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.

類似メソッドの集約のための差分抽出支援

政井　智雄† 　吉田　則裕† 　松下　誠† 　井上　克郎†

† 大阪大学 大学院情報科学研究科
〒 560–0871 大阪府吹田市山田丘 1 番 5 号
E-mail: †{t-masai,n-yosida,matusita,inoue}@ist.osaka-u.ac.jp

あらまし　コードクローンとは，ソースコード中に存在する同一，または類似したコード片のことである．例えばあるコード片に欠陥が含まれている場合，そのコードクローン全てに対し修正を検討する必要が生じる．コードクローンの保守を容易にする方法の 1 つとして，コードクローンの集約が挙げられるが，差分を含むコードクローンの集約は比較的困難である．本論文では，差分を含むコードクローンである類似メソッドを集約するために，差分の抽出を支援する手法を提案し，有効性を確認した．

キーワード　コードクローン, Template Method パターン, リファクタリング, ソフトウェア保守

Supporting Difference Extraction for Merging Similar Methods

Tomoo MASAI†, Norihiro YOSHIDA†, Makoto MATSUSHITA†, and Katsuro INOUE†

† Graduate School of Information Science and Technology, Osaka University
1–5 Yamadaoka, Suita, Osaka, 565–0871 Japan
E-mail: †{t-masai,n-yosida,matusita,inoue}@ist.osaka-u.ac.jp

Abstract　A code clone is a code fragment that has identical or similar code fragments to it in the source code. For example, if we modify a code fragment which has code clones, it is necessary to consider whether we have to modify each of its code clones. Merging code clones is one of ways to reduce maintenance cost for code clones. However, merging code clones that contain differences is more difficult. In this paper, we propose a technique which supports extracting differences among code clones in order to merge code clones, and show the effectiveness of our technique.

Key words　Code Clone, Template Method Pattern, Refactoring, Software Maintenance

1. まえがき

ソフトウェアの保守を困難にしている要因の 1 つとして，コードクローン [1] [2] [3] が指摘されている．コードクローンとは，ソースコード中に存在する同一，または類似したコード片のことであり，開発，保守の工程において，あるコード片に修正を行う必要が生じた場合，そのコードクローン全てに対して修正を検討しなくてはならない [4] [5] [6]．そして近年のソフトウェアの大規模化，複雑化に伴ない，このような作業にかかるコストは非常に大きくなっている．したがってコードクローンを効率的に集約し，取り除くことができれば，これらのコストを低減させることが可能である．しかし，コードクローンの中でも，差分を含むコードクローンの集約は差分の修正，除去を考慮する必要があり，比較的困難なため修正作業の効率が低下する場合が考えられる [7].

ソフトウェアからコードクローンを取り除き，保守性を高めるための手段の 1 つとして，リファクタリングが挙げられる [8]. リファクタリングとは “ 外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること ” [9] である．メソッドの抽出や，フィールドの移動などの繰り返し行われているとされるリファクタリングは，リファクタリングパターンとして書籍 [9] やウェブサイト [10] にまとめられている．

コードクローンに対し効果的であるといわれているリファクタリングパターンの 1 つに “ Template Method の形成 ” がある．“ Template Method の形成 ” とは，デザインパターンの 1 つである Template Method パターンが適用された状態にソースコードを修正するリファクタリングパターンである [9]. Template Method パターンとは，親クラスでメソッドの処理のフレームワークを定め，子クラスでその処理の具体的内容を定めるデザインパターンである．共通である部分のみを親クラスで定義することで，子クラス間でコードクローンが現れるこ

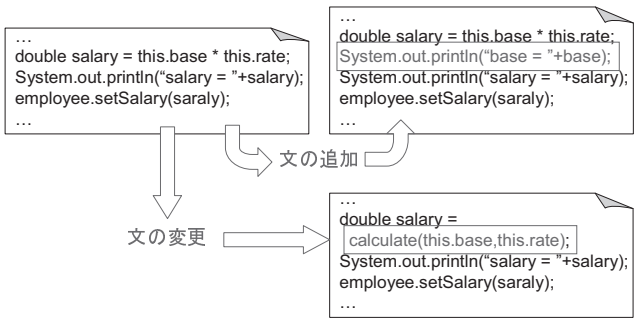

図 1　差分を含むコードクローンの例

とを回避している．このリファクタリングパターンの利点の 1 つは，差分を含むコードクローンでも集約することができる点である．

この利点に注目し，本研究では，この“ Template Method の形成 ”を支援する手法を提案する．“ Template Method の形成 ”は，大きく分けて，差分の検出，差分の抽出，メソッドの引き上げの 3 つのステップで行われる．本手法はこの内，差分の検出，および差分の抽出の支援を行った．この提案手法は統合開発環境 Eclipse に組み込んで利用する Eclipse プラグインとして実装し，Eclipse を用いた開発作業における必要な状況で利用できるようにした．この Eclipse プラグインを用いて，オープンソースソフトウェアの中から選出した 2 つの類似したメソッドを対象とした適用実験を行った．提案手法を用いて，実際に“ Template Method の形成 ”を行い，そのリファクタリングの前後でプログラム全体の外部的動作に変化が無いことを確認できたため，提示された候補の有効性を確認できた．

以降，2 節では本研究に関わる用語について説明する．また 3 節では，本研究の提案手法を説明し，4 節では Eclipse プラグインを用いた適用実験の結果を，5 節では考察を，6 節で関連研究を述べ，最後に 7 節で本研究のまとめと今後の課題について述べる．

2.　コードクローンを集約するリファクタリング

これまでにコードクローンにはいくつか分類定義が考えられている．コードクローン間の相違の度合いに基づいた 3 つの分類を Bellon は定義しており，その中の 1 つにタイプ 3 のコードクローンがある [11]．タイプ 3 のコードクローンは，あるコード片をコピーアンドペーストした後，開発者が文の挿入や削除を行うことで作成される（図 1 参照）．以降，このタイプ 3 のコードクローンを，**差分を含むコードクローン**と呼ぶ．

リファクタリングとは“ 外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること ”[9] である．Fowler は設計の問題を匂いに例え，コードの不吉な匂い（Bad Smell）[9] と呼んでおり，その不吉な匂いを解消する為にどのようなリファクタリングが最も役に立つかを説明している．コードの不吉な匂いとされる設計の問題の 1 つにコードクローンがあり，その対処としていくつかのリファクタリングパターンが挙げられている [9]．そのうち

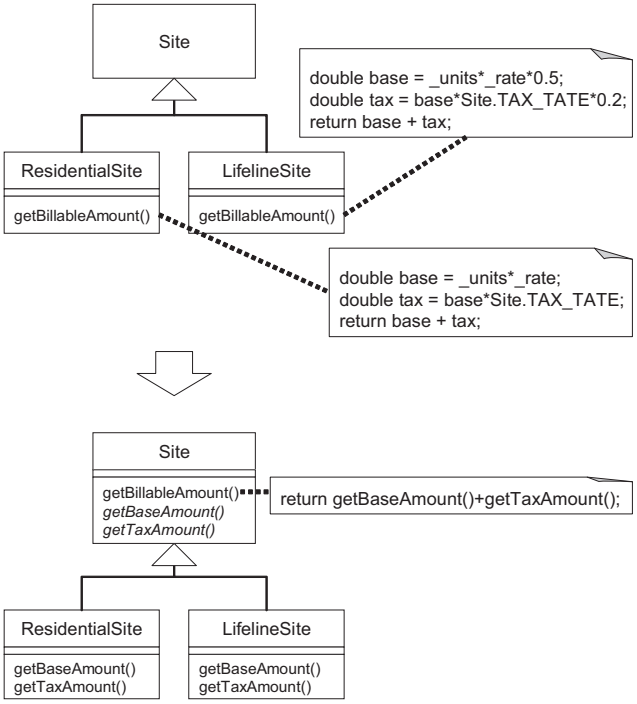

図 2　Template Method の形成 [9]

中で，差分を含むコードクローンに効果的であると言われているリファクタリングパターンの 1 つが“ Template Method の形成 ”である．

“Template Method の形成”は，親クラスが共通で，類似した処理の流れのメソッドを互いに有する子クラス間に，Template Method パターンを適用するリファクタリングパターンである．Template Method パターンとは，共通な処理の流れを親クラスで実装し，処理の具体的な実装は子クラスで行うデザインパターンである．“ Template Method の形成 ”は，以下の 4 つのステップに分けられる．これらのステップを，図 2 を用いて説明する．

a）**ステップ A：メソッド間の差分検出**

図 2 の類似した 2 つのメソッド (ResidentialSite クラスと LifelineSite クラスの getBillableAmount メソッド) では，戻り値とする式は同じだが，その式に用いられる 2 つの変数のローカル宣言文に違いがあるため，これらを差分とする．

b）**ステップ B：メソッドとして抽出するコード片の決定**

ステップ A で検出した差分を吸収できるように，抽出するコード片を決定する．図 2 においては，変数 base の宣言文及び参照，変数 tax の宣言文及び参照をそれぞれ抽出するコード片とする．

c）**ステップ C：コード片の抽出**

ステップ B で決定した抽出するコード片を，実際にメソッドとして抽出する．この際，抽出したコード片で定義されるメソッドのシグニチャ，引数として渡す変数，及び値を戻す変数は同じでなくてはならない．シグニチャとは，呼び出し時にメソッドを特定するために必要な，メソッド名と引数の型の列である．同じシグニチャ，戻り値の型の抽象メソッドを親クラスで定義しておくことで，各子クラスでのメソッドの実装

は抽象メソッドをオーバーライドする形になる．図 2 においては，まずそれぞれの変数の宣言文を抽出し，2 つのメソッド（getBaseAmount メソッド，getTaxAmount メソッド）を定義し，同じシグニチャ，戻り値の型の抽象メソッドを親クラス（Site クラス）に定義する．次に，変数の宣言文を消去し，変数の参照をそれぞれのメソッド呼び出し文に置き換えることで記述を揃えている．

d） **ステップ D：類似メソッドの引き上げ**

ステップ C で記述を揃えたメソッド（getBillableAmount）を，親クラスへと引き上げることで，Template Method パターンが適用されたソースコードに修正を終えた．

上記の 4 つのステップで“ Template Method の形成 ”がなされ，コードクローンを取り除くことができた．このリファクタリングにより，図 2 の Site メソッドの子クラスを新たに追加する場合，リファクタリング後は 2 つの抽象メソッドを実装するだけで子クラスを追加できるため，拡張性の高い設計となっており，また類似メソッドが増加する可能性も低くなる．

3. 提 案 手 法

本節では，提案手法の概要，また提案手法における実装を述べる．

本手法は入力として 2 つの類似メソッドが与えられ，動作は以下の 6 つのステップに分けられる．図 3 は提案手法全体の流れを示す．

ステップ 1　2 つの類似メソッドのソースコードから，抽象構文木をそれぞれ生成する．

ステップ 2　2 つの抽象構文木間の差分となっている部分木を検出する．

ステップ 3　差分となっている部分木を含む，メソッドとして抽出可能な部分木列を検出する．

ステップ 4　ステップ 3 で検出した部分木列から範囲を拡大した，抽出可能な部分木列を可能な限り検出する．

ステップ 5　ステップ 3，ステップ 4 で検出されたメソッドとして抽出可能な部分木列を，実際に抽出を行った後，記述を揃えることが容易か否かにより分類する．

ステップ 6　ステップ 5 で行った分類に従い，メソッドとして抽出可能な部分木に対応するソースコード中のコード片を提示する．

3.1 [ステップ 1] 抽象構文木の生成

Eclipse の抽象構文木生成の機能を用いて，用意された 2 つのメソッドの抽象構文木を生成する．この抽象構文木を成すノードは，主に以下の 4 つの種類に分けられる．

値のみを持つノード　ユーザ定義名や，定数などがこれに当たる．

子ノードのみを持つノード　return 文などがこれに当たる，子ノードを持たない場合もある．

子ノードと値を持つノード　代入文などがこれに当たる．

子ノードの列を持つノード　{} で囲まれた記述がこれに当たる，セミコロンや {} で区切られる文がそれぞれ子ノードとなる．

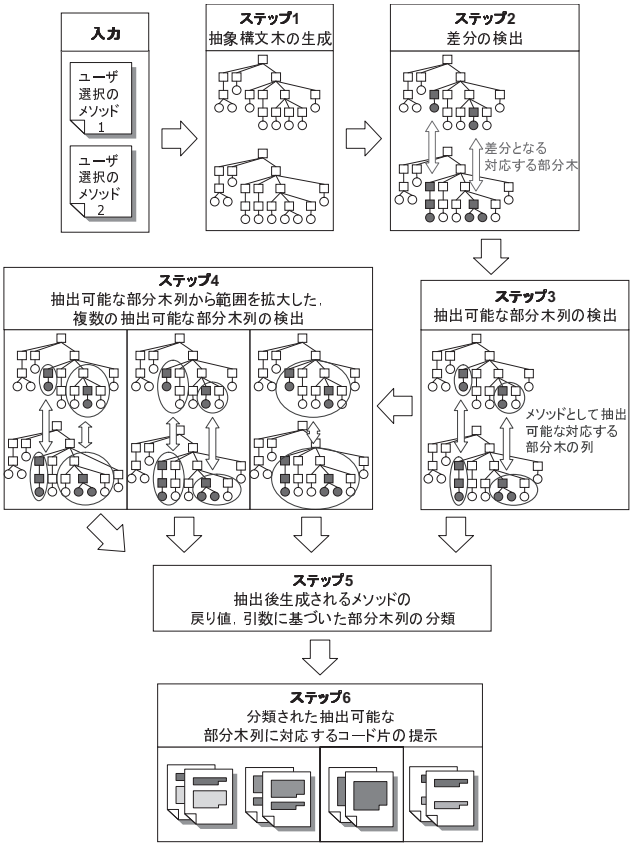

図 3　提案手法全体の流れ

これらのノードをそれぞれ，ノード A，ノード B，ノード C，ノード D と呼ぶ．ただし，この分類は，ノードの種類（ユーザ定義名，if 文，代入文など）とは異なる．

3.2 [ステップ 2] 差分となっている部分木の検出

ステップ 1 で生成された 2 つの抽象構文木の比較を行い，抽象構文木間の差分となっている部分木を検出する．

抽象構文木の比較は，メソッド宣言に対応するノードから子ノードに向かい，再帰的にノードを比較することで行う．ノードの比較は主に以下の 4 つの操作から成る．

種類の比較　ノードの種類が同じか比較する，異なっていれば対応する差分とする．同じであれば，以下の比較を行う．

値の比較　ノード A，ノード C の場合，ノードの持つ値をそれぞれ比較する，異なっていれば対応する差分とする．

子ノードの比較　ノード B，ノード C の場合，子ノードをそれぞれ比較する．異なっていた場合，その子ノードと対応する記述がメソッドとして抽出し得ない場合（代入文の左辺など）のみ差分とする．

列の比較　ノード D の場合，子ノードの列を比較する．比較の方法は既存の類似文字列マッチングアルゴリズム [12] を用いている．比較の結果，列全てが差分となった場合のみ，差分とする．ノード列の比較において，このアルゴリズムを用いることで，同一であるノード，差分となるノードを，全て比較を行った上で把握することができ，差分となるノードの数を，単純な列の比較に比べ少なくすることができる．

差分と判断されたノードの子ノードは全て差分と判断（対応

関係は取らない）するため，差分となるノードを以降，差分となる部分木とも呼ぶ．

3.3 [ステップ 3] 抽出可能な部分木の検出

差分となる部分木を含む，メソッドとして抽出可能な部分木，または部分木の列（ノード D の子ノードの一部）を検出する．部分木に対応するソースコード中のコード片が，メソッドとして抽出可能か否かは Eclipse のメソッド抽出機能の事前条件チェッカーを用いて判定する．Eclipse のメソッド抽出機能が，メソッドとして抽出することが不可能であると判定する条件の中で，本手法に関係するものは以下の 3 つである．

条件 1　複数の変数の宣言文，または変数への代入文が含まれており，それらの変数に対する参照が後のコード記述されている．

条件 2　抽出するコード片に break 文，continue 文が含まれているが，それらに対応する制御文が含まれていない．

条件 3　構文木を生成しているメソッドの戻り値が void 型であり，かつ抽出するコード片が return 文を含んでいる．

差分となる部分木に対応するソースコード中のコード片が，メソッドとして抽出可能であった場合は，残りの差分となる部分木にも同様に判定を行う．メソッドとして抽出不可能と判定された場合には，以下の操作 1 と操作 2 を同時に開始し，再帰的に抽出可能な部分木を検出する．

操作 1　ノード D の子ノードである場合，列において 1 つ前のノード（ソースコード上で左，または上に位置する文）を抽出する部分木の列に含め，抽出可能であれば検出する．部分木列が抽出可能であると判定されるまで，繰り返し操作 1 と操作 2 を開始する．列において 1 つ前が無い，またはノード A，ノード C の子ノードである場合，または類似メソッドの対応する部分木列が抽出不可能と判定された場合，操作 3 を行う．

操作 2　ノード D の子ノードである場合，列において 1 つ後のノード（ソースコード上で右，または下に位置する文）を抽出する部分木の列に含め，抽出可能であれば検出する．部分木列が抽出可能であると判定されるまで，繰り返し操作 2 を開始する．列において 1 つ後が無い，またはノード A，ノード C の子ノードである場合，操作を終了する．

操作 3　範囲を，親ノードを含む部分木とし，抽出可能であれば検出する．抽出が不可能であると判定された場合，または類似メソッドの対応する部分木列が抽出不可能と判定された場合，操作 3 を行う．この部分木に対し操作 1 と操作 2 を開始する．

上記の操作を全ての差分となる部分木に対して行い，それぞれに対してメソッドとして抽出可能な部分木列を検出する（図 4，参照）．

3.4 [ステップ 4] 範囲を拡大した複数の部分木列の検出

ステップ 3 で検出した部分木列から範囲を拡大した，メソッドとして抽出可能な部分木列を可能な限り複数検出する．検出は，ステップ 3 と同様の操作で行う．ただし，メソッドとして抽出可能な部分木列が全て検出されるまで操作は終了しない．

3.5 [ステップ 5] 部分木列の分類

検出されたメソッドとして抽出可能な部分木に対応するコード片を抽出した場合に生成されるメソッド呼び出し文の引数と

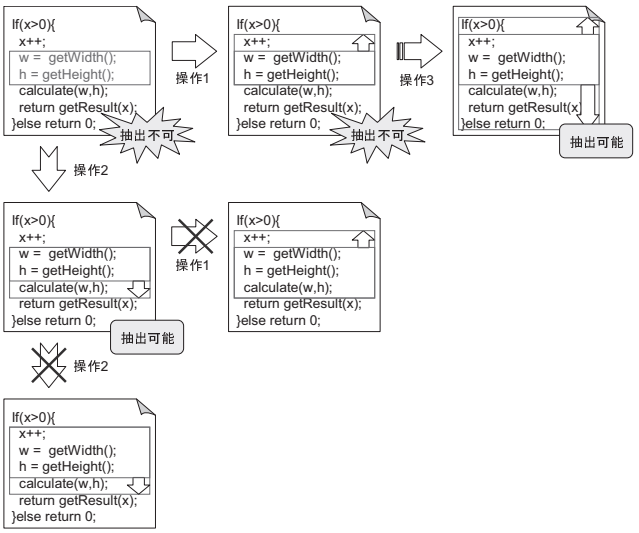

図 4　メソッドとして抽出可能な部分木列検出のための操作

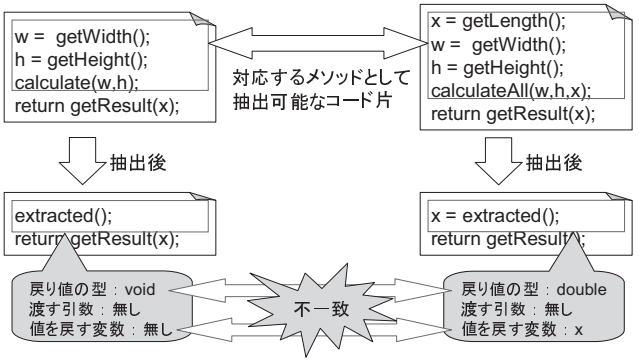

図 5　抽出後の記述が一致しない例

して渡す変数，及び値を戻す変数を調べる．これらが，対応する抽出箇所で互いに異なる場合，単純なメソッドの抽出で記述を揃えることができず，容易に Template Method を形成することができない（図 5，参照）．以下の条件を用いて，メソッドとして抽出可能な部分木に対応するコード片を 6 つに分類する．

条件 1　値を戻す変数が異なる．

条件 2　引数として渡す変数が異なる．

条件 3　片方にのみ対応する位置にコード片がない

分類 1　条件 1，条件 2 を満たす抽出箇所が存在しない．

分類 2　条件 1 を満たすが抽出箇所が 1 つ存在する．

分類 3　条件 2 を満たす抽出箇所が 1 つ存在する．

分類 4　条件 1，及び条件 2 を満たす抽出箇所が 1 つ存在する．

分類 5　条件 3 を満たす抽出箇所が 1 つ存在する．

分類 6　条件 1〜3 のうちいずれか，もしくは複数を満たす抽出箇所が複数存在する．

これらの分類のうち，分類 1 はそのままメソッドの抽出を行うことで記述を揃えることができ，Template Method の形成が容易といえる．分類 2〜5 は，抽出を行うだけでは，1 箇所記述が揃わず，分類 6 に至っては複数箇所の記述が揃わないため，メソッドの抽出を行う前にソースコードを修正する必要があり，Template Method の形成が容易とはいえない．したがっ

て，これらを分類することで，Template Method の形成が容易な候補の理解を促すことが可能となる．

3.6 [ステップ 6] コード片の提示

ステップ 5 で行った分類に従い，メソッドとして抽出可能な部分木に対応するソースコード中のコード片を提示する．コード片は，Eclipse におけるウィザードで表示することで提示する．

4. 適 用 実 験

本節では，3. 節で説明した提案手法を，実際のソースコードに適用した事例について述べる．

提案手法は，リファクタリングを支援することを目的としている．そのため，提示される候補がリファクタリングに有効であり，また必要な作業に応じて候補が正しく分類されている必要がある．したがって本実験の目的は，提示される候補の有効性，および候補に対して行う分類の有効性をそれぞれ確認することとした．

4.1 準　　備

適用実験には，Java，C#，C++等に対応したコンパイラ・コンパイラであるオープンソースソフトウェアの ANTLR2.7.4(注1)を対象に用いた．また，ANTLR2.7.4 から，提案手法を適用する類似メソッドを検出するためのコードクローン検出ツールとして，Scorpio [13](注2)を使用した．Scorpio は特殊なプログラム依存グラフを用いるコードクローン検出ツールである．プログラム依存グラフを用いたコードクローン検出は，差分を含むコードクローンを検出することができるという長所を持つ．Scorpio を用いて，ANTLR2.7.4 から差分を含むコードクローンを検出し，その中からメソッドの記述を大きく占めており，差分を複数含むコードクローンを調査し，それらを含む 2 つの類似メソッドを対象に決定した．対象としたメソッドは CppCodeGenerator クラスの genErrorHandler メソッド と，JavaCodeGenerator クラスの genErrorHandler メソッドである．

4.2 提案手法を用いたリファクタリング

CppCodeGenerator クラスの genErrorHandler メソッド と JavaCodeGenerator クラスの genErrorHandler メソッドを提案手法を実装したプラグインへの入力とした結果，表 1 に示す数の候補が提示された．提示された候補の中から，分類 1 及び分類 3 の候補から 10 個ずつ，分類 6 の候補から 20 個を選び，実際にリファクタリングを行った．まずリファクタリングに必要だった操作を調査した．ただし，メソッドの抽出，メソッドの引き上げ，及び親クラスにおける抽象メソッドの定義は，Eclipse の既存の機能を用いることで容易に行うことができ，また全てのリファクタリングにおいて行うため，これらの操作には含まない．調査する操作は以下のとおりである．

引数の変更（追加）　節 3.5 における条件 2 を満たしているコード片の場合，抽出後のメソッドの引数が一致するよう引数を追加する．

戻り値の変更　節 3.5 における条件 1 を満たす場合，戻り値の型，及び値を返す変数が共通となるように抽出前，または抽出後のコード片を修正する．

メソッドの作成　節 3.5 における条件 3 を満たしている場合，片方が抽出を行うコード片がないため，新たなメソッドを作成し，その呼出文を対応する位置に挿入する．

リファクタリングを行った結果，候補に対しそれぞれ必要な操作は表 2 のようになった．

続いて，それぞれのリファクタリングを行ったソフトウェアに対し，86 個のテストケースを用いたテストを行い，リファクタリングの前後において外部的動作に変化がないことを確認した．

4.3 考　　察

分類された候補を用いてリファクタリングを行った結果，それぞれに対して必要な操作の数から，回数の違い毎に正しく分類されていることが確認できた．さらに分類 1 の候補を用いたリファクタリングについては，表 2 のように，必要な操作がなく，容易に“Template Method の形成”を行うことができた．そのため記述を揃えることが容易な候補を絞り込むことができたと考えられる．これらの結果から，分類の有効性を確認でき，目的 2 を達成できた．しかし，分類 6 については，表 2 のように必要な操作の種類及び回数に差が生じているため，より多くの分類を考案する必要があると考えられる．さらに，提示された候補の総数は 312 個であり，分類を行ったとしても多くなっている．そのため，これらの候補の中から有効性の高いものを特定する，または必要のないものを除外するためのメトリクスを考案する必要もあると考えられる．

次に，リファクタリングの前後で外部的動作に変化がないことを確認できたため，提示された候補がリファクタリングに有効であると確認でき，目的 1 を達成できた．しかし候補を用いたリファクタリングの結果，親クラス（CodeGenerator クラス）に新たに定義した抽象メソッドを，Template Method パターンとは関係のない子クラスでもオーバーライドする必要がある構造になってしまった．また，CppCodeGenerator クラスと JavaCodeGenerator クラスの genErrorHandler メソッドだけでなく，CSharpCodeGenerator クラスの genErrorHandler メソッドも，差分となる位置は異なるが，前者の 2 つのメソッ

表 1　ツールの実行結果

	分類 1	分類 2	分類 3	分類 4	分類 5	分類 6
候補の数	31	0	122	0	0	159

表 2　必要だった操作

	分類 1	分類 3	分類 6
操作なし	10	0	0
引数の変更 1 回	0	10	0
引数の変更 2 回	0	0	4
引数の変更 1 回＋メソッドの作成 1 回	0	0	6
引数の変更 2 回＋メソッドの作成 1 回	0	0	1
引数の変更 1 回＋メソッドの作成 2 回	0	0	9

(注1)：http://www.antlr.org/

(注2)：http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio/

ドと差分を含むコードクローンとなっているため，これら 3 つのメソッド間において抽出箇所を検出できれば，さらに優れた構造になる可能性があると考えられる．

5. 関 連 研 究

Juillerat らは，Template Mehtod の形成をソースコードへ行う作業を自動化する手法を提案している [14]．Juillerat らの手法は，本手法と同様にソースコードから生成した抽象構文木を用いて，差分となる部分木を検出する．しかし本手法が差分となる部分木を検出するために，抽象構文木の根から構造的に比較を行ったのに対し，Juillerat らの手法では，2 つの抽象構文木を深さ優先探索の帰りがけ順に探索することで作成した AST ノードの列に対して比較を行う．列の比較によって得た差分となるノードを，全て含むような最小の部分木を特定することで，差分となる部分木を検出している．本手法の比較に比べ，抽象構文木の構造的な情報を失うが，高速度の比較を行うことが可能となっている．

メソッドの抽出においても本手法と異なる点がある．本手法では，節 3.3 で説明した条件を満たした差分である部分木が検出された場合，抽出する部分木列に隣り合う部分木を加え，抽出範囲を広げることで抽出可能な部分木列を検出している．比べて Juillerat らの手法では，差分となる部分木が節 3.3 の条件を満たすそれぞれの場合に対し以下の方法で解決を行っている．

条件 1 を満たす場合　互いに共通な代入文を特定し，片方に足りない代入文は，対応する位置に同じ変数に対する値が変化しない代入文を追加し，抽出する場合はそれらを 1 つずつ抽出することで，抽出範囲を変化させず抽出を行える状態に変化させている．

条件 2，または条件 3 を満たす場合　抽出したメソッドの戻り値に専用のクラスインスタンスを使用し，抽出後の記述にその戻り値によって break 文，continue 文，return 文に分岐する記述を加えることで，抽出範囲を変化させず抽出を行える状態に変化させている．

本手法は，範囲の拡大を行い，抽出する部分木列の候補を複数検出することで，開発者が適切だと考える候補を選ぶことの支援としているが，Juillerat らの手法は，全ての抽出を自動化することを目的としているため，範囲を変化させず抽出後の形を 1 つとしている．

6. まとめと今後の課題

本研究では，類似した 2 つのメソッドに対し，Template Method パターンを適用するリファクタリングの支援を行った．

今後の課題としては，2 つのメソッド間での異なるユーザ定義名の対応付け，及び 3 つ以上の類似メソッドへの適用支援，の 2 つが考えられる．

1 つ目の課題として，類似メソッド間での異なるユーザ定義名の対応付けが考えられる．本手法では，ユーザ定義名は文字列が一致していない限り差分として扱う．しかしこの場合，ユーザ定義名が異なるが，共通の動作を含む 2 つのメソッドを対象としたリファクタリングを支援することが難しい．よって，類似メソッド間でのユーザ定義名を，記述が異なる場合でも対応させ，その上で共通であるコード片と差分であるコード片を把握することができれば，より多くの類似メソッドに対しリファクタリングの支援を行うことが可能となる．

2 つ目の課題として，3 つ以上の類似メソッドへ対応が考えられる．共通の親クラスを持つ 3 つ以上の子クラスが全て類似メソッドを持っている場合があるが，その場合も本手法では 2 つの類似メソッドを対象としたリファクタリングしか支援できない．よって，3 つ以上のの類似メソッド間の差分の検出，及び抽出箇所の検出を行うことができれば，本手法を用いたリファクタリングを行った結果得られるソースコードよりも，優れた構造を持つソースコードを得ることが出来ると考えられる．

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）の助成を得た．

文　献

[1] L. Jiang, G. Misherghi, Z. Su and S. Glondu: “DECKARD: Scalable and accurate tree-based detection of code clones”, Proc. of ICSE 2007, Minneapolis, MN ,USA, pp. 96–105 (2007).

[2] T. Kamiya, S. Kusumoto and K. Inoue: “CCFinder: A multilinguistic token-based code clone detection system for large scale source code”, IEEE Trans. Softw. Eng., **28**, 7, pp. 654–670 (2002).

[3] R. Komondoor and S. Horwitz: “Using slicing to identify duplication in source code”, Proc. of SAS 2001, Paris, France, pp. 40–56 (2001).

[4] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo and J. Hudepohl: “Assessing the benefits of incorporating function clone detection in a development process”, Proc. of ICSM ’97, Bali, Italy, pp. 314–321 (1997).

[5] Z. Li, S. Lu, S. Myagmar and Y. Zhou: “CP-Miner: Finding copy-paste and related bugs in large-scale software code”, IEEE Trans. Softw. Eng., **32**, 3, pp. 176–192 (2006).

[6] A. Zeller: “Why Programs Fail”, Morgan Kaufmann Pub. (2005).

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë and K. Kontogiannis: “Measuring clone based reengineering opportunities”, Proc. of METRICS ’99, Boca Raton, FL, USA, pp. 292–303 (1999).

[8] W. F. Opdyke: “Refactoring object-oriented frameworks”, PhD thesis, University of Illinois at Urbana-Champaign (1992).

[9] M. Fowler: “Refactoring: Inproving the Design of Existing Code”, Addison Wesley (2000).

[10] M. Fowler: `http://refactoring.com/`.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo: “Comparison and Evaluationof Clone Detection Tools”, IEEE Trans. Softw. Eng., **33**, 9, pp. 577–591 (2007).

[12] R. B. Yates and B. R. Neto: “Modern Information Retrieval”, Addison Wesley (1999).

[13] 肥後芳樹, 楠本真二：“実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法”, ソフトウェアエンジニアリング最前線 2009, pp. 97–104 (2009).

[14] N. Juillerat and B. Hirsbrunner: “Toward an Implementation of the “Form Template Method” Refactoring”, Proc. of SCAM 2007, Paris, France, pp. 81–90 (2007).