Master Thesis

Title

A Method for Measuring Modifications of Class Diagrams Based on Version Change Information

Supervisor
Prof. Shinji KUSUMOTO

by
Shinya YAMADA

February 7, 2011

Department of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

A Method for Measuring Modifications of Class Diagrams Based on Version Change Information

Shinya YAMADA

Abstract

In order to effectively manage the development of a software, managers need to precisely understand the progress of the development and measure against the delay of them. In the lower processes of software developments, the managers grasp the progress by using various metrics obtained from the source code and the update history information. However, since there are no appropriate metrics to the upper processes, it is difficult to grasp the progress of them. In this paper, we propose a measurement process on the modifications of class diagrams which can be used to estimate the effort to change from one class diagram to another. To measure the modifications of class diagrams, first, we construct a tree based on the generalizations and the inner information of the classes in the class diagrams. Next, we obtain the sequence of edit operations to change one tree to another. Then, each edit operation is weighted based on the assigned cost. Finally, by calculating a sum of the total costs, we obtain the modifications of the class diagrams. We have utilized this approach to measure the modifications of several class diagrams and compared with the data from actual software development. As the results, we confirmed the validity of using this approach to estimate the effort that is taken during the modification.

Keywords

Project Management
Design Process
Class Diagram
Modification
Metric

Contents

List of Figures

1 Introduction

Software systems play an critical role in our society, as they are used in daily basis such as in banking and communication systems. In recent years, software systems become huge[1] and encompass increasing amount of information, which accelerates the complexity of the software development. Due to the increased complexity, it is a challenge to reduce the term and the cost of the software development to meet the demands from software industries.

In order to meet the demands, a software management with quantitative data is necessary[2]. Adopting the quantitative management, the managers can grasp the actual progress of the development and measure against the delay of them.

In the lower processes of the software development, a variety of metrics are used to grasp the progress. For example, source code modifications[3] are used to grasp the progress of the development, lines of code is used to evaluate the size of the system, complexity metrics (e.g. cyclomatic complexity[4] and CK metrics[5]) are used to evaluate the complexity of the system. In the higher processes, some metrics also exist such as "estimation and the actual effort for modifying the system requirement[6]." However, the assessment is time consuming since the metrics are measured by hand. Moreover, the developers should not only engage in their own jobs but also in other tasks such as sudden meeting with others. Thus, it is difficult to accurately measure the effort individually. This problem would be solved if the effort for modifying design documents could be estimated from the updated information between two versions of design documents.

Unified Modeling Language (UML) is a widely used modeling language for designing software systems. UML provides several diagrams for many purposes. One of the most important diagrams is class diagrams. Therefore, as the first step of estimating the effort from design documents, we focus on class diagrams.

Traditionally, several metrics have being measured to manage projects e.g. the number of classes, the number of attributes, and the number of operations[7] from class diagrams. However, since they are size metrics, it is difficult to estimate the effort for modifying class diagrams. The size metrics represent the size of the object on one occasion, thus they do not reflect the context of modification.

In this paper, we first propose the measurement process on the *modifications of class diagrams* (in the following, we call it *MCD*) which can be used to estimate the effort to

change one class diagram to another. Measurement of MCD consist of three processes. (1) Constructing trees based on the generalizations and the inner information of the classes in the class diagram. (2) Obtaining the sequence of editing operations to change one tree to another. (3) By weighting each of the editing operations and accumulating them, we obtain the MCD. Second, we introduce the method for implementing the proposed method. Finally, we evaluate MCD in terms of its validity in estimating the class diagram modification effort and implemented tool in terms of running time by adapting versions of class diagrams in the actual development data. The result showed the proposed MCD correlates with the modification effort, and the measurement costs a few seconds for a middle size project, thus the cost for measuring is sufficiently low.

In the following, Section 2 introduces terminologies we use in this paper and refer the algorithm we used in our method. Section 3 proposes the metric and the overview of the measurement. Section 4 shows how we implemented the tool realizing the proposed method mainly about its feature for easily change input files as well as the idea for optimization. Section 5 evaluates if MCD is applicable for estimating the modification effort, in terms of the correlation between MCD and the actual modification effort as well as the time spent for measuring MCD. Section 6 introduces MCD applications. In Section 7, known issues for MCD are described. Section 8 reviews the related work. Finally, Section 9 summarizes the paper and refers to the future works.

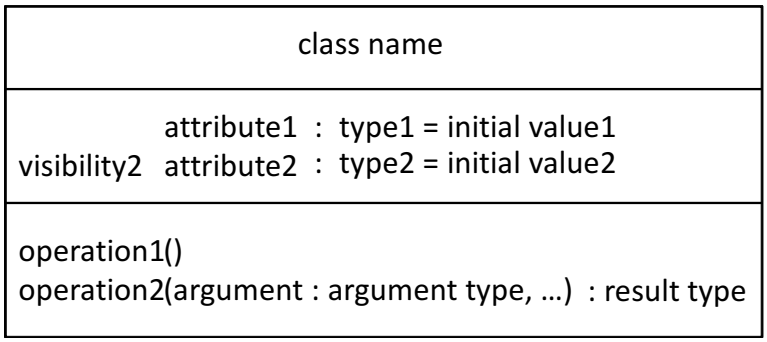

Figure 1: Structure of a Class in a Class Diagram

Figure 2: Structure of an Interface in a Class Diagram

2 Preliminaries

Our method of measuring the modification from given two class diagrams is based on the algorithm presented by Chawathe et al.[8]. In the following, we describe a class diagram, and then, we introduce the terminologies useful to understand the latter sections and an overview of the original algorithm MHDIFF.

2.1 Class Diagram

The class diagram is a type of static structure diagram in UML, which describes classes and the relationships between the classes in a software system. Figure 1 shows the structure of a class in a class diagram.

The vertices in a class diagram are classes. A Class in a class diagram consist of three area, which are separated by horizontal lines. The upper area represents the name of the class. The middle area represents the attributes of the class, where one can determine its attribute names, their types, initial values, and visibility. The bottom area represents operations of the class. This area provides the name of the operations, their arguments, argument types, and result types. Figure 2 shows the structure of an interface in a class diagram.

The interface can be considered similar to a class. The interface is depicted as a

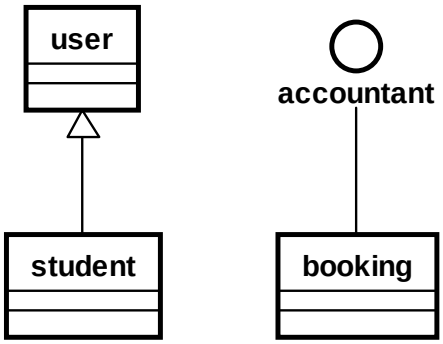

Figure 3: Example of a generalization and a realization

circle. It contains attributes and operations, however, we omit the detailed explanations about their notations.

A class diagram may contain several types of relations between the classes and the interfaces, however, in this paper, we only explain a generalization and a realization. A generalization shows a class inheriting another class. It is illustrated as an arrow originating from the sub class to the super class. A realization indicates that a class realizes an interface. It is illustrated as a solid line between the bottom of the interface to the top of the class. As an example of the generalization and realization, Figure 3 shows that "student" inherits "user", and "booking" realizes "accountant".

A class diagram may also provide other information. Detailed specification is available in [9].

2.2 Terminologies

In this sub-section, we introduce the terminologies used in graph theory and the terms specifically used in the paper using Figure 4. In this paper, the labels of nodes are located on the right side of the nodes.

We use $l(n)$ as a label of the node n. We denote a tree T using its nodes N, parent function p, and labeling function l, and write $T = (N, p, l)$. The children of a node n are denoted by $C(n)$, and its parent by $P(n)$. In Figure 4, the label of a node 2 is "user" and depicted as $l(2)$. A node 4 is denoted as C(2), and a node 1 is denoted as P(2). A root node is an ancestor of all the other nodes in the tree. And it is labeled as "ROOT". A leaf node is a node without child node. An inner nodes contains one or more child nodes.

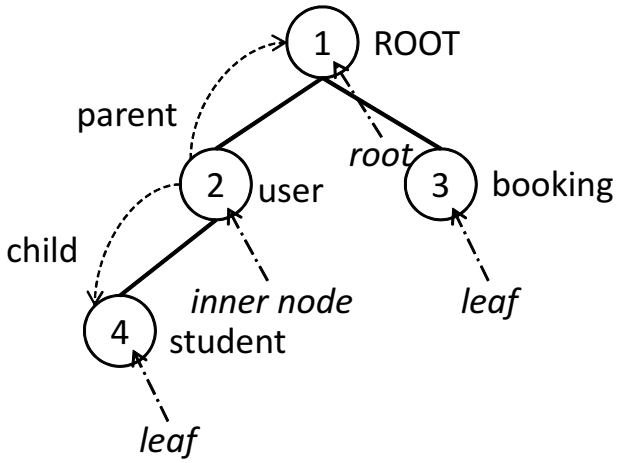

Figure 4: Example of a Tree

2.3 MHDIFF

MHDIFF[8] is the tree difference algorithm proposed by Chawathe et al. The input of MHDIFF is two trees and the output is an edit script. The Edit script is a sequence of edit operations which transforms one tree to another. The edit script is composed of the following six edit operations.

INS Intuitively, an insertion operation creates a new tree node with a given label, and places it at a given position in the tree. The position of the new node n in the tree is specified by giving its parent node p and a subset C of the children of p. The result of this operation is that n is a child of p, and the nodes C, that were originally children of p, are now children of the newly inserted node n.

Formally, an insertion operation is denoted by INS(n, v, p, C), where n is the (unique) identifier of the new node, v is the label of the new node, $p \in N_1$ is the node that is to be the parent of n, and $C \subseteq C(p)$ is the set of nodes that are to be the children of n. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup n$, $p_2(n) = p$, $p_2(c) = n$, $\forall c \in C$, $p_2(c) = p_1(c)$, $\forall c \in N_1 - C$, $l_2(n) = v$, and $l_2(m) = l_1(m)$, $\forall m \in N_1$.

DEL This operation is the inverse of the insertion operation. Intuitively, DEL(n) causes n to disappear from the tree; the children of n are now the children of the (old) parent of n. The root of the tree cannot be deleted.

Formally, a deletion operation is denoted by DEL(n), where $n \in N_1$ and n is not the root of T_1. When applied to $T_1 = (N_1, p_1, l_1)$, we get a tree $T_2 = (N_2, p_2, l_2)$ with $N_2 = N_1 - n$, $p_2(c) = p_1(n)$, $\forall c \in C(n)$, $p_2(c) = p_1(c)$, $\forall c \in N_2 - C(n)$, and $l_2(m) = l_1(m)$, $\forall m \in N_2$.

UPD The operation UPD(n, v) changes the label of the node n to v.

Formally, an update operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by UPD(n, v), where $n \in N_1$, and produces $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $p_2 = p_1$, $l_2(n) = v$, and $l_2(m) = l_1(m)$, $\forall m \in N_2 - n$.

MOV A move operation MOV(n, p) moves the subtree rooted at n to another position in the tree. The new position is specified by giving the new parent of the node, p. The root cannot be moved.

Formally, a move operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by MOV(n, p), where $n, p \in N_1$, and p is not in the subtree rooted at n. The resulting tree is $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1$, $l_2 = l_1$, $p_2(n) = p$, and $p_2(c) = p_1(c)$, $\forall c \in N_2 - n$.

COPY A copy operation COPY(m, p) copies the subtree rooted at n to another position. The new position is specified by giving the node p that is to be the parent of the new copy. The root cannot be copied.

Formally, a copy operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by COPY(n, p), where $n, p \in N_1$, and nZ is not the root. Let $T_3 = (N_3, p_3, l_3)$ be a new tree that is isomorphic to the subtree of T_1 rooted at n, and let n' be the root of T_3. The result of the copy operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 \cup N_3$, $l_2(m) = l_1(m)$, $\forall m \in N_1$, $l_2(m) = l_3(m)$, $\forall c \in N_3$, $p_2(n') = p$, $p_2(m) = p_1(m)$, $\forall m \in N_1$, and $p_2(m) = p_3(m)$, $\forall m \in N_3$.

GLUE This operation is the inverse of a COPY operation. Given two nodes n_1 and n_2 such that the subtrees rooted at n_1 and n_2 are isomorphic, GLU(n_1, n_2) causes the subtree rooted at n_1 to disappear. (It is conceptually "united" with the subtree rooted at n_2.) The root cannot be glued. Although the GLU operation may seem unusual, note that it is a natural choice for an edit operation given the existence of the COPY operation. The symmetry in the structure of edit operations is useful in the design of the algorithm.

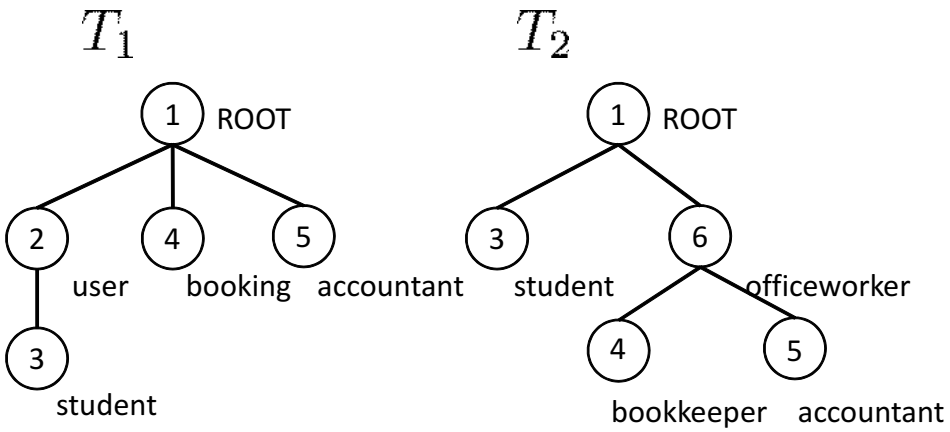

Figure 5: Sample Trees

Formally, a glue operation applied to $T_1 = (N_1, p_1, l_1)$ is denoted by GLUE(n_1, n_2). Let T_3 be the subtree rooted at n_1, and let $T_4 = (N_4, p_4, l_4)$ be the subtree rooted at n_2. The precondition of this GLUE operation is that T_4 is isomorphic to $T_3 - T_4$. The result of the glue operation is the tree $T_2 = (N_2, p_2, l_2)$, where $N_2 = N_1 - N_4$, $p_2(c) = p_1(c)$, $\forall c \in N_2$, and $l_2(c) = l_1(c)$, $\forall c \in N_2$.

Here, we use an example to explain MHDIFF algorithm. Figure 5 shows the sample input trees. T_1 depicts an original tree and T_2 depicts a modified tree. Note that identifications in each vertex is intentionally assigned for the purpose of explanation; however, MHDIFF and our method do not use the identifications in each element used by UML modeling tools. In comparison between T_1 and T_2, the node 2 labeled "user" is deleted in T_2 and a new node 6 labeled "officeworker" is inserted as $C(1)$ in T_2. Moreover, the label "booking" at the node 4 is modified to "bookkeeper" and "accountant" at the node 5 becomes a child of "officeworker" at the node 6. During the algorithm process, MHDIFF assigns the operation (*annotates*) to each vertex as shown in Figure 6. In Figure 6, dot-dashed arrows show the vertices are annotated by edit operations.

These annotated edit operations coincide with intuition that they are synonymous with the modifications described in the above paragraph. In this example, where the node 2 labeled "user" in T_1 is deleted in T_2, the corresponding annotated operation is DEL(2).

At the end, MHDIFF outputs an edit script, a sequence of edit operations described in Section 2.3. In this example, the edit script, ε, may

$\varepsilon = \{$INS(6, officeworker, 1, $\{4, 5\}$), DEL(2), UPD(4, bookkeeper), MOV(5,6)$\}$

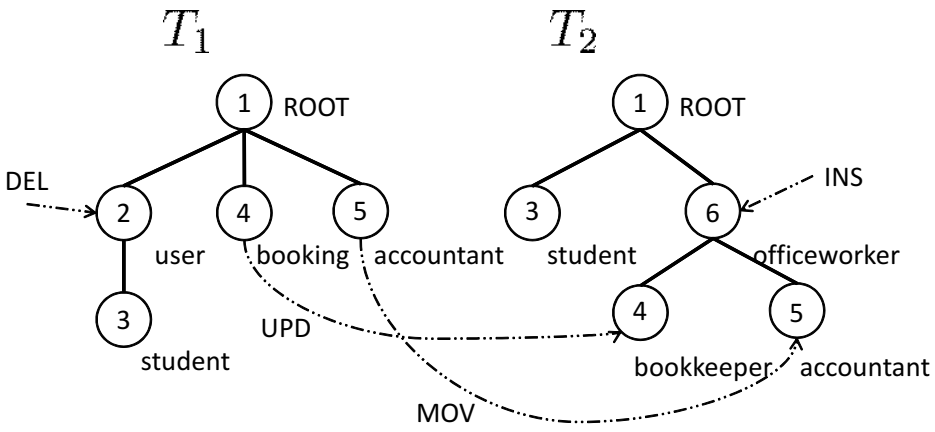

Figure 6: Annotated Trees

3 A New Metric for Modifications of Class Diagrams

3.1 Modifications of Class Diagrams

Assume that a class diagram C_2 is created based on a class diagrams C_1. That is, C_2 is obtained by applying some modifications to C_1 in the design phase. Then, MCD evaluates the effort of modifying C_1 to C_2 in design phase of the software development.

First, this method calculates and determines an edit script which is the sequence of edit operations. For an instance, if we apply the edit operations to the class diagram C_1, the class diagram C_2 is obtained. Next, a proposed method assigns a weight for each edit operation in the edit script. Finally the MCD between the C_1 and C_2 are calculated as a sum of the total weights.

We would mention why we designed MCD as a scalar value. During the process, the edit script is obtained. As it consists of a sequence of edit operations, we can represent the modification as a vector value $\mathcal{V}$, such as

$$\mathcal{V} = \begin{pmatrix} INS \\ DEL \\ \vdots \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ \vdots \end{pmatrix}.$$

Both notations have advantages and disadvantages. The process of measuring MCD is very similar to the Function Point method[10]. The Function Point is a scalar value which measures a functional size of the software independent of any development tools or technologies. It differs from the size of the code particularly in light of its independency.

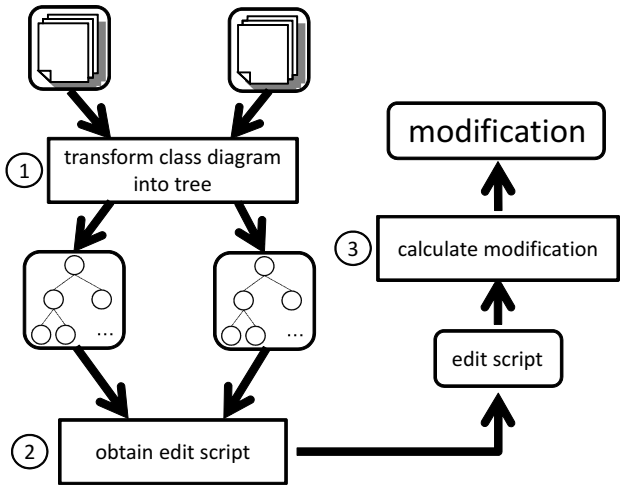

Figure 7: Flow of the modification measurement

The size of the code may differ even if a developer implements the same functionality when he/she uses different programming languages. The Function Point method first evaluates the system and obtains the number of functions. The functions are categorized in 5 elements and weighted. Lastly, the Function Point is obtained by a sum of the weighted values in each category in the Function Point method.

The Function Point share similar advantages[11] and disadvantages[12] to MCD. According to Furey[11], the Function Point provides a better way to compare organizations and project productivities. It is much easier to evaluate and compare between two variables if a scalar value is utilized. Therefore, we also design the MCD as a scalar value.

3.2 Flow of the Measurement

In this section, we present a flow of MCD measurement. The measurement consists of the following three steps.

Step1: Transform two input class diagrams C_1 and C_2 into trees.

Step2: Calculate and obtain the edit script which represents a sequence of edit operations, which transforms C_1 to C_2. This step is essentially the same process as MHDIFF.

Step3: By weighting each edit operation and accumulating the weight, the MCD is obtained.

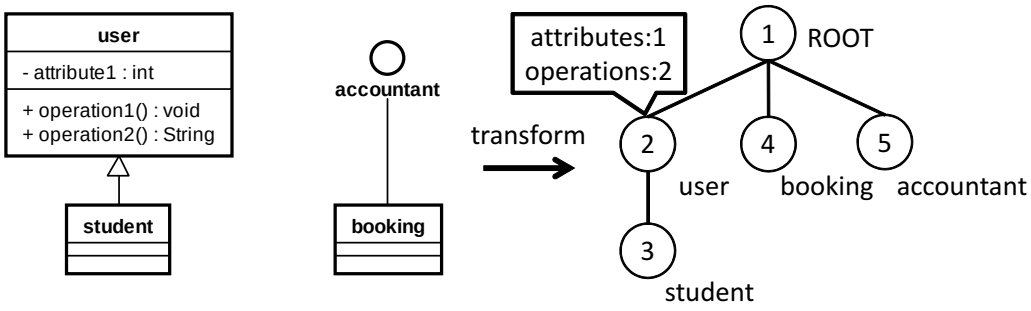

Figure 8: Tree Construction

Figure 7 shows the flow of measurement. Each number in Figure 7 corresponds to the steps described above.

3.3 Tree Construction

In the tree construction, the classes and the interfaces in the class diagrams are transformed into nodes. Each node of the constructed tree includes inner information, which represents the information of the original class. The inner information consists of three information: the name of the class, the number of attributes, and the number of operations. The edges in trees are derived from the class relations. The transformation of edges only targets on generalizations of classes. Other relations in class diagrams such as realization and composition are not be transformed. Therefore, the existence of these relations does not affect the modification. The class with no super class in the original class diagram connects to the newly added node, “ROOT”.

Figure 8 shows an example of the tree construction. The class “user”, “student”, “booking” and the interface “accountant” is transformed into nodes. The class “user” in the left hand has one attribute and two operations, thus the corresponding node 2 in the right hand has inner information, the number of attribute is one and operations is two. The class “student” has the super class, therefore the relation is transformed into edge of the tree. The others has no super classes, thus the corresponding nodes in the right hand connect to “ROOT”. Recall the relations except generalization does not transformed into edges. So the realization between “accountant” and “booking” is disregarded. In the paper, we will call the inner information of node m as *Inner*(m) e.g. the inner information of the node 2 in Figure 8 is described as Inner(2).

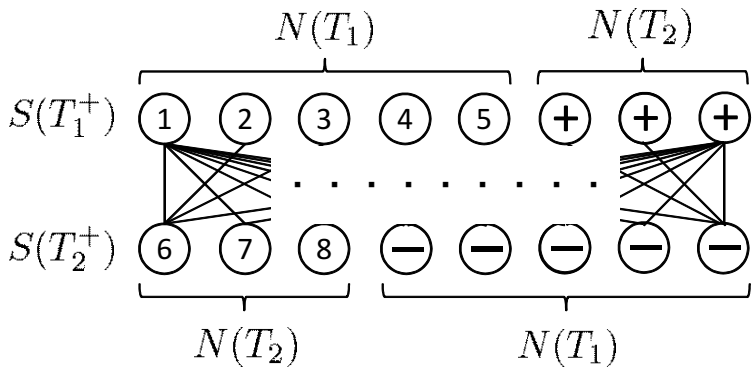

Figure 9: Bipartite Graph

The two class diagrams are independently transformed into trees. In the following, we call the tree T_1 derived from the first input class diagram and T_2 derived from the second input class diagram.

3.4 Weighted Bipartite Graph Construction

3.4.1 Graph Construction

Based on the nodes in T_1 and T_2 in Section 3.3, a complete bipartite graph is constructed. In order to eliminate COPY and GLUE operations, we modified a method in MHDIFF. Originally, a complete bipartite graph in MHDIFF is called *Induced Graph*, where the number of added special nodes, $\oplus$ and $\ominus$, is limited to one. However, our method does not limit the number of special nodes which are added in the graph.

Let be define a function $N(T_x)$ as the number of nodes in a tree T_x and $S(T_x)$ as the set of nodes in the tree T_x. For special nodes $\oplus$ and $\ominus$, $N(\oplus)$ represents the number of special nodes $\oplus$ and $S(\oplus)$ represents the set of nodes $\oplus$. Assuming $T_1^+ = T_1 \cup S(\oplus)$ and $T_2^+ = T_2 \cup S(\ominus)$, the nodes are added to satisfy $N(T_1^+) = N(T_2^+)$. After adding the nodes to the bipartite graph, edges are added to the graph between nodes in each side of the graph i.e. the edge e is added between the nodes $m \in S(T_1^+)$ and $n \in S(T_2^+)$. Sample of constructed complete bipartite graph is shown in Figure 9.

3.4.2 Cost Assignment

Each edge of the bipartite graph is weighted as a cost. The cost of an edge represents the difference between nodes from one to other. Originally in MHDIFF, a node only

contains a label; therefore, a cost of the edge between the nodes m and n is the difference between $l(m)$ and $l(n)$. However, since we use the nodes with inner information, we must define the difference properly between nodes. The equation 1 represents the difference between node m and n.

$$\text{Diff}(m, n) = \text{LabelUPD}(m, n) + \text{InnerChange}(m, n) \tag{1}$$

In this equation, LabelUPD(m, n) and InnerChange(m, n) are defined by

$$\text{LabelUPD}(m, n) = \text{Len}(m) + \text{Len}(n) - 2 \times \text{COMMON}(m, n)$$

and

$$\text{InnerChange}(m, n) = 2 \times \{|\text{AttrN}(m) - \text{AttrN}(n)| + |\text{OprN}(m) - \text{OprN}(n)|\},$$

respectively. Len(m) represents the label length of the node m, AttrN(m) represents the number of attributes in m, OprN(m) represents the number of operations in m and COMMON(m, n) represents *Longest Common Subsequence* (LCS) in each label of the node.

Thus, the cost of edge e connecting from m and to n is

$$\text{Cost}(e) = \text{Diff}(m, n).$$

3.5 Bipartite Graph Matching

After applying Weighted Bipartite Graph Construction, we obtained a complete weighted bipartite graph, each edge of the complete bipartite graph (see Figure 9) has costs representing the difference between nodes. In this section, first we present about how to prune (eliminate) edges, and next how to obtain a minimum weight perfect matching of the bipartite graph.

3.5.1 Pruning Edges

One can obtain a perfectly matched bipartite graph by applying the algorithm described in Section 3.5.2. However, the calculation described in Section 3.5.2 requires $O(n^2 logn)$ and more and the calculation in this step requires $O(n^2)$. Therefore by pruning unnecessary edges in the graph, we reduce the total calculation cost.

We describe how to prune (eliminate) the apparently unnecessary edges in the weighted complete bipartite graph. In original MHDIFF, the two pruning rules are defined. Both of them are based on the idea that the target edges can be replaced by other edges. However, in order to omit some edit operations (see Section 3.8), we only use one of the pruning rules. Intuitively, the pruning targets are edges connected from m and to n whose difference Diff(m, n) is sufficiently high. They are apparently not necessary since the accumulated cost of the final edges is smaller if deleting m and adding n than remaining them. Remaining edges will be annotated and affect MCD. Therefore, if we define the cost representation of one operation as Cost(Opr), an edge connected from m and to n as $[m, n]$, the pruning targets are edges $e = [m, n]$ which satisfy Inequation 2.

$$\textit{Diff}(m, n) \geq Cost(\textit{INS}(n)) + Cost(\textit{DEL}) \tag{2}$$

Rigorously, the elimination condition should satisfy not only Inequation 2 but also other conditions. Suppose finally remaining edge $e' = [m', n']$, where m' is one element of $C(m)$ and n' is one element of $C(n)$, it occurs no problem when the edge $e = [m, n]$ is also remaining. However, if e does not exist, m' is forced to move to n'. Figure 10 (a) shows an example without any problem using Inequation 2. However, with respect to Figure (b), $e' = [m', n']$ exists but dose not exist $e = [m, n]$. This situation forces m' to move n'. Define α as a number of finally remained edges connected from $C(m)$ to $C(n)$, the pruning rule should rigorously satisfy the following condition:

$$\textit{Diff}(m, n) + \alpha \times Cost(\textit{MOV}) \geq Cost(\textit{INS}(n)) + Cost(\textit{DEL}).$$

However, since the number of edges connected from $C(m)$ to $C(n)$ will be determined after applying the process described in Section 3.5.2, the number of these edges cannot be determined. This impasse cannot be broken. Hence, we use the pruning rule that only satisfy Inequation 2.

3.5.2 Perfect Matching

A problem obtaining a minimum weight perfect matching of the bipartite graph is called assignment problem. $O(n^2 logn)$ of the fastest algorithm is described in [13]. We used Hungarian Method[14]; the calculation requires $O(n^3)$ for the ease of implementation. After applying this step, a graph shown in Figure 11 is obtained.

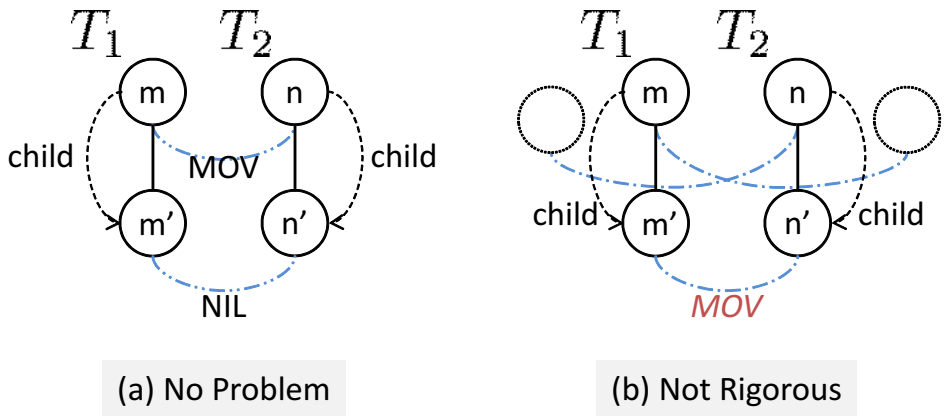

Figure 10: Pruning Problem

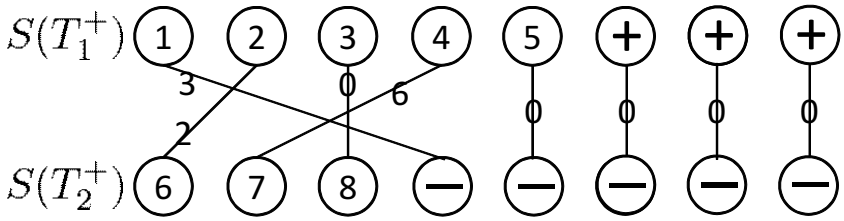

Figure 11: Minimum Weight Perfect Matching of the Bipartite Graph

3.6 Edit Script Acquisition

After applying Bipartite Graph Matching, the minimum weight perfect matching of the bipartite graph is obtained. Next, an edit operation is annotated to each edge. Originally, in MHDIFF, there are six edit operations, however, we use only four edit operations. The reason for this is described in Section 4.2.

Here we describe conditions for annotating the following edit operation on each edge e.

NIL NIL is annotated when e is connected from $m \in S(T_1^+)$ to $n \in S(T_2^+)$, inner information of m and n is the same and the edge connected from $P(m)$ to $P(n)$ exists.

INS and DEL INS is annotated when a node n which is not $\ominus$ connects to $\oplus$. DEL is annotated when a node m which is not $\oplus$ connects to $\ominus$.

forall $e = [m, n]$ **do** $\{$
  if (Inner(m) = Inner(n) && exist $e' = [P(m), P(n)]$) **then** $e \leftarrow$ NIL
  else if ($m = \oplus$ && $n = \ominus$) **then** $e \leftarrow$ NIL
  else if ($m \in T_1^+ = \oplus$ && $n \in T_2^+ \neq \ominus$ && exist edge $e = [m, n]$) **then** $e \leftarrow$ INS
  else if ($m \in T_1^+ \neq \oplus$ && $n \in T_2^+ = \ominus$ && exist edge $e = [m, n]$) **then** $e \leftarrow$ DEL
  else if (not exist $e' = [P(m), P(n)]$) **then** $e \leftarrow$ MOV
  else if (Inner(m) != Inner(n)) **then** $\{$
    $e \leftarrow$ UPD
    if (not exist $e' = [P(m), P(n)]$) **then** $e \leftarrow$ MOV.UPD
  $\}$
$\}$

Figure 12: Pseudo Code for *Annotate*

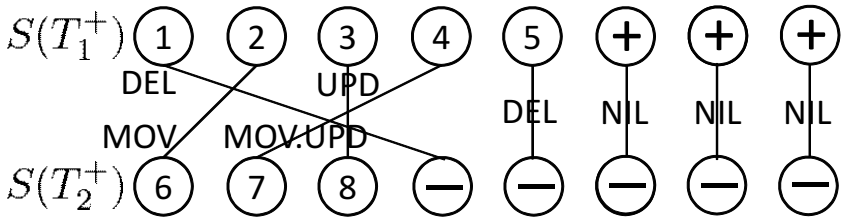

Figure 13: Annotated Bipartite graph

MOV MOV is annotated when an edge e connected from $m \in S(T_1^+)$ to $n \in S(T_2^+)$ exists as well as the edge connected from $P(m)$ to $P(n)$ does not exist.

UPD and MOV.UPD UPD is annotated when an edge e is connected from $m \in S(T_1^+)$ to $n \in S(T_2^+)$ whose inner information is different. In this time, if the edge also satisfies condition MOV, then MOV.UPD is annotated.

Figure 12 shows the pseudo code for *Annotate*. Figure 13 shows an example of the bipartite graph each edge is assigned the edit operation.

It is easy to obtain an edit script if the annotated bipartite graph exists. One can obtain the graph just counting the number of each edit operations annotated in the bipartite graph. There are some ordering constraints in MHDIFF, however, in our method, there is no ordering constraint. Originally, COPY and GLUE operation impose the ordering

constraint to the edit script. As our method omits these two operations, there is no ordering constraint.

For example, the edit script from the annotated graph in Figure 13 is

$$\{\text{DEL}(1), \text{DEL}(5), \text{UPD}(3, 8), \text{MOV}(2, 6), \text{MOV.UPD}(4, 7)\}.$$

3.7 Calculation for Modifications of class diagrams

Calculation of MCD is accomplished in following three steps. First, we count all edit operations from an edit script. Second, we weigh each operation based on the assigned cost. Finally, by accumulating the costs, we obtain the MCD.

Each operation cost is defined by users of their purpose. An example is shown in Section 5.1.

3.8 MHDIFF Optimization

We used MHDIFF-like algorithm in our method to measure the modification between given two class diagrams (C_1, C_2). Our method measures the modification from C_1 to C_2. Thus it can tailor in some points.

Firstly, we omit two edit operations, COPY and GLUE. COPY indicates that one sub tree in the tree is copied and inserted in another location of the tree. For class diagram, this operation means a set of classes is copied and inserted in another location of the same class diagram. In the design phase, class diagrams may glow only by adding of new classes. Hence, our proposed method should allow insertion and disallow copy of some sub trees. This is the reason we omit COPY operation. GLUE is the inverse operation of the COPY, therefore it is not appropriate for this time.

Secondly, we optimized a condition of operation assignment for MOV. Originally, MHDIFF does not assign MOV to a child node if the inserted node is a inner node. Figure 14 shows the insertion example using MHDIFF. NIL showed that the assigned node has not been operated in any edit operation.

However, this does not meet the intuition. Let us think the situation that the class M inherits the class N (see Figure 15). To insert a new class X as a super class of M and a sub class of N, the developer needs to create the class X and inherits N and change the generalization edge of M. To design a method to meet the intuition, we changed the MOV condition.

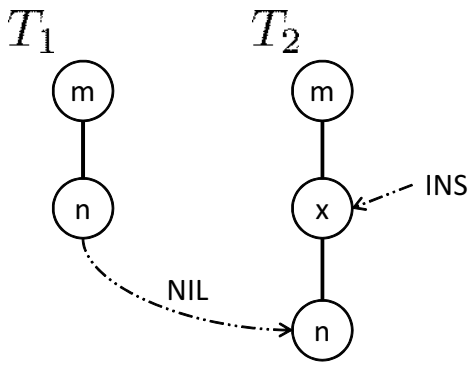

Figure 14: Original Insertion

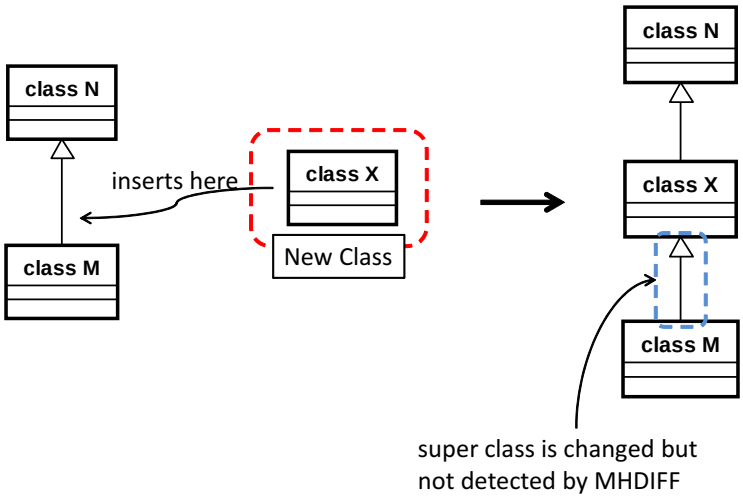

Figure 15: Problem of Original MOV Condition

The detailed MOV condition is described in Section 3.6.

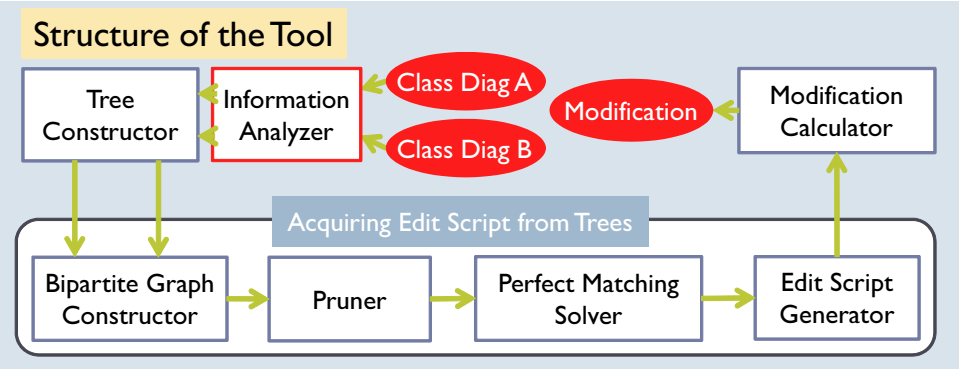

Figure 16: Structure of the Tool

4 Implementation

4.1 Structure of the Tool

Figure 16 shows the structure of the implemented tool.

The tool firstly inputs two class diagrams. They are analyzed and abstracted into information independent from input files. Using the information, Tree Constructor constructs trees. Bipartite Graph Constructor inputs two trees and constructs the weighted complete bipartite graph. Some edges in the weighted complete bipartite graph are eliminated by pruner. The weighted pruned bipartite graph is transformed into the minimum weight perfect matching of the bipartite graph by Perfect Matching Solver. It is analyzed by Edit Script Generator and obtained the edit script. From the edit script, Modification Calculator calculates the MCD.

The modules obtaining the information from input files (Information Analyzer) and constructing trees (Tree Constructor) are distinct (See Figure 16). Thus, it is easy to change the type of input files, for the developer develops a module which obtains the information from the targets and replaces the module. This architecture yields a large profit. If one is eager to calculate the modification between two source code in design level abstraction, he/she only need to develop a module which analyzes source codes and changes the module.

4.2 Ideas for Optimization

The method described in Section 3 first constructs a complete bipartite graph and then obtains a perfect matching. This is good when the input files are absolutely differ-

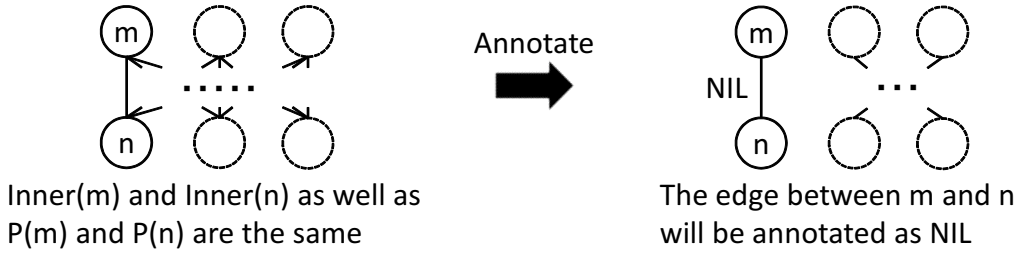

Figure 17: Optimization Idea

ent. Constructing the "complete" bipartite graph in MHDIFF assumes that all nodes may have been changed, since each edge implies a possibility to be matched between nodes. However, since our proposed method measures the modifications between given two class diagrams whose ancestor is identical, most of the information in the class diagrams would not be changed; The design is rarely changed completely.

Based on this idea, it is not necessary to construct a complete bipartite graph. If there are nodes $m \in T_1$ and $n \in T_2$ whose inner information Inner(m) and Inner(n) as well as the parent nodes $P(m)$ and $P(n)$ are identical, the edge between them will be annotated as NIL. Therefore, the method only adds an edge between them. Thus, the method does not require to construct bipartite graph for all nodes (see Figure 17.)

This method first investigates nodes and classifies them. In the first set, there are node pairs which have the same node (the same inner information and the same parent.) For the nodes in this set, one edge is added to each node. In the second set, there are nodes each of which does not have the same node. In this set of the nodes, the method constructs a complete bipartite graph by using the method described in Section 3.4.

However, this method contains an issue when the above idea is simply implemented. Assume that two class diagrams, where each diagram holds two classes A and B, satisfy Equation 3

$$\text{Inner}(A) = \text{Inner}(B), \text{ however,} P(A) \neq P(B). \tag{3}$$

. Applying the above optimization, these classes can be two types of annotation (Type 1 and Type 2 in Figure 18.) The two input class diagrams are identical. Thus, Type 1 annotation which produces no MCD is intuitively correct. Hence, the method generates edges between the nodes m and n does not only hold the same inner information (Inner(m) and Inner(n)) but also the same parent ($P(m)$ and $P(n)$.)

This optimization reduces many edges. When p nodes in $S(T_1)$ and $S(T_2)$ have the

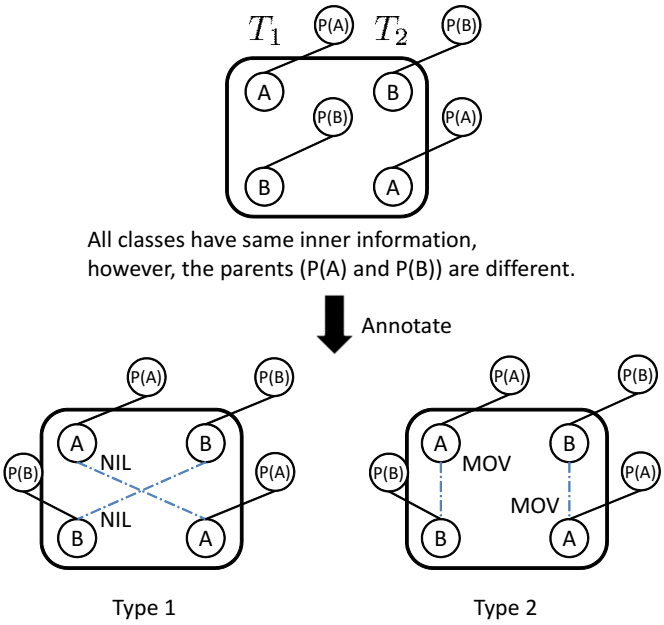

Figure 18: Problem with Intuitive Optimization

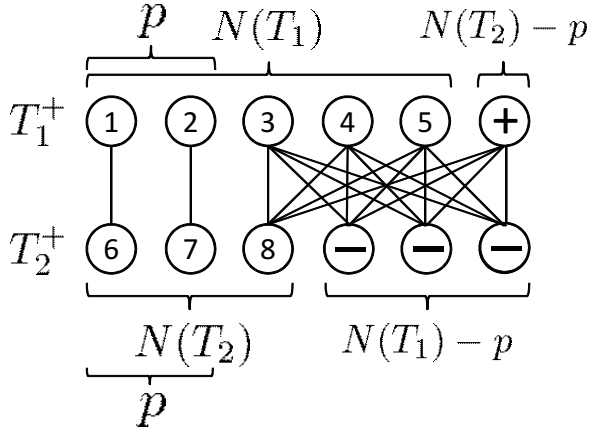

Figure 19: Constructed Bipartite Graph Applying Optimization

same inner information, the constructed bipartite graph is shown in Figure 19. Suppose the number of classes in two input class diagrams is identical (n classes) and the number of classes modified is a fixed number, the number of edges in constructing a bipartite graph reduces from O(n^2) to O(n). Details are described in Appendix A.

5 Evaluation

In this section, we evaluate the system which has been implemented based on the proposed method. We evaluate the system based on two aspects. For the first aspect, we evaluate the validity of the MCD by analyzing the correlation between MCD and the effort for designing the class diagrams. For the second aspect, we evaluate the performance of the tool to measure the MCD.

5.1 Metric for Effort Estimation

5.1.1 Evaluation Target

The MCD is designed to evaluate an effort for modifying class diagrams (see Section 3.1). "An effort for modifying class diagrams" implies not only its real modification effort but also other effort e.g. time spent for thinking about the design or meeting which determines it. Intuitively, the best method to evaluate the validity of the MCD is to analyze the correlation between the MCD and the effort spent in modifying the class diagram. However, it is difficult to obtain an actual effort for modifying class diagrams. Therefore, we use the estimated effort obtained from an actual job assignment data in design phase to estimate the correlation.

We use the actual development data for analysis provided by "IT Spiral[15]". These data include daily UML diagrams produced during the design phase. This project started on March 6, 2007, and lasted for 45 days, which resulted in 38 class diagrams in the design phase. In this project, the class diagram contained a total of 361 classes.

Since the data do not indicate the work hours of each person involved in the project, we assume the eight hours of design work per person per day. In the design phase, 3 developers were involved in the project, therefore, we estimate $8 \times 3 = 24$ hours of the effort in one day. We can estimate the total effort spent in the design phase by using this assumption since the data includes the job assignment data for Microsoft Office Project©format. We manually modified the calculated effort when there are records indicating that the developers attended a meeting. For example, they had a review in Wakayama University (It is not close to the development company) on April 23 for 7 hours. We regard the total effort on the day as $7 \times 3 = 21$ hours.

The effort spent in design phase may have a strong correlation with the effort spent in modifying class diagrams, for the other diagrams described by UML is based on class

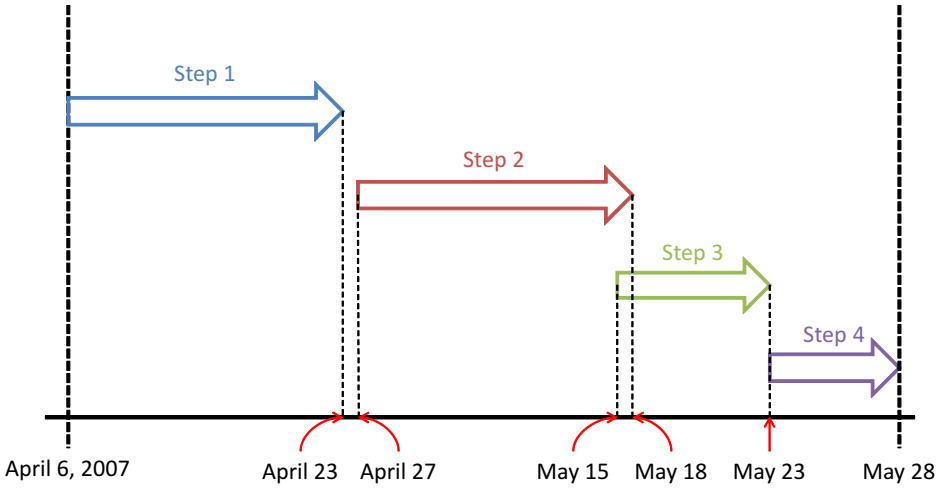

Figure 20: Each Period in Design Phase

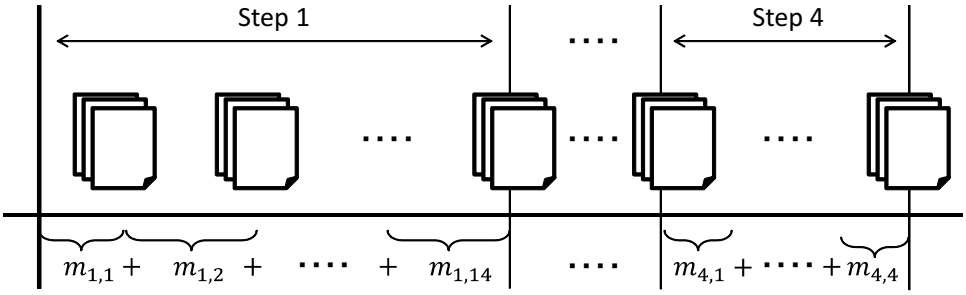

Figure 21: How to Measure the Total Measurement

diagrams. This assumption means that in the evaluation, we can substitute the total effort spent in modifying class diagram for the effort spent in design phase. Therefore, we use the estimated total effort spent in design phase in our evaluation.

The final class diagram was developed based on 4 steps and illustrated in Figure 20. Regarding the product of the each step as a version of class diagram, we analyze the correlation between the MCD and the total effort for modifying class diagrams in each step. The total MCD in one step is obtained by accumulating the MCD in each pair of class diagrams. For example, since there are 15 class diagrams in step 1, let $m_{x,1}$ be the MCD between C_1 and C_2 which is designed in step x, the total MCD m_1 is $m_1 = m_{1,1} + m_{1,2} + \cdots + m_{1,14}$ (see Figure 21.)

Note that Step 3 started (on May 15th) before Step 2 ended (on May 18th.) However, we simplified that Step 2 ended on May 11th, since the actual design ended on May 11th,

though, it had the review on May 18th.

Let Effort(x) be the total estimated effort spent in phase x. By analyzing the correlation between m_x and Effort(x), we will show the validity of the MCD as a metric for estimating an effort for modifying class diagrams.

5.1.2 Weight of the Each Edit Operation

We determined the weight of each operation as shown in Table 1.

In calculating a MCD from an edit script, it is more realistic to use flexible values rather than fixed values for weight of INS and UPD, since the weight shows the effort to modify one class to another. In regard to UPD, major differences in the classes require a large effort in comparison to the small differences requiring a small effort in a class modification in a class modification. Similarly in INS, an insertion of large class with many attributes and operations requires a large effort.

In order to realize the flexibility, we use the weight of UPD between the node m and n as Diff(m, n) as defined in Equation 1. For class diagrams, INS shows that one class is created and added. Thus, the cost of INS needs to reflect the followings:

1. The cost of class creation, and

2. The cost of modifying inner information.

Therefore we determined the weight of INS shown in Table 1. $\emptyset$ shows the empty node, which contains no attribute and operation. We assume that the weight of DEL is similar to the cost of creating a class. The weight of MOV is lower than DEL. MOV refers to a

Table 1: Weight of Each Edit Operation

Edit Operation	Weight
INS	$10 +$ InnerChange($m, \emptyset$)
DEL	10
MOV	5
UPD	Diff(m, n)
MOV.UPD	$5 +$ Diff(m, n)

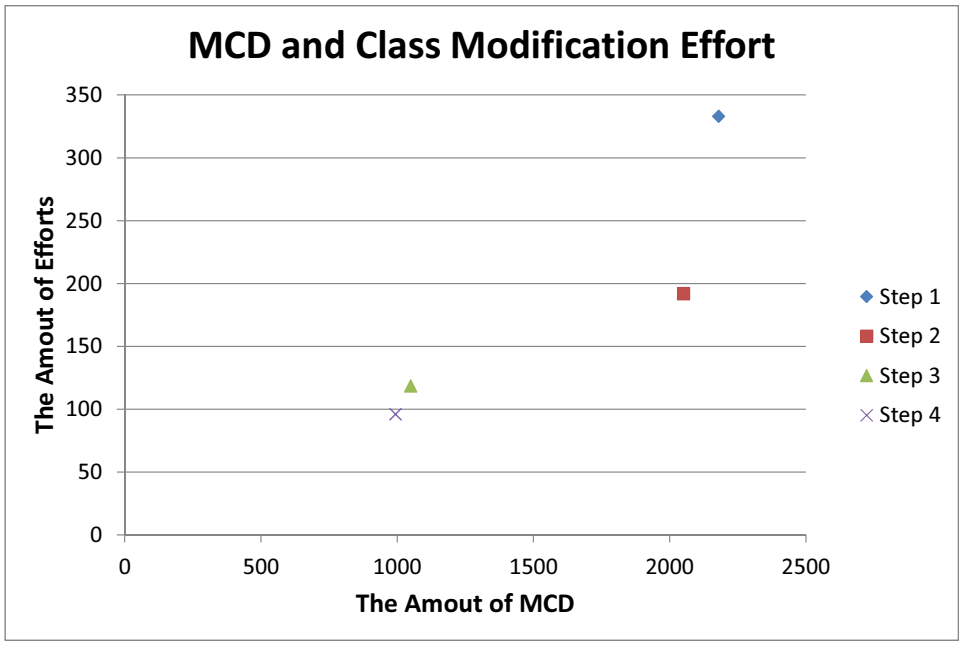

Figure 22: MCD and Modification Effort

change in the super class of a node. Its effort is lower than creating a class. The cost of UPD should reflect the difference between classes. Therefore we use the value Diff(m, n), proposed in Section 3.4.2. MOV.UPD is simply calculated by adding the cost of MOV and UPD.

5.1.3 Result

The measurement result of the MCD and the modification effort are shown in Figure 22. The x-axis shows the amount of MCD and y-axis shows the effort of modifying class diagram in each phase. Analyzing their correlation, the correlation coefficient between them is 0.882. Thus MCD and class diagram modification effort may have a strong correlation. However, as the result of testing the validity of the correlation coefficient by t-test, it rejects its null hypothesis by 5 percent of a significance level. It is because the number of target data is too small.

The result shows the MCD may be used in estimating the effort for modifying class diagrams, however, the evidence provided is not sufficient to make an conclusion. It is necessary to apply MCD method on more software development data to confirm the validity of this method.

Table 2: Running Time for Class Diagram Measurement

Step	Times[times]	Amout of Time[sec]	Ave. time[sec/times]
Step 1	14	20.9	1.5
Step 2	8	18.8	2.4
Step 3	5	13.9	2.8
Step 4	4	11.5	2.9
Amount	31	65.1	2.1

5.2 Running Time

We measure running time of our tool for the class diagrams. The target files are class diagrams used in Section 5.1.1. Table 2 shows the running time of each phase.

This class diagram finally have a total of 361 classes. Thus the scale of the project is middle[1]. Commonly, measurement requires $O(n^2)$ time. However, although the number of classes in the final class diagram is three times larger than that of in the first class diagram (91 classes), the average running time in phase 4 (2.9 sec.) is less than half in phase 1 (1.5 sec.) This is due to the optimization of the original MHDIFF described in Section 3.8.

The result shows the time spent for measuring the MCD is sufficiently low. Moreover, the measurement requires less than $O(n^2)$ times.

[1]In [16], the median lines of code for the similar type of software is 64000. This project finally implement around 200 classes, and the lines of code are 26033. If the developers implement all classes, lines of code are around 47000. This result shows the scale of project is middle.

6 Application

The metric measured by proposed method is applicable to some areas. In this section, we introduce two application areas.

6.1 Gap between Design and Implementation

It becomes different the designs of a system and its implementation during the development process[17]. It is because design reengineering is difficult[18], for reengineering requires a recognition of the current design and an accurate implementation, or even in some software developments, developers choose not to modify the designs.

One of the reasons against modifying the design is its high cost. However, when developers conduct software maintenance (especially in adaptive maintenance), they cannot utilize the design documents since they are different from their implementation. Gap between design and implementation result in maintenance difficult or even impossible.

Regarding an agile development, frequent iteration is a key to minimize risks[19]. This means the software is changed in short terms. That is, design documents and their implementation become different in each iteration, however, when the source code modification in one iteration only affects trivial changes to the design document, the developers need not to change it.

The problems would be solved if one can detect the proper timing of modifying design documents. The developers can concentrate on development until the adequate gap between design and implementation occurs. When the modification is under the threshold, it means the gap between them is not so high.

In order to accomplish it, first, we enabled the tool to measure a difference between design and implementation. Abstracting source code into class diagram information is realized by MASU[20].

Taguchi shows the result of application in [21]. The result shows that the high MCD from a design and implementation reveal the difference in design and implementation.

6.2 Estimation of the Functional Modification

Quantitative analysis is a key to effectively manage the software development. To analyze the software development quantitatively, the managers should acquire the progress information. However, if the development are taken place in remote area, it is difficult to

grasp the progress accurately. When the manager has access to only weekly report and source code from the software configuration management, it is also difficult to grasp it accurately. If the number of implemented function can be estimated from source code, it can be used to monitor a daily progress in functional level abstraction. In this way, we may estimate the progress of implementation by using the MCD.

In [21], Taguchi applies the MCD to some open source projects to evaluate its applicability as a metric to estimate functional modification. The result showed the MCD is more efficiently estimate the number of implemented functions than the modifications of source code and the number of classes.

7 Limitations

The proposed method has some limitations.

First, it does not have a sensibility to detect all modifications in a class diagram. Since it does not use all information included in a class diagram, the modification value zero does not always ensure the input class diagrams are the same. For example, when the input class diagrams differ only in some relations e.g. realization or association, the MCD is calculated to be zero.

Second, the weight of each edit operation in the proposed method has no scientific or statistical evidence. The weights are determined based on our experience in designing a class diagram and the weights determined in MHDIFF. The weights should be determined based on a lot of domain specific data in actual software development projects. However, researchers cannot easily obtain the actual data, since the data are often proprietary information. Due to the difficulties in obtaining actual data, we determined the weights of each edit operation intuitively based on the ideas referred in Section 5.1.2 and evaluate the proposed method.

Third, suppose that $m \in T_1$ is a node with many children (we refer it as *flower*), and m is modified largely in $m' \in T_2$. This situation forces the node m DEL, and m' INS since the cost Diff(m, m') is larger than DEL + INS. In this situation, large MCD is produced even if the modified class between two class diagrams is only m, for the parent of $C(m)$ is not the same; a newly added node is regarded as a alien node. Generally, the flower classes in the first input class diagram are largely modified, therefore, the method produce large MCD, for the children of the flower is detected as being moved (see Figure 23). We are not able to prevent this problem, for the method cannot know the children of flowers in the first class diagram remain in the same position in the second class diagram without having perfect matching of the bipartite graph. To prevent the problem, the method needs to know that the flower will be deleted before assigning costs to edges. Therefore the problem cannot be solved.

Forth, MCD method does not determine the importance of classes, therefore, modification on important and less important classes are regarded as having the same value. Intuitively, the modification on important classes e.g. Object class for Java language, is expected to produce higher MCD compared with less valuable classes e.g. classes defined by each developer. This problem may be solved if a depth of node in trees is considered

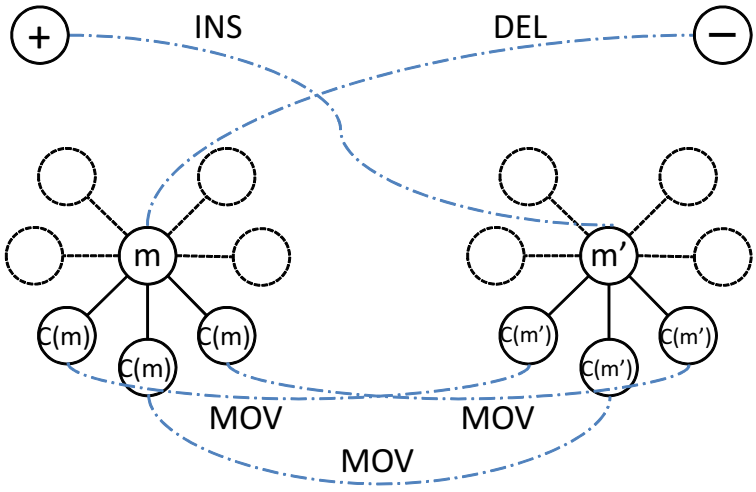

Figure 23: Large Modification Problem

and is included in weighting process. We consider this potential as one of the important work in MCD improvement.

8 Related Work

Some researchers study the method to detect changes between two class diagrams. Dirk Ohst et al. developed a method to visualize the changes between UML documents[22]. This method uses an identifier assigned in each element in diagrams. Thus we are not able to use this algorithm since we are also interested in measuring the class diagram modification between files such as source code. Martin Girschick proposes an algorithm to visualize the difference between class diagrams[23], called UMLDiff$_{cld}$. This algorithm is based on MHDIFF, which is the same algorithm in our proposed method. UMLDiff$_{cld}$ differs from the MCD method in detecting other modifications, *add*, *delete*, *rename*, *move*, *clone* and *modify property*. Moreover, UMLDiff$_{cld}$ visualizes the differences; our method output into a scalar value. Christoph Treude et al.[24] developed a new algorithm SiDiff. SiDiff calculates differences between class diagrams quickly. First, it calculates hashes of the elements, second constructs high-dimensional search trees called S^3V tree which is similar to LSD Tree[25]. Finally SiDiff searches the most similar element for all elements. Using S^3V tree, one can detect some elements that are similar to the target elements without investigating all the elements. Udo Kelte et al.[26] presents the difference algorithm for each UML diagrams. It translates input diagrams into unique data model and compare the elements. It does not use the unique class identifier assigned by some modeling tools.

Studies on measuring the modification between given two classes has not been conducted as we know. However, modification between source code is measured widely in practice. In [3], Aman et al. show the example to measure the modification between source code using Unix diff and detect the class to be widely modified or not.

Some of CK metrics[5] are measured from the relations between classes. However, since they are values for estimating the complexity of the class, the aspect of the metric is different.

In [27], Beat Fluri et al. shows the method of extracting the meaningful source code changes. They first abstract source code into AST(Abstract Syntax Tree). Next, they obtain the edit script of two AST derived from target source code using LaDiff[28]. Finally they transform the subsequence of edit script into *Change Types*. The rules of the transformations are defined in [29]. This research is correlated to our research, because both researches first transform the input files into trees and then obtain differences. The

distinction is the abstraction level of the target. The method abstracts source code into AST, however, our research abstracts class diagram into unordered tree.

With respect to the algorithm to use in our method, there are some selection. Similarity Flooding[30] is a matching algorithm which compares two graphs. It is applicable to graphs, therefore, we can use all relations in class diagram e.g. association and composition which is not used in our method. However, in order to detect the differences with meaningful operations, we cannot select the algorithm. We chose MHDIFF, which detect structurally meaningful changes between two trees and output a edit script, a sequence to edit operations which change one tree to another.

9 Conclusion

This paper proposed a method for measuring modifications of given two class diagrams (MCD). The MCD is a metric which is designed to estimate the effort for modifying one class diagram to another. MCD can be obtained by three steps. First, the class diagrams are transformed into trees. Second, the sequence of edit operations which transforms one tree to another are obtained. Finally, by weighting each edit operation and accumulating it, the MCD is obtained.

We represented how to implement the proposed method and the idea for optimization. Then, we evaluated the MCD in terms of its validity to estimate the class modification effort and its running time. The result showed the MCD probably can be used in estimating the class modification effort. We also referred to the application area of the MCD.

As for the future works, we will apply MCD method to the other development project and verify the validity. Also, we will continue to improve the approaches used in MCD method. Fluri, B. et al.[29] studied the classifications of the change types by analyzing a sequence of edit operations. The same idea can be used to improve the MCD method. For example, one node which is added right after it is deleted is found, we regard it as modification.

Acknowledgements

This work could not have been possible without valuable contributions from many people.

First, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, at the Osaka University, for his continuous support, encouragement, and adequate guidance of the work.

Also I would like to thank to Associate Professor, Kozo Okano, at the Osaka University for his guidance, valuable suggestions and discussions throughout this work.

I am grateful to Assistant Professor, Yoshiki Higo, at the Osaka University for his helpful comments and suggestions.

I am thankful to Mr. Akiyoshi Taguchi, an undergraduate student at the Osaka University, for his valuable help in this work.

I would like to thank Ms. Naomi Yamada, a research technologist at Pennsylvania State University, for her feedback to this work.

Finally, I would like to thank all the members in Kusumoto Laboratory at the Osaka University for their helpful advices.

References

[1] M. Hirayama: “Toward Improving Quality in Embedded Software Development,” *Embedded Technology*, 2002(in Japanese.)

[2] P. Johnson, H. Kou, M. Paulding, Q. Zhang, A. Kagawa and T. Yamashita: “Improving Software Development Management through Software Project Telemetry,” *IEEE Software*, pp. 76-85, 2005.

[3] H. Aman, N. Mochiduki, H. Yamada and M. Noda: “An Estimation of Software Modification Cost Using Class Size Metric,” *JaSST’04: Japan Symposium on Software Testing*, 2004(in Japanese.)

[4] T. McCabe: “A Complexity Measure,” *IEEE Trans. on Software Engineering*, pp. 308-320, 1976.

[5] S. R. Chidamber and C. F. Kemerer: “A Metrics Suite for Object Oriented Design,” *IEEE Trans. on Software Engineering*, pp. 476-493, 1994.

[6] H. Ogasawara, K. Miura, H. Kimura, Y. Wakamatsu, K. Kiyose and T. Muroya: “Research of the Measurement Data Based on the GQM Technique for CMMI Introduction,” *Union of Japanese Scientists and Engineers*, 2004(in Japanese.)

[7] M. Genero, E. Manso, A. Visaggio, G. Canfora and M. Piattini: “Building Measure-Based Prediction Models for UML Class Diagram Maintainability,” *Empirical Software Engineering*, pp. 517-549, 2007.

[8] S. S. Chawathe and H. Garcia-Molina: “Meaningful Change Detection in Structured Data,” *Proc. of SIGMOD’ 97*, pp. 26-37. 1997.

[9] UML Specifications, http://www.omg.org/spec/UML/

[10] A. Albrecht: “Measuring Application Development Productivity,” *IBM Application Development Symp*, pp.83-92, 1979.

[11] S. Furey: “Point: Why We Should Use Function Points,” *IEEE Software*, pp. 28-30, 1997.

[12] B. Kitchenham: “Counterpoint: The Problem with Function Points,” *IEEE Software*, pp. 29-31, 1997.

[13] B. Korte and J. Vygen: “Combinatorial Optimization,” *Springer Japan*, 2009.

[14] H. W. Kuhn: “The Hungarian Method for The Assignment Problem,”*Naval Research Logistics Quarterly*, pp. 83-97, 1955.

[15] ITSpiral, http://it-spiral.ist.osaka-u.ac.jp/

[16] IPA/SEC: “Software Development Data,” *Nikkei Business Publications, Inc.*, 2008(in Japanese.)

[17] G. C. Murphy, D. Notkin and K. J. Sullivan: “Software Reflexion Models: Bridgeing the Gap between Design and Implementation,” *IEEE Trans. on Software Engineering*, pp. 364-380, 2001.

[18] S. Voigt, J. Bohnet and J. Dollner: “Object Aware Execution Trace Exploration,” *IEEE Int’l. Conf. on Software Maintenance*, 2009

[19] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams and M. Zelkowitz: “Empirical Findings in Agile Methods,” *Extreme Programming and Agile Methods XP/Agile Universe*, pp. 81-92, 2002.

[20] MASU, http://sourceforge.net/projects/masu

[21] A. Taguchi: “Applications for Modifications of Class Diagrams Based on Version Change Information,” Bachelor Thesis, *School of Engineering Science, Osaka University*, 2011(in Japanese.)

[22] D. Ohst, M. Welle and U. Kelter: “Differences between Versions of UML Diagrams,” *ESEC/FSE’ 03*, 2003.

[23] M. Girschick: “Difference Detection and Visualization in UML Class Diagrams,” Technical Report, *UD Darmstadt*, 2006.

[24] C. Treude, S. Berlik, S. Wenzel and U. Kelter: “Difference Computation of Large Models,” *Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGMOD Symposium on the Foundations of Software Engineering*, pp. 295-304, 2007.

[25] A. Henrich, H. Six and F. Hagen: “The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects,” *Proc. of the 15th Intl. Conf. on Very Large Data Bases*, pp. 45-53, 1989.

[26] U. Kelter, J. Wehren and J. Niere: “A Generic Difference Algorithm for UML Models,” *Proc. of the Software Engineering*, 2005.

[27] B. Fluri, M. Wursch, M. Pinzger and H. C. Gall: “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction,” *IEEE Trans. on Software Engineering*, pp. 725-743, 2007.

[28] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina and J. Widom: “Change Detection in Hierarchically Structured Information,” *Proc. of ACM SIGMOD Int’l Conf. Management of Data*, pp. 493-504, 1996.

[29] B. Fluri and H. C. Gall: “Classifying Change Types for Qualifying Change Couplings,” *Proc. of Int’l Conf. Program Comprehension*, pp. 35-45, 2006.

[30] S. Melnik, H. Garcia-Molina and E. Rahm: “Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching,” *Proc. of 18th Int’l Conf. on Data Engineering*, 2002.

A Calculation Cost

Suppose p of nodes are completely the same i.e. the nodes have the same inner information and the same parent between T_1 and T_2, applying the method described in Section 3.4, the constructed graph is like Figure 9. In this graph, the number of edges are $\{N(T_1) + N(T_2)\}^2$, however, applying the method described in Section 4.2, the constructed graph is like Figure 19. This graph is constructed first generates edges between completely the same p nodes, and the next, constructs complete bipartite graph by the method in Section 3.4 for other nodes. Therefore the number of edges are $p + \{(N(T_1) - p) + (N(T_2) - p)\}^2$. Define $N(T_1) + N(T_2) = A$, the number of edges created by the method are

$$A^2 \tag{4}$$

and

$$A^2 - 4pA + 4p^2 + p \tag{5}$$

, respectively. The proposed method targets for two versions of class diagrams in the same project. Therefore, the difference between them are not so high. In the following, we classify input files as (1) large class diagrams, (2) small class diagrams and discuss individually.

A.1 Large class diagrams

In this case, the target class diagrams include many classes. Thus we can ignore the difference between the number of classes in each class diagram. Let the number of classes in each class diagrams be n, and the modified classes be a constant number $q(\ll n)$. The reason why the number of modified classes are a constant is based on the hypothesis that the number of modified classes in the same term is constant. For example, the class diagram which consists of 1000 classes is modified for 10 classes in a day. What if the class diagram consists of 100000 classes. Does the number of classes which will be modified in one day scale the size of the class diagram i.e. 1000? Maybe not. The number of modified classes in the same term is not affected by the size of the class diagram.

Keep in mind $p = n - q$, the number of edges in each bipartite graph are

$$4n^2 \tag{6}$$

and

$$4q^2 + n - q \tag{7}$$

, respectively. The dimension of variable n in Equation 7 is 1. Thus the number of edges is $O(n^2)$ and $O(n)$, respectively.

The number of reduced edges are

$$4n^2 - 4q^2 - n + q \tag{8}$$

A.2 Small class diagrams

In this case, the number of generated edges is $O(n^2)$, however, we do not need to discuss the number of reduced edges rigorously, for the small class diagrams the number of edges generated in constructing bipartite graph is small and calculation cost is not so high. Note that the number of edges with optimized method is always lower than the original one.