers of faults discovered [12]. For example, verification effort might be roughly proportional to some properties of the verified classes such as size or complexity. Hence, large or complex classes that are predicted faulty will require more effort for scrutiny and retest. Having said that, an ideal prediction model should assist testers to prioritize verification effort so that a small percentage of lines of code (i.e., small classes) that contain most of the faults can be identified first for further scrutiny.

To assess the cost-effectiveness of the constructed prediction models, we use the same approach described in [12]. The cost-effectiveness of the prediction models is visualized using cost-effectiveness (CE) graphs presented in Figure 7.2. The solid (blue) line represents the CE curve, which indicates the actual percentage of cumulative faults related to a percentage of cumulative lines of code (in kilo SLOC). We obtain the CE curve by ordering classes from high to low fault probability (classes with similar fault probability will be further ordered from large to small classes).

The dotted (green) line represents an optimal curve. This curve shows the cost-effectiveness

of a prediction model that prioritizes on faulty classes with small sizes for verification. The optimal curve is determined by first placing the faulty classes, and then sort them in an increasing order of size. The next step is to place the non-faulty classes, and also order them based on size from small to large. Performing this procedure maximizes the area under the curve for a set of faulty classes. Finally, the dashed (black) line is a baseline that represents an identical percentage of lines of code to the percentage of faults of classes selected for verification. From a cost-effectiveness point of view, a prediction model should have a CE curve above the baseline to have any value.

By visually inspecting the CE curves of the prediction models in Figure 7.2, we can see that Model-U has the largest area under the curve. Hence, we can consider Model-U as the most cost-effective prediction model compared to the other two models. On the other hand, Model-C is the worst in terms of cost-effectiveness. We can see in Figure 7.2(b) that using Model-C for prioritizing verification effort is no longer cost-effective when we inspect beyond, roughly, 50% of the total size of the classes selected for verification. In fact, after inspecting 28% of the total size, testers can not expect to catch substantially more faults.

Depending on the nature of the CE curve of a prediction model, software testers can determine the amount of effort to be spent for verification. For example, if in a given project the allocated verification effort can only get testers to scrutinize and retest 60% of the total lines of code, then by using Model-U verifying 55% (corresponds to roughly 85% identified faults) of the total lines of code seems to be the most cost-effective (increasing coverage would not significantly increase the number of identified faults). Similarly, in order to be cost-effective, testers can decide to verify only 28% and 50% (correspond to roughly 46% and 71% identified faults) of the total lines of code for Model-C and Model-H respectively.

7.6 Discussion

Up to this point we have learnt some important facts. First, we have seen that two design metrics measured from sequence diagrams (i.e., ImpCoupling and SD_{msg}) are good predictors of class fault-proneness. However, SD_{msg} is a significant predictor only when it is combined with ImpCoupling, or ImpCoupling plus KSLOC. Furthermore, we have learnt that an increase in SD_{msg} of a class decreases the odds of that class for being faulty. On the contrary, an increase in ImpCoupling of a class increases the odds of that class for being faulty. While there is evidence about the significant impact of import coupling on fault-proneness (see for example in [28]), the effect of SD_{msg} (i.e., the level of detail of messages in sequence diagrams) on class fault-proneness was unexplored. Therefore, this result gives a new insight about how the quality of UML models might affect class fault-proneness.

Second, we have seen that the accuracy of the prediction model built solely using UML design metrics (i.e., ImpCoupling and SD_{msg}) is modest. Nevertheless, we need to emphasize the fact that using design metrics as predictors still gives a better accuracy (on average, it has nearly 6% higher accuracy) than using KSLOC. This phenomenon has an important implication for quality assurance activity: it is visible to perform early quality predictions (prior to software construction) using UML design metrics (e.g., using Model-U), from which sufficient indications about fault-prone classes can be expected. Based on this prediction,

software designers can recheck the design of predicted faulty classes and perform refactoring when necessary (i.e., by increasing level of detail or reducing coupling). We also learn that ImpCoupling and SD_{msg} can be a significant input to prediction models built using code metrics, which might be useful for predicting fault-prone classes in the maintenance phase.

Third, we have seen in Figure 7.2 that both Model-C and Model-H are less cost-effective compared to Model-U. In essence, this result indicates the very effect of using size-related variables as predictors such as SLOC. It has been quite well established that lines of code significantly correlates with class fault-proneness—that is, an increase in lines of code correspond to a higher number of defects and fault-proneness [57, 121]. As such, employing predictors that are highly influenced by size will generally result in predictions in which large classes are ranked first as fault-prone. Assuming that verification effort is roughly proportional to code size, prioritizing large classes for verification might not be cost-effective—this is particularly true because the number of faults contained in fault-prone classes is unknown. All in all, the results suggest that the choice of predictors is an important factor to the cost-effectiveness of the prediction model in its real application.

Finally, the cost-effectiveness assessment of the prediction models does not take into account the effort spent on constructing the models. In particular, collecting metrics from sequence diagrams of UML models created using Rational XDE is quite labour-intensive if done manually. Nevertheless, the metric calculation can be done automatically by a tool that can read sequence diagram information from an XDE file. We are working on incorporating such capability to the MetricView tool.

The implication of this study for software engineering practice is two-fold. First, as suggested by the results of this study, high import coupling and low level of detail in sequence diagrams might increase the probability of a class for being faulty in the implementation. Given this result, software architects are advised to create/update their design guidelines such that coupling and level of detail aspects are also taken into consideration. Second, software engineers can adapt the approach discussed in this chapter to perform quality prediction in their own projects. Depending on their specific needs and constraints, software engineers can choose whether to use one prediction model over the other (i.e., UML-based or hybrid model), or they might even decide to add more predictors to the models. The results of the prediction can then be used to prioritize testing effort on classes that have high probability to contain faults.

Additionally, fault probability resulted from prediction models can be combined with fault diagnosis techniques to improve fault localization. For example, the work of Feldman, Provan, and Gemund proposes a model-based diagnosis (MBD) technique called FRACTAL (Framework for Active Testing Algorithms) to isolate faults [47]. Based on a system model, an initial observation, and a diagnosis, the technique computes the set of input test vectors that will minimize diagnostic ambiguity with the least number of test vectors. Our approach enhances such model-based diagnosis technique by providing fault probability of software components as input, which can further reduce the number of diagnoses, and thus improve the practicality and accuracy of the fault diagnosis technique.

7.7 Threats to Validity

In this section, we discuss some limitations of this study. We particularly put emphasis on threats to internal and external validity.

Threat to the internal validity of correlation studies mainly concerns the influence of confounding factors—that is, whether the observed effect is substantially due to the experimental treatments and not confounded by other factors. In the context of this study, the significant correlation between message detailedness (or import coupling for that matter) and class-fault proneness may be confounded by the characteristics of the implemented classes. For example, one may think that implementation classes that are poorly modeled in sequence diagrams (i.e., in terms of level of detail of messages) are those classes with high coupling, large sizes, or high complexity. To control the effects of such confounding factors, in one of the multivariate analyses we have also combined code metrics (SLOC, CBO, and McCabe's complexity) with the UML metrics; as such we accounted for the effects of the code metrics. Interestingly, we have seen that both message detailedness and import coupling remain significant predictors before and after the inclusion of the code metrics. Another possible threat is presuming that classes that are modeled poorly in sequence diagrams are generally also incorrectly specified (semantically). While this seems plausible, we have learnt from observations of the bug registrations that defects are very rarely caused by incorrect specifications in the UML model. Even if such a phenomenon occurred, the frequency is so low that it will not have a significant effect on the results of this study.

Threat to external validity concerns the ability to generalize the results of a study to broader contexts. Considering the fact that this study is based on a single case study, we obviously cannot make a strong claim that the results discussed in this chapter are generalizable to other software projects. This is particularly true because project-specific characteristics will have significant impacts on the results of the study. For example, the case study that we used has a reasonably good model–code correspondence. Clearly, we can not expect to observe similar results from projects in which the implementation code deviates to a large extent from the UML model. Therefore, further research is needed to assess the usefulness of the proposed metrics as predictors of fault-prone classes across different projects possessing similar 'key' characteristics.

7.8 Conclusion and Future Work

This chapter reports on an empirical investigation about the usefulness of UML design metrics as predictors of class fault-proneness. We collect empirical data from a significant industrial project, and extract metrics from class diagrams and sequence diagrams of the UML model and from source code to construct three prediction models: 1) based on UML metrics, 2) based on code metrics, and 3) based on UML and code metrics.

Results show that two UML design metrics, namely *message detailedness* and *import coupling*—both measured from sequence diagrams, are significant predictors of class fault-proneness. Furthermore, we have found that the prediction model built using the combination of UML design metrics and code metrics demonstrates the best performance. Interest-

ingly, we have also observed that using UML metrics as predictors gives better prediction accuracy than using code metrics only. Finally, if we take into account the cost-effectiveness of the prediction models, the model built using UML metrics turns out to be the most cost-effective.

We consider the results of this study as an initial exploration to unveil the usefulness and cost-effectiveness of using UML metrics as predictor of class fault-proneness. While the results show some promising findings, we cannot make a strong claim concerning the generalizability of the results. This is particularly true because a fault prediction model is highly influenced by the data set from which it was constructed—which is also highly dependent on the software development settings [24].

Future research needs to investigate whether the UML design metrics used in this study are also significant predictors of class fault-proneness in different software projects. Also important is to apply the prediction model built using the UML metrics to ongoing projects during the design phase. By applying the prediction model to a running project, we can assess the real accuracy and cost-effectiveness of the prediction model, and thus we expect to learn much more about the benefits and cost-effectiveness of using design metrics for early prediction of software faults.