社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

改版履歴情報を用いたクラス図の変更量の計測の試み

山田　慎也† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{s-yamada,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発を効果的に管理するためには，開発工程の進捗の把握や遅延に対する対策を効率よく行うことが必要である．下流工程における進捗管理では，ソースコードやその更新履歴を対象とした様々なメトリクスが利用されているが，設計工程では有効なメトリクスが存在せず，進捗の把握が一般に難しい．本稿では，設計書のうちクラス図を対象として，一方のクラス図から他方のクラス図に変更するために必要な変更作業量と関連性が高いクラス図変更量の計測方法を提案する．クラス図変更量の計測では，まずクラス図におけるクラスの汎化関係および内部情報から木を構築し，次に，一方の木から他方の木に変更するために必要な編集操作列を算出する．最後に編集操作列の各操作に対して重みをつけて総和をとり，クラス図変更量を得る．実際のソフトウェア開発において作成されたクラス図を対象にクラス図変更量の計測を行ったところ，得られたクラス図変更量は，クラス図変更作業量と相関を持つ可能性があることが確認できた．

キーワード　プロジェクト管理，設計工程，クラス図，変更量，メトリクス

Toward Measuring Modifications of Class Diagram Based on Version Change Information

Shinya YAMADA†, Yoshiki HIGO†, and Shinji KUSUMOTO†

† Guraduate School of Information Science and Technology, Osaka University
E-mail: †{s-yamada,higo,kusumoto}@ist.osaka-u.ac.jp

Abstract　In order to manage software development projects effectively, the managers should grasp the actual progress information of the developments 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 codes 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 technical report, we propose a measurement process on the modifications of class diagrams which would reflect the effort to change one class diagram to another. To measure the modifications of class diagrams, at first, we construct a tree based on the generalizations and the inner information of the classes in the class diagram. Next, we obtain the sequence of editing operations to change one tree to another. Then, each edit operation is weighted based on the assigned cost . Finally, by accumulating the costs, we obtain the modifications of the class diagrams. We have measured the proposed modifications from the class diagrams in the actual software development. As the results, we confirmed the modifications of class diagrams reflect the effort to change them.

Key words　Project Management, Design Process, Class Diagram, Modification, Metric

1. ま え が き

ソフトウェア開発を効果的に管理するためには，開発工程の進捗の把握や遅延に対する対策を効率よく行うことが必要である．

下流工程における進捗管理では，ソースコードやその更新履歴を対象とした様々なメトリクスが利用されている．設計工程においても，“システム要件の変更に要する工数の予定および実績” [1] 等のメトリクスを利用した進捗管理が可能であるが，このメトリクスの計測を行うためには，開発者がシステム要件の変更に要した時間を計測しなければならない．一般に，開発者は，自身に与えられた仕事に注力するだけではなく，他の開発者との打ち合わせ等に時間を割かれるため，作業時間の計測は容易ではない．

この問題は，システム要件の変更により変更される前の設計書と，変更後の設計書から，変更作業工数の予測が可能になれば解決する．それは，システム要件の変更に要する工数が，変更前設計書と変更後設計書を入力として計測可能な値となり，開発者がシステム要件の変更に要した時間の計測が不要となるからである．

ソフトウェアシステムの設計において用いられる記述言語として Unified Modeling Language(UML) がある．UML では，ソフトウェア設計書の用途に応じて様々な図の記述が可能である．その中でも重要な役割を果たす図としてクラス図がある．本研究では，設計書からの変更時間計測を行うための第一歩として，クラス図からの変更時間計測を行うことを目指す．

クラス図においては，これまで，文献 [2] に挙げられているようにクラス図中のクラス数，クラス内の属性数，操作数というようなメトリクスが計測されてきた．しかし，これらはクラス図の規模をあらわすメトリクスであるため，クラス図変更の作業工数を予測するために利用するのは難しい．そこで，本稿では，一方のクラス図から他方のクラス図に変更するために必要な作業量を, 変更に関連したクラス図におけるクラス間の汎化関係，クラスの名称，属性数，操作数を利用し，間接的に評価するためのクラス図変更量の提案とその計測システムの実装について述べる．また，計測システムを実際の開発プロジェクトにおいて作成された複数バージョンのクラス図に適用し，変更量の計測を行った．その結果，クラス図変更量は設計書の変更作業量と相関関係を持つ可能性があること，計測システムの実行時間は数秒程度であり，実際の開発現場において必要となる計測コストは十分に小さいことが確認できた．

以降，2. では関連研究について紹介し，3. でクラス図変更量の提案を行う．4. でクラス図変更量計測システムの構成および実装上の工夫点を述べ，5. で計測システムの適用実験について述べる．6. にてクラス図変更量の利用例について述べ，7. でまとめと今後の課題を述べる．

2. 関 連 研 究

クラス図に対して変更量を計測しようという試みは，著者らの知る限りこれまで行われていない．しかし，ソフトウェアのソースコードに対する変更量の計測は広く行われている．文献 [3] では，UNIX の diff コマンドを用いて，ソースコード行の追加・削除・変更箇所数を用いたソフトウェア変更量を計測する例が示されており，ソフトウェア変更量の値を用いてのちに大きな変更が加えられることになるクラスとそうでないクラスを判別するための方法を提案している．

オブジェクト指向プログラミング言語におけるさまざまな関係を用いて算出するメトリクスとして CK メトリクス [4] がある．CK メトリクスは，あるクラスの複雑度を数値化したものであり，今回の試みで計測する変更量とは値の種類が異なる．

文献 [5], [6] では主に，ある時点でのクラス図と他の時点でのクラス図を比較しその違いを分かりやすくユーザに表示するという研究を行っている．今回の我々の試みでは，クラス図間の違いを変更量として数値化することが目的であり，最終的な出力は異なる．しかし，二つのクラス図を入力とする点や，クラス図間の違いを特定する点など，類似点が多い．

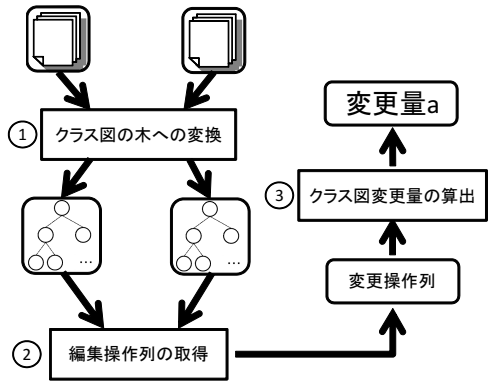

図 1　クラス図変更量計測の流れ

3. クラス図変更量計測の概要

本節では，まず 3.1 にて，計測を行うクラス図変更量の概要を述べる．その後，3.2 においてクラス図変更量計測の手順を説明し，3.3 から 3.5 にて変更量計測の詳細を説明する．

3.1 クラス図変更量の概要

今回計測を行うクラス図変更量は，システム開発の設計工程で行われる，クラス図変更に要する作業量を間接的に評価するためのものである．

本試みではまず，あるクラス図からもう一方のクラス図に変更するために必要な編集操作列を算出する．得られた編集操作列に対して，各変更操作に応じた重みを掛け合わせたのちに総和を取ることでクラス図変更量を計測する．

3.2 変更量計測手順

クラス図変更量の計測は，次の Step1 から Step3 で行う．

Step1:　計測対象となる 2 つのクラス図を木構造に変換する．

Step2:　二つの木の一方からもう一方に変換するために必要な編集操作列を取得する．

Step3:　得られた編集操作列に対し，操作の種類に応じた重みを掛け合わせ，総和をとることでクラス図変更量を算出する．

図 1 に，計測の概要を示す．図中の角のある四角で囲まれた部分は処理をあらわし，角が丸い四角で囲まれた部分はデータをあらわす．

以下 3.3 から 3.5 でこれらの手順を詳しく説明する．

3.3 クラス図の木への変換

クラス図からの木への変換において，木のノードとなる要素は，クラス図におけるクラスおよびインターフェースである．また，ノード内部で保持する情報は，クラス名，属性数，操作数である．エッジに変換される要素は，クラスの汎化関係である．その他の要素および関係，たとえば関連や集約などは変換を行わない．つまり，これらの有無は変更量計測時に値に反映されない．クラス図において，汎化元要素を持たないクラスは，木への変換時に新たに追加される ROOT ノードに連結する．

クラス図を木に変換する例を図 2 に示す．この例では，クラス「ユーザ」,「学生」,「教務」，およびインターフェース「事務」

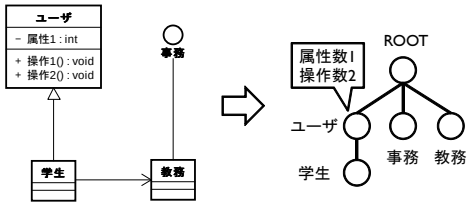

図 2　クラス図の木への変換

がノードに変換されている．また,「学生」には汎化元クラス「ユーザ」が存在するため，木構造では,「ユーザ」から「学生」に対してエッジが生成されている．その他のノードには汎化元クラスが存在しないため，新たに追加された「ROOT」ノードに対してエッジが生成されている．また，クラス「ユーザ」は，属性を 1 つと操作を 2 つ持つため，ノード「ユーザ」は内部情報として属性数 1 および操作数 2 という情報を保持している．

3.4　編集操作列の取得

ここでは，二つの木を入力として，一方の木から他方の木に変換するための編集操作列を取得する方法について述べる．編集操作列の取得は，MHDIFF アルゴリズム [7] を参考とし，クラス図変更量計測に合わせて変更を加えたものであり，基本的な手順は MHDIFF を踏襲している．

3.4.1 ではまず，編集操作列を構成する編集操作の種類を紹介する．その後完全 2 部グラフの構築，不要エッジの除去，2 部グラフの重み付きマッチング，編集操作列の決定という 4 つの手順の詳細を説明する．

3.4.1　編集操作の種類

編集操作列は 5 種類の編集操作から構成される．

以下に編集操作の種類と，それぞれがクラス図編集においてどのような操作となるかを説明する．

a）NIL

NIL は，クラス図編集において該当クラスになにも変更が加えられていないことをあらわす．

b）INS

INS は，クラス図編集においてクラスを追加するという操作に相当する．図 3 に，*INS* 操作によりクラス図上で起こりうる変化の例 (A，B，C，D の 4 例) を示す．この操作は，図中の C のように，あるクラス c と，c の汎化先クラス $Sub(c)$ の間に新たにクラスを挿入するという操作も含む．これに伴い，$Sub(c)$ の汎化元が変化するが，その変化コストも *INS* コストに含まれているものとする．

c）DEL

DEL は，クラス図編集においてクラスを削除するという操作に相当する．

d）MOV

MOV は，クラス図編集においてあるクラスの移動に相当する．この移動クラス c の汎化先クラス $Sub(c)$ は，*MOV* により移動するクラス c に付随して移動する．つまり，c の移動に伴い c と $Sub(c)$ の汎化関係は変化しない．

e）UPD

UPD は，クラス図編集においてクラスの内部情報の変更に

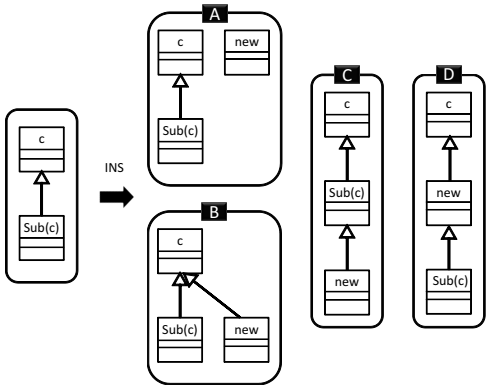

図 3　INS 操作のバリエーション

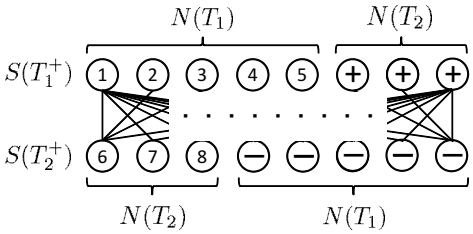

図 4　完全 2 部グラフ

相当する (内部情報は 3.3 を参照)．

f）MOV.UPD

MOV.UPD は，クラス図編集において，あるクラスが移動したのちに内部情報が変更されたという操作に相当する．

3.4.2　完全 2 部グラフの構築

一つ目のクラス図から生成された木 T_1 と，二つ目のクラス図から生成された木 T_2，および特別なノードを用いて完全 2 部グラフを構築する．以降，木 T のノード数を $N(T)$ と表記し，木 T に含まれるノードの集合を $S(T)$ と表記する．

$S(T_1)$ に対して，$N(T_2)$ 個の特殊なノード $\oplus$ を加えた集合を $S(T_1^+)$，$S(T_2)$ に対して，$N(T_1)$ 個の特殊なノード $\ominus$ を加えた集合を $S(T_2^+)$ とする．まず，$S(T_1^+)$ および $S(T_2^+)$ から 2 部グラフ B を作成する．さらに，$\forall m \in S(T_1^+), \forall n \in S(T_2^+)$ について，エッジ $e = [m, n]$ を生成したものが完全 2 部グラフとなる．

ここで構築された完全 2 部グラフは図 4 のようになる．

3.4.3　エッジへのコスト割り当て

2 部グラフのノード間に存在するエッジには，コストの割り当てが行われる．このコストは，エッジが存在するノード間の違いの大きさを表す値である．つまり，あるノード m とあるノード n の違いが大きければこのノード間に存在するエッジ $e = [m, n]$ に割り当てられるコストは大きくなる．あるノード $m \notin \oplus$ とあるノード $n \notin \ominus$ 間の違い $Diff(m, n)$ の大きさは次式 (1) のように定める．

$$Diff(m, n) = LabelUPD(m, n) + InnerChange(m, n) \quad (1)$$

ここで，$LabelUPD(m, n)$，$InnerChange(m, n)$ はそれぞれ，

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

および，

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

とする．$Len(m)$ はノード m のクラス名の長さを，$AttrN(m)$ は m の属性数を，$OprN(m)$ は m の操作数を，$COMMON(m,n)$ は m のクラス名および n のクラス名に 対する Longest Common Subsequence(最長共通部分列) の長さをあらわす．m と $\ominus$ の違いの大きさ，および $\oplus$ と n の違いの大きさは自由に決定してよい．

3.4.4 不要エッジの除去

ここでは，$S(T_1^+)$ および $S(T_2^+)$ およびエッジから構成される完全 2 部グラフから，エッジの一部除去を行う手順について述べる．

エッジは，ノードとノードの対応関係をあらわすものであり，最終的には，すべてのノード間で一対一対応を得る必要がある．このグラフを 2 部グラフの最小重み完全マッチングと呼ぶ．エッジに重みをもつ完全 2 部グラフから最小重み完全マッチングを得るには 3.4.5 にて紹介するアルゴリズムを用いれば良いが，このアルゴリズムの計算量は $O(n^2logn)$ 以上であり，不要エッジの除去における計算量 $O(n^2)$ を上回るため，本小小節の手順で一部エッジを取り除く．

取り除くのは，ノード間で違いの大きい m および n の間に存在するエッジである．m，n 間の違い $Diff(m,n)$ が非常に大きければ，m，n 間のエッジを残すよりも，m を削除し，新たに n を追加するほうがコストが小さくなる．ある操作 OPR のコストを $Cost(OPR)$ であらわすとすると，この考えに基づいたエッジの削除条件は，

$$Diff(m,n) \geqq Cost(INS)+Cost(DEL)$$

となる．

3.4.5 2 部グラフの重み付きマッチング

3.4.4 の操作で，エッジに重みをもつ完全 2 部グラフからある程度エッジの除去されたグラフが得られる．ここでは，ノード間で 1 対 1 の対応をする 2 部グラフを得るための方法を述べる．

ノードに重みがある 2 部グラフにおいて，重み和最小でノードの 1 対 1 対応 (2 部グラフの最小コスト完全マッチング) を得る問題は，割当問題と呼ばれる．文献 [8] において，計算量が $O(n^2logn)$ である割当問題の解法が述べられているが，4. で述べる計測ツールにおいては，計算量が $O(n^3)$ であるハンガリアン法 [9] を用いた．

得られた 2 部グラフは図 5 のようになる．

3.4.6 編集操作列の決定

3.4.5 までの操作で，ノード間の 1 対 1 の対応が得られた．ここでは，得られた対応関係から，エッジごとの編集操作を決定し，編集操作列を得る方法について述べる．

a） NIL

NIL の決定条件は，$m \in S(T_1^+)$，$n \in S(T_2^+)$ 間にエッジが存在し，m および n の内容が同一で， かつ m の親 $p(m)$ と $p(n)$ の間にエッジがあることである．また，$\ominus$ と $\oplus$ の間にエッジが存在する場合も NIL となる．

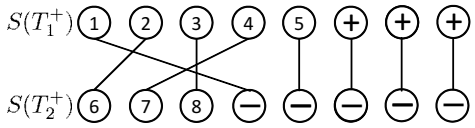

図 5　2 部グラフの最小重み完全マッチング

b） INS および DEL

INS の決定条件は，$\ominus$ でない $n \in T_2^+$ が $\oplus$ との間にエッジを持っていることである．また，DEL は，$\oplus$ でない $m \in T_1^+$ が $\ominus$ との間にエッジを持っていることである．

c） MOV

MOV の決定条件は，m，n 間にエッジが存在し，かつ m の親 $p(m)$ と n の親 $p(n)$ の間にエッジが存在しないことである．

d） UPD および MOV.UPD

UPD の決定条件は，m，n 間にエッジが存在し，かつ m の内部情報と n の内部情報が異なっていることである．MOV 条件も同時に満たしている場合は $MOV.UPD$ となる．

決定した編集操作を用いて，T_1^+ から T_2^+ を得るための編集操作列を定める．これは，決定された編集操作のうち，NIL 以外のものを取り出して並べるだけでよい．編集操作列内部の編集操作の出現順序は結果に影響を与えない．

3.5 クラス図変更量の算出

クラス図変更量は，編集操作列における各編集操作のコストの大きさの総和をとることで算出する．各編集操作のコストの大きさは自由に定めることができる．

4. 計測ツールの実装

各操作の変更量の割り当てにおいて，UPD のコスト $Cost(UPD)$ については変更内容に応じた値を算出するのが望ましいため式 (1) の $Diff$ の値を用いた．

4.1 ツールの構成

本節では，提案手法を実現するクラス図変更量計測ツールの構成について説明を行う．ツールの構成を図 6 に示す．

ツールは，まず，入力であるクラス図から，3.3 で示した情報を取得する．その後，取得した情報を用いて木を構築する．作成した二つの木を入力として，図 4 に示した完全 2 部グラフを構築する．その後，3.4.4 の方法によってエッジの除去を行う．続いて，3.4.5 の方法によって図 5 のような 2 部グラフを得る．その後，3.4.6 の方法によって編集操作列を取得し，3.5 の方法でクラス図の変更量を計測する．

図 6 で示したように，入力データからの情報取得を行う部分と木の構築を行う部分は異なるモジュールで実装している．つまり，このシステムは，容易に入力データのファイルの種類の変更が可能であるように設計されている．たとえば，入力とするデータをクラス図からソースコードに切り替え，ソースコード内に存在するクラスの情報を利用してソースコード間の変更量を計測したい場合は，ソースコードから情報を取得する部分

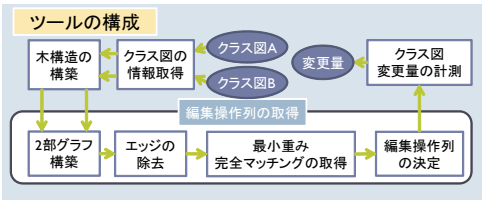

図 6　ツールの構成

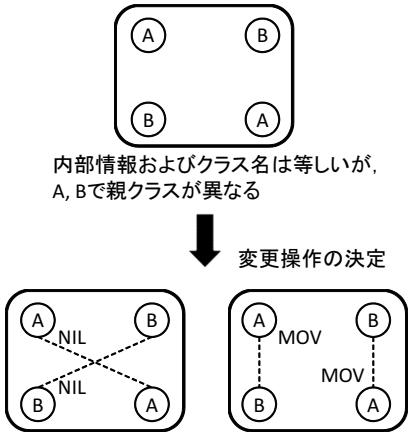

図 7　内部情報同一ノードによる完全一致判定で生じる問題

を新たに開発するだけでよい.

4.2　実装上の工夫

3. で述べた方法では，まず完全 2 部グラフを作成したのちに完全マッチングを得た．実際のクラス図の編集作業では，編集作業前後でクラスの汎化関係やクラスの内部情報の変更が行われていないクラスが多く存在すると考えられる．クラス図変更量の計測を行う際，編集作業前後で汎化関係および内部情報の変更が行われていないクラスは最終的に編集後の自身のノードと対応づけられるため，変更操作として *NIL* が割り当てられる．ゆえに，完全に一致するクラスから作成されるノードに対しては，2 部グラフ構築時にあらかじめ対応ノードとの間にエッジを作成し，それ以外のノードとの間にエッジを作成しないようにすれば，エッジ保持のための空間計算量およびノード間で 1 対 1 対応を得る際の時間計算量を小さくすることができる．また，完全に一致するノードは追加や削除が行われることはないので $\ominus$ や $\oplus$ を付け加える必要もなくなる．つまり，完全に一致するノードについてはエッジを 1 本だけにし，その他のノードにおいて 3.4.2 から 3.4.5 の操作を行えばよい.

注意したいのは，内部情報が一致するノードが複数個存在する場合である．内部情報が一致するノードは完全一致しているというように判定を行うと，内部情報が一致するノードが複数個存在する際に，一致判定を行うまでの探索順序によっては工夫前の方法と異なる判定を行う可能性がある．従来の方法と異なる一致判定が起こった場合，該当ノードには最終的な編集操作として *MOV* が割り当てられることになり，クラス図変更量が増加する．ゆえに,「完全に一致」する条件は，内部情報が同一かつ自身の先祖が同一であると定めなければならない．図 7 に，この問題が起こる例を示す.

この工夫により，$S(T_1)$ 内のノードと $S(T_2)$ 内のノードが p 個完全一致する場合に作成される 2 部グラフは図 8 のようになる．クラス図中に含まれるクラス数を n とし，変更量算出を行う二つのクラス図の間で変更されているクラス図の数をある固定値と仮定するならば，3.4.2 の方法で作成されるエッジの本数は $O(n^2)$ であったものが工夫により $O(n)$ に減少する.

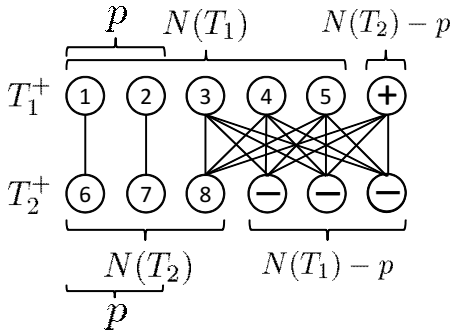

図 8　工夫後の 2 部グラフ

5.　適 用 事 例

手法の適用対象として，ITSpiral [10] で作成された実プロジェクト教材のデータを用いた．具体的には，ある大学の教務システム開発における設計フェーズで作成されたクラス図 40 個に対して，一週間間隔で 9 回変更量を計測した．クラス図は 2007 年 3 月 28 日から 2007 年 5 月 31 日までの 2 か月間編集が行われており，最終的なクラス数は 361 である．各計測日における，クラス図変更量とクラス数を図 9 に示す．対象のプロジェクトデータには，クラス図の変更に要した工数を記したデータが含まれていないため，クラス増加数とクラス図変更量の値では，どちらがクラス図変更作業量とより強い相関を持つか判断できない．しかし，4 月 6 日および 5 月 11 日のデータにおいて，クラス増加数はともに 11 であり大きなクラス数変化がないのに対し，変更量はそれぞれ 361 および 302 という値が計測されている．そこで，4 月 6 日時点でのクラス図と 4 月 13 日時点でのクラス図の内容を比較することで実際にどのような変更が行われたのかを調べたところ，21 クラスが追加され，10 クラスが削除されていることが分かった．また，5 クラスの内部情報が変更され，5 クラスの継承関係が変更されていることが分かった．このような，クラス数に反映されないクラスの変更が行われた場合には，クラス数はクラス図変更作業量を正確にあらわさない反面，本稿で提案するクラス図変更量ではそのような変更が反映されている．ゆえに，クラスの追加や削除，およびクラスの内部情報の編集が同時に行われた場合は，クラスの増加数ではなく，クラス図変更量の方がクラス図変更作業量をより正確に評価できると考えられる.

また，CPU:Xeon E5405 2.00GHz，メモリ:8GB 環境において実行した際の実行時間は表 1 のようになった．表中で，計測日は計測対象の二つのクラス図のうち古い方のクラス図の作成日をあらわし，情報取得はクラス図から情報を取得する際にかかった時間，変更量計測は，クラス図からの情報を用いて変更量計測をする際にかかった時間をあらわす．計測システムの実装から，情報取得に必要な時間は主にクラス数に依存し，変更

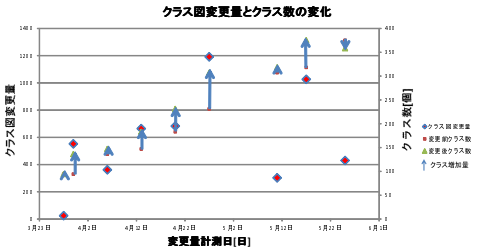

図 9　適 用 結 果

表 1　計測の実行時間

計測日 [日]	情報取得 [秒]	変更量計測 [秒]	全体実行時間 [秒]
3 月 28 日	3.1	0.6	3.7
3 月 30 日	2.0	1.2	3.2
4 月 6 日	4.2	0.7	4.9
4 月 13 日	4.2	0.7	4.9
4 月 20 日	3.3	0.9	4.2
4 月 27 日	2.9	1.4	4.3
5 月 11 日	3.2	1.4	4.6
5 月 17 日	3.3	2.6	5.9
5 月 25 日	3.7	1.4	5.1

量計測に必要な時間は主にクラス図間の完全一致しないクラスの数に依存すると考えられる．

いずれの計測においても全体の実行時間は 10 秒以内であり，クラス図変更量の計測は開発現場において大きな負担にならないと考えられる．また，計測対象のクラス図中に含まれるクラス数は，3 月 28 日の時点と比較し，5 月 25 日の時点では 4 倍になっているが，計測に要した時間は 2 倍弱であることから，本稿で提案するクラス図変更量計測の方法はスケーラビリティを持つと考えられる．

6. 利　用　例

本手法は，さまざまな場面での利用が考えられる．たとえば，ソフトウェア開発において，ある程度ソフトウェアに変更が生じた際に顧客との協議を行うように決めている場合，その協議を行うタイミングの決定基準として本試みで計測されるメトリクスが利用可能である．期間を基準として協議をするというように定めると，ある期間であまり設計書の変更が行われていない場合でも協議を行うことになる．それとは異なり，設計書に対するクラス図変更量計測を定期的に行い，ある一定の値よりも大きな変更量が計測された場合は顧客との話し合いをするということを定めておくことで，設計書の変更の大きさを基準とした協議が可能になる．

また，ソースコードから生成したクラス図と，設計段階で作成したクラス図との間でクラス図変更量計測を行うことで，設計と実装との間の乖離の大きさを知ることもできる．この情報は，特に開発プロジェクトを管理する人間にとって有用なものとなる．設計書と実装の乖離は，良い影響を与えないと考えられる [11] が，実装後（あるいは，実装途中）に，設計書とソースコードの間でクラス図変更量を計測し，乖離がある場合は，設計書に修正を加えることで，プロダクト間の一貫性を保つことが可能になる．

7. あ と が き

本稿では，クラスの汎化関係を利用した設計書間の変更量メトリクス計測方法の提案を行った．ソフトウェア設計書間での変更量計測は類似する手法の提案がこれまで行われていない．しかし，このメトリクスは顧客との協議における利用や，設計書と実装との乖離を防ぐ目的での利用などといった，さまざまな場面での利用が考えられる．

適用の事例として，大学の教務システムの開発過程で得られた設計書を対象に一週間単位でのクラス図変更量を計測し，結果を示した．また，処理にかかった時間を示した．その結果，計測を行ったクラス図変更量は，クラス図の変更作業量と相関を持つ値である可能性があることが分かった．また，計測は短時間で終わるため，実際のプロジェクト管理に利用する際に開発者の負担は小さいことが分かった．

今後の課題としては，(1) 変更量計測における挿入，削除の扱いの改良，(2) 計測システムの高速化，(3) 大規模な変更が生じているクラス図間での変更量計測の効率化，が考えられる．

謝辞　本研究は一部，日本学術振興会科学研究費補助金基盤研究（A）（課題番号: 21240002），基盤研究（C）（20500033）の助成を得た．また一部，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われた．

文　　献

[1] 小笠原秀人, 三浦邦彦, 木村初夫, 若松祐, 清瀬勝美, 室谷隆: “CMMI 導入の為の GQM 手法による測定データの研究”, ソフトウェア品質管理研究会第 1 分科会成果報告書, 2004.

[2] M. Genero, M.E. Manso, A. Visaggio, G. Canfora, and M. Piattini: “Building measure-based prediction models for UML class diagram maintainability”, *Empirical Software Engineering*, vol.12, pp.517-549, 2007.

[3] 阿萬裕久, 望月尚美, 山田宏之, 野田松太郎: “クラスサイズメトリクスを用いたソフトウェア変更量予測に関する考察”, ソフトウェアテストシンポジウム 2004 予稿集, pp.132-136, 2004.

[4] S.R. Chidamber, and C.F. Kemerer: “A Metrics Suite for Object Oriented Design”, *Trans. on Software Engineering*, vol.20, no.6, pp.476-493, 1994.

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

[6] D. Ohst, M. Welle, and U. Kelter: “Differences between Versions of UML Diagrams”, *Proc. of ESEC/FSE'03*, pp.227-236, 2003.

[7] S.S. Chawathe, and H. Garcia-Molina: “Meaningful Change Detection in Structured Data”, *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp.26-37, 1997.

[8] B. Korte, J. Vygen: “組合せ最適化”, 浅野孝夫, 浅野泰仁, 小野孝男, 平田富夫 (訳), シュプリンガー・ジャパン, 2009.

[9] H.W. Kuhn: “The Hungarian method for the assignment problem”, *Naval Research Logistics Quarterly*, vol.2, pp.83-97, 1955.

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

[11] G.C. Murphy, D. Notkin, and K.J. Sullivan: “Software Reflexion Models: Bridging the Gap between Design and Implementation”, *IEEE Trans. on Software Engineering*, vol.27, pp.364-380, 2001.