情報処理学会論文誌 Vol.67 No.4 748–761 (Apr. 2026)

C#におけるラムダ式を用いたメソッドの抽出リファクタリングによるコードクローン集約

川本 琢人[1,a)] 肥後 芳樹[1,b)]

受付日 2025年7月29日, 採録日 2026年1月13日

概要：コードクローンは，ソフトウェアの一貫性のない変更を助長し，ソフトウェアの保守性を低下させる．コードクローンのリファクタリングはソフトウェアの保守性を高く保つために有効であると考えられる．リファクタリング手法の1つにメソッド抽出リファクタリングある．このリファクタリングは一般に，識別子名の違いや空白文字の違いを含むコードクローンに対して適用できる．文単位の差異を含むコードクローンであっても，ラムダ式を用いて互いに異なる動作をパラメータ化するとメソッドの抽出リファクタリングによるリファクタリングが可能になる．しかしラムダ式はプログラミング言語によって仕様が様々に異なる．最適な手法はプログラミング言語ごとに熟慮を要する．本稿ではC#を対象にラムダ式を用いたメソッドの抽出リファクタリングの手法を提案する．提案手法を実際のプロジェクトのクローンペア2,714件に対して適用したところ，全体の11.6%のクローンペアで正確なリファクタリングが確認された．また，24.0%のクローンペアでリファクタリング後のソースコードはビルドを通過し，62.6%のクローンペアをリファクタリング不可能と判断した．

キーワード：コードクローン，リファクタリング，メソッドの抽出，ラムダ式

A Technique to Remove Code Clone by Applying Extract Method Refactoring with Lambda Expression for C#

Takuto Kawamoto[1,a)] Yoshiki Higo[1,b)]

Received: July 29, 2025, Accepted: January 13, 2026

Abstract: Code clones encourage inconsistent software changes and degrade software maintainability. Refactoring code clones is considered effective for improving software maintainability. One refactoring technique is Extract Method refactoring. This refactoring is generally effective for code clones that differ in identifier names and whitespace. Even for code clones containing statement-level differences, Extract Method refactoring becomes possible by parameterizing the differing behaviors using lambda expressions. However, the detailed specifications of lambda expressions vary significantly across programming languages. Therefore, the optimal approach requires careful consideration for each programming language. This paper proposes an Extract Method refactoring technique using lambda expressions specifically for C#. When the proposed technique was applied to 2,714 clone pairs from actual projects, 11.6% of the clone pairs were accurately refactored. Furthermore, the refactored source code passed the build process for 24.0% of the clone pairs, while 62.6% of the clone pairs were determined to be unrefactorable.

Keywords: code clone, refactoring, extract method, lambda expression

1. はじめに

コードクローンは，ソースコード中の類似したコード片である．互いに類似する2つのコード片のペアをクローン

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, The University of Osaka, Suita, Osaka 565–0971, Japan
[a)] tk-kawam@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp

ペアという [1], [2]．コードクローンのリファクタリングは重複した記述に起因する一貫性のない変更を防止し [3]，保守にかかるコストを削減する [4]．

コードクローンのリファクタリング手法の 1 つにメソッドの抽出リファクタリングがある [5], [6]．メソッドの抽出リファクタリングは，重複したソースコードの共通部分を単一の新たなメソッドに抽出し，コードクローンをメソッド呼び出しに置き換える．多少の差異を含むコードクローンを集約するために，差異の部分をメソッドのパラメータに抽出するパラメータ化が有効である [5]．定数値やローカル変数の差異を式の単位でパラメータ化する手法がすでに提案されている [7]．

コードクローンに含まれる差異は定数値やローカル変数の差異のみに限らない．Roy らは，コードクローンをその一致の程度から 4 つのタイプに分類した [8]．この分類におけるタイプ 3 のコードクローンは，いくつかの文を挿入，削除，あるいは変更して一致する．このため，タイプ 3 のコードクローンには式単位より大きな文単位の差異が含まれる．文単位の差異は式単位のパラメータ化では抽出できない．

文単位の差異を含むコードクローンを集約する方法の 1 つに，文の順序を入れ替える方法がある [9]．互いに異なる文をコードクローンの前後に移動させて，コードクローンを式単位のパラメータ化で十分集約できるように変形する．しかし，多くの文には副作用があるため，自由な入れ替えは不可能であり，この方法で集約可能なコードクローンは限られる．副作用による影響を受けない文の入替えのため，文間のデータ依存関係の解析が必要である [9]．

また他の方法として差異を文単位でパラメータ化する方法が考案されている．この方法はラムダ式を用いて 0 個以上の連続した文をパラメータに抽出する [7]．コードクローン内の文の評価順序は変わらないため，データ依存関係の解析は必要ない．先行研究では，文単位のパラメータ化を用いたリファクタリングを，Java で書かれたプログラムに対して試みている．

ラムダ式に関係する詳細な言語仕様はプログラミング言語によって異なる．言語仕様の違いは，ラムダ式による文単位のパラメータ化を用いたリファクタリングの適用可能性に影響を及ぼす．本研究では，先行研究で取り上げられていない C#を対象に，ラムダ式を用いたリファクタリング手法を提案する．

先行研究は，`break` 文や `continue` 文のようにプログラムの実行位置をジャンプさせる文を含むコードクローンのリファクタリングの対象から除外した．本研究は，これらの文を含むコードクローンを単一のメソッドに集約する手法を提案する．

Java には，指定した範囲のコードカバレッジを最大にするユニットテストを自動で生成するツールが存在する [10]．このようなテストの自動生成ツールを用いると，施したリファクタリングがソフトウェアの挙動を変えていないか調べるのが容易になる．しかし，C#には同様のツールが存在せず，自ら手作業でテストを書く以外にはソフトウェアに付属しているテストを使うしかソフトウェアの挙動の変化を観察する手段がない．本研究での提案手法の評価では，ソフトウェアに付属するテストを活用してリファクタリングを評価する方法を提案する．

2 章では研究の背景となるコードクローンやリファクタリングについて説明し，3 章では提案手法について述べる．4 章ではリファクタリング手法の評価について説明し，5 章で考察を述べる．6 章以降では妥当性の脅威について，7 章でまとめを述べる．

2. 背景

本章では，本研究で扱うコードクローンおよびそのリファクタリングについて述べる．

2.1 コードクローン

コードクローンは，ソースコード中の類似したコード片である．互いに類似するコード片のペアをクローンペアという [2]．コードクローンは，典型的にはコピー・アンド・ペースト操作によるソースコードの複製によって発生する．コードクローンに一貫しない変更が施されると変更漏れが生じ，バグの原因になる [11]．Roy らはコードクローンをその一致の程度から次の 4 つのタイプに分類した [8]．

タイプ 1 空白文字，改行やコメント等を除いて一致するコードクローン．

タイプ 2 タイプ 1 の除外対象のほか，リテラル，型，識別子を除いてはじめて一致するコードクローン．

タイプ 3 タイプ 2 の除外対象のほか，文の変更，挿入，または削除の違いを許容してはじめて一致するコードクローン．

タイプ 4 同様な処理を行うが，構文が異なるコードクローン．

2.2 コードクローンのリファクタリング

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業である．リファクタリングには様々なパターンがある．長すぎる処理の一部を新たなメソッドに切り出すリファクタリングはメソッドの抽出と呼ばれる．このメソッドの抽出リファクタリングはコードクローンを集約するために活用できる．簡単のため，リファクタリングによって切り出される新たなメソッドをサブルーチンと呼ぶ．

メソッドの抽出リファクタリングによって，互いに類似したコード片はサブルーチンの定義のただ 1 カ所にまとまる．1 つのサブルーチンにまとめるために，変数をサブ

```
void PrintPerLine(string path)
{
  StreamReader reader = new StreamReader(path);
  while (reader.Peek() != -1) {
    string line = reader.ReadLine();
    Console.WriteLine(line);
  }
  reader.Close();
}

void PrintPerCharacter(string path)
{
  StreamReader reader = new StreamReader(path);
  while (reader.Peek() != -1) {
    string line = reader.ReadLine();
    for (int i = 0;i < line.Length; i++) {
      Console.WriteLine(line[i]);
    }
  }
  reader.Close();
}
```

リファクタリング前のソースコード.

```
void PrintPerLine(string path)
{
  Print(path, (string v3) => {
    Console.WriteLine(v3);
  });
}

void PrintPerCharacter(string path)
{
  Print(path, (string v3) => {
    for (int i = 0;i < v3.Length; i++) {
      Console.WriteLine(v3[i]);
    }
  });
}

delegate void LambdaDelegate1(string v3);
void Print(string v2, LambdaDelegate1 lambda1)
{
  StreamReader v1 = new StreamReader(v2);
  while (v1.Peek() != -1) {
    string v3 = v1.ReadLine();
    lambda1.Invoke(v3);
  }
  v1.Close();
}
```

リファクタリング後のソースコード.

図 1 動作のパラメータ化を用いたリファクタリング例

Fig. 1 An example of refactoring with behavior parametarization.

ルーチンの引数としてわたさなければならない場合がある．必要な変数をサブルーチンの引数として渡す方法をパラメータ化という．メソッドの抽出リファクタリングは，このパラメータ化によってローカル変数や差異を含むコードクローンの集約を可能にする [12].

2.2.1 式のパラメータ化

ローカル変数を参照する場合，あるいは異なる定数値を使用する場合は変数および定数をパラメータにしてサブルーチンに渡す必要がある．サブルーチンは元のローカル変数や定数値の代わりに，引数として渡されたパラメータを参照する．元のローカル変数や定数値はサブルーチンの呼び出し部にて，サブルーチンの実引数として記述される．このように値をパラメータに抽出する方法を，本稿では特に式のパラメータ化と呼ぶ.

式のパラメータ化はコードクローン間で互いに類似した文中に含まれる，互いに異なる式の差異を吸収できる．式のパラメータ化を用いたメソッドの抽出リファクタリングは，タイプ 1 およびタイプ 2 のコードクローンの集約に適しているといわれている [12]．しかし，タイプ 3 のコードクローンに含まれる差異は式の差異に限らない．タイプ 3 のコードクローンはいくつかの文の追加や削除を許容するため，互いに異なる文を含む．式の差異より大きな文の差異を吸収するには，異なるパラメータ化が必要である.

2.2.2 動作のパラメータ化

コードクローン間で互いに異なる文をパラメータにして差異を吸収するために，ラムダ式で書き表される関数オブジェクトが活用できる.

ラムダ式を用いて文単位の差異を吸収するパラメータ化をここでは動作のパラメータ化と呼ぶ．動作のパラメータ化はタイプ 3 のコードクローンに対して特に有効である.

本稿では，外部の変数をキャプチャでき，メソッド本体に 0 個以上の文を書ける関数オブジェクトの記法を指して単にラムダ式と呼ぶ．Python のラムダ式は 1 つの式のみ書けるが文は書けない．代わりにローカル関数がラムダ式として使用できる．また，JavaScript ではアロー関数式と呼ばれる無名関数がラムダ式に相当する.

動作のパラメータ化を用いてタイプ 3 のコードクローンにメソッドの抽出リファクタリングを施す例を**図 1** に示す．コードクローンの一致部分をオレンジ色で，不一致部分をそれぞれ異なる色でハイライトしている．リファクタリング前のソースコードには，ストリームから読み込んだ文字列をそれぞれの方法でコンソール画面に出力する命令が含まれる．メソッドの抽出によって，文単位の差異をラムダ式を用いて吸収しながら共通部分を集約している．このリファクタリングによって命令の実行順序に変化は生じない.

2.3 動作のパラメータ化の言語依存性

動作のパラメータ化は文単位の差異をパラメータに抽出できるが，ラムダ式の詳細な言語仕様はプログラミング言語ごとに異なる.

プログラミング言語ごとに異なる仕様の 1 つにキャプチャ可能な変数の制限がある．たとえば Java ではラムダ

式がキャプチャできる変数は final 変数および実質的な final 変数に限られる [13]．検査例外の存在はパラメータ化した動作の型の決定に影響を及ぼし，リファクタリング可能なコードクローンをさらに限定する [7]．言語依存性の高いリファクタリングはプログラミング言語ごとに熟慮を要する．

先行研究では Java を対象にした，ラムダ式を用いたコードクローンのリファクタリング手法を提案している [7]．しかし，これ以外にはラムダ式を用いたコードクローンのリファクタリング手法に関する研究は見られない [7], [14]．本稿では，C#を対象にした，ラムダ式を用いたコードクローンのリファクタリング手法を提案する．

3. 提案手法

3.1 提案手法の概要

提案手法の概観を**図 2** に示す．図中の A，B，C および D は何らかのメソッドを表し，x および y は実パラメータを表す．提案手法はプロジェクトおよびタイプ 1，2 あるいは 3 のクローンペアを入力し，メソッドの抽出リファクタリングが施されたプロジェクトを出力する．このリファクタリングによって，プロジェクトにサブルーチンの定義が追加され，コードクローンがサブルーチンの呼び出しに書き換えられる．リファクタリングが不可能な場合はエラーを出力して終了する．

サブルーチンを生成する処理は 5 つのステップに分けられる．

(1) 一致する文のマッピング
(2) 変数のマッピング
(3) 出口ブロックの検出
(4) パラメータの決定
(5) コード生成

次節より，各ステップの詳細について述べる．

3.2 一致する文のマッピング

このステップでは，入力されたクローンペアからコードクローン間で互いに類似する文の組を発見する．図 2 では，互いに一致した A と C の呼び出し文をそれぞれ組にする操作がこのステップに相当する．サブルーチンの本体には，両方のコードクローンにともに含まれる文と，ラムダ式の呼び出しが含まれる．このステップは各コードクローンに含まれる文のそれぞれが生成されるサブルーチン定義にそのまま含まれるか，ラムダ式に含まれるか決定するステップである．ラムダ式に含まれる文は可能な限り少なくするのが望ましく，類似する文がコードクローン内に出現する順番は双方のコードクローン間で前後してはならない．

この制約を満たすために，最長共通部分列（LCS）のアルゴリズムを適用する．メソッドは一連の処理をまとめたものであり，0 個以上の文の列からなる．提案手法ではコードクローンを文の列と見なし，LCS を求める．LCS アルゴリズム内で文の等価性を判定するため，次に示す文の比較関数を使用する．

文の比較関数

文の比較のために，識別子名にマスクを施し，正規化する．パラメータ化で差異を吸収できる型名，変数名および定数値を，TYPE や VARIABLE のように予約語と重複しない名前に書き換える．空白文字や改行は事前に定めたルールに沿って挿入し，空白文字等による差異を取り除く．また，for 文のような内部に文を含む制御構造は内部の文どうしでさらに LCS を求める．以上のとおり正規化を施した文で単純な文字列の比較をすると，サブルーチンに直接含めるべき文の際長共通部分列が得られる．

コードクローンのソースコードだけではコードクローン内のトークンが変数名なのか型名なのか区別できない場合がある．区別できない例の 1 つにフィールドへのアクセスと定数へのアクセスがある．何らかのクラスのインスタンス instance のフィールド Field へのアクセスは instance.Field と書かれる．一方，列挙型 SampleEnum の定数 Value へのアクセスは SampleEnum.Value と書かれ，これらの構造は構文木上では識別子名の違いを除いて完全に一致する．インスタンスが代入された変数の名前はコードクローン間で一致している必要はないが，列挙型の定数へのアクセスではクラス名が一致している必要がある．これらのように，構文解析のみでマスクすべきか判断できない識別子のために，意味解析を用いる．意味解析はソースコード中の識別子に関する詳細な情報を提供し，識別子がクラスを表しているのか，変数を表しているのかを明らかにする．

for 文や if 文のように内部に文を含む制御構造はさらに詳細なマッチングを要する．たとえば，条件式を同じくする if 文で，内部の文に大きな差異が含まれる場合，if 文の内部で動作のパラメータ化を施すリファクタリングが考えられる．ここで文自体をパラメータ化するのではなく，文の中の一部をパラメータ化している．このようなリファクタリングのために，互いに類似した if 文の内部の文どうしで再度文のマッピングを行う必要がある．内部に文を含む制御構造は，内部の文を除いて一致しているか調べる．上述の文の比較関数では内部の文を除去して正規化を施す．if 文であれば条件式や else 節の構造は残し，else 節を含む内部の文はすべて削除して正規化する．マッピングによって互いに一致した制御構造は，その後，内部の文どうしで再度マッピングする．

3.3 変数のマッピング

式のパラメータ化のために，文のマッピングステップで得た互いに一致する文どうしで対応する変数および定数をマッピングする．図 2 では，互いに一致した文である C の

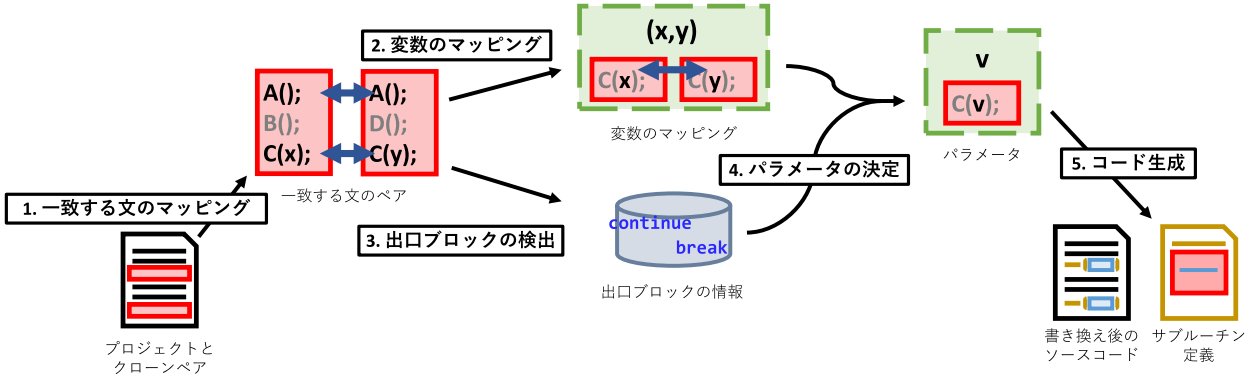


図 2 提案手法の概観
Fig. 2 Overview of our approach.

実パラメータ `x,y` をマッピングする操作がこのステップに相当する．このステップでは，互いに出現箇所が一致する変数アクセスの組を得る．

さらに，この変数アクセスの組をコードクローンごとに変数名が一致する組ごとにグループ分けする．このステップで同一のグループに分類された組には，それぞれ同じパラメータが割り当てられる．

ここで得られたグループの中には，変数の定義がサブルーチンやラムダ式内に含まれるグループも含まれうる．サブルーチンやラムダ式内に定義が含まれる場合，これはパラメータとはいえない．これはサブルーチンやラムダ式のパラメータとして与えるのではなく，サブルーチンやラムダ式の中でローカル変数として定義されるためである．さらに，ラムダ式内で使用する変数はラムダ式に直接キャプチャさせられる場合がある．このステップではパラメータとローカル変数，キャプチャさせる変数を区別せずマッピングする．後述するパラメータの決定ステップでパラメータとパラメータ以外の変数を区別する．

3.4 出口ブロックの検出

プログラムの経路を有効グラフで表す制御フローグラフ（CFG）で，最後に実行されるノードを出口ブロックと呼ぶ．本稿ではコード片のうち最後に実行される可能性がある文およびコード片の末尾を出口ブロックと呼ぶ．

サブルーチンにまとめるコード片やラムダ式にまとめるコード片の制御フローは，複数の出口ブロックを持つ場合がある．たとえば，`if` 文があり，内部にメソッドから脱出する `return` 文が含まれるコード片は 2 つの出口ブロックを持つ．一方，メソッド呼び出し文の制御フローは例外の送出を無視すれば単一の出口ブロックを持つ．複数の出口ブロックを持つコード片は単なるメソッド呼び出しに置き換えられない．しかし，サブルーチンの呼び出し側に分岐構造を導入すると，複数の出口ブロックを持つコード片の動作を模倣できる．サブルーチンの戻り値で出口ブロックの種類を表す．戻り値をもとに `if` 文を用いてそれぞれの出口ブロックの挙動を模倣する文を挿入する．

C#のコンパイラは，値を返すメソッドが return 文に到達しない経路を持つとコンパイルエラーを発する．よって，呼び出し部分の分岐構造では不必要な出口ブロックを模倣する文は除去する必要がある．出口ブロックが 2 つの場合，サブルーチンの戻り値の型は真偽値型でよく，分岐構造は `if` 文あるいは `if-else` 文を使用できる．出口ブロックが 3 つ以上であれば `if` 文を入れ子にした `if-else` 文が使用できる．

C#ではコード片の末尾のほか，ジャンプステートメントと呼ばれる `break` 文，`continue` 文，`return` 文，`goto` 文が出口ブロックになりうる [15]．本研究では，ソースコードの可読性を下げる `goto` 文 [16] を除いた 3 種類の文およびコード片の末尾がつくる出口ブロックのみを対象とする．

コード片の出口ブロックを数え上げるために意味解析から得られる制御フローを使用する．制御フローの出口ブロックをすべて列挙し，各種ジャンプステートメントが含まれるか，その他の文すなわちコード片の末尾に到達できるか調べる．

他言語への適用可能性

C#を対象にする場合，出口ブロックは最大で 4 つに限られる．模倣すべき出口ブロックの数が定数に収まらない場合，出口ブロックの模倣はさらに困難になる．C#においても，`goto` 文は出口ブロックの数を無制限に増やす文である．`goto` 文は指定したラベルの位置に制御を移す．このラベルは複数作成できるため，出口ブロックの数はラベルの数だけ増加する．

Java および JavaScript には `goto` 文は存在しないが，ラベル付き `break` 文とラベル付き `continue` 文が使用できる．これらのラベル付き文はラベルの付いた `for` 文や `while` 文等のループを脱出する，あるいはループの先頭に戻るために使用できる．このため，ラベルの数だけ出口ブロックの数が増加する．ラベル付き文の出口ブロックを模倣するには，コード片の出口ブロックの数え上げや複数の出口ブロックを模倣する呼び出し部分のコード生成に改良が必要

である．

3.5 パラメータの決定

抽出したパラメータには型が必要である．このステップではパラメータの型を決定し，サブルーチンやラムダ式の戻り値とパラメータリストを構築する．変数のマッピングステップで得たグループ分けされた変数アクセスの組と，出口ブロックの検出ステップの検出結果を使用する．図 2 では，変数のマッピングステップで得た組 (x,y) からサブルーチンのパラメータ v を生成している．

パラメータ化したラムダ式はメソッドを代入できる型であるデリゲート型の引数としてサブルーチンに渡す．デリゲート型はメソッドの戻り値の型とパラメータリストを指定する．サブルーチンで使用するラムダ式の戻り値とパラメータリストに一致するデリゲート型を定義する．

パラメータ化した式は，変数のマッピングで得たグループから抽出する．このステップでパラメータにするべきグループとそれ以外のグループに分類する．パラメータにならない変数ペアのグループは，変数がサブルーチンやラムダ式内で定義されるローカル変数と，サブルーチンの外部で定義され，ラムダ式の内部でのみ使用されるラムダ式にキャプチャさせる変数のいずれかにさらに分類される．いずれの場合でも変数の定義位置とコードクローンの範囲および動作のパラメータ化によってラムダ式に抽出される範囲を比較すると分類が可能である．

パラメータ修飾子

C#でメソッドの引数は，原則として値渡しされる．値渡しの場合，パラメータを通してメソッドの呼び出し元の変数に値を再代入できない．しかし，パラメータに `ref` 修飾子を付与するとパラメータは参照として渡されるようになり，メソッド呼び出し元の変数への値を再代入が可能になる．`ref` 修飾子はすでに初期化された変数のみ受け取れる．`ref` 修飾子の代わりに `out` 修飾子を付与すると初期化されていない変数も参照渡しで受け取れる．ただし，`out` 修飾子はメソッドを脱出するまでに必ず初期化が完了している必要がある．適切なパラメータ修飾子を用いると，元のソースコードの形を大きく変えずに複数の再代入可能なパラメータを持つサブルーチンを生成できる．

パラメータ修飾子の制約

C#は `async`，`await` キーワードを用いて非同期タスクを記述できる．非同期タスクを対象にメソッドの抽出リファクタリングを施すとき，サブルーチンも非同期タスクにしなければならない場合がある．しかし，C#は，非同期タスクの引数ではパラメータ修飾子の使用を許可しない．そのほか，C#は反復子[*1]を記述する反復子メソッドを提供している．反復子メソッド内でもパラメータ修飾子の使用が許可されていない．このほか，ローカル関数や入れ子になったラムダ式等の内部ではパラメータ修飾子の付いた変数の使用が許可されていない．よって本研究では，パラメータ修飾子が使用できないコンテキスト下にあるコードクローンは，リファクタリング不可能と判断すべきコードクローンと見なすようにした．

[*1] Python 等ではジェネレータと呼ばれる．

表 1 提案手法が出力するエラーの一覧
Table 1 Tables of errors sent by the proposed approach.

エラー記号	エラーの内容
E_1	プロジェクトを開けない．
E_2	文のマッピングで文がマッチしない．
E_3	`ref` 変数や `out` 変数は ラムダ式でキャプチャできない．
E_4	パラメータの型が決定できない．
E_5	コードクローン中にインスタンスメソッドへの アクセスが含まれる．
E_6	コードクローンが反復子メソッドや 非同期タスク内に存在する．
E_7	ハンドルされない例外をキャッチして終了した．

3.6 コード生成

このステップではプロジェクトにサブルーチンを追加し，コードクローンをサブルーチンの呼び出しに置き換える．サブルーチンの定義はクローンペアの一方のコード片から生成する．

サブルーチンは新たに追加する静的クラス内に配置する．新たに追加する静的クラスや戻り値のための列挙型，デリゲート型はすべて新たに作成したソースファイル内に記述する．既存のソースファイルはコードクローンの部分のみ，サブルーチンの呼び出しに書き換える．

3.7 エラー出力

各ステップでリファクタリング不可能と判断された場合，エラーを出力して終了する．エラーの一覧を**表 1** に示す．提案手法は未分類なエラーを含む 7 種類のエラーを出力する．以後，各エラーは簡単のため表中のエラー記号を用いて表記する．

3.8 実装

筆者は提案手法に沿ってリファクタリングを施すサブルーチン生成器を C#で実装した．提案手法は C#のソースコードの構文木と意味解析を必要とする．Microsoft は C#で書かれたプログラムの構文解析および意味解析を提供するライブラリである roslyn を公開している [17]．本研究では，構文解析および意味解析のために roslyn を使用した．

表 2 プロジェクトごとのリファクタリングされたクローンペアの割合とエラーの内訳
Table 2 Detail of refactoring code clones and errors by project.

	S	E_1	E_2	E_3	E_4	E_5	E_6	E_7	総クローンペア数	リファクタリング率
プロジェクト 1	0	0	1	0	0	0	0	0	1	0.0%
プロジェクト 2	0	0	0	0	9	0	0	0	9	0.0%
プロジェクト 3	0	0	0	0	0	1	101	0	102	0.0%
プロジェクト 4	123	3	71	0	25	10	0	2	236	52.1%
プロジェクト 5	7	0	0	1	5	0	46	1	60	11.7%
プロジェクト 6	1	0	0	0	0	0	1	0	2	50.0%
プロジェクト 7	22	0	11	0	1	0	0	0	34	64.7%
プロジェクト 8	313	10	375	15	20	177	1	9	920	34.0%
プロジェクト 9	17	2	0	0	1	7	24	0	51	33.3%
プロジェクト 10	22	0	13	26	4	25	0	0	90	24.4%
プロジェクト 11	36	5	1	0	1	7	116	5	171	21.0%
プロジェクト 12	1	0	0	0	1	0	7	0	9	11.1%
プロジェクト 13	6	0	0	0	22	2	0	0	30	20.0%
プロジェクト 14	0	0	0	0	1	0	0	0	1	0.0%
プロジェクト 15	8	0	4	0	0	18	0	0	30	26.7%
プロジェクト 16	193	0	10	0	33	3	3	6	248	77.8%
プロジェクト 17	1	0	1	0	0	0	0	2	4	25.0%
プロジェクト 18	3	0	0	0	2	0	0	0	5	60.0%
プロジェクト 19	55	0	1	56	2	9	1	1	125	77.8%
プロジェクト 20	0	0	0	0	0	0	0	63	63	0.0%
プロジェクト 21	14	0	0	0	0	1	2	0	17	82.4%
プロジェクト 22	28	0	24	1	25	3	0	1	82	32.2%
プロジェクト 23	4	0	1	0	0	1	5	1	12	33.3%
プロジェクト 24	5	0	0	0	0	11	3	0	19	26.3%
プロジェクト 25	6	1	1	0	1	0	16	0	25	24.0%
プロジェクト 26	57	16	6	2	4	28	13	12	138	41.3%
プロジェクト 27	43	12	14	18	2	6	21	16	132	32.6%
プロジェクト 28	50	0	0	8	0	6	36	0	100	50.0%
エラー別合計	1,015	49	534	127	159	315	396	119	2,714	37.4%

4. 評価

提案するサブルーチン生成器の適用性を評価する．実際のプロジェクトのコードクローンに提案手法でサブルーチンを生成し，提案手法を適用する．

4.1 評価手法

提案手法によるリファクタリングの正確性は，プロジェクトに付属するテストの実行結果をもとに評価する．リファクタリングの前後でプロジェクトに付属するテストを実行する．リファクタリングの前後でテストの実行結果が変化したならば，リファクタリングは失敗したと分かる．一方，リファクタリングの前後でテストの実行結果が一致してもリファクタリングが成功したとはいい切れない．リファクタリングによる変更箇所がテストされているとは限らないためである．リファクタリングによる変更箇所がテストされているか調べるために，恣意的に誤ったリファクタリングを施したプロジェクトを作成する．誤ったリファクタリングの前後でテストの実行結果が変化したならば，リファクタリング箇所がテストされていると分かる．この評価のために自動で誤ったリファクタリングを施すサブルーチン生成器が必要である．誤ったサブルーチン生成器は提案手法の一部を改変して作成する．具体的に，コード生成のステップで生成するサブルーチンが実行する処理を例外を投げる処理に変更する．

元のプロジェクトとリファクタリング後のプロジェクト，誤ったリファクタリング後のプロジェクトの 3 つを使用し，それぞれに対するテストの実行結果を比較してリファクタリングが成功したか判断する．

また，提案手法は，文のマッピングで一致する文をより多く発見するほどより多くのソースコードを集約できる．リファクタリングによってソースコードをどの程度集約したか調べるため，リファクタリングされたコードクローンの呼び出し部分の長さとサブルーチン定義の長さとの比を算出した．

表 3 テストによって判断されたプロジェクトごとのリファクタリング成否
Table 3 Refactoring success or failure for each project determined by testing.

	R_1	R_2	R_3	R_4	S	リファクタリング成功率	ビルド成功率
プロジェクト 4	2 (2)	121 (69)	0	0	123	1.6%	100.0%
プロジェクト 5	4 (4)	0 (0)	0	3	7	57.1%	57.1%
プロジェクト 6	0 (0)	1 (0)	0	0	1	0.0%	100.0%
プロジェクト 7	0 (0)	14 (8)	2	6	22	0.0%	63.6%
プロジェクト 8	190 (144)	2 (2)	1	120	313	60.7%	61.3%
プロジェクト 9	0 (0)	9 (4)	4	4	17	0.0%	52.9%
プロジェクト 10	6 (6)	10 (9)	0	6	22	27.3%	72.7%
プロジェクト 11	0 (0)	26 (13)	7	3	36	0.0%	72.2%
プロジェクト 12	1 (1)	0 (0)	0	0	1	100.0%	100.0%
プロジェクト 13	2 (0)	0 (0)	0	4	6	33.3%	33.3%
プロジェクト 15	6 (5)	0 (0)	0	2	8	75.0%	75.5%
プロジェクト 16	4 (4)	106 (91)	0	83	193	2.0%	57.0%
プロジェクト 17	1 (0)	0 (0)	0	0	1	100.0%	100.0%
プロジェクト 18	0 (0)	3 (3)	0	0	3	0.0%	100.0%
プロジェクト 19	36 (26)	0 (0)	0	19	55	65.5%	65.5%
プロジェクト 21	0 (0)	6 (5)	0	8	14	0.0%	42.9%
プロジェクト 22	0 (0)	20 (8)	1	7	28	0.0%	71.4%
プロジェクト 23	2 (2)	0 (0)	0	2	4	50.0%	50.0%
プロジェクト 24	1 (0)	0 (0)	0	4	5	20.0%	20.0%
プロジェクト 25	0 (0)	0 (0)	0	6	6	0.0%	0.0%
プロジェクト 26	14 (12)	4 (4)	0	39	57	24.6%	31.6%
プロジェクト 27	0 (0)	15 (11)	15	13	43	0.0%	34.9%
プロジェクト 28	46 (44)	0 (0)	0	4	50	92.0%	92.0%
結果別合計	315 (250)	337 (227)	30	333	1,015	31.0%	62.3%

4.1.1 評価対象

筆者は GitHub で C#のプロジェクトを収集した．スター数が多い順にプロジェクトを収集し，実行可能で安定なテストが付属し，1 つ以上のクローンペアを含むプロジェクトのみに絞り込んだ．安定なテストとは，何度実行しても実行結果が変わらないテストをいう．テストの安定性を確認するため，あらかじめ元のプロジェクトでテストを 4 回実行し，テスト結果を比較した．4 回のテスト結果が 1 つでも一致しなければ，評価対象から除外した．

提案手法は入力にクローンペアを必要とする．本研究では，タイプ 1, タイプ 2 およびタイプ 3 のコードクローンを検出するコードクローン検出器 NiCad [18] を用いて得られたクローンペアを使用した．コードクローン検出にあたって，検出するコードクローンの行数の閾値は 10 行，タイプ 3 における類似度の閾値は 0.3 に設定した．

対象のプロジェクトは 28 件，クローンペアは合計で 2,714 件が検出された．

4.1.2 評価結果

プロジェクトごとに出力されたエラーの内訳を**表 2** に示す．S は提案手法がリファクタリング可能と判断し，実際にリファクタリングを施したクローンペアの数を示す．E_1 から E_7 はそれぞれのエラーを出力して終了したクローンペアの数を示す．すべてのクローンペアでエラーが出力されたプロジェクトには下線を引いている．リファクタリング率は全クローンペアのうち，エラーを出力せず，リファクタリングが施されたクローンペアの割合を示す．全クローンペアの 37.4%にあたる 1,015 件のクローンペアでリファクタリングが施された．

続いて，リファクタリングが施されたクローンペアについて，テストの実行結果をもとに R_1，R_2，R_3 と R_4 の 4 種類に分類する．すべてのクローンペアでリファクタリングが施されなかったプロジェクトは表から除外している．

R_1 リファクタリング前後でテストの実行結果が一致し，かつ誤ったリファクタリングの前後でテストの実行結果が変化したクローンペア．正確なリファクタリングが確認されたクローンペアが分類される．

R_2 リファクタリング前後と誤ったリファクタリングの前後の双方でテストの実行結果が一致したクローンペア．ビルドには成功したが，テストの不足によって正確なリファクタリングが確認できなかったクローンペアが分類される．

R_3 リファクタリングの前後でテストの実行結果が一致せず，誤ったリファクタリングの前後でテストの実行結果が一致したクローンペア．テストが不足しかつサブルーチンのシグネチャ部分に構文的な問題が含まれないクローンペアが分類される．誤ったリファクタリン

グとリファクタリングの違いはサブルーチンのボディのみであるため，サブルーチンのボディに構文的な問題が含まれると推測できる．

R_4 リファクタリングの前後と誤ったリファクタリングの前後の双方でテストの実行結果が一致しないクローンペア．テストは網羅されているが正確なリファクタリングができなかったケースのほか，構文的な問題が発生し，誤ったリファクタリングを施したプロジェクトでもビルドができなかったケースでもここに分類される．

クローンペアごとのテストの実行結果を**表 3** に示す．R_1 と R_2 に分類されるクローンペアについて，うちラムダ式による動作のパラメータ化を用いてリファクタリングされているクローンペアの数を括弧で囲って記した．リファクタリングが施されたクローンペアの 31.0%にあたる 315 件のクローンペアで正確なリファクタリングが確認された．また，リファクタリングが施されたクローンペアの 62.3%にあたる 652($= R_1 + R_2$) 件のクローンペアでプロジェクトのビルドまで成功した．正確なリファクタリングが確認されたクローンペアの 79.4%にあたる 250 件のクローンペアでラムダ式による動作のパラメータ化が用いられていた．

続いて，実際にリファクタリングが施され，テストによって正確さが確認された具体例を示す．いずれの具体例も，簡単のため型名を簡潔にしており，インデントを調整している．

図 3 は 2 つのパラメータにパラメータ修飾子が用いられたリファクタリングの例である．ハイライトされた部分は互いに一致する部分を表す．リファクタリング前後のソースコードは，メソッド本体の一部をそれぞれ抜粋している．サブルーチンは外部で参照される 2 つの変数に新たな値を代入する．この変数の一方はサブルーチンに抽出したコード片の内部で宣言され，もう一方は外部ですでに宣言されている．サブルーチンに抽出したコード片の内部で宣言される変数には `out` 修飾子が付され，コード片内の変数宣言文は代入文に変換されている．外部で宣言される変数には `ref` 修飾子が付されている．パラメータ修飾子を使用して複数の出力をもつサブルーチンを実現している．

図 4 はラムダ式を用いた動作のパラメータ化を適用した例である．オレンジ色にハイライトされた部分は互いに一致する部分，緑と青色にハイライトされた部分はコードクローン中の互いに異なる部分を表す．実際にはソースコードはより長いが，ここでは一部を省略している．互いに共通した処理がサブルーチンに抽出され，異なる部分がラムダ式に抽出されている様子が確認できる．

クローンペアのタイプ別内訳およびリファクタリング成否を**表 4** に示す．ただし，E は E_1 から E_7 までのすべてのエラーの合計を表す．タイプ 3 のクローンペアが全クローンペアの 84.8%を占め，タイプ 1 のクローンペアは

```
1 int bits = input.AvailableBits;
2 lookahead = input.PeekBits(bits);
3 symbol = tree[lookahead];
4 if (symbol >= 0 && (symbol & 15) <= bits)
5 {
```

```
1 int bits = input.AvailableBits;
2 lookahead = input.PeekBits(bits);
3 symbol = tree[subtree | (lookahead >> 9)];
4 if ((symbol & 15) <= bits)
5 {
```

リファクタリング前のソースコード．

```
1 Extracted(out var bits, input, ref lookahead);
2 symbol = tree[lookahead];
3 if (symbol >= 0 && (symbol & 15) <= bits)
4 {
```

```
1 Extracted(out var bits, input, ref lookahead);
2 symbol = tree[subtree | (lookahead >> 9)];
3 if ((symbol & 15) <= bits)
4 {
```

リファクタリング後のソースコード．

```
1 static void Extracted(out int v1, Input v2, ref int
      v3)
2 {
3   v1= v2.AvailableBits;
4   v3 = v2.PeekBits(v1);
5 }
```

生成されたサブルーチン．

図 3 複数の出力を持つリファクタリングの具体例
Fig. 3 A real example of refactoring with multiple outputs.

1.9%であった．

R_1 に分類されたすべてのクローンペアについて，リファクタリング前後のパフォーマンスの変化を調査した．具体的には，リファクタリング箇所をカバーするテストを実行し，メモリ使用量と実行時間を計測した．しかし，いずれのクローンペアにおいてもランタイムに起因する実行ごとの誤差の方が大きく，リファクタリングの前後で明確な差は見られなかった．

5. 考察

表 2 によると，提案手法を用いてコードクローンをリファクタリングできる程度はプロジェクトによって大きく異なっている．たとえば反復子メソッドや非同期タスク内のコードクローンをリファクタリングしようとすると発生する E_6 は，プロジェクトに反復子メソッドや非同期タスクが含まれなければ当然ながら発生しない．たとえば，ほとんどのクローンペアで E_6 を出力したプロジェクト 3 は WEB アプリケーションに対して実行できるリクエストのレートを制御するミドルウェアである．HTTP リクエストを扱うソフトウェアは非同期プログラミングを用いる典型

```
string tempFile = Helper.GetTempFilePath(filePath);
try {
  var workbook = new Workbook(tempFile);
  var ws = workbook.Worksheet(1);
  var rngTable = ws.Range("B2:F6");
  rngTable.Transpose(Options.MoveCells);
  workbook.SaveAs(filePath);
} finally {
  if (File.Exists(tempFile)) File.Delete(tempFile);
}
```

```
string tempFile = Helper.GetTempFilePath(filePath);
try {
  var workbook = new Workbook(tempFile);
  var ws = GetWorksheet(workbook);
  ws.Row(1).InsertRowsAbove(2);
  ws.Column(1).InsertColumnsBefore(2);
  workbook.SaveAs(filePath);
} finally {
  if (File.Exists(tempFile)) File.Delete(tempFile);
}
```

リファクタリング前のソースコード.

```
Extracted(filePath, (Workbook v3)=>{
  var v4= v3.Worksheet(1);
  var rngTable = v4.Range("B2:F6");
  rngTable.Transpose(Options.MoveCells);
});
```

```
Extracted(filePath, (Workbook v3)=>{
  var v4= GetWorksheet(v3);
  v4.Row(1).InsertRowsAbove(2);
  v4.Column(1).InsertColumnsBefore(2);
});
```

リファクタリング後のソースコード.

```
delegate void LambdaDelegate1(Workbook v3);
static void Extracted(string v2, LambdaDelegate1
    lambda1) {
  string v1 = Helper.GetTempFilePath(v2);
  try {
    var v3 = new Workbook(v1);
    lambda1(v3);
    v3.SaveAs(v2);
  } finally {
    if (File.Exists(v1)) File.Delete(v1);
  }
}
```

生成されたサブルーチン.

図 4 動作のパラメータ化を用いたリファクタリングの具体例
Fig. 4 A real example of refactoring with behavior parameterization.

的な例の 1 つといえる．このように，プロジェクトの性質によってリファクタリングの可否は大きく異なる．

$R_{1,L}$ および $R_{2,L}$ に注目すると，半数以上のクローンペアで動作のパラメータ化を用いている．動作のパラメータ化が実際のリファクタリングに対して有効であると分かる．ただし本研究では，実際には副作用による動作の変化を起こさないクローンペアを厳密に調べていない．副作用による動作の変化を厳密にチェックする手法 [9] では，本研究での動作のパラメータ化を過剰と判断する可能性がある．また，互いに異なる文をコードクローンの前後に移動する方法 [9] を用いれば，動作のパラメータ化を使わずにリファクタリングできる可能性もある．

表 3 によると，プロジェクトによってリファクタリングが成功する割合やビルドを通過する割合にも大きな差が見られる．プロジェクトのクローンペアに含まれる構文の種類に強く依存しており，提案手法の実装が一部の構文に対応しきれていない様子がうかがえる．

表 4 より，タイプ 2 のクローンペアであるにもかかわらず，動作のパラメータ化を用いたリファクタリングが施されているケースが確認できる．タイプ 2 のクローンペアは識別子名の差異を許容し，文単位の差異は許容しないため，直感的には動作のパラメータ化は不要と考えられる．しかし，識別子名の差異を許容すると，互いに異なるメソッドを呼び出すプログラムであってもこれらは一致したプログラムと見なされてしまう．互いに異なるメソッドの呼び出しはサブルーチンにまとめられないため，ラムダ式による動作のパラメータ化が必要になる．

表 2 によると，62.6%のコードクローンがリファクタリング不可能と判断された．さらに表 3 によれば 35.8%のコードクローンでリファクタリングに失敗している．これらの判断および失敗の原因を定性的に調査した．

表 4 クローンペアのタイプ別リファクタリング成否
Table 4 Refactoring success or failure for each type.

タイプ	R_1	R_2	R_3	R_4	E	合計
タイプ 1	4 (0)	12 (0)	4	2	29	51
タイプ 2	64 (42)	68 (55)	5	39	186	362
タイプ 3	247 (208)	257 (172)	21	292	1,484	2,301
合計	315 (250)	337 (227)	30	333	1,699	2,714

5.1 リファクタリング不可能なコードクローン

提案手法がリファクタリング不可能と判断した場合に発生する E_1 から E_7 のエラーの原因をそれぞれ定性的に調査した．

5.1.1 E_1 プロジェクトを開けないコードクローン

関連するプロジェクトを発見できなかった場合に発生していた．どのプロジェクトでもコンパイルされないサンプルコードからコードクローンを検出しているケースも見られた．

5.1.2 E_2 文のマッピングで文がマッチしないコードクローン

NiCad がコードクローンとして検出したものの，提案手法が一致部分を発見できなかった場合に発生していた．たとえば `case` 節の数が異なる `switch` 文は NiCad ではよく

類似した文と見なされる場合があるが，提案手法では case 節の数が異なる switch 文は異なる文と見なす．一致あるいは類似と見なす基準が異なるためにこのエラーが発生していた．同様のエラーは条件式が異なる if 文等を含むコードクローンでも見られた．提案手法は条件式が一致しない if 文等は一致した文と見なさないが，NiCad では条件式部分の一致に関係なく内部の文を比較する．

5.1.3 E_3 ラムダ式で ref 変数や out 変数をキャプチャしようとするコードクローン

C#のラムダ式は参照渡しの変数をキャプチャできない．コードクローンの不一致部分，すなわちラムダ式を用いてパラメータ化しようとする部分で参照渡しの変数を用いているコードクローンはリファクタリングできない．提案手法はコードクローンの特性を調べて適切にこのエラーを発生させていた．

5.1.4 E_4 パラメータの型が決定できないコードクローン

提案手法は，パラメータの決定ステップでパラメータの型を決定しようとする．しかし，両方のコード片でパラメータの型がまったく異なる場合，適切な共通の親を発見できない場合がある．提案手法はコードクローンの特性を調べて適切にこのエラーを発生させていた．

5.1.5 E_5 インスタンスメソッドへのアクセスが含まれるコードクローン

提案手法は，サブルーチンを新たな静的クラス内に定義する．このクラスはコードクローンが含まれるクラスとは異なるため，サブルーチン内では，コードクローン内で呼び出されていたインスタンスメソッドを呼び出せない．提案手法はコードクローンの特性を調べて適切にこのエラーを発生させていた．インスタンスメソッドへのアクセスを可能にするには，コードクローンを含むクラスと同じクラスや共通の親クラスにサブルーチンを追加する必要がある．

5.1.6 E_6 反復子メソッドや非同期タスク内のコードクローン

反復子メソッドや非同期タスク内ではパラメータ修飾子を原則として使用できない．提案手法はパラメータ修飾子を用いるため，反復子メソッドや非同期タスク内のコードクローンはリファクタリングの対象外とした．提案手法はコードクローンの特性を調べて適切にこのエラーを発生させていた．

5.1.7 E_7 その他の例外が発生したコードクローン

表 2 によると，119 件のコードクローンで意図せぬ例外による E_7 が発生している．例外は 2 種類確認された．発生した例外ごとに分類したうえで原因を分析した．

まず 1 つ目に，構文解析および意味解析を担うライブラリが解析中に例外を発するケースが見られた．すべてのケースで失敗しているプロジェクト 20 は，この問題のためにリファクタリングに失敗していた．一方でこれらはプロジェクトのビルドには成功しており，リファクタリング前のプロジェクトは問題なくテストを通過した．

2 つ目に，文の正規化に失敗するケースが見られた．一致する文のマッピングステップでは，型名や変数名，定数値をマスクし，正規化する．この正規化の処理中に例外が発生するケースが見られた．本研究の実装では，文の正規化のために構文木を改変する方法を選択した．この構文木の改変作業中に，通常は許可されない改変を試みてしまっていた．正規化は文字列の置換で十分実現できるため，構文木を直接改変しない方法で再実装する解決策が考えられる．

5.2 リファクタリングに失敗したコードクローン

リファクタリング可能と判断され，リファクタリングを施したところビルドに失敗，あるいはテストを通過しなかった R_3 および R_4 のケースについて，その原因を調査した．具体的には，各クローンペアについてリファクタリング後のビルドログから，コンパイルエラーを収集した．さらに，コンパイルエラーごとに原因を定性的に分析した．

リファクタリングに失敗する原因は (1) 誤ってリファクタリング可能と判断されたコードクローン，(2) 誤ったリファクタリングを施してしまうコードクローンの 2 種類に大別された．以降それぞれについて詳しく述べる．

5.3 誤ってリファクタリング可能と判断されたコードクローン

提案手法は各ステップ内でリファクタリング不可能と判断されるとエラーを出力して終了する．しかし，実際には集約が困難なコードクローンであるにもかかわらず，リファクタリング不可能と判断できなかったケースを発見した．本節では，リファクタリング不可能と判断できなかったケースを定性的に分析する．

5.3.1 アクセシビリティに違反するケース

提案手法はサブルーチンを新たに追加したクラス内に定義する．新たに追加したクラスはコードクローンを含むクラスとは明らかに異なるため，サブルーチンからコードクローン内で呼び出している private な静的メソッドにはアクセスできない．また，コードクローンを含むクラスに定義された private な入れ子クラスやコードクローンを含むクラスの親クラスに定義された protected な入れ子クラスは，サブルーチンを定義したクラスからはアクセスできない．

しかし，提案手法ではコードクローン内で呼び出しているメソッドや使用するクラスのアクセシビリティを考慮していない．メソッドやクラスのアクセシビリティを考慮するか，サブルーチンを定義する場所を変更する必要がある．

5.3.2 オーバーロードした異なるメソッドを呼び出すケース

提案手法はコードクローンの内部で使用しているメソッ

ドにオーバーロードがある場合を考慮できていない．文のマッピングで互いに一致していると判断されたメソッド呼び出し文は，それぞれオーバーロードされた別のメソッドを呼び出している可能性がある．それぞれのメソッド呼び出しで引数に用いられている型の異なる変数を，式のパラメータ化によってパラメータに抽出した場合，パラメータの型にはそれぞれの変数の共通の親が選択される．変数の型が変わると，オーバーロードの解決結果が変化したり，オーバーロードを解決できずコンパイルエラーが発生したりする場合がある．

本研究の評価では，オーバーロードを解決できなくなり，コンパイルエラーが発生するケースが確認された．

5.3.3 実パラメータが正しく指定できないケース

提案手法では，双方のコードクローンで変数の定義箇所が異なるケースについて網羅的に考慮できていなかった．

問題に直接関連する部分のみを抽出した，極度に単純化された例を図 5 に示す．コードクローンを青くハイライトしている．リファクタリング前のソースコードには，2 つのローカル変数 `i`，`j` を定義するメソッドが 2 つある．特に `j` の初期化に注目すると，`Func1` は外部で定義された変数の値を代入しているが，`Func2` ではローカル変数の初期化に別のローカル変数の値を使用している．このコードクローンを集約しようとすると，実パラメータが決定できない問題が発生する．例にあげたソースコードでは，`j` に代入する変数で実パラメータを決定できない．`Func1` では実パラメータには `a` を与えればよい．しかし `Func2` では `j` にローカル変数の `i` を代入している．この `i` はサブルーチンの呼び出し側からは与えられない．式のパラメータ化によって抽出された式がサブルーチンの呼び出し側で使用できない場合，リファクタリングに失敗する．

この問題を解消するには，パラメータの決定ステップに改良を加え，適切にエラーを出力する必要がある．変数ペアのグループを分類する際に，一方がコードクローンの内部で，他方がコードクローンの外部で変数を定義するグループを発見したらリファクタリング不可能と判断し，エラーを出力するべきだと考えられる．

5.4 誤ったリファクタリングを施してしまうコードクローン

提案手法の実装の不備によって，正しくリファクタリングを施せなかったケースが見られた．評価結果から発見された，実装の不備について述べる．

5.4.1 プリプロセッサを含むコードクローン

本稿で用いた実装では，プリプロセッサを適切に考慮できていない．C#におけるプリプロセッサとは，コンパイラに特別な指示を出すために用いられる構文である．

5.4.2 型名を識別子名と誤認してしまうコードクローン

C#において，型は変数に代入できず，サブルーチンの引数には抽出できない．識別子と型を区別する必要があるが，構文解析だけでは識別子名なのか型名なのか分からない文法がある．意味解析の情報を正しく活用できず，型名を識別子名と誤認してパラメータ化してしまうケースが見られた．

```
1  int a, b;
2
3  void Func1()
4  {
5    int i = ComplexCalculation();
6    int j = a;
7  }
8
9  void Func2()
10 {
11   int i = ComplexCalculation();
12   int j = i;
13 }
```

リファクタリング前のソースコード．

```
1  int a;
2
3  void Func1()
4  {
5     Subroutine(a);
6  }
7
8  void Func2()
9  {
10   Subroutine(?);
11 }
12
13 void Subroutine(int v1)
14 {
15   int i = ComplexCalculation();
16   int j = v1;
17 }
```

リファクタリング後の誤ったソースコード．

図 5 実パラメータの決定に失敗する例
Fig. 5 Example of failure to determine an actual parameter.

5.4.3 適切なパラメータ修飾子を決定できないパラメータ

宣言するものの，初期化しない変数がサブルーチン内に出現するとき，提案手法はパラメータ修飾子を適切に決定できなかった．サブルーチン内で宣言したローカル変数がサブルーチンを脱出した後に再度参照されるならば，`out` 修飾子を付すのが適当だが，`out` 修飾子が付されたパラメータは，ラムダ式脱出時に値が割り当てられている必要がある．宣言するだけで初期化されないローカル変数がサブルーチンに抽出されるケースでコンパイルエラーが発生した．この場合に適切なパラメータ修飾子は存在せず，問題を解決するにはローカル変数の宣言場所を変更する必要がある．

6. 妥当性の脅威

6.1 構成概念妥当性

6.1.1 評価対象のテストカバレッジ

本研究の評価は，リファクタリング対象のプロジェクトに付属したテストのテストカバレッジに大きく依存している．テストカバレッジが上昇すれば，R_2 にあたるコードクローンは R_1 あるいは R_3 のいずれかに分類されるようになり，評価結果が変化する．R_2 に分類されたコードクローンは，リファクタリングの成否がまだ判断できていないコードクローンであることに注意されたい．

6.2 内的妥当性

6.2.1 NiCad の検出結果の妥当性

本研究で評価対象のプロジェクトからコードクローンを検出するために NiCad を用いた．この評価結果は NiCad 自体のコードクローンの検出性能，および NiCad に与える行数や類似度の閾値設定に大きく影響を受ける．

6.2.2 C#のバージョンによる影響

提案手法は，C#のラムダ式およびパラメータ修飾子の機能を活用している．リファクタリングの成否は使用している C#のバージョンに影響される．ラムダ式は 2007 年にリリースされた C#3.0，パラメータ修飾子はすべての C#で使用できる機能である．本研究で評価対象に選択したソフトウェアはすべて C#3.0 以降のバージョンを使用している．

6.2.3 実装の妥当性

提案手法の実装は筆者が自ら手掛けた．複数のテストケースでリファクタリングが正しく施されるか確認しているが，あらゆるソースコードに対する網羅的な確認はできていない．考察の章にあげたケース以外にも，リファクタリングを正しく施せないケースが存在する可能性がある．

6.3 外的妥当性

6.3.1 評価対象プロジェクトの妥当性

本研究の評価では GitHub 上のスター数が多いプロジェクトを対象にしている．この評価で得られた知見は，他のプロジェクトに対して適用できない可能性がある．

6.3.2 コードクローン検出器による変化

本研究の評価では，提案手法に入力するクローンペアに NiCad で検出されたクローンペアを用いた．コードクローン検出手法は複数提案されている．手法の違いによって検出されるコードクローンは変化する．タイプ 1，タイプ 2 およびタイプ 3 のコードクローンを検出するコードクローン検出器には NiCad のほかに CCFinder [19] や Simian [20] 等があげられる．異なるコードクローン検出器を用いると異なる結果が得られる可能性がある．

6.3.3 クローンセットへの適用可能性

本研究はクローンペアを対象にリファクタリングを試みた．しかし実際のプロジェクトには，3 つ以上の互いに類似したコードクローン，すなわちクローンセットが発生する可能性がある．クローンセットを集約して 1 つのサブルーチンを抽出したいケースが考えられる．提案手法をクローンセットのために拡張するにあたって，LCS アルゴリズムを用いた文のマッピングが主な障壁になる．任意の個数の系列から LCS を発見する問題は NP 困難であるためである．

7. おわりに

本稿では，ラムダ式による動作のパラメータ化を用いてコードクローンを集約するリファクタリング手法を提案した．また，提案手法を実際のプロジェクトのコードクローンに対して適用し，どの程度コードクローンを集約できるか評価した．提案手法はクローンペア全体のうち 62.6%がリファクタリング不可能だと判断し，リファクタリング可能と判断した 37.4%のクローンペアのうち 31.0%，つまり全体の 11.6%のクローンペアで正しくリファクタリングを施せた．さらに，全体の 24.0%のクローンペアがビルドを正常に通過した．実際のクローンペアにリファクタリングを施した結果を観察し，提案手法の課題点を考察した．

提案手法はパラメータの名前を `v1` のような意味のない単語にしてしまう．変数名の適切な命名はソースコードの可読性を向上させるために重要な要素である．変数名のリファクタリング [21] との併用による可読性の向上は今後の課題の 1 つにあげられる．その他の課題は以下があげられる．

- より柔軟な式のパラメータ化によって，文のマッピングを成功しやすくする．
- プリプロセッサを含むソースコードのような未実装部分の処理を実装する．
- エラーのバリエーションを増やし，リファクタリング不可能なケースの見落としを防ぐ．

謝辞 本研究は JSPS 科研費 25K03102，24H00692，23K24823 の助成を受けたものです．

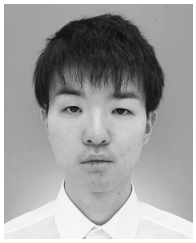
参考文献

[1] 井上克郎，神谷年洋，楠本真二：コードクローン検出法，コンピュータソフトウェア，Vol.18, No.5, pp.529–536（オンライン），DOI: 10.11309/jssst.18.529 (2001).

[2] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone detection using abstract syntax trees, *Proc. International Conference on Software Maintenance* (*Cat. No.98CB36272*), pp.368–377, IEEE (online), DOI: 10.1109/ICSM.1998.738528 (1998).

[3] Mondal, M., Roy, C.K. and Schneider, K.A.: Automatic Identification of Important Clones for Refactoring and Tracking, *2014 IEEE 14th International Working Con-*

ference on Source Code Analysis and Manipulation, pp.11–20, IEEE (online), DOI: 10.1109/SCAM.2014.11 (2014).
[4] Lozano, A. and Wermelinger, M.: Assessing the effect of clones on changeability, *2008 IEEE International Conference on Software Maintenance*, pp.227–236, IEEE (online), DOI: 10.1109/ICSM.2008.4658071 (2008).
[5] 肥後芳樹，吉田則裕：コードクローンを対象としたリファクタリング，コンピュータソフトウェア，Vol.28, No.4, pp.4_43–4_56（オンライン），DOI: 10.11309/jssst.28.4_43 (2011).
[6] Yoshida, N., Numata, S., Choiz, E. and Inoue, K.: Proactive Clone Recommendation System for Extract Method Refactoring, *2019 IEEE/ACM 3rd International Workshop on Refactoring* (*IWoR*), pp.67–70, IEEE (online), DOI: 10.1109/IWoR.2019.00020 (2019).
[7] Tsantalis, N., Mazinanian, D. and Rostami, S.: Clone Refactoring with Lambda Expressions, *2017 IEEE/ACM 39th International Conference on Software Engineering* (*ICSE*), pp.60–70 (online), DOI: 10.1109/ICSE.2017.14 (2017).
[8] Roy, C. and Cordy, J.: A Survey on Software Clone Detection Research, *School of Computing TR 2007-541*, Vol.541, No.115, pp.3–7 (2007).
[9] Tsantalis, N., Mazinanian, D. and Krishnan, G.P.: Assessing the Refactorability of Software Clones, *IEEE Trans. Software Engineering*, Vol.41, No.11, pp.1055–1090 (online), DOI: 10.1109/TSE.2015.2448531 (2015).
[10] Fraser, G. and Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software, *Proc. 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11*, pp.416–419, Association for Computing Machinery (online), DOI: 10.1145/2025113.2025179 (2011).
[11] Mondal, M., Roy, C.K. and Schneider, K.A.: A Fine-Grained Analysis on the Inconsistent Changes in Code Clones, *2020 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), pp.220–231, IEEE (online), DOI: 10.1109/ICSME46990.2020.00030 (2020).
[12] Robert, T. and Jeff, G.: Increasing clone maintenance support by unifying clone detection and refactoring activities, *Information and Software Technology*, Vol.54, No.12, pp.1297–1307 (online), DOI: 10.1016/j.infsof.2012.06.011 (2012). Special Section on Software Reliability and Security.
[13] Oracle: Java Documentation, Lambda Expressions (2014), available from ⟨https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html⟩ (accessed 2025-02-19).
[14] AlOmar, E.A., Mkaouer, M.W. and Ouni, A.: Behind the Intent of Extract Method Refactoring: A Systematic Literature Review, *IEEE Trans. Software Engineering*, Vol.50, No.4, pp.668–694 (online), DOI: 10.1109/TSE.2023.3345800 (2024).
[15] Microsoft: C#, Jump Statements (2023), available from ⟨https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/jump-statements⟩ (accessed 2025-02-22).
[16] Dijkstra, E.W.: *Notes on structured programming*, pp.1–82, Academic Press Ltd. (1972).
[17] Microsoft: C#, roslyn (2025), available from ⟨https://learn.microsoft.com/ja-jp/dotnet/csharp/roslyn-sdk/⟩ (accessed 2025-07-10).
[18] Cordy, J.R. and Roy, C.K.: The NiCad Clone Detector, *2011 IEEE 19th International Conference on Program Comprehension*, pp.219–220, IEEE (online), DOI: 10.1109/ICPC.2011.26 (2011).
[19] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Software Engineering*, Vol.28, No.7, pp.654–670 (2002).
[20] Quandary Peak Research: Simian (2021), available from ⟨https://simian.quandarypeak.com/⟩ (accessed 2025-05-25).
[21] Liu, H., Wang, Y., Wei, Z., Xu, Y., Wang, J., Li, H., and Ji, R.: RefBERT: A Two-Stage Pre-trained Framework for Automatic Rename Refactoring, *Proc. 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*, pp.740–752, Association for Computing Machinery (online), DOI: 10.1145/3597926.3598092 (2023).



川本 琢人

2024 年名城大学理工学部卒業．現在，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程在学中．リファクタリング支援に関する研究に従事．

肥後 芳樹 （正会員）

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学院情報科学研究科コンピュータサイエンス専攻助教．2015 年同准教授．2022 年同教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援，ソフトウェアリポジトリマイニングおよび自動プログラム修正に関する研究に従事．日本ソフトウェア科学会，IEEE 各会員．