Vol.2017-SE-195 No.1
2017/3/12

識別子名一括変更支援における推薦精度の向上に向けて

梅川 尚孝[1,a)] 林 晋平[1,b)] 佐伯 元司[1,c)]

概要：プログラム中の識別子は適切に名付けられるとプログラム理解に貢献するとされている．開発中に識別子名は様々な意図によって変更が加えられるが，大規模プロジェクトでは大きな手間となる．これに対して識別子名の一括変更を行う手法が提案されているが，検出精度に課題がある．本論文では識別子名一括変更を行う手法に対して，推測する変更意図の詳細化や識別子名の安定性を考慮等の拡張を行い，推薦精度の向上を図った．またツールによる自動化を行い，オープンソースソフトウェアで実際に行われた識別子名変更を用いて従来手法との比較を行った．

1. はじめに

既存のプログラムに対しその振る舞いを変更せずに理解性等を向上させることはリファクタリング [1] と呼ばれる．リファクタリングに関する様々な書物が出版され多くの研究が行われている．中でも名前変更に関するリファクタリングは頻繁に行われている．プログラム中の識別子名の自由度は高いため，開発者は識別子がプログラム中で何の役割を果たしているのか分かるように適切な命名を行う必要がある．また，開発者は作成したプログラムに適切な名前がついてるか検査し，もしそうでないなら適切に命名し直すことでリファクタリングを行う [2]．

しかし大規模なソフトウェアでは多くの識別子が存在するため，識別子に対する大規模な仕様変更が起きたときに全ての識別子に対して名前が正しいかチェックを行い，必要に応じて一つ一つに対して変更を行う必要があるがこれは大変な作業である．そのため，一つの識別子の名前変更から変更意図を検出し，他の識別子にも同様の意図によって作成された識別子名を推薦するという名前一括変換の支援は有用である．

小俣らはソフトウェアの開発履歴から識別子の名前変更を調べた結果いくつかのパターンに大別され，同じパターンによって名前変更が同時に起きていることから識別子の一括変換が有用であることを確かめた [3]．また，一つの名前変更から名前変更の意図を抽出しその意図を他の識別子に働きかけることで，一括名前変更にかかわる問題点を一部解決する手法を提案した [3]．しかしこの手法は識別子の命名パターンの漏れによる誤推薦や本来識別子として命名できない名前を誤推薦する等の問題点があった．

そこで本論文では小俣らの手法をもとに命名パターンや意図検出の漏れを防ぐ手法を提案する．本論文の主要な貢献は以下のとおりである．

- **名前変更推薦手法の提案**．一括でパターン付けしていた大文字小文字の情報や単語間のアンダースコア等の記号の情報を細分化することで，複雑な命名規則にも対応し，単語の追加・削除・置換とアンダースコアの変化についての名前変更意図を検出した．また細分化された情報をもとに意図適用を行い，新しい識別子名を作成することで誤推薦や検出漏れを減らす．
- **提案手法と従来手法の比較評価**．実際にオープンソースソフトウェアで行われた名前変更を用いて従来手法と提案手法，二つの手法の比較を行った． その結果，適合率，再現率ともに提案手法が従来手法を上回り本手法が有用であることを示した．

本論文の構成を示す．2 章では従来手法を紹介し，その問題点について議論する．3 章では従来手法に拡張を加えた識別子一括変更の手法について述べる．4 章では 3 章で提案した手法に従い実装を行った支援ツールについて述べる．5 章では実装したツールを用いて行った実験とその結果について述べる．6 章で識別子に対してどのような研究が行われているか俯瞰する．7 章では本論文のまとめと今後の方針を示す．

2. 一括名前変更における問題点

小俣らは開発中のプロジェクトにおける識別子名の名前変更に着目し，どのような変更の同時名前変更が何件行わ

1 東京工業大学 情報理工学院
School of Computing, Tokyo Institute of Technology
a) umekawa@se.cs.titech.ac.jp
b) hayashi@se.cs.titech.ac.jp
c) saeki@se.cs.titech.ac.jp

© 2017 Information Processing Society of Japan

```
...
public class ClientSideEJBTestCase extends TestCase {
        public ClientSideEJBTestCase(···) {
            ···}
        public void testClient_accessEJB() throws Exception {
            ···}
        public void testClient_accessJSE() throws Exception {
            ···}
        private double clientSideSeverConnection(···) {
            ···}
    }
```

図 1　例題プログラムのソースコード

れているか，名前変更特定手法 REPENT [4] によって取得されたオープンソースソフトウェアの開発履歴を調査することで，開発者が同じ意図によって多数の同時名前変更を行っていることを明らかにした．また，いくつかのパターンの同時名前変更に対して一つの名前変更から変更点を検出し，それを他の識別子に適用することで新しい名前を推薦する手法を提案した．この手法では，識別子の命名パターンに関して，大文字小文字の使用に基づく UPCASE, downcase，UpperCamel，downCamel の 4 種，またそれらに対してアンダースコアを用いる snake_case に従っているかも加えて検出を行う．さらに，一つの名前変更からいくつかのパターンに従った意図検出を行い，他の識別子に対して検出した意図とのマッチングを行う．マッチした識別子に関して同様の変更を行い，特定した命名パターンに従って新しい識別子名を作成する．

この手法は同時名前変更を提案する手法として有用であるものの，現実の識別子の命名パターンにはより複雑なものもあり，誤検出が生じる．図 **1** は Jboss Application Server[*1] のリビジョン 20870 におけるソースコードに基づき作成した例である．図中の識別子 `testClient_accessEJB` は，単純な snake_case に従っておらず，部分的には camelCase に従った文字列がアンダースコアで結合されている．小俣らの手法ではこういった単純でないパターンに基づく識別子を正しく特定できず，誤検出したパターンに基づき識別子を構成するため，`test_Client_Access_EJB` と変形され，誤推薦を引き起こす．また，名前変更の意図検出では，単語の追加を特定する位置に限りがあり，先頭以外の部分への追加を検出できない．そのため，図 1 の例における識別子 `testClient_accessEJB` を `testClientSide_accessEJB` に名前変更した際，単語 `side` の追加を検出できず，識別子 `testClient_accessJSE` に対して `testClientSide_accessJSE` を推薦できない．

これらの問題点に対して，以下の改善が必要になると考える．

- より精密な追加，削除，置換意図の特定．識別子に対する単語の追加や削除，置換は，識別子中のさまざまな位置に対して起こるため，追加・削除などの変更が起きた単語の位置に関係なく検出する必要がある．
- より柔軟な命名パターンの特定．現実のソフトウェアでは，前述したように，必ずしもすべての識別子が命名規約等に正しく基づいているわけではなく，複数の規約を部分的に併用したような識別子も存在する．こういった識別子に対しても，その意図する命名パターンを保持したまま名前変更を適用する必要がある．

*1 http://jbossas.jboss.org/

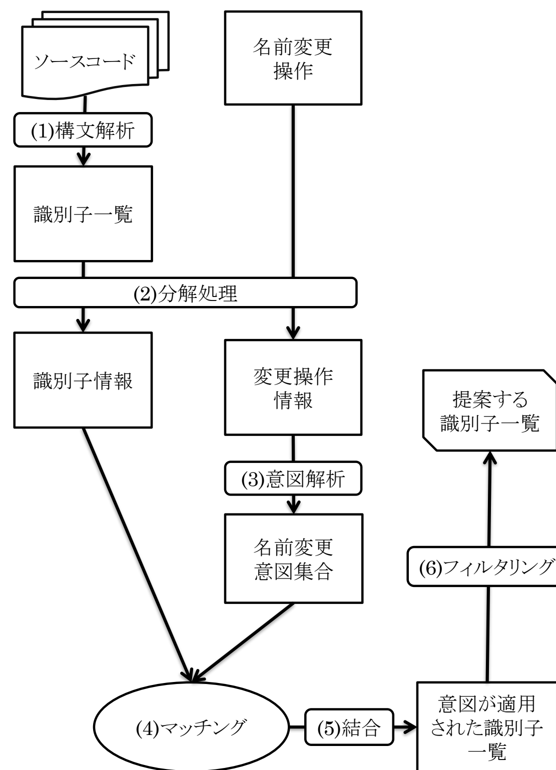

図 **2**　提案手法の概要

3. 識別子一括変更支援のためのアプローチ

3.1 概要

図 **2** に提案手法の概要を示す．開発中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，r から得た情報をもとに新しく識別子を作成し推薦する．

提案手法ではまず，ソースコードを構文解析し識別子の抽出を行う (1) ことで名前変更の候補となる識別子の一覧を得る．次に，得られた識別子および開発者による名前変更操作における変更前，変更後の識別子名に対して，複数の単語によって構成される識別子を単語群に分解する分解処理を実行する (2)．分解処理により，ソースコード中の識別子からは識別子情報が，名前変更操作からは変更操作情報を得る．さらに変更操作情報に対しては意図解析を行い，名前変更意図を推定する (3)．推定された名前変更意図と識別子情報に対してマッチングを行い (4)，意図にマッチ

© 2017 Information Processing Society of Japan

した識別子は名前変更の候補となる．候補に対して，意図に基づき適切に識別子中の単語を置換し，再度識別子として復元する結合処理を行い (5)，名前変更後の識別子を得る．最後に，フィルタリング (6) により不適なものを除いたものを開発者に推薦する識別子とする．なお，本論文では前章で述べた問題点の解決に焦点をあてており，従来手法で対処されていた省略語の考慮等については扱わない．

以降に図 1 に示す例題プログラムにおけるの 4 つの変更に基づき，名前変更がどのように推薦されるかを説明する．

- 例 1: `testClient_accessEJB` → `testClientSide_accessEJB`
- 例 2: `ClientSideEJBTestCase` → `ClientEJBTestCase`
- 例 3: `testClient_accessEJB` → `runClient_accessEJB`
- 例 4: `testClient_accessEJB` → `_testClient_accessEJB`

3.2 識別子の抽出

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

- $i.kind$: 識別子の種類 （Class, Method 等）
- $i.name$: 識別子の名前

を得る．例えば図 1 の識別子 `testClient_accessEJB` に対しては，

$$\texttt{testClient_accessEJB}.kind = \text{Method (関数)}$$

$$\texttt{testClient_accessEJB}.name = \texttt{testClient_accessEJB}$$

となる．

また同様に，入力とする名前変更についてもその種類等を特定しておく．名前変更 r は以下の情報を含む．

(1) $r.kind$: 対象識別子の種類
(2) $r.old$: 対象識別子の旧名
(3) $r.new$: 対象識別子の新名

例えば図 1 のプログラムにおいて関数 `testClient_accessEJB` を `testClientSide_accessEJB` へ名前変更を行った場合，名前変更操作 r は

- $r.kind = \text{Method}$
- $r.old =$ `testClient_accessEJB`
- $r.new =$ `testClientSide_accessEJB`

と構成される．

3.3 分解処理

抽出して得られた識別子 $i.name$ と $r.name$ を分解する．まず，アンダースコアで区切られた snake_case 記法と，大文字小文字の境目によって区切られた camelCase 記法に基づいて構成された識別子を単語単位に分割する．本手法では，単語数 n の識別子名 $i.name$ に対して

- $i.name^w = [w_0, w_1, \ldots, w_{n-1}]$: 構成される単語
- $i.name^s = [s_0, s_1, \ldots, s_n]$: 単語間の記号
- $i.name^c = [c_0, c_1, \ldots, c_{n-1}]$: 単語の形式

の 3 つの列に分割する．単語の形式は，

- num: 数字
- UP: すべて大文字
- $Uplow$: 先頭文字のみが大文字
- low: 全て小文字

の 4 種とし，全ての単語の形式を記憶した．

全ての単語の前後にアンダースコアやドルマークがないか記憶するようにすることで部分的な snake_case 等にも対応した．

例えば関数 `testClient_accessEJB` は

- $\texttt{testClient_accessEJB}.name^w = [\texttt{test}, \texttt{client}, \texttt{access}, \texttt{ejb}]$
- $\texttt{testClient_accessEJB}.name^s = [\text{“”}, \text{“”}, \text{“_”}, \text{“”}, \text{“”}]$
- $\texttt{testClient_accessEJB}.name^c = [low, Uplow, low, UP]$

となる．

以降，列 l の m 番目の要素を l_m と表記する．例えば $i.name^c$ の m 番目の要素は $i.name^c_m$ となる．また同様に $r.old$ と $r.new$ に対しても上記の分解を行っておく．

3.4 意図解析

名前変更における旧名 $r.old$ と新名 $r.new$ を比較し，それらの間の差分を特定することにより，名前変更を行った意図を推定する．意図は，大きく (1) 単語の追加，(2) 削除，(3) 置換，(4) 識別子先頭の記号の追加削除，に分けられる．特定した意図をまとめ，意図集合 U として出力する．

3.4.1 単語の追加

名前変更において，旧名には含まれない単語が新名にあり，かつ新旧名に共通する単語が含まれている場合，すなわち

$$w \in r.old^w \setminus r.new^w \wedge r.old^w \cap r.new^w \neq \emptyset$$

をみたす単語 w が存在する場合，$r.old$ に単語 w が追加されたとみなし，意図 $R = added(w, m)$ を検出する．ここで，m は単語 w の新名 $r.new$ における出現位置である．このとき，新名における追加単語の前単語 $r.new^w_{m-1}$ を変更意図の前文脈 R^{pre}，後単語 $r.new^w_{m+1}$ を後文脈 R^{post} として記録しておく．

例 1 では，名前変更前後で単語 `side` が追加されており，$\texttt{side} = \texttt{testClientSide_accessEJB}.name_2$ であるため，意図として $R = added(\texttt{side}, 2)$ が検出される．また，$R^{pre} = \texttt{client}$，$R^{post} = \texttt{access}$ が記録される．

3.4.2 単語の削除

名前変更において，新名には含まれない単語が旧名にある場合，すなわち

$$w \in r.old^w \setminus r.new^w$$

をみたす単語 w が存在する場合，$r.old$ から単語 w が削除されたとみなし，意図 $R = removed(w, m)$ を検出する．ここで，m は単語 w の旧名 $r.old$ における出現位置である．

例 2 では，名前変更前後で単語 side が削除されており，side = ClientSideEJBTestCase.$name_1^w$ であるため，意図として $R = removed(\texttt{side}, 1)$ が検出される．

3.4.3 単語の置換

特定の名前変更において，単語の追加と削除をいずれも検出した場合，それらの単語の前後の単語のいずれかが一致すれば，単語の置換が起こったものとみなす．すなわち，単語の追加 $added(t, m)$ と削除 $removed(w, n)$ が検出されたとき，

$$r.old^w_{n-1} = r.new^w_{m-1} \lor r.old^w_{n+1} = r.new^w_{m+1}$$

をみたす場合，単語 w は t に置換されたとみなし，意図 $replaced(w, t)$ を検出する．置換が検出された場合，対応する単語の追加と削除の意図はこれまでに検出した意図集合 U から削除しておく．

また，共通する単語集合がないものの新旧名の単語数が一致する，すなわち

$$r.old^w \cap r.new^w = \emptyset \land |r.old^w| = |r.new^w|$$

である場合，旧名中の各単語はそれぞれ新名中の単語に置換されたとみなし，$replaced(r.old^w_i, r.new^w_i)$ $(i = 1, \ldots, m-1)$ を得る． 一方，単語数が一致しない場合，すなわち

$$r.old^w \cap r.new^w = \emptyset \land |r.old^w| \neq |r.new^w|$$

である場合，旧名に含まれる全単語が新名に含まれる全単語に置き換わったという特別な意図 $replaced_{all}(r.old^w_{1 \cdots m-1}, r.new^w_{1 \cdots n-1})$ $(m \neq n)$ を検出する．

例 3 では $added(\texttt{test}, 0)$ と $removed(\texttt{side}, 0)$ を検出するが，testClient_accessEJB.$name_1^w$ = client = runClient_accessEJB.$name_1^w$ であるため，$replaced(\texttt{test}, \texttt{run})$ が検出される．

3.4.4 先頭記号の追加・削除

$r.old^s_0 \neq r.new^s_0$ である場合，つまり先頭の記号が変わっていた場合，これを識別子種 $r.kind$ における特別な変更意図 $prefixChanged(r.old^s_0, r.new^s_0, r.kind)$ として検出する．

例 4 では名前変更前後で識別子の先頭にアンダースコアが追加されている．変更前には先頭に記号がなかったものがアンダースコアに変更されたため，$prefixChanged(\text{``''}, \text{``_''}, \text{Method})$ を得る．

3.5 マッチング

発見した意図を識別子に対して適用する．発見した意図は単語の置換，単語の追加，単語の削除，先頭記号の追加・削除の順番で適用する．

3.5.1 単語の追加

単語の追加意図 $R = added(w, m)$ に対し，前後の文脈が識別子中の単語に合致した場合，対応する箇所へ単語 w を追加する．具体的には，識別子 i に対して，

$R^{pre} = i.name^w_n$ のとき n 番目の単語として
$R^{post} = i.name^w_n$ のとき $n + 1$ 番目の単語として

w を挿入する．ただし，i が既に単語 w を名前として含んでいる（$w \in i.name^w$）ときは追加しない．

単語 w が追加された場合，単語の形式もあわせて追加する必要がある．ここでは，

$R^{pre} = i.name^c_n$ のとき n 番目の要素として
$R^{post} = i.name^c_n$ のとき $n + 1$ 番目の要素として

$r.new^c_m$ を $i.name^c$ に挿入する．ただし，$R^{pre} = i.name^c_0$，すなわち先頭に単語を追加する場合，形式の不用意な変化を避けるために $r.new^c_m$ の挿入箇所を 1 番目とする．

さらに，単語の追加により発生する単語間の記号を考慮する必要があるため

$R^{pre} = i.name^c_n$ のとき n 番目の要素として $r.new^s_m$ を
$R^{post} = i.name^c_n$ のとき $n + 1$ 番目の要素として $r.new^s_{m+1}$ を

$i.name^s$ に挿入する．

例 1 では意図 $R = added(\texttt{side}, 1)$ が検出され，識別子 i = testClient_accessJSE に対して R^{post} = client = $i.name^w_1$ であるため， i の client と access の間に side を追加する．i' = ClientSideEJBTestCase に対しても文脈のマッチが成立するものの，単語 side がすでに含まれているため，追加されない．

3.5.2 単語の削除

単語の削除意図 $R = removed(w, n)$ に対し，単語 w が識別子中に存在した場合，該当の単語を削除する．具体的には，識別子 i に対して $i.name^w_m = w$ のとき，i の m 番目の単語を削除する．

単語の削除にともなって，単語間の記号 $i.name^s_m$ および単語の形式 $i.name^c_m$ も削除する．ただし，先頭の記号は特別な意図と関連するため，先頭単語の削除，すなわち $m = 0$ のときは，$i.name^s_0$ ではなく $i.name^s_1$ を削除する．

例 2 では $removed(\texttt{side}, 1)$ を検出しており，識別子 i = clientSideServerConnection においては $i.name^w_1$ = side であるため，単語 side が削除される．

3.5.3 単語の置換

単語の置換意図 $R = replaced(w, t)$ に対し，単語 w が識別子中に存在した場合，該当の単語を単語 t に置換する．具体的には，識別子 i に対して $i.name^w_m = w$ のとき，i の

© 2017 Information Processing Society of Japan

m 番目の単語を t に変更する．

例 3 では $replaced(\texttt{test}, \texttt{run})$ を検出している．識別子 $i = \texttt{testClient}_{\texttt{a}}\texttt{ccessJSE}$ は単語 test を含むため，これを run に置換する．

また，複数の単語の一括置換意図 $replaced_{all}(W, T)$ に対し，単語列 W が識別子中に存在した場合，該当の単語列を単語列 T に置換する．具体的には，識別子 i に対して，

$$i.name^w_{m+j} = W_j \ (\forall j \in \{0, 1, \ldots, |W| - 1\})$$

のとき，$i.name^w$ の m 番目から $m + l - 1$ 番目の要素を削除し，該当箇所に単語列 T を挿入する．同様に，単語の形式や間の記号についても，先頭要素を例外視して置換する．つまり，$i.name^c$ の $m + 1$ 番目から $m + l - 1$ 番目の要素を削除し，該当箇所に $r.new^c$ の 1 番目以降の要素を順に挿入する．また，$i.name^s$ の $m + 1$ 番目から $m + l - 1$ 番目の要素を削除し，該当箇所に $r.new^c_1 \cdots r.new^c_{n-2}$，つまりの先頭と末尾以外の要素を順に挿入する．

3.5.4 先頭記号の削除・追加

先頭記号の変更意図 $R = prefixChanged(p, p', k)$ に対し，種別 k の識別子の先頭記号が p に合致した場合，該当の記号を p' に置換する．具体的には，識別子 i が $i.name^s_0 = p \wedge r.kind = k$ をみたす場合，$i.name^s_0$ を p' に変更する．

例 4 では $prefixChanged(\text{``''}, \text{``_''}, \text{Method})$ を検出している．一方，識別子 clientSideServerConnection の先頭には記号はなく，種類が同様に Method であるため，先頭に "_" を追加する．

3.6 結合

分割された識別子情報を結合することで識別子名を新しく作成する．識別子 i 中の各単語 $i.name^w$ に対して，対応する $i.name^c$ に従った文字列表現 $i.name^{cw}$ を生成する．これに基づき，新しい識別子 $i.new$ を

$$i.name^s_0 \ i.name^{cw}_0 \ i.name^s_1 \ \cdots i.name^{cw}_{n-1} \ i.name^s_n$$

として構成する．

例 3 において識別子 $i = \texttt{testClient_accessJSE}$ は，マッチング後

$$i.name^w = [\texttt{run}, \texttt{client}, \texttt{access}, \texttt{jse}],$$
$$i.name^s = [\text{``''}, \text{``''}, \text{``_''}, \text{``''}, \text{``''}],$$
$$i.name^c = [low, Uplow, low, UP]$$

に変化しているため，ここから

$$i.name^{cw} = [\texttt{run}, \texttt{Client}, \texttt{access}, \texttt{JSE}]$$

を構成し，結合して $i.new = \texttt{runClient_accessJSE}$ を得る．

3.7 フィルタリング

意図を適用し新しく識別子名を作成した際，プログラムで用いるには不適切なものが生成される場合がある．例えば，Server1 という識別子に対して単語 server を削除する意図を適用した場合，最終的には 1 という識別子名が作成されるが，多くのプログラミング言語でこの識別子名は有効なものではない．このような，対象のプログラミング言語の規則により使用できない識別子名の提案を防ぐよう，フィルタリングを行う．本論文では Java 言語を対象とするため，Java における識別子の命名条件に基づき，以下の条件全てを満たさないものは推薦対象から除外した．

(1) 1 文字以上かつ全て半角の英数字，ドルマーク ($)，アンダースコア (_) のみで構成されている．
(2) 先頭の文字が数字でない，つまり 1 文字目がアルファベット，ドルマーク，アンダースコアのいずれかである．
(3) int，const，char などの予約語でない．

4. 支援ツール

提案手法に基づき支援ツールを実装した．ツールの入力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一つの名前変更とし，出力はソースコード中の識別子の変更前識別子名と変更後識別子名のセットとする．ソースコードから識別子を抽出するために使う構文解析には jxplatform[*2] を用いた．

5. 評価

5.1 設計

実際にソフトウェア開発において行われた名前変更履歴を用いて実装ツールが推薦する名前変更の制度を評価した．入力に JBoss Application Server の各プロジェクトと名前変更特定手法 REPENT[4] より取得された名前変更履歴データセットから，小俣らが評価に用いた 8 例（R1-8）に 4 例（R9-12）を新たに加えた 12 例を用いた．名前変更入力を従来手法 [3] と本論文の提案手法に適用した．

5.2 結果

表 **1** に入力として用いた名前変更の所属プロジェクト，抽出元リビジョンおよび名前変更の対象識別子を示す．R4，R12 で単語の追加，R7，R9 で単語の削除，R1，R2，R3，R8，R10 で単語の置換が行われ，R5，R6，R11 では先頭のアンダースコアに対する操作が行われている．正解に関しては該当する名前変更から，REPENT データセットが提供する過去の名前変更履歴のうち最も古いものを特定し，同時に変更されたものを用いた．ツールの出力結果 D と正解集合 O から正しく推薦できた名前変更 $C = D \cap O$

*2 https://github.com/katsuhisamaruyama/jxplatform

表 1 実験対象

	プロジェクト	版	種類	名前変更操作
R1	microkernel	25938	クラス	KernelFactory → KernelInitializer
R2	jboss-jms/tests	40608	クラス	RMIServer → TestServer
R3	jboss-jms	41406	クラス	TransactionLog → PersistenceManager
R4	jboss-jms	31457	クラス	Comsumer → ServerConsumerDelegate
R5	webservice/test	35471	メソッド	_testElementTypeRoot → testElementTypeRoot
R6	webserveice/test	37778	メソッド	testEchoIn → _testEchoIn
R7	contrib/varia	41330	仮引数	pDoItNow → doItNow
R8	webservice/test	41745	フィールド	ns1_ArrayOfPerson_TYPE_QNAME → ns5_ArrayOfPerson_TYPE_QNAME
R9	aop-mc-int	45155	メソッド	getAnnotationDependenciesReflect → getAnnotationDependencies
R10	Admin_Console/src	35829	クラス	DestinationServiceException → JmsServerException
R11	webservice/test	26334	メソッド	_testBase64Binary → testBase64Binary
R12	jboss-jms/tests	40608	クラス	StressQueueNotSaneConnection → StressQueueNotSameConnectionTest

を求め，適合率 $P = |C|/|D|$ と再現率 $R = |C|/|O|$ 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，検出結果から著者のうち 1 名が妥当なものを追加した正解集合 C' に基づき，修正適合率 $P' = |C'|/|D|$ も求めた．

実験結果を表 **2** に示す．

表には左から正解数 ($|O|$)，実験により得られた名前変更推薦の提案数 ($|D|$)，正解数，($|C|$)，修正正解数 ($|C'|$)，適合率 (P)，修正適合率 (P')，再現率 (R) が示されている．全体的な結果としては適合率，修正適合率，再現率，いずれの指標においても従来手法を上回る結果となった．これは部分的な snake_case に対して対応できるようになったことにより誤推薦が減少したことが大きく影響している．R3, R4 以外の再現率では 1.00 を記録し，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．R7,R11 では単語の削除が行われたが提案手法では先頭が数字であったり名前が空である等，識別子として提案してはならないものをフィルタリングにより弾いたことにより適合率と修正適合率はともに従来手法を上回った．R12 では先頭以外の場所で単語の追加が行われ，従来手法では単語が追加されたことを検出できず適切な推薦ができなかったが，提案手法では意図を検出し適切な推薦をすることができた．

ただしいくつかの例では提案手法が従来手法を下回った．R3,R4 ではそれぞれ単語の置換と単語の追加が行われ再現率が従来手法では共に 1.00 であるのに対し，提案手法では R3 が 0.54，R4 が 0.50 と従来手法より下回った．これは，従来手法では AMAP[5] を用いた短縮識別子展開を行うことで短縮された識別子名に対しても変更を加えているが，提案手法では短縮された識別子名を考慮していないことによる．従来手法では R3 において識別子中の単語 `tl` を `TransactionLog` の略語であると特定し，対応する `PersistenceManager` の略語の `pm` を推薦したが，提案手法では `tl` に対して新しい識別子名を推薦できなかった．この事に関しては提案手法にも短縮された識別子名に対して従来手法同様に AMAP や Corraza らが提案した短縮された識別子の略語元特定手法 [6] を用いることによって略語元を特定する手法を取り入れることで改善が期待される．R8 では `ns1` から `ns5` への置換と捕えるべきところを 1 から 5 への変更と捕えてしまったため，本来名前変更するべきではない `file1` という識別子名に対して `file5` を推薦してしまった．同様の誤推薦が R8 では多く見られたため修正適合率も 0.34 と従来手法の 0.20 を上回ったものの他の修正適合率と比べると低くなってしまった．これは数字を置換する際前後の単語を考慮に入れることで改善が見込まれる．また，従来手法よりは高めの数字を記録したものの適合率はかなり低い数字となってしまった．これはユーザーに一括変更を行う範囲を設定させることなどで改善が見込まれる．

5.3 妥当性の脅威

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と，目視によって作成された．より正確な適合率と再現率を求める場合，より多くの開発者が同時に名前変更をすべきである識別子を，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，実験対象が JBoss Application Server の 12 例のみであるため他のソフトウェア等の名前変更を対象とした場合正答率が大きく変動する可能性がある．この脅威に対しては実験対象を増やして更に実験を行う必要がある．

6. 関連研究

プログラム中の識別子はプログラム理解の際に重要な役割を果たしている [7], [8], [9]．Deissenboeck と Pizka は，識別子はプログラム全体のうち約 70%を占めると述べている [10]．また，Corazaa らは，開発者が開発意図やドメイン知識を考慮した適切な名前を識別子につけることにより，開発者間の情報伝達を円滑としていると述べている [6]．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

表 2 実験結果

		従来手法 [3]						提案手法					
	$\|O\|$	$\|D\|$	$\|C\|$	$\|C'\|$	P	P'	R	$\|D\|$	$\|C\|$	$\|C\|'$	P	P'	R
R1	3	89	3	89	0.03	1.00	1.00	89	3	89	0.03	1.00	1.00
R2	2	485	2	9	0.00	0.02	1.00	9	2	9	0.22	1.00	1.00
R3	13	804	13	324	0.02	0.40	1.00	307	7	307	0.02	1.00	0.54
R4	2	340	2	3	0.01	0.01	1.00	48	1	1	0.02	0.02	0.50
R5	2	60	2	2	0.03	0.03	1.00	2	2	2	1.00	1.00	1.00
R6	26	2919	13	2230	0.00	0.76	0.50	2370	26	2370	0.01	1.00	1.00
R7	11	166	11	145	0.07	0.87	1.00	147	11	147	0.07	1.00	1.00
R8	4	3785	1	762	0.00	0.20	0.25	2043	4	690	0.00	0.34	1.00
R9	5	6	5	6	0.83	1.00	1.00	6	5	6	0.83	1.00	1.00
R10	10	160	10	155	0.06	0.97	1.00	155	10	155	0.06	1.00	1.00
R11	32	125	32	41	0.26	0.33	1.00	41	32	41	0.78	1.00	1.00
R12	3	470	1	2	0.00	0.00	0.33	224	3	224	0.01	1.00	1.00
合計	113	9409	95	3786	0.01	0.40	0.84	5541	106	4041	0.02	0.74	0.94

や動作を推測することができる [11].

一貫しない識別子名はプログラム理解を妨げるため，これを是正する手法が提案されている．手法のひとつは，識別子の正規化であり，プログラム全体に対して，識別子の命名規則が一貫したものとなるよう，識別子名を変換する．Caprile と Tonella は識別子名に対して規則や辞書を用いた正規化を施し，新しい識別子名を生成する手法を提案している [8]．Lawrie らは情報検索の技術を用いて語彙の類似性を計算し，識別子中の語彙を正規化する手法を提案している [8]．これらは，プログラム中の識別子名の非一貫性の是正に有効である．

一方，開発者による名前変更もソフトウェア開発中に多発する．一貫しない名前変更は，一貫しない識別子名を生むため，これの防止も重要である．Fowler のリファクタリング・カタログ [1] には Rename Method などの名前変更リファクタリングが含まれており，これらを実現したリファクタリングツールを利用することにより，特定の識別子名を一貫して変更できる．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 [12], [13]，広く開発者に利用されている．ただし，これらのツールは関連する識別子名の一括名前変更機能を持たない．

ソフトウェア開発では，複数の名前変更が同時に起こりがちである．名前変更対象の識別子に関連性のある他の識別子がある場合，それを含めて一貫した名前変更を行う必要がある．Saika らは，開発者によって行われたリファクタリングの操作履歴を分析し，Rename の連鎖など，一部のリファクタリングが連なって行われることが多いことを報告している [14]．また，丸山はリファクタリング時に，続けて行うべき他のリファクタリングを推薦することができるツールを開発している [15]．同様に，連なったリファクタリングを自然に実行するためのインタフェースも提案されている [16]．提案手法は，小俣らの手法 [3] と同様，特定の名前変更に対して，同時に行うべき名前変更を自動で特定しており，こういった複数の名前変更の実現を支援している．

7. おわりに

本論文では識別子名の一括変更において従来手法を基に拡張を行い，一括変更のアプローチを提案した．また既存ツールから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を対象に提案手法と従来手法を比較し，従来手法より良好な結果を得た．今後の課題としては以下のものが考えられる．

- **活用形等への規則の適用．** 単語の置換や削除の際に識別子に含まれる単語が複数形等の形に活用されている場合，現在の手法ではマッチングに失敗してしまう．この場合，一度原型に戻してから変更を行うなどの処理が必要である．また，名前変更には識別子中の名詞を単数形から複数形にするといったものもあり，その場合は単語の置換ではなく単語の活用としてとらえる必要がある．
- **プラグインとしての提供．** 実際に開発者への支援ツールとして提供する場合，Eclipse プラグインなどの形で提供することが望ましい．また，支援ツールを提供し実際に開発者に利用してもらうことにより実用性を測る実験も可能となる．
- **短縮された識別子名への支援．** 本論文で行った実験では短縮された識別子名を考慮していないため，いくつかの名前変更で提案漏れがあった．これは AMAP [5] や Corraza らの省略元特定手法 [6] 等を提案手法と組み合わせることで解決を試みる．
- **前後の単語をより考慮に入れた単語の置換．** 本論文の提案手法では置換の意図を検出した時のマッチングで，識別子中に置換前の単語があれば条件なしに置換を行ったが，文脈や前後の単語の関係を考慮していない

ため誤推薦が多く発生した．これを防ぐには前後の単語や文脈を考慮変更の可否を判断させる必要がある．

謝辞　本研究の一部は科学研究費補助金（JP15K15970，JP15H02685，JP15H02683）の助成を受けた．

参考文献

[1] Fowler, M.: *Refactoring: Improving the Design of Existing Code*, Addison Wesley (1999).
[2] Boswell, D. and Foucher, T.: *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code (Theory in Practice)*, O'Reilly Media (2011).
[3] 小俣仁美，林　晋平，佐伯元司：命名方法の関連性に基づく識別子名の一括変更支援，情報処理学会研究報告，Vol. 2016-SE-191, No. 23, pp. 1–8 (2016).
[4] 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).
[5] 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 (MSR 2008)*, pp. 79–88 (2008).
[6] Corazza, A., Martino, S. D. and Maggio, V.: LINSEN: An Efficient Approach to Split Identifiers and Expand Abbreviations, *Proc. 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pp. 233–242 (2012).
[7] Madani, N., Guerrouj, L., Penta, M. D., Gueheneuc, Y.-G. and Antoniol, G.: Recognizing Words from Source Code Identifiers Using Speech Recognition Techniques, *Proc. 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pp. 68–77 (2010).
[8] Caprile, B. and Tonella, P.: Restructuring program identifier names, *Proc. International Conference on Software Maintenance (ICSM 2000)*, pp. 97–107 (2000).
[9] Lawrie, D., Binkley, D. and Morrell, C.: Normalizing Source Code Vocabulary, *Proc. 17th Working Conference on Reverse Engineering (WCRE 2010)*, pp. 3–12 (2010).
[10] Deissenboeck, F. and Pizka, M.: Concise and Consistent Naming, *Software Quality Journal*, Vol. 14, No. 3, pp. 261–282 (2006).
[11] Wake, W. C.: *Refactoring Workbook*, Addison-Wesley (2003).
[12] 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).
[13] 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).
[14] 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 (IWESEP 2014)*, pp. 31–36 (2014).
[15] 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).
[16] 田島香織，丸山勝久：ネスト化によるリファクタリングの連続的適用，ソフトウェア工学の基礎 XXI，pp. 225–230 (2014).