Constructing Modifiability Metrics by Considering Different Relationships
Matinee Kiewkanya* and Pornsiri Muenchaisri
Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, 10330, Thailand.
*Author for correspondence; e-mail: g4122109@yahoo.com

ABSTRACT

Class diagrams provide mainly graphical notations to represent design artifacts used as blueprints for object-oriented software development. The result gained from measuring modifiability of class diagrams in the early phase can be used as a prime indicator whether to redesign software in order to reduce cost of software maintenance in a later phase. This paper proposes a new metric set for evaluating modifiability of object-oriented designs. These metrics can be measured from class diagrams. Different kinds of relationship influence the dependency between classes to a different degree. In this work modifiability metrics are constructed by considering relationships among classes. An experiment carried out in order to validate the ability of the proposed metrics is also presented. The experimental result shows that the new proposed metrics significantly correlate with human beings’ view of modifiability.

Keywords: modifiability, class diagram, software metric.

1. INTRODUCTION

The Unified Modeling Language (UML) [1] has widely been accepted as an industrial standard for modeling object-oriented designs. It provides a range of OO diagrammatic notations for expressing the structural and behavioral aspects of software. Class diagrams, the most important structural model and indeed the central model of the UML, show static aspects in terms of the classes of objects in the software, relationships among these classes, and constraints in the relationships. A quality of the class diagrams can have a significant impact on the quality of the software which is ultimately implemented. Software requirements are often changed. This requires software to be modified several times after its initial development. Maintainability is the attribute that represents how easy/hard and time consuming the process of maintaining the software is.

For the software to be maintainable, we want it to be easy to understand, to enhance, or to correct. Many researchers reported that 50-70% of software production cost was directly related to software maintenance [2-4].

Several software quality models have been defined. Boehm’s model, McCall’s
model, and ISO 9126 are the models of software quality that have gained the acceptance of the software engineering community. These models introduce maintainability sub-characteristics as follows [5]:

- Boehm’s model – This model proposes a multi-level hierarchy of software criteria. The model defines testability, understandability and modifiability as criteria which are then used to evaluate maintainability of software.

- McCall’s model – McCall’s model has described quality using a decompositional approach. The approach called Factor, Criteria, and Metric is based on product operation, product revision, and product transition. The model measures maintainability based on a combination of quality criteria such as consistency, simplicity, conciseness, self-descriptiveness and modularity.

- ISO 9126 – The International Standards Organization is a derivation of McCall’s model. The standard claims that the quality is composed of 6 factors: functionality, reliability, efficiency, usability, maintainability and portability. These can be further broken down into sub-characteristics that have measurable attributes. Four quality criteria are recommended for measuring maintainability. These criteria are analyzability, changeability, stability, and testability.

Successful maintenance of software requires two things: in-depth understanding in software’s structure and behavior, and ability

![Activity diagram of the research.](image-url)
to make changes easily. We can measure maintainability by considering its sub-characteristics. According to Boehm’s model [5], this work focuses on measuring modifiability, a dominant sub-characteristic of maintainability, from class diagrams.

Modifiability is defined as the ease with which software can be changed. Assessing modifiability at the design level will help software designers to decide if the design of software should be altered, which leads to the ease of maintenance.

The structural complexity measure is one of the most important measures to evaluate the quality of class diagrams. Sheldon et al. [6] suggested new metrics for modifiability of a class inheritance hierarchy which can be measured from class diagrams. They extended work of Li and work of Chidamber and Kemerer who proposed six well known object-oriented metrics (CK’s metrics). They also compared their metrics with two other metrics named DIT proposed by Chidamber and Kemerer and AID proposed by Henderson and Sellers in order to validate the new metrics. However, in [6] the authors focus on only generalization relationship. In this work, we extend their work by considering the important relationships represented in class diagrams including dependency, association, aggregation, composition, generalization and realization. An overview of this research can be shown as figure 1. The detail of each activity will be described in later sections. This paper has two main objectives:

1. To propose a new metric set for measuring modifiability from class diagrams.
2. To present an experiment carried out in order to validate the proposed metrics.

The remainder of this paper is organized into 7 sections as follows. The next section presents related work. Section 3 proposes definitions and derivations of the new metrics for modifiability. Section 4 presents an experiment carried out in order to validate the proposed metrics. Section 5 discusses various issues that threaten the validity of the experiment and the attempt to alleviate them. Section 6 presents conclusion and discussion. Future work is given in the last section.

2. RELATED WORK

The UML class diagram is the most important structural model and indeed the central model of the UML. Various metrics for class diagrams have been suggested, which helps software developers to analyze reliability, maintainability, and complexity of software in the early phase of the object-oriented software lifecycle. The structural complexity measure is one of the most important measures to evaluate the quality of class diagrams. The experimental results of Kiewkanya et al. [7] showed that structural complexity metrics can be good indicators of maintainability. Zhou [8] and Kang et al. [9] proposed measuring the structural complexity of class diagrams based on an entropy distance. This method can measure the structural complexity of class diagrams objectively. In essence, the Zhou’s and Kang’s metrics are similar. The proposed metrics consider the number of relationships among classes, the interaction pattern of classes, and the kinds of relationship. Yi et al. [10] presented an empirical analysis of the entropy distance metric for class diagrams, specifically Zhou’s metric. This work explored a correlation between the entropy distance metric and the three sub-characteristics of maintainability: understandability, analyzability and modifiability measured from human rating. The experimental result indicated that the metric was basically consistent with human beings’ intuition.

Marchesi [11] proposed a set of indicators to measure the complexity of class diagrams. Among them, the total number of
inheritance hierarchies, the average number of direct dependencies of classes, and the standard deviation of the number of direct dependencies are used to measure relationships among classes. However, he did not take into account some UML measurable elements, such as associations, aggregations and dependencies. To overcome the limitations of Marchesi’s method, Genero et al. [12] proposed new metrics for class diagrams. They classified these metrics into two categories: open-ended metrics and close-ended metrics. They used the maximum longest path from a class to its root of the hierarchy, the respective total number of association relationships, aggregation relationships, dependency relationships, generalization relationships, generalization hierarchies, and aggregation hierarchies to measure relationships among classes. They carried out controlled experiments with the goal of predicting class diagram maintainability and its sub-characteristics, namely understandability and modifiability, from the proposed metrics using various techniques such as Fuzzy classification, Kolmogrov-Smirnov test, Principal component analysis and Regression model [13-15]. Kim and Boldreff proposed metrics that can be applied to class diagrams such as the number of the associations linked to a class, the number of the superclasses of a class, the number of the elements in the transitive closure of the superclasses of a class, the number of the subclasses of a class, and the number of the elements in the transitive closure of the subclasses of a class, the coupling between classes, the depth of inheritance tree [16]. Zhou applied PageRank and HITS algorithms to measure the importance of classes [17]. Yi proposed metrics for measuring complexity of relationships among classes [18]. In his work, he proposed classes as web pages and relationships among classes as links among web pages. He inferred complexity of relationships according to the PageRank algorithm.

Many researchers simply count, average, or maximize relationships. They did not differentiate different relationships. Different relationships may have a different effect on classes. Therefore, in this work, we construct modifiability metrics by considering different relationships among classes.

3. NEW METRICS FOR MODIFIABILITY

Programming is often called an art. In this sense, programming may be viewed as a technique completed with heuristics such as those with art [6]. It follows from this comparison that software metrics should consider heuristic properties. Accordingly, it is not reasonable to measure the software product and process, which is actually a labor-intensive industry, only by mathematical and logical metrics without considering the human aspects. Therefore, we shall consider the measurement of modifiability in a heuristic way.

3.1 Metrics Definition

Sheldon et al. proposed metrics for modifiability of class inheritance hierarchies in [6]. Their work focused on modifiability related to inheritance only. We adapt their idea and extend their work by considering modifiability related to other relationships. This section begins with the definitions of our proposed metrics. Then, the issue of how to compute these metrics is described in section 3.2.

Metrics for modifiability consist of 8 metrics for a class and 1 metric for software. All metrics are defined as follows.

**Metrics for modifiability of a class**

- \( \text{Mod}_{\text{Gen}}(X) \) is the degree of
modifiability of class X related to generalization.
- Mod_{Agg}(X) is the degree of modifiability of class X related to aggregation.
- Mod_{Com}(X) is the degree of modifiability of class X related to composition.
- Mod_{CAssoc}(X) is the degree of modifiability of class X related to common association.
- Mod_{AssocC}(X) is the degree of modifiability of class X related to association class.
- Mod_{Dep}(X) is the degree of modifiability of class X related to dependency.
- Mod_{Real}(X) is the degree of modifiability of class X related to realization.
- Mod_{Class}(X) is the degree of modifiability of class X.

Metric for modifiability of software
- AvgMod_{Sys}(S) is the average degree of modifiability of software S.

In order to compute metrics introduced above, terms and functions used to compute them are defined as follows.

Head and tail classes
For each relationship, let an arrowhead of a relationship line indicates a position of each class. If a relationship line points from class A to class B, then class B is called a head class of class A and class A is called a tail class of class B.

Consider a hierarchy of an i relationship.
Let i be a relationship type.
Gen(Genlialization), CAssoc(Common association), AssocC(Association class), Agg(Aggregation), Com(Composition), Dep(Dependency) or Real(Realization), related to class X. All head and tail classes of class X are defined as follows.
- ImHead_{i}(X) is the immediate head classes of class X related to class X by an i relationship.
- AllHead_{i}(X) is all head classes of class X related to class X by an i relationship.
- AllTail_{i}(X) is all tail classes of class X related to class X by an i relationship.

For better understanding, see examples of the usage of these metrics as shown in Figure 2 and Figure 3.

For Figure 2,
ImHead_{Gen}(X) = B,C.
ImTail_{Gen}(X) = D,E.
AllHead_{Gen}(X) = A,B,C and
AllTail_{Gen}(X) = D,E,F,G.

For Figure 3,
ImTail_{Agg}(AllHead_{Gen}(X)) = ImTail_{Agg}(A,B,C)
= ImTail_{Agg}(A) + ImTail_{Agg}(B) + ImTail_{Agg}(C)
= { } + D,E + F,G
= D,E,F,G

Class complexity
C(X) is the complexity of class X. It is simply defined as follows.
C(X) = number of methods of class X + number of attributes of class X.

In case of many classes in parenthesis, the value of the complexity is the summation of the complexity of all classes. For example
C(X,Y,Z) = C(X) + C(Y) + C(Z).

Figure 2. A hierarchy of generalization related to class X.
Figure 3. A hierarchy of generalization and aggregation related to class X.

Dependency weight values of relationships

$W_i$ is a dependency weight value of an $i$th relationship. The dependency between classes is the main cause of the amount of the complexity on modifying relationships between classes. Different kinds of relationships influence the dependency between classes to a different degree. So, the dependency weight value of each relationship should be defined in order to indicate its dependency degree. In this work, we consider 7 kinds of relationships consisting of dependency, common association, association class, aggregation, composition, generalization, and realization. Table 1 shows the dependency weight value of relationships proposed by Kang et al.[9]. Relationships shown in this table are sorted from weak to strong dependency degrees.

Dependency is the most common relationship. A dependency shows that there is dependency between two classes without any explanations and restrictions. So $W_{Dep}$ should be the minimum. Common association denotes the relationship between instances of classes and it cannot be weaker than dependency relationship. Association class adds restrictions to association. It may be more complex and the dependency between classes with this relationship may be stronger than between classes with common association. Aggregation is a specific association, and composition is a specific aggregation. For example, A is a composite of B; when A is destroyed, B should be destroyed or given to another object. Aggregation does not have this restriction, but it is more restrict than dependency relationship, as one object cannot aggregate itself directly or indirectly. In generalization, subclasses inherit all non-private characteristics of the parent classes, and composition classes can only access the public elements of the nested classes. When parent classes are concrete, subclasses can add new elements and override inherited operations. When parent classes are abstract, subclasses should implement the virtual operations of the parent classes or they cannot have any instances. Considering realization, when realizing a class (usually interface), an implementation class must realize all the operations of the interface. So realization has the highest weight value.

Table 1. Dependency weight value of relationships.

<table>
<thead>
<tr>
<th>No.</th>
<th>Relationship</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dependency</td>
<td>$W_{Dep}$</td>
</tr>
<tr>
<td>2</td>
<td>Common assoc</td>
<td>$W_{CAssoc}$</td>
</tr>
<tr>
<td>3</td>
<td>Assoc class</td>
<td>$W_{AssocC}$</td>
</tr>
<tr>
<td>4</td>
<td>Aggregation</td>
<td>$W_{Agg}$</td>
</tr>
<tr>
<td>5</td>
<td>Composition</td>
<td>$W_{Com}$</td>
</tr>
<tr>
<td>6</td>
<td>Generalization</td>
<td>$W_{Gen}$</td>
</tr>
<tr>
<td>7</td>
<td>Realization</td>
<td>$W_{Real}$</td>
</tr>
</tbody>
</table>

In [2], all weight values are summarized in the following form:

$$W_{Dep} <= W_{CAssoc} <= W_{AssocC} <= W_{Com} \quad (1)$$

$$W_{Dep} < W_{Agg} < W_{Com} < W_{Gen} < W_{Real} \quad (2)$$

Following [9], we give each relationship a weight value satisfying inequation (1) and
3.2 Metrics Derivation

This section describes derivation of all metrics defined in the previous section.

3.2.1 Modifiability of a class

Metrics for modifiability of a class are proposed with the idea that if we want to modify class X, class X will be modified. Moreover, if class X affects other classes, these classes will be modified too. In the best case, only class X will need to be modified. In the worst case, class X and all classes affected from class X must be modified. In the average case, class X will be modified and half of classes affected from class X should be modified. In this work, modifiability will be measured in the average case.

In order to measure modifiability of class X, we will consider the efforts put in for modifying class X and modifying classes related to modifying class X.

In this work, the effort put in for modifying class X is represented by the complexity of class X or C(X) because if class X has a high complexity we should put much effort to modify it. The effort used in modifying classes related to class X will be represented by 7 metrics: ModGen, ModAgg, ModCom, ModAssoc, ModAssocC and ModDep explained above. They are calculated according to the kind of relationships that class X is related to other classes.

The degree of modifiability of class X is the summation of the efforts used in modifying class X and modifying classes related to class X as follows.

\[
\text{Mod}_{\text{Class}}(X) = C(X) + \text{Mod}_{\text{Gen}}(X) + \text{Mod}_{\text{Agg}}(X) + \text{Mod}_{\text{Com}}(X) + \text{Mod}_{\text{Assoc}}(X) + \text{Mod}_{\text{AssocC}}(X) + \text{Mod}_{\text{Dep}}(X) + \text{Mod}_{\text{Real}}(X)
\]

3.2.2 Modifiability related to generalization

Consider classes related to class X by generalization. If we modify class X, we may need to modify descendant classes of class X. These classes can be represented by AllTail_{gen}(X) as an example shown in Figure 4. In the average case, half of them will be modified. Mod_{gen} can be defined in expression of the multiplication between the dependency weight value of generalization and the complexity of classes related to generalization considering in the average case as follows.

\[
\text{Mod}_{\text{Gen}}(X) = W_{\text{Gen}} C(\text{AllTail}_{\text{gen}}(X)) / 2
\]

![Figure 4. Generalization.](image_url)

3.2.3 Modifiability related to aggregation

Consider classes related to class X by aggregation. If we modify class X, classes which compose of class X may be modified. These classes can be represented by ImHead_{agg}(X) as an example shown in the dotted oval area of Figure 5. Classes in the
dotted rectangle area of Figure 5 are classes related to class X by an indirect aggregation. If class X is changed, they may be affected. These classes can be represented by \( \text{AllTail}_\text{Gen} (\text{ImHead}_\text{Agg}(X)) \). Considering in the average case, \( \text{Mod}_\text{Agg}(X) \) is defined as follows.

\[
\text{Mod}_\text{Agg}(X) = \frac{W_{\text{Agg}} [C(\text{ImHead}_\text{Agg}(X)) + C(\text{AllTail}_\text{Gen}(\text{ImHead}_\text{Agg}(X)))]}{2}
\]

**Figure 5.** Aggregation.

**Modifiability related to composition**

Classes affected from class X by composition can be considered similarly to classes affected from class X by aggregation. Considering in the average case, \( \text{Mod}_\text{Com}(X) \) is defined as follows.

\[
\text{Mod}_\text{Com}(X) = \frac{W_{\text{Com}} [C(\text{ImHead}_\text{Com}(X)) + C(\text{AllTail}_\text{Gen}(\text{ImHead}_\text{Com}(X)))]}{2}
\]

**Modifiability related to common association**

Consider classes related to class X by an unidirectional common association. If we modify class X, classes which directly invoke methods of class X may be affected. These classes can be represented by \( \text{ImTail}_\text{CAssoc}(X) \) as an example shown in the dotted oval area of Figure 6. Furthermore, classes related to class X by indirect common association may be affected as an example shown in the dotted rectangle area of Figure 6. These classes can be represented by \( \text{AllTail}_\text{Gen}(\text{ImTail}_\text{CAssoc}(X)) \). Considering in the average case, \( \text{Mod}_\text{CAssoc}(X) \) is defined as follows.

\[
\text{Mod}_\text{CAssoc}(X) = \frac{W_{\text{CAssoc}} [C(\text{ImTail}_\text{CAssoc}(X)) + C(\text{AllTail}_\text{Gen}(\text{ImTail}_\text{CAssoc}(X)))]}{2}
\]

**Figure 6.** Common association.

For a bidirectional common association, we will transform the relationship to an unidirectional common association as an example shown in Figure 7. After that modifiability can be measured by using the same formula as used for the unidirectional common association.

**Figure 7.** Transforming bidirectional association to unidirectional association.

**Modifiability related to association class**

For classes related to class X by an association class, we will transform the association class to an unidirectional association form. Then we will consider each pair of relationships. For examples, in order to measure modifiability related to direct and indirect association class of class X, Figure 8 will be transformed to Figure 9, and Figure 10 will be transformed to Figure 11. Then \( \text{Mod}_\text{AssocC} \) will be measured from Consider boxes of Figures 9 and 11. For example in Consider box of Figure 9, if we want to
modify class $X$, we may also modify class $A$ and $B$. Mod$_{\text{Assoc}}(X)$ is defined in a similar way as defining Mod$_{\text{CAssoc}}$ as follows.

$$\text{Mod}_{\text{Assoc}}(X) = \frac{W_{\text{Assoc}} \left[ C(\text{ImTail}_{\text{Assoc}}(X)) + C(\text{AllTail}_{\text{Gen}}(\text{ImTail}_{\text{Assoc}}(X))) \right]}{2}$$

**Modifiability related to dependency**

Consider classes related to class $X$ by dependency. If we modify class $X$, classes depending on class $X$ may be modified. These classes can be represented by $\text{ImTail}_{\text{Dep}}(X)$ as an example shown in the dotted oval area of Figure 12. Furthermore, classes related to class $X$ by an indirect dependency may be modified as an example shown in the dotted rectangle area of Figure 12. These classes can be represented by $\text{AllTail}_{\text{Gen}}(\text{ImTail}_{\text{Dep}}(X))$. Consider the average case. Mod$_{\text{Dep}}(X)$ is defined as follows.

$$\text{Mod}_{\text{Dep}}(X) = \frac{W_{\text{Dep}} \left[ C(\text{ImTail}_{\text{Dep}}(X)) + C(\text{AllTail}_{\text{Gen}}(\text{ImTail}_{\text{Dep}}(X))) \right]}{2}$$

**Modifiability related to realization**

Consider classes related to class $X$ by realization. If we modify class $X$, implementation classes of class $X$ must be modified. These classes can be represented by $\text{ImTail}_{\text{Real}}(X)$ as an example shown in the dotted oval area of Figure 13. Furthermore, classes related to class $X$ by an indirect realization may be modified as an example shown in the dotted rectangle area of Figure 13. These classes can be represented by $\text{AllTail}_{\text{Gen}}(\text{ImTail}_{\text{Real}}(X))$. Consider the average case. Mod$_{\text{Real}}(X)$ is defined as follows.

$$\text{Mod}_{\text{Real}}(X) = \frac{W_{\text{Real}} \left[ C(\text{ImTail}_{\text{Real}}(X)) + C(\text{AllTail}_{\text{Gen}}(\text{ImTail}_{\text{Real}}(X))) \right]}{2}$$

**Figure 8.** Direct association class.

**Figure 9.** Transformed direct association class.

**Figure 10.** Indirect association class.

**Figure 11.** Transformed indirect association class.

**Figure 12.** Dependency.
3.2.2 Modifiability of Software

Modifiability of software will be calculated from the modifiability of all classes in the software. Generally, large software is more complex than smaller one. Accordingly, it is more reasonable to compare modifiability of one piece of software to modifiability of another piece of software with the same size. For comparing modifiability of software with different sizes, we should introduce the concept of average. We have defined the average degree of software’s modifiability as follows.

\[ \text{AvgModSys}(S) = \left( \sum_{i=1}^{n} \frac{\text{Mod}_{\text{class}}(X_i)}{n} \right) \]

where \( X_i \) is a class of software \( S \); \( i = 1,2,\ldots,n \).

\( n \) is the total number of classes of software \( S \).

4. EMPIRICAL VALIDATION

This section describes the experiment carried out to validate the proposed metrics.

4.1 Participants

The participants of this experiment were 60 graduate students from the Department of Computer Engineering at Chulalongkorn University, Bangkok, Thailand, who passed classes on Software Requirements Engineering and Object-Oriented Technology. During lectures, students were taught basic software engineering principles and object-oriented development techniques. The lectures were supplemented by practical lessons which the students had the opportunity to design real-world object-oriented software using UML diagrams.

The information captured from the debriefing questionnaire based on the ordinal scale of 1 to 5 revealed that the subjects had medium experience with

- software engineering practice – median response 3 (min 2, max 4),
- design documents – median response 3 (min 2, max 4),
- modeling with UML – median response 3 (min 2, max 4) and
- software maintenance – median response 3 (min 2, max 4).

<table>
<thead>
<tr>
<th>Software requirements engineering</th>
<th>Object-Oriented Technology</th>
<th>Subject Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>B+</td>
<td>A</td>
</tr>
<tr>
<td>B+</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B+</td>
<td>B+</td>
<td>B+</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>B+</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B+</td>
</tr>
<tr>
<td>B+</td>
<td>B+</td>
<td>B+</td>
</tr>
<tr>
<td>B</td>
<td>B+</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B+</td>
</tr>
</tbody>
</table>

In order to control differences among students, the students were categorized into types A, B+ and B by considering the grades they obtained from 2 classes mentioned above. Table 2 shows how to categorize the students. For example, if a student got an A’s in Software Requirements Engineering and got a B+’s in Object-Oriented Technology, that student would be categorized into type A. After that, they were randomly grouped into 20 teams of three students. Each team had one student with type A, one student with type B+ and one student with type B.
4.2 Experimental Materials and Tasks
- The material used in this experiment consisted of 40 class diagrams of different application domains. Each class diagram had a test that included a brief description of what the diagram represented, and examination for assessing modifiability.

- Examination for assessing modifiability contained 10 questions. Subjects were asked to modify class diagrams with tasks covering on changing and adding software functionality. So, the elements in class diagrams including attributes, methods, classes, and relationships can be changed.

- Each subject team was asked to complete the tests of 2 class diagrams that were randomly assigned to the team. Each subject had to carry out the tests alone.

- The subjects were given the materials described and 2 monitors who explained to them how to carry out the tests. Subjects were told verbally that the designs they were working upon were different, but were not told anything about the nature of the study, i.e., what hypotheses were being tested. During this time subjects were told not to talk among themselves, but to direct any questions they had to the two monitors.

- The time-out period for each modifiability examination was 40 minutes. There was a 15-minute break between experimental tasks of 2 class diagrams. This time-out period was determined from a pilot test.

- A pilot test was performed using four experienced subjects. The pilot test was conducted in order to find mistakes in the experimental procedure, to test that the experimental instructions were clear and to check tasks had reasonable complexity, but that they could be completed within the allotted time. No significant issues were encountered during the pilot test.

- The second task was to complete a debriefing questionnaire. This questionnaire captured personal information, experience, motivation, and subjective opinion of each subject.

4.3 Data Collection
In many studies [13, 19, 20], modifiability had been operationalized in terms of modifiability time (time required to make changes). In this work, time for performing experimental tasks was restricted. So, modifiability was considered in terms of accuracy instead. For each class diagram, we collected a modifiability score which is defined as the average of 3 subjects’ scores of the examination for assessing modifiability.

4.4 Data Analysis and Result
In section 3 we introduced metrics for modifiability, which still needs empirical validation in order to validate their usability. Generally, a metric will be invalid in practice but valid in theoretical argument, and vice versa. Whether a metric is valid depends on whether it is consistent with human beings’ intuition or not. In section 4.3, we collected the degree of modifiability in terms of Modifiability score. In order to validate our proposed metrics, we measured the degree of modifiability of the same 40 class diagrams using the new proposed metric, \( \text{AvgUnd}_{sys} \).

To find out the correlation between the degree of modifiability measured by the proposed metrics and human beings’ intuition, the following hypothesis was formulated as follows:

\[ H_0 : \text{There is no correlation between } \text{AvgMod}_{sys}, \text{ and Modifiability score}. \]

\[ H_1 : \text{There is correlation between } \text{AvgMod}_{sys} \text{ and Modifiability score}. \]

In order to test the hypothesis, we applied the Pearson's correlation test. The correlation between two variables reflects the degree to
which the variables are related. Correlation value ranges from -1 to +1. Value ‘1’ means that there is a perfect positive relationship between both variables. Value ‘-1’ indicates a perfect negative relationship and value ‘0’ indicates no relationship. The result of Pearson’s correlation test at level 0.01 showed that the correlation between AvgModSys and Modifiability score is -0.68. Therefore, it can be concluded that H1 is accepted. There is a correlation between AvgModSys and Modifiability score. The correlation value of -0.68 indicates that AvgModSys and Modifiability score have a negative correlation. AvgModSys is measured from software structural complexity represented in class diagrams while the Modifiability score is measured from examination. If software is easy to modify, AvgModSys will be low but Modifiability score will be high. In contrast, if software is difficult to modify, AvgModSys will be high but Modifiability score will be low.

4.5 Metric Threshold

In order to find a threshold for AvgModSys, we classified modifiability into 3 levels: difficult, medium and easy. After that, we calculated value ranges of AvgModSys that lie in each modifiability level.

For each class diagram, we captured Modifiability level by converting Modifiability score into difficult, medium or easy levels by using the following condition:

If Modifiability score < The average value of Modifiability scores – (W * The standard deviation value of Modifiability scores)

Then Modifiability level = “difficult”

Else If Modifiability score > The average value of Modifiability scores + (W * Standard deviation value of Modifiability scores)

Then Modifiability level = “Easy”
Else Modifiability level = “Medium”

Kolmogorov-Smirnov is a statistical technique used to decide if a sample comes from a population with a specific distribution [21]. The results of Kolmogorov-Smirnov test on Modifiability scores showed that Modifiability scores had normal distribution. Therefore, this approach could be considered valid. The approach can be summarized as shown in Figure 14.

W is a constant number. Its value is adjusted according to data distribution. In this experiment, value of W is 0.5.

To estimate a threshold or a value range of AvgModSys that lies in 3 modifiability levels: easy, medium and difficult, we calculate a lower confidence limit (L) and an upper confidence limit (U) values of mean (μ) of population of AvgModSys for each modifiability level. The range of mean of population can be expressed as the follows [21].

\[
L < \mu < U
\]

where L and U can be computed from the following formula.

\[
\bar{x} - Z_{\alpha/2} \frac{\sigma}{\sqrt{n}} < \mu < \bar{x} + Z_{\alpha/2} \frac{\sigma}{\sqrt{n}}
\]

where \( \bar{x} \) is the mean of samples, \( Z_{\alpha/2} \) is the probability value in Z table, and \( \frac{\sigma}{\sqrt{n}} \) is the standard deviation.
At significant level (\(\sigma\)) 0.05, we computed lower and upper confidence limit values: \(L_2\) and \(U_2\), \(L_1\) and \(U_1\), and \(L_0\) and \(U_0\), for easy, medium and difficult levels of modifiability respectively as shown in Figure 15.

From this figure, we can imply that if a class diagram has \(\text{AvgModSys} < U_2\), it should be classified to be in the easy level of modifiability. If a class diagram has \(\text{AvgModSys} > L_0\), it should be classified to be in the difficult level of modifiability. The preliminary result obtained from our experimental data is shown in Table 3.

5. Threats to Validity

Following several empirical studies [12, 19, 22], this section discusses various issues that threaten the validity of the experiment and the attempts to alleviate them.

5.1 Threats to internal validity

Threats to internal validity are influences that can affect the independent variables with respect to causality. Thus they threat the conclusion about a possible causal relationship between treatment and outcome. The following possible treats are identified.

- Differences among subjects. Each class diagram was evaluated by groups of 3 subjects. Although the ability of modifying class diagram was not exactly equivalent among groups. Differences among groups were reduced by assigning one student with type A, one student with type B+, and one student with type B to each group.

- Knowledge of the universe of discourse among class diagrams. The class diagrams were designed from different universes of discourse, but they were
common enough to be easily understood by the subjects. So, knowledge of the domain did not affect internal validity.

- **Accuracy of subject responses.** The information captured from the debriefing questionnaire revealed that subjects had medium experience in modeling the software design with UML. So, their responses to examinations could be considered valid.

- **Effects on learning.** Each subject performed an experimental task of only 2 class diagrams. The effect on learning was little relevant.

- **Fatigue effects.** The fatigue had a little effect because each subject performed an experimental task of 2 class diagrams with a 15-minute break between them.

- **Persistence effects.** In order to avoid persistent effects, the experiment was carried out by subjects who had never performed a similar experiment.

- **Subject motivations.** All subject participated this experiment voluntarily. The responses of debriefing questionnaires indicated that 93% of subjects paid a heavy attention in performing examinations.

- **Other factors.** Plagiarism and influence between subjects were controlled. Two subjects who sat adjacently performed different tests. Subjects were told that talking to each other was forbidden. The experiment was controlled by 2 monitors.

5.2 Threats to External Validity

Regarding to the external validity, i.e. the degree to which the results of the research can be generalized to the population under study and other research settings. The greater the external validity, the more results of an empirical study can be generalized to actual software practices. The following possible threats have been identified.

- **Materials and tasks used.** Examination questions tried to capture modifiability of class diagrams. All questions were approved by experts. We tried to use class diagrams which can be representative of real cases, but the class diagrams used were small and simple. The class diagram which had a maximum number of classes contained only 25 classes. This is a limitation of the study since it is not easy to find class diagrams of real world software.

- **Experimental Subjects.** To solve the difficulty of obtaining professional subjects, we used students as experimental subjects. We are aware that it is necessary to replicate this experiment with practitioners and professionals in order to be able to generalize these results. However, this experiment did not require a high level of industrial experience. Students could be usually accepted as valid subjects [12, 19, 22].

6. CONCLUSION AND DISCUSSION

This paper proposes a new metric set to assess modifiability of object-oriented designs. These metrics are developed by considering the number and the kind of relationships among classes. To validate the new proposed metrics, a correlation analysis between modifiability measured by the proposed metrics and human beings’ intuition was performed. From our experiment, we can conclude that the new proposed metric is significantly correlated with human beings’ intuition at significant level 0.01. The result shows that AvgModSys and Modifiability score have a negative correlation.

Modifiability captured by our new proposed metric and by human beings’ intuition has a negative correlation because of the following reason. AvgModSys measures the degree of modifiability of class diagrams by considering the efforts put for modifying the class diagram. Modifiability score captures the degree of modifiability of class diagrams by using examination. If the class diagram is easy
to modify, $AvgMod_{Sys}$ will be low but Modifiability score will be high. In contrast, if the class diagram is difficult to modify, $AvgMod_{Sys}$ will be high but Modifiability scores will be low.

Inheritance increases reuse and improves similarity of implementation. On the other hand, it also increases the complexity of software and the coupling between classes leading to increasing of the effort put in for maintenance. In this work, we classify relationships between 2 classes into 2 kinds: direct and indirect relationships where indirect relationship is relationship through inheritance. Using our proposed metric, if software has many numbers of inheritance relationships among classes the values of $AvgMod_{Sys}$ will be high. This value indicates that the effort put in for modifying the software will be high. This result is supported in some studies. An experimental investigation found that making changes to a C++ program with inheritance consumed more effort than a program without inheritance [23]. Another controlled experiment was conducted to establish the effects of varying levels of inheritance on understandability and modifiability [24]. The result of the experiment indicated that the software without inheritance was easier to modify than the corresponding software containing three or five levels of inheritance. It was also easier to understand the software without inheritance than a corresponding version containing three levels of inheritance.

It is generally accepted that the quality of the software is highly dependent on decisions made early in the development. It is necessary to have measurement support for class diagrams, the most important blueprints of the software, produced in the early phase of software life cycle. The new proposed metrics could be used as early modifiability indicators. Evaluating modifiability at the design phase using these metrics will help software designers to alter the design of the software for better performance that leads to a reduced maintenance cost.

7. FUTURE WORK

In the future, we plan on conducting further studies as listed below.

- $AvgMod_{Sys}$ is developed in order to measure the modifiability of class diagrams during the design phase of the software life cycle. One function used for computing $AvgMod_{Sys}$ is the complexity of a class. The metric computes the complexity of a class roughly from the summation of the number of attributes and the number of methods in the class. This metric can be extended to enable more understanding of the complexity of a class. For example, visibility of attributes and methods (i.e. public, protected and private) should be considered.

- As mentioned in section 3.1, we can specify a dependency degree of each relationship but we do not know exact weight values. Ideally, these weight values should be assigned by experts or captured from empirical study with enough supported data. In this work, we give each relationship a weight value satisfying unequation (1) and (2) introduced in section 3.1: $W_{Dep} = 1$, $W_{Gen} = 2$, $W_{assoc} = 3$, $W_{Agg} = 4$, $W_{Com} = 5$, $W_{Gen} = 6$, $W_{Real} = 7$. Different weight values should be used in the future experiment to find another proper dependency weight value of each relationship.

- The effect of each relationship on modifiability should be explored.

- The experiment uses only forty class diagrams. This is a limitation of the research, since it is difficult to find class diagrams in real world software. The experiment should be repeated with a higher number of sample class diagrams in order to increase the reliability of the experimental result.

- The size of class diagrams should be
increased. By increasing the size of class diagrams, we will have examples that are closer to reality. In addition, if we are working with professionals, we can make a better use of their potential capability and conclude that the results are more general.

An automated tool for measuring the proposed metrics should be constructed.

REFERENCES


