Vol.2011-SE-173 No.7
2011/7/21

コードクローンの特徴に基づくメソッド引き上げリファクタリングパターンの提案

吉岡一樹[†1] 吉田則裕[†2] 徳永将之[†1]
松下誠[†1] 井上克郎[†1]

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．あるコード片に欠陥が含まれていた場合，そのコード片のコードクローンすべてに対し，修正を検討しなければならない．これには大きなコストが必要になる．コードクローンを除去，集約する方法の一つとしてリファクタリングパターンの適用が挙げられる．リファクタリングとは，外部的動作に変更を加えずに内部の構造を理解や修正がしやすいよう変更することである．特に繰り返し行われるものを適用する状況と手順をまとめたものをリファクタリングパターンという．しかし，既存のリファクタリングパターンはコードクローンに対して，詳細に分類していない．そこで，本研究では細かく特徴毎に分類されたコードクローンに対して，詳細なコードクローンのリファクタリングの手順を考案し，パターンとして提案する．適用実験として，コードクローン検出ツールを用いて発見した，オープンソースソフトウェアに存在するコードクローンに対して，被験者にリファクタリングを行ってもらった．提案したパターンと既存のパターンの比較を行い，外部的動作に変更を加えずにメソッドの引き上げを行えることにより妥当性を，提案したパターンが既存のパターンよりも分類された特徴を持つコードクローンに対しての優位性が確認できたために，本手法の有効性を確認できた．

Proposal of sequence of pulling up method of refactoring pattern on code clone classification

Kazuki Yoshioka,[†1] Norihiro Yoshida,[†2]
Masayuki Tokunaga,[†1] Makoto Matsushita[†1]
and Katsuro Inoue[†1]

A code clone is defined as code fragments that are similar to each other. If code fragments including a bug, developers have to consider fixing all code clones corresponding to those code fragments. Simultaneous editing of code clones takes much cost. Refactoring is one of way to merge code clones and remove them. It is a technique altering an internal structure to make easy to understand and modify without changing external behavior. A pattern consist of a repeated situation to apply refactoring and corresponding sequence of procedures is called refactoring pattern. However, existing patterns do not categorize code clones in detail. In this study, we propose a detailed procedure of refactoring a code clone categorized by its feature and propose them as a refactoring pattern. We performed an experiment to evaluate that the proposed pattern does not change external behavior and superior to an existing pattern. The experiment compared an existing pattern to the proposed pattern. The test subjects are performing refactoring code clones on an open source software system. The result of the experiment shows the proposed pattern does not change external behavior, and is superiority to the existing pattern by opinions of test subjects.

1. はじめに

ソフトウェアの保守を困難にしている要因の一つとして，コードクローンが挙げられる．コードクローンとは，ソースコード上に存在する同一，もしくは類似したコード片のことを意味し，コピーペーストやキューの挿入などの定義上簡単な繰り返し処理を書くことにより，プログラムテキスト中に作り込まれる．コードクローンはレガシーシステムに対する変更や拡張においても作り込まれることが多く，実際に約 20 年間保守しながら利用されている，ある大規模ソフトウェアでは約半数のモジュールに何らかのコードクローンが存在していることが確認されている [13]．あるコードにバグが存在した場合，そのコードクローンすべてに修正を検討する必要があるが，近年のソフトウェアは大規模化，複雑化が進んでおり修正作業に伴うコストも非常に大きなものになっている．したがって，コードクローンをパターン化した手順で取り除くことができれば，修正におけるコストを小さくすることができる．コードクローンは Bellon の定義 [2] に基づくと，3 つのタイプに分類することができる．

タイプ 1 空白やタブの有無，括弧の位置などのコーディングスタイルを除いて，完全に一致するコードクローン

タイプ 2 関数名や変数名などのユーザ定義名，変数の型などの一部の予約語のみが異なる

†1 大阪大学
Osaka University
†2 奈良先端科学技術大学院大学
Nara Institute of Science and Technorogy

コードクローン

タイプ 3 タイプ 2 における変更に加えて，文の挿入や削除，変更が行われたことで不一致部分を含むコードクローン

本研究では，タイプ 2 のコードクローンに着目している．コードクローンの修正手段のひとつとして，リファクタリングが挙げられる．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” [4]，[9] であり，コードクローンに対して行えば 1 つにまとめて取り除くことができる．特にリファクタリングはパターンとして書籍やウェブサイト [3] にまとめられている．Fowler はメソッドの抽出，メソッドの引き上げ，フィールドの引き上げ，Templete method の形成など多くのリファクタリングパターンを提案している [4]．しかし，上に挙げた既存の書籍やウェブサイト [3] にはコードクローンの特徴が詳細に分類されていない．例えば，Fowler は書籍 [4] の中でコードクローンの特徴に関して，同一クラスにある，兄弟クラスにある，完全に一致する，似通っている，異なるアルゴリズムである，程度にしか分けていない．リファクタリングの記述に関しても全く同一のコード片しか対象にしていないので詳細に記されていない．パターンの適用条件およびリファクタリング作業の記述が不十分であるために、コードクローンのリファクタリング作業が困難なものになっている．徳永らは，コードクローンを特徴ごとに分類している [14]．7 つの分類観点を定義し，観点毎に分類木を作成し，分類木を組み合わせることでコードクローンのコード片の状況を詳細化している．そして，頻出するコードクローンの特徴からリファクタリングパターンを作成していき，リファクタリングパターンが提案されていない分類をリファクタリングできない分類とすることと提案している．そこで，本研究では徳永らの提案した分類に基づいたリファクタリングパターンの一つを提案する．Fowler が提案しているリファクタリング手法は多くの状況に対応しようとしているために記述が曖昧であり，コード修正の実行者に判断を委ねる部分が多い．特にメソッドの引き上げにおいては，コードが同一であることを想定しているためにコードに差異があっても，“必要に応じて修正を行う” としか書かれていない．また，適用する条件も同じ結果をもたらすメソッドが複数のサブクラスに存在するときとしか示されていない．徳永らの提案した分類を利用することでパターン適用の状況を明白にし，詳細なリファクタリング手順を提案する．提案手法を用いて適用実験を行った結果，リファクタリングの前後で動作が変わらず，かつ詳細化できていることを確かめた．以降，2 節では本研究に関わるコードクローン及びリファクタリングパターンについて説明する．3 節ではリファクタリングパターン提案の手順について説明し，4 節では本手法を用いた適用実験の結果と考察を述べ，5 節では関連研究について述べ，6 節では本研究のまとめと今後の課題について述べる．

2. 背　　景

本研究の研究手法の背景として，問題となるコードクローン，リファクタリングパターンおよびコードクローンの分類について説明する

2.1 コードクローン

コードクローンとは，プログラムテキスト中に存在する，同一，あるいは類似したコード片を意味する．コードクローンはコピーペーストや定型処理等が原因で発生する [15]．コードクローンの存在するプログラムでは，欠陥が見つかったコード片を修正する場合，そのコード片のコードクローンすべてに修正を検討しなければならない．しかし、近年のソフトウェアは大規模化，複雑化の一途をたどっており、このような作業のコストは大きくなっている．

2.1.1 定　　義

コードクローンには様々な検出法が存在し，そのどれもが異なったコードクローンの定義を持つために普遍的な定義は存在しない．Bellon は，コードクローンの相違の度合いに基づいた 3 つの分類を定義している [2].

タイプ 1 空白やタブの有無，括弧の位置などのコーディングスタイルを除き，完全に一致するコードクローン（**図 1** を参照）.

タイプ 2 変数名や関数名などのユーザ定義名，また変数の方などの一部の予約語のみが異なるコードクローン（**図 2** を参照）.

タイプ 3 タイプ 2 における変更に加えて，文の挿入や削除，変更が行われたコードクローン
本研究ではタイプ 2 のコードクローンに着目している．

2.2 リファクタリングパターン

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，ソフトウェアの変化させること” であり，繰り返し行われるリファクタリングはパターンとしてまとめられている．このようなパターンはリファクタリングパターンと呼ばれ，適用を検討すべきソースコードの特徴や手順などが述べられている [4]．Fowler は設計の問題を匂いにたとえ，設計の問題を不吉な匂いに例え，その中でもコードクローンを不吉な匂いの “はえある 1 等賞である” [4] としており，また，Kerievsky は文献 [8] の中で重複

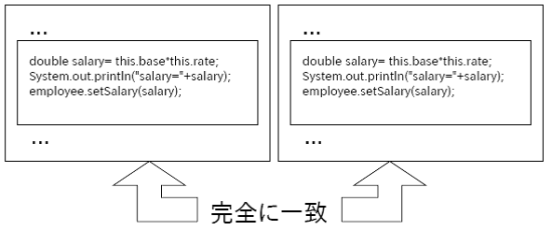

図 1 タイプ 1 クローン
Fig. 1 clone of type1

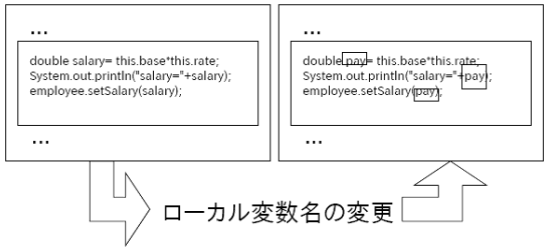

図 2 タイプ 2 クローン
Fig. 2 clone of type2

したコードはソフトウェアの中でもっともよく見つかる刺激臭であると述べている．これを解決するために Fowler は文献 [4] の中でどのようなリファクタリングパターンが役立つかを述べている．

2.2.1 メソッドの抽出

メソッド内のコード片を抽出し，新しいメソッドをして定義するリファクタリングパターンである（**図 3**）．抽出されたコード片は定義されたメソッド呼び出し文に置き換える．もし，抽出したコード片のコードクローンが同じクラス内にある場合，同様にメソッド呼び出し文に置き換えることで，コードクローンを取り除くことができる．メソッドの抽出においては，引数として受け渡す変数，戻り値になる変数を明確に決めなければならず，明確に定まらない場合はメソッドの抽出をすることはできない．メソッド抽出は効果的で頻繁に行われるリファクタリングパターンであるため，自動化の効果は大きく既存の研究においてもメソッド抽出の必要性を評価するメトリクスや [12]，様々な自動化手法やツールが提案されている [10]．

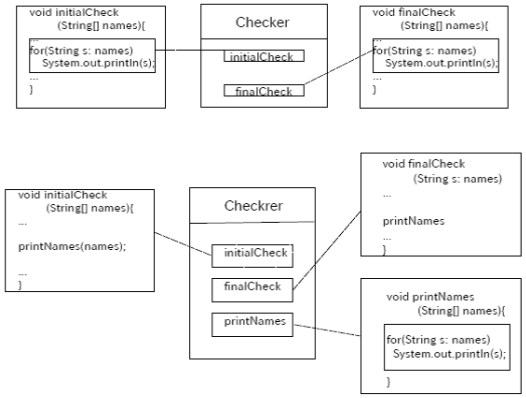

図 3 メソッドの抽出
Fig. 3 extract method

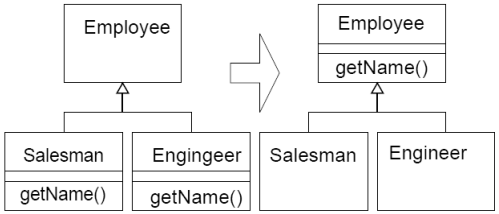

図 4 メソッドの引き上げ
Fig. 4 pull up method

2.2.2 メソッドの引き上げ

子クラスで定義されたメソッドを親クラスへと引き上げるリファクタリングパターンである（**図 4**）．同じ親クラスを持つ兄弟クラスに同じ処理をするメソッドがある場合，親クラスへとそのメソッドを引き上げることでコードクローンを取り除くことができる．メソッドの一部がコードクローンである場合，上記のメソッドの抽出を行い，同一のメソッド

Vol.2011-SE-173 No.7
2011/7/21

3. コードクローンの分類

3.1 コードクローンの位置

コードクローンのリファクタリングについてファウラーは文献 [4] において，先述のメソッドの抽出やメソッドの引き上げなどのリファクタリングパターンが述べられているが，それは重複したコードという，非常に大きい分類でしか語られていない．Fowler が分類しているのは以下の状況である．[4]

- 同一クラス内にある場合
- 兄弟クラス内にあり
  - 完全に一致する場合
  - 似通っている場合
  - 異なるアルゴリズムの場合
- 無関係なクラスにある場合

これは，コードクローンの存在するクラスの継承関係及び，ソースコードのアルゴリズムの類似度にしか注目していない．実際のコードにおいては，様々な要因で抽出ができなくなる，あるいは可能ではあるが修正が難しくて適用ができないという事態が発生しうる．そこで，徳永らはコードクローンの分類について定義してしている [14]．例えば，メソッドの抽出など既存のリファクタリングパターンを適用する際，それが困難，あるいは不可能になるであろうコードクローンのおかれている状況を定義し，分類する．分類した状況それぞれにおいて，リファクタリングの手順を考案し，リファクタリングが考案されていない分類をリファクタリングできない分類とする．コードクローンの分類には観点が 7 つ必要であり、それぞれ、以下のように定義している．

3.2 クローンペアの差異

クローンペアの差異がローカル変数なのか，シグニチャなのか，メソッド名なのかなどの差異の種類で分類する．先述のタイプ 2 のコードクローンをさらに分類したものとなっている（**図 5**）．ここでもし，オブジェクトの呼び出すメソッド名が違ったり，クラス名が違ったりすると単純にメソッドとして抽出することは難しくなる．

3.3 クローンペアの位置

クローンペア同士の位置が同じクラスなのか，兄弟クラスなのか，あるいは無関係なクラスなのかで分類する（**図 6** 参照）．

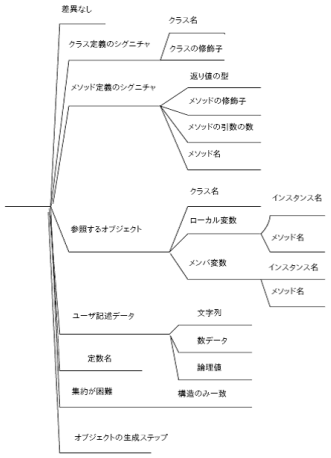

図 5 クローンの差異
Fig. 5 difference of clones

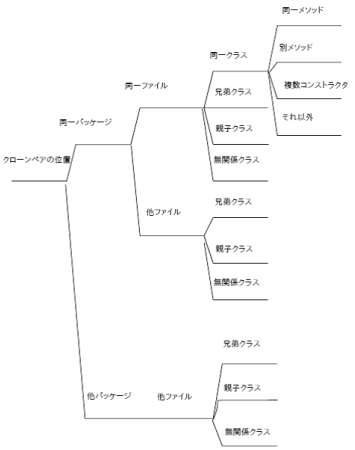

図 6 クローンの位置
Fig. 6 locations of clones

3.4 クローン部の長さ

クローン部がブロック単位なのか，メソッド単位なのか，クラス単位なのかで分類する．

3.5 メソッド抽出の際に必要になる引数オブジェクトの種類

メソッド抽出の際に必要となる引数のオブジェクトの権限やスコープで分類する．

3.6 メソッド抽出の際に必要になる引数の数

メソッド抽出の際に必要となる戻り値の数で分類する．0，もしくは 1 であれば問題はないが，もしも複数の戻り値が必要となる場合，単純に抽出することはできない．

3.7 制御構造要素の有無

クローン部が制御構造要素にまたがっているかどうかで分類する．例えば，if 文の宣言がコードクローンのコード片に入っているのに if 文を閉じていない，if 文の宣言部がコード片に含まれていないのに if 文を閉じる中括弧が入ってしまっている，等がある．クローン部に制御構造要素が含まれている場合，単純に抽出することはできず，抽出を検討する範囲を制御構造要素を含まない範囲にしなければならない．

3.8 instanceof の有無

クローン部内に instanceof が含まれているかどうかで分類する．

Vol.2011-SE-173 No.7
2011/7/21

4. 提 案 手 法

本節では，提案するリファクタリングパターンのコードクローンの特徴とメソッド引き上げの手順について述べる．

4.1 コードクローンの特徴

本研究で作成したパターンにおけるコードクローンの特徴を以下に述べる．

クローンペアの差異　ユーザ記述データ，定数

クローンペアの位置　兄弟クラス

クローン部の長さ　ブロック単位

メソッド抽出の際に必要になる引数オブジェクトの種類　private

メソッド抽出の際に必要な戻り値の数　0，1

制御構造要素の有無　なし

instanceof の有無　なし

4.2 メソッド引き上げの手順

本研究で提案するリファクタリングパターンの手順を以下に記述する．

(1)　親クラスにメソッドを宣言する．
- (a)　戻り値を一時的に void にする．
- (b)　そのメソッドの機能に適切な名前をつけ，引数の括弧をつける．
- (c)　中括弧をつけて（メソッドの体裁を整える），コンパイルしてテストする．
- (d)　クローンになっている部分を親クラスに引きあげる．
- (e)　クローン部の差異以外の変数，オブジェクトの引数を用意する．
- (f)　戻り値があるならオブジェクトの return を追化し，メソッドの型を戻り値の型にする．
- (g)　差異のための引数を作成し，差異を引数で置き換える．
- (h)　親クラスをコンパイルしてテストする．

(2)　元の子クラスのクローン部を取り除き，メソッド呼び出しを追加する．

(3)　コンパイルし，テストする．

図 7 に今回のパターンを適用するコードの実例を示す．このコードは銀行の顧客の口座を作成する際に，情報をセットしている．引数は顧客の名前を示す String 型の変数 name，口座番号を表す int 型の変数 number，利息率を示す int 型の定数の差異，それぞれ ORDINARY RATE，FIXED RATE を持つ．

```
:
customer = new Account();
customer.setName(name);
customer.setNumber(number);
custoomer.setRate(ORIDINARY_RATE)
:
```

```
:
customer = new Account();
customer.setName(name);
customer.setNumber(number);
custoomer.setRate(FIXED_RATE)
:
```

図 7　実例
Fig. 7　an actual case

```
void intialAccount(){


}
:
```

図 8　例 1
Fig. 8　example 1

```
void intialAccount(
   String name,int number){
customer = new Account();
customer.setName(name);
customer.setNumber(number);
customer.setRate(ORDINARY_RATE);
}
```

図 9　例 2
Fig. 9　example 2

手順 1，親クラスにメソッドを宣言する． 手順 1 においてはメソッドの引き上げにおける親クラスでのメソッドの宣言の手順について記述する．

戻り値を一時的に void にする　まずメソッドの型を決めるのであるが，最初は戻り値の型に関わらず，void に指定する．これは return 文を書いていない状況でもエラーが出ないようにするためである．

そのメソッドの機能に適切な名前をつけ，引数の括弧をつける． メソッドの名前は処理内容ではなく，役割，何をするかを表す単純なものとする．ここでは顧客の口座を作成し，初期値をセットしているので initialAccount とした．

中括弧をつけて，コンパイルしてテストする． 中括弧を付けるのは，ここでエラー内容を確認し，名前の衝突等がないようにする（**図 8**）．

クローンになっている部分を親クラスに引きあげる． このときクローン部をコピーするのであるが，どちらのクラスからコピーしてもよい．どちらのクラスからコピーして来たとしても後の手順に変わりはないからである．

差異以外の変数，オブジェクトの引数を用意する． 引数がないときには，ここでの操作

```
Account intialAccount(
   String name,int number){
customer = new Account();
customer.setName(name);
customer.setNumber(number);
customer.setRate(ORDINARY_RATE);
return customer;
}
```

図 10 例 3
Fig. 10 example 3

```
Account intialAccount(
   int rate,
   String name,int number){
customer = new Account();
customer.setName(name);
customer.setNumber(number);
customer.setRate(rate);
return customer;
}
```

図 11 例 4
Fig. 11 example 4

はない．引数がある時には，変数，オブジェクトの名前をそのままメソッド宣言に追加し，型名を加える．例（**図 9** 参照）では，引数として，String 型の変数 name，int 型の変 number を持つのでこの 2 つを引数として宣言している．このときサブクラスの両方で name，number という名前で定義されているので，親クラスでもこの名前で宣言する．

戻り値があるならオブジェクトの return 文を追化し，メソッドの型を戻り値の型にする． 最初に void で指定した部分はここで変える．もしも戻り値がないなら，ここでの操作はない．例では，Account 型のオブジェクト customer を返す必要があるので return 文を用意し，メソッドの型を Account に変更している（**図 10** 参照）．

差異のための引数を作成し，差異を引数で置き換える． 差異の引数の名前は差異の意味的な共通部分から考える．もしくは，その変数，オブジェクトが何をしているかを考える．例では，差異の名前が ORDINARY RATE，FIXED RATE なので意味的な共通部分 rate を変数の名前に設定し，引数として宣言している．

親クラスをコンパイルしてテストする． ここでエラーが出なければ，親クラスでの処理は終了である．

手順 2，元の子クラスのクローン部を取り除き，メソッド呼び出しを追加する．

手順 3，コンパイルし，テストする． 手順 3 は手順 2 と並行して行う．各子クラスのクローン部の除去とメソッド呼び出しの追加を行ったあと，コンパイルとテストを行う．ここで問題がなければ全手順は終了である．

5. 適 用 実 験

本節では，3 節で提案したリファクタリングパターンを実際のオープンソースソフトウェアに適用した事例について述べる．提案したリファクタリングパターンは既存のリファクタリングパターンよりも詳細化されており，分類された特徴のコードクローンに対して，既存のものよりも容易にリファクタリングできるということを目的にしている．したがって，本研究の実験においては 3 名の被験者にパターンを用いて実際にリファクタリングを行ってもらい，評価してもらった．まず，適用対象及び準備について説明する．続いて提案手法を用いて，リファクタリングを行ってもらった結果と，その考察について述べる．

5.1 準　　備

適用実権にはオープンソースソフトウェアの sootsrc-2.2.4 を対象に用いた．また，sootsrc-2.2.4 から提案手法を適用するコードクローンを検出するためのコードクローン検出ツールとして，CCFinder [6] を用いた．CCFinder はトークン列からコードクローンを検出する．CCFinder を用いて sootsrc-2.2.4 からコードクローンを検出し，その中から提案手法を適用する条件を満たす特徴を持つコードクローンを手作業で探し，3 つのクローンペアを対象に決定した．

本研究の実験においては，3 名の被験者に協力してもらい実験対象のコードクローンに対し，リファクタリングをしてもらう．その際，提案したパターンと既存のパターン [4] で比較をしてもらう．どの対象に対し，そのパターンを適用するかは，**表 1** に示す．

それぞれ順番を変えているのは，パターンは学習効果の影響が大きいためである．すなわち，単一の順番で実験を行うと前に行ったパターンの学習による効果なのか，現在実験しているパターンの効果なのかが分からないために，それぞれ順番を変えて実験を行うことでパターンそれぞれの効果について明白にしようということである．

リファクタリング前後の動作確認 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，66 個の Junit のテストケースを用いたテストを行ってもらった．それによって，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．

5.2 結　　果

適用実験を行った結果について述べる．まず，対象へのリファクタリングを行った後に

表 1 実験の流れ
Table 1 candidate of the expriment

	被験者 A	被験者 B	被験者 C
対象 1	提案したパターン	既存パターン	既存パターン
対象 2	既存パターン	提案したパターン	提案したパターン
対象 3	提案したパターン	既存パターン	既存パターン

提案したパターンが既存のパターンより優秀である点，劣っている点について，述べてもらった．

なお，具体的にどのような点で優劣をつけたのかは**表 2** に示す．

引数に対する扱いの詳細さは，手順 1.5 で述べられている内容で，既存パターンでは引数に対する扱いは述べられていない．戻り値がある時の扱いの詳細さは，手順 1.6 で述べられている内容で，既存パターンでは引数に対する扱いは述べられていない．差異に対する扱いの詳細さは手順 1.7 で述べられている内容で，既存パターンでは必要に応じて修正するとだけ記されている．また，既存パターンに劣っていた点は**表 3** のようになった．

手順全体の柔軟さが足りないというのは，詳細化を進めたために，実際にコードを書く開発者に判断を委ねている部分が少ないということである．被験者 B は最初にコードを見て，オブジェクトを返す必要があるのを確認した後親クラスでのメソッド宣言で型を戻り値の型にしようとして，手順に反しているという状況があったために，柔軟さの欠如があるという評価を下した．

5.3 考　　察

まず，それぞれのリファクタリング後のソフトウェアに対して，テストを行った結果外部的動作に変化がないことを確認できたため，提案したリファクタリングパターンが妥当なものであることが確認できた．また，被験者 3 人それぞれが既存のリファクタリングパターンよりも本研究で提案したリファクタリングパターンの方が有効であると評価した．しかし，被験者の 1 人は，手順の柔軟さの点において既存のパターンを評価している．そのため，それぞれのコーディングスタイルを重視させてほしいと思う人間にとってはその点で，詳細化を進めていない既存の手法を評価している．また，リファクタリングにかかった時間は**表 4** のようになっている．実験前の予想では，提案したパターンを使ったときにの方が全体的にリファクタリングにかかる時間が少なくなるのではないかと考えていた．しかし，実際の結果では，回数を重ねる毎に時間がかからなくなる傾向にあり，提案したパターンの時間的優位性は認められなかった．これはパターンの正確さを保証するために，実験の際に手順を逐次確認していたためであり，この確認作業のコストを減らす手段を考えるのは今後の課題である．

表 2　パターンの比較
Table 2　comparing of patterns

	被験者 A	被験者 B	被験者 C
引数に対する扱いの詳細さ	○	○	○
戻り値がある時の扱いの詳細さ	○	○	○
差異に対する扱いの詳細さ	○		○

表 3　提案したパターンが劣っている点
Table 3　inferior of a proposed pattern

	被験者 A	被験者 B	被験者 C
手順全体の柔軟さ		○	

6. 関 連 研 究

Balazinska らは，保守支援を目的として，コードのクローンの差異に基づく分類を行っている [1]．また，Kapser らは，コードクローンをプログラム要素（関数や構造体など）との対応関係に基づいて分類している [7]．これら研究は，2 節で述べた徳永らの研究とは異なり，リファクタリングパターンを作成を目的とした分類ではないため，本研究では用いていない．今後，Kapser らや Balazinska らの研究を参考に，徳永らの分類を改善すると，有用なリファクタリングパターンを作成できると考えられる．メソッド抽出に基づいてコードクローン集約を支援する手法が数多く提案されている [11] [5]．これら手法を拡張し，本研究で提案したリファクタリングパターンに基づいたコードクローン集約支援手法を実現することで，より有用な支援を実現できると考えられる．

7. まとめと今後の課題

本研究では，特徴毎に分類されたコードクローンのコード片に対し，既存のものよりも詳細なリファクタリングパターンを提案した．具体的には，徳永らの提案したコードクローンの特徴による分類を用いて，既存のリファクタリングパターンであるメソッドの引き上げを詳細化し，特定のコードクローンに対し，既存のパターンよりも簡単にリファクタリング作業を行えるようにした．この提案手法を実際にオープンソースソフトウェアのソースコードを対象に，まず提案したパターンの分類を探し，その後，実際に被験者に適用実験を行って

表 4　実験に要した時間
Table 4　time to cost for experiment

	被験者 A	被験者 B	被験者 C
対象 1	10:57	9:32	10:00
対象 2	9:16	6:38	5:39
対象 3	6:27	3:02	6:08

もらった．それによってリファクタリングの前後で外部的動作が変わらないことを確認し，分類されたコードクローンに対しての提案したパターンの有効性を確かめた．今後の課題として，別の更なるリファクタリングパターンの提案と，コードクローンの分類の自動化が考えられる．

更なるリファクタリングパターンの提案　本手法では，利用した分類のコードクローンすべてに対してのリファクタリングパターンを提案できていない．まだパターンを考案できていない分類において，パターンをすべて考える，もしくはリファクタリングできない分類であると位置づけることができればこの分類自体の有効性も上がると考えられる．

コードクローンの分類の自動化　また，求めている特徴を満たすコードを自動で検出する手法も必要である．本研究においてはコードクローンを検出したあと，自ら提案したパターンの特徴を満たすコードクローンを選んだが，この作業を自動化できれば，実際のリファクタリング作業において，効率化を図ることができると考えられる．

謝辞　本研究は，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），および研究活動スタート支援（課題番号:22800040）の助成を得た．

参 考 文 献

1) Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K.: Measuring clone based reengineering opportunities, *Proc. of METRICS'99*, pp.292–303 (1999).
2) Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and evaluation of clone detection tools., *IEEE Trans. Softw. Eng.*, Vol.33, No.9, pp. 577–591 (2007).
3) Fowler, M.: Refactoring Home Page, ThoughtWorks (online), available from ⟨http://refactoring.com/⟩ (accessed 2010-12-14).
4) Fowler., M.: *Refactoring: improving the design of existing code*, Addison Wesley (1999).
5) Higo, Y., Kusumoto, S. and Inoue, K.: A metric-based approach to identifying refac- toring opportunities for merging code clones in a java software system, *Journal of Software Maintenance and evolution: Research and Practice*, Vol.20, No.6, pp.435–461 (2008).
6) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
7) Kapser, C. and Godfrey, M.: Aiding comprehension of cloning through categorization, *Proc. of IWPSE2004*, pp.85–94 (2004).
8) Kerievsky, J.: *Refactoring to Patterns*, Addison Wesley (2004).
9) Opdyke, W.F.: Refactoring Object-Oriented Frameworks, *PhD Thesis, University of Illinois at Urbana-Champaign* (1992).
10) 丸山勝久：基本ブロックスライシングを用いたメソッド抽出リファクタリング.，情報処理学会論文誌，Vol.50, No.2, pp.1625–1637 (2002).
11) 吉田則裕，肥後芳樹，神谷年洋，楠本真二，井上克郎：コードクローン間の依存関係に基づくリファクタリング支援，情報処理学会論文誌，Vol.48, No.3, pp.1431–1442 (2007).
12) 三宅達也，肥後芳樹，井上克郎：メソッド抽出の必要性を評価するソフトウェアメトリックスの提案，電子情報通信学会論文誌 D，Vol.J92-D, No.7, pp.1071–1073 (2009).
13) 中江大海，神谷年洋，門田暁人，加藤裕史，佐藤慎一，井上克郎：レガシーソフトウェアを対象とするクローンコードの定量的分析，電子情報通信学会技術研究報告，Vol.100, No.570, pp.57–64 (2001).
14) 徳永将之，吉田則裕，吉岡一樹，松下　誠，井上克郎：コードクローンの分類に基づくリファクタリングパターン集の提案に向けて，ウィンターワークショップ 2011 イン修禅寺論文集，情報処理学会シンポジウムシリーズ，Vol.2011, No.2, pp.79–80 (2011).
15) 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会，Vol.J87-D-I, No.12, pp.1060–1068 (2004).