A UML Approximation of a Subset of the CK Metrics and Their Ability to Predict Faulty Classes

by

CAMARGO CRUZ Ana Erika

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Ochimizu Koichiro

School of Information Science
Japan Advanced Institute of Science and Technology

September 2011

Abstract

Design-complexity metrics, while measured from the code, have shown to be good predictors of fault-prone object-oriented programs and related to several other managerial factor such as productivity, re-work effort for reusing classes and design effort, and maintenance effort. Some of the most often used metrics are the Chidamber and Kemerer metrics (CK). Because earlier assessments of such managerial factors are desirable, prior to the code implementation, our research mainly concerns two topics. The first one concerns to the how can we approximate the code CK metrics using UML diagrams; and the second one concerns the use of such UML approximations to predict faulty object-oriented classes.

First, we define our UML metrics, approximations of the Weighted Methods per Class (WMC), Response For Class (RFC) and Coupling Between Objects (CBO) CK code metrics using UML communication diagrams. Second, we evaluate our UML metrics as approximations to their corresponding code metrics. Third, in order to improve the approximations of our UML metrics, we study the application of two different data normalization techniques, and select the best one to be used in our experiments. Finally, because code CK metrics have shown repetitively their ability to predict faulty code in several previous works, we evaluate our UML CK metrics as predictors of faulty code. In order to do so, we first construct three prediction models using logistic regression with the source code of a package of an open source software project (Mylyn from Eclipse), and we test them with several other of its packages. Then, we applied these models to three different small-size software projects, using, on the one hand, their UML metrics, and on the other hand, their corresponding 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. As for the WMC metrics, both the proposed UML and its respective code metric showed a poor fault-proneness prediction ability.

Our plans for future work mainly concern the exploration of other areas of research in which our UML metrics can be applied and as for the topic of fault prediction the following subjects for further study have been considered: data normalization and other pre-processing techniques, the study of other metrics to be included in our prediction models (such metrics should be easily obtainable before the implementation of the system and different to design-complexity metrics), and other methodologies to predict fault-proneness of code (different to logistic regression).

Acknowledgments

Foremost, the author wishes to express her most sincere gratitude to her principal advisor Professor OCHIMIZU Koichiro of Japan Advanced Institute of Science and Technology for his constant encouragement, advises, kind guidance and support.

The author would like to extend her gratitude to the rest of her thesis committee: Professor OGAWA Mizuhito, Associate Professor SUZUKI Masato and Associate Professor AOKI Toshiaki of Japan Advanced Institute of Science and Technology, and Professor MATSUMOTO Kenichi of Nara Institute of Science and Technology for their valuable reviews, advises and suggestions.

Last but not least, the author would like to express her gratitude to her mother Maria de los Angeles Cruz Salas, her father Valentin Camargo Perez, brother Emmanuel Ivan Camargo Cruz and sisters Alba Olivia Camargo Cruz and Daysi Iliana Camargo Cruz for their continuous encouragement, unconditional love and moral support.

Contents

Chapter 1

Introduction

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 subjected to 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 such elements 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 complexity of the design, and they 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 [14, 20]. In addition, they have been widely used in prediction models to determine how faulty classes would be after coding when their testing takes place, showing to be good predictors of fault-proneness of code [5, 9, 30, 32, 40, 43].

Although the authors of the CK metrics established that they can be gathered prior to program execution [13], they cannot be measured accurately during the design phase, unless there exists a very well detailed 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 as *objective* as possible, preventing individual bias and avoiding inconsistencies, which means that if different people perform the measurement, they will give similar values [7]. 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 elabora-

tion of the software, for measuring the CK metrics accurately, rather than simply written documentation.

The present paper reports the results of our research on two main topics, first how to approximate a subset of the CK metrics using UML diagrams, because early attempts of approximation using UML class diagrams were inefficient. And second, how such approximations can be used to detect those classes which would result most faulty after their implementation.

Our hypothesis is that if the code CK metrics are per se good predictors of fault-proneness, and if our UML approximations are close enough to their respective code metrics, then we can expect to predict fault-proneness of code at design time as well as if we were using code CK metrics.

To test the ability of our UML metrics to predict faulty classes, the following steps are carried out. First, we build three univariate logistic prediction models using the code CK metrics of a large-size-open-source software project. Second, we test the resulting models using the code CK metrics of three small-size projects developed by students of JAIST. Third, the same models were tested using the UML CK metrics of the JAIST projects, which are approximated according to our proposal. Last, using the results of the two previous steps, we evaluate the prediction ability of our UML CK metrics in comparison with the ability of their corresponding code metrics.

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, more details are given in a later Chapter.

To sum up, please see Figure 1.1, because the CK metrics, when measured from the code, have shown repetitively to be good predictors of faulty code in previous research works 1.1-A, and because earlier predictions are desirable, we are proposing, first, an approach to approximate a subset of the CK metrics using UML diagrams 1.1-B; and second, a methodology to enable our UML approximations to predict faulty code 1.1-C.

This document is divided into the following Chapters: In Chapter 2, we present a brief background theory on the different areas and methodologies were needed to carry out our study. In Chapter 3, we define our UML metrics, approximations of a subset of the code CK metrics. In Chapter 4, we present the results of our initial evaluation of our UML metrics as approximations of their corresponding code metrics. In Chapter 5, we present the results of the application of two data transformations for data normalization, and final evaluation of our UML metrics. In Chapter 6, we empirically validate our UML metrics as as predictors of faulty code. In Chapter 7, we summarize some related work to our research. In Chapter 8, we discuss other applications of our UML metrics. And finally, in Chapter 9, we draw our conclusions and describe our plans for future work.

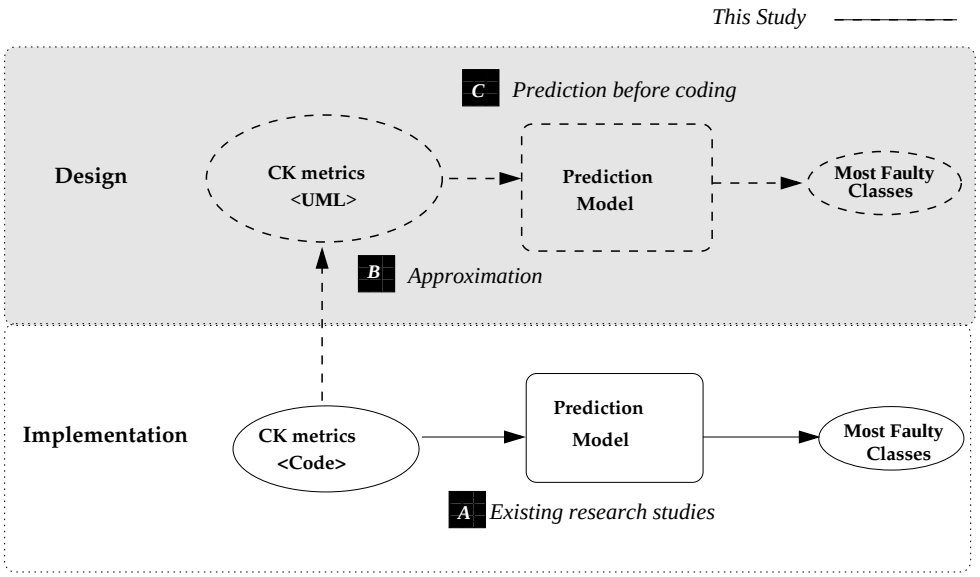

Figure 1.1: Research Outline

Chapter 2

Background

This Chapter presents a brief background theory on the different fields and methodologies required to carry out our research study. We first introduce the concepts of software metrics and measurement. Second, we provide the CK metrics' definition and guidelines according to their authors. Third, we review the concepts and elements of UML communication diagrams, because our UML metrics are derived from this kind of diagrams. Finally, we introduce some of the techniques, which are generally used to build prediction models, so that we can empirically validate our UML metrics as predictors of fault-proneness of code.

2.1 Measurement and Metrics

Within the software engineering community, there is sometimes confusion and inconsistency over the use or the terms metric and measure. In this study, we understand metric as defined in [44]. A metric is defined to be the mathematical definition, algorithm or function used to obtain a quantitative assessment of a product or process. The actual numerical value produced by a metric is a measure. Thus, for example, Cyclomatic complexity is a metric, but the value of this metric is the Cyclomatic complexity measure.

Software measurement is the continuous process of defining, collecting and analyzing data on the software development process and its products in order to understand an control the process and its products, and to supply meaningful information to improve that process and its products. Measuring will normally comprise several metrics, again resulting in several measurements per metric.

Metrics can be classified into various categories, the most common, according to [50], are:

- Product and process metrics. A product metric is a measurement of an intermediate or final product of software development, and therefore addresses the output of a software development activity. Examples of such metrics are a size metric for the number of requirements, a complexity metric for the software code, etc. Process

metrics measure the characteristics of the overall development process, such as the number of defects found throughout the process during different kinds of reviews, etc.

- Objective and subjective metrics. Objective metrics are absolute measures, and count attributes or characteristics in an objective way. Subjective metrics are measurements that involve human subjective judgement.

- Direct and Indirect metrics. A direct metric is a measurement that does not depend on the measurement of any other characteristic. In contrast, a indirect metric needs the measurement of one or more other, therefore, it always contains a calculation of at least two other metrics.

The use of efficient metrics and measurements in any process let us have baselines to control, to predict future trends and to take decisions during the process. Metrics are measured to learn something. Thus, it is important not just to measure but also to have the proper guidelines of the metric used. In other words, we must be able to interpret the meaning of the metric correctly [47].

Ideally a good metric should be simple, objective, easily collected, robust and valid [7, 20]. A metric is:

- simple, if its definition and usage are simple.

- objective, when different people performs the same measurement and all of them obtain similar values.

- easily collected, when the cost and effort to obtain the measure is reasonable.

- robust, when is insensitive to irrelevant changes, allowing useful comparisons. A metric is robust, if it can cope with abnormal inputs.

- valid, if the metric is measuring what is supposed to measure, so that we can trust its value.

Software measurement data provide visibility of the current development process and characteristics. Such visibility is required to reduce complexity and increase understanding of the process and products. Understanding means determining the different variables that exist during execution of a process. Once basic understanding has been established (the variables are known), the collected and analyzed data can be used to control the process and the products, by defining corrective and preventive actions. Furthermore, based on analysis, the collected data measurements can be used to assess the process, and therefore act as an indicator of development process problem areas, from which improvement actions can be identified.

2.2 The Chidamber and Kemerer Metrics

The CK metrics were defined to measure unique aspects of the object-oriented approach and complexity of the design. They have been traditionally measured from the implementation of the classes. In the following paragraphs, for the sake of completeness, we re-state the definition of the CK metrics as given by their authors in [13]:

- Depth of inheritance tree (DIT). It is the maximum number of steps from the class node to the root of the inheritance tree and is measured by the number of ancestor classes.

  Deeper inheritance trees constitute greater design complexity, but also promote the re-usability of inherited methods. Therefore, DIT values are a trade-off between re-usability and complexity.

- 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. Therefore, NOC values are also a trade-off between re-usability and complexity.

- 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. Therefore, low values of WMC are desirable.

- 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. Therefore, low values of CBO are desirable.

- 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. 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. Therefore, low values of RFC are desirable.

- 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.

The higher LCOM in a class, the higher the dissimilarity of its methods is. Therefore, low values of LCOM are desirable.

Table 2.1: Guidelines for the CK metrics

Metrics	**Objective**
Weighted methods per class	⇓
Response for class	⇓
Lack of cohesion of methods	⇓
Coupling between objects	⇓
Depth of Inheritance	◇
Number of Children	◇

⇓: Low Values ◇: Trade-off (between complexity and re-usability)

The final purpose of the definition of the CK metrics was, according to their authors, to improve the development of the software. This can be achieved through the identification of outliers (extreme value deviations), which might be a sign of high complexity and/or possible design violations [13]. But how can we properly identify these outliers?

In 1994, Chidamber and Kemerer provided the definition of their metrics along with their guidelines, which can be summarized in Table 2.1 [13]. Yet, it was not clear how much is high or low. In other words, which the thresholds of these metrics were. In 1998, the same authors described in [14] the managerial use of these metrics, then, it was said that thresholds of these metrics can not be determined before their use, and they should be derived and used locally for each dataset (each project). Finally, it was recommended to use 80th and 20th percentiles of the distributions to determine high and low values of the metrics [14, 31, 46].

Furthermore, when measured from the code, the CK metrics have been related to managerial factors such as productivity, re-work effort for reusing classes and design effort and maintenance effort [14, 20]. In addition, the CK metrics have shown their ability to predict fault-prone code in several empirical studies [5, 9, 30, 32, 40, 43]. Further details are given in the Chapter of Related Work.

2.3 Communication Diagrams UML 2.0

A communication diagram shows interactions organized around the parts of a composite structure or the 'roles' of a 'collaboration'. It models the 'objects' and 'links' involved in the implementation of an 'interaction' and ignores the others.

'Roles' and 'connectors' describe the configuration of 'object's and 'links' that may occur when an instance of a context is executed. When the context is instantiated, 'objects' are bound to the 'roles', and 'links' are bound to the 'connectors'. Connectors, in their turn, may also be bound to various kinds of temporary links, such as procedure arguments or local procedure variables.

In a communication diagram a series of 'messages' among 'roles' are shown which are 'labeled arrows' attached to 'connectors'. Each message has a 'sequence number', an optional 'guard condition', a 'name' and 'argument list', and an optional 'return value'. The 'sequence number' includes the optional 'name of a thread'. All messages in the same thread are sequentially ordered. Messages in different threads are concurrent unless there is an explicit sequencing dependency. Various implementation details may be added, such as a distinction between 'asynchronous' and 'synchronous' messages.

Only objects that are involved in the context are modeled, although there may be others in the entire system.

UML communication diagrams are similar to sequence diagrams, both show interactions. Sequence diagrams show time sequences clearly, but do not show relationships explicitly. Communication diagrams show object relationships clearly, but time sequences must be obtain from sequence numbers.

Key Elements of a Communication Diagram

In the previous Section, we have introduced communication diagrams, and we have identified many important elements to build a communication digram. In the following paragraphs, for the sake of completeness, we will re-state the definition of such elements as found in [26].

1. Collaboration. It is a specification of a contextual relationship among objects that interact within a context to implement a desired functionality. The behavior of a collaboration may be specified by 'interactions' that show 'messages' flow in the collaboration over time.

2. Behavior. It is a generic term for the specification of how the state of an instance of a classier changes over time in response to external events and internal computations.

3. Object. An instance of a 'class'

4. Link. It is an individual connection among two or more objects.

5. Role. It is a description of an object in an interaction. It is possible for an object to play multiple roles in one or many collaboration instances.

6. Message. The conveyance of information from one role to another as part of an interaction within a context; at instance level, a communication from one object to another.

   A message may be a 'signal' or a 'call' of an operation. A signal is a message transited from one object (the sender) to one or more other objects (the receivers). The receipt of a signal may trigger a state machine transition. The sending of a signal is 'asynchronous'; the sender continues execution after sending it. A call, on the other hand, is a call of an operation of one 'object' (the 'receiver') by another 'object' (the 'sender' or caller). It can be modeled as TWO messages, a 'call message' and a later 'return message'. The receipt of an operation call may invoke a procedure

or may trigger a state machine transition or both. A call may be 'asynchronous' or 'synchronous'. If it is 'synchronous'; the sender continues execution after sending it.

(a) Structure

- interaction. The interaction that owns the message
- connector. The connector over which the message is transmitted
- send event. The occurrence specification of sending the message. The lifetime owing this occurrence specification is the 'sender' of the message
- receive event. The occurrence specification of receiving the message. The lifetime owning the occurrence specification is the 'receiver' message
- signature. The signal or operation the type of message
- arguments. A list of argument values compatible with the signature. A message has arguments, which must agree in numbers and types with the parameters of the specified operation or the attributes of the specified signal.
- return values. If any, must match the return types of the specified signal. Any return values for an asynchronously called operation are ignored and do not generate a reply message.
- synchronicity. Either asynchronous (signal or call) and synchronous (only calls).
- kind. Whether the message is complete (default, includes sender and receiver), lost (no receiver), or found (no sender). Lost and found messages are used in advanced modeling situations for noisy systems or system with incomplete knowledge. They are unnecessary in most models.

(b) Message Format

- A message is shown as a small 'labeled arrow'; attached to a 'path' between the 'sender' and the 'receiver' objects. The 'path' is the 'connector' used to access the 'target object'. The 'arrows' point in the direction of the 'target object'. In the case of a message from an object to itself, the message appears on a path looping back to the same object and the keyword $<< self >>$ appears on the target end.
- More than one message may be attached to one 'link'.
- The relative order message is shown by the 'sequence number' portion of the 'message label'.
- A message may be label with a 'guard condition'.
- Synchronicity. According to [26], the same 'arrow types', used in sequence diagrams, may be used to indicate the type of synchronicity of the message. Figure 2.1 shows an example of synchronous and asynchronous messages in a communication diagram, which shows the interactions that take place when an object *sensor* detects something. These interactions are:

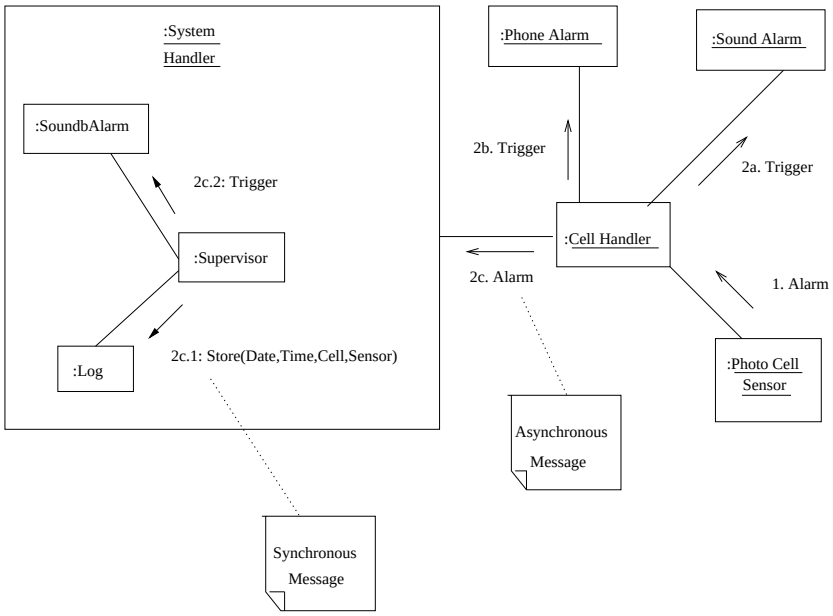

Figure 2.1: Example: Synchronicity in Communication Diagrams

  - First, the *sensor* sends an asynchronous alarm signal to the *Cell Handler*. Such a signal must be asynchronous given the requirements of a sensor, which does not need to wait for a return signal and is used to alert the system to an external event.
  - Next the *Cell Handler* sends in parallel trigger signals to all alarms (in this case, a *phone* and a sound *alarm*) and an asynchronous alarm signal to the System Handler.
  - Inside the *System Handler*, the alarm signal is handled synchronously, the *supervisor* thread first calls the internal *sound alarm* and then writes the event to the *log*.

  This example was taken from [16].
- Message label. The general syntax for a message label is expressed in Equation 2.1, further details are given in the next Section. Some examples are:
  - A simple message: $2 : display(x, y)$
  - A nested call with return value: $1.3.1 : p = find(spects) : status$
  - Conditional within second concurrent thread $1b.4[x < 0] : invert(x, colo)$
  - An iteration: $3.1 * [i := 1..n] : update()$

Message Label

The general syntax of a message label is the following:

$$sequence\text{-}expression_{opt} message \tag{2.1}$$

where,

- *sequence-expression* is a dot-separated list of sequence-terms followed by a colon (':'). Each term represents a level of procedural nesting within the overall interaction. If all the control is asynchronous, then nesting does not occur and a simple one-level numbering system is used. Each sequence-term has the following syntax:

$$label \qquad iteration\text{-}expression_{opt} \tag{2.2}$$

where,

- *label* is an integer and/or name. An integer serves to represent the sequential order of the message. An example is: message 3.14 follows message 3.13 within the activation of message 3.1. A name is used to represent a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. An example is: Message 3.1a and message 3.1b are concurrent within the activation of message 3.1. All threads of control are equal within the nesting depth.

-*iteration-expression* represents conditional or iterative execution. This represents zero or more messages that are executed, depending on the conditions. The choices are:

  - An iteration: $*[iteration\text{-}clause]$, where *iteration-clause* is meant to be expressed in pseudo-code or an actual programming language; UML does not prescribe its format. an example would be : $*[i := 1..n]$
  - A branch: $[condition\text{-}clause]$, where *condition-clause* represents a message whose execution is contingent on the truth of the condition clause. The condition-clause is meant to be expressed in pseudo-code or an actual programming language. An example would be: $[X > y]$

- and *message*, in its turn, has the following syntax:

$$\lfloor atribute = \rfloor_{opt} name \lfloor (argument_{list}) \rfloor_{opt} \lfloor : return\text{-}value \rfloor_{opt} \tag{2.3}$$

where,

- *name*, is the name of the signal or operation. The entire name string can be replaced by an asterisk, to indicate that any message in compatible with the model.

- *attribute*, is the optional name of an attribute to store the return value

- *argument values* can be replaced by a dash, to indicate that any value is compatible with the model. A parameter name may be include with an argument value, so an argument may have one of the following syntaxes.

$$argument\text{-}value \tag{2.4}$$

$$parameter\text{-}name = \text{argument-value} \tag{2.5}$$

In order to ease the understanding of how all these elements interact with each other, we have selected the elements we think are the most important, and we have elaborated a meta-model of a communication diagram, such model is shown in Figure 2.2.

2.4 Prediction Techniques

As mentioned earlier, after defining our UML metrics, we evaluate them not only as approximations of the CK metrics, but also as predictors of fault-proneness of code. Therefore, this Section is dedicated to the study of those techniques that can help us to model our data and to build our fault-prediction models.

In order to build any sort of prediction model, first, the data to be modeled is collected. Then, using a statistical technique, a prediction model can be derived. In order to select which is the most suitable statistical technique to be applied, the types of measurement scales of the variables used must be analyzed. Therefore, in the following paragraphs, we first introduce some background theory on measurement scales, and later, we give the details of the candidate statistical techniques for our study. For further study on this topic, the reader can refer to [49].

On the other hand, apart from the statistical techniques, in the field of fault-proneness prediction, some researchers have used Artificial Neural Networks (ANN), an example is the research work reported in [32]. ANN is a computational model used for predicting one or more dependent variables using various independent variables. Further details, concerning the application of this technique, are provided in a later Section.

2.4.1 Types of Measurement Scales

Measurement is a process by which numbers or symbols are attached to given characteristics or properties according to predetermining rules. It is said that measurements can be classified into the following types: nominal, ordinal, interval and ratio. In the following lines we described briefly these four types of scales.

- Nominal Scale

  Consider we have as a variable the gender of a subject. We could use numbers to represent subject's genders. For example, we can arbitrarily assign number 1 for females and number 2 for males. The assigned numbers themselves do not have any meaning, and therefore it would be inappropriate to compute such statistics as mean and standard deviation of the gender variable. The numbers are simply use to categorize subjects into different groups or for counting how many are in each category.

  The described measurements scales are called nominal scales and the resulting data are called nominal data. The statistics that are appropriate for nominal scales are those which are based on counts such as mode and frequency distribution.

- Ordinal Scale

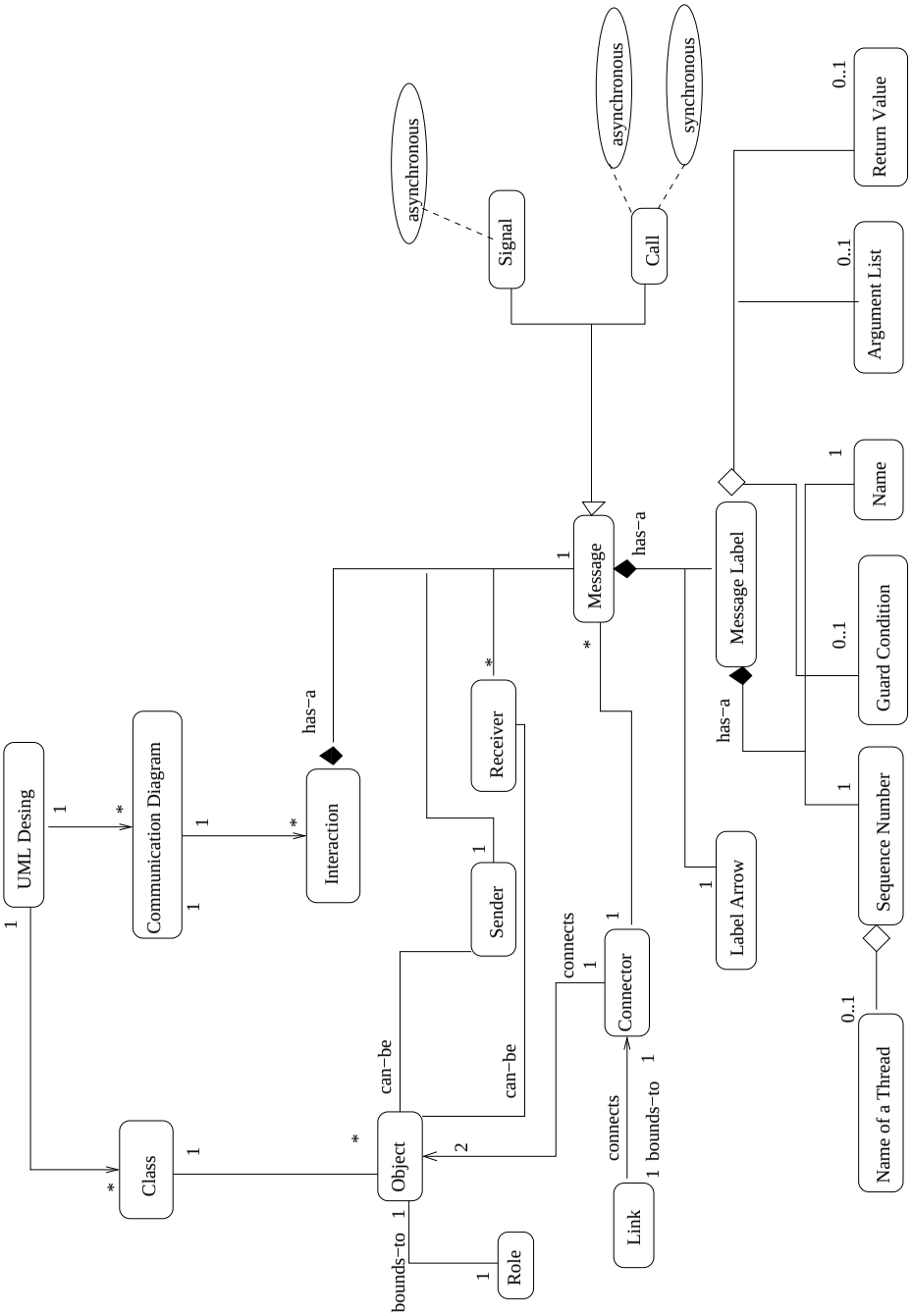

Figure 2.2: Meta-model of a UML Communication Diagram

Suppose we want to measure person's preferences for four brands of soda, Brands A, B, C and D. We would ask each subject to rank the four brands from 1 to 4, by assigning a 1 to the most preferred brand and 4 to the least preferred one. Consider a particular subject ranks the four brands as follows: Brand B - 1, Brand D - 2, Brand A-3, Brand C - 4. Then, we could conclude that this subject prefers Brand B to Brand D, Brand D to Brand A and Brand A to Brand C. However, even though the differences in the successive numerical values of the ranks are equal, we cannot state by how much the subject prefers one brand to another one. In other words, successive categories do not represent equal differences of the measured attribute.

The described measurement scales are referred as ordinal scales and the resulting data are called ordinal data. Valid statistics that are computed for ordinal-scaled data are mode, media, frequency distributions and non-parametric statistics such as rank order correlation.

- Interval Scale

  Quantitative attributes are all measurable on interval scales, as any difference between the levels of an attribute can be multiplied by any real number to exceed or equal another difference. A highly familiar example of interval scale measurement is temperature with the Celsius scale. In this particular scale, the unit of measurement is 1/100 of the difference between the melting temperature and the boiling temperature of water at atmospheric pressure. The "zero point" on an interval scale is arbitrary; and negative values can be used.

  Continuing with our previous example on brand preferences, suppose that instead of asking people to rank order the brands, we ask them to rate their brand preference according to the following five-point scale: 1 Very high preference, 2 High preference, 3 Moderate preference, 4 Low preference and 5 Very low preference. If we assume that successive categories represent equal degrees of preferences, then we could say that the difference in a subject's preference from the two brands that received ratings 1 and 2 , it is the same as the difference in the subject's preference for two other brands that receive rating of 4 and 5. However, we still can not say that the subject's preference for a brand that received a rating of 5 is five times the preference for a brand that received a rating of 1. Moreover, suppose we multiply each rating point by 2 and then add 10. This would result in the following scales: 12 Very high preference, 14 High preference, 16 Moderate preference, 18 Low preference and 20 Very low preference. The differences between the new categories are equal to the ones in the original categories; however, the ratio of the last to first category is not the same for the original scale. The ratio for the original scale is 5 and 1.67 for the transformed scale.

  Measurement scales, whose successive scales categories represent equal level of the characteristics that is being measured, and whose base values are arbitrary, are called interval scales. And the resulting data are called interval data. Properties of the interval scale are preserved under the following transformation: $Y_t = a + bY_o$, where Y_o and Y_t, respectively, are the original and transformed scale, and a and b

are constants. All statistics, except those ones based on ratios such as the coefficient of variation, can be computed for interval-scaled data.

- Ratio Scale

  Most measurement in the physical sciences and engineering is done on ratio scales. Mass, length, time, plane angle, energy and electric charge are examples of physical measures that are ratio scales.

  Ratio scales, in addition to having all the properties of the interval scale, have a natural base value that cannot be changed. For example a subject's age that a natural based value, which is zero. Ratio scales can be transformed by multiplying by a constant; however they cannot be transformed by adding a constant, as this will change the base value. That is, only the following transformation is valid for the ratio scale: $Y_t = b_o$.

  Since the ratio scale has a natural base value, statements such as "Subject A's age is twice Subject B's age" are valid. Data resulting from ratio scales are referred as ratio data. There is no restriction on the kind of statistics that can be computed for ratio-scaled data. Variables measured using interval and ratio scales are called metric variables.

To conclude, let us analyze which are the measurement scale of the variables used in our study. We have basically two groups of variables: the CK metrics and fault-proneness of classes.

The CK metrics' scale can be classified as ratio. It is no nominal because their values are not arbitrary and do have some meaning. It is not ordinal because successive values of these metrics have equal differences. It is not interval because although the CK measures are successive values representing equal level of the characteristic they are measuring, their base values are not arbitrary. Therefore, the scale of the CK metrics can best classified as a ratio scale, because they have a natural base value that cannot be changed, 0.

On the other hand, to measure fault-proneness of a class, it can be done in different ways, the two most common used by fault-prediction practitioners are using number of faults per class, or simply a categorization such as Most Faulty and Least Faulty classes. The first case can be classified as a ratio scale, and the second case as a nominal scale.

2.4.2 Statistical Prediction Techniques

There are different statistical techniques that can be used to analyzed data. The objective of analyzing data is to extract the relevant information contained in the data, which can be used to solve a given problem. This given problem, normally is formulated in one or more null hypothesis. And the statistical techniques help us to extract that relevant information in the data and test our null hypothesis. In the following paragraphs we will give a brief description of these statistical methods, according to [49].

Statistical methods can be divided basically into: dependence methods and interdependence methods. Consider a data set, which consists of n observations on p variables. The dependence methods test for the presence or absence of relationship between two

sets of variables of the data set. However, if one of these data sets is designated with dependent variables and the second one with independent variables, then the objective of the dependence methods is to determine if the independent variable affects the set of dependent variables. In other words, statistical techniques only test for the presence or absence of relationship between two sets of variables. On the other hand, interdependence methods are used to identify how and why the variables are related among themselves.

Since we are looking for a technique that help us to find the relationship between CK metrics acting as independent variables and fault-proneness of classes as the dependent variable, we focus on the study of the most used dependence statistical methods.

The dependence methods for studies which consider one dependent variable and more than one independent variables are:

- *Linear analysis* is a technique that is used when the dependent variable and the independent variables are measured using a metric scale, resulting in a metric data such as income measured in dollars. Linear analysis considers a model where the dependent variable is a linear combination of the independent variables.

- *Analysis of variance* (ANOVA) is an appropriated technique to estimate linear models when the independent variables are nominal or categorical.

- *Discriminant analysis* is appropriated when we have the dependent variable measured using a nominal scale and the independent variables are measured using an interval or ratio scale.

- *Logistic regression* is an alternative procedure to Discriminant Analysis. Logistic regression which does not make any distributional assumption for the independent variables, as it does Discriminant Analysis, is more robust to the violation of the multivariate normality assumption (which will not hold when the independent variables are combinations of metric and nominal variables).

- *Discrete discriminant analysis* is appropriated when both sets of variables, the dependent variable and independent variables, are measured using categorical or nominal scale.

- *Conjoint analysis* is appropriated when the dependent variable is ordinal and the independent variables are measured using a nominal scale.

Considering the subject matter of our research, in which the dependent variable is the fault-proneness of code, which can be measured using a ratio scale, metric scale or nominal scale; and the independent variables, the CK metrics, which can be measured using a ratio or metric scale; linear analysis, discriminant analysis or logistic regression techniques are the techniques that can help us to build our prediction model.

Linear Regression

It is used when we have one or more independent variables (x) and one dependent variables (y). The dependent variable is a continuous metric. And the independent variables can not be linear combination of each others.

The goal of this technique is to find an equation that best predict the value of the dependent variable (y), as a linear function of the independent variables (x_s). Look at Equation 2.6, y represents the expected value of the dependent variable for a given set of x_s values, x_s represents the independent variables, b_i is the estimated slope of a regression of y on x_i and a is the intercept.

$$y_{exp} = a + b_1x_1 + b_2x_2 + ...b_nx_n \tag{2.6}$$

Two-group Discriminant Analysis

It is used when we have that our dependent variable is measured using the nominal scale, and the independent variables, on the other hand, are measured using an interval or a ratio scale. Thus, we have a data set in which the dependent variable is categorical or nominal and the independent variables are metric or continuous. Two-group discriminant analysis is a special case of multiple linear regression.

Consider the following examples:

- A medical researcher is interested in determining factors that significantly differentiate between patients who have had a heart attack and those who have no yet had a heart attack. The medical researcher then wants to use the identifies factors to predict whether a patient is likely to have a heart attack in the future.
- The marketing manager of a customer packaged goods firm is interested in identifying salient attributes that successfully differentiate between purchasers and non purchasers of brands, an employing this information to predict purchase intentions of potential customers.

Each of the above examples attempts to meet the following three objectives:

1. Identify the variables that discriminate "best" between the two groups.
2. Used the identify variables or factors to develop an equation or function for computing a new variable or index that will represent the differences between the two groups.
3. Used the identify variables or the computed index index to develop a rule to classify future observations into one of the groups.

The two-group case discriminant analysis can also be thought as multiple regression. In general, in the two-group discriminant analysis, we fit a linear equation of the type,2.7:

$$Group = a + b1 * x1 + b2 * x2 + ... + bm * xm \tag{2.7}$$

Where a is a constant and b1 through bm are regression coefficients. The interpretation of the results of a two-group problem is straightforward and closely follows the logic of multiple regression: Those variables with the largest (standardized) regression coefficients are the ones that contribute most to the prediction of group membership.

Logistic Regression

It is appropriated when we have one or more independent variables and our dependent variable has just two values, in other words, when the dependent variable is dichtonomous (faulty/no faulty, male/female, healthy/unhealthy, 1/0).

Consider the following scenarios:

- The marketing manager of a cable company is interested in determining the probability that a household would subscribe to a package of previous channel given the occupant;s income, education, occupation, age, marital status, and number of children.
- An auditor is interested in determining the probability that a firm will fail given a number of financial ratios and the size of the firm (i.e., large or small).

Discriminant analysis could be used for addressing each of the above problems, however, because the independent variables are a mixture of categorical and continuous variables, the multivariate normally assumption will not hold. In these cases one could used logistic regression, as it does *not make any assumption about the distribution of the independent variables. Logistic regression is recommended when the independent variables do not satisfy the multivariate normality assumption.*

The goal of logistic regression is to predict the probability of getting one of the two values of the dependent variable, given the independent variables. It uses the logit function to link the values of the independent variables to the probability of occurrence of one of the values of the dependent variable.

Looking at Equation 2.8, P is the probability of occurrence of one of the values of y, for example the probability of having a faulty class, x_s variables are the independent variables, b_i variable is the estimated slope of regression of P on x_i and a is the intercept.

$$P = \frac{1}{1 + e^{-(a+b_1x_1+b_2x_2+...b_nx_n)}} \tag{2.8}$$

2.4.3 Artificial Neural Networks

In the field of exploratory multivariate data modeling, Artificial Neural Networks (ANN) offer many advantages over conventional multiple linear regression methods, but also the use of ANN have some disadvantages, for our purposes. In the following paragraphs we will summarize some of its advantages and disadvantages cited by Kristen in [33].

ANN can identify patterns between the dependent and independent variables in data sets. Furthermore, they can create specialized regression models or model adjustments for all patterns discovered during certain analysis. Neural networks deal very effectively with data discontinuities and even nonlinear transformations. Additionally, it is said that neural networks adapt and learn in such way that they can choose models correctly for recurring patterns.

Many researchers have used neural networks in conjunction with regression analysis. Several researchers have also compared ANN regression to multiple linear regression. In

the majority of cases, the ANN analysis perform better than statistical regression methods. Some advantages of using ANN are:

- ANN are nonlinear and versatile, thus they can deal very effectively with data sets demonstrating not easily recognized or unknown nonlinearity, as well as with linear relationships among the data.
- Prior model specification is not required. When performing ANN regression analysis, the model under examination does not need to be specified in advance. Thus, ANN do not require explicit relationships between inputs and outputs to be defined in the data set.
- ANN do not make any assumptions about the population distributions of the variables used. Neither on multicollinearity of the independent variables.
- ANN can discover stable patterns in fuzzy data sets, where irrelevant data hide or disguise the model.

On the other hand, some strong disadvantages of this technique are:

- Model parameters are unidentifiable. ANN are often referred to as black boxes, because the parameters and coefficients they derive to approximate patterns cannot be easily analyzed or interpreted. Unlike conventional regression techniques, ANNs lack marginal evaluators, which obscures the network from revealing the functional relationships among the variables.
- Although ANN often provide better results than multiple linear regression, the method is difficult to interpret.
- Data sets are usually larger than those used in statistical regression techniques.

To summarize ANN strength is, its ability to model no-linear relationships or to find difficult patterns. On the other hand, ANN do not make use of a function as it does linear regression and logistic regression. Therefore, they are often described as a black box, because the parameters and coefficients derived are not easy to interpret and to analyze.

2.4.4 Summary and Conclusions

In the previous sections, we have analyzed the candidate techniques to predict fault-proneness of code.

We learned that in order to select the most suitable statistical technique to model our data, the measurement scales of the used variables must be first analyzed. Therefore, since our CK metrics (independent variables) have a ratio scale and our fault-proneness variable can be expressed using either a ratio or nominal scale, the candidate statistical techniques to be applied in our study are: Linear Analysis, Discriminant Analysis or logistic regression.

On the other hand, we also learned that ANN, which has been used in previous studies of fault-proneness prediction, have the ability to model no-linear relationships or to find difficult patterns among data, however the resulting parameters are difficulty to interpret and analyze. Therefore, the dependent statistical techniques seemed to be more appropriated for our purposes.

From our three candidate statistical techniques, logistic regression is the technique that seems to be the most suitable for our study. This mainly because the two other techniques, linear regression and two-group discriminant analysis, make assumptions on the normality of the distribution of the independent variables. Moreover, if we use linear regression, we were assuming that a linear relationship exists between our independent metrics, CK metrics, and our dependent variable, fault-proneness of code.

Chapter 3

Definition of Goals and our UML CBO, RFC and WMC Metrics

Metrics serve as baseline to monitor the elaboration of a product. They can also be used to predict future trends and to detect possible flaws within the elaboration process before the actual release of the product. A good practical example is the construction of a house, which is based on a design detailed in some blueprints and a set of metrics can tell us how many walls the house will have, and then we could approximate the cost of these, among other things. However, on the field of object-oriented software, although UML diagrams have served as the blueprints of a software solution, UML metrics have not yet fully defined, neither explore a possible existent relationship among them, nor to the final software product.

Different from the construction of a house, design metrics for object-oriented software, such as design complexity metrics, have been traditionally measured from the code, and not from the design. Some of the most well known metrics are the Chidamber and Kemerer metrics.

The few works, that have tried to approximate the CK metrics out of UML diagrams, focus mainly on UML class diagrams. 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 [4, 15, 37, 38, 53]. More details concerning this topic can be found in the Chapter of Related Work. Because previous research work has indicated that CBO and RFC metrics are difficult to measure using only UML class diagrams, we present an easy approach to measure these metrics using UML communication diagrams. Moreover, we also provide a metric that approximates the number of methods per class, or WMC considering the method's static complexities to be unity, using communication diagrams.

In this Chapter, we define a set of metrics for objects measured from communication diagrams. Then, using such metrics, we propose a UML approximation of the CBO,

RFC and WMC CK metrics for classes. In order to define these metrics, we followed the basic steps of the Goal/Question/Metric approach (GQM), which is introduced briefly in the following section. Then, we describe our plan; after that, we define our goals and questions; and finally, we provide the definition of our UML metrics. We used the study cases applying the GQM approach provided in [50] as guidance.

3.1 Goal Question Metric Approach

In this Section, we briefly introduce the Goal/Question/Metric approach, according to [50].

During the development of software projects, we often find that developers work in an unstructured and stressful way, resulting in a poor or unknown level of the software quality. Such problems are known as the 'software crisis'. Organizations try to address this crisis by developing software process improvement (SPI) approaches.

SPI approaches consider software process to be the main area for quality improvement, because they contain the activities during which the product is actually created. The earlier quality problems are detected, the easier and cheaper their solution is.

In order to improve the software development process, a definition of clear improvements goals are needed, otherwise the improvement activities will turn out to be as chaotic as the development process itself. These improvements goals should support business objectives in the best possible way.

An organization can define improvement goals by several different ways. For example, they could first prescribe quality objectives, from which improvement goals can be derived. They could also conduct software process assessments in order to identify main areas suitable for improvement, and based on the identification of such areas, improvement goals can again be defined. Measurements also provide the means to identify improvements goals. By applying measurement to a specific part of the process of the project, problems within the process can be defined with respect to particular needs.

SPI practitioners usually focus in four main areas: the decrease of project cost, the decrease of project risk, the shortening of project cycle time and the increase of quality. Several improvement models, methods and techniques are available for such purposes. These can be divided into two major streams: Top-down and Bottom-up approaches.

Top-down approaches are mainly based on assessments and benchmarking, examples of these approaches are the Capability Maturity Models (CMM), the Software Process Improvement and Capability determination model (SPICE) and BOOTSTRAP. Bottom-up approaches mainly measures software development to increase understanding within a specific context. Goal Question Metric (GQM) and the Quality Improvement Paradigm (QIP) are the typical examples of these approaches.

GQM represents a systematic approach for tailoring and integrating goals to models of the software process, products and quality perspectives of interest, based upon the specific needs of the project and the organization.

The principle behind the GQM method is that *measurement should be goal-oriented*. Therefore, in order to improve process, organizations have to define their measurement goals based upon corporate goals and transform these goals into activities that can be

measured during the execution of the project.

GQM defines a certain goal, which then is refined into questions. Then, metrics are defined to provide the answer to these questions. Thus, GQM defines metrics from a top-down perspective and analyzes and interprets the measurement data bottom-up.

The GQM method different phases are:

1. The Planning phase. In this phase a project for measurement application is selected, defined, characterized, and planned, resulting in a project plan.

2. The Definition phase. In this phase the measurement program is defined (goal, questions, metrics, and hypothesis) and documented.

3. The Data Collection phase. During this phase the actual data collection takes place.

The *Definition* phase is of our main interest, since it is where the metrics and their guidelines definition take place. To carry on with such definitions, the following steps need to take place:

- Step 1. Goal Definition

  Measurement goals should be defined in an comprehensible way and should be clearly structured. For this purpose, the following template is available:

Analyze	the object under measurement
For the purpose of	understanding, controlling, or improvement the object
With respect to	the quality focus of the object that the measurement focuses on
From the point of view of	the people that measures the object
In the context of	the environment in which measurement takes place

  We should keep in mind that metrics are meant *to answer* the questions which, altogether, provide the information to achieve certain goal.

- Step 2. List of measurement questions and hypotheses, defined with respect to the measurement goals.

  Questions should be defined to support data interpretation towards a measurement goal. As goals are defined in an abstract level, questions are *refinements of goals* to a more operational level, which is more suitable for interpretation. By answering the question definitions, one should be able to conclude whether the goal is reached. Therefore, during the question definition, various checks are recommended to verify that the defined questions have the ability to support the conclusion of the goal in a satisfactory way.

Subsequently, for each question, expected answers are formulated as hypotheses. Formulating hypotheses triggers the project team to think about the current situation and therefore stimulates a better understanding of the process and/or product. Furthermore, after measurement during data interpretation, these hypotheses of measurement results will be compared with the actual measurement results. The purpose of this comparison should not be to evaluate a possible correctness of the hypotheses, but to encourage the project team to identify and analyze the underlying reasons that caused the actual result to deviate, or conform, from their expectations. In other words, hypotheses are formulated to increase the learning effect from measurement.

- Step 3. Metrics Definition.

  Once goals are defined into a list of questions, metrics should be defined to provide all the quantitative information to answer the questions in a satisfactory way. Therefore, metrics are a refinement of questions into quantitative processes and/or product measurements.

  Furthermore, factors that could possibly influence the outcome of the metrics should also be identified. After all, factors that directly influence metrics, also influence the answer to the questions that metrics are related to.

- Step 4. GQM plan

  A GQM plan is a document that contains the goals, questions, metrics and hypotheses for a measurement program as defined in previous steps. The GQM plan describes the refinement from measurement goals into questions and subsequently from questions into metrics. As some of theses metrics may be indirect metrics, it also describes all direct measurements that should be collected for each indirect metric.

- Step 5. Measurement Plan

  A measurement plan describes the following aspects of each direct measurement that was identified in a GQM plan.

  - It provides formal definitions of direct measurements.
  - It provides textual descriptions of direct measurements.
  - It defines all possible outcomes (values) of the direct measurements.
  - It identifies a person that collects a particular direct measurement.
  - It identifies the particular moment in time when the person should collect the direct measurement.
  - It defines by which means the direct measurement should be collected.

  Furthermore, a measurement plan defines and describes both manual data collection forms and automated data collection tools.

- Step 6. Produce Analysis Plan

  An analysis plan is a document that simulates data interpretation outcomes of the metrics, graphs and tables are presented in this document. such documentation should be related to the questions and goals as defined in the GQM plan.

3.2 Planning

The studies, that previously have provided a formal definition of the CBO and RFC metrics using UML class diagrams, have concluded that UML class diagrams' information is not sufficient to capture all the information that the code normally provides. Therefore, this study focuses on the approximation of these metrics using communication diagrams. The main reason for which we decided to use communication diagrams can be explained in the following paragraphs.

Recalling the definition of the CBO and RFC metrics, we have that:

- CBO 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.

- RFC is a set of all *'methods'* that can be invoked in response to a message to an object of the class. 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.

Both metrics require information on the number of classes or methods which are been instantiated within a given class's body. Class diagrams provide this information to a limited extent. It can tells us, for example, class A uses class B, but cannot tell us precisely that method *ma* of class A uses method *mb* of class B. However, because communication diagrams can tell us more explicitly how the interactions between objects of the different classes of the system take place, we can know which objects call which methods of other objects. Therefore, communication diagrams provide a much more fine-grained information in this sense. For example, suppose that we have two objects *a1* and *b1*, *a1* is an object of class A, and *b1* is an object of class B. When *a1* sends a call message *mb* to b1, *mb* can be translated into a method of class B. So that, by analyzing all communication diagrams of a system design, we can give an approximation of the number of methods of class B. Moreover, not only the number of the different classes instantiated by class A (and B) can be known, but also the number of **methods** of other classes instantiated by class A (and B).

In order to approximate the code CBO, RFC and WMC metrics using communication diagrams, we first provide a general set of metrics measured from communication diagrams, then we derived from them our metrics.

3.3 Definition: Goal

Based on our previous description, we define our goal in Table 3.1. For this purpose, we used the goal template definition provided in [50].

Table 3.1: Our Goal

Analyze	UML Communication Diagrams
For the purpose of	obtaining an approximation of the CK RFC, CBO and WMC metrics
With respect to	their objects interactions
From the point of view of	the researcher
In the context of	software quality studies

3.4 Definition: Questions

The most important activity of the definition phase of the GQM approach is the refinement of the selected goals into questions and metrics. The questions we have defined for our goals are shown in Table 3.2.

Table 3.2: Questions

Q1. How can we approximate the CK-RFC metric?
Q2. How can we approximate the CK-CBO metric?
Q3. How can we approximate the CK-WMC metric?

3.5 Definition: Metrics

In the next paragraphs, we first provide the definition of a set of metrics measured directly from communication diagrams. Then, we provide the definition of a set of metrics for object-oriented classes that can answer our formulated questions in the previous Section. Moreover, details on the how to measure these metrics manually and how they should be interpreted are also provided.

3.5.1 Metrics for UML Communication Diagrams

First, we propose a set of seven metrics listed in Table 3.3, which can be directly measured from UML communication diagrams.

In the following paragraphs, we explain how to measure these *direct* metrics through a practical example. Let us say that we would like to measure these metrics for the

Table 3.3: Direct Metrics for *Objects* in UML Communication Diagrams

Metric		**Definition**
Abbreviation	Stands for	
NSCMsgs	Number of Sent Call Messages	For an object acting as a 'sender', the number of call messages sent to other objects (no actors).
NDSCMsgs	Number of Different Sent Call Messages	For an object acting as a 'sender', the number of *different* call messages sent to other objects.
NDRO	Number of Different Receiver Objects	For an object acting as a 'sender', the number of *different* objects who 'receive' a message from it.
NRCMsgs	Number of Received Call Messages	For an object acting as a 'receiver', the number of call messages received.
NDRCMsgs	Number of Different Received Call Messages	For an object acting as a 'receiver', the number of *different* call messages received.

ATMControl object in the communication diagram of Figure 3.1. According to Table 3.3, the values of such metrics are:

- NSCMsgs equals 4 because *ATMControl* sends call messages 1.3, 2.5, 2.7, and 2.7a to other objects. And NDSCMsgs = NSCMsgs, because there are not repeated messages.
- NRCMsgs equals 2 because *ATMControl* receives call messages 1.2 and 2.4. And NDRCMsgs = NRCMsgs because there are not repeated messages. Notice that message 2.6 is not counted because it is a return value, is not a call message.
- NDRO equals 3 because the different objects that receive messages from *ATMControl* are: *CustomerInterface*, *ATMTransaction* and *BankServer*.

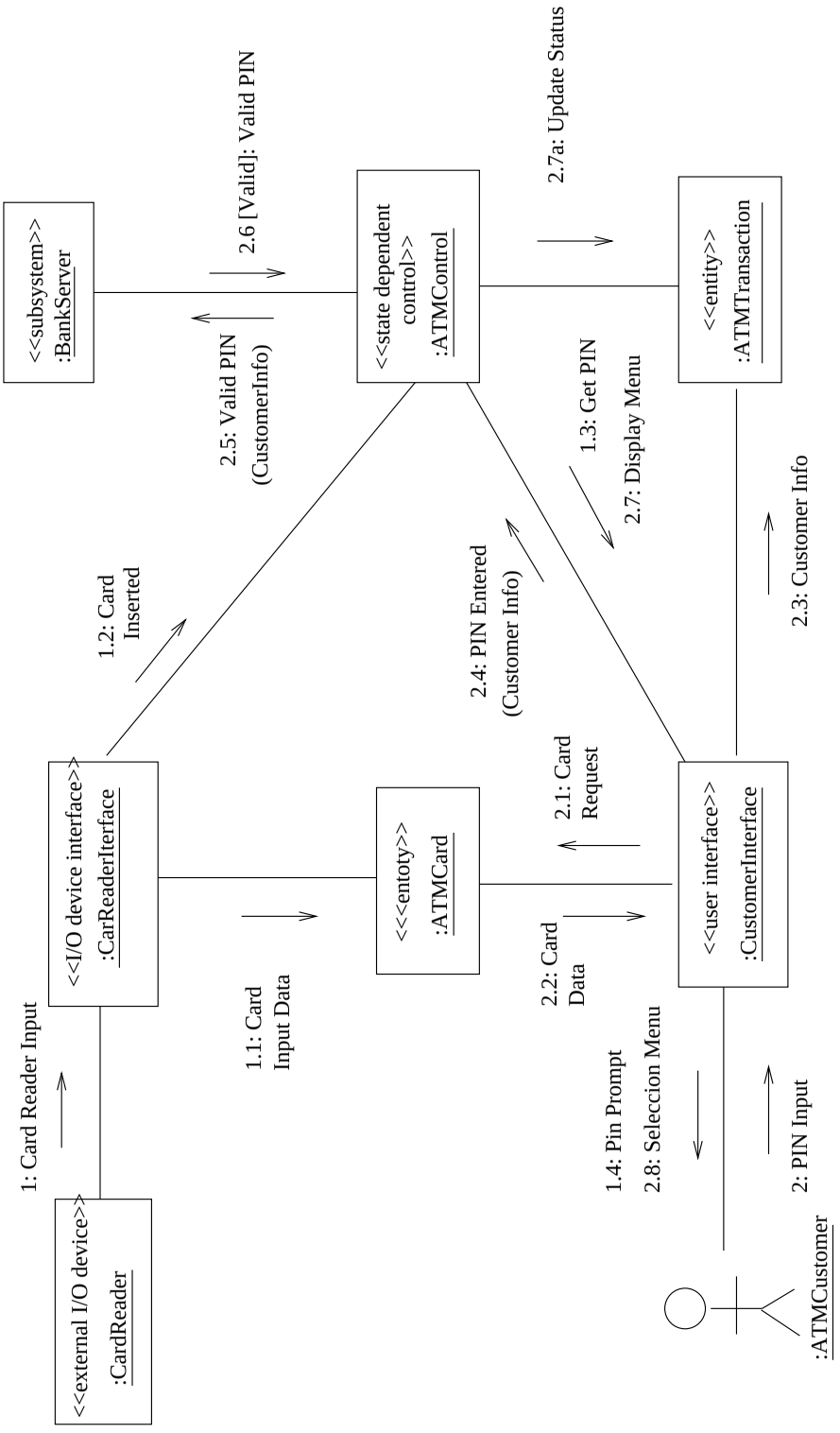

Figure 3.1: Example of a Communication Diagram

3.5.2 Metrics for Classes derived from UML Communication Diagrams

From our previous set of metrics, a new set is derived to be applied to object-oriented classes, these metrics are listed in Table 3.4.

Table 3.4: Metrics for *Classes* from UML Communication Diagrams

Metric	**Definition**
$NIMO$	For a given class X, the Number of Instantiated Methods of Other classes, $NIMO(X)$, is the sum of all $NSCMsgs$ of all objects of class X.
$NDIMO$	For a given class X, the Number of Different Instantiated Methods of Other classes, $NDIMO(X)$, is a count of all different $NDSCMsgs$ of all objects of class X.
NIM	For a given class X, the Number of Instantiated Methods of class X by other classes, $NIM(X)$, is the sum of all $NRCMsgs$ of all objects of class X.
$NDIM$	For a given class X, the Number of Different Instantiated Methods of class X by other classes, $NDIM(X)$, is a count of all different $NDRCMsgs$ of all objects of class X.
$NDRC$	For a given class X, the Number of Different Receiver Classes, $NDRC(X)$, is a count of all different $NDRO$ of all objects of class X.

To illustrate how these metrics are measured and to explain their meaning, we are giving a simple example based on Figure 3.2 and 3.3, which illustrate a *Calculator* application that multiplies and sums two integer numbers. Diagram $'A'$ in Figure 3.2 shows the communication diagram for the multiplication use case of the application, and diagram $'B'$, in the same Figure, represents the communication diagram for the addition use case of the application. Figure 3.2, in its turn, shows the code of the classes of the application: *Calculator*, *Multiplier*, *Adder* and *Display*.

The metrics for the classes of our application are shown in Table 3.5, 3.6, 3.7 and 3.8. We first measure the direct metrics from all instances of our classes in both communication diagrams of Figure 3.2, according to Table 3.3. Then, totals per class are calculated according to definitions provided in Table 3.4. The summary of the metrics for the classes of our application *Calculator* are shown in Table 3.9. From these results, the following observations can be made:

- $NIMO$, $NDIMO$ and $NDRC$ **represent a degree of dependency a given class has to others**. In other words, they represent a degree of coupling, $NIMO$ and $NDIMO$ measure the number of instances to other classes' methods by a given class. And $NDRC$ is also a measure of coupling but in terms of classes, because

it is based on counts of all the different objects to which a given class is coupled. Therefore, its measure can approximate the code CK CBO metric.

In our examples, class *Calculator* has a total of $NIMO = 4$, $NDIMO = 3$ and $NDRC = 3$, which is reflected in its code, where CK CBO = 3 coupled to class *Display*, *Adder* and *Multiplier*. So that, unless all of these classes work properly, class *Calculator* will not. Notice that message $A5$ of diagram $'A'$ and message $B4$ of diagram $'B'$ are not counted because they are sent to an actor and not to an object.

The same kind of observations can be made for class Multiplier. In the case of classes *Adder* and *Display*, $NIMO = NDIMO = NDRC = 0$, indicating that they do not depend on any other class, which is clearly reflected in their code.

- NIM represents **the total number of method instances**, of a given class, made by other classes' objects at some point in the execution of the application.

  In our examples, classes *Calculator*, *Adder* and *Display* have a $NIM = 2$, while class *Multiplier* has a $NIM = 1$, which indicate that class *Multiplier* is less instantiated than the other two classes. This also can can be observed in their code.

- $NDIM$ can approximate the **number of the different (public) methods** of a given class, because it measures the different methods, of a given class, instantiated by other classes' objects.

  In our example, we can observe from the code of the *Calculator*, *Multiplier*, *Adder* and *Display* classes that their $NDIM$ values equal the number of their methods (2,1,1 and 1 respectively).

From our previous observations, we can draw the following hypothesis:

- *Hypotheses 1: For a given class X, a UML approximation of its CK WMC measure, considering the method's static complexities to be unity, can be obtained using Equation 3.1.*

$$UWMC(X) = NDIM(X); \tag{3.1}$$

- *Hypotheses 2: For a given class X, a UML approximation of its CK CBO measure can be obtained using Equation 3.2.*

$$UCBO(X) = NDRC(X); \tag{3.2}$$

- *Hypotheses 3: Since $NDIM$ can approximate the number of methods of a given class X, and $NDIMO$ the number of other classes' methods instantiated by X, we can have an approximation of X's code RFC measure using Equation 3.3.*

$$URFC(X) = NDIM(X) + NDIMO(X); \tag{3.3}$$

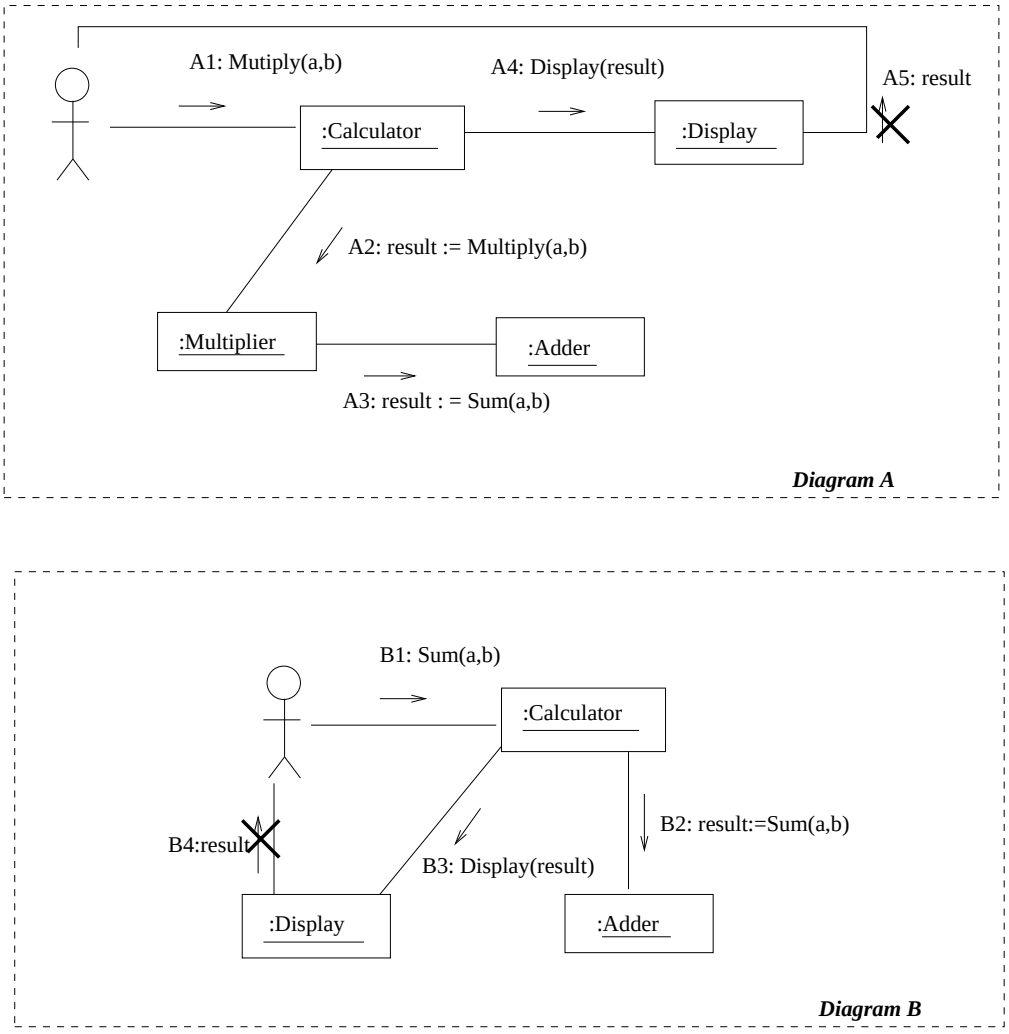

Figure 3.2: UML Communication Diagrams for the Calculator Application

Table 3.5: Metrics for Class *Calculator*

Diagram	NSCMsgs	NDSCMsgs	NRCMsgs	NDRCMsgs	NRO
A	$2 =$ $\{A2, A4\}$	$2 =$ $\{A2, A4\}$	$1 =$ $\{A1\}$	$1 =$ $\{A1\}$	$2 =$ $\{:D, :M\}$
B	$2 =$ $\{B1, B3\}$	$2 =$ $\{B1, B3\}$	$1 =$ $\{B1\}$	$1 =$ $\{B1\}$	$2 =$ $\{:D, :A\}$
Total	$4 =$ $\{A2, A4, B1, B3\}$ $= NIMO$	$3 =$ $\{A2, A4 = B3, B1\}$ $= NDIMO$	$2 =$ $\{A1, B1\}$ $= NIM$	$2 =$ $\{A1, B1\}$ $= NDIM$	$3 =$ $\{:D, :M, :A\}$ $= NDRC$
:D is a Display object, :A is a Adder object, :M is a Multiplier object					

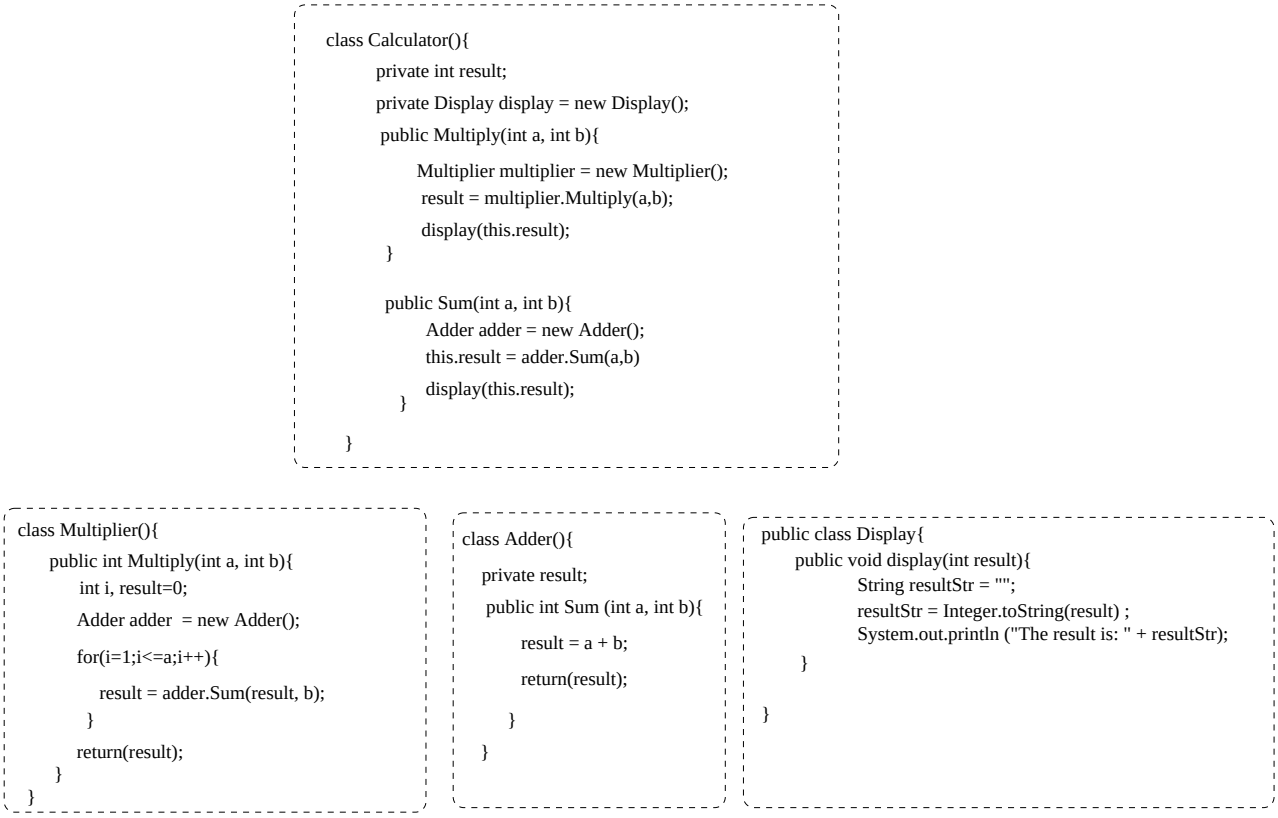

Figure 3.3: Code of the Calculator Application

Table 3.6: Metrics for Class *Adder*

Diagram	NSCMsgs	NDSCMsgs	NRCMsgs	NDRCMsgs	NRO
A	$0 =$ $\{\}$	$0 =$ $\{\}$	$1 =$ $\{A3\}$	$1 =$ $\{A3\}$	$0 =$ $\{\}$
B	$0 =$ $\{\}$	$0 =$ $\{\}$	$1 =$ $\{B2\}$	$1 =$ $\{B2\}$	$0 =$ $\{\}$
Total	$0 =$ $\{\}$	$0 =$ $\{\}$	$2 =$ $\{A3, B2\}$	$1 =$ $\{A3 = B2\}$	$0 =$ $\{\}$
	$= NIMO$	$= NDIMO$	$= NIM$	$= NDIM$	$= NDRC$
:D is a Display object, :A is a Adder object, :M is a Multiplier object					

Table 3.7: Metrics for Class *Multiplier*

Diagram	NSCMsgs	NDSCMsgs	NRCMsgs	NDRCMsgs	NRO
A	$1 =$ $\{A3\}$	$1 =$ $\{A3\}$	$1 =$ $\{A2\}$	$1 =$ $\{A2\}$	$1 =$ $\{: A\}$
B	-	-	-	-	-
Total	$1 =$ $\{A3\}$ $= NIMO$	$1 =$ $\{A3\}$ $= NDIMO$	$1 =$ $\{A2\}$ $= NIM$	$1 =$ $\{A2\}$ $= NDIM$	$1 =$ $\{: A\}$ $= NDRC$
:D is a Display object, :A is a Adder object, :M is a Multiplier object					

Table 3.8: Metrics for Class *Display*

Diagram	NSCMsgs	NDSCMsgs	NRCMsgs	NDRCMsgs	NRO
A	$0 =$ $\{\}$	$0 =$ $\{\}$	$1 =$ $\{A4\}$	$1 =$ $\{A4\}$	$0 =$ $\{\}$
B	$0 =$ $\{\}$	$0 =$ $\{\}$	$1 =$ $\{B3\}$	$1 =$ $\{B3\}$	$0 =$ $\{\}$
Total	$0 =$ $\{\}$ $= NIMO$	$0 =$ $\{\}$ $= NDIMO$	$2 =$ $\{A4, B3\}$ $= NIM$	$1 =$ $\{A4 = B3\}$ $= NDIM$	$0 =$ $\{\}$ $= NDRC$
:D is a Display Object, :A is a Adder Object, :M is a Multiplier object					

Table 3.9: Summary of Metrics for Calculator Application

Class	NIMO	NDIMO	NIM	NDIM	NDRC
Calculator	4	3	2	2	3
Adder	0	0	2	1	0
Multiplier	1	1	1	1	1
Display	0	0	2	1	0

Table 3.10: Summary of Metrics for Calculator Application: UML CK Metrics

Class	UCBO	URFC	UWMC
Calculator	3	5	2
Adder	0	1	1
Multiplier	1	2	1
Display	0	1	1

3.6 Summary

To sum up the definition of our goals and metrics, we provide the answer of our questions defined in previous Sections in terms of the our proposed metrics. This can be found in Table 3.11.

Table 3.11: Metrics Definition based on the GQM Approach

Goal / Questions / Metrics
Analyze *UML Communication Diagrams* For the purpose of *obtaining an approximation of the CK RFC, CBO and WMC metrics* With respect to *their objects interactions* From the point of view of *the researcher* In the context of *software quality studies*
Q1. How can we approximate the CK-RFC metric? Using the metric URFC (Equation 3.3).
Q2. How can we approximate the CK-CBO metric? Using the metric UCBO (Equation 3.2).
Q3. How can we approximate the CK-WMC metric? Using the metric UWMC (Equation 3.1).

Chapter 4

UML CK Metrics Evaluation

The contents of this Chapter are related to the evaluation of our UML RFC, CBO and WMC metrics as approximations to their respective code metrics. To perform such an evaluation, we first measure the CBO, RFC and WMC CK metrics from the code of three different small-size software projects written in Java. Then, we measure the same metrics from their UML communication diagrams (called collaboration diagrams in previous UML versions to the 2.0 version). Such measurements are performed following the procedures provided in the previous Chapter. Finally, we compare both sets of measures, UML and code, and evaluate the goodness of our approximation by calculating the approximation error of our UML metrics to their respective code metrics.

4.1 Data Collection

The four small-size software projects used for our evaluation were developed by different teams of students of JAIST during a 3-month lecture. All of the four projects were 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], in fact, it is supposed that every group developed a more complete design before proceeding with the implementation. The four projects are:

- Two banking systems (BNA and BNB), which were developed by different student groups, following the same basic design (BNS).
- A cruise control and monitoring system (CRS).
- An e-commerce system (ECS).

In order to evaluate the goodness of our UML metrics as approximations to the code WMC, CBO and RFC metrics, the following procedure was undertaken:

1. UML measures. We first calculated the UML CBO, RFC and WMC metrics per class of every of the three different systems using the UML communication diagrams provided in [23]. This calculation was performed according to the procedure described in Chapter 3, and although, the UML diagrams provided in [23] are not referred as communication diagrams but collaboration diagrams instead, the same procedures to measure our UML CK metrics can be applied.

2. Code measures. The different teams of students used CVS repositories to keep record of their implementations. CBO, RFC and WMC measures per class were collected from the last version of the four systems from their respective CVS repository. Such measurement was carried out using the CKJM tool, which calculates CK metrics by processing the byte-code of compiled Java files [1].

3. Evaluation. Finally, we compared both resulting sets of measures by using a simple line graph, measuring the average relative errors of our approximations, and calculating the correlation coefficients between UML and code measures to determine whether it exists a possible linear relationship between them. The numbers of classes analyzed were 11, 10, 16 and 10 from the BNA, BNB, CRS and ECS respectively.

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 UML diagrams provided in Gomaa's book, which were used by the students to elaborate a more complete design to finally implement the systems.

4.2 Measures for our Evaluation

4.2.1 Error Approximation

The approximation error of some data refers to the discrepancy between an exact value and some approximation to it. An approximation error can occur either because the measurement of the data is not precise, due to the instruments used to carry out the measurement, or because approximations are used instead of the real data, which is our case.

Given an approximation *aprox* of a correct value x, the absolute error is defined as the absolute value of the difference between the approximation and the correct value, which is expressed by Equation 4.1.

$$Error_{abs} = |x - aprox| \tag{4.1}$$

There are two problems with using the absolute error:

- Significance. The absolute error can tell us the size of the error, but cannot tell us how significant this error is. For example, if we had an absolute error of 3.52, this could not be considered significant, if the correct value is x = 5030235.23, however

[1] CKJM was written by Diomidis Spinellis and can be found in http://www.spinellis.gr/sw/ckjm/

if the correct value is x = 5.03023523, then our absolute error probably could be very significant.

- Units. The absolute error will change depending on the units used, which can be troublesome if we want to use the absolute error for comparison. For example, the absolute error of the approximation, 2.4 MV, of an actual voltage of 2.57 MV, is *0.17 MV*, whereas the absolute error of the approximation, 2400000 V, to an actual voltage of 2573000 V is *173000 V*. Having both results, we cannot tell which error is more or less significant than the other.

To solve the problem of significance and units, the relative error is used. This is defined as the ratio between the absolute error and the absolute value of the correct value, see Equation 4.2.

$$Error_{rel} = \frac{|x - approx|}{|x|} \tag{4.2}$$

In this equation, any units are canceled, so the relative errors of the approximations 2.4 MV and 2400000 V versus the actual voltages of 2.57 MV and 2573000 V, respectively, are equal. Also, a relative error of 0.01 means that the approximation is correct to within one part in one hundred, regardless of the size of the actual value.

For our evaluation, relative errors were calculated. In order to avoid divisions by zero, measurement observations in which the true value (x in Equation 4.2) resulted to be zero were excluded from our evaluation.

4.2.2 Correlation

As it is well known in probability and statistics, correlation indicates the strength and direction of a linear relationship between two random variables. In general statistical usage, correlation or co-relation refers to the departure of two variables from independence.

A number of different coefficients are used for different situations. The most well known is the Pearson correlation coefficient, which is obtained according to the following formula:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum^{n}{}_{i=1}(y_i - \bar{y})^2}} \tag{4.3}$$

Where r is the Pearson coefficient based on a sample of paired data (x_i, y_i), and n is the size of the sample.

Statistical Significance of Correlation Coefficients

Within the context of correlation, the question of statistical significance concerns the relationship between r, which is the correlation that is observed within a limited sample of pairs of data and the existent correlation ρ in the larger reality beyond the sample, we explain further in the following paragraphs[2].

[2]Online textbook of "Concepts and Applications of Inferential Statistics" (http://faculty.vassar.edu/lowry/webtext.html)

A sample is a relatively very small window through which the investigator glimpses the outlines of some larger reality. In some cases, that glimpse might truly represent the larger reality, in other cases it might misrepresent it. There is always the possibility that the observed facts in a sample result from nothing but from mere chance. *Statistical significance* is the logical and mathematical apparatus by which the investigator can assess whether the yielded correlation coefficient is not just a fluke of mere chance coincidence.

The criterion for statistical significance is conventionally set at the 5% level. That is, an observed result is regarded as statistically significant (as something more than a mere fluke) only if it had 5% or smaller likelihood of occurring by mere chance coincidence.

Table 4.1 shows the positive and negative values of r that are required for statistical significance at the 5% level for various sample sizes N. For example, for any particular sample size, an observed value r is regarded as statically significant at the 5% level if and only if its distance from zero is equal to our greater than the distance of the value of r in Table 4.1. This, for a sample of size N=20, an observed value of $r = +0.4$ or $r = -0.4$ would be significant at the 5% level for a directional hypothesis, but non-significant for a non-directional hypothesis.

Within the context of correlation, a directional hypothesis is one that leads the investigator to specify, in advance, one or the other of the following expectations:

- Positive directional hypothesis: The relationship between two variables X and Y in the general population is positive, the more of X the more of Y, hence a particular sample of pairs (x_i, y_i) will show a positive correlation.
- Negative directional hypothesis: The relationship between two variables X and Y in the general population is negative, the more of X the less of Y, hence a particular sample of pairs (x_i, y_i) will show a negative correlation.

A non-directional hypothesis, on the other hand, leads only to the expectation that the correlation between X and Y within the general population might be something other than zero, hence a particular sample of pairs (x_i, y_i) will show a non-zero correlation either positive or negative, though we have no basis for predicting which of these it will be.

Table 4.1: Correlation Coefficients Required for Statistical Significance at 5% Level

N	**Directional** ($\pm r$)	**Non-Directional** ($\pm r$)
5	0.81	0.88
10	0.55	0.63
16	0.43	0.50
20	0.38	0.44
27	0.32	0.38
32	0.30	0.35

In our study, in order to determine whether the obtained correlation coefficients between our UML approximation and code metrics can be regarded as statistically significant, and unlikely to occur by mere chance, we used as a threshold the conventional 5% level (p-value $<= 0.05$) considering a non-directional hypothesis.

4.2.3 Precision and Accuracy

Precision and Accuracy are often used in the evaluation of methods or instruments of measurement. Ideally a measurement device must be both accurate and precise.

Precision

Precision is defined as the degree of refinement in the method used to make a measurement, and it can be observed in the closeness to each other of the data observations of a set of measurements. The closer the observations are to each other, the more precise the method of measurement is considered. Precision of a measurement system is also called reproducibility or repeatability, referring to the degree to which repeated measurements under unchanged conditions show the same results.

Precision can be usually discussed in terms of standard deviation (σ for the entire population or s for a given sample). Assuming that the distribution of the data under observation is a normal distribution, the observer can say that 68% of the results fall into the range of the data mean $\pm 1s$, or 95% of them fall into the range of the data mean $\pm 2s$. The less the different observations vary from their mean or expected value, the higher the precision is.

Accuracy

The accuracy of a measurement is the degree of agreement or conformity with its true value. It refers to the degree of perfection obtained in measurements.

In mathematics, the accuracy of a measure, when this is an approximation, is expressed in terms of its relative error, previously introduced. The smaller the relative error, the more accurate the measure is.

Precision, Accuracy and Errors

The theory of errors and measurement assumes that any measurement is composed of its true value plus some error value, which in its turn is composed of two types errors:

- Random. Such type of error in measurement lead to measurable values being inconsistent when repeated measures of a constant attribute or quantity are taken.

  Random errors are caused by unpredictable fluctuations in the readings of a measurement apparatus, or in the experimenter's interpretation of the instrumental reading; these fluctuations may be in part due to interference of the environment with the measurement process. They are caused by factors beyond the control of the observer. The rules of random errors are:

  - Small errors occur more frequently than large ones.
  - Both positive and negative errors are equally likely.

- Systematic or Bias. This type of error is considered to be predictable, and typically *constant or proportional* to the true value.

  Systematic errors are caused by imperfect calibration of measurement instruments or imperfect methods of observation, or interference of the environment with the measurement process, and always affect the results of an experiment in a predictable direction. They are cumulative, and thus their effect can be computed and corrections can be applied.

There is a definite relationship between precision and accuracy of a measurement system, and random and systematic errors:

- The lower the number of random errors is, the higher the precision.
- The lower the number of systematic errors, the higher the accuracy.

This is illustrated in Figure 4.1, where accuracy is represented by the proximity of measurement results to the true value, and precision is represented by the spread of the data under the curve.

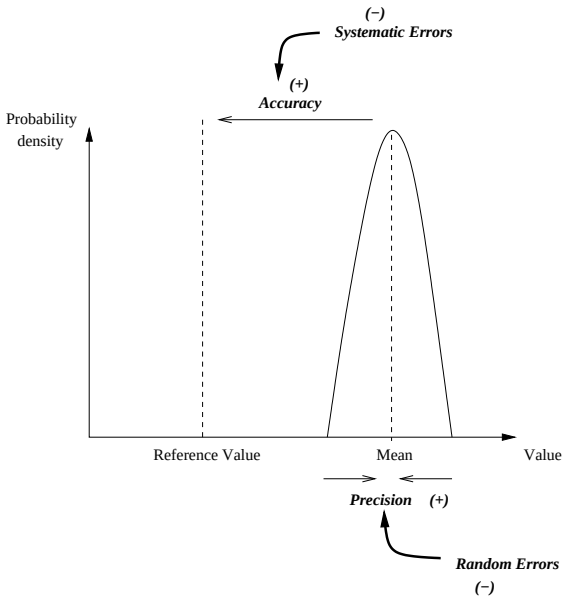

Figure 4.1: RAW RFC measures

4.3 RFC

The graph in Figure 4.2 shows the raw RFC measures of the BNA, BNB, CRS and ECS software projects. The line with squares represents RFC measured from UML diagrams, and the line with circles represents RFC measured from the code. As we can observe, there is a gap between the UML measures and the code measures, which is apparently larger in the BNA project. Extreme deviations, such as the code of the class *BNA-05*, occur mainly because the number of calls to other classes' methods exceeds that one estimated using UML communication diagrams at design time, more details are given in a later section.

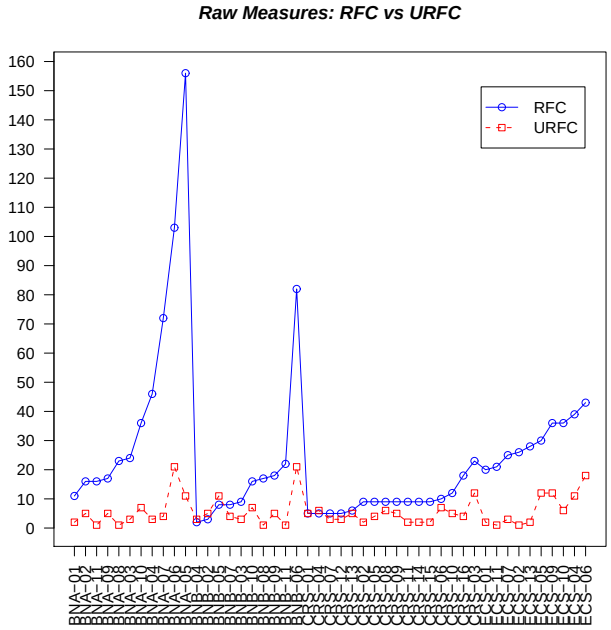

Figure 4.2: RAW RFC measures

Accuracy and Precision

The average relative errors of our UML RFC approximations to their code RFC measures are 0.85, 0.66, 0.48 and 0.8 for the BNA, BNB, CRS and ECS respectively.

Moreover, the standard deviations of our relative errors are 0.1, 0.19, 0.25 and 0.15, for the BNA, BNB, CRS and ECS respectively. Assuming a normal distribution of our relative errors, we can say that 68% of them fall into the following ranges: 0.85 ± 0.1 , 0.66 ± 0.19, 0.48 ± 0.25 and 0.8 ± 0.15 for the BNA, BNB, CRS and ECS projects respectively.

Considering a threshold of 0.5 for our average relative errors and 0.25 for our standard deviations to evaluate accuracy and precision, we can say that our methodology to approximate RFC to its code is more precise than accurate. Thereafter, the number of random errors should be lower than the number of systematic errors.

Linear Relationship

Moreover, the correlation coefficients between the code and UML measures tell us that there is a strong linear relationship between them. The correlation coefficients are 0.7, 0.8, 0.57 and 0.86 for the BNA, BNB, CRS and ECS projects respectively. These correlation coefficients resulted to be statically significant to a minor p-value than the one used conventionally; the obtained p-values are 0.016, 0.005, 0.02 and 0.0014 for the BNA, BNB, CRS and ECS projects respectively, suggesting that the three correlation coefficients are unlikely to occur by mere chance.

4.4 CBO

The graph in Figure 4.3 shows the raw CBO measures of the BNA, BNB, CRS and ECS software projects. The line with squares represents CBO measured from UML diagrams, and the line with circles represents CBO measured from the code. As we can observe, the gap between the UML and the code measures is not as large as the one found between RFC measures. In fact, the average relative errors obtained for our CBO measures are smaller than those ones obtained for our RFC approximations.

Accuracy and Precision

The obtained average relative errors are 0.52, 0.56, 0.55 and 0.49 for the BNA, BNB, CRS and ECS respectively.

Moreover, the standard deviations of our relative errors are 0.37, 0.33, 0.41 and 0.32, for the BNA, BNB, CRS and ECS respectively. Assuming a normal distribution of our relative errors, it can be said that 68% of them fall into the range of 0.52 ± 0.37 ,0.56 ± 0.33, 0.55 ± 0.41 and 0.49 ± 0.32 for the BNA, BNB, CRS and ECS projects respectively.

Considering the same threshold values for evaluating precision and accuracy used previously, we can say that our methodology to measure CBO is more accurate than precise; and less inaccurate than our methodology to approximate RFC.

Notice that the number of classes used for this analysis changed due to CBO values equal to zero, the number of classes used were 6, 6, 15, 6 for the BNA, BNB, CRS and ECS projects respectively.

Linear Relationship

On the other hand, the correlation coefficients between our UML and code measures tell us that there exists a stronger linear relationship between them. The correlation coefficients are 0.85, 0.87. 0.56 and 0.92 for the BNA, BNB, CRS, and ECS projects respectively. The obtained p-values are 0.0009, 0.001, 0.024 and 0.0016 for the BNA, BNB, CRS and ECS projects respectively, which are smaller to the conventional p-value used, suggesting that the three correlation coefficients are statistically significant and unlikely to occur by mere chance[3].

[3]This results are slightly different to those published in [11] due to a higher number of classes analyzed.

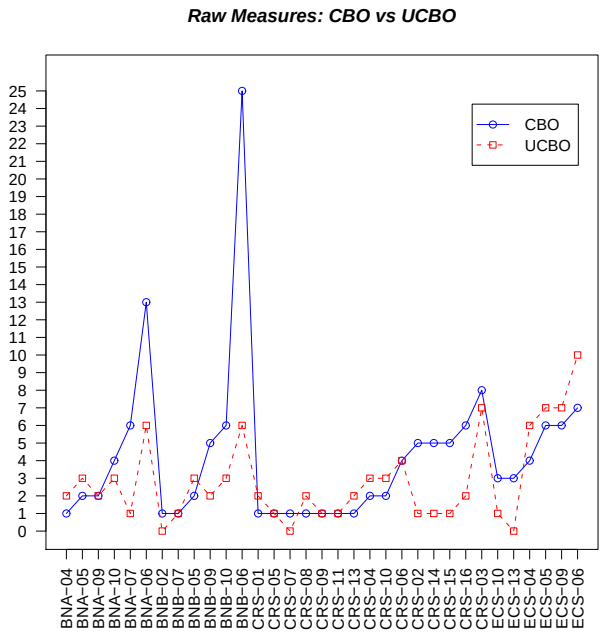

Figure 4.3: RAW CBO measures

4.5 WMC

The graph in Figure 4.4 shows the raw WMC measures of the BNA, BNS, CRS and ECS software projects. The line with squares represents WMC measured from UML diagrams, and the line with circles represents WMC measured 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.

Accuracy and Precision

The average relative errors of our approximations are 0.69, 0.56, 0.43 and 0.94 for the BNA, BNB, CRS and ECS respectively. Considering a threshold of 0.5 for our average relative errors, only our measures for the CRS data were accurate.

As for precision, the standard deviations of our relative errors are 0.15, 0.41, 0.26 and 0.79, for the BNA, BNB, CRS and ECS respectively. Assuming a normal distribution of our relative errors, we can say that 68% of them fall into the range of 0.69 ± 0.15, 0.56 ± 0.41, 0.43 ± 0.26 and 0.94 ± 0.79 for the BNA, BNB, CRS and ECS projects respectively. Considering a threshold of 0.25 for our standard deviations, only the approximations of WMC for the BNS project can be regarded as precise.

From the previous resulting measures, the only conclusion we can draw so far, is that our methodology to approximate WMC is much more inaccurate and imprecise than our two previous methodologies to approximate RFC and CBO.

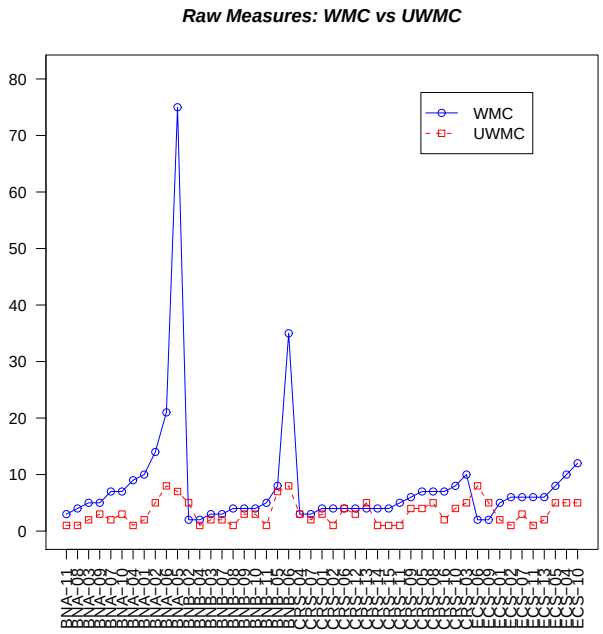

Figure 4.4: RAW WMC measures

Linear Relationship

On the other hand, the correlation coefficients between the UML and code measures of the BNA, BNB, CRS and ECS projects are 0.69, 0.71, 0.51 and -0.08 respectively. The obtained p-values are 0.01, 0.02, 0.04 and 0.82 for the BNS, CRS and ECS projects respectively, suggesting that there exists a linear relationship between code and UML WMC measures, except for the ECS measures[4].

4.6 The Gap

In this section, we try to analyze some of the possible causes originating the difference of values between our UML and code measures of our different study cases, such causes are:

1. Detail Level of the Design. Since two different designers can provide different levels of detail in their design, even if their designs corresponds to the same system, measuring UML metrics from such designs would yield different gaps between the resulting design measures and their respective code measures taken from their actual implementation.

   For our specific study cases, we consider that the same level of detail was used in the different communication diagrams of the design of the BNS, CRS and ECS. Therefore, under this assumption, our UML measures of all classes can be considered to

[4]Our previous results published in [11] indicated no linear relationship for the BNA due to a smaller number of classes analyzed

be equally proportional to their respective code metrics. This is illustrated in Figure 4.5, where the horizontal axis represents the different classes of a given system, and the vertical axis represents CBO; the dashed line represents the different UML CBO measures and continue line represents their respective code measures.

2. Software type. Depending on which kind of application the software is, it may require the usage of different technologies in different degrees, such as GUIs, database connectivity, IPC communication, etc.

   This factor may cause a proportional difference of values between code and UML measures in different groups of classes of the system. For example, consider two different calculator systems, A and B, which were implemented based on the same design having exactly the same core functionalities, such as adding, multiplication, etc., with only one difference, which is that system A uses GUIs and system B uses a command line interface. Now, let us say that we would like to measure CBO, which is a coupling measure. In general, classes interacting with end users (*eu-classes*) of system A may have a higher degree of coupling than those *eu-classes* of system B because of the usage of GUI functionalities. What could be the results of measuring code and UML CBO measures from both systems? There can be two possibilities:

   (a) If GUI functionalities were not decoupled from the main domain classes considered in the design of the system, we may find that although system A is using GUIs and system B is not, their respective code measures may be proportional to their common UML metrics. In principle, the group of main domain classes of both systems, not implementing GUIs, would have almost the same code CBO values. The code CBO measures of the groups of classes *eu-classes*, on the other hand, may be different, but still proportional to their design. Such an imaginary situation is represented in Figure 4.6, where the horizontal axis represents the different classes (*eu-class-i* represents end-user classes and *class-i* other classes), and the vertical axis measures CBO. The line graph with circles represents the UML CBO measures, the line graph with triangles represents the code CBO of system A, and the line graph with squares the code CBO of system B.

   (b) If GUI functionalities were decoupled from the main domain classes considered in the design of the system into different classes, we may find that code CBO measures of both systems A and B are almost the same, and therefore both sets of CBO measures would be equally proportional to their common UML measures.

3. Developer programming style. It refers to the particular programming style of a developer. Software systems developed by more than one developer can affect in different degrees the difference of values between UML and code measures to different groups of classes of a system. The same degree of influence over all classes of a system can be expected, if only one developer implements all of the classes of a system, or if all the developers, involved in the implementation of the system, have more or less the same programming style.

4. Inconsistent code. This occurs when a developer decides to implement a class or classes changing their original design. Such factor is expected to be sporadic. However, if it occurs, then the degree in which influences the difference of values between UML and code measures is not equal, neither proportional, for all of the classes of a given system.

The gap or errors produced by factors 1 and 2 could be classified as systematic type error or bias, because both factors are assumed to be somehow proportional. On the other hand, errors produced by factors 3 and 4 are less predictable, therefore we could classified them as random type of errors.

In order to have a better understanding of such a gap, we compared the RFC, CBO and WMC measures of the two different implementations BNA and BNB of the BNS in Figures 4.7, 4.8, 4.9. In these figures the horizontal axis corresponds to the different designed classes of the BNS and vertical axis to measures CBO. The line with circles corresponds to the UML measures, the line with squares to the BNA implementation and the line with diamonds to the BNB implementation. BNA tends to have higher values of RFC and WMC, which are metrics given in terms of methods. Our first hypothesis was that the difference between the two implementations was due to factor 3, different programming styles among developers can produce different CK measures. Inspecting the code of the designed classes *BNS-04* and *BNS-05*, of both projects BNA and BNB, we found that:

- BNA-04 and BNA-05 have much higher values of RFC and WMC than those of the design (BNS-04 and BNS-05) due to the usage of GUIs functionalities.
- BNB was implemented different from the original design. Designed classes BNS-04 and BNS-05 were implemented as interfaces in BNA-04 and BNA-05. The class that actually implements these two interfaces is BNB-00 (not considered in the original design).
- BNB-00 CK measures still much lower than the CK measures of BNA-04 and BNA-05. This is due to the manner GUI functionalities are handled. BNB-00 (different to BNA classes) decouples the GUI functionalities from those specific to the class, which is achieved by defining an inner class to handle all GUIs. BNA classes does not make this decoupling.
- Although BNB implementation is inconsistent with the design, it handles GUIs in a smarter manner than BNA does.

Therefore, from these observation we can say that the difference between the two sets of measures of the two different implementations, BNA and BNB, is caused mainly due to factor 4, the original design was changed in BNB. Factor 3 may also contributing to this difference, however to assess accurately how much is the effect of this factor is not possible with the information we currently count with. So far, we can only say that despite the two implementations were developed by different groups of developers, factor 3 is not causing a large difference between the measures of both implementations.

Some other factors may be influencing the difference of values between UML and code measures. To identify all of them and the degree in which each of them contributes to such a difference is a troublesome task. In such situations, what statisticians do is to target systematic errors and identify some feature which correlates to this type of error. Moreover, in a broader manner, the correction of systematic errors is also targeted by applying data normalization methods. This kind of approach has been widely practised in other research fields such as bio-informatics for micro-array data analysis and image processing for feature normalization [2, 8].

For the treatment of random errors, which remain after mistakes and systematic errors have been eliminated, application of adjustment techniques to obtain adjust observations that fit a given math model can be applied.

Our work has been delimited and focused to decrease the systematic error of our approximations using data normalization methods, independently of their nature.

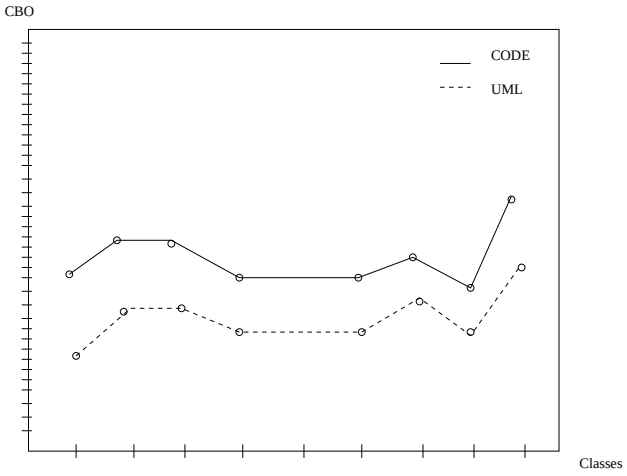

Figure 4.5: Design Level of Detail Effect

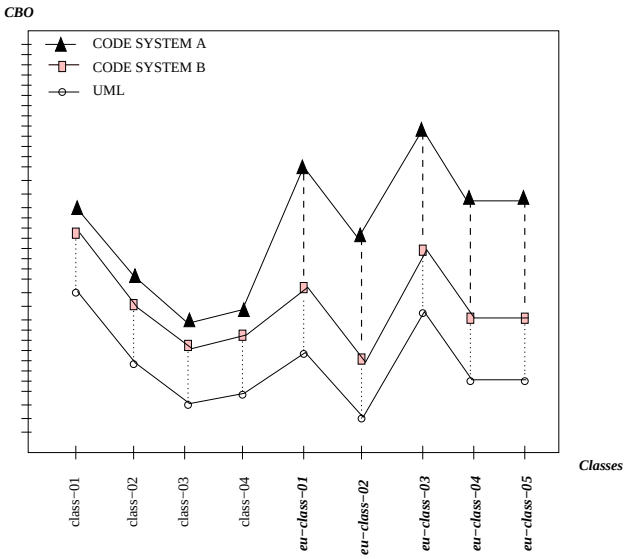

Figure 4.6: Software Type Effect

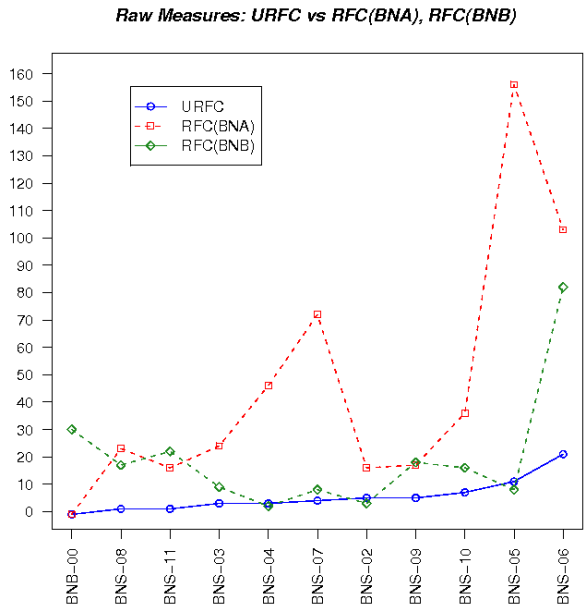

Figure 4.7: RFC: Same design, two different implementations

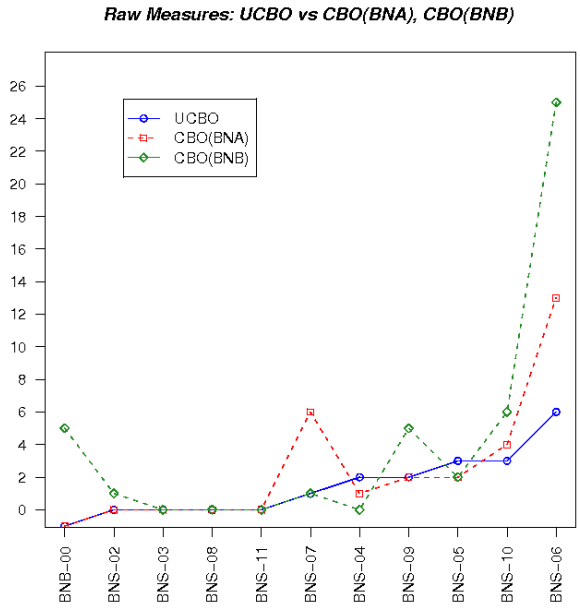

Figure 4.8: CBO: Same design, two different implementations

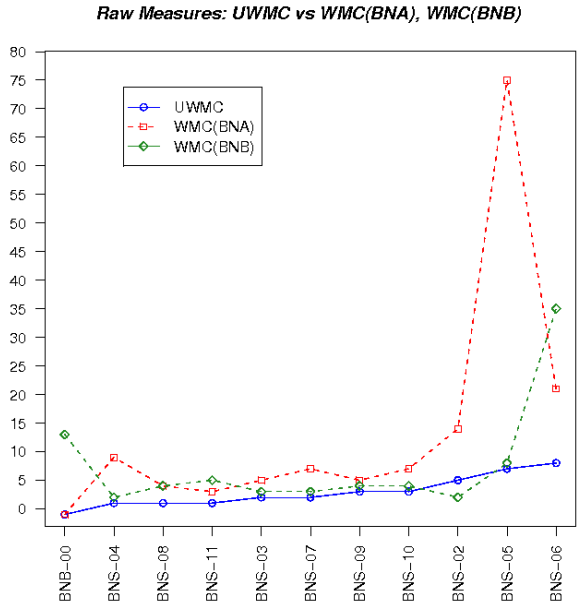

Figure 4.9: CBO: Same design, two different implementations

4.7 Conclusion

Table 4.2 summarizes the results of our evaluation, it shows the average relative errors of our approximations and their corresponding standard deviations, as well as the correlation coefficients between UML and code measures. Based on these measures, we evaluate accuracy and precision of our UML metrics, and linear relationship to their respective code measures. If we suppose a normal distribution of the resulting relative errors, using average values, the curves of their distributions would take the form of the curves shown in Figure 4.10, which can be interpreted as follows.

Considering a 0.5 threshold for our average relative errors and 0.25 for our standard deviations, our UML CBO metric results to be the least inaccurate, and our UML RFC, the most precise. Therefore, it can be said that the impact of random type of errors on the gap between our UML and code measures is less harmful for RFC than for CBO, and the impact of systematic type of errors is stronger for our RFC approximations than for our CBO approximations. This can be explained due to the units used for measuring CBO and RFC. Since CBO is given in terms of classes and RFC in terms of methods, it is somehow understanding that our UML approximation of RFC results to be less accurate than our UML approximation for CBO.

Furthermore, a strong linear relationship between code and UML measures was found, for both of our UML CBO and RFC approximations in all projects.

As for our UML WMC metric, the following conclusions can be made. Because of the resulting error approximations of our UML WMC measures to their code measures in all classes analyzed were highly inaccurate and imprecise, we conclude that the number of designed methods per class is still far from the actual number of the implemented methods, at least at the stage the UML communication diagrams of our projects were designed.

Last, we mentioned that to identify the true sources of our approximation errors and their contribution to the overall error is a troublesome task. Therefore, our work has been delimited and focused to reduce simply the systematic type of error of our approximation, independently of their nature, using data normalization methods.

Table 4.2: Summary of Evaluation

System	*Average* (*RE*)	*SD* (*RE*)	*Correlation* (*code*, *UML*)	*Evaluation*		
				Accuracy	Precision	Linearity
RFC Approximation						
BNA	0.85	0.1	0.7		+++	++
BNB	0.6	0.19	0.8		++	++
CRS	0.48	0.25	0.57		+	+
ECS	0.8	0.15	0.86		++	++
Average	**0.7**	**0.17**	**0.73**		+	+
CBO Approximation						
BNA	0.5	0.37	0.85	+		+++
BNB	0.6	0.33	0.87			+++
CRS	0.5	0.41	0.56	+		+
ECS	0.49	0.32	0.92	+		+++
Average	**0.5**	**0.36**	**0.8**	+		++
WMC Approximation						
BNA	0.69	0.15	0.69		++	+
BNB	0.56	0.41	0.71			++
CRS	0.43	0.26	0.51	+		+
ECS	0.94	0.79	-0.08			
Average	**0.65**	**0.4**	**0.46**			

+:Fair, ++: Good, +++: Very Good

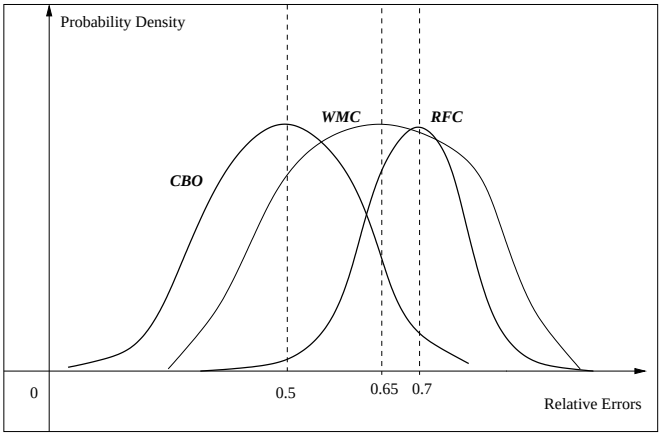

Figure 4.10: Normal Distributions of Relative Errors of or UML Approximations

Chapter 5

Normalization

Recalling our second main goal is to test the ability our UML metrics to predict faulty code. In order to do so, code-based prediction models are built, and tested using as their inputs, on the one hand, UML measures, and on the other hand, their respective code measures from the JAIST projects. However, this task can be troublesome, not only because our UML approximations are not close enough to their respective code metrics (according to the results obtained in the previous Chapter), but also because data distributions of the CK metrics varies from project to project, or within the same project from package to package, as it is observed from a further data analysis presented in a later section.

In this Chapter, we first present our results and observations from further data analysis on different datasets of code and UML CK measures using boxplots. Then, we study the application of two data normalization methods, which has a two-fold purpose to decrease the difference of values (approximation error) between UML and code measures and the variability of data spread among software projects and packages.

The data used in this Chapter are from the same four different JAIST projects used in the previous Chapter, and from the code of an open source project, which is used in a later step to build our code-based prediction models.

5.1 Comparing CK Data Distributions using Boxplots

In order to analyze and compare the CBO, RFC and WMC data of our projects, we use simple boxplots. A boxplot provides a visual impression of several important aspects of the empirical distribution of a batch data. It is especially useful for comparing several batches of data. From a boxplot we can have at glance important characteristics of our data such as: location, spread, skewness, tail length and outlying data points [25].

To construct a single boxplot of a given dataset, the following information is required:

- The median. To compare location of batches.

- The fourth spread. To compare spread among batches. It is an indicator of how concentrated the values are. And it is calculated as follows:

$$\text{fourth-spread} = d_F = F_U - F_L \tag{5.1}$$

  Where,

  - F_U is the lower fourth of the dataset, also called first quartile, and tell us that 25% of the total of the observations are smaller to its value.
  - F_L, call the upper fourth or third quartile, corresponds to the value; so that 75% of the total of the observations are smaller.

  Therefore, the forth spread or interquartile range (IQR) represents the range containing the middle 50% of the data.

- The identification of outliers. Commonly in statistics data values that are far enough the fourths are considered as potential outliers. Specifically $F_L - 1.5d_F$ and $F_U + 1.52d_F$ are defined as the outlier cutoffs [25]. In this study, we use this convention for the identification of outliers.

  The presence of significant outliers produces biased regression estimates and reduces the predictive power of regression models [18]. From there lies in the importance of identification and proper treatment of outliers.

As an example, we show in Figure 5.1 a boxplot for the CBO code measures of the BNA project. Table 5.1 shows the values of the CBO measures for 11 classes of the BNS, as well as the information required to build the boxplot, which was built according to the following procedure:

1. A box was drawn with the ends at the lower fourth F_L and upper fourth F_U. For the BNA CBO data these values are 0 and 3 respectively, but $CBO = 3$ does not exist, therefore the limit of the box is drawn at $CBO = 4$.

2. A crossbar was drawn at the median, at 1.

3. A line was drawn from each end of the box to the outlier cutoffs. The lower outlier cutoff, $F_L - 1.5d_F$, equals -4.5, which does not exist, therefore we take the nearest value, 0. For the upper outlier cutoff, $F_U - 1.5d_F$, we have a value of 7.5, which neither exists, therefore we choose the closest value, 6.

4. Finally, outliers are represented by drawing circles individually, beyond the outlier cutoffs. For the BNA the only outlier value is 13.

Having explained the functionality of a boxplot, we can easily compare the CBO, RFC and WMC data measures of our projects and look for similarities and differences among them using boxplots.

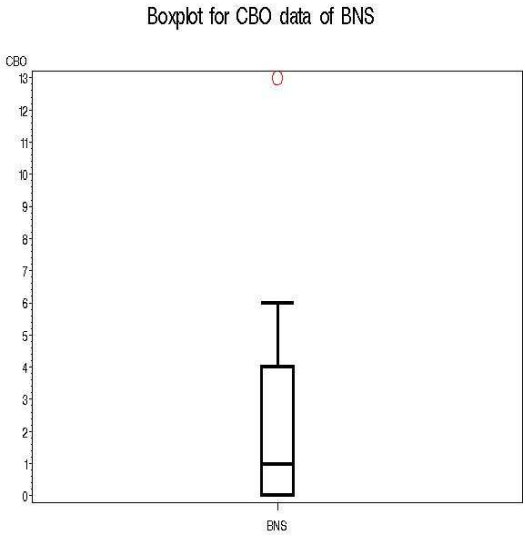

Figure 5.1: Boxplot for CBO data of a Banking System

Table 5.1: CBO Code measures of the BNS

	Class	**CBO**	**General Data**
1	BNA-06	13	
2	BNA-07	6	
3	BNA-10	4	
4	BNA-05	2	$Median = 1$,
5	BNA-09	2	$F_L = 0$, $F_U = 3 \rightarrow 4$,
6	BNA-04	1	$d_F = 3$,
7	BNA-01	0	$F_L - 1.5d_F = -4.5 \rightarrow 0$,
8	BNA-02	0	$F_U + 1.5d_F = 7.5 \rightarrow 6$
9	BNA-03	0	
10	BNA-08	0	
11	BNA-11	0	

As we mentioned earlier, apart from exploring the data from the JAIST projects, used in the previous Chapter, we have included the CBO, RFC and WMC measures collected from 13 packages of an open-source project named Mylyn from Eclipse (MYL). The MYL system was written in Java by six developers in different locations. A total of 37 classes from the JAIST projects, and 602 classes from the MYL project are used for this analysis. RFC, CBO and WMC code and UML measures were collected from these classes. Figures 5.2, 5.3,5.4 show the different boxplots for the code and UML CBO, RFC and WMC measures collected respectively. In these Figures, 'COD' stands for code measures, 'UML' for UML measures, and 'M-x', where x ranges from 1 to 13, refers to the different 13 packages of the MYL. Tables 5.2, 5.3, and 5.4 we provide the detailed information of each of the datasets represented in these figures. From these figures and tables, the following observations can be made:

- Datasets with the high number of classes tend to have more number of outliers.
- In general most of the data are positively skewed. Their right tail is longer, which means that most of the data is concentrated on the left, having relatively few high values.
- A slight tendency for data to spread as the level increases can be observed, specially for CBO and RFC data. These can be more clearly observed in Figures 5.5, 5.6, 5.7, where level (M) versus spread (dF) plots of our our CBO, RFC and WMC data, respectively, are shown.

Overall, there exists a great variability among the batches, using either code or UML. In order to promote equality of data spread and reduce such variability, data can be re-expressed by using data transformations. In the next Section, we study the application of two data transformations and determine which of these transformations is more suitable for our data.

Table 5.2: Detailed Information for CBO Boxplots

CBO Boxplots				
Batch	Total Classes	Median	d_F	Upper Outlier Cutoff
BNA-COD	11	1	3	7.5
BNA-UML	11	1	2.5	6.25
BNB-COD	10	1	4.25	10.63
BNB-UML	10	1.5	2.75	6.88
CRS-COD	16	1.5	4	11
CRS-UML	16	1	1	3.5
ECS-COD	10	3	5.5	13.7
ECS-UML	10	0.5	6.7	16.8
MYL-P01	39	3	6	15.5
MYL-P02	30	3.5	4.5	12.5
MYL-P03	56	7	10.5	30
MYL-P04	54	12	9.7	31.38
MYL-P05	23	14	19.5	57.7
MYL-P06	49	12	15	42.5
MYL-P07	24	9.5	19.5	52.5
MYL-P08	10	13	10	37.2
MYL-P09	199	10	13	37.5
MYL-P10	21	7	18	48
MYL-P11*	49	5	16	41
MYL-P12	17	5	7	21.25
MYL-P13	31	29.5	29.5	80.2

(*) Dataset from which the best CBO prediction models were derived

Table 5.3: Detailed Information for RFC Boxplots

RFC Boxplots				
Batch	Total Classes	Median	d_F	Upper Outlier Cutoff
BNA-COD	11	24	42.5	122
BNA-UML	11	4	3.5	11.25
BNB-COD	10	12.5	9.75	32.38
BNB-UML	10	4.5	3.5	11.75
CRS-COD	16	9	3.7	14.6
CRS-UML	16	4	2.5	8.7
ECS-COD	10	29	10.7	52.1
ECS-UML	10	4.5	9.7	26.3
MYL-P01	39	22	24	72.5
MYL-P02	30	35	43.5	121.7
MYL-P03	56	21	27	79.2
MYL-P04	54	25	21.5	68
MYL-P05	23	21	31	92.5
MYL-P06	49	23	30	84
MYL-P07	24	22	43.7	122
MYL-P08	10	37	28.2	88
MYL-P09	199	19	28.2	82
MYL-P10	21	10	32	85
MYL-P11	49	18	33	88
MYL-P12*	17	13	13	42
MYL-P13	31	35	65	182

(*) Dataset from which the best RFC prediction models were derived

Table 5.4: Detailed Information for WMC Boxplots

WMC Boxplots				
Batch	Total Classes	Median	d_F	Upper Outlier Cutoff
BNA-COD	11	7	7	22.5
BNA-UML	11	2	2.5	7.7
BNB-COD	10	4	1.75	7.38
BNB-UML	10	2.5	3.25	9.38
CRS-COD	16	4	3	11.5
CRS-UML	16	3	2.2	7.4
ECS-COD	10	6	2.2	10.8
ECS-UML	10	4	3	9.5
MYL-P01	39	6	10.5	29
MYL-P02*	30	17.5	18	54
MYL-P03	56	6	6.7	21
MYL-P04	54	7	8.5	24
MYL-P05	23	7	5.5	18
MYL-P06	49	5	7	20.5
MYL-P07	24	6.5	8.2	26
MYL-P08	10	6.5	6.5	21.5
MYL-P09	199	5	6	18
MYL-P10	21	3	7	19.5
MYL-P11	49	6	8	24
MYL-P12	17	6	6	17
MYL-P13	31	11	12	36

(*) Dataset from which the best WMC prediction models were derived

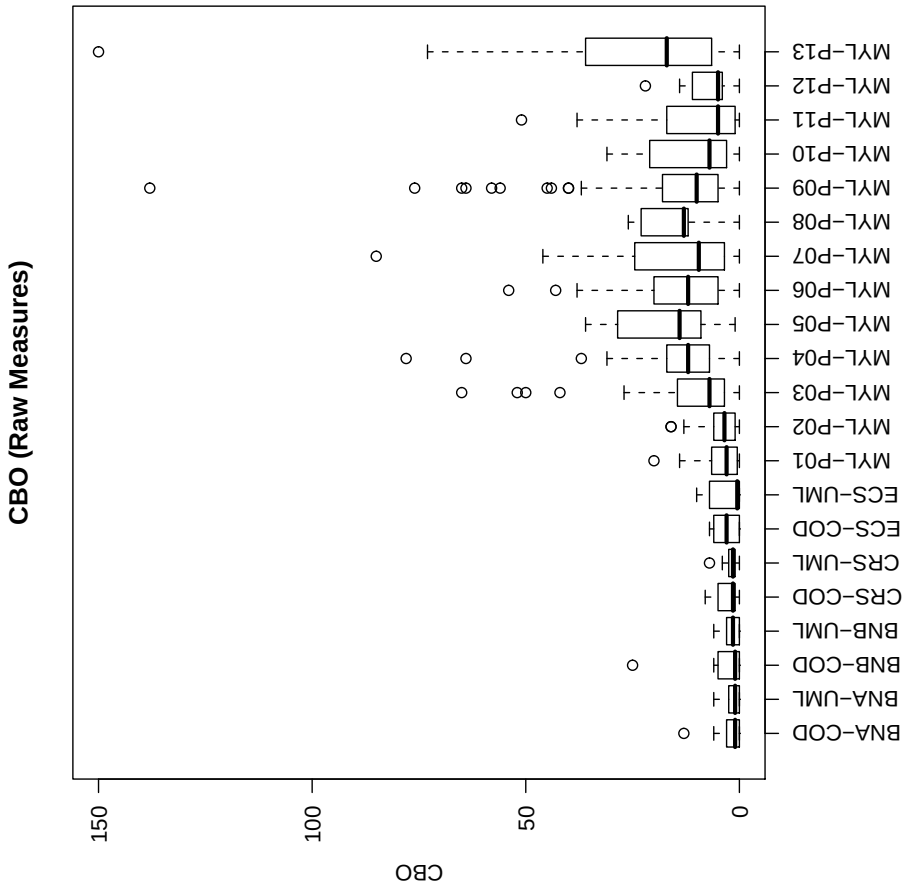

Figure 5.2: Boxplot for CBO data

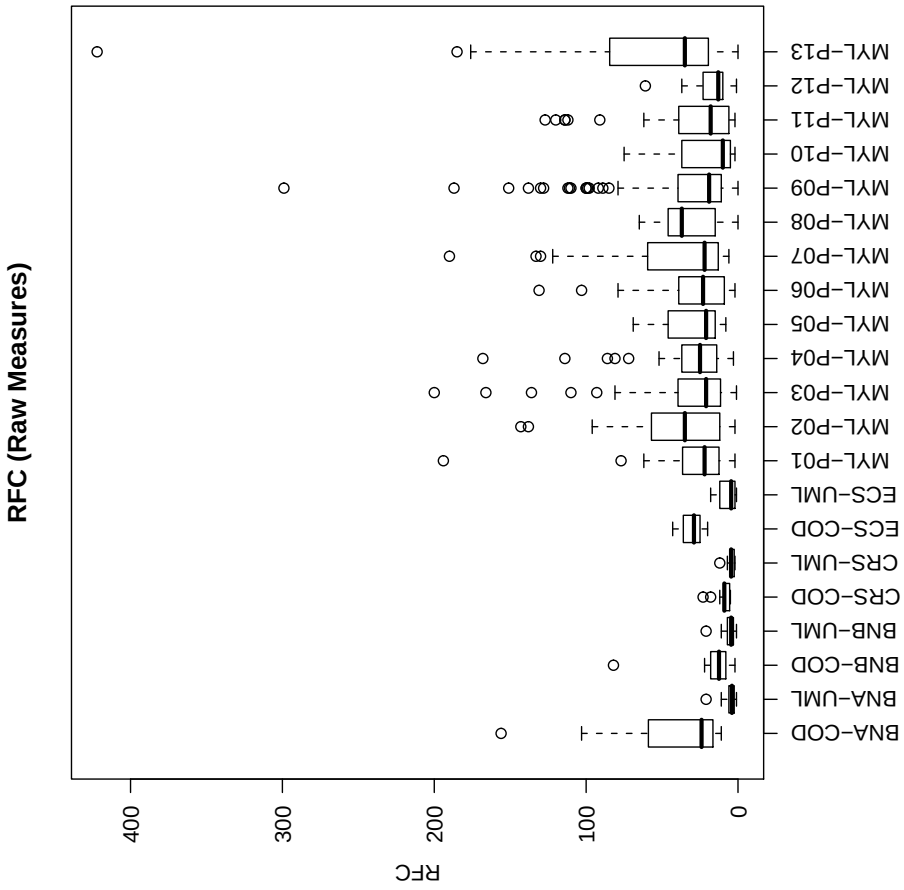

Figure 5.3: Boxplot for RFC data

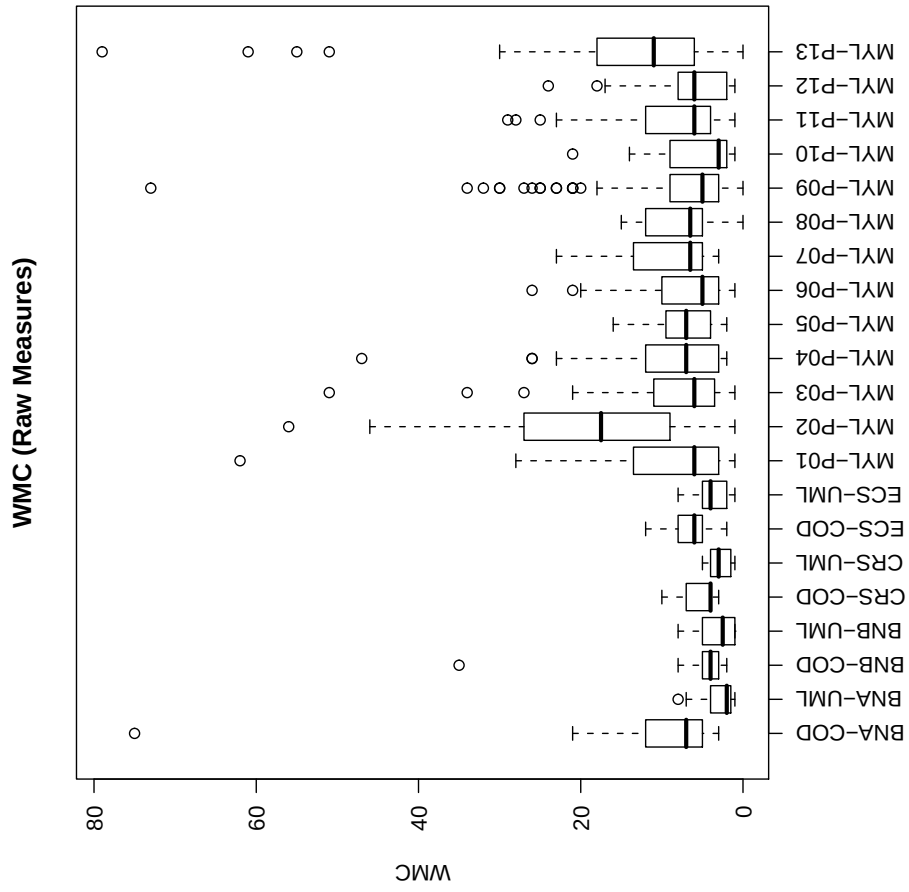

Figure 5.4: Boxplot for WMC data

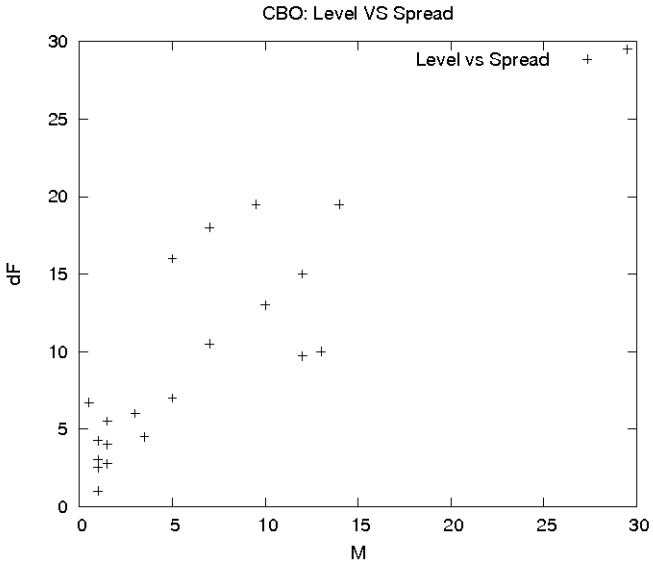

Figure 5.5: CBO Level vs Spread

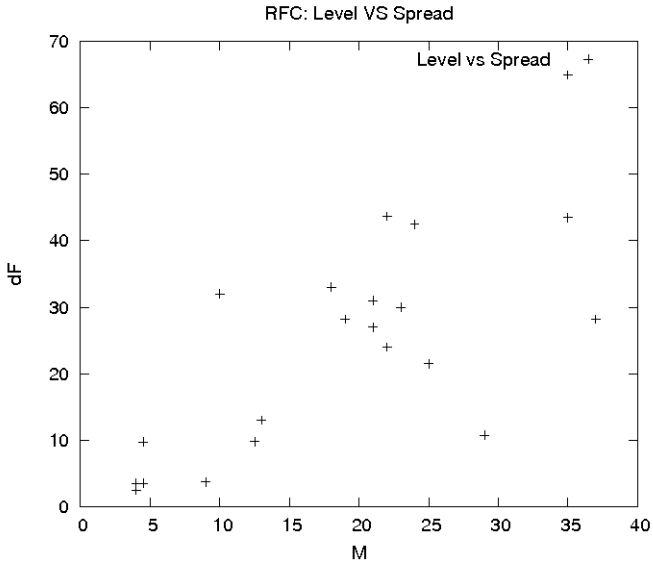

Figure 5.6: RFC Level vs Spread

5.2 Data Transformation and Normalization

In this Section we study the application of two data data transformations to normalize our data, these are *logarithmic* data transformations and *linear scaling to unit variance*.

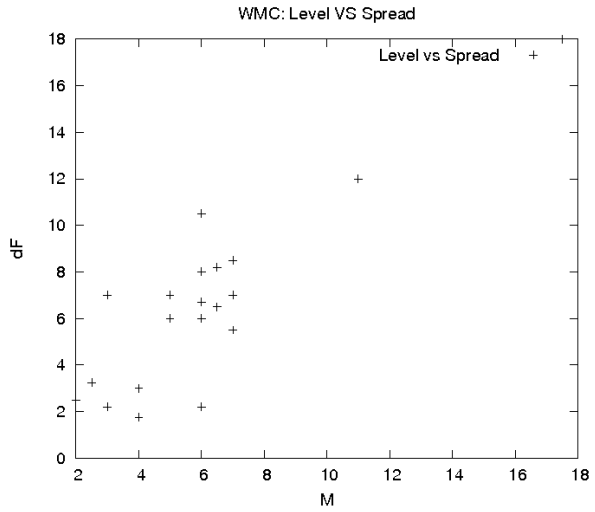

Figure 5.7: WMC Level vs Spread

Data transformations in general enable us to re-express data in a new scale that offers several improvements over the original scale of data collection [25]. The reasons we have to apply the different two chosen data transformations are the following:

- To promote stable spread across several datasets. It is said that when there exists a strong relationship between spread and level (as it was observed in our previous data explorations), power data transformations can be applied to reduce or eliminate such dependency.

  A particular case of this type of transformations is logarithmic transformations. Transformed data will be better suited for comparison, improving normality and symmetry of the distributions, and decreasing number of outliers, which are found to be detrimental for statistical techniques such as logistic regression.

- To acquire a common scale among batches. Having a common scale among our datasets would ease data comparison and let us acquire better fault predictions when using the same prediction model across packages and projects, using either UML or code metrics.

  A transformation that can change the scale of a dataset to a range $[0, 1]$ is the so called linear scaling to unit variance, which is a linear transformation. Linear transformations are often seen as more "transparent" than logarithmic transformations because their interpretation is easier. They only change origin and scale of the original data, conserving the shape of the data distribution (different to logarithmic transformations).

Since by transforming data, we would achieve equality of data spread and/or a common scale among dataset, using either logarithmic transformations or linear scaling to unit variance, we expect that the errors of our UML approximations decrease, further evaluation is performed in a later Section.

5.2.1 Logarithm Data Transformations

Power transformations are defined as the transformation that replaces x by x^p. If $p = 0$ then $log(x)$ is used instead. Power transformations can help us to promote stable spread among our different datasets, so that data can be better suited both for visual exploration and for common analytic techniques of batch comparison.

The procedure for diagnosing the power p that can stabilize spread is explained in the following lines. Based on our previous observations in Section 5.1, we can say that the fourth-spread f_d of our datasets is proportional to a power p of the median M, then we have that:

$$\begin{aligned} d_F &= cM^p; \\ log(d_F) &= log(c) + blog(M); \\ log(d_F) &= k + blog(M); \end{aligned} \tag{5.2}$$

Where, d_F is the fourth-spread, M is the median and $k = log(c)$. Thus, the logarithms of the fourth-spreads and the logarithms of the medians are linearly related. If we use our CBO data and plot the logarithms of the fourth-spreads versus the logarithms of the medians of the different batches, we have the resulting plot shown on Figure 5.8, which shows a tendency for $log(d_F)$ to increase as $log(M)$ increases, the relationship appears to be somehow linear, of the kind of Equation 5.2.

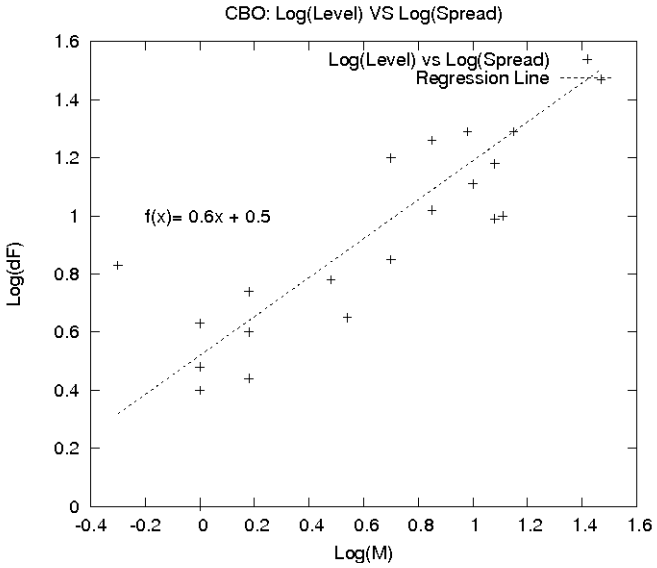

Figure 5.8: Spread-versus-level Plot for CBO data

Using linear regression, for our CBO spread-versus-level plot, we have that the best linear function that models the different data points is described by Equation 5.3:

$$f(x) = 0.6x + 0.5 \tag{5.3}$$

Finally, knowing the value of the slope of the spread-versus-level plot, b, we can easily approximate p using Equation 5.4:

$$p = 1 - b \tag{5.4}$$

Therefore, for our CBO data the more suitable value of p would be 0.4. Applying the same methodology for our RFC and WMC data, we obtained p values almost equal to zero, -0.09 and 0.05 for RFC and WMC respectively. The spread-versus-level plots for our RFC and WMC data are shown in Figures 5.9 and 5.10 respectively[1].

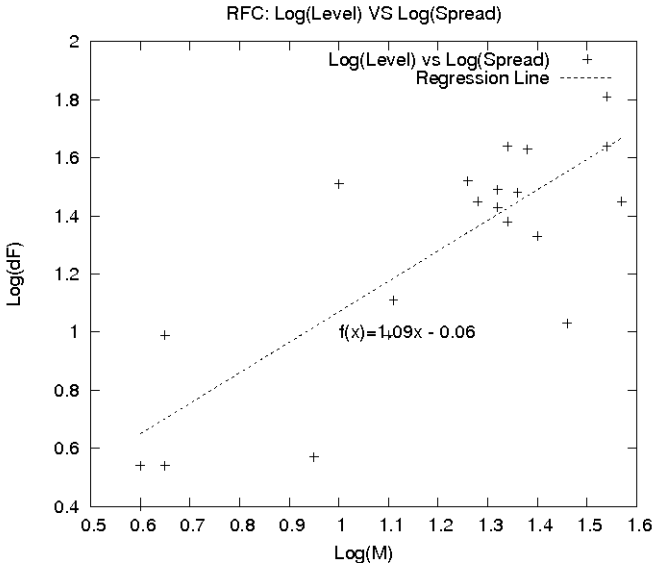

Figure 5.9: Spread-versus-level Plot for RFC data

At this time we chose to work with $p = 0$ to transformed all of our data. When using power transformations with $p = 0$, we might face zero and infinitive values. For example if we have a class with $CBO = 0$ then if we use the power transformation $p = 0$ then we will have $CBO' = log(CBO) = log(0)$, which is an undefined value. A simple and common solution is transforming as follows: $CBO' = log(CBO + 1)$ [39].

In general, CBO, RFC and WMC measures were subjected to the transformation of equation 5.5.

$$x' = log(x + 1) \tag{5.5}$$

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

After applying this transformation to our code and UML data of our JAIST projects, we compare again our UML to their corresponding code measures using simple line graphs, as we did in our first evaluation. Figures 5.11, 5.12, 5.13 show our RFC, CBO and WMC measures respectively; the line with squares corresponds to the UML measures, and the

[1]These results present a slight change to those ones published in [11] due to the inclusion of BNB data

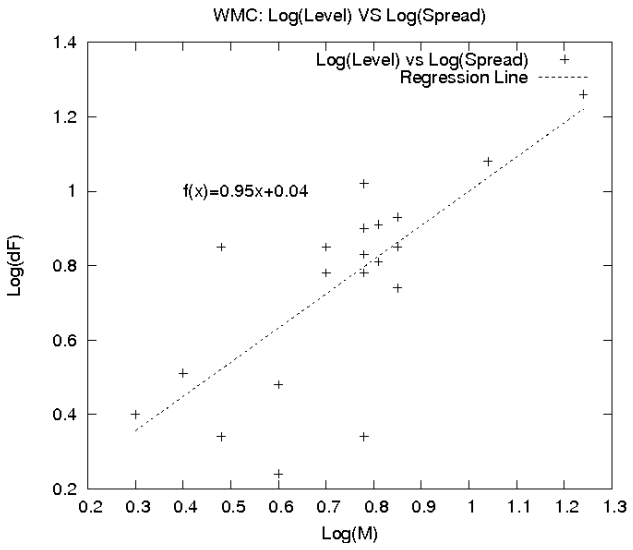

Figure 5.10: Spread-versus-level Plot for WMC data

line with circles corresponds to the code measures. Although it cannot be observed clearly, the gap between UML and code measures has decreased, in comparison with the one obtained in our first evaluation (in Chapter 4). The average relative errors of our approximations decreased to: 0.55, 0.35 and 0.38 for RFC, CBO and WMC respectively (considering average values of our 4 JAIST projects). For comparison with our previous results, please refer to Table 5.5, where the subscript *LOG* refers to our results obtained using log data transformations.

Figures 5.14, 5.15 and 5.16 show the new boxplots for the RFC, CBO and WMC measure of our different projects and packages under study. As expected, the application of logarithmic transformations resulted in fewer outliers, symmetry improvement and data spread somehow more stable. Moreover, since logarithmic transformations dampens exponential growth, it reduced our original scale across datasets, ranging from 0 to 2.5.

5.2.2 Linear Scaling to Unit Variance

Linear Scaling refers to transform a dataset to a new scale using a transformation of the form of Equation 5.6:

$$x' = a + bx \tag{5.6}$$

, where:
- x is the original or raw measure,
- x' is the transformed measure,
- b is the multiplicative component of the linear transformation, sometimes called the slope, and
- a is the additive component, sometimes refers as the intercept.

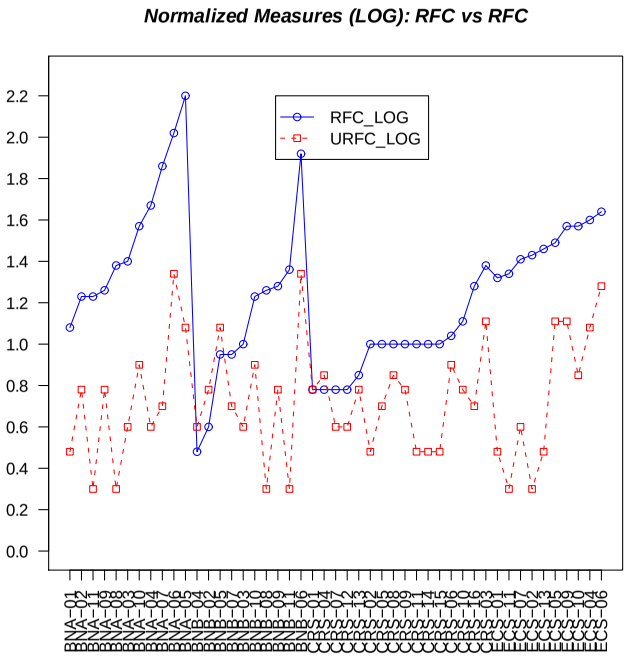

Figure 5.11: RFC measures using Logarithm Transformation

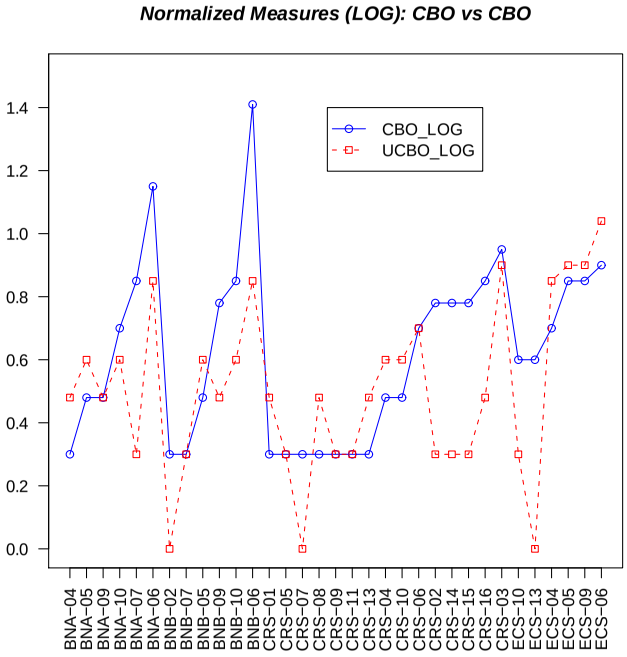

Figure 5.12: CBO measures using Logarithm Transformation

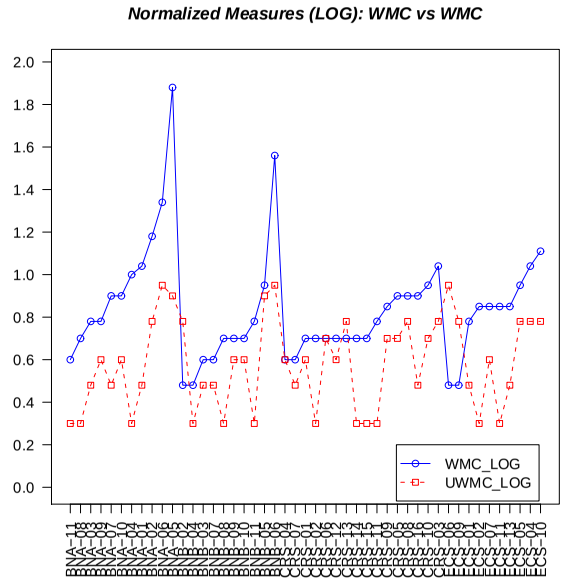

Figure 5.13: WMC measures using Logarithm Transformation

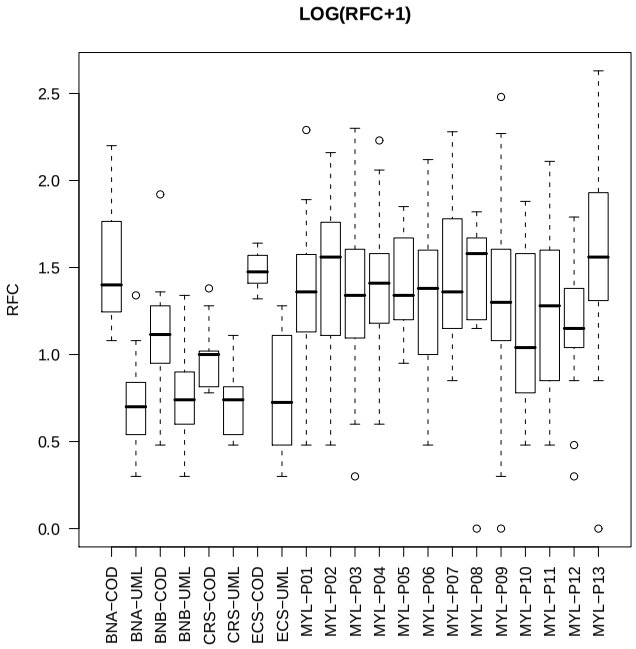

Figure 5.14: RFC Boxplots using Logarithmic Transformation

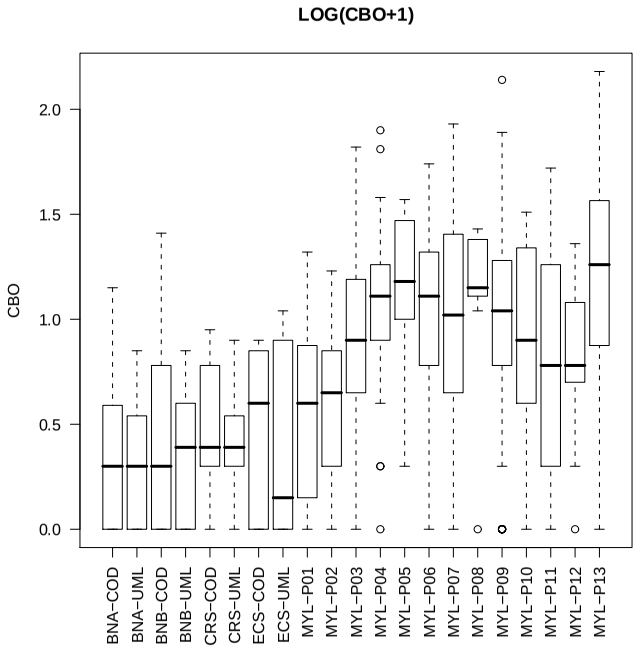

Figure 5.15: CBO Boxplots using Logarithmic Transformation

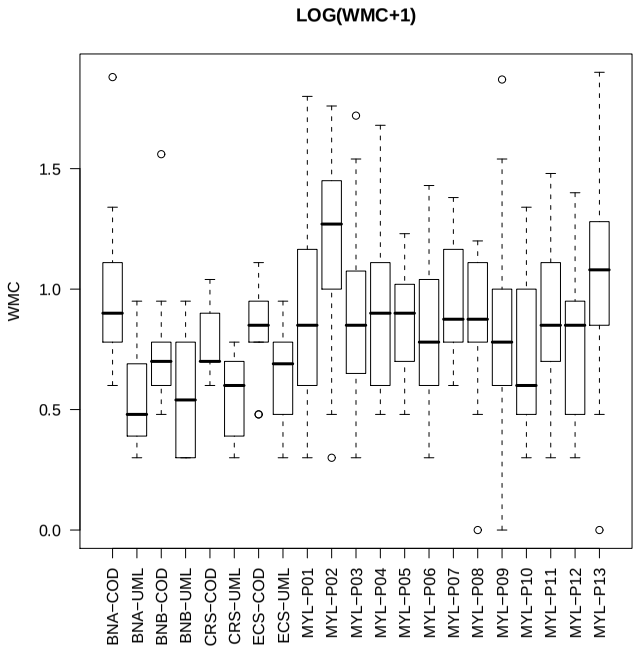

Figure 5.16: WMC Boxplots using Logarithmic Transformation

When using linear scaling transformations the distributions of the transformed and original data are similar. For example if the distribution was positively skewed before the transformation, the distribution of the transformed data will remain the same. The median $\bar{x}$ in the original scale, changes according to Equation 5.7 and the standard deviation s would be affected only by the multiplicative component, according to Equation 5.8.

$$\bar{x'} = a + b\bar{x} \tag{5.7}$$

$$s'_x = bs_x \tag{5.8}$$

Linear scaling to unit variance, as defined in [2], is a transformation based on the widely used standard score or z-score transformation. The z-score transformation is a linear transformation with a transformed mean equals 0 and standard deviation equals 1. The z-score transformation defined by Equation 5.9 can be easily derived, if we substitute $\bar{x'} = 0$ and $s'_x = 1$ in Equations 5.7 and 5.8 respectively, so that the values of a and b can be found and then used in Equation 5.6.

$$x' = \frac{x - \bar{x}}{s_x} \tag{5.9}$$

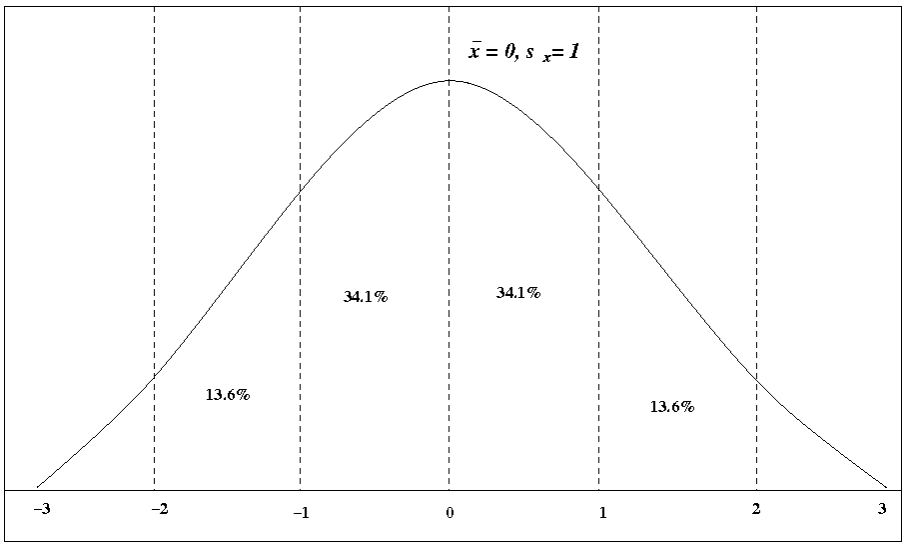

Figure 5.17: Normal distribution with media zero and standard deviation 1

Linear scaling to unit variance makes an additional shift and rescaling so that data lies into the range [0,1]. Assuming that the distribution of x is normal, the distribution of x' in Equation 5.9 would take the form of the curve shown in Figure 5.17. To guarantee that 99% of the data under study lies in the range [0,1] an additional shift and rescaling are made, taking the form of Equation 5.10. Values outside this range can be truncated either to 0 or 1.

$$x' = \frac{\frac{x-\bar{x}}{3s} + 1}{2} \tag{5.10}$$

Where:

- x' is the new normalized value,
- x is the raw value of the metric,
- $\bar{x}$ is the mean of the distribution of x,
- s the standard deviation of the distribution of x.

We applied linear scaling to unit variance to our raw data, and compared again our UML metrics to their respective code metrics. Figures 5.18, 5.19 and 5.20 show the UML RFC, CBO and WMC measures respectively; the line with squares corresponds to the UML measures, and the line with circles corresponds to the code measures. We observe that in comparison with our previous results (applying logarithmic transformations), the gap between UML and code measures was *successfully* decreased for all of the three projects, reducing the relative errors of our approximations by 50% and standard deviations also decreased. The average relative errors for RFC, CBO and WMC are 0.19, 0.17 and 0.29 respectively (considering average values of the four different JAIST projects). For comparison with our previous results, please refer to Table 5.5. The subscript LS denotes the results obtained using Linear Scaling.

The data distribution of the new normalized datasets can be observed in the boxplots in Figures 5.21, 5.22 and 5.23, as it is observed and was pointed out earlier, linear scaling to unit variance does not change the original distribution of the data.

In conclusion by transforming our UML and code measures to a common scale $[1, 0]$ using linear scaling to unit variance, we greatly improved the accuracy and precision of our UML approximations.

At this stage, we still do not know which is the effect of data normalization when predicting faulty classes across packages and projects, but we expect better prediction results using normalization rather than simply raw measures, this is a topic of the following Chapter.

Table 5.5: Average Relative Errors Information

Statistic	$URFC$	$URFC_{LOG}$	$URFC_{LS}$
Average	0.7	0.55	0.19
SD	0.17	0.17	0.14
Statistic	$UCBO$	$UCBO_{LOG}$	$UCBO_{LS}$
Average	0.53	0.35	0.17
SD	0.36	0.32	0.11
Statistic	$UWMC$	$UWMC_{LOG}$	$UWMC_{LS}$
Average	0.65	0.38	0.29
SD	0.4	0.2	0.28

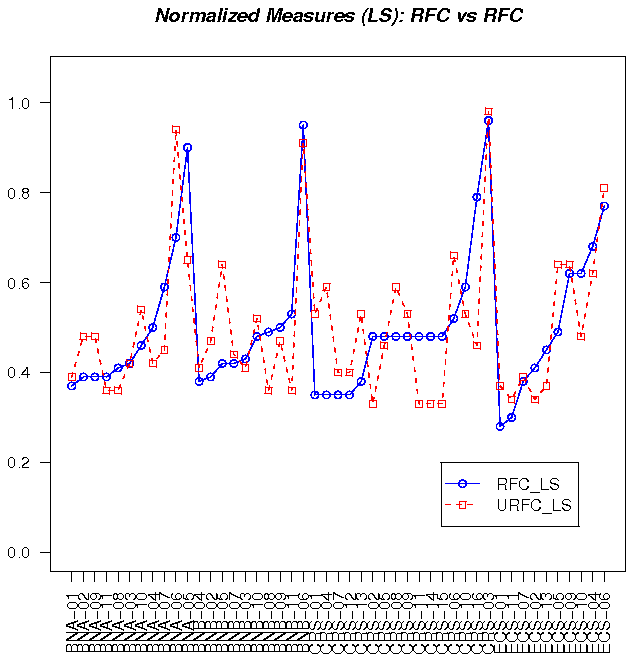

Figure 5.18: RFC measures using Linear scaling to unit variance

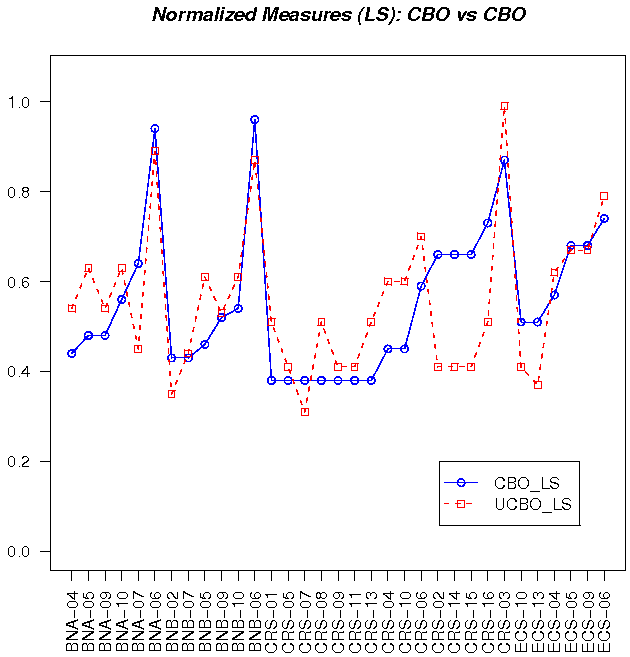

Figure 5.19: CBO measures using Linear scaling to unit variance

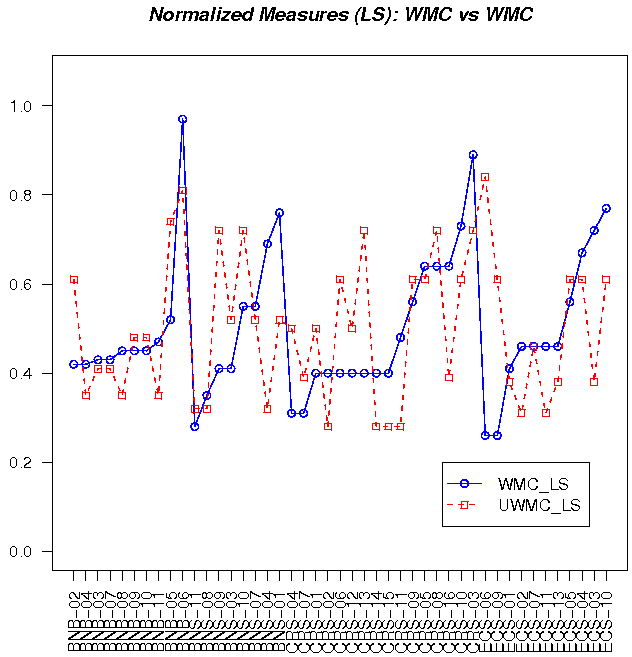

Figure 5.20: WMC measures using Linear scaling to unit variance

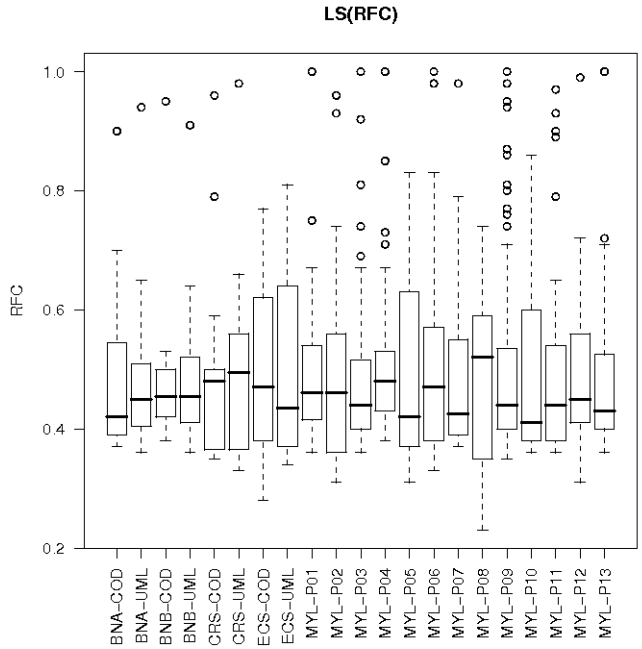

Figure 5.21: RFC Boxplots using Linear Scaling to Unit Variance

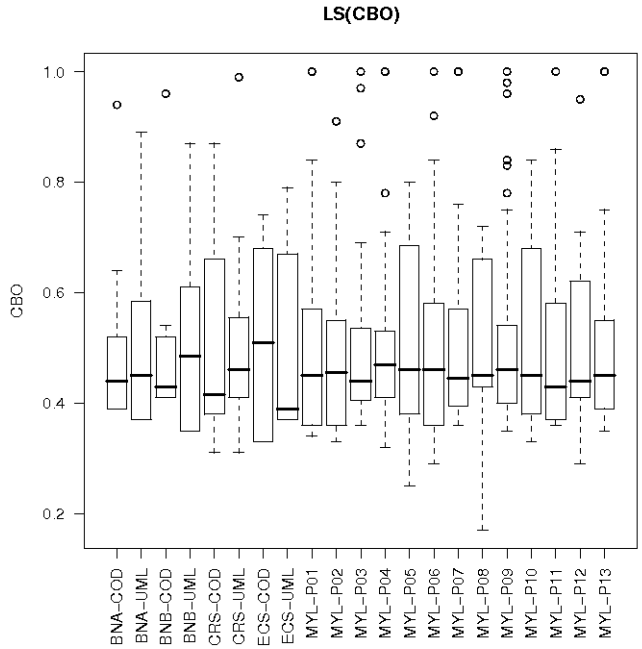

Figure 5.22: CBO Boxplots using Linear Scaling to Unit Variance

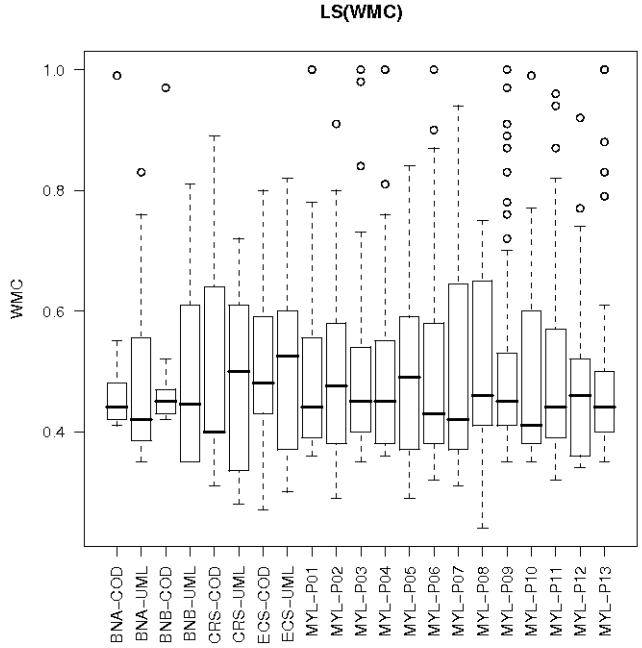

Figure 5.23: WMC Boxplots using Linear Scaling to Unit Variance

Chapter 6

Fault Prediction

This Chapter has as a main purpose to evaluate our UML CBO, RFC and WMC metrics as predictors of faulty code. The methodology used for this purpose is explained as follows, please see Figure 6.1. We first built three code-based prediction models using univariate logistic regression and linear scaling to unit variance to normalize data. For the construction of these models, the code of the MYL system, introduced in Chapter 5, was used (Figure 6.1-A). Then, the CK UML measures collected from our JAIST projects, as described in Chapter 4, were used as inputs of our prediction models (Figure 6.1-B). After that, the code CK measures from three of the JAIST projects (ECS, CRS and BNS[1]) were also used as inputs of the same prediction models (Figure 6.1-C). Finally, the prediction results from our two previous steps were compared to evaluate the prediction ability of our UML CK metrics (Figure 6.1-D).

A different approach could have been undertaken by building UML-based prediction models, without the need of using code-based prediction models; however, the latter approach was adopted because our initial intentions were to include in a sole prediction model more than one UML metric as predictors of faulty code (multivariate analysis). For such a purpose the number of data observations required is, as a rule of thumb, 10 at least per independent variable. This is because for every independent variable regressor coefficients need to be estimated to build the multivariate prediction model, and unless we provide enough data observations, such parameters would be poorly estimated. Therefore, building a multivariate prediction model, using our JAIST projects, from which UML metrics are available, would not be proper since the number of classes per project is not larger than 16.

Moreover, this approach also allowed us to perform a better analysis of the application of the same prediction model across different projects and packages (JAIST and MYL projects) using raw data and normalized data.

[1]In this Chapter, the code implementation used of BNS was that one referred as BNA in previous Chapters

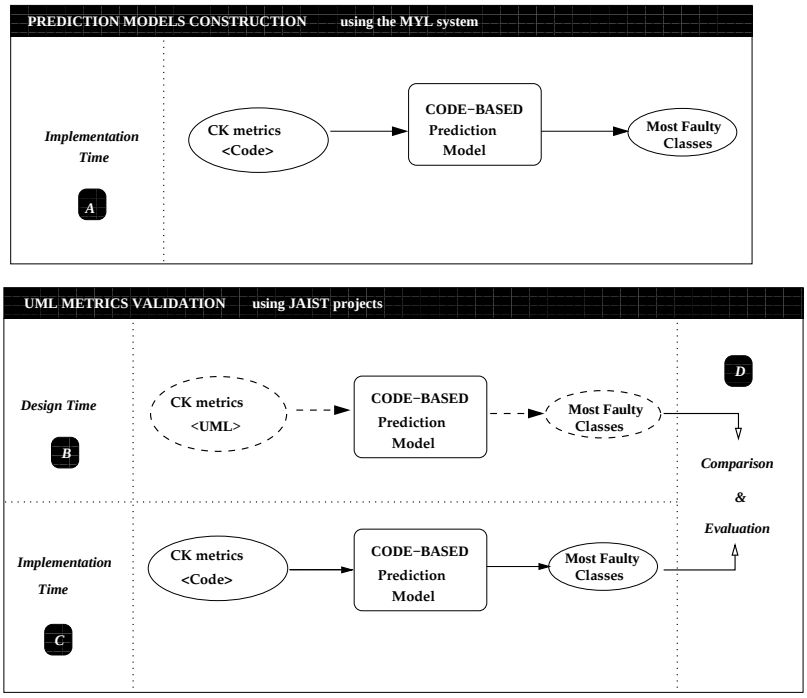

Figure 6.1: Methodology to Evaluate our UML metrics as Predictors of Faulty Code

6.1 Construction of Univariate Logistic Models

To construct our prediction models, we chose to use logistic regression due to the following reasons:

1. Logistic regression does not strictly require normally distributed variables as other statistical techniques such as linear regression and discriminant analysis.

2. 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. Although, ANN are non linear and do not make any assumption about the population distributions of the variables used, they are often referred as black boxes because their derived parameters and coefficients cannot be easily analyzed or interpret.

3. To construct prediction models, do not require large amount of data in comparison with the amount of data ANNs usually require.

We constructed univariate logistic models of the kind of Equation 6.1 to predict Most Faulty (MF) classes. In this equation x is the independent variable and $P(MF)$ is the probability of having a Most Faulty class. A is the intercept value and B is the regression coefficient of x. Figure 6.2 shows the resulting graph of this model, the horizontal axis represents the independent variable x and the vertical axis $P(MF)$.

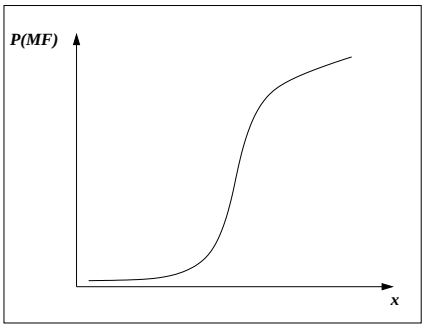

Figure 6.2: Univariate Logistic Model

$$P(MF) = \frac{1}{1 + e^{-(A+Bx)}} \tag{6.1}$$

6.1.1 Independent Variables

For the construction of the models, we used 13 packages of the first-released version of the Mylyn software (MYL), introduced in a previous Chapter. From every package of the MYL, we obtained three univariate logistic (UL) models using CK-RFC, CK-CBO and CK-WMC as their independent variables.

6.1.2 Dependent Variable

The Mylyn web site provides well organized records of the number of faults (or bugs) found in every released version of the system. Such records are registered in a bugzilla database[2]. This information and the information obtained from parsed CVS repository logs per class were used to assess the number of faults per class for the first release version of the MYL. Since logistic regression requires a dichotomous variable as dependent variable, we chose to classify our classes as Most Faulty (MF) and Least Faulty (LF). For the construction of the UL models, 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.

The distribution of the number of faults per class of the different JAIST projects and MYL packages used to construct and test our models (Section 6.2.4) are represented by the boxplots shown in Figures 6.3 and 6.4 for the CBO models, Figures 6.5 and 6.6 for the RFC models, and Figures 6.7 and 6.8 for the WMC models. Most of the fault data is positively skewed, having most of the classes relatively low number of faults.

More information about the distribution of the number of faults per class of every dataset can be found in Tables 6.1, 6.2 and 6.3, where $Q1$ refers to the value of the first quartile, $Q2$ to the second quartile, $Q3$ to the third quartile and $Maximum$ to the maximum number of faults found in a class of a given dataset. Notice that the distribution

[2]Bugzilla is the leading open-source/free software bug tracking system

of faults is slightly different among the data used for the construction of the different UL models, which is result of the elimination of outliers of CBO, RFC and WMC data (this procedure is further explained in the following section).

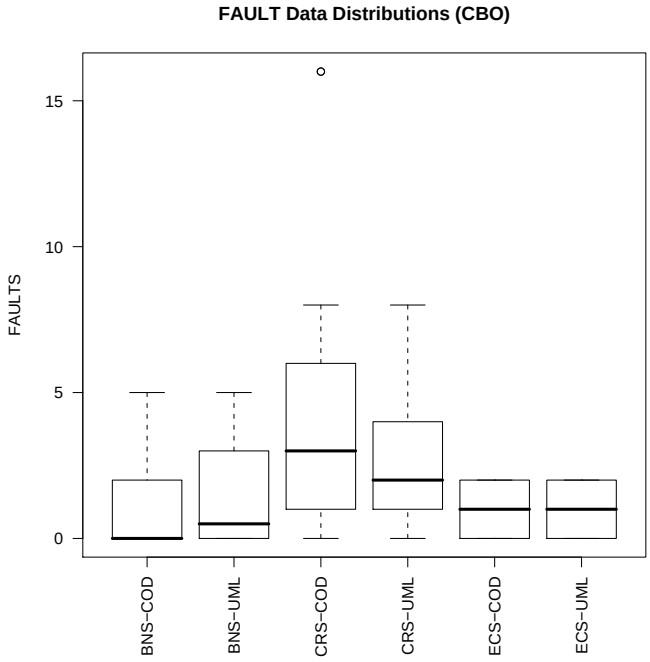

Figure 6.3: Boxplots of Fault Data of JAIST Projects (for CBO Models)

6.1.3 Procedure for Model Construction

We construct several UL models to predict each of our independent variables RFC, WMC and CBO according to the following procedure. For every package of the MYL system:

1. 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) to improve normality. Outlier data points of the CK metrics are per se focus of attention indicating high complexity and/or possible design violations. Such data points, as recommended in [14], should be subjected to further analysis.
3. We normalized the dataset using Linear Scaling to Unit Variance.
4. We built 3 UL models for predicting RFC, CBO and WMC independently.

Then, from the different 13 UL CBO, RFC and WMC models we chose the best resulting models. The best CBO model was built with the data of the Mylyn package

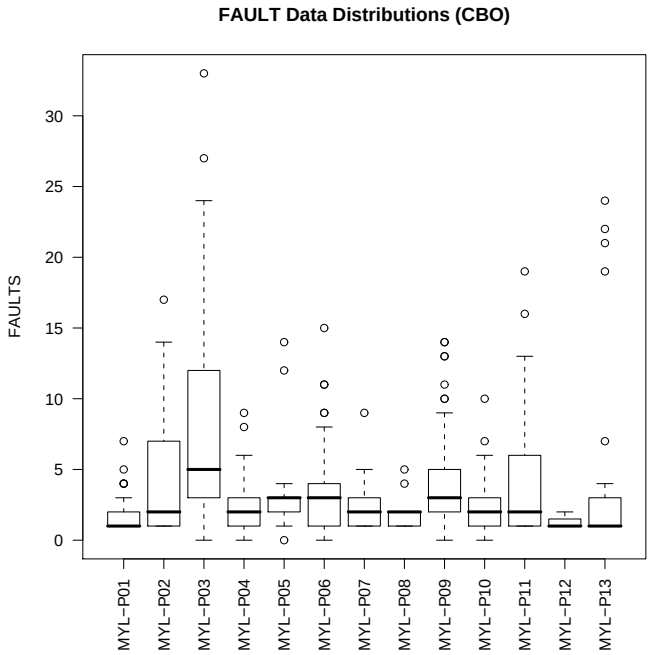

Figure 6.4: Boxplots of Fault Data of Mylyn Packages (for CBO Models)

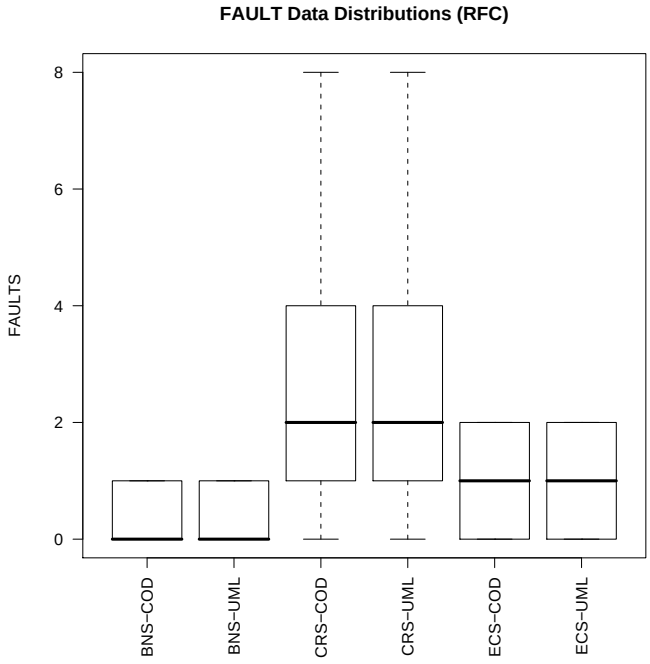

Figure 6.5: Boxplots of Fault Data of JAIST Projects (for RFC Models)

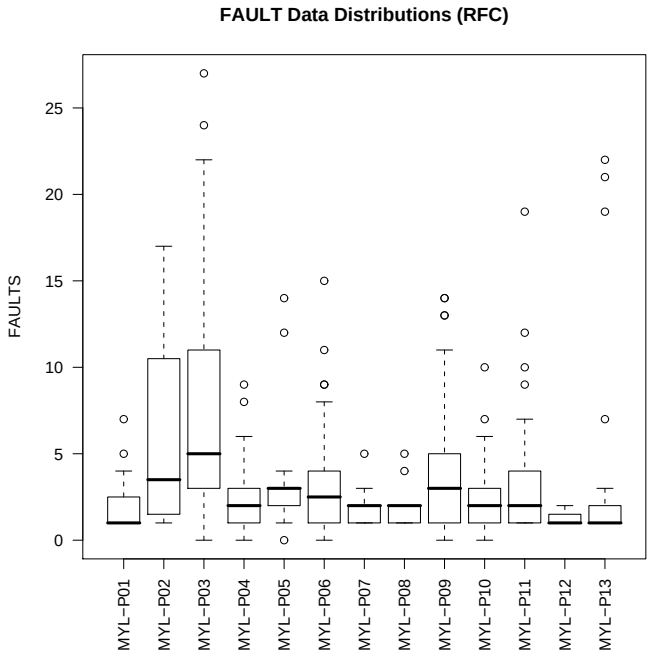

Figure 6.6: Boxplots of Fault Data of Mylyn Packages (for RFC Models)

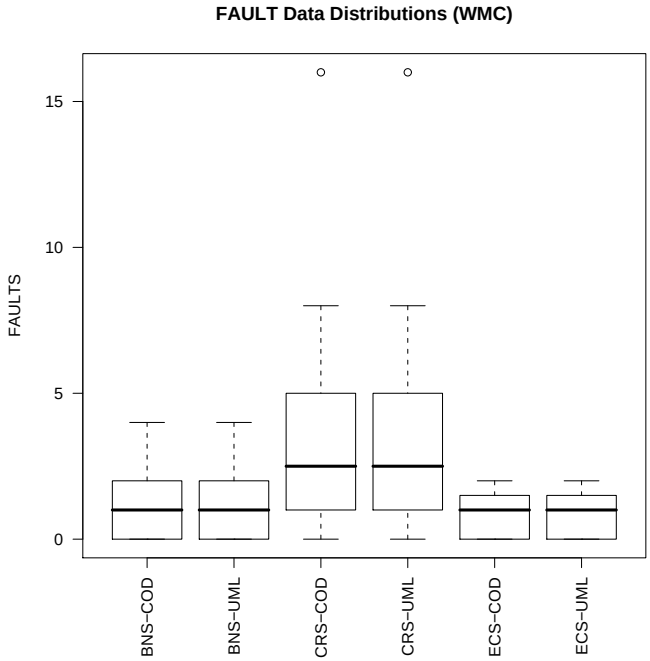

Figure 6.7: Boxplots of Fault Data of JAIST Projects (for WMC Models)

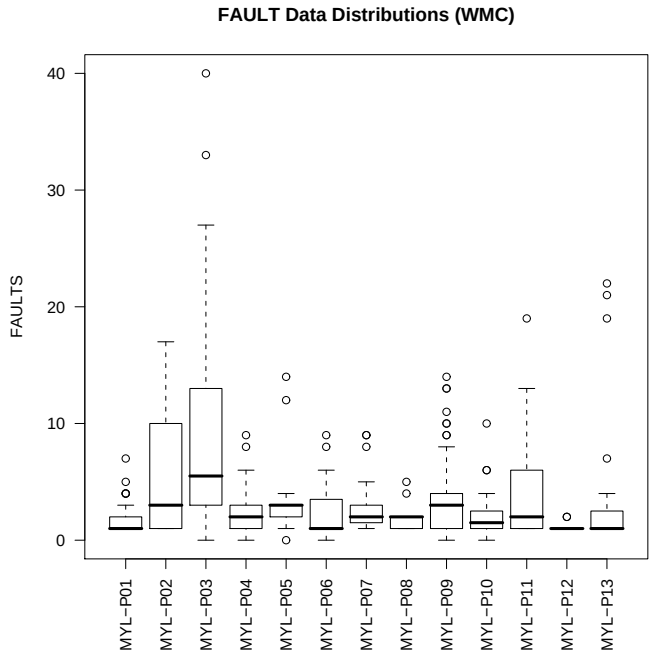

Figure 6.8: Boxplots of Fault Data of Mylyn Packages (for WMC Models)

labeled as MYL-P11 in Table 5.2, as for the best RFC and WMC models, packages MYL-P12 and MYL-P02 in Tables 5.3 and 5.4, respectively, were used.

Table 6.4 shows the coefficients of the three best UL models found, of the kind of Equation 6.1. 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 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.

6.2 Validation of UL models

In the next sections we give the details of the construction of every univariate logistic model, using SAS[3]. After estimating the regressor coefficients of our logistic models of the kind of Equation 6.1, there are several steps necessary to assess the appropriateness, adequacy and usefulness of our models [6]. First, the importance of each of the variables is assessed by carrying out statistical tests of significance. Second, the overall goodness of fit of the model is tested. Third, the ability of every model to discriminate between Most Faulty and Least Faulty classes is tested. And finally, the model using different data is validated.

[3]Statistical Analysis Software

Table 6.1: Detailed Information of Fault Data Distribution for CBO Models

Dataset	Total Classes	Total Faults	$Q1$	$Q2$	$Q3$	$Maximum$
BNS-COD	9	11	0	0	2	5
BNS-UML	10	15	0	0.5	2.75	5
CRS-COD	16	57	1	2.5	4.5	16
CRS-UML	14	38	1	2	4	8
ECS-COD	10	9	0	1	1.75	2
ECS-UML	10	9	0	1	1.75	2
MYL-P01	38	69	1	1	2	7
MYL-P02	25	118	1	2	7	17
MYL-P03	51	435	3	5	12	33
MYL-P04	50	119	1	2	3	9
MYL-P05	23	76	2	3	3	14
MYL-P06	47	153	1	3	4	15
MYL-P07	21	52	1	2	3	9
MYL-P08	10	22	1.25	2	2	5
MYL-P09	183	657	2	3	5	14
MYL-P10	21	52	1	2	3	10
MYL-P11*	48	199	1	2	6	19
MYL-P12	16	20	1	1	1.25	2
MYL-P13	27	125	1	1	3	24

(*) Dataset used for the construction of the best CBO prediction models

6.2.1 Importance of each independent variable

To determine the importance of every independent variable of our models, we use the *-2LogL* statistic, which has a χ^2 (chi square) distribution with 1 degree of freedom (df), which can be explained as follows.

For univariate logistic models, SAS provides us with two values of the *-2LogL* statistics, where L is the likelihood of a model, defined as the probability that the estimated hypothesized model represents the input data [49]. One of these values is for the complete model (intercept and independent variable) and the second is for a second model with only the intercept. The difference of these two values is what determines the importance of the independent variable in the model. The degrees of freedom used are also the difference between the two models, 1 for our case.

For this test a significance level of $\alpha = 0.05$ was chosen. A significance level is defined as the probability of making a decision to REJECT the null hypothesis of a test. If a test of significance gives a p-value lower than the α*-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.* Therefore, for a chosen $\alpha = 0.05$ a $-2LogL > \chi^2_{1,0.05} = 3.8$ is required.

The yielded *-2LogL* values of our models suggest that the three variables are signifi-

Table 6.2: Detailed Information of Fault Data Distribution for RFC Models

Dataset	Total Classes	Total Faults	$Q1$	$Q2$	$Q3$	$Maximum$
BNS-COD	6	2	0	0	0.75	1
BNS-UML	6	2	0	0	0.75	1
CRS-COD	14	35	1	2	3.75	8
CRS-UML	14	35	1	2	3.75	8
ECS-COD	10	9	0	1	1.75	2
ECS-UML	10	9	0	1	1.75	2
MYL-P01	32	61	1	1	2.25	7
MYL-P02	24	138	1.75	3.5	10.25	17
MYL-P03	43	334	3	5	11	27
MYL-P04	49	117	1	2	3	9
MYL-P05	23	76	2	3	3	14
MYL-P06	46	142	1	2.5	4	15
MYL-P07	15	30	1	2	2	5
MYL-P08	10	22	1.25	2	2	5
MYL-P09	174	613	1	3	5	14
MYL-P10	21	52	1	2	3	10
MYL-P11	42	141	1	2	4	19
MYL-P12*	16	20	1	1	1.25	2
MYL-P13	25	96	1	1	2	22

(*) Dataset used for the construction of the best RFC models

cantly important for its corresponding model to fit the data, using transformed and raw values. These values are shown in Table 6.5.

6.2.2 Overall goodness of fit

To test the overall goodness of fit, we use the Hosmer-Lemeshow (HL) test. This statistic has a χ^2 distribution. Large values of HL or small values of the probability P obtained from it indicate that the fit of the overall goodness of fit of the model may not be good. Moreover, it is important to take into consideration that when less than 4 df are used, then the statistic is likely to indicate that the model under testing fits the data [1].

Therefore, considering an $\alpha = 0.05$, 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 6.5; values on the left side of the column correspond to the models using data transformations, while those in the right to the models using raw data.

Table 6.3: Detailed Information of Fault Data Distribution for WMC Models

Dataset	Total Classes	Total Faults	$Q1$	$Q2$	$Q3$	$Maximum$
BNS-COD	9	11	0	1	2	4
BNS-UML	9	11	0	1	2	4
CRS-COD	16	57	1	2.5	4.5	16
CRS-UML	16	57	1	2.5	4.5	16
ECS-COD	11	10	0	1	1.5	2
ECS-UML	11	10	0	1	1.5	2
MYL-P01	37	67	1	1	2	7
MYL-P02*	29	163	1	3	10	17
MYL-P03	52	493	3	5.5	13	40
MYL-P04	48	115	1	2	3	9
MYL-P05	23	76	2	3	3	14
MYL-P06	39	92	1	1	3.5	9
MYL-P07	24	72	1.75	2	3	9
MYL-P08	10	22	1.25	2	2	5
MYL-P09	163	550	1	3	4	14
MYL-P10	20	45	1	1.5	2.25	10
MYL-P11	41	156	1	2	6	19
MYL-P12	14	16	1	1	1	2
MYL-P13	24	97	1	1	2.25	22

(*) Dataset used for the construction of the best WMC models

6.2.3 Discrimination

In this section, we measure the ability of the models to discriminate between the two groups defined by the response. In other words how good our models classify a class as Most Faulty or Least Faulty. For this step, the Hubert's statistical procedure can be used [49].

The Hubert's statistical procedure requires the following statistics are required:

- O is the number of correct classification.
- e is the expected number of correct classifications due to chance.
- Z statistic follows a normal distribution, and if its value is significant at an $\alpha = 0.05$, suggests that the number of correct classifications is significantly greater than those obtained due to chance.

For all our models the yielded values of O are greater than those of e, which indicates *a good discrimination* between the MF and the LF classes. Also, the yielded Z values suggest that the number of correct classifications is significantly greater than that obtained

Table 6.4: Univariate Logistic Regression Models

x	**A**	**B**
RFC	-7.75 | -4.83	12.05 | 0.2
WMC	-6.11 | -2.59	13.39 | 0.17
CBO	-12.76 | -1.86	30.72 | 0.46

Table 6.5: Univariate Logistic Regression Models Validation

x	**-2Logl**	**HL**
RFC	7.06 | 7.17	$\chi^2_{6,0.12} = 10.31 \mid \chi^2_{6,0.12} = 10.16$
WMC	14.68 | 14.93	$\chi^2_{7,0.83} = 3.56 \mid \chi^2_{7,0.82} = 3.59$
CBO	33.39 | 33.91	$\chi^2_{5,0.12} = 8.581 \mid \chi^2_{5,0.13} = 8.5$

by chance. The values of these statistics can be found in Table 6.6; 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 LF classes is 0.5.

6.2.4 Validation using different data

Finally, we present the results obtained from the most important test of our empirical study, concerning fault prediction, which is the validation of our models using different data to those used for the construction of our models.

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 previous Chapter, using code and UML metrics. Such a test has two purposes: Firstly, to determine whether

Table 6.6: Discrimination Test

Group	x	e	O	Z
MF	CBO	16.33	23 | 23	2.56 | 2.56
	RFC	1	3 | 3	2.31 | 2.31
	WMC	8.83	13 | 13	2.1 | 2.1
LF	CBO	8.33	19 | 18	4.84 | 4.38
	RFC	9	11 | 12	1.33 | 2
	WMC	5.83	9 | 10	1.77 | 2.33
Both	CBO	24.67	42 | 41	24.67 | 24.67
	RFC	10	14 | 15	2.07 | 2.58
	WMC	14.66	22 | 23	2.73 | 3.1

the elimination of outliers and normalization procedure, described in 6.1, can improve the results obtained using the same prediction model with other datasets (packages and projects); secondly, to determine whether this same procedure can be used suing UML metrics instead, so that we can predict before coding.

Based on the yielded prediction results, different threshold values were chosen to classify between MF and LF classes of the datasets for testing our models:

- For testing the CBO models, in most cases the second quartile was chosen. In some other few cases, the third quartile resulted to be a better option for classification threshold (BNS-COD, BNS-UML, MYL-P01, MYL-12 and MYL-13). Please refer to Table 6.1 for detailed information.
- For testing the RFC models, classes with number of faults equal or greater than the maximum number of faults of a class within a dataset divided by two were classified as MF and LF otherwise. So that, for example, looking at Table 6.2, for testing the RFC model with the dataset ECS-COD, the threshold value for classification equals 1 ($Maximum/2 = 2/2$).
- For testing the WMC models, the same threshold value used for testing our RFC models was chosen, except for testing with the MYL-P06 dataset, for which the second quartile was used. Please refer to Table 6.3 for detailed information.

Validation

After the application of our models in the different datasets, for the sake of simplicity, we used three indicators to assess how well our models can classify or discriminate between MF and LF classes, these indicators were:

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

Tables 6.11, 6.12 and 6.13 show the percentages of Correctness, Specificity and Sensitivity obtained from the application of our models in the different datasets 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 6.7, 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.

Table 6.7: Normalized code measures vs UML measures

Model	Project	Correctness		Specificity		Sensitivity	
		Code	UML	Code	UML	Code	UML
NRFC	ECS	80%	80%	100%	100%	67%	67%
	CRS	57%	64%	80%	80%	0%	25%
	BNS	33%	67%	50%	75%	0%	50%
NCBO	ECS	80%	80%	75%	100%	83%	67%
	CRS	88%	71%	88%	33%	88%	100%
	BNS	89%	80%	83%	67%	100%	100%
NWMC	ECS	55%	45%	25%	25%	71%	57%
	CRS	56%	38%	57%	36%	50%	50%
	BNS	78%	44%	67%	33%	100%	67%

6.2.5 Observations

From the results obtained in the previous Section, the following observation can be made:

Effect of Normalization on code measures

To summarize, comparing the 16 datasets (packages 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.

6.3 Multivariate Analysis and Correlation

In order to improve the prediction ability of our models, we could use more than one metric as independent variable in a single prediction model. However, multivariate logistic regression and linear regression require uncorrelated independent variables, and some relationship between our dependent variable and the selected independent variables must exist.

Let us say we would like to build a prediction model of the kind of Equation 6.2, where $P(MF)$ is the probability of having a "Most Faulty" class, A is the intercept value, B, C and D are the regressor coefficients of RFC, CBO and WMC respectively. For such construction, we would use the data of the package of the MYL system labeled as MYL-P11 in Table 5.2, from which the best UL prediction model using CBO was derived. We chose this package because from this package were also derived one of the best 5 prediction models using RFC and WMC independently, moreover the size (49 classes) of the dataset is enough large to calculate the coefficients A, B, C and D.

$$P(MF) = \frac{1}{1 + e^{-(A+B*RFC+C*CBO+D*WMC)}} \tag{6.2}$$

The first step would be to find out if there exists a possible relationship between our our independent variables and dependent variable, number of faults per class. A scatter plot can help us to determine the existence of such a relationship, when this relationship results to be a linear one, correlation coefficients are of great utility. For our case study, the correlation coefficients found are shown in Table 6.8. These coefficients suggest the existence of a strong linear relationship between number of faults and CBO, the same for RFC and as for WMC suggest a weaker linear relationship.

The second step would be to calculate the correlation coefficients among independent variables, RFC, CBO and WMC. These are shown in Table 6.9, which is called correlation matrix. It lists the variable names down the first column and across the first row. The diagonal of a correlation matrix always consists of ones, because the correlation between a variable and itself is always perfect (1). To localize the correlation coefficient for any pair of variables, we should find the value in the table for the row and column intersection for those two variables. For instance, the correlation between variables RFC and CBO is 0.9. As we can observe the three variables are highly correlated among them, which prevents us of using these variables as predictors in the same prediction model.

Table 6.8: Correlation Between Independent and Dependent Variables

	RFC	CBO	WMC
FAULTS	0.7	0.9	0.5

Table 6.9: Correlation Among Independent Variables

	RFC	CBO	WMC
RFC	1	-	-
CBO	0.9	1	-
WMC	0.8	0.7	1

The previous results conform other studies' findings, which suggest a high correlation

not only among the CK metrics [14, 52], but also among other sets of design-complexity metrics, such as the QMOOD and the MOOD metrics [32, 43].

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 [36, 55]. These metrics have been associated with the maintenance and understandability of UML class diagrams [19, 21, 36].

A common solution for the problem of correlated independent variables 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 [28]. Unfortunately, when using these new variables, the resulting 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.

How much better a prediction model with correlated independent variables can do?

For instance, a previous research study [43], found a good multivariate logistic 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.

To enhance the understanding of why correlated independent variables cannot be used in multiple regression, a rapid analysis can be made supposing that we want to build, not a multivariate logistic regression model, but a multiple linear regression model to predict Number of Faults, NF, using the data of our previous example. For this purpose, we use shared variance coefficients and the multiple coefficient of correlation, which are explained in the the following paragraphs.

The shared variance between two variables, X and Y is the amount that the variations of the two variables tend to vary together. The percentage of shared variance or coefficient of determination is represented by the square of their correlation coefficient, r^2. Another way to visualize this is with a Venn diagram, for example in Figure 6.9, the amount of shared variance between X and Y variables is represented by their overlap of variation r^2.

The multiple coefficient of correlation, R^2, is a measure of the fit of a multiple linear regression model. It falls somewhere between zero and one. A value of one indicates that all data points fall exactly on a line in multidimensional space, and a value of zero indicates that there is not any relationship between the independent variables collectively and the dependent variable. When the independent variables are not correlated, the sum of their shared variance (r^2) with the dependent variable is equal to R^2. However, when the independent variables are correlated, this strategy overestimates the contribution of each variable because the variance that they share is counted more than once; and therefore the sum of r^2 cannot be used.

For our specific case study, the coefficients of determination between NF and every of

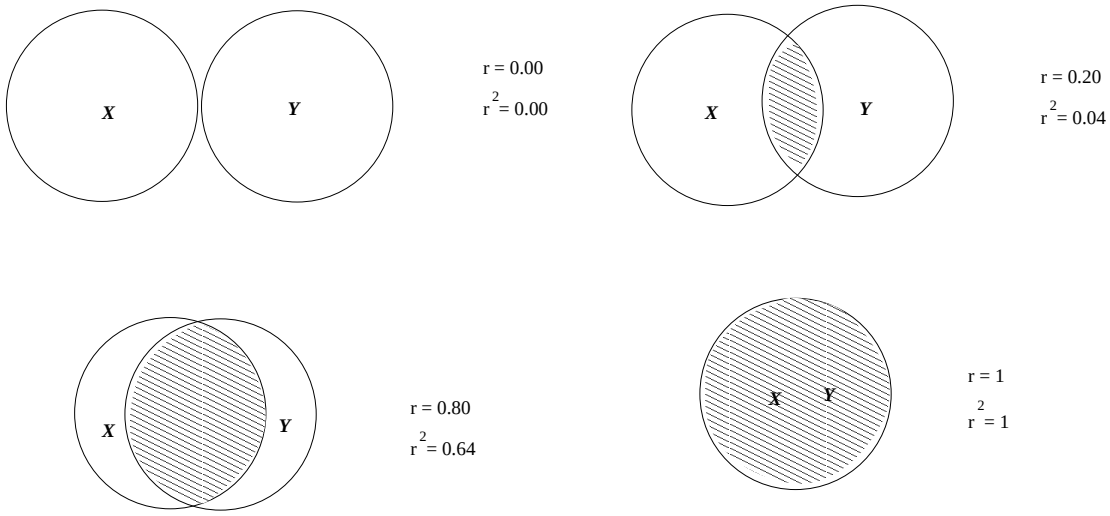

Figure 6.9: Correlation and Shared Variance

our independent variables are shown in Table 6.10. Based on these results, we can say that 49% of the variability of RFC can be related to NF of a class. Likewise, 81% and 35% of the variability of CBO and WMC, respectively, can be related to NF independently.

Now, since CBO has the highest shared variance with NF ($r^2_{NF.CBO} = 0.81$), we would chose it first as predictor to build a univariate multiple linear regression model and predict NF, this is represented in Figure 6.10-a. On the other hand, because RFC's shared variance with NF is the second highest ($r^2_{NF.RFC} = 0.49$), represented in Figure 6.10-b, we could think that the inclusion of RFC as independent variable in our CBO univariate prediction model could yield better results; however, because CBO and RFC are highly correlated to each other ($r = 0.9$, $r^2_{CBO.RFC} = 0.81$) the shared variance of RFC with NF is very likely to be included in the shared variance with CBO, Figure 6.10-c), in other words, the benefits of including RFC in our regression model would not be significant, resulting in a multiple correlation coefficient $R^2_{NF.CBO,RFC} \approx r^2_{NF.CBO} = 0.81$. From here originates the need of including uncorrelated independent variables to regression models.

Table 6.10: Coefficients of Determination Among Variables

Dependent Variable	RFC	CBO	WMC	Faults
RFC	1	-	-	-
CBO	0.81	1	-	-
WMC	0.64	0.49	1	-
Faults	0.49	0.81	0.25	1

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*

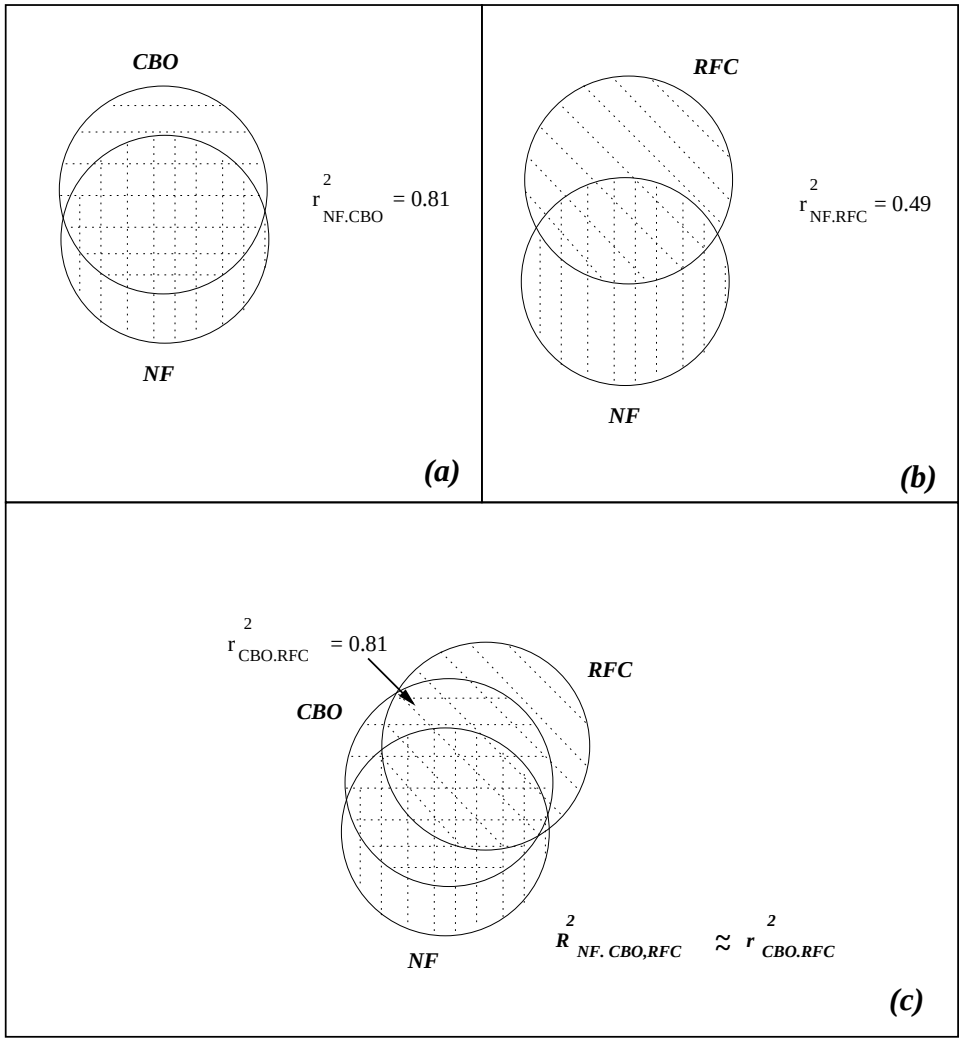

Figure 6.10: Shared Variance and Correlated Independent Variables

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 communication, etc. We do agree with Fenton's criticism [17], 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.

6.4 Guidelines for Further Use

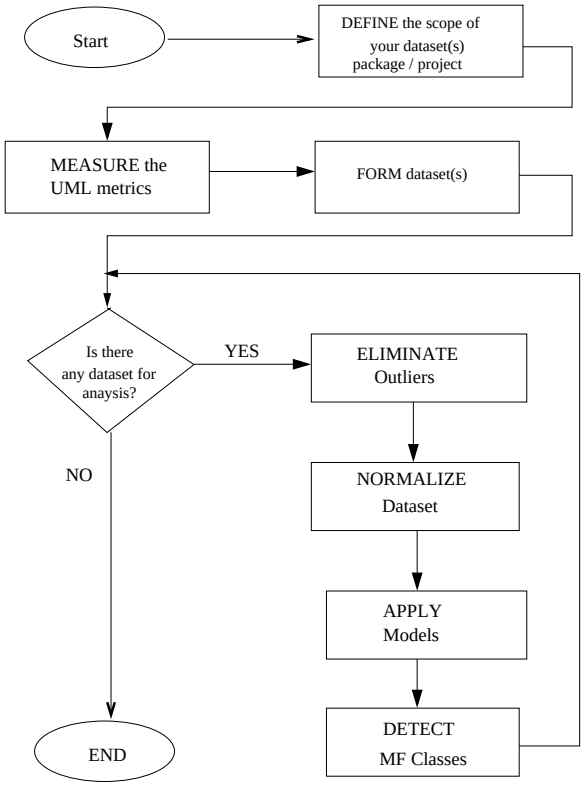

Figure 6.11: Flow Chart for Guidelines

In this Section, we provide a set of simple guidelines to apply the models and methodology exposed in this paper, see Figure 6.11:

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 Chapter 3, Section 3.5.

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 [14].

5. Normalize the datasets according to Chapter 5, Section 5.2.2.

6. Using the resulting normalized datasets, apply the normalized models proposed in Section 6.1.

7. The models will yield the probability of having a MF class within the chosen dataset.

6.5 Conclusions

The results of our empirical study 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, this 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 (different to design-complexity 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 [29]. As we can observe in Tables 6.11, 6.12,6.13 and 6.7, 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 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, for the construction of our prediction models, 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.

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

Table 6.11: RFC: Normalized vs Raw Code Measures

Project	**Correctness**		**Specificity**		**Sensitivity**	
	RFC	**NRFC**	**RFC**	**NRFC**	**RFC**	**NRFC**
ECS	80%	80%	50%	100%	100%	67%
CRS	71%	57%	100%	80%	0%	0%
BNS	67%	33%	100%	50%	0%	0%
MYL-P01	63%	75%	59%	74%	80%	80%
MYL-P02	63%	75%	44%	81%	100%	63%
MYL-P03	72%	72%	78%	78%	43%	43%
MYL-P04	51%	67%	52%	75%	40%	0%
MYL-P05	48%	61%	52%	67%	0%	0%
MYL-P06	61%	85%	56%	83%	10%	100%
MYL-P07	62%	80%	59%	77%	77%	92%
MYL-P08	60%	80%	50%	88%	100%	50%
MYL-P09	75%	81%	74%	84%	83%	63%
MYL-P10	76%	90%	76%	94%	75%	75%
MYL-P11	74%	79%	72%	79%	100%	100%
MYL-P12	87.5%	87.5%	91.7%	91.7%	75%	75%
MYL-P13	56%	88%	50%	86%	100%	100%

Table 6.12: CBO: Normalized vs Raw Code Measures

Project	Correctness		Specificity		Sensitivity	
	CBO	NCBO	CBO	NCBO	CBO	NCBO
ECS	70%	80%	100%	75%	50%	83%
CRS	81%	80%	100%	88%	63%	88%
BNS	67%	89%	100%	88%	63%	88%
MYL-P01	58%	61%	64%	48%	46%	85%
MYL-P02	52%	80%	89%	67%	31%	88%
MYL-P03	56%	56%	56%	56%	50%	50%
MYL-P04	54%	73%	31%	63%	90%	90%
MYL-P05	57%	43%	0%	33%	93%	50%
MYL-P06	64%	68%	35%	57%	92%	79%
MYL-P07	86%	81%	83%	100%	87%	73%
MYL-P08	80%	70%	33%	33%	100%	86%
MYL-P09	68%	73%	40%	57%	91%	86%
MYL-P10	81%	81%	80%	80%	82%	82%
MYL-P11	88%	88%	95%	95%	82%	82%
MYL-P12	81%	81%	78%	78%	83%	83%
MYL-P13	84%	67%	26%	58%	100%	88%

Table 6.13: WMC: Normalized vs Raw Code Measures

Project	Correctness		Specificity		Sensitivity	
	WMC	NWMC	WMC	NWMC	WMC	NWMC
ECS	36%	55%	100%	25%	0%	71%
CRS	88%	56%	100%	57%	0%	50%
BNS	67%	78%	100%	67%	0%	100%
MYL-P01	70%	57%	86%	57%	22%	56%
MYL-P02	79%	76%	77%	69%	81%	81%
MYL-P03	75%	56%	93%	50%	0%	80%
MYL-P04	83%	38%	93%	37%	0%	90%
MYL-P05	83%	39%	90%	43%	0%	0%
MYL-P06	87%	67%	100%	62%	0%	10%
MYL-P07	82%	68%	98%	63%	13%	93%
MYL-P08	80%	50%	100%	50%	0%	50%
MYL-P09	88%	53%	100%	49%	0%	85%
MYL-P10	85%	60%	100%	59%	0%	67%
MYL-P11	85%	56%	100%	49%	0%	100%
MYL-P12	86%	64%	100%	58%	0%	100%
MYL-P13	71%	42%	81%	73%	0%	33%

Chapter 7

Related Work

In this Chapter, we first present some of the research work has been done on how to obtain the CK metrics from UML diagrams. Then, we introduce some related work on fault prediction taking place after the implementation of the code (using code metrics) and before (using elements from the design of the software).

7.1 CK and UML

The few works, that have tried to approximate the CK metrics out of UML diagrams, focus mainly on UML class diagrams. 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 [15], 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 [4]. 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. [53] 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-abstract 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 [37] 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 [38], 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 communication diagrams. Further details are given in a later Section.

7.2 Fault Prediction for Object-Oriented Software

There is a considerable large number of works on this topic. For several years practitioners of fault prediction of object-oriented software have measured design-complexity metrics, such as the CK metrics, from the code to build their prediction models. Another research branch of fault prediction for object-oriented software, in which the number of works are not as large as in the former, focuses on how to make earlier predictions, previous to the implementation of the code. In the following Sections, we briefly summarize some of the works found in the literature in both branches of research work.

7.2.1 After Coding

In Tables 7.1 7.2, we have summarized some of the works on this area. Table 7.1 gives a general description of the study and Table 7.2 gives the results obtained. In Table 7.1, the second column refers to the metrics used as predictors of faulty code, inputs of their prediction models. The third column refers to the measure used of fault-proneness of code, the predicted output. And finally the fourth column of this table refers to the technique applied for the construction of the prediction model. In Table 7.2, the second column refers to those single metrics detected as good predictors of fault-prone code. The third column refers to their best model. And finally, the last column refers to their general results. More in details, these studies are:

1. *A Validation of Object-Oriented Design Metrics as Quality Indicators* by Basili et al. in 1996. The goal of this study was to evaluate the CK metrics as predictors of fault-prone classes. In order to validate their hypothesis, an empirical study was run over four months. Ungraduated and graduated students of the University of Maryland were grouped into eight teams of three students. Each team was asked to developed a medium-sized management information system in C++. The development process of these systems were performed according to a sequential software engineering life-cycle model, derived from the Waterfall model. They collected the CK metrics from source code of the C++ programs delivered at the end of the implementation phase, 180 classes in total. They also collected data about faults found during the testing phase. After the data collection, logistic regression was used to build a model that predicts fault-prone classes.

Their prediction model was built using DIT, RFC, NOC, CBO, and the level of reuse of a class. By using this model, they predicted correctly 138 classes out of 180, which results in a percentage of 76% of correct prediction and they detected 48 faulty classes of 58, which is 82% of faulty classes correctly predicted. This 82% of faulty classes results is a 93% of faults of the entirely system. They found DIT, RFC and NOC to be very significant for the prediction of faulty classes, while CBO was found significant, and WMC was somewhat significant, but not as the previous metrics mentioned. LCOM, on the other hand, was found not to be significant for this study. They concluded that the studied CK metrics are useful predictors of fault-proneness [5].

2. *Exploring the relationships between design measures and software quality* by Briand et al.in 2000. They carried out an empirical study for exploring the relationships between design complexity metrics (28 of coupling, 10 of cohesion, and 11 of inheritance, among them CK metrics) and the probability of fault detection in classes during testing phase. This study used the same data collected in the study previously mentioned by [5]; and logistic regression was also used to build their prediction models.

   Coupling measures, such as RFC, and inheritance measures were found to be strongly related to the probability of fault detection in a class. Multivariate analysis results show that by using some of the coupling and inheritance measures, very accurate models can be derived to detect fault-proneness of classes. Their best model obtained a percentage of correct classifications of about 82%; with this model 81% of faulty classes were correctly detected, which can be translated in a 94% of faults found in the entirely system [9].

3. *O-O Software Quality Prediction using General Regression Neural Networks* by Kanmani et al. in 2004, which discusses the application of general regression neural networks for predicting software quality in terms of fault-proneness of code through an empirical study. The software analyzed in this study was developed by students of the Pondicherry Engineering College. The systems were developed in a period of about 1 month. The students were grouped in 200 teams, each team with one or 2 students. The developed software was in C++. Object design measures (10 of cohesion, 29 of coupling, 18 of inheritance measures) and size measures (7) were collected from the source code, among these measures are the CK suite. Fault-proneness of a class is measured as the fault ratio of the number of test cases not satisfied over the total number of test cases responsible to the class.

   Two groups of metrics were found to perform well at predicting fault-proneness of code. The first group was the size-cohesion-coupling group, and the second group was size-coupling-inheritance group. Both groups obtained an R-square of about 0.93. R-square measures the accuracy of the model, a perfect fit of the model would result in an R-square value of 1 [32].

4. *Early estimation of software quality using in-process testing metrics* by Nachiappan et al. in 2005. The objective of this study is to construct and to validate a set of

metrics called STREW-J (9 metrics. CBO, DIT and WMC among these) that can be used as an early indication of an external measure of field quality. To evaluate the predictive ability of a proposed set of metrics called STREW-J, a case study was carried out in a junior/senior level software engineering course at North Caroline State University in the fall 2003 semester. The system developed was an Eclipse plug-in for collection of static code metrics, written in Java. A total of 22 projects were analyzed, each project was developed by a group of five or six students. Fault proneness of code was measured via black box test, number of failures per KLOC were obtained from 45 test cases. Multiple linear regression was performed to build the prediction model using the STEW-J metrics of fifteen projects. To construct and to test the model, the data collected of the 22 projects was split randomly 7 times. Every time, 15 projects were taken to construct the model, and the remain 7 projects were used to test it. To evaluate the fit of the model, in terms of predictability, it was compute the average absolute error(AAE) and the average relative error(ARE) of the prediction of $number of failures/KLOC$ of the seven different projects not used in the construction of the model. AAE ranks from 2.04 to 7.56 and ARE from 0.82 to 2.10 of the 7 models built. They conclude that the proposed STREW-J metrics are efficient to predict fault-proneness of code [40].

5. *Empirical Validation of Three Software Metrics Suites* by Olague et. al in 2007. This paper describes an empirical validation of three suite of metrics to predict fault-prone classes, which are are: the CK metrics, the Abreu's Metrics for Object-Oriented Design (MOOD) and Bansiya and Davis' Quality Metrics for Object-Oriented Design (QMOOD). Data was collected from six version of the Mozilla Rhino project, an open source software written in Java. The CK-RFC and CK-WMC were found to be consistent predictors of fault-proneness of code. As well as two metrics of the QMOOD set: Class Interface Size (CIS) and Number of Methods (NOM), which is similar to the CK-WMC. Applying multivariate logistic regression was found that three CK models predict more accurately fault-proneness of code, followed by one QMOOD model. The accuracy of the CK models range from 62.6 to 90.6 percent through the six version of Rhino. And the accuracy of the QMOOD model range from 63.5 to 85.2 percent through the six version of Rhino [43].

7.2.2 Before Coding using non UML metrics

We found few works that use metrics from design artifacts to predict faulty code before coding. Only one of these uses a subset of the CK metrics, which were partially obtained from design documents [51]. Further details of these works are given in the following paragraphs.

In [42], 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

Table 7.1: Studies on fault-proneness code prediction

Study	**Input Metrics**	**Output**	**Prediction Technique**
1	CK	Fault-prone classes	Logistic regression
2	49 metrics of coupling,cohesion and inheritance metrics (CK among them)	Fault-prone classes	Logistic Regression
3	CK among 64 metrics of: inheritance, size, cohesion and coupling	Fault ratio	General Regression Neural Network
4	STREW-J suite (3 CK metrics and others)	Fault ratio	Multiple Lineal Regression
5	CK, MOOD, QMOOD	Fault-prone classes	Logistic Regression

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 [1] 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 [51], 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 [48], 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.

In the same year, Yue Jiang et al. [27] 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*. We cannot reverse engineered the code

[1]Specification and Description Language: A modeling language used to describe real-time systems

Table 7.2: Studies on fault-proneness code prediction: Results

Study	Good predictors	Best models	Results
1	- RFC, DIT, NOC (+++) - CBO (++) - WMC (+)	DIT,RFC,NOC,CBO and Level of reuse of a class	- Faulty classes detected 82% - Correct classification 76%
2	- RFC and others coupling (+++) - NOC, DIT and others of inheritance (++)	4 coupling metrics plus 3 of inheritance	Faulty Classes detected 81% Correct classification 82%
3	No details	- 7 size metrics, 10 of cohesion and 29 of coupling - 7 size metrics, 29 of coupling and 18 of inheritance group	R-square = 0.93 (A perfect fit gives: R-square = 1)
4	No details	STREW-J set (CBO, DIT, WMC and others)	AAE: 2.04 - 7.56 , ARE: 0.82 - 2.10 through 7 different models
5	- CK: RFC, WMC - QMOOD: NOM, CIS	- 3 CK Models - 1 QMOOD Model	Through 6 versions of the SW analyzed: - CK Models accuracy: 62.9 - 90.6 % - QMOOD Model accuracy: 63.5 - 82.5 %

(+++) very significant (++) significant (+) somehow significant

to obtain UML communication diagrams, and then measure our UML CK metrics from them, to later predict fault-prone code; this 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 Chapter 4 of this paper.

7.2.3 Before Coding using UML metrics

Concerning the use of UML metrics as predictors of fault-proneness of code, we found the following two related works:

1. Our own previous work published in [10], where we presented the results of our exploratory study on seven UML metrics as predictors of fault-proneness of code. The seven metrics were derived from both UML class and collaboration diagrams, and they were introduced as approximations of two CK metrics and three QMOOD metrics (used in Olague's work [43]). The proposed seven UML metrics were 2 approximations of the RFC CK metric, 2 of the CBO CK metric and 3 more UML approximations of the CIS, NOM, DAM QMOOD metrics (some details of these metrics are given in the following Section).

   Twelve univariate logistic models were built in total using the design and implementation of one of the JAIST projects previously introduced, the ECS project; the number of classes analyzed were 13. Seven from the twelve models were UML based and the rest code based. The dependent variable in the models was the probability of class of being Faulty during the testing phase.

   Our results in this work can be summarized as follows. The prediction accuracy of our UML models ranked from 69% to 91%. In general code metrics could predict better than the proposed UML metrics. One of our approaches to approximate RFC with UML diagrams performed equally to the code RFC metric. And the model using CIS measures from UML class diagrams performed similarly to the model using code measures, being the best UML model in this experiment at detecting fault-prone classes with 91% of accuracy prediction, followed by the previous one mentioned, the UML RFC model with 84.6% accuracy. In this study no normalization method was used.

2. Recently, Nugroho et al. [41] proposed five UML metrics measured from UML class and sequence diagrams to predict faulty classes. Perhaps, this research work is the most similar to ours. Their proposed five UML metrics are classified into *level of detail* and *coupling* metrics. An industrial health-care system written in Java (266 UML classes and 152,017 SLOC) was used to build and test three logistic regression models. One model was built solely with two UML metrics, the second model with code metrics, and the third model with KSLOC and the same two UML metrics used in their UML model.

   From their five proposed UML metrics, only one, *'ImpCoupling'*, resulted significantly correlated to class fault-proneness. Although their UML metric *'SDmsg'* was

not significantly correlated to fault-proneness, it was used along with the *'ImpCoupling'* in their prediction models.

They indicate that their UML metric *'SDmsg'* as a significant predictor when is combined with other metrics, namely *'ImpCoupling'* or KSLOC. And these two metrics were always significant in both univariate and multivariate analysis. Moreover, their hybrid model's prediction accuracy resulted to be the best with 78%, which is 16% above the accuracy percentage of their UML model with 62%. Last but not least, they indicate that log transformations were applied for normalizing their data.

For comparison purposes, the metrics used and prediction results of these two works are further detailed in the following Section.

7.3 Comparison of UML metrics and Prediction Results

As we mentioned in the previous Section, there are only two related works concerning UML metrics and fault prediction. These to works are our own previous work published in [10] (Study 1), and the research work of Nugroho et al. in [41] (Study 2). In the following lines, we give further details of the UML metrics used in both research works and summarized their fault prediction ability, so that we can compare them with the proposed metrics in this paper and their corresponding prediction results.

[2008] Study 1

The UML CBO and RFC metrics proposed in this paper, compare to those ones given in our previous work [10], are more clearly defined and are given in more proper terms (related to messages of communication diagrams): "call messages" and "instantiated methods". In principle our UCBO is the same as the previous defined as CBO-UML2; and as for our URFC, the main difference is that the previous two proposals of approximation, RFC-UML1 and RFC-UML2, still count return messages.

The other three QMOOD metrics used in our previous work were derived from class diagrams, and their definitions are the following:

- CIS, Class Interface Size, is a count of public methods in a class, which UML approximation from class diagrams would be similar to the proposed UWMC metric in this paper.
- DAM, Data Access Metric, is the ratio of the number of private or protected attributes to the total number of attributes declared in a class.
- NOM, Number of Methods.

[2010] Study 2

Although the proposed two UML coupling metrics by Nugroho et al. in [41] are not measured from communication diagrams and were not proposed as approximation of the code CK metrics, at first glance, they seem to be similar to our proposed UML metrics to approximate the code CK metrics.

Their proposed UML coupling metrics are named *'ExCoupling'* and *'ImpCoupling'*, and their respective definitions are:

- *'ExCoupling'* measures the total number of incoming methods invocations $MsgIN$ to a particular object modeled in sequence diagrams. For a given implementation class x, a corresponding design class x' and j number of UML sequence diagrams in which x' appears, *'ExCoupling* of x is measured according to Equation 7.1:

$$ExCoupling(x) = MsgIN_1(x') + MsgIN_2(x')...MsgIN_j(x') \tag{7.1}$$

- *'ImpCoupling'* measures the total number of outgoing methods invocations $MsgOUT$ FROM a particular object modeled in sequence diagrams. For a given implementation class x, a corresponding design class x' and j number of UML sequence diagrams in which x' appears, *'ImpCoupling'* of x is measured according to Equation 7.2:

$$ImCoupling(x) = MsgOUT_1(x') + MsgOUT_2(x')...MsgOUT_j(x') \tag{7.2}$$

Therefore, *'ExCoupling'* is similar to our metric *'NIM'* (Number of Instantiated Methods) and *'ImpCoupling* to our metric *'NIMO'* (Number of Instantiated Methods of Other classes), which are different from those metrics used to approximate the code CK metrics, as defined in Chapter 3.

Moreover, as it was indicated earlier, only *'ImpCoupling'* resulted to be significant predictor of fault-proneness of code. The second UML metric used in their prediction models was the so-called *'SDmsg'*, message detailedness, which is an aggregate metric that measures the level of detail of messages of class instances (objects) modeled in sequence diagrams, *'SDmsg'* can be calculated using Equations 7.3 and 7.4.

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

where:

- i, is a given object is a sequence diagram,

- SD_{ops} measures the RATIO 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 return messages with label TO the total number if messages of the object.

- SD_{par} measures the RATIO of messages with parameters TO the total number of messages of the object.

$$SDMsg(x) = \frac{1}{n}\sum_{i=1}^{n} MsgLoD_i(x') \tag{7.4}$$

where:
- x is the implemented class of a designed class x',
- n is the number of sequence diagrams in which x' appears.

Fault Prediction

To ease the comparison between the previous two studies and our current study, we summarize in Tables 7.3 and 7.4 the UML metrics used and their corresponding prediction results.

The findings of our previous study (Study 1) suggest that UML metrics can be acceptable predictors of fault-proneness of code using UML-based prediction models, which obtained accurate predictions ranking from 69% to 90.9%. The UML metrics were suggested as approximations of the CK CBO and RFC metrics, and of the QMOOD DAM, NOM and CIS metrics. Since only the QMOOD metrics can be approximated using class diagrams, our current research work (contents of this paper) was focused on how to approximate the CK metrics that cannot be measured straightforwardly from the UML class diagrams. The prediction models used, different to our previous study, were code-based, because our initial intentions were to perform multivariate analysis, which due to correlated independent metrics was not possible (as explained in Section 6.3).

If we compare the results of Study 1 with our current results, the different versions of the UML RFC and CBO metrics obtained similar prediction results using either code-based models or UML-based models. As for the UML CIS metric, which in principle measures the same as our UWMC metric, resulted to be better predictor than the latter. This could be explained because CIS is measured from class diagrams, which details more the number of public methods to be developed than communication diagrams.

The prediction accuracy of the UML-based model proposed in Study 2 is slightly lower than those obtained in our studies. Based on our findings, we think that their prediction accuracy could be increased by:

- Using linear scaling to unit variance instead of log transformations to normalize data.
- Using our URFC metric instead of *'ImpCoupling'* of Study 2, for the following reasons. Since, *'ImpCoupling'* is similar to our metric defined as NIMO, our URFC could perform better, because URFC carries with it more information than NIMO. As defined in Chapter 3, URFC is a sum of two metrics NDIM and NDIMO; being NDIMO almost similar to NIMO and in consequence to *'ImpCoupling'* (with the difference that NDIMO counts for different messages), URFC could be better predictor than NIMO metric.

Table 7.3: Fault Prediction using UML Metrics

Study	Model	Normalizatio	Construction Data	Test Data	Accuracy
1.	(U):CBO1	None	13 *ECS* classes [UML]	[S]	69.20%
	(U): CBO2				69.20%
	(U): RFC1				76.92%
	(U): RFC2				84.61%
	(U): CIS				90.9%
	(U): DAM				72.7%
	(U): NOM				72.7%
	(U): CBO	None	13 *ECS* classes [Code]	[S]	92.30%
	(U): RFC				84.61%
	(U): CIS				90.9%
	(U): DAM				36.3%
	(U): NOM				90.9%
	(U):CBO1	None	13 *ECS* classes [UML]	11 *BNS* classes [UML]	72.7%
	(U): CBO2				63.6%
	(U): RFC1				72.7%
	(U): RFC2				72.7%
2.	(M): SDmsg + ImpCoupling	Log	266 *IHS* classes [UML]	[S]	62%
	(M): SDmsg + ImpCoupling + KSLOC		266 *IHS* [UML + Code]		78%
This	(U): RFC	LS	17 MYL-P12 classes [Code]	10 *ECS* classes [UML]	80%
				10 *ECS* classes [Code]	80%
				16 *CRS* classes [UML]	64%
				16 *CRS* classes [Code]	57%
				11 *BNS* classes [UML]	67%
				11 *BNS* classes [Code]	33%

(U): Univariate Model, (M): Multivariate Model, LS: Linear Scaling to Unit Variance

[Code]: Code Measures, [UML]: UML Measures, [UML + Code]: UML and Code Measures, [S]: Same Measures used in the Construction

IHS: Integrated Health-care System, ECS: E-commerce System (JAIST), CRS: Cruise control System (JAIST), BNS: Banking System (JAIST)

Table 7.4: Fault Prediction using UML Metrics

Study	Model	Normalizatio	Construction Data	Test Data	Accuracy
This	(U): CBO	LS	49 MYL-P11 classes [Code]	10 *ECS* classes [UML]	80%
				10 *ECS* classes [Code]	80%
				16 *CRS* classes [UML]	71%
				16 *CRS* classes [Code]	88%
				11 *BNS* classes [UML]	80%
				11 *BNS* classes [Code]	89%
	(U): WMC	LS	30 MYL-P02 classes [Code]	10 *ECS* classes [UML]	45%
				10 *ECS* classes [Code]	55%
				16 *CRS* classes [UML]	38%
				16 *CRS* classes [Code]	56%
				11 *BNS* classes [UML]	44%
				11 *BNS* classes [Code]	78%

(U): Univariate Model, (M): Multivariate Model, LS: Linear Scaling to Unit Variance

IHS: Integrated Health-care System, ECS: E-commerce System (JAIST), CRS: Cruise control System (JAIST), BNS: Banking System (JAIST)

Chapter 8

Other Applications

In this Chapter, we discuss the potential use of our UML metrics in other two areas of research. The first one is on software implementation progress estimations and the second one is on dynamic coupling measures and their relationship with change proneness of code. In the first area, we have started to do some research work, which details can be found in [12].

8.1 Software Implementation Progress Estimations

Scheduling and monitoring progress of software development are crucial activities to meet the deliverable deadlines and, in general, to control a project. One of the reasons software products are delivered late is the inability to recognize on time that the project is falling behind schedule. Specifically talking of software implementation, progress reports are often estimated subjectively by software developers. Such reports are often open to misconception leading to inaccurate schedule estimations [34, 54], which is one of the major causes of software schedule slippage [35]. Therefore, objective and accurate measurements for assessing software implementation progress timely are clearly needed.

The number of Source Lines of Code (SLoC) and used memory in bytes are frequently used to measure the size of a software product [31]. However, assessing the percentage of what is being developed in terms of the same metrics is not possible, because their final values remain unknown till the project is completed.

Considering that the overall functionality of a software is achieved by relating every single part or module composing the software, code metrics measuring such relationship might be able to tell us how much of the totality of the software is being developed, only if, the final values of the same metrics are known beforehand. One of the available object-oriented metrics that can measure a kind of relationship between the different modules of the software is the CK CBO metric. Now, let us say that we would like to use the CBO metric to track software implementation progress, we still would need a *target* value of what must be reached at the end of the implementation in terms of the same metric.

Based on our evaluation results of our UML CBO metric in Chapter 5, Section 5.2.2, which values were close to the final code CBO values in different small-size software projects, we hypothesize that the same UML CBO metric can serve as the sought *target* value to estimate implementation progress readily.

In [12], we discuss our first attempts of estimating software implementation progress readily. Our approach is based on CBO measures using as a baseline UML approximations of the same measures. The assessment of the implementation progress consists, basically, in the normalization of both UML and code CBO measures, so that, as the code approximates to its design, decreasing the error approximation, some progress is "earned".

Our initial results suggest that our methodology can be helpful in the estimation of software implementation progress, which encourages us to perform further improvements of our approach.

8.2 Dynamic Coupling Measures

Coupling has been traditionally measured through structural properties and static *code* analysis. However, because of polymorphism, dynamic binding and the common presence of unused code in commercial software, the resulting coupling measures are imprecise as they do not perfectly reflect the actual coupling taking place among classes at runtime. The type of coupling that can define and precisely measured based on *dynamic* analysis of systems is referred as ***dynamic coupling***. A number of code dynamic coupling measures have been defined and the related to change proneness of classes, their results indicate that such measures are good indicators of change proneness and can be complement of existing static measures for coupling, more information can be found in [3].

Using the same reasoning, it is possible that coupling measures derived from communication diagrams, which reflect the dynamic behavior of a system, could provide better information than those metrics derived from UML static views.

On this investigation line, a very interesting work was recently published by Ah-rim Han et al. [24], who have proposed a compound measure of dynamic coupling that is derived from UML structural and behavioral information. They used such a measure to predict change proneness of classes. From their experiment results, they conclude that although the goodness-of-the-fit of their prediction model is apparently lower than a code-based model, their model can be of great aid in early stages of the development of the software. It might be worth of investigation the application of our UML CBO and RFC metrics, an in general all metrics derived from UML communication diagrams (Chapter 3, Section 3.5), to factors such as change proneness of classes.

Chapter 9

Conclusions and Plans for Future Work

The CK metrics were defined to measure complexity of the design, and they have traditionally been measured from the code and then related to various managerial factors such as productivity, re-work effort for reusing classes and design effort, maintenance effort [14, 20], and fault-proneness of code [5, 9, 30, 32, 40, 43].

In this paper, we proposed a simple approach to approximate CBO, RFC and WMC CK metrics using UML communication diagrams. Special focus on the CBO and RFC was given, because previous research studies reported such approximations difficult to achieve using only class diagrams. Our UML metrics were defined using the Goal/Question/Metric approach. And, an empirical study was carried out to evaluate our metrics, first, as approximations of their corresponding code CK metrics, and second as predictors of fault-prone code.

The results of our empirical study lead us to conclude that the proposed UML RFC and CBO metrics can predict faulty code almost with the same accuracy as their corresponding 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 our prediction results can be improved by using UML-based prediction models, or by the inclusion of other metrics (different to design-complexity metrics) in the predictions models. Such metrics should be easily obtainable before the implementation of the system, with low or zero correlation among them, and strongly related to fault-proneness of code. Moreover, other preprocessing techniques for datasets, such as normalization and over/under sampling, can also be helpful.

There is also the need to explore different methodologies to predict fault-proneness. Recalling the main reason we chose logistic regression as prediction technique is because

this technique, different to other statistical techniques, does not require normally distributed variables, and the resulting 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 in Chapter 6) that can 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.

Our main intentions are to direct others' researchers attention towards our UML metrics, not only as early predictors of faulty code, but also as early measures of design-complexity metrics and/or coupling, which could be used in other research areas such as change proneness of software and software implementation progress, as previously discussed in Chapter 8. Moreover, we hope that the results presented in this paper encourage those who have access to real-life software projects to carry out more experiments, so that external validity of our study can be extended.

In the field of empirical software engineering studies, apart of the internal and external validity of their designed models (in our case our fault-prediction models), there exists another important aspect of validity for the metrics used, which is called *construct validity*. A metric exhibits *construct validity* when it measures what it is purported to measure. When software metrics are defined using plain-language terms, problems can arise because they can be easily misinterpreted, resulting in other different measures to the ones originally intended [45]. On this topic, as complementary work of our study, we present the validation of the construction of our UML metrics in Appendix A.

To sum up, our plans for future work, mainly concern the exploration of other areas of research in which our UML metrics can be applied, and concerning the topic of fault prediction, the following subjects for further study have been considered: data normalization and other pre-processing techniques, the inclusion of other metrics in our prediction models (different to design-complexity metrics), as well as the study of other prediction techniques (different to logistic regression).

List of Tables

List of Figures

Bibliography

[1] Abdelmonem Afifi, Virginia Clark, and Susanne May. *Computer-Aided Multivariate Analysis*. Chapman & Hall/CRC, USA, 2004.

[2] Selim Aksoy and Robert M. Haralick. Feature normalization and likelihood-based similarity measures for image retrieval. *Pattern Recogn. Lett.*, 22(5):563–582, 2001.

[3] E. Arisholm, L.C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on*, 30(8):491 – 506, 2004.

[4] A. Baroni and F. B. Abreu. An OCL based formalization of the MOOSE metrics suite. In *7th ECOOP workshop on quantitative approaches in object oriented softw. eng.*, 2003.

[5] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[6] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 14: Logistic regression. *Critical Care*, 9(1), 2005.

[7] Thirumalesh Bhat and Nachiappan Nagappan. Building scalable failure-proneness models using complexity metrics for large scale software systems. In *APSEC '06: Proc. of the XIII Asia Pacific Softw. Eng. Conference*, pages 361–366, Washington, DC, USA, 2006. IEEE Comp. Soc.

[8] B. Bolstad, R. Irizarray, M. Astrand, and T. Speed. A comparison of normalization methods for high density oligonucleotide array data based on bias and variance. *Bioinformatics*, 19(2):185–193, 2003.

[9] Lionel C. Briand, Jürgen Wüst, John W. Daly, and D. Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51(3):245–273, 2000.

[10] Ana E. Camargo C. and Koichiro Ochimizu. Quality prediction model for object oriented software using UML metrics. In *Proc. of the 4th World Congress for Softw. Quality*, Bethesda, Maryland, USA, 2008. ASQ.

[11] Ana Erika Camargo Cruz and Koichiro Ochimizu. A UML approximation of three chidamber-kemerer metrics and their ability to predict faulty code across software projects. *IEICE TRANSACTIONS on Information and Systems*, E93-D(11):3038–3050, 2010.

[12] Ana Erika Camargo Cruz and Koichiro Ochimizu. Towards objective estimations of software implementation progress. Technical report, IEICE Tech. Rep., March 2011.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[14] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629–639, 1998.

[15] Twan van Enckevort. Refactoring UML models: using openarchitectureware to measure uml model quality and perform pattern matching on UML models with ocl queries. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN Conf. companion on Object oriented programming systems languages and applications*, pages 635–646, New York, NY, USA, 2009. ACM.

[16] Hans-Erik Eriksson, Magnus Penker, and David Fado. *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[17] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999.

[18] George Fernández. *Data mining using SAS applications*. CRC Press, Inc., Boca Raton, FL, USA, 2002.

[19] Marcela Genero, Mario Piatini, and Esperanza Manso. Finding "early" indicators of UML class diagrams understandability and modifiability. In *ISESE '04: Proc. of the 2004 Intl. Symposium on Empirical Softw. Eng.*, pages 207–216, Washington, DC, USA, 2004. IEEE Comp. Soc.

[20] Marcela Genero, Mario Piattini, and Coral Caleron. A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4:59–92, 2005.

[21] Marcela Genero, Mario Piattini, Esperanza Manso, and Giovanni Cantone. Building UML class diagram maintainability prediction models based on early metrics. In *METRICS '03: Proc. of the 9th Intl. Symposium on Softw. Metrics*, page 263, Washington, DC, USA, 2003. IEEE Comp. Soc.

[22] Marcela Genero, Geert Poels, and Mario Piattini. Defining and validating metrics for assessing the understandability of entity-relationship diagrams. *Data Knowl. Eng.*, 64:534–557, March 2008.

[23] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison Wesley-Object Technology Series Editors, Boston, MA, USA, 2000.

[24] Ah-Rim Han, Sang-Uk Jeon, Doo-Hwan Bae, and Jang-Eui Hong. Measuring behavioral dependency for improving change-proneness prediction in UML-based design models. *J. Syst. Softw.*, 83:222–234, February 2010.

[25] David C. Hoaglin, Frederick Mosteller, and John W. Tukey. *Understanading Robust and Exploratory Data Analysis*. John Wiley & Sons, Inc., USA, 2000.

[26] Rumbaugh James, Jacobson Ivar, and Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, Massachusetts, USA, 2006.

[27] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *PROMISE '08: Proc. of the 4th Intl. workshop on Predictor Models in Softw. Eng.*, pages 11–18, New York, NY, USA, 2008. ACM.

[28] I.T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, New York, NY, USA, 1986.

[29] Yasutaka Kamei, Akito Monden, Shinsuke Matsumoto, Takeshi Kakimoto, and Ken-ichi Matsumoto. The effects of over and under sampling on fault-prone module detection. In *ESEM '07: Proc. of the First Intl. Symposium on Empirical Softw. Eng. and Measurement*, pages 196–204, Washington, DC, USA, 2007. IEEE Comp. Soc.

[30] Yasutaka Kamei, Akito Monden, Shuji Morisaki, and Ken-ichi Matsumoto. A hybrid faulty module prediction using association rule mining and logistic regression analysis. In *ESEM '08: Proc. of the Second ACM-IEEE Intl. symposium on Empirical Softw. Eng. and Measurement*, pages 279–281, New York, NY, USA, 2008. ACM.

[31] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[32] S. Kanmani, V. Rhymend Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object oriented software quality prediction using general regression neural networks. *SIGSOFT Softw. Eng. Notes*, 29(5):1–6, 2004.

[33] Bell DeTienne Kristen and Lewis Lee W. Artificial neural networks for the management researcher: The state of the art. *The Forum: A Publication of the Research Methods Division of the Academy of Management*.

[34] Jinhua Li, Zhibing Ma, and Huanzhen Dong. Monitoring software projects with earned value analysis and use case point. *ACIS International Conference on Computer and Information Science*, pages 475–480, 2008.

[35] Z. Ma, J.S. Collofello, and D.E. Smith-Daniels. Causes and solutions for schedule slippage: a survey of software projects. In *Performance, Computing, and Communications Conference, 2000. IPCCC '00. Conference Proceeding of the IEEE International*, pages 373 –379, February 2000.

[36] M Esperanza Manso, Marcela Genero, and Mario Piattini. No-redundant metrics for UML class diagram structural complexity. In *Lecture Notes on Comp. Sci.*, pages 127–142. Springer, 2003.

[37] Jacqueline A. Mc Quillan and James F. Power. A definition of the chidamber and kemerer metrics suite for the unified modeling language. Technical report, Department of Comp. Sci., Co. Kildare, Ireland, 2006.

[38] Jacqueline A. Mc Quillan and James F. Power. On the application of software metrics to UML models. *Models in Softw. Eng.*, pages 217–226, 2007.

[39] Frederick Mosteller and John W. Tukey. *Data Analysis and Regression*. Addison-Wesley Publishing Company, USA, 1977.

[40] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. Early estimation of software quality using in-process testing metrics: a controlled case study. In *3-WoSQ: Proc. of the third workshop on Softw. quality*, pages 1–7, New York, NY, USA, 2005. ACM.

[41] Ariadi Nugroho, Michel R. V. Chaudron, and Erik Arisholm. Assessing UML design metrics for predicting fault-prone classes in a java system. In *MSR*, pages 21–30, 2010.

[42] Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[43] Hector M. Olague, Sampson Gholston, and Stephen Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.*, 33(6):402–419, 2007. Senior Member-Letha H. Etzkorn.

[44] Wendy W. Peng and Dolores R. Wallace. *Software Error Analysis*. U.S. Dept. of Commerce, National Institute of Standards and Technology, Gaithersburg, MD, USA, 1999.

[45] Geert Poels and Guido Dedene. Distance: a framework for software measure construction. Open access publications from katholieke universiteit leuven, Katholieke Universiteit Leuven, 1999.

[46] Linda Rosenberg. Applying and interpreting object oriented metrics. In *Software Assurance Technology Conference*.

[47] Linda Rosenberg. Applying and interpreting object oriented metrics. In *Software Technology Conference*, Utah, USA, 1998.

[48] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *ISESE '06: Proc. of the 2006 ACM/IEEE Intl. symposium on Empirical Softw. Eng.*, pages 18–27, New York, NY, USA, 2006. ACM.

[49] Subhash Sharma. *Applied Multivariate Techniques*. Addison-Wiley & Sons, Inc., USA, 1996.

[50] Rini van Solingen and Egon Berghout. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill Publishing Company, England, UK, 1999.

[51] Ramanath Subramanyam and M.S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, 2003.

[52] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Softw. Eng.*, 10(1):81–104, 2005.

[53] Mei-Huei Tang and Mei-Hwa Chen. Measuring oo design metrics from UML. In *UML '02: Proc. of the 5th Intl. Conference on The Unified Modeling Language*, pages 368–382, London, UK, 2002. Springer-Verlag.

[54] M. P. Ware, F. G. Wilkie, M. Shapcott, and N. G. Lester. The use of product measures in tracking code development to completion within small to medium sized enterprises. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 263–270, Washington, DC, USA, 2008. IEEE Computer Society.

[55] Tong Yi and Fangjun Wu. Empirical analysis of entropy distance metric for UML class diagrams. *SIGSOFT Softw. Eng. Notes*, 29(5):1–6, 2004.

Publications

[1] Camargo Cruz Ana E., Ochimizu Koichiro, "A UML Approximation of Three Chidamber-Kemerer Metrics and Their Ability to Predict Faulty Code across Software Projects", IEICE TRANSACTIONS on Information and Systems, Vol.E93-D,No.11,Nov. 2010, pp. 3038-3050, Japan.

[2] Camargo Cruz Ana E., "Exploratory Study of a UML Metric for Fault Prediction", Proceedings of the 32nd. International Conference on Software Engineering (ICSE 2010), Vol. 2 pp. 361-364, Doctoral Symposium, May 2-8 2010, Cape Town, South Africa.

[3] Camargo Cruz Ana E., Ochimizu Koichiro, "Towards Logistic Regression Models for Predicting Fault-prone Code across Software Projects", Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), IEEE Catalog Number: CFP09ENM, pp. 460-463, Oct 15th-16th 2009, Lake Buena Vista, USA.

[4] Camargo Cruz Ana E., Ochimizu Koichiro, "Quality prediction model for object oriented software using UML metrics", Proceedings of the 4th World Congress for Software Quality, Septemeber 2008, Bethesda, Maryland, USA.

Appendix A

Theoretical Construction of our CK metrics

In the field of empirical software engineering studies, most of the critiques concern their *internal* and *external* validity. In our empirical study, these two types of validity were correctly addressed for our prediction models according to [6]. On the other hand, there exists another important aspect of validity for the metrics used, which is called *construct validity*. A metric exhibits *construct validity* when it measures what it is purported to measure. When software metrics are defined using plain-language terms, problems can arise because they can be easily misinterpreted, resulting in other different measures to the ones originally intended [45].

As complementary work of our empirical study, the contents of this Appendix have as a main purpose to validate the construction of our CK UML metrics, for which the *DISTANCE framework* is applied. The DISTANCE framework has been used to validate the construction of metrics for UML class diagrams [22], and according to its authors, it is specially suited to satisfy the measurement needs of empirical research, offering both the modeling of the software attributes of interest and the definition of the corresponding software metrics, which altogether ensures the construct validity of the resultant metrics [45].

The framework is called DISTANCE as it builds upon the concepts of distance and dissimilarity. It models software attributes as conceptual distances between the software entities they characterize and other software entities that serve as reference points [45]. Suppose that from a 'UML communication diagram' we are interested to know its 'number of different objects'. Being the attribute of interest the 'number of different objects', the software entities that can characterize such an attribute are precisely the 'objects' of the given 'communication diagram'. And the obvious reference point is a 'communication diagram' without 'objects'. Then, once identified the software entities that characterize the attribute of interest and the reference point, distances are measured using some functions that are called 'metrics' in mathematics.

Following the basic example provided by the authors of the DISTANCE framework

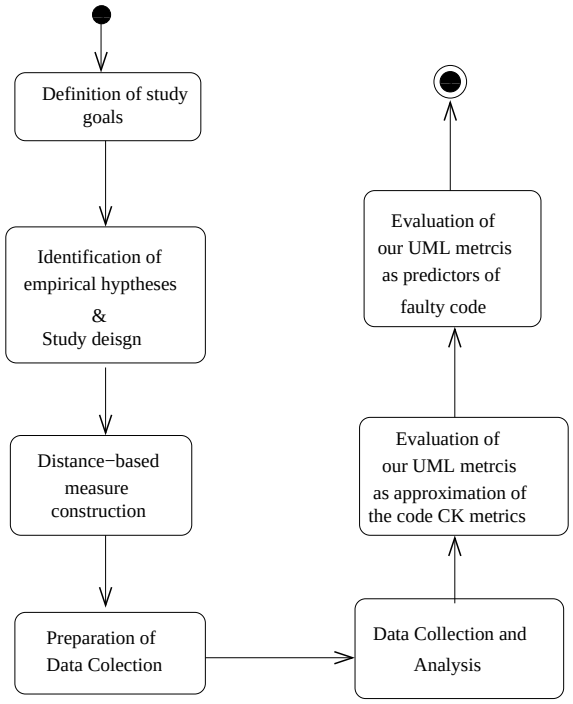

Figure A.1: Distanced-based Measure Construction incorporated in our Empirical Study

approach in [45], we illustrate in Figure A.1 the how this distance-based measure construction can be incorporated into our empirical study. The procedures provided by the DISTANCE framework for attribute modeling and measure construction, are embedded into a goal-oriented approach. The resulting metrics then are further validated through an empirical study.

In this Appendix, we validate theoretically the *construction of our metrics*, defined in Chapter 3, using the DISTANCE Framework approach. First, we validate the construction of the five direct metrics measured from communication diagrams in Table 3.3. Moreover, in order to ease the understanding of such construct validations and definitions, we model and validate the construction of two more metrics (Number of Objects and Number of Senders). Finally, we provide the means for modeling and validate the construction of our second set of UML metrics for classes in Table 3.4.

A.1 The DISTANCE Framework and Number of Objects

This Section has as a main purpose the understanding of the application of the DISTANCE framework approach to validate the construction of software metrics or measures. For such purpose, we follow and apply its five basics steps to validate the construction of the 'Number of Objects' metric, defined in Chapter 3.

The DISTANCE Framework is called that way because it builds upon the concepts

of distance and dissimilarity entities. Think of 'UML communication diagrams' as software entities, and as the attribute of interest their 'numerousness of objects'. Software attributes are modeled as conceptual distances between the software entities they characterize and other software entities that serve as reference points. The distances are then measured by functions that are called 'metrics' in mathematics. To avoid inconsistency with the DISTANCE framework, in this section the term 'metric' refers to such functions, and to refer to any of our UML metrics, the term 'measure' is used instead.

The distance-based measure construction process consists of five basic steps. This process is triggered by a request to find or build a measurement instrument for a software attribute *attr*, such as the 'numerousness of objects'. The five basic steps to validate the construction of its measure 'Number of Objects' $NObjects$ are:

- **Step 1: Find a measurement abstraction**

  Measurement abstractions are used in software measurement to emphasize an attribute of interest, while simultaneously de-emphasizing other attributes.

  Formally in this step, for the **set of software entities** P that are characterized by an **attribute** *attr*, a **set of software entities** M must be found that can be used as measurement abstractions to emphasize *attr* and to define the mapping $abs : P \rightarrow M$.

  In our case, in order to find M let us say that:

  - P is the 'Universe of communication diagrams' $UCmD$ relevant to the Universe of Discourse UoD.
  - *attr*, the attribute of interest, is the 'numerousness of objects'.

  Moreover, we should consider that the 'numerousness of objects' in a communication diagram CmD within $UCmD$ is determined by its 'set of objects' $SObjects(CmD)$. And, if $UObjects$ is the 'Universe of Objects' relevant to UoD, then the 'set of objects' of the 'communication diagram' CmD is a subset of $UObjects$. The set that contains all the 'sets of objects' of the $UCmD$ is the 'power set of $UObjects$', denoted by $\wp(UObjects)$.

  Therefore, the set of software entities M that emphasize the 'numerousness of objects' of **set of software entities** P is the $\wp(UObjects)$. And the abstraction function *abs* results in a projection function that maps a 'communication diagram' CmD onto its 'set of objects' $SObjects(CmD)$.

  $abs_{NObjects} : UCmD \rightarrow \wp(UObjects) : CmD \rightarrow SObjects(CmD)$

  Example: For the communication diagram A in Figure A.2, we have that :

  $$\begin{aligned} abs_{NObjects}(CmDA) &= SObjects(CmDA) \\ &= \{c, m, s, d\} \end{aligned} \tag{A.1}$$

  And as for the communication diagram A' in the same Figure A.2, we have that :

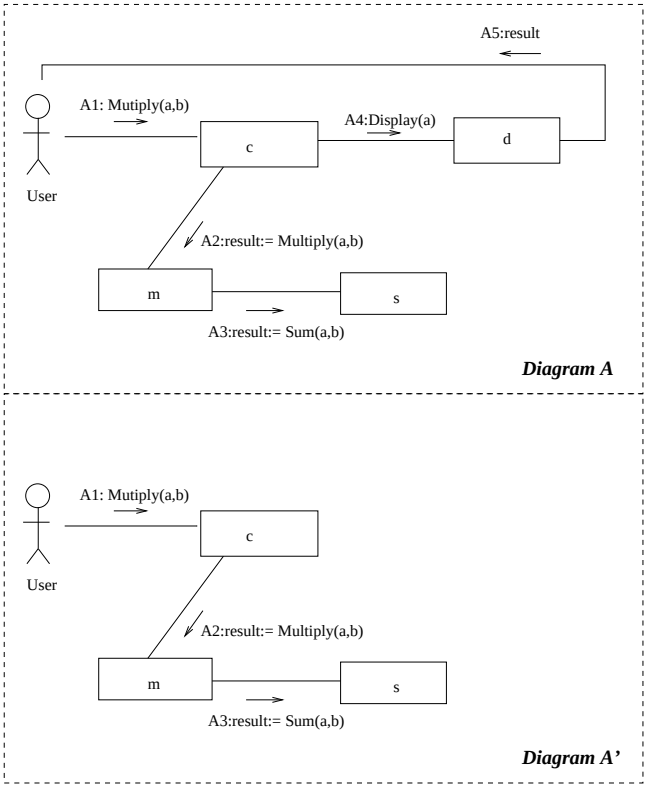

Figure A.2: Example of Different Two Communication Diagrams

$$\begin{aligned} abs_{NObjects}(CmDA') &= SObjects(CmDA') \\ &= \{c, m, s\} \end{aligned} \tag{A.2}$$

- **Step 2: Model distances between measurement abstractions**

  Distances between the elements of M are modeled as sequences of elementary transformations. Such sequences represent a series of atomic changes applied to an element of M to arrive at another element of M. The number of atomic changes that are required to transform one element into the other determines the distance between these elements.

  The formal outcome of this step is a set T_e of elementary transformations on M.

  Continuing with our example, because the elements of M are elements of $\wp(UObjects)$, 'sets of objects', T_e must contain two elementary transformations: one for adding an 'object' to a set and one for removing an 'object' from a set.

  Given two 'sets of objects' s_1 and s_2, where $s_1, s_2 \in \wp(UObjects)$, s_1 can always be transformed into s_2 by removing first all objects from s_1 that are not in s_2, and then adding all objects to s_1 that are not in s_2, but not in the original s_1. In the worst imaginary case, s_1 can be transformed into s_2, via an empty set of interactions.

  Formally, $T_e = \{t_{0NObjects}, t_{1NObjects}\}$, where:

  - $t_{0NObjects} : \wp(UObjects) \rightarrow \wp(UObjects) : s \rightarrow s \cup \{obj\}$, with $obj \in UObjects$
  - $t_{1NObjects} : \wp(UObjects) \rightarrow \wp(UObjects) : s \rightarrow s - \{obj\}$, with $obj \in UObjects$

  Example: The distance between $abs_{NObjects}(CmDA)$ and $abs_{NObjects}(CmDA')$ can be modeled as a sequence of one elementary transformation:

$$\begin{aligned} SObjects(CmDA) &= \{c, m, s, d\} \\ SObjects(CmDA') &= \{c, m, s\} \\ &= t_{1NObjects}(SObjects(CmDA), d) \end{aligned} \tag{A.3}$$

  This is example is represented in Figure A.3.

- **Step 3: Quantify distances between measurement abstractions**

  A metric $\delta : M \times M \rightarrow \Re$ is defined to quantify the distances between the elements of M. Formally, this step results in the definition of a metric space (M, δ).

  In our case, to quantify the distances between elements of M, or $\wp(UObjects)$, we can define the metric $\delta_{NObjects}$ by using the symmetric difference model. The symmetric difference model can always be used to define a metric when the set of measurement abstractions is a power set. Therefore, $\delta_{NObjects}$ is:

  $\delta_{NObjects} : \wp(UObjects) \times \wp(UObjects) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  This definition is equivalent to state that the distance between two 'sets of objects' is measured by the count of elementary transformations in the shortest sequence of elementary transformations between these two sets.

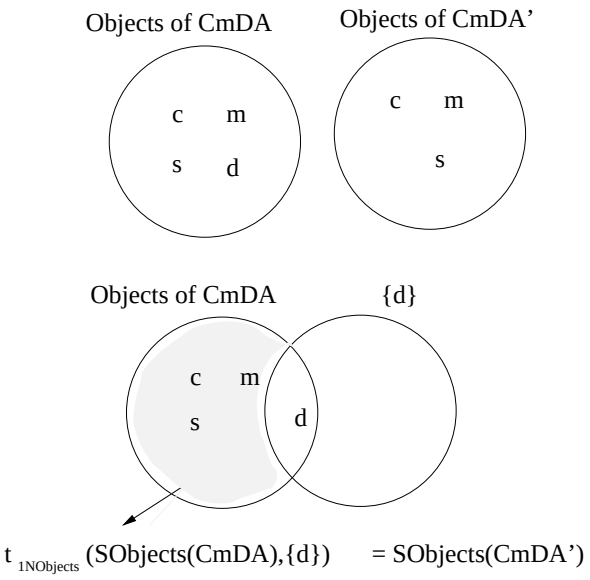

Figure A.3: Transformation of $SObjectes(CmDA) \rightarrow SObjects(CmDA')$

Notice that for any element in s but not in s' and for any element in s' but not in s, an elementary transformation is needed.

Example: Applying the symmetric difference model, the distance between the set of objects of $CmDA$ and $CmDA'$ equals 1:

$$\begin{array}{lcl} \delta_{NObjects}(abs_{NObjects}(CmDA), abs_{NObjects}(CmDA')) & = & \\ |\{c,m,s,d\} - \{c,m,s\}| + |\{c,m,s\} - \{c,m,s,d\}| & = & \\ |\{d\}| + |\emptyset| & = & 1 \end{array} \quad \text{(A.4)}$$

- **Step 4: Find a reference abstraction**

  We need to determine what the model of a software in P must look like in case that entity is characterized by the theoretical lowest value of $attr$. This hypothetical 'null' model or reference model can then be used as reference point or norm for measurement.

  The result of this step is thus the definition of a function ref that returns for each software entity in P a reference abstraction for $attr$ in M; $ref : P \rightarrow M$.

  For our example, the obvious reference point for measurement is the 'empty set of objects', which means that a communication diagram without 'objects' is the lowest possible value for the $NObjects$ measure. Therefore, we define the following function:

  $ref_{NObjects} : UCmD \rightarrow \wp(UObjects) : CmD \rightarrow \emptyset$

- **Step 5. Define the software measure**

  The results of the previous steps are combined to produce a formal definition of the software attribute $attr$ and its measure.

The extent to which *attr* characterizes a software entity $p \in P$ is defined by the distance between the actual model of p for *attr* (i.e. $abs(p)$) and the reference model for *attr* (i.e. $ref(p)$). The larger this distance is, the more the actual measurement of abstraction differs from the norm that has been set, and thus the greater the extent to which *attr* characterizes p. Hence, the value of *attr* for p is the value returned by the metric δ for the pair $(abs(p), ref(p))$.

The formal outcome of this step is the measure $\mu : P \rightarrow \Re$ defined such that $\forall p \in P : \mu(p) = \delta(abs(p), ref(p))$.

For our example, the 'numerousness of objects' of a 'communication diagram' CmD can be defined as the distance between its 'set of objects', $SObjects(CmD)$, and the 'empty set of objects', $\emptyset$. Therefore, its measure 'Number of Objects' $NObjects$ can be defined as a function that returns the value of the metric $\delta_{NObjects}$ for the pair of sets $SObjects(CmD)$ and $\emptyset$ for any communication diagram.

$$\begin{array}{ll} \forall CmD \in UCmD : NObjects(CmD) & = \\ \delta_{NObjects}(SObjects(CmD), \emptyset) & = \\ |SObjects(CmD) - \emptyset| + |\emptyset - SObjects(CmD)| & = \\ |SObjects(CmD)| & \end{array} \tag{A.5}$$

Consequently, a measure that returns the count of 'objects' of a communication diagram qualifies as a measure of the 'numerousness of objects'.

Example: The 'Number of Objects' of the 'communication diagram A' $CmDA$ in Figure A.2 equals 4:

$$\begin{array}{ll} \delta_{NObjects}(SObjects(CmDA), \emptyset) & = \\ |\{c, m, s, d\} - \emptyset| + & \\ |\emptyset - \{c, m, s, d\}| & = \\ |\{c, m, s, d\}| + |\emptyset| & = \\ 4 & \end{array} \tag{A.6}$$

The details underlying the theoretical principles of the DISTANCE framework can found in [45].

A.2 For an Object: Number of Received Call Messages

- **Step 1: Find a measurement abstraction**

  The set of software entities modeled is the 'Universe of Objects' within a communication diagram, $UObjects$, that are relevant to the Universe of Discourse UoD.

  The attribute of interest are: *The numerousness of Received Call Messages.*

  A call message can be distinguished from another, if its sequence expression of its label are different, e.g. the call message $A1 : print()$ is different to the call message $A2 : print()$.

Let be $UObjects$ as the 'Universe of Objects' relevant to UoD. All the 'sets of Received Call Messages' of an object of $UObjects$ are elements of the 'power set of $URCMsgs$', denoted by $\wp(URCMsgs)$.

Therefore, the set of software entities that can be used as measurement abstractions is $\wp(URCMsgs)$, and the abstraction function abs results in a projection function that maps 'objects' onto their 'sets of Received Call Messages':

$abs_{NRCMsgs} : UObjects \rightarrow \wp(URCMsgs) : Obj \rightarrow SRCMsgs(Obj)$

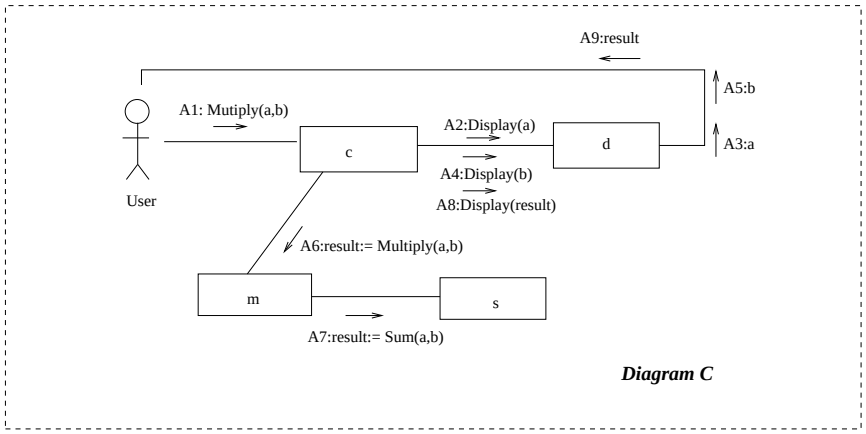

Figure A.4: Example of Different Two Communication Diagrams

Example: For the communication diagram C in Figure A.4, we have that :

$$\begin{aligned} abs_{NRCMsgs}(c) &= SRCMsgs(c) \\ &= \{A1\} \end{aligned} \tag{A.7}$$

$$\begin{aligned} abs_{NRCMsgs}(m) &= SRCMsgs(m) \\ &= \{A6\} \end{aligned} \tag{A.8}$$

$$\begin{aligned} abs_{NRCMsgs}(s) &= SRCMsgs(s) \\ &= \{A7\} \end{aligned} \tag{A.9}$$

$$\begin{aligned} abs_{NRCMsgs}(d) &= SRCMsgs(d) \\ &= \{A2, A4, A8\} \end{aligned} \tag{A.10}$$

- **Step 2: Model distances between measurement abstraction**

  As explained previously, in our first validation, since the elements of $\wp(URCMsgs)$ are 'sets of Received Call Messages', T_e must contain two elementary transformations: one for adding a 'Received Call Messages' to *a set* and one for removing a 'Received Call Message' from *a set*. Formally, $T_e = \{t_{0NRCMsgs}, t_{1NRCMsgs}\}$, where:

  - $t_{0NRCMsgs} : \wp(URCMsgs) \rightarrow \wp(URCMsgs) : s \rightarrow s \cup \{rcmsg\}$, with $rcmsg \in URCMsgs$

– $t_{1NRCMsgs} : \wp(URCMsgs) \rightarrow \wp(URCMsgs) : s \rightarrow s - \{rcmsg\}$, with $rcmsg \in URCMsgs$

Example: The distance between $abs_{NRCMsgs}(c)$ and $abs_{NRCMsgs}(d)$ can be modeled as follows. Given that:

$$\begin{array}{lcl} SRCMsgs(c) & = & \{A1\} \\ SRCMsgs(d) & = & \{A2, A4, A8\} \end{array} \tag{A.11}$$

Recalling that for any element in $SRCMsgs(c)$ and not in $SRCMsgs(d)$ and vice-versa, an elementary transformation is needed, the sequence of elementary transformations needed to model the distance between $abs_{NRCMsgs}(c)$ and $abs_{NRCMsgs}(d)$ are equal to 4:

$$\begin{array}{l} SRCMsgs(d) \\ t_{0NRCMsgs}(t_{0NRCMsgs}(t_{0NRCMsgs}(t_{1NRCMsgs}(SRCMsgs(c), A1), A2), A4), A8) \end{array} = \tag{A.12}$$

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NRCMsgs}$ that models the distances in $\wp(URCMsgs)$ is defined as:

  $\delta_{NRCMsgs} : \wp(URCMsgs) \times \wp(URCMsgs) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the 'sets of Received Call Messages' of c and d equals 4:

$$\begin{array}{lcl} \delta_{NRCMsgs}(abs_{NRCMsgs}(c), abs_{NRCMsgs}(d)) & = & \\ |\{A1\} - \{A2, A4, A8\}| + |\{A2, A4, A8\} - \{A1\}| & = & \\ |\{A1\}| + |\{A2, A4, A8\}| & = & 4 \end{array} \tag{A.13}$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the 'empty set of received call messages', which means that an 'object' without 'received call messages' is the lowest possible value for the $NRCMsgs$ measure. Therefore, the reference abstraction function can be defined as:

  $ref_{NRCMsgs} : UObjects \rightarrow \wp(URCMsgs) : Obj \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Received Call Messages' of a given 'object' Obj, can be defined as the distance between its 'set of received call messages' $SRCMsgs(Obj)$, and the 'empty set of received call messages' $\emptyset$. Therefore, its measure 'Number of Received Call Messages' $NRCMsgs$ can be defined as a function that returns the value of the metric $\delta_{NRCMsgs}$ for the pair of sets $SRCMsgs(object)$ and $\emptyset$ for any object.

$$\begin{array}{lcl} \forall Obj \in UObjects : NRCMsgs(Obj) & = & \\ \delta_{NRCMsgs}(SRCMsgs(Obj), \emptyset) & = & \\ |SRCMsgs(Obj) - \emptyset| + |\emptyset - SRCMsgs(Obj)| & = & \\ |SRCMsgs(Obj)| & & \end{array} \tag{A.14}$$

Consequently, a measure that returns the count of 'received call messages' of an 'object' qualifies as a measure of the 'numerousness of Received Call Messages'.

Example: The 'Number of Received Call Messages' of object d in diagram C of Figure A.4 equals 3:

$$\begin{array}{lcl} \delta_{NRCMsgs}(SRCMsgs(d), \emptyset) & = & \\ |\{A2, A4, A8\} - \emptyset| + & & \\ |\emptyset - \{A2, A4, A8\}| & = & \\ |\{A2, A4, A8\}| + |\emptyset| & = & \\ 3 & & \end{array} \tag{A.15}$$

A.3 For an Object: Number of Received Call Messages

- **Step 1: Find a measurement abstraction**

  The set of software entities modeled is the 'Universe of Objects' in a communication diagram, $UObjects$, that are relevant to the Universe of Discourse UoD.

  The attribute of interest are: *the numerousness of Different Received Call Messages*.

  A call message can be distinguished from another, if their names are different, e.g. the call message $A1 : print()$ is exactly the same as the call message $A2 : print()$.

  Let be $UObjects$ as the 'Universe of Objects' relevant to UoD. All the 'sets of Different Received Call Messages' of an object of $UObjects$ are elements of the power set of $UDRCMsgs$, denoted by $\wp(UDRCMsgs)$.

  Therefore, the set of software entities that can be used as measurement abstractions is $\wp(UDRCMsgs)$, and the abstraction function *abs* results in a projection function that maps 'objects' onto their 'sets of Different Received Call Messages':

  $abs_{NDRCMsgs} : UObjects \rightarrow \wp(UDRCMsgs) : Obj \rightarrow SDRCMsgs(Obj)$

  Example: For the communication diagram C in Figure A.4, we have that :

$$\begin{array}{rcl} abs_{NDRCMsgs}(c) & = & SDRCMsgs(c) \\ & = & \{A1\} \end{array} \tag{A.16}$$

$$\begin{array}{rcl} abs_{NDRCMsgs}(m) & = & SDRCMsgs(m) \\ & = & \{A6\} \end{array} \tag{A.17}$$

$$\begin{array}{rcl} abs_{NDRCMsgs}(s) & = & SDRCMsgs(s) \\ & = & \{A7\} \end{array} \quad (A.18)$$

$$\begin{array}{rcl} abs_{NDRCMsgs}(d) & = & SDRCMsgs(d) \\ & = & \{A2 = A4 = A8\} \end{array} \quad (A.19)$$

Notice the difference between the $abs_{NDRCMsgs}(d)$ and $abs_{NRCMsgs}(d)$ defined in the previous Section.

- **Step 2: Model distances between measurement abstraction**

  Just as in the validation of the construction of our previous measures, since the elements of $\wp(UDRCMsgs)$ are 'sets of Different Received Call Messages', T_e must contain two elementary transformations: one for adding a 'Different Received Call Messages' to a set and one for removing a 'Different Received Call Message' from a set. Formally, $T_e = \{t_{0NDRCMsgs}, t_{1NDRCMsgs}\}$, where:

  - $t_{0NDRCMsgs} : \wp(UDRCMsgs) \rightarrow \wp(UDRCMsgs) : s \rightarrow s \cup \{drcmsg\}$, with $drcmsg \in UDRCMsgs$
  - $t_{1NDRCMsgs} : \wp(UDRCMsgs) \rightarrow \wp(UDRCMsgs) : s \rightarrow s - \{drcmsg\}$, with $drcmsg \in UDRCMsgs$

  Example: The distance between $abs_{NDRCMsgs}(c)$ and $abs_{NDRCMsgs}(m)$ can be modeled as follows. Given that:

$$\begin{array}{rcl} SDRCMsgs(c) & = & \{A1\} \\ SDRCMsgs(m) & = & \{A6\} \end{array} \quad (A.20)$$

  Recalling that for any element in $SDRCMsgs(c)$ and not in $SDRCMsgs(m)$ and vice-versa, an elementary transformation is needed, the sequence of elementary transformations needed to model the distance between $abs_{NDRCMsgs}(c)$ and $abs_{NDRCMsgs}(m)$ equals 2.

$$\begin{array}{l} SDRCMsgs(m) \\ t_{0NDRCMsgs}(t_{1NDRCMsgs}(SDRCMsgs(c), A1), A6) \end{array} = \quad (A.21)$$

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NDRCMsgs}$ that models the distances in $\wp(UDRCMsgs)$ is:

  $\delta_{NDRCMsgs} : \wp(UDRCMsgs) \times \wp(UDRCMsgs) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the 'sets of Different Received Call Messages' of c and m equals 2:

$$\begin{array}{rcl} \delta_{NDRCMsgs}(abs_{NDRCMsgs}(c), abs_{NDRCMsgs}(m)) & = & \\ |\{A1\} - \{A6\}| + |\{A6\} - \{A1\}| & = & \\ |\{A1\}| + |\{A6\}| & = & 2 \end{array} \quad (A.22)$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the empty set of 'different received call messages', which means that an 'object' without 'different received call messages' is the lowest possible value for the $NDRCMsgs$ measure. Therefore, the reference abstraction function can be defined as:

  $ref_{NDRCMsgs} : UObjects \rightarrow \wp(UDRCMsgs) : Obj \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Different Received Call Messages' of a given 'object', Obj, can be defined as the distance between its 'set of different received call messages', $SDRCMsgs(Obj)$, and the 'empty set of different received call messages', $\emptyset$. Therefore, its measure 'Number of Different Received Call Messages' $NDRCMsgs$ can be defined as a function that returns the value of the metric $\delta_{NDRCMsgs}$ for the pair of sets $SDRCMsgs(Obj)$ and $\emptyset$ for any object.

$$\begin{array}{ll} \forall Obj \in UObjects : NDRCMsgs(Obj) & \\ \delta_{NDRCMsgs}(SDRCMsgs(Obj), \emptyset) & = \\ |SDRCMsgs(Obj) - \emptyset| + |\emptyset - SDRCMsgs(Obj)| & = \\ |SDRCMsgs(Obj)| & \end{array} \tag{A.23}$$

  Consequently, a measure that returns the count of 'different received call messages' of an 'object' qualifies as a measure of the 'numerousness of Different Received Call Messages'.

  Example: The 'Number of Different Received Call Messages' d object equals 1:

$$\begin{array}{ll} \delta_{NDRCMsgs}(SRCMsgs(d), \emptyset) & = \\ |\{A2 = A4 = A8\} - \emptyset| + & \\ |\emptyset - \{A2 = A4 = A8\}| & = \\ |\{A2 = A4 = A8\}| + |\emptyset| & = \\ 1 & \end{array} \tag{A.24}$$

Notice the difference between this metric and the $\delta_{NRCMsgs}$ for the same object in the previous Section.

A.4 For a Communication Diagram: Number of Senders

- **Step 1: Find a measurement abstraction**

  The set of software entities modeled is the 'Universe of Communication Diagrams', $UCmD$, that are relevant to the Universe of Discourse UoD.

  The attribute of interest are: *The numerousness of objects or actors that act as senders.*

Let be $USenders$ as the 'Universe of Sender Objects' relevant to UoD. All the 'sets of senders' within the $UCmD$ are elements of the 'power set of $USenderes$', denoted by $\wp(USenders)$.

Therefore, the set of software entities that can be used as measurement abstractions is $\wp(USenders)$, and the abstraction function abs results in a projection function that maps 'communication diagrams' onto their 'sets of senders':

$abs_{NSenders} : UCmD \rightarrow \wp(USenders) : CmD \rightarrow SSenders(CmD)$

Example: For the communication diagram A in Figure A.2, we have that :

$$\begin{aligned} abs_{NSenders}(CmDA) &= SSenders(CmDA) \\ &= \{User, c, m, d\} \end{aligned} \tag{A.25}$$

And as for the communication diagram A' in the same Figure A.2, we have that :

$$\begin{aligned} abs_{NSenders}(CmDA') &= SSenders(CmDA') \\ &= \{User, c, m\} \end{aligned} \tag{A.26}$$

- **Step 2: Model distances between measurement abstraction**

  Just as in the construction validation of our previous measures, since the elements of $\wp(USenders)$ are 'sets of senders', T_e must contain two elementary transformations: one for adding a 'sender' to a set and one for removing a 'sender' from a set. Formally, $T_e = \{t_{0NSenders}, t_{1NSenders}\}$, where:

  - $t_{0NSenders} : \wp(USenders) \rightarrow \wp(USenders) : s \rightarrow s \cup \{sender\}$, with $sender \in USenders$
  - $t_{1NSenders} : \wp(USenders) \rightarrow \wp(USenders) : s \rightarrow s - \{sender\}$, with $sender \in USenders$

  Example: The distance between $abs_{NSenders}(CmDA)$ and $abs_{NSenders}(CmDA')$ in Figure A.2 can be modeled as follows:

$$\begin{aligned} SSenders(CmDA) &= \{User, c, m, d\} \\ SSenders(CmDA') &= \{User, c, m\} \\ &= t_{1NSenders}(SSenders(CmDA), d) \end{aligned} \tag{A.27}$$

  Therefore, the number of elementary transformations needed to model the distance between $abs_{NSenders}(CmDA)$ and $abs_{NSenders}(CmDA')$ equals 1.

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NSenders}$ that models the distances in $\wp(USenders)$ is:

  $\delta_{NSenders} : \wp(USenders) \times \wp(USenders) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the 'sets of senders' of $CmDA$ and $CmDA'$ equals 1:

$$\begin{array}{lll} \delta_{NSenders}(abs_{NSenders}(CmDA), abs_{NSenders}(CmDA')) & = & \\ |\{User, c, m, d\} - \{User, c, m\}| + |\{User, c, m\} - \{User, c, m, d\}| & = & \\ |\{d\}| + |\emptyset| & = & 1 \end{array} \tag{A.28}$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the 'empty set of senders', which means that a communication diagram without 'senders' is the lowest possible value for the $NSenders$ measure. Therefore, the reference abstraction function can be defined as:

  $ref_{NSenders} : UCmD \rightarrow \wp(USenders) : CmD \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Senders' in a communication diagram, CmD, can be defined as the distance between its 'set of senders', $SSenders(CmD)$, and the 'empty set of senders', $\emptyset$. Therefore, its measure 'Number of Senders' $NSenders$ can be defined as a function that returns the value of the metric $\delta_{NSenders}$ for the pair of sets $SSenders(CmD)$ and $\emptyset$ for any communication diagram.

$$\begin{array}{lll} \forall CmD \in UCmD : NSenders(CmD) & = & \\ \delta_{NSenders}(SSenders(CmD), \emptyset) & = & \\ |SSenders(CmD) - \emptyset| + |\emptyset - SSenders(CmD)| & = & \\ |SSenders(CmD)| & & \end{array} \tag{A.29}$$

  Consequently, a measure that returns the count of 'senders' in a communication diagram qualifies as a measure of the 'numerousness of Senders'.

  Example: The 'Number of Senders' in communication diagram A, $CmDA$, in Figure A.2 equals 4:

$$\begin{array}{lll} \delta_{NSenders}(SSenders(mDA), \emptyset) & = & \\ |\{User, c, m, d\} - \emptyset| + & & \\ |\emptyset - \{User, c, m, d\}| & = & \\ |\{d\}| + |\emptyset| & = & \\ 4 & & \end{array} \tag{A.30}$$

A.5 For an Object: The Number of Sent Call Messages

- **Step 1: Find a measurement abstraction**

  The set of entities modeled is the 'Universe of Senders' (objects or actors), $USenders$, in a communication diagram that are relevant to the Universe of Discourse UoD.

  The attribute of interest is: *The numerousness of Sent Call Messages.*

A call message can be distinguished from another, if its sequence expression of its label are different, e.g. the call message $A1 : print()$ is different to the call message $A2 : print()$.

Let be $USCMsgs$ as the 'Universe of Sent Call Messages' relevant to UoD. All the 'sets of Sent Call Messages' within the $USenders$ are elements of the 'power set of $USCMsgs$', denoted by $\wp(USCMsgs)$.

Therefore, the set of software entities that can be used as measurement abstractions is $\wp(USCMsgs)$, and the abstraction function abs results in a projection function that maps 'senders', objects or actors, onto their 'sets of Sent Call Messages':

$abs_{NSCMsgs} : USenders \rightarrow \wp(USCMsgs) : sender \rightarrow SSCMsgs(sender)$

Example: For the communication diagram C in Figure A.4, we have that :

$$\begin{aligned} abs_{NSCMsgs}(User) &= SSCMsgs(User) \\ &= \{A1\} \end{aligned} \tag{A.31}$$

$$\begin{aligned} abs_{NSCMsgs}(c) &= SSCMsgs(c) \\ &= \{A2, A4, A6, A8\} \end{aligned} \tag{A.32}$$

$$\begin{aligned} abs_{NSCMsgs}(m) &= SSCMsgs(m) \\ &= \{A7\} \end{aligned} \tag{A.33}$$

$$\begin{aligned} abs_{NSCMsgs}(s) &= SSCMsgs(s) \\ &= \emptyset \end{aligned} \tag{A.34}$$

$$\begin{aligned} abs_{NSCMsgs}(d) &= SSCMsgs(d) \\ &= \emptyset \end{aligned} \tag{A.35}$$

Notice that messages $A3, A5$ and $A9$ are not call messages.

- **Step 2: Model distances between measurement abstraction**

  Since the elements of $\wp(USCMsgs)$ are 'sets of senders', T_e must contain two elementary transformations: one for adding a 'Sent Call Message' to a set and one for removing a 'Sent Call Message' from a set. Formally, $T_e = \{t_{0NSCMsgs}, t_{1NSCMsgs}\}$, where:

  - $t_{0NSCMsgs} : \wp(USCMsgs) \rightarrow \wp(USCMsgs) : s \rightarrow s \cup \{scmsg\}$, with $scmsg \in USCMSgs$
  - $t_{1NSCMsgs} : \wp(USCMsgs) \rightarrow \wp(USCMsgs) : s \rightarrow s - \{scmsg\}$, with $scmsg \in USCMsgs$

  Example: The distance between $abs_{NSCMsgs}(c)$ and $abs_{NSCMsgs}(s)$ can be modeled as follows:

$$\begin{array}{lcl} SSCMsgs(s) & = & \emptyset \\ SSCMsgs(c) & = & \{A2, A4, A6, A8\} \\ & = & t_{0NSCMsgs}(t_{0NSCMsgs}(t_{0NSCMsgs}(t_{0NSCMsgs}(SSCMsgs(s), A2), A4), A6), A8) \end{array} \quad (A.36)$$

Therefore, a total of 4 elementary transformations are sufficient to model the distance between $abs_{NSCMsgs}(c)$ and $abs_{NSCMsgs}(s)$.

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NSCMsgs}$ that models the distances in $\wp(USCMsgs)$ is:

  $\delta_{NSCMsgs} : \wp(USCMsgs) \times \wp(USCMsgs) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the sets of Sent Call Messages of s and d equals 4:

$$\begin{array}{lcl} \delta_{NSCMsgs}(abs_{NSCMsgs}(s), abs_{NSCMsgs}(d)) & = & \\ |\emptyset - \{A2, A4, A6, A8\}| + |\{A2, A4, A6, A8\} - \emptyset| & = & \\ |\emptyset| + |\{A2, A4, A6, A8\}| & = & 4 \end{array} \quad (A.37)$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the empty set of 'Sent Call Messages', which means that a sender without 'Sent Call Messages' is the lowest possible value for the $NSCMsgs$ measure. Therefore, the reference abstraction function can be defined as:

  $ref_{NSCMsgs} : USenders \rightarrow \wp(USCMsgs) : sender \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Sent Call Messages' of a 'sender', $sender$, can be defined as the distance between its 'set of Sent Call Messages', $SSenders(sender)$, and the 'empty set of Sent Call Messages', $\emptyset$. Therefore, its measure 'Number of Sent Call Messages' $NSCMsgs$ can be defined as a function that returns the value of the metric $\delta_{NSCMsgs}$ for the pair of sets $SSCMsgs(sender)$ and $\emptyset$:

$$\begin{array}{l} \forall sender \in USenders : NSCMsgs(sender) \\ \delta_{NSCMsgs}(SSCMsgs(sender), \emptyset) = \\ |SSCMsgs(sender) - \emptyset| + |\emptyset - SSCMsgs(sender)| = \\ |SSCMsgs(sender)| \end{array} \quad (A.38)$$

  Consequently, a measure that returns the count of 'Sent Call Messages' of a 'sender' in a communication diagram qualifies as a measure of the 'numerousness of Sent Call Messages'.

  Example: The 'Number of Sent Call Messages' of c in diagram C of Figure A.4 equals 4:

$$\begin{array}{lcl} \delta_{NSCMsgs}(SSCMsgs(c), \emptyset) & = & \\ |\{A2, A4, A6, A8\} - \emptyset| + & & \\ |\emptyset - \{A2, A4, A6, A8\}| & = & \\ |\{A2, A4, A6, A8\}| + |\emptyset| & = & \\ 4 & & \end{array} \quad (A.39)$$

A.6 For an Object: The Number of Different Sent Call Messages

- **Step 1: Find a measurement abstraction**

  The set of entities modeled is the 'Universe of Senders', actors or objects, $USenders$, that are relevant to the Universe of Discourse UoD.

  The attribute of interest is: *The numerousness of Different Sent Call Messages*.

  A call message can be distinguished from another, if their names are different, e.g. the call message $A1 : print()$ is exactly the same as the call message $A2 : print()$.

  Let be $UDSCMsgs$ as the 'Universe of Different Sent Call Messages' relevant to UoD. All the 'sets of Different Sent Call Messages' within the $USenders$ are elements of the power set of $UDSCMsgs$, denoted by $\wp(UDSCMsgs)$.

  Therefore, the set of software entities that can be used as measurement abstractions is $\wp(UDSCMsgs)$, and the abstraction function abs results in a projection function that maps 'senders', objects or actors, onto their 'sets of Different Sent Call Messages':

  $abs_{NDSCMsgs} : USenders \rightarrow \wp(UDSCMsgs) : sender \rightarrow SDSCMsgs(sender)$

  Example: For the communication diagram C in Figure A.4, we have that :

$$\begin{array}{rcl} abs_{NDSCMsgs}(User) & = & SDSCMsgs(User) \\ & = & \{A1\} \end{array} \quad (A.40)$$

$$\begin{array}{rcl} abs_{NDSCMsgs}(c) & = & SDSCMsgs(c) \\ & = & \{A2 = A4 = A8, A6\} \end{array} \quad (A.41)$$

$$\begin{array}{rcl} abs_{NDSCMsgs}(m) & = & SDSCMsgs(m) \\ & = & \{A6\} \end{array} \quad (A.42)$$

$$\begin{array}{rcl} abs_{NDSCMsgs}(s) & = & SDSCMsgs(s) \\ & = & \emptyset \end{array} \quad (A.43)$$

$$\begin{array}{rcl} abs_{NDSCMsgs}(d) & = & SDSCMsgs(d) \\ & = & \emptyset \end{array} \quad (A.44)$$

  Notice the difference between $abs_{NDSCMsgs}(c)$ and $abs_{NDSCMsgs}(c)$ defined in the previous Section.

- **Step 2: Model distances between measurement abstraction**

  Since the elements of $\wp(UDSCMsgs)$ are 'sets of senders', T_e must contain two elementary transformations: one for adding a 'Different Sent Call Message' to a set and one for removing a 'Different Sent Call Message' from a set. Formally, $T_e = \{t_{0NDSCMsgs}, t_{1NDSCMsgs}\}$, where:

  - $t_{0NDSCMsgs} : \wp(UDSCMsgs) \rightarrow \wp(UDSCMsgs) : s \rightarrow s \cup \{dscmsg\}$, with $dscmsg \in UDSCMSgs$
  - $t_{1NDSCMsgs} : \wp(UDSCMsgs) \rightarrow \wp(UDSCMsgs) : s \rightarrow s - \{dscmsg\}$, with $dscmsg \in UDSCMsgs$

  Example: The distance between $abs_{NDSCMsgs}(c)$ and $abs_{NDSCMsgs}(s)$ can be modeled by the following sequence of elementary transformations:

$$\begin{array}{lcll} SDSCMsgs(s) & = & \emptyset & \\ SDSCMsgs(c) & = & \{A2 = A4 = A8, A6\} & \\ & = & t_{0NDSCMsgs}(t_{0NDSCMsgs}(SDSCMsgs(s), A2), A6) & \end{array} \quad \text{(A.45)}$$

  Therefore, a sequence of 2 elementary transformations is sufficient to model the distance between $abs_{NDSCMsgs}(c)$ and $abs_{NDSCMsgs}(s)$.

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NDSCMsgs}$ that models the distances in $\wp(UDSCMsgs)$ is:

  $\delta_{NDSCMsgs} : \wp(UDSCMsgs) \times \wp(UDSCMsgs) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the sets of Different Sent Call Messages of s and c equals 2:

$$\begin{array}{lcl} \delta_{NDSCMsgs}(abs_{NDSCMsgs}(s), abs_{NDSCMsgs}(c)) & = & \\ |\emptyset - \{A2, A6\}| + |\{A2, A6\} - \emptyset| & = & \\ |\emptyset| + |\{A2, A6\}| & = & 2 \end{array} \quad \text{(A.46)}$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the 'empty set of Different Sent Call Messages', which means that a sender without 'Different Sent Call Messages' is the lowest possible value for the $NDSCMsgs$ measure. Therefore, the reference abstraction function can be defined as follows:

  $ref_{NDSCMsgs} : USenders \rightarrow \wp(UDSCMsgs) : sender \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Different Sent Call Messages' of a 'sender', $sender$, can be defined as the distance between its 'set of Different Sent Call Messages', $SDSCMsgs(sender)$,

and the 'empty set of Different Sent Call Messages', $\emptyset$. Therefore, its measure 'Number of Different Sent Call Messages' $NDSCMsgs$ can be defined as a function that returns the value of the metric $\delta_{NDSCMsgs}$ for the pair of sets $SDSCMsgs(sender)$ and $\emptyset$:

$$\begin{array}{l} \forall sender \in USenders : NDSCMsgs(sender) \\ \delta_{NDSCMsgs}(SDSCMsgs(sender), \emptyset) \quad = \\ |SDSCMsgs(sender) - \emptyset| + |\emptyset - SDSCMsgs(sender)| \quad = \\ |SDSCMsgs(sender)| \end{array} \tag{A.47}$$

Consequently, a measure that returns the count of 'Different Sent Call Messages' of a 'sender' in a communication diagram qualifies as a measure of the 'numerousness of Different Sent Call Messages'.

Example: The 'Number of Different Sent Call Messages' of c in diagram C of Figure A.4 equals 2:

$$\begin{array}{ll} \delta_{NDSCMsgs}(SDSCMsgs(c), \emptyset) & = \\ |\{A2, A6\} - \emptyset| + & \\ |\emptyset - \{A2, A6\}| & = \\ |\{A2, A6\}| + |\emptyset| & = \\ 2 & \end{array} \tag{A.48}$$

A.7 For a Sender Object: The Number of Different Receiver Objects

- **Step 1: Find a measurement abstraction**

  The set of entities modeled is the 'Universe of senders', actors or objects, $USenders$, that are relevant to the Universe of Discourse UoD.

  The attribute of interest is: *The numerousness of Receiver Objects.*

  Let be $UReceivers$ as the 'Universe of receiver' objects relevant to UoD. All the 'sets of receivers' within the $USenders$ are elements of the 'power set of $UReceivers$', denoted by $\wp(UReceivers)$.

  Therefore, the set of software entities that can be used as measurement abstractions is $\wp(UReceivers)$, and the abstraction function *abs* results in a projection function that maps 'senders', objects or actors, onto their 'sets of receivers':

  $abs_{NReceivers} : USenders \rightarrow \wp(UReceivers) : sender \rightarrow SReceivers(sender)$

  Example: For the communication diagram C in Figure A.4, we have that :

$$\begin{array}{rcl} abs_{NReceivers}(User) & = & SReceivers(User) \\ & = & \{c\} \end{array} \tag{A.49}$$

$$\begin{aligned} abs_{NReceivers}(c) &= SReceivers(c) \\ &= \{m, d\} \end{aligned} \tag{A.50}$$

$$\begin{aligned} abs_{NReceivers}(m) &= SReceivers(m) \\ &= \{s\} \end{aligned} \tag{A.51}$$

$$\begin{aligned} abs_{NReceivers}(s) &= SReceivers(s) \\ &= \emptyset \end{aligned} \tag{A.52}$$

$$\begin{aligned} abs_{NReceivers}(d) &= SReceivers(d) \\ &= \emptyset \end{aligned} \tag{A.53}$$

- **Step 2: Model distances between measurement abstraction**

  Since the elements of $\wp(UReceivers)$ are 'sets of receiver' objects, T_e must contain two elementary transformations: one for adding a 'receiver' to a set and one for removing a 'receiver' from a set. Formally, $T_e = \{t_{0NReceivers}, t_{1NReceivers}\}$, where:

  - $t_{0NReceivers} : \wp(UReceivers) \rightarrow \wp(UReceivers) : s \rightarrow s \cup \{receiver\}$, with $receiver \in UReceivers$
  - $t_{1NReceivers} : \wp(UReceivers) \rightarrow \wp(UReceivers) : s \rightarrow s - \{receiver\}$, with $receiver \in UReceivers$

  Example: The distance between $abs_{NReceivers(c)}$ and $abs_{NReceivers(s)}$ in diagram C of Figure A.4 can be modeled by the following sequence of elementary transformations:

$$\begin{aligned} SReceivers(s) &= \emptyset \\ SReceivers(c) &= \{m, d\} \\ &= t_{0NReceivers}(t_{0NReceivers}(SReceivers(s), m), d) \end{aligned} \tag{A.54}$$

  Therefore, a sequence of 2 elementary transformations are needed to model the distance between $abs_{NReceivers(c)}$ and $abs_{NReceivers(s)}$.

- **Step 3: Quantify distances between measurement abstractions**

  The function $\delta_{NReceivers}$ that models the distances in $\wp(UReceivers)$ is:

  $\delta_{NReceivers} : \wp(UReceivers) \times \wp(UReceivers) \rightarrow \Re : (s, s') \rightarrow (|s - s'| + |s' - s|)$

  Example: Continuing with our previous example, applying the symmetric difference model, the distance between the 'sets of receivers' of s and c equals 2:

$$\begin{aligned} &\delta_{NReceivers}(abs_{NReceivers}(s), abs_{NReceivers}(c)) = \\ &|\emptyset - \{m, d\}| + |\{m, d\} - \emptyset| = \\ &|\emptyset| + |\{m, d\}| = 2 \end{aligned} \tag{A.55}$$

- **Step 4: Find a reference abstraction**

  The obvious reference point for measurement is the 'empty set of receivers', which means that a 'sender' without 'receivers' is the lowest possible value for the $NReceivers$ measure. Therefore, the reference abstraction function can be defined as follows:

  $ref_{NReceivers} : USenders \rightarrow \wp(UReceivers) : sender \rightarrow \emptyset$

- **Step 5. Define the software measure.**

  The 'numerousness of Receivers' of a 'sender' *sender* can be defined as the distance between its 'set of receivers', $SReceivers(sender)$ and the 'empty set of receivers', $\emptyset$. Therefore, its measure 'Number of Receivers' $NReceivers$ can be defined as a function that returns the value of the metric $\delta_{NReceivers}$ for the pair of sets $SReceivers(sender)$ and $\emptyset$:

$$\begin{array}{ll} \forall sender \in USenders : NReceivers(sender) & = \\ \delta_{NReceivers}(SReceivers(sender), \emptyset) & = \\ |SReceivers(sender) - \emptyset| + |\emptyset - SReceivers(sender)| & = \\ |SReceivers(sender)| & \end{array} \tag{A.56}$$

  Consequently, a measure that returns the count of 'receiver objects' of a 'sender' in a 'communication diagram' qualifies as a measure of its 'numerousness of Receivers'.

  Example: The 'Number of Receivers' of c in diagram C of Figure A.4 equals 2:

$$\begin{array}{ll} \delta_{NReceivers}(SReceivers(c), \emptyset) & = \\ |\{m, d\} - \emptyset| + & \\ |\emptyset - \{m, d\}| & = \\ |\{m, d\}| + |\emptyset| + |\{m, d\}| & = \\ 2 & \end{array} \tag{A.57}$$

As we could observe, through all the Chapter, all of the validated measures can be modeled by means of its abstraction function and. Tables A.1 and A.2 summarize the abstract functions of the proposed measures for UML communication diagrams. Furthermore, we also provide the abstract functions for the UML class metrics as defined in Chapter 3. Such functions are listed in Tables A.3 and A.4, which in their turn serve us to approximate the RFC, CBO and WMC CK metrics, as indicated in Table 3.4.

Table A.1: Abstract Functions to Model Direct Metrics for Objects

Measure	***Abstract Function***
NObjects (Number of Objects)	$abs_{NObjects} : UCmD \rightarrow \wp(UObjects) : CmD \rightarrow SObjects(CmD)$ where: • $NObjects$ is the Universe of objects in a communication diagram relevant to the UoD $UCmD$, • $UCmD$ is the Universe of Communication Diagrams, • $SObjects(CmD) \subsetneq UObjects$ is the set of objects of a communication diagram.
NRCMsgs (Number of Received Call Messages)	$abs_{NRCMsgs} : UObjects \rightarrow \wp(URCMsgs) : obj \rightarrow SRCMsgs(obj)$ where : • $URCMsgs$ is the Universe of Received Call Messages relevant to the UoD $UObjects$, • $UObjects$ is the Universe of of Objects within a Communication Diagram, • $SRCMsgs(obj) \subsetneq URCMsgs$ is the set of received call messages of an object, obj.
NDRCMsgs (Number of Different Received Call Messages)	$abs_{NDRCMsgs} : UObjects \rightarrow \wp(UDRCMsgs) : obj \rightarrow SDRCMsgs(obj)$ where: • $UDRCMsgs$ is the Universe of Different Received Call Messages relevant to the UoD $UObjects$, • $UObjects$ is the Universe of Objects within a Communication Diagram • $SDRCMsgs(obj) \subsetneq UDRCMsgs$ is the set of different received call messages of an object, obj.

Table A.2: Abstract Functions to Model Direct Metrics for Objects

Measure	***Abstract Function***
NSenders (Number of Senders)	$abs_{NSenders} : UCmD \rightarrow \wp(USenders) : CmD \rightarrow SSenders(CmD)$ where: • $USenders$ is the Universe of objects or actors acting as senders in a communication diagram relevant to the UoD $UcmD$, • $UCmD$ is the Universe of Communication Diagrams. • $SSenders(CmD) \subsetneq USenders$ is the set of senders of a communication diagram.
NSCMsgs (Number of Sent Call Messages)	$abs_{NSCMsgs} : USenders \rightarrow \wp(USCMsgs) : sndr \rightarrow SSCMsgs(sndr)$ where: • $USCMsgs$ is the Universe of Sent Call Messages relevant to the UoD $USenders$, • $USenders$ is the Universe of objects or actors acting as senders within a communication diagram, • $SSCMsgs(sndr) \subsetneq USCMsgs$ is the set of sent call messages of a sender $sndr$.
NDSCMsgs (Number of Different Sent Call Messages)	$abs_{NDSCMsgs} : USenders \rightarrow \wp(UDSCMsgs) : sndr \rightarrow SDSCMsgs(sndr)$ where: • $UDSCMsgs$ is the Universe of Different Sent Call Messages relevant to an UoD $USenders$, • $USenders$ is the Universe of objects or actors acting as senders within a communication diagram, • $SDSCMsgs(sndr) \subsetneq UDSCMsgs$ is the set of different sent call messages of a sender $sndr$.
NDRO (Number of Different Receiver Objects)	$abs_{NReceivers} : USenders \rightarrow \wp(UReceivers) : sndr \rightarrow SReceivers(sndr)$ where: • $UReceivers$ is the Universe of the different objects acting as receivers for a sender to an UoD $USenders$, • $USenders$ is the Universe of objects or actors acting as senders within a communication diagram, • $SReceiverss(sndr) \subsetneq UReceivers$ is the set of receiver objects of a sender $sndr$.

Table A.3: Abstract Functions to Model Metrics for Classes

Measure	***Abstract Function***
NClasses (Number of Classes)	$abs_{NClasses} : USysDsgs \rightarrow \wp(UObjects) : SysDsg \rightarrow SObjects(SysDsg)$ where: • $NClasses$ is the Universe of objects of a system design relevant to the UoD $USysDsgs$, • $USysDsgs$ is the Universe of UML Designs for different object-oriented software systems, in which for each of theme there exists a set of UML communication diagrams, • $SObjects(SysDsg) \subsetneq UObjects$ is the set of classes in a UML Design, derived from its UML communication diagrams.
NIM (Number of Instantiated Methods)	$abs_{NIM} : UClasses \rightarrow \wp(URCMsgs) : cls \rightarrow SRCMsgs(cls)$ where : • $URCMsgs$ is the Universe of Received Call Messages relevant to the UoD $UClasses$, • $UClasses$ is the Universe of classes within a UML Design of an object-oriented software system, • $SRCMsgs(cls) \subsetneq URCMsgs$ is the set of received call messages by all objects of the class cls. A call message can be distinguished from another, if its sequence expression of its label are different, e.g. the call message $A1 : print()$ is different to the call message $A2 : print()$.
NDIM (Number of Different Instantiated Methods)	$abs_{NDIM} : UClasses \rightarrow \wp(UDRCMsgs) : cls \rightarrow SDRCMsgs(cls)$ where: • $UDRCMsgs$ is the Universe of Different Received Call Messages relevant to the UoD $UClasses$, • $UClasses$ is the Universe of classes within a UML Design of an object-oriented software system, • $SDRCMsgs(cls) \subsetneq UDRCMsgs$ is the set of different received call messages of all objects of the class $class$. A call message can be distinguished from another, if their names are different, e.g. the call message $A1 : print()$ is exactly the same as the call message $A2 : print()$.

Table A.4: Abstract Functions to Model Metrics for Classes

Measure	***Abstract Function***
NIMO (Number of Instantiated Methods of Other Classes)	$abs_{NIMO} : UClasses \rightarrow \wp(USCMsgs) : cls \rightarrow SSCMsgs(cls)$ where: • $USCMsgs$ is the Universe of Sent Call Messages relevant to the UoD $UClasses$, • $UClasses$ is the Universe of classes within a UML Design of an object-oriented software system, • $SSCMsgs(cls) \subsetneq USCMsgs$ is the set of sent call messages of all objects of a class cls. A call message can be distinguished from another, if its sequence expression of its label are different, e.g. the call message $A1 : print()$ is different to the call message $A2 : print()$.
NDIMO (Number of Different Instantiated Methods of Other Classes)	$abs_{NDIMO} : UClasses \rightarrow \wp(UDSCMsgs) : cls \rightarrow SDSCMsgs(cls)$ where: • $UDSCMsgs$ is the Universe of Different Sent Call Messages relevant to an UoD $UClasses$, • $UClasses$ is the Universe of classes within a UML Design of an object-oriented software system, • $SDSCMsgs(cls) \subsetneq UDSCMsgs$ is the set of different call messages sent by the objects of the class cls. A call message can be distinguished from another, if their names are different, e.g. the call message $A1 : print()$ is exactly the same as the call message $A2 : print()$.
NDRC (Number of Different Receiver Classes)	$abs_{NDRC} : UClasses \rightarrow \wp(UReceivers) : cls \rightarrow SReceivers(cls)$ where: • $UReceivers$ is the Universe of the different objects acting as receivers of a message sent by any of the objects of the class cls to an UoD $UClass$, • $UClasses$ is the Universe of classes within a UML Design of an object-oriented software system, • $SReceiverss(cls) \subsetneq UReceivers$ is the set of objects that receive a call message from any of the objects of the class cls.