Vol.2016-SE-191 No.23
2016/3/15

命名方法の関連性に基づく識別子名の一括変更支援

小俣 仁美[1,a)]　林 晋平[1,b)]　佐伯 元司[1,c)]

概要：プログラム全体で識別子の種類の表記方法とプログラム中の概念に対する表現を統一するためには，識別子の命名規約や使用する単語を変更する際に複数の識別子名を一括して変更する必要がある．しかし，既存の手法やツールで変更すべき全ての識別子を名前変更することはできず，開発者自身による特定では漏れが生じる．本論文では，ある名前変更操作と一括して行うべき他の識別子に対する名前変更操作を推薦する手法を提案する．実際のプログラムの改版履歴を調査したところ，型名や記号等を用いた識別子の種類の表記方法や，概念に対して使用される単語を統一するための複数の識別子に対する同時名前変更操作が頻繁に行われていた．そこで，提案手法では開発者による 1 つの名前変更操作を受け取り，その操作における命名規約や単語の変更を検出する．検出結果に基づき，同一の命名規約で表されるべき同種の識別子及び同じ単語を使用している識別子をプログラム中から探索し，入力の名前変更操作と同様の変更を行って得られた新しい名前を推薦する．識別子名の一括変更操作の実例を再現できるかを確認したところ，平均再現率 0.75 で実際の名前変更履歴中の同時変更を推薦できた．

1. はじめに

プログラムの理解において，識別子の名前は重要な役割を果たす [1]．識別子はプログラム全体の約 70%を占めると言われており [2]，プログラム理解のために開発者は識別子に適切な名前をつける必要がある．また，ソフトウェア開発ではプロジェクトごとに命名規約が定められ，識別子の表記方法の統一が図られている．例えば，識別子の型の明示のためにハンガリアン記法が開発されており [3]，広く利用されている．識別子に適切な名前がついていれば，開発者は識別子の意図や動作を推測することができる [4]．

識別子を定義・命名した後も，開発者は識別子の名前をより良くするための変更を行う [1]．例えば，識別子の名前変更を自動的に行うリファクタリング技術 [5] やそのツールが存在する．名前変更は最も使われているリファクタリング操作であり [6], [7]，広く開発者に利用されている．名前変更のリファクタリングツールを用いることにより，ひとつの識別子の出現箇所を一括で編集できる．

しかし，既存のリファクタリングツールで名前変更できる識別子の数は限られており，類似の識別子全てを一括で名前変更することはできない．識別子名における表現や表記方法の変更の影響が複数の識別子にわたる場合，開発者は複数回リファクタリングツールを使用しなければならない．しかし，開発者のプログラムに対する理解が不十分だったり，不注意があったりすると，リファクタリングを行う箇所の全てを見つけられないことがある [8]．名前変更に漏れがあり，古い識別子名と新しい識別子名が混在してしまうと，プログラム中に同じ概念に対する表現が複数存在してしまい，プログラムの理解が難しくなる．

本論文では，ある名前変更操作と一括して行うべき他の識別子に対する名前変更操作を推薦する手法を提案する．本論文の主要な貢献は以下の通りである．

- **名前変更の実例に対する調査**．名前変更特定手法 REPENT [9] が提供する名前変更データセットに基づき，いくつかの種類の同時名前変更がどれくらい行われているかを調査した．その結果，最大で 268 の識別子にわたり，多数の同時名前変更が起こっていることがわかった．
- **同時名前変更推薦手法の提案および評価**．提案手法では，開発者による 1 つの名前変更操作を受け取り，その操作における命名規約や単語の変更を検出する．検出結果に基づき，同一の命名規約で表されるべき同種の識別子及び同じ単語を使用している識別子をプログラム中から探索し，新しい名前を推薦する．前述の名前変更履歴に基づく評価により，平均再現率 0.75 で名前変更履歴中の同時変更を推薦できた．

本論文の構成を以下に示す．2 章で同時名前変更の例をあげ，それらが既存ツールで解決できないことを説明する．

[1] 東京工業大学
Tokyo Institute of Technology
[a)] hkomata@se.cs.titech.ac.jp
[b)] hayashi@se.cs.titech.ac.jp
[c)] saeki@se.cs.titech.ac.jp

3 章で現実のソフトウェアで行われた名前変更の履歴を解析し，その傾向を述べる．4 章では一括名前変更を行うにあたっての問題点とそれに対する解決の方針を示す．5 章では提案手法，6 章では提案手法を実現したツールについて述べる．7 章では提案手法の評価を行う．最後に，8 章で本論文のまとめと今後の課題を示す．

2. 同時名前変更

名前変更を行う際，同じ概念に対する表現の揺れを生じないよう識別子に名前を与える必要がある．同じ概念に対して複数の表現が存在することはソフトウェアに対する誤った理解に繋がるためである [2]．

名前変更の操作は連続して行われることが多い．例えば名前変更対象の識別子に関連性のある他の識別子が存在する場合，それを含めた複数の識別子を名前変更しなければならない [10]．実際，名前変更は連なって行われることが多いことが報告されている [11]．

図 1 は，JBoss Application Server[*1] のリビジョン 5447 で適用された変更差分の一部である．ソースコード中の赤色の行が削除された箇所，緑色の行が追加された行を表す．図 1(a) では，インタフェース *TimeOutNotifier* が *TimeOutHelper* に名前変更されている．図 1(b) は同リビジョンにおける他の変更箇所であり，同様に，クラス *IntervalSnapshotStrategy.TimeOutNotifier* が *IntervalSnapshotStrategy.TimeOutHelper* に，クラス *MyTimeOutNotifier* が *MyTimeOutHelper* に，フィールド *_notifier* が *_helper* に，引数 *notifier* が *helper* に名前変更されている．すなわち，リビジョン 5446 まで *Notifier* と表現されていた概念がリビジョン 5447 以降では *Helper* と表現されるように変更されており，この変更が複数の識別子名に影響している．

開発者は，単一の概念に複数の表現が存在しないよう名前変更を行う必要がある．しかし既存のリファクタリングツールでは，関連する全ての識別子を一括で名前変更することはできない．図 1 の例では，インタフェース *TimeOutNotifier* で使用されている単語を *Notifier* から *Helper* に変更する場合，開発者は図 1 中で示されているような識別子をすべて特定し，名前変更しなければならない．ここで名前変更漏れがあると，同一の概念に *Notifier* と *Helper* の 2 つの表現が存在することになり，開発者が正しくプログラムの意味を理解できなくなる恐れが生じる．

他の同時名前変更の例として，識別子の接頭辞の修正がある．開発者はアンダースコアや英単語を接頭辞に用いて識別子の分類を識別子名に表現することがある．この場合，命名規約が変更された場合には，該当する全識別子に対し接頭辞の修正を行う必要がある．**図 2** は dnsjava[*2] のリビジョン 87 で適用された変更差分の一部である．ここでは，

```
    public void stop();

    public interface
-       TimeOutNotifier
+       TimeOutHelper
        {
```

(a) インタフェース *TimeOutNotifier* への変更

```
    class InternalSnapshotStrategy
        extends SnapshotStrategy
    {
-       class TimeOutNotifier
-           implements AbstractTimeOutManager.TimeOutNotifier
+       static class
+           TimeOutHelper
+           implements AbstractTimeOutManager.TimeOutHelper
        {

============================

-   class MyTimeOutNotifier
-       implements AbstractTimeOutManager.TimeOutNotifier
+   class MyTimeOutHelper
+       implements AbstractTimeOutManager.TimeOutHelper
    {

============================

    {
        final List
        final long
-       final TimeOutNotifier   _notifier;
-       final TimeOutTester     _tester;
+       final TimeOutHelper     _helper;

        public
-           NaiveTimeOutManager(long interval, TimeOutNotifier notifier, TimeOutTester tester)
+           NaiveTimeOutManager(long interval, TimeOutHelper helper)
            {
```

(b) その他の変更

図 1 JBoss Application Server における名前変更例

```
    import java.io.*;
    import java.util.*;
+   import DNS.utils.*;

-   public class dnsName {
+   public class Name {

    private String [] name;
    private byte labels;

-   public static dnsName root = new dnsName("");
+   public static Name root = new Name("");

    static final int MAXLABELS = 64;

    public
-   dnsName(String s, dnsName origin) {
+   Name(String s, Name origin) {
        labels = 0;
        name = new String[MAXLABELS];

============================

    import java.io.*;
    import java.lang.reflect.*;
    import java.util.*;
+   import DNS.utils.*;

-   abstract public class dnsRecord {
+   abstract public class Record {

-   dnsName name;
+   Name name;
    short type, dclass;
    int ttl;
    int oLength;

-   dnsRecord(dnsName _name, short _type, short _dclass, int _ttl) {
+   Record(Name _name, short _type, short _dclass, int _ttl) {
        name = _name;
        type = _type;
        dclass = _dclass;
```

図 2 dnsjava における名前変更例

クラス *dnsName* が *Name* に，クラス *dnsRecord* が *Record* に名前変更されている．すなわち，これまで識別子がクラスであることを表すために識別子名の先頭に接頭辞 *dns* を追加する命名規約を設けていたが，リビジョン 87 でその命名規約を廃し，クラス名から *dns* を削除している．こういった名前変更においても，変更漏れがあるとその識別子がクラスかそうでないかといった識別子の役割の理解に悪影響が生じる．

[*1] http://jbossas.jboss.org/
[*2] http://www.dnsjava.org/

表 1 名前変更履歴における $|\mathbb{U}_T|$

プログラム	1	2	3	4	5	6	7	8	9	10	11	12	13
ArgoUML	18	0	0	0	0	0	0	0	0	0	0	0	0
dnsjava	67	0	0	0	0	0	0	0	0	0	0	0	0
JBoss	631	15	5	1	1	0	1	1	0	0	0	0	1
Eclipse-JDT	172	6	1	1	0	0	0	0	0	0	0	0	0
tomcat	65	3	1	1	0	0	0	0	0	0	0	0	0

表 2 名前変更履歴における $|\mathbb{U}_{PW}|$

プログラム	1	2	3	4	5	6	7	8	9	10–99	100–999
ArgoUML	85	26	13	5	3	5	3	2	2	1	0
dnsjava	3	3	1	5	1	1	0	0	0	3	1
JBoss	533	87	27	21	7	7	2	2	1	10	0
Eclipse-JDT	697	111	26	11	8	4	3	1	0	8	0
tomcat	108	22	1	2	1	2	0	2	1	0	0

3. 同時名前変更に関する調査

本章では，どのような識別子が同時に名前変更されているかを実例から調査した結果について述べる．

3.1 使用する名前変更履歴

本調査では名前変更特定手法 REPENT [9] により取得された既存のオープンソースソフトウェアの名前変更履歴データセットを用いた．このデータセットには，実際に行われた名前変更が，メタデータとともに記録されている．例えば，名前変更 r に対して，以下の情報が取得できる．

- $r.rev$：変更が適用された（Subversion の）リビジョン番号あるいはリリースバージョン名
- $r.kind$：対象識別子の種類（クラス定義など）
- $r.type_{\{old,new\}}$：対象識別子の旧型名，新型名
- $r.name_{\{old,new\}}$：対象識別子の旧名，新名

特定の関連性をもとに同時に行われた名前変更操作の集合を**ユニット** $\mathbb{U} = \{\ldots, r, \ldots\}$ として定義する．ユニットの大きさ $|\mathbb{U}|$ は，同時に行われる名前変更の数を示している．例えば $|\mathbb{U}| = 2$ のとき，同時に行われた名前変更操作は 2 件であり，特定の名前変更と同時に行われるべきもう 1 つの名前変更があったことを表す．

以降の分析では，ユニットを特定する際，ユニットを構成する同時変更の条件として，変更の近さを用いている場合がある．近い変更のみからユニットを構成する場合，該当ユニット内の全変更が同一のリリースバージョンに属するか，あるいはユニット内の最も古い変更と最も新しい変更とのリビジョン番号差が 50 以内になるようにした．また，ユニットを数え上げる際には，他のユニットの真部分集合となるようなものは除外した．

3.2 型による同時名前変更

前章で述べたように，インタフェース名に用いられる表現の変更がフィールドや引数，ローカル変数の名前変更につながることがある．これは，型名をもとに命名された識別子名に，型名に由来する表現が含まれるからである．本稿ではこれを**型による同時名前変更**と呼び，そのユニットを，型の名前変更 r_T が変数の名前変更 r を引き起こしているとして，ユニット $\mathbb{U}_T = \{r_T, \ldots, r, \ldots\}$ として表す．ここで，$r_T.kind$ がクラス定義，インタフェース定義，列挙型定義のいずれかである場合，型の名前変更とみなす．また，r_T と r は以下の条件を満たす．

$$\{r_T.name_{old}, r_T.name_{new}\} \cap \{r.type_{old}, r.type_{new}\} \neq \emptyset \wedge \quad (1)$$

$$(deleted(r_T) \subseteq deleted(r) \wedge added(r_T) \subseteq added(r) \vee \quad (2)$$

$$acronym(r_T.name_{old}) \neq acronym(r_T.name_{new}) \wedge$$
$$acronym(r_T.name_{old}) \in w(r.name_{old}) \wedge$$
$$acronym(r_T.name_{new}) \in w(r.name_{new})) \quad (3)$$

ここで，$w(n)$ は名前 n に含まれる単語の配列を，$added(r) = w(r.name_{old}) \setminus w(r.name_{new})$ は名前変更における追加単語を，$deleted(r) = w(r.name_{new}) \setminus w(r.name_{old})$ は削除単語を，$acronym(n)$ は識別子名 n を構成する単語 $w(n)$ それぞれの頭文字からなる頭字語を表す．つまり，ある型の名前変更と，その型を利用している変数の名前変更であり，型名におこった単語変更が変数名の単語変更と短縮を考慮して対応するものを集めている．

表 1 に $\mathbb{U}_T$ の要素数を示す．ArgoUML 及び dnsjava では，型による名前変更操作が行われていなかった．Eclipse-JDT と tomcat についても，型となる識別子の名前変更と同時に変更された識別子は 3 件以内だった．JBoss では，型に関して名前変更操作があったとき，最大で 12 件の他の識別子を名前変更していた．

3.3 prefix の関連性に基づく同時名前変更

分類の等しい識別子に対し，共通の接頭辞を含めて命名するとき，本論文ではその接頭辞を *prefix* と呼ぶ．名前変更操作において，元の名前に接頭辞を含めて名前変更することを prefix の追加，元の名前に含まれていた接頭辞を含めずに名前変更することを prefix の削除，接頭辞に用いる単語や記号を変えて名前変更することを prefix の変更という．

名前変更履歴に含まれる prefix としての単語の関連性に基づくユニット $\mathbb{U}_{PW}$ の要素数を調査した．ここで，$\mathbb{U}_{PW}$ 中の任意の要素 r_i と r_j は近接しており，共通の接頭辞のみを変更しているものに限定した．**表 2** に求めた $\mathbb{U}_{PW}$ の要素数を示す．prefix としてのアンダースコアと同様，dnsjava において prefix としての単語に関して同時に 100 件以上の名前変更を行っていた．

同様に，名前変更履歴に含まれる prefix としてのアンダースコアの関連性に基づくユニット $\mathbb{U}_{PU}$ の要素数を調査した．ここで，$\mathbb{U}_{PU}$ 中の任意の要素 r_i，r_j は近接してお

表 3 名前変更履歴における $|\mathbb{U}_{PU}|$

プログラム	1	2	3	4	5	6	7	8	9	10–99	100–999
ArgoUML	29	9	6	5	6	6	3	2	2	45	0
dnsjava	1	3	0	2	0	0	0	0	0	3	1
JBoss	15	8	3	2	2	0	0	0	0	10	0
Eclipse-JDT	4	0	0	1	0	0	0	0	0	0	0
tomcat	3	0	0	0	0	0	0	0	0	0	0

表 4 名前変更履歴における $|\mathbb{U}_S|$

プログラム	1	2	3	4	5	6	7	8	9	10	11	12	13
ArgoUML	11	2	2	0	0	0	0	0	0	0	0	0	0
dnsjava	2	0	0	1	0	0	0	0	0	0	0	0	0
JBoss	94	25	7	4	4	2	0	0	0	1	1	0	1
Eclipse-JDT	100	31	23	10	11	8	4	2	2	0	1	1	0
tomcat	22	12	1	0	0	2	0	0	0	0	0	0	0

り，prefix としてのアンダースコア数の増減が等しいものに限定した．**表 3** に $\mathbb{U}_{PU}$ の要素数を示す．prefix としてのアンダースコアを同時に追加，変更や削除した件数はほとんどが 100 件未満だったが，dnsjava では最大 268 件の名前変更操作を行っていた．

3.4 単語の語幹変化に基づく同時名前変更

変更前後で使用されている単語の種類が変化せず，単語の活用形や単複数形のみが変化している名前変更操作のユニット $\mathbb{U}_S$ を求めた．ここで，$\mathbb{U}_S$ 中の任意の要素 r は同種で近接しており，$normalize(r.name_{old}) = normalize(r.name_{new})$，つまり単語の活用形や単複数形を正規化した場合に等しくなるものに限定した．$normalize(n)$ は，lemmatizer[*3] を用いて語の正規化を行う．**表 4** に $\mathbb{U}_S$ の要素数を示す．単語の形式に関する名前変更操作は JBoss 及び Eclipse-JDT で比較的多く行われていた．また，同時に行った名前変更操作の件数は他の名前変更操作と比べると少なく最大でも 13 件だった．

3.5 考察

どのプログラムにおいても，同種の変更を複数の識別子名に同時に適用している例があった．複数の識別子に適用する変更の種類の傾向はプログラムによって異なっており，ArgoUML や dnsjava では prefix に関しての名前変更が，Eclipse-JDT や JBoss では識別子名で使用する単語に関しての変更が多かった．dnsjava では 2003 年に prefix としてのアンダースコアを含めた命名規約に関する議論が行われていた [*4]．プログラムの命名規約に変更があるとき，プログラム全体を名前変更しなければならないため，同時に行う名前変更操作の件数が増えると考える．

1 ユニットあたりの名前変更履歴の数は，1 件が最も多い一方で，同時に 100 件以上の名前変更操作が行われることもあった．名前変更の際に，他に同時に名前変更をしなければならない識別子がどの程度存在するかは，プログラムや名前変更操作の内容に依存するため，対象の識別子を探索する開発者にとって負担となる．

[*3] https://github.com/yohasebe/lemmatizer
[*4] http://argouml.tigris.org/ds/viewMessage.do?dsForumId=450&dsMessageId=287367

4. 一括名前変更に対するアプローチ

3 章の結果から，開発者は関連性を基に同時にいくつもの識別子を名前変更していることがわかった．既存手法ではこれら全ての識別子を見つけ出し名前変更することはできず，開発者は関連する全ての識別子を探さなければならない．

これに対して，開発者が名前変更操作を行う際，その変更の意図や詳細を読み取って同様の操作を他の識別子にも適用すべきかどうかを調べ，一括で名前変更をしなければならない他の識別子の一覧を提示することで，名前変更操作を漏れなく行える．

prefix に関わる名前変更の場合，識別子の種類に着目することにより他の識別子にも同様の変更が必要かどうか判断できる．開発者の行った名前変更がどの種類に属するかを調べ，プログラム中から他の識別子で同じ種類に属するものを特定し，同じ prefix を持つように名前変更操作を推薦すればよい．

種類ごとの表記方法に関わる名前変更の場合も同様に，識別子の種類に着目することで一括名前変更すべき他の識別子を探索・推薦することができる．変更操作前後で識別子名の命名方法の変化を調べ，同じ種類に属する他の識別子も同じ表記方法に基づく名前になるよう名前変更操作を推薦すればよい．

短縮形に関わる名前変更の場合，識別子の種類に加え，短縮形である名前から，どのような文字列に対するどのような短縮操作によって得られた名前であるかという情報を抽出した上で名前変更操作を行う必要がある．

使用する単語の派生形・単複数形に関する名前変更については，識別子中の品詞の組み合わせ及び派生形・単複数形に対する複雑な構文解析や自然言語処理が必要である．そのため，本論文での提案では取り扱わないこととする．

5. 提案手法

5.1 概要

図 3 に提案手法の概要を示す．提案手法は，対象プログラムのソースコードと開発者による名前変更操作 r を入力とし，それらから得た情報を基に新しい識別子名の一覧を作成し推薦する．まず，入力の名前変更操作の規約変更意図，単語変更意図を特定する（1，2）．また，ソースコー

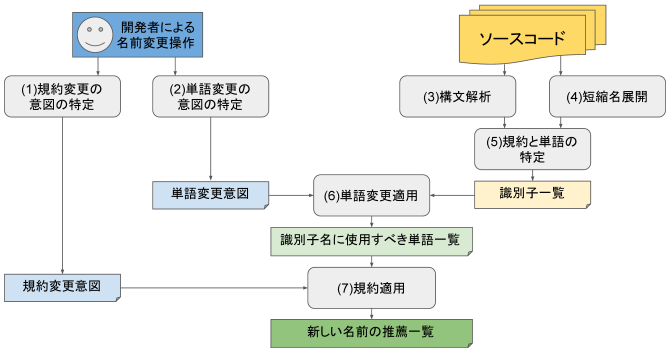

図 3　提案手法の概要

```
class Member_Data {
    ...
}

class Group_Data {
    Member_Data md;

    ... {
        Member_Data leader;
        Member_Data member0;
    }
}
```

図 4　例題プログラムのソースコード

ドを構文解析し（3），名前変更の候補となる識別子の一覧を得る．同時に，識別子中の単語の短縮名を展開し（4），短縮名と展開名の対応関係を得ておく．構文解析と短縮名の短縮方法の情報から候補の識別子に現在使用されている命名規約と単語を得る（5）．得られた識別子に対して，特定した単語変更意図に従って単語の変更を行う（6）．最後に，単語の一覧に対して特定した規約を適用し，名前変更後の識別子を得る（7）．名前変更前後の識別子名に変化があれば，それを新しい名前として推薦する．

以降，**図 4** に示す例題に基づき，名前変更がどのように推薦されるかを説明する．この例では，開発者が行った，クラス *Member_Data* の識別子名を *UserData* とする名前変更操作を入力 *r* とする．このプログラムには他にも，クラス *Group_Data* があり，これは *Member_Data* 型のフィールド *md* とローカル変数 *leader* 及び *member0* を持つ．

5.2　（1）規約変更の意図の特定

まず，開発者の行う名前変更操作における識別子の規約変更の意図を特定する．提案手法では，Java の命名規約 [12] を参考に，以下の 3 つの観点から 16 パターンの命名規約に分類する．

(1) prefix としてのアンダースコアの使用の有無
(2) 識別子の大小文字の使用方法について：UPCASE, downcase, UpperCamel, lowerCamel のいずれか
(3) 識別子中の単語をアンダースコアで区切っているか（snake_case か否か）

特定は簡易な正規表現によるマッチングにより行う．開発者の行う名前変更操作において識別子の属する命名規約のパターンが変更された場合にその識別子の種類に対する命名規約が変更されたと判断する．ただし，識別子名で使用されている文字が全て英小文字か数字でなおかつアンダースコアで区切られていない時，その識別子は lowerCamel であるとみなす．

図 4 の例では，変更前の識別子 $r.name_{old}$ は *Member_Data* であり，これは UpperCamel と snake_case を併用する Upper_Camel_Snake と特定する．変更後の識別子 $r.name_{new}$ は *UserData* であり，これは UpperCamel と特定する．その結果，Upper_Camel_Snake から UpperCamel への変更と判断する．

5.3　（2）単語変更の意図の特定

次に，名前変更においてどういった単語の変更があったかを特定する．これは，英大文字や数字，アンダースコアに着目して変更前の識別子名 $r.name_{old}$ と変更後の識別子名 $r.name_{new}$ をそれぞれ単語列に分解し，その差を Mcllroy-Hunt longest common subsequence アルゴリズム [13] により比較することで，追加，変更または削除された単語を抽出する．

図 4 の例では，変更前の単語列は [member, data]，変更後の単語列は [user, data] となり，これらを比較することにより，単語 *member* が *user* に変更されたと特定する．

5.4　（3）構文解析

入力のソースコードを構文解析し，名前変更すべき識別子の候補とその情報を特定する．ソースコードの各識別子 *i* に対して，

- $i.kind$：識別子の種類
- $i.type$：識別子の型名
- $i.name$：識別子の名前

を得る．このうち，入力された名前変更と同種（$i.kind = r.kind$），または型が関連する（$i.type = r.name_{old}$）ものを候補とする．

図 4 の例では，同種の識別子として *Group_Data* が，型が関連する識別子として *md*，*leader*，*member0* が得られる．

5.5　（4）短縮名の展開

構文解析して得られた識別子中の短縮名を，Hill らの短縮識別子展開手法 [14] を用いて展開し，短縮名と展開名との対応関係を得る．提案手法では，Hill らの手法における短縮方法の分類のうち，prefix（単語の先頭数文字を取り出したもの）あるいは single letters（単語群の頭文字をとった頭字語）と判断された識別子名について，短縮名を展開し，単語列とする．この 2 種に限定する理由は，それ以外の分類には短縮名に関する辞書が追加で必要となるためである．短縮名とみなされなければ，5.3 節同様に単語列に分解する．

図 4 の例では，*md* は短縮識別子展開手法により，single

letters によって生成された識別子名と判断される．

5.6 （5）規約と単語の特定

構文解析と短縮名の展開で得られた情報からプログラム中の各識別子で使用されている命名規約と単語を特定する．（1）と同様に識別子で使用されている命名規約を特定するほか，短縮名については（4）で得られた短縮方法についても合わせてその識別子名の命名規約とする．また，識別子名を（2）と同様に単語列に分解し，さらに（4）で得られた短縮前の単語の情報を合わせることでその識別子で使用されている単語を特定することができる．

図 4 においては，*Group_Data* に対しては命名規約 Upper_Camel_Snake と使用されている単語の列 [*group*, *data*] を，*md* については命名規約 lowerCamel と使用されている単語の列が [*member*, *data*] を，*leader* については命名規約 lowerCamel と使用されている単語の列 [*leader*] を，*member0* については命名規約 lowerCamel と使用されている単語の列 [*member*, 0] を得る．

5.7 （6）単語変更の適用

（5）で得た単語列に対して，（2）で特定した単語変更を適用する．入力された名前変更において追加，削除，変更された単語は，構文解析によって得た識別子から生成した単語列でも追加，削除，変更される．ただし，追加された単語については，挿入位置が明らかな先頭単語のみを追加する．

図 4 においては，*Group_Data* に対しては使用されている単語の列が [*group*, *data*] で単語 member を含まないため単語変更の適用をせず，*md* については使用されている単語の列が [*member*, *data*] のため変更の適用を受けて [*user*, *data*] となり，*leader* については使用されている単語の列が [*leader*] のため単語変更の適用をせず，*member0* については使用されている単語の列が [*member*, 0] のため変更の適用を受けて [*user*, 0] となる．

5.8 （7）規約適用

（6）で得られる新しい識別子で用いる単語に対し，命名規約を適用することで新たな識別子名を生成する．基本的にはもとの識別子が従う命名規約を用いるが，該当識別子が入力された名前変更の対象識別子と同種である場合，変更後の命名規約を用いる．

図 4 においては，*Group_Data* からは開発者が命名規約を変更したと考えられるクラスの識別子であるため [*group*, *data*] に新たな規約 UpperCamel を適用し *GroupData* が，*md* からは [*user*, *data*] に元の lowerCamel かつ single letters を適用し *ud* が，*leader* からは [*leader*] に元の lowerCamel を適用し *leader* が，*member0* からは [*user*, 0] に元の lowerCamel を適用し *user0* が生成される．

表 5 dnsjava リビジョン 86 への適用結果

種類		件数
履歴に含まれる推薦		183
履歴に含まれない推薦:	規約変更に基づく推薦	13
	名前変更に基づく推薦	4
	誤推薦	2
	合計	202

以上をまとめると，

$$\begin{aligned} \textit{Group\_Data} &\rightarrow [\textit{group, data}] \rightarrow [\textit{group, data}] \rightarrow \textit{GroupData},\\ \textit{md} &\rightarrow [\textit{member, data}] \rightarrow [\textit{user, data}] \rightarrow \textit{ud},\\ \textit{leader} &\rightarrow [\textit{leader}] \rightarrow [\textit{leader}] \rightarrow \textit{leader},\\ \textit{member0} &\rightarrow [\textit{member, 0}] \rightarrow [\textit{user, 0}] \rightarrow \textit{user0} \end{aligned}$$

という変更が行われる．

こうして得られた新たな識別子名のうち，元の識別子名から変更のあった *Group_Data*，*md*，*member0* を名前変更の必要性があると判断して，新しい名前とともに推薦する．

6. 支援ツール

提案手法に基づき支援ツールを実装した．ツールは対象とするプログラムのソースコードおよびそのうちの 1 つの名前変更を入力として，ソースコード中の識別子に対する新たな名前の一覧を名前変更の推薦として出力する．提案手法における（3）ソースコードの構文解析にはツール jxplatform[*5] を，（4）短縮形の展開にはツール AMAP [14] を用いた．

Java プログラムである dnsjava のリビジョン 86 に本ツールを適用した．このリビジョンでは，クラスの prefix として使用されていた文字列 *dns* を削除する名前変更が，クラスに対して 56 件，コンストラクタに対して 134 件行われている．そのうちのひとつである，クラス *dnsName* を *Name* に変更する名前変更操作を入力としてツールを適用したところ，新たに 202 件の名前変更が推薦された．203 件の内訳を**表 5** に示す．例えば，クラス *dnsMessage* は *Message* と変更するよう推薦されており，これはリビジョン 87 で行われていた名前変更に含まれる．前変更推薦があった．リビジョン 87 ではクラス名の prefix だった dns が削除されており，ツールによりリビジョン 86 まで使用されていた prefix としての dns の削除が推薦された．

リビジョン 87 で名前変更が行われなかったものの推薦された名前変更もあった．例えば，クラス *dnsServer* も同様に *Server* と名前変更されるべきであり，ツールにより推薦されたが，該当リビジョンでは変更されていなかった．また，クラス *hmacSigner* に対し新たな識別子名として *HmacSigner* が推薦された．これは，同種の識別子，つまりクラス名が今回の変更により lowerCamel から UpperCamel に変更されているため，この識別子もこれに従うように促した変更である．提案ツールの出力結果を既存のリファク

[*5] https://github.com/katsuhisamaruyama/jxplatform

表 6 実験結果

	プロジェクト	版	種類	名前変更操作	$\|O\|$	$\|D\|$	$\|C\|$	$\|C'\|$	P	P'	R
R1	microkernel	25938	クラス	*KernelFactory* → *KernelInitializer*	3	89	3	89	0.03	1.00	1.00
R2	jboss-jms/tests	40608	クラス	*RMIServer* → *TestServer*	2	485	2	9	0.00	0.02	1.00
R3	jboss-jms	41406	クラス	*TransactionLog* → *PersistenceManager*	13	804	13	324	0.02	0.40	1.00
R4	jboss-jms	31457	クラス	*Consumer* → *ServerConsumerDelegate*	2	340	2	3	0.02	0.01	1.00
R5	webservice/test	35471	メソッド	*_testElementTypeRoot* → *testElementTypeRoot*	2	60	2	2	0.03	0.03	1.00
R6	webservice/test	37778	メソッド	*testEchoIn* → *_testEchoIn*	26	2919	13	2230	0.00	0.76	0.50
R7	contrib/varia	41330	仮引数	*pDoItNow* → *doItNow*	11	166	11	145	0.07	0.87	1.00
R8	webservice/test	41745	フィールド	*ns1_ArrayOfPerson_TYPE_QNAME* → *ns5_ArrayOfPerson_TYPE_QNAME*	4	3785	1	762	0.00	0.20	0.25
				合計	63	8648	47	3564	0.01	0.41	0.75

タリングツールと組み合わせて名前変更することで，開発者はこういった変更漏れを防ぐことができる．

ただし，誤推薦もみられた．たとえばクラス *dns* に対しては，空文字列が推薦された．これは，このクラス名における単語が prefix であるとみなされた結果による．空文字列は識別子として適当でないため，この推薦を行うべきではない．

7. 評価

7.1 設計

対象ソフトウェアの変更履歴に基づいて，実装したツールが推薦する名前変更の精度を評価した．入力するプログラムとして JBoss Application Server の各プロジェクトを用いた．これは，3 章で扱った 5 つのプログラムの中で最も多くの種類の名前変更が行われていたためである．このうち，実際に行われた 8 例の名前変更を入力として提案手法に適用した．

表 6 に，入力として用いた名前変更の所属プロジェクト，抽出元リビジョン（版）および名前変更の対象識別子を示す．R1–4 は型による名前変更が行われており，特に R3, R4 では短縮名を持つ識別子名が型名とともに変更されている．R5 と R6 は，prefix としてのアンダースコアに関する名前変更が行われている．R7 と R8 は，prefix としての単語に関する名前変更が行われている．

該当の名前変更に対して，3 章で分析したユニットのうち，該当する名前変更をもっとも古いものとして持つものを特定し，正解とした．ツールの出力結果 D と上記の正解集合 O から，正しく推薦できた名前変更 $C = |D \cap O|$ を求め，適合率 $P = |C|/|D|$ と再現率 $R = |C|/|O|$ の値を求めた．ただし，実際には行われておらず履歴に含まれなかった名前変更操作をツールが推薦した場合も考慮し，検出結果を著者らのうち 1 名が目視で精査し，妥当なものを追加した正解推薦結果 C' に基づき，修正適合率 $P' = |C'|/|D|$ も求めた．

7.2 結果

実験結果を表 6 に示す．表右側には，実験により得られた名前変更推薦の適合率（P），修正適合率（P'），再現率（R）が示されている．型による名前変更（R1–4）については，再現率が 1.00 となっており，過去に行われた名前変更操作と同様の名前変更を推薦できている．また，prefix としてのアンダースコアに関しての名前変更操作（R5，R6）については，$\mathbb{U}_{PU}$ に含まれていたアンダースコアの追加の取り消し操作を除いては，過去の名前変更操作を再現できた．一方，prefix についての名前変更について過去の名前変更操作で行われた操作のうち推薦できないものがあり，prefix に関する再現率の値が低かった．また，全体的に推薦されるべきでない名前変更が多く推薦され，適合率や修正適合率の値は低かった．

推薦の失敗もあった．例えば，R8 では，識別子の prefix である *ns1* を *ns5* に変更しており，他の識別子もこれに追従する必要がある．その結果，識別子 *ns1_myArrayOfPerson_LiteralSerializer* に対しても先頭部分を変更し，*ns5_myArrayOfPerson_LiteralSerializer* とする必要がある．ところが，入力となった識別子は提案手法が想定している命名規約に則っておらず，変更前後の命名規約が *lower_Camel_Snake* と誤検出されてしまった．この識別子は，基本的にはアンダースコアを挟んで単語を連結されているものの，すべての単語間に挟んでいるわけではない．提案手法は Oracle が定める命名規約 [12] のみを想定しており，こういった複数の規約が混在したような識別子を想定していない．その結果，命名規則に基づき，*ns_5_My_Array_Of_Person_Literal_Serializer* と誤った識別子を構成してしまった．これは，元の識別子の命名規約を最大限尊重するよう，命名規約変更の効果を局所化することにより解決できると考える．

R4 では，クラス名 *Consumer* の先頭に *Server* を，末尾に *Delegate* を追加している．提案手法は，この *Server* の追加を prefix の変更とみなしてしまい，この名前変更とは無関係のクラス *Ticket* に対しても prefix を追加した名前 *ServerTicket* を推薦してしまった．これは，識別子を構成する名前集合の変化を，識別子が意味として持つ概念の変化とみなすべきか，prefix の変化としてみなすべきかの区別が難しいことに起因している．誤りなく名前変更操作を推

薦するためには，文法や単語の意味に着目して識別子名を解析する必要がある．より正確に単語変更や規約変更の意図を行うことができれば，正確な名前変更推薦を行える．

7.3 妥当性の脅威

本実験の正解は REPENT によって明らかにされた名前変更履歴から探索や目視により作成されており，より正確に適合率と再現率を求めるためには多くの開発者が同時に名前変更すべき識別子をプログラム中から探索し正解セットを作成する必要がある．

また，本研究で使用した個々のプログラムの名前変更操作履歴はそのプログラムの命名規約や使用する単語に由来する異なった傾向を持つため，より一般的な傾向を得るためには開発・保守段階におけるプロジェクトの識別子に対する方針も踏まえた上で様々なプログラムに対する更なる検証を行う必要がある．

8. おわりに

本論文では実際の名前変更履歴を調査し，一括名前変更を行う際の問題点を明らかにした．また，命名方法に着目して一括変更に関わる問題点の一部を解決する手法を提案した．既存ツールによりプログラム中の情報を取得し，その情報を基に一括で名前変更すべき識別子と新しい名前を推薦するツールを実装した．実際のソースコードに提案手法を適用し，本提案手法により関連する名前変更を一括で行うことが可能であることを確認した．

今後の課題として以下を考えている．

識別子名の文構造の解析　本論文では主に識別子名に含まれる個別の単語に着目した．そのため識別子名の文構造や文法に依存する名前変更には対応できていない．識別子名における名詞・動詞・修飾子の使い方を解明し開発者の名前変更操作の意図を正確に読み取る必要がある．

識別子の初期化に関係する名前変更　ローカル変数である識別子の名前の由来は型だけでなく，初期化する際に使用したメソッドに由来することがある．この場合，メソッドに対する名前変更操作があれば識別子も名前変更しなければならない．識別子単体だけでなくソースコードの構成に着目し識別子同士の関連性を見抜く必要がある．

謝辞　ツールをご提供頂いた Drew 大学の Emily Hill 助教に感謝する．本研究の一部は科研費（15K15970, 15H02685）の助成を受けた．

参考文献

[1] Di Martino, S., Maggio, V. and Corazza, A.: LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations, *Proc. 28th IEEE International Conference on Software Maintenance*, pp. 233–242 (2012).

[2] Deissenboeck, F. and Pizka, M.: Concise and Consistent Naming, *Software Quality Journal*, Vol. 14, No. 3, pp. 261–282 (2006).

[3] Simonyi, C.: Hungarian Notation, available from ⟨https://msdn.microsoft.com/en-us/library/aa260976.aspx⟩ (accessed 2016–02–16).

[4] Wake, W. C.: リファクタリングワークブック設計の改善テクニックを学ぶ，株式会社アスキー (2004).

[5] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman (1999).

[6] Murphy-Hill, E., Parnin, C. and Black, A.: How We Refactor, and How We Know It, *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 5–18 (2012).

[7] Murphy, G. C., Kersten, M. and Findlater, L.: How Are Java Software Developers Using the Eclipse IDE?, *IEEE Software*, Vol. 23, No. 4, pp. 76–83 (2006).

[8] Pinto, G. H. and Kamei, F.: What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow, *Proc. 6th ACM Workshop on Refactoring Tools*, pp. 33–36 (2013).

[9] Arnaoudova, V., Eshkevari, L. M., Di Penta, M., Oliveto, R., Antoniol, G. and Guéhéneuc, Y.: REPENT: Analyzing the Nature of Identifier Renamings, *IEEE Transactions on Software Engineering*, Vol. 40, No. 5, pp. 502–532 (2014).

[10] 丸山勝久：オブジェクト指向ソフトウェア向けリファクタリングツールの開発，（オンライン），入手先 ⟨http://www.fse.cs.ritsumei.ac.jp/refactoring/rise/paper.zip⟩ (2002).

[11] Saika, T., Choi, E., Yoshida, N., Goto, A., Haruna, S. and Inoue, K.: What Kinds of Refactorings are Co-Occurred? An Analysis of Eclipse Usage Datasets, *Proc. 6th International Workshop on Empirical Software Engineering in Practice*, pp. 31–36 (2014).

[12] Sun Microsystems, Inc: Code Conventions for the Java Programming Language, available from ⟨http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html⟩ (accessed 2016–02–16).

[13] Free Software Foundation, Inc.: diff-lcs-1.2.4 Documentation, available from ⟨http://diff-lcs.rubyforge.org/⟩ (accessed 2016–02–16).

[14] Hill, E., Fry, Z. P., Boyd, H., Sridhara, G., Novikova, Y., Pollock, L. and Vijay-Shanker, K.: AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools, *Proc. 5th Working Conference on Mining Software Repositories*, pp. 79–88 (2008).