チュートリアル

コードクローンを対象としたリファクタリング

肥後 芳樹　吉田 則裕

コードクローンとは，ソースコード中に存在する互いに一致，もしくは類似したコード片を指す．コードクローンの存在は，ソフトウェアの開発および保守に悪影響を与える恐れがあるといわれている．本稿では，コードクローンを取り除くためのリファクタリング方法と近年の研究成果について紹介する．

A code clone is a code fragment that has identical or similar code fragments to it in the source code. It is generally said that the presence of code clones makes software development and maintenance more difficult. This paper describes several refactoring patterns that are suitable for removing code clones and introduces several research efforts related to code clone refactoring.

1　はじめに

一旦実装が完了したソフトウェアシステムであっても，さまざまな理由によりそのソースコードには改変が加え続けられる．例えば，顧客からの要求変更や法改正に伴い，機能の追加・修正が必要になる．また，運用開始後にバグが顕在化し，修正を行うことも多い．このような度重なる改変により，ソースコードは，開発された当初の (すぐれた設計の) コードから，理解できない無秩序状態のコードへと崩れていく．

このようなコードの品質低下を防ぐための方法として**リファクタリング** (**Refactoring**) が挙げられる．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つつ内部の構造を改善することである．リファクタリングを行うことにより，システムの機能性を変更することなく，そのコードを改善できる．

リファクタリングの先駆者マーチン・ファウラーは，リファクタリングを考慮するべきコードの悪い状態をいくつか紹介している [6]．代表的なものとしては，巨大なクラスや長いメソッドなどが挙げられる．その著書の中では，**重複コード** (**Duplicate Code**)(以降，**コードクローン** (**Code Clone**)) もリファクタリングを考慮すべきとして紹介されている．コードクローンとは，ソースコード中に存在する互いに一致，もしくは類似したコード片を指す．さまざまな理由によりコードクローンはソースコード中に生成されるが，その最も大きな原因はコピーアンドペーストである [11]．新しい機能を追加するときに，すでにその機能と類似した機能の実装があれば，その部分からコピーアンドペーストを行うことが多い．コピーアンドペースト後は，細かい修正を加えて新しい機能が完成する．このような実装を行うことにより，大同小異なコードが散在することになる．

コピーアンドペーストは，必要な機能を素早く実装できるという点では優れている．しかし，コピー元にバグが含まれていた場合，そのバグをまき散らしてしまう．これにより，保守管理者が顕在化したバグ (不具合) を修正したとしても，そのコードクローンのチェックをしなかった場合，同様のバグがシステムに残ってしまう．コードクローンに対してあらかじめリファクタリングを行っておくことにより，コードク

An Introduction to Code Clone Refactoring.

Yoshiki Higo, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

Norihiro Yoshida, 奈良先端科学技術大学院大学情報科学研究科, Graduate School of Information Science, Nara Institute of Science and Technology.

コンピュータソフトウェア, Vol.28, No.4 (2011), pp.43–56.
[解説論文] 2011 年 3 月 29 日受付.

ローンが 1 つのモジュールとしてまとめられ，将来のソースコード改変に必要なコストを削減することができる．

以降，2 章ではコードクローンの検出方法について述べ，3 章ではコードクローンを除去するために利用可能なリファクタリング方法を紹介する．4 章ではコードクローンに対するリファクタリング支援の研究成果を紹介し，5 章では研究のニーズと今後の展開について著者なりの考えを述べる．最後に 6 章で本稿をまとめる．なお，本稿は，オブジェクト指向のプログラミング言語を対象としている．

2 コードクローン検出技術

現在までにさまざまなコードクローン検出技術が提案されている．紙面の都合上個々の手法を紹介することができないので，興味のある方は文献 [11] を参照されたい．既存の検出技術はコードクローンをどのように検出するのかによって，大まかに以下の 5 つに分類することができる．

- 行単位の検出
- 字句単位の検出
- 抽象構文木を用いた検出
- プログラム依存グラフを用いた検出
- メトリクスやフィンガープリントなど，その他の技術を用いた検出

また，コードクローンは下記の 3 つに分類される [11]．

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 変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローン．

タイプ 3 タイプ 2 における違いに加えて，文の挿入や削除，変更が行われたコードクローン．

プログラム依存グラフを用いた検出とメトリクスやフィンガープリントなどを用いた検出では，全てのタイプのコードクローンを検出可能であるのに対して，行単位，字句単位，抽象構文木を用いた検出手法では，その性質上タイプ 1 とタイプ 2 のみが検出可能である．しかし，現在では，検出時に細粒度の違いを無視する [17]，または複数の検出技術を組み合わせる [7]，などの工夫を行い，全てのタイプのコードクローンを検出可能な手法も存在する．

提案されたコードクローン検出技術に基づき多くの検出ツールが開発されている．しかし，一般に公開されていないものもあれば，商用として有償のみで提供されているものもある．無償で公開されており，コードクローンに関する研究で頻繁に用いられるツールは，以下のものである．もし，初めてコードクローン検出ツールを用いる場合は，これらを使うと良いだろう．

- 行単位の検出ツール Simian [26]．
- 字句単位の検出ツール CCFinder [18] とその後継機 CCFinderX [4]．
- 抽象構文木を用いた検出ツール Declard [17]．

以降，本章では，5 つの検出技術の簡単な説明を行う．

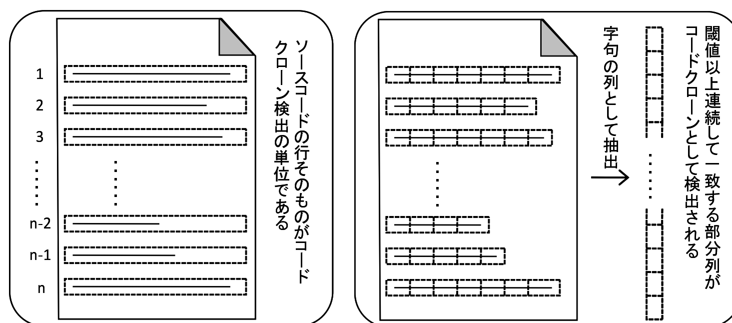
2.1 行単位の検出

ソースコードを行単位で比較することにより，コードクローンを特定するのが行単位の検出手法である (図 1(a))．行単位の検出手法では，閾値以上連続して重複している行がコードクローンとして検出される．この検出技術は，他の技術に比べて検出処理が高速であるという特徴を持つが，同じ処理を行っているコードであってもコーディングスタイルが違う場合はコードクローンとして検出できないという弱点を持つ．

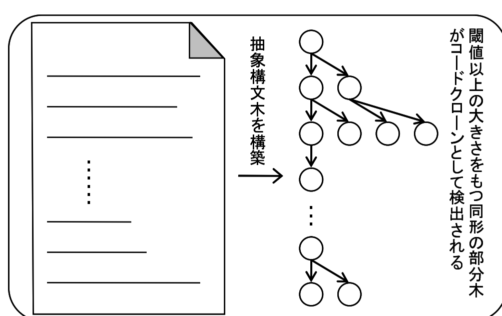
リファクタリングは，クラスやメソッド，メソッド内の文など，プログラミング言語の構造を単位として行われることが多い．しかし，行単位での検出では，検出されるコードクローンは連続して重複している行であり，プログラミング言語の構造とは一致しない．そのため，行単位で検出したコードクローンをリファクタリング対象とするのは適切ではない．

2.2 字句単位の検出

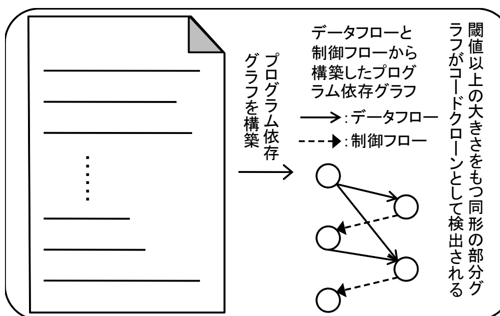
字句単位のコードクローン検出では，検出の前処理としてソースコードは字句の列に変換される (図 1(b))．閾値以上連続して一致している字句列がコー



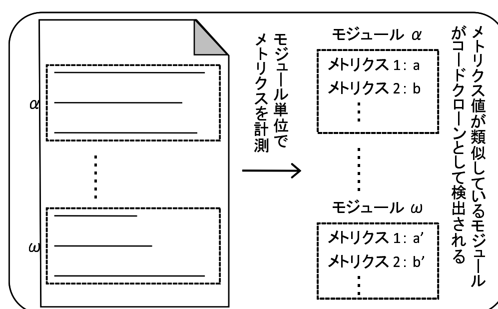
(a) 行単位の検出　(b) 字句単位の検出



(c) 抽象構文木を用いた検出



(d) プログラム依存グラフを用いた検出



(e) メトリクスを用いた検出

図 1　コードクローン検出の前処理

ドクローンとして検出される．字句単位のコードクローン検出は，行単位の検出のように検出結果がコーディングスタイルに依存することはない．また，字句列への変換には高い計算コストを必要としないため，高速にコードクローンを検出可能である．しかし，行単位の検出と同様，検出されるコードクローンはプログラミング言語の構造と一致しているわけではないため，リファクタリングという視点では，利用しづらいコードクローンが検出される．

2.3 抽象構文木を用いた検出

抽象構文木 (Abstract Syntax Tree) を用いた検出では，検出の前処理としてソースコードに対して構文解析を行い，抽象構文木が構築される (図 1(c))．抽象構文木上の同形の部分木がコードクローンとして検出されるため，検出結果がコーディングスタイルに依存することはない．コードクローンの検出処理を行う前に，抽象構文木を構築する必要があるため，行単位や字句単位の検出に比べて，検出に必要な時間的及び空間的なコストは高くなるが，実用的な検出法として知られている．さらに，プログラムの構造から構築した抽象構文木上での一致部分がコードクローンになるため，行単位や字句単位の検出とは異なり，ある関数定義の終わりから次の関数定義の先頭までの類似部分のような，プログラムの構造を無視したコードクローンは検出されない．よって，抽象構文木を用いた検出は，リファクタリングという視点では，適切な検出法といえる．

2.4 プログラム依存グラフを用いた検出

プログラム依存グラフ (Program Dependency Graph) を用いた検出では，検出の前処理としてソースコードに対して意味解析を行い，ソースコードの要素 (文や式など) 間の依存関係が抽出され，要素を頂点，依存関係を有向辺とするプログラム依存グラフが構築される (図 1(d))．グラフ上の同形の部分グラフがコードクローンとして検出される．プログラム依存グラフを用いた検出の長所は，他の技術では検出することのできないコードクローンを検出できることである[†1]．また，プログラム依存グラフを用いて検出されたコードクローンは，意味的に等しい処理を行っているため，リファクタリングの対象としては，適切である．しかし，同形部分グラフの特定に高い計算コストを必要とするため，この技術を大規模ソフトウェアに対して適用することは難しい．

†1 順序入れ替わりクローンや巻きつきクローンなど．詳しくは文献[11] を参照されたい．

2.5 メトリクスやフィンガープリントなどその他の技術を用いた検出

プログラムのモジュール (ファイルや，クラス，メソッドなど) に対してメトリクスを計測し，その値の一致または近似の度合いを検査することによって，そのモジュール単位でのコードクローンを検出する手法が，メトリクスを用いたコードクローン検出である (図 1(e))．メトリクス値が近似していればコードクローンとして判定されるため，多少の違いがあるコードクローンも検出することが可能である．しかし，規模の小さいモジュールの場合はメトリクス値に差がでないため誤検出の可能性が高くなり，また規模の大きいモジュールの場合はその内部の一部分が類似していてもコードクローンとして判定されない，という問題点がある．このため，この手法により検出されるコードクローンはプログラミング言語の構造と一致しているが，リファクタリングに利用できるものは少ない．

3 コードクローンの除去に適用可能なリファクタリングパターン

コードクローンに対するリファクタリングとは，コードクローンをソースコード中から除去することを意味する．これまでにさまざまなリファクタリングパターンが考案されており，その中のいくつかはコードクローンを除去するために利用可能である．本節では，それらのうち代表的なものを紹介する．また，表 1 は，各リファクタリングパターンが対象にしているコードクローンのタイプを表している．タイプ 1 とタイプ 2 のコードクローンについては複数の除去方法があるが，タイプ 3 のコードクローンについては，テンプレートメソッドの形成が主な除去方法である．

3.1 メソッドの抽出 (Extract Method)

メソッドの抽出とは，既存メソッドの一部分を新たなメソッドとして抽出することである．コードクローン間に共通するロジックを表すメソッドを 1 つ用意し，コードクローン間の差異部分 (用いている変数やリテラル) を引数として渡せば，全てのコードクローンを共通のメソッドで代替可能である．図 2 はメソッ

表 1 リファクタリングパターンが対象としているコードクローン

リファクタリングパターン	対象コードクローン		
	タイプ 1	タイプ 2	タイプ 3
メソッドの抽出	○	○	△[†2]
(親) クラスの抽出	○	○	-
メソッドの引き上げ	○	○	-
メソッドの移動	○	○	-
メソッドのパラメータ化	-	○	-
テンプレートメソッドの形成	-	-	○

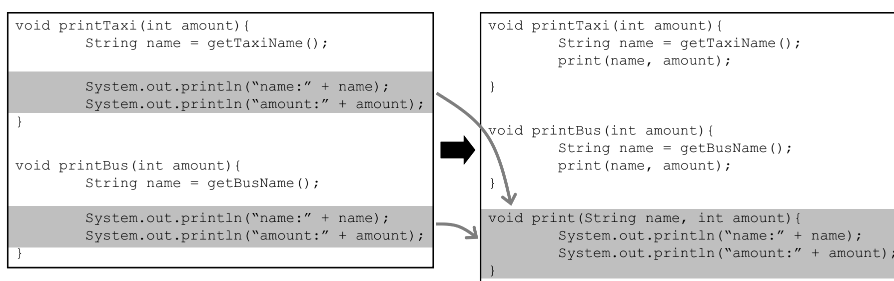


図 2 メソッドの抽出の例

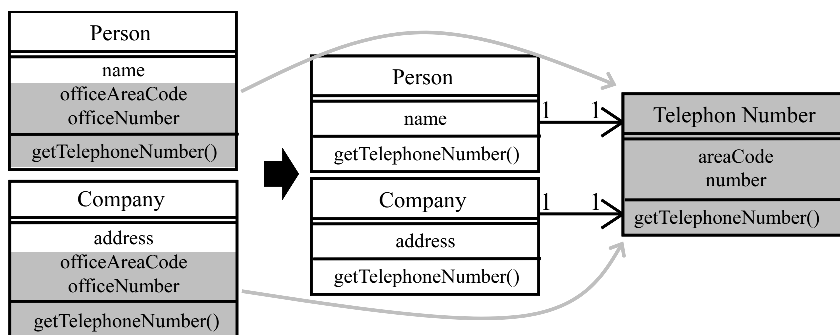


図 3 クラスの抽出の例

ドの抽出リファクタリングの例を表している．この例では，2 つのメソッドに存在している情報出力命令が新しいメソッドとして抽出されている．

†2 ソースコード上でタイプ 3 のコードクローンになっていても，振る舞いを変えることなく，命令の順番を入れ替えることにより，タイプ 1 やタイプ 2 のコードクローンに変換できれば，メソッドの抽出を適用可能である．ただし，著者らの経験から，このようなコードクローンは極めて少ない．

3.2 クラスの抽出 (Extract Class)，親クラスの抽出 (Extract SuperClass)

クラスの抽出と**親クラスの抽出**は，既存クラスの一部分を新たなクラスとして抽出することである．抽出したクラスを既存クラスの親クラスとするかどうかはリファクタリングを行う状況に応じて決定する．例えば，既存クラスがすでに親クラスを所有している場合は，Java や C#では，1 つの親クラスしか継承

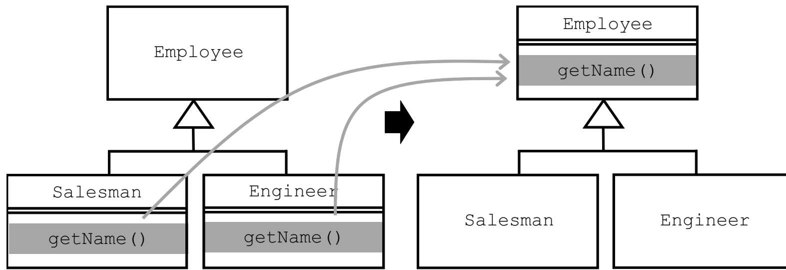


図 4　メソッドの引き上げの例

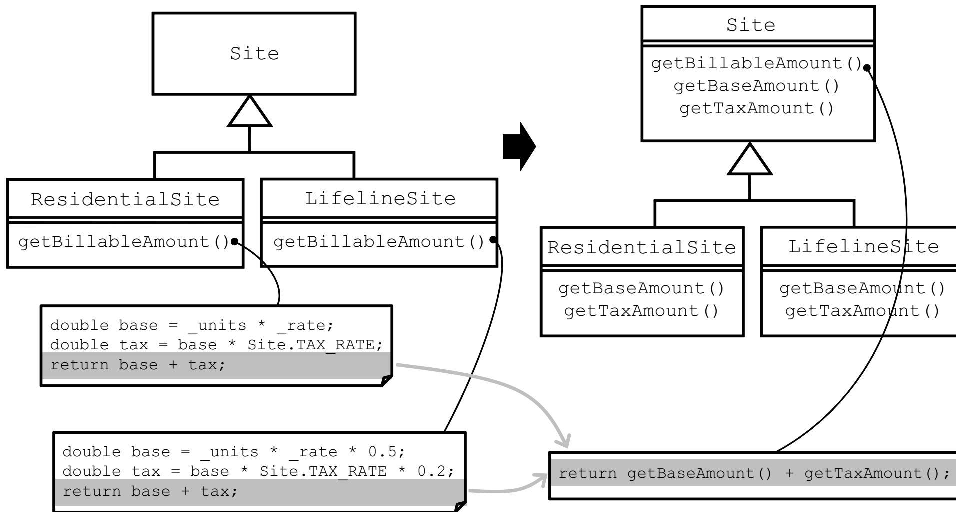


図 5　テンプレートメソッドの形成の例

できないため，抽出したクラスを対象クラスの親クラスとすることはできない．このような場合は，クラスの抽出を適用し，既存クラスの機能を新しいクラスに委譲するようにリファクタリングを行う．複数のクラスが多くのコードクローンを共有している場合，クラスの抽出や親クラスの抽出を行うことによって，既存クラス間のコードクローンをなくすことができる．図 3 は，クラスの抽出の例を表している．この例では，Person と Company という 2 つのクラスが電話番号に関する機能をそれぞれ実装していたが，この機能を TelephonNumber というクラスとして抽出することにより，既存クラス間のコードクローンをなくしている． Person と Company は共に電話番号に関する機能を持つが，2 つのクラス間に意味的なつながりが薄いため，クラスの抽出を行っている．2 つのクラスは共に親クラスを持たないため，親クラスの抽出を行うことも可能である．

3.3　メソッドの引き上げ (Pull Up Method)

メソッドの引き上げとは，既存のメソッドを親クラスに引き上げることである．共通の親クラスを持つ，複数の子クラスに重複したメソッドが存在した場合，それらを共通の親クラスに引き上げることにより，コードクローンをなくすことができる．図 4 はメソッドの引き上げの例を表している．この例では，Salesman クラスと Engineer クラスに getName メソッドが存在していたが，リファクタリングにより共通の親クラスである Employee クラスに引き上げら

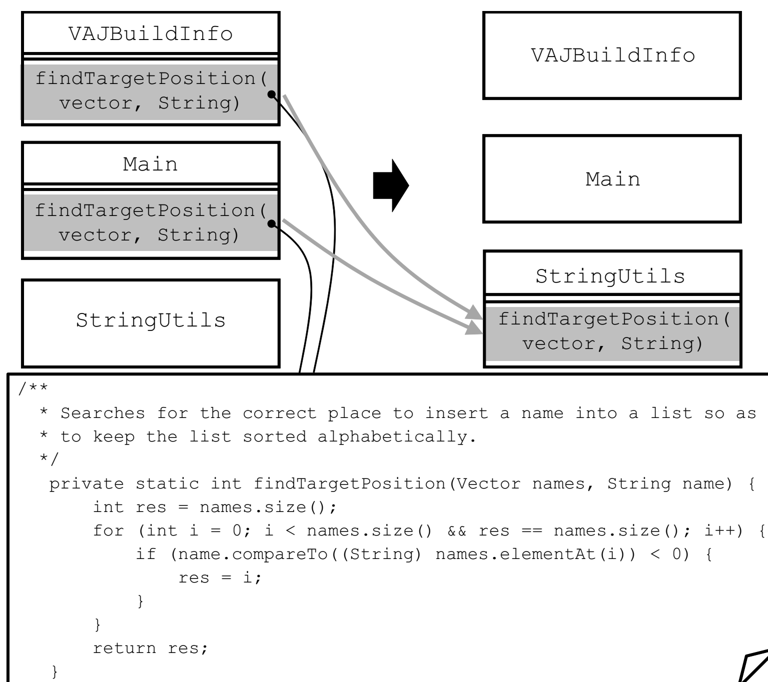


図 6　メソッドの移動の例

れている．このリファクタリングにより， Salesman クラスと Engineer クラス間のコードクローンがなくなっている．なお，メソッド全体ではなく一部分のみがコードクローンになっている場合でも，先に説明したメソッドの抽出を組み合わせることにより，重複部分のみを親クラスに引き上げることが可能である．また，複数のクラスがコードクローンを所有しているが，それらが共通の親クラスを持たない場合は，先に説明した親クラスの抽出を適用することも可能である．

3.4　テンプレートメソッドの形成 (Form Template Method)

テンプレートメソッドの形成とは，複数の子クラスのメソッドにおいて，処理の手順は同じであるが詳細な処理内容は異なっている場合に，処理の手順を共通化する方法である．図 5 はテンプレートメソッドの形成の例を表している．この例では，RegisentialSite と LifelineSite という 2 つのクラスが getBillableAmount というメソッドを所有している．これらのメソッドは，base と tax を計算しその和を返す，という処理の手順は同じであるが，base と tax の計算方法は異なっている．この異なった部分を共通のシグネチャを持つメソッドとして抽出することにより，処理の手順を共通化することができる．コードクローンを共有しているメソッドに対して適用する場合は，コードクローンでない部分をメソッドとして抽出し，コードクローン部分を親クラスに引き上げることにより，コードクローンを削除することができる．

3.5　メソッドの移動 (Move Method)

メソッドの移動とは，メソッドを他のクラスに移動させる処理である．親クラスに移動させる場合は，先に述べたメソッドの引き上げになる．複数のクラスに存在している重複したメソッドを別のクラスにに移動させ，重複したメソッドを持つクラスは，その移動させたメソッドを利用するようにソースコードを変更することにより，コードクローンを削除できる．

図 6 はメソッドの移動の例である．この例はオープ

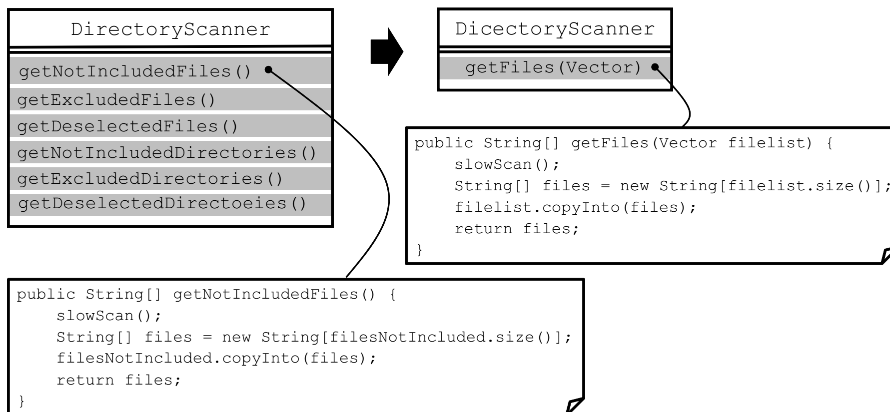


図 7　メソッドのパラメータ化

ンソースソフトウェア Ant[†3]の一部をクラス図で表現している．リファクタリング前は，VAJBuildInfo クラスと Main クラスには，文字列に対して処理を行うメソッドが定義されており，コードクローンとなっていた．これらのメソッドを文字列に関するさまざまな処理を定義している StringUtils クラスに移動させることにより，コードクローンを削除できる．

3.6　メソッドのパラメータ化 (Parameterize Method)

メソッドのパラメータ化とは，メソッド内で利用されている変数やリテラルを，引数に変更することである．よく似た振る舞いをするものの，異なる値を利用している複数のメソッドがあった場合に，その値を引数として受け取るように変更することにより，コードクローンを削除できる．

図 7 はメソッドのパラメータ化の例を表している．これは Ant に含まれる DirectoryScanner クラスを表している．このクラスでは，図 7 に記載されている 6 つのメソッドがコードクローンとなっていた．このメソッドは異なるフィールド変数を使っているものの，処理内容は同じであるため，メソッドのパラメータ化を適用することができる．図 7 の右側は適用後のソースコードを表している．リファクタリング前に用いていたフィールド変数は，引数としてメソッドに与えられるように変更されている．

†3　http://ant.apache.org

4　コードクローンを対象としたリファクタリング支援手法

本章では，コードクローンを対象としたリファクタリング支援手法を紹介する．まずはじめに，リファクタリングは複数の手順からなる作業であることを述べなければならない．

手順 1　対象システムのリファクタリングすべき部分を特定する

手順 2　特定した部分をどのようにリファクタリングするのかを決定する

手順 3　そのリファクタリングが，システムの振る舞いを変えないことを確認する

手順 4　ソースコードの改変を行う

手順 5　メトリクスの計測やレビューなどを行い，その改変によりソフトウェアの品質が向上したことを確認する

手順 6　設計書などのソースコード以外の生産物に対して，改変されたソースコードとの一貫性を維持するように変更する

上記の手順は，文献 [23] で紹介されているものである．組織やシステムのドメインによって多少異なる部分はあるだろう．これまでに発表されているコードクローンを対象としたリファクタリング支援手法では，主に手順 1，手順 2 および手順 4 が支援の対象となっている．コードクローンに限らない一般のリファクタ

表 2　各支援手法が対象としているコードクローンおよびリファクタリング手順

支援手法	対象コードクローン			支援内容					
	タイプ 1	タイプ 2	タイプ 3	手順 1	手順 2	手順 3	手順 4	手順 5	手順 6
4.1 節	○	○	-	○	○	-	○	-	-
4.2 節	○	○	-	○	○	-	-	-	-
4.3 節	-	-	○	○	○	-	-	-	-
4.4 節	○	○	○	-	-	-	○	-	-
4.5 節	○	○	○	○	△†4	-	-	-	-

表 3　各支援手法が対象としているリファクタリングパターン

支援手法	対象リファクタリングパターン
4.1 節	メソッドの抽出，メソッドの引き上げ，メソッドの移動，メソッドのパラメータ化
4.2 節	3 章で詳細されているパターンのうち，テンプレートメソッドの形成以外のすべて
4.3 節	テンプレートメソッドの形成

リング支援手法では，手順 1 から手順 6 まで，さまざまな手法が提案されている．興味のある方は文献 [23] を参照されたい．

なお，各支援手法が対象としているコードクローンのタイプ，およびリファクタリングの手順をまとめたものを表 2 に表す．また，支援手法のうち，リファクタリングの手順 2(リファクタリング方法の決定) を支援しているものについては，どのリファクタリングパターンに対応しているのかを表 3 に表す．

4.1　リファクタリング後のコードを提示

Baxter らは，CloneDR というコードクローン検出ツールを開発している [1]．このツールは抽象構文木を用いたコードクローン検出を行っているため，検出されるコードクローンはプログラミング言語の構造と一致しており，リファクタリングに適している．また，このツールは，検出されたコードクローンを 1 つの関数やメソッドにまとめた場合のコードも出力する．よって，リファクタリングという点では，この検出ツールは非常に有効である．このツールの情報は，[25] のウェブページで得ることができる．C/C++，Java，C#などさまざまなプログラミング言語に対応しているのも強みである．商用ツールではあるが，無料の試用版も用意されている．

4.2　リファクタリング方法の決定支援

肥後らは，Java 言語で記述されたソフトウェアにおいて，コードクローンの特徴をメトリクスとして表現することにより，各コードクローンがどのように除去できるのかを予測する手法を提案している [9]．用いているメトリクスは主に結合度メトリクスと位置メトリクスである．

結合度メトリクス　コードクローン部分がその周囲のコードとどの程度強く結び付いているのかを表す．コードクローン部分において，その外部で定義された変数を利用しているほど，結合度が高くなるように定義されている．よって，このメトリクス値が高い場合はコードクローン部分に対する抽出処理が難しいことになる．

位置メトリクス　コードクローンが対象システム内でどのように分布しているのかを表す．このメトリクスを参照することにより，コードクローン

†4　4.5 節で紹介する手法は，コードクローン間の差異部分を強調表示するため，リファクタリング方法の決定に役立てることができる．しかし，その情報だけでは不十分なため，ここでは△としている．

が，1 つのクラス内に存在しているのか，共通の親クラスを持つ複数の子クラス内に存在しているのか，共通の親クラスをもたない複数のクラスに分散して存在しているのかを瞬時に把握できる．コードクローンの位置関係を把握することにより，3 章で紹介したリファクタリングパターンのどれが適用可能かを判断する助けとなる．

この手法はツールとして実装され，[20] のウェブページでその情報を得ることができる．ツールは無償で利用することができるが，対応しているのは Java 言語のみである．

吉田らは，メソッド間の依存関係を用いることにより，肥後らの手法をより効果的に適用する方法を考案している [30]．例えば，コードクローンになっており，かつ呼び出し関係にあるメソッド群はまとめてリファクタリングの対象にすることができ，より効率的にリファクタリングを行うことができる．また，メソッドとフィールドの利用関係も用いるため，メソッドのみでは別クラスへの移動が難しい場合でも，利用しているフィールドも同時に移動させることにより，リファクタリングの機会が多くなることを示している．

なお，肥後らの手法および吉田らの手法では，コードクローン検出は，字句単位の検出ツールである CCFinder [18] を用いている．既に述べたように，字句単位のコードクローンは，リファクタリングに向いていないものが多いため，彼らの手法では，検出されたコードクローンから，プログラミング言語の構造と一致している部分を抽出して用いている．

4.3 テンプレートメソッドの形成の支援

4.1 節および 4.2 節で紹介した手法は，メソッドの抽出やメソッドの引き上げなど，比較的抽出処理が単純なリファクタリングを支援する手法である．より複雑なリファクタリングを支援する手法としては，政井ら，および堀田らがテンプレートメソッドの形成を利用したコードクローンの除去手法を提案している [12] [21]．これら 2 つの手法は，まず与えられたメソッド間のコードクローンを検出する．政井らの手法では，メソッドの抽象構文木間で重複している部分木とそうでない部分木を特定している．堀田らの手法では，プログラム依存グラフを用いた検出ツール Scorpio [10] を用いてコードクローンを検出している．そして，メソッドを抽象構文木やプログラム依存グラフで表現し，検出したコードクローンをその上にマッピングする．これにより，各メソッドの抽象構文木やプログラム依存グラフは，コードクローン部分とコードクローンでない部分に分けることができる．コードクローンでない部分については，メソッド間の対応関係を取る (同じシグネチャを持つメソッドとして抽出する部分として特定する)．コードクローン部分は共通処理であるので共通の親クラスに引き上げられ，コードクローンでない部分はそのメソッド独自の処理であるので，そのメソッド内に残される．これらの手法は，ユーザが入力として与えたメソッドの対のどの部分を共通の親クラスに引き上げるのか，どの部分を抽出して子クラスに残すのかをユーザに提示する．これらは，まだ始まったばかりの研究であり，今後の発展が期待される．現在のところ，利用可能なツールは公開されていない．

4.4 プログラミング言語の抽象化機構に頼らないリファクタリング支援

Jarzabek は，従来のプログラミング言語の抽象化機構ではまとめることが難しい，複数のクラスにまたがるような，大きい単位での類似部分を集約するためのフレームワーク XVCL を提案及び開発している [14]．XVCL を用いることによって，類似クラス群は，メタコンポーネントと呼ばれるモジュールに集約される．メタコンポーネントは，使用されているプログラミング言語を用いたコードの雛形と，そのメタコンポーネントがどのようにコンパイル可能なクラスに展開されるかを記述した XVCL の命令文を含んでいる．

図 8 は簡単なメタコンポーネントの例を表している†5．図 8 の左上は，XVCL メタコンポーネント中のコードの雛形を表している．この雛形は Java 言語を用いて作成されているが，Java 言語にはない名前 (`@className` や`@common`) や命令 (`<while messages>`) を含んでおり，その部分が図

†5 これは XVCL のウエブページ [13] で紹介されている例である．

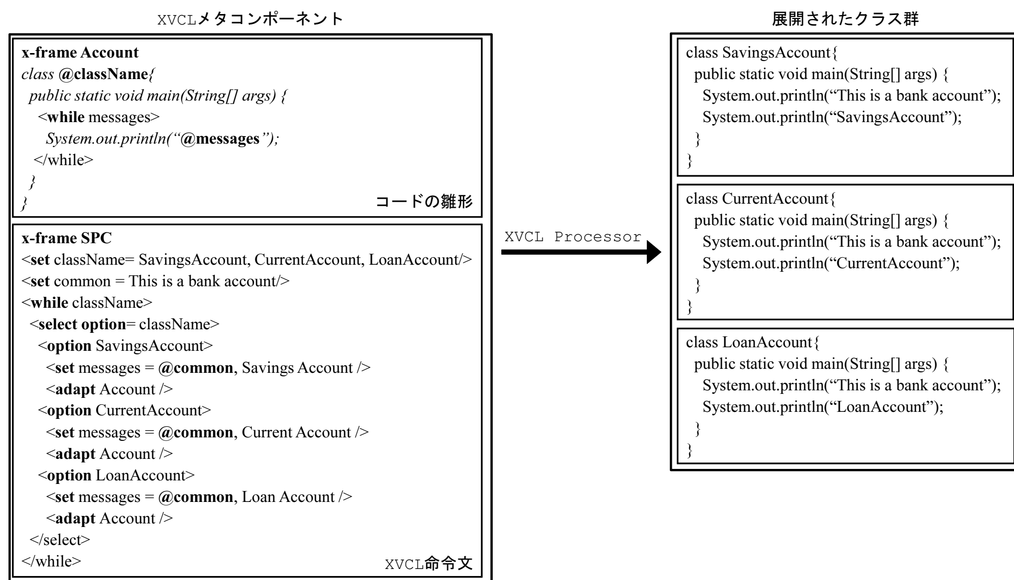


図 8　XVCL メタコンポーネントの例

8 の右下の XVCL 命令文によって展開される．この例では，展開によって 3 つのクラスが生成されている．この例の場合では，XVCL を用いても XVCL 命令文には重複箇所が存在しているが，重複部分を 1 つのファイル内にまとめることができるという利点がある．

実際に，XVCL を用いて多くのコードクローンを集約できたとの報告もある [15] [16]．XVCL に関する情報は [13] のウェブページで得ることができる．

なお，XVCL には，コードクローン検出機能は含まれていない．XVCL を用いる場合には，検出ツールを用いてあらかじめコードクローン情報を得ておく必要がある．

4.5　統合開発環境のプラグイン

Tairas らは，コードクローンを分析するための Eclipse プラグインを開発している [28]．このプラグイン自身はコードクローン検出機能を持たず，既存の検出ツールの出力結果を読み込むことにより，コードクローン情報を扱う．対応しているツールは，字句単位の検出ツールである CCFinder [18]，抽象構文木を用いた検出ツールである CloneDR [1]，Deckard [17]，SimScan [3] である．Eclipse のプラグインであり，開発・保守作業とコードクローンの分析をすべて Eclipse 上で行えるため，効率的に作業を進めることができる．リファクタリング支援としては，コードクローン間の差異部分を強調表示する機能がある．ツールに関する情報は [27] で得ることができるが，ツールは公開されておらず，電子メールなどで直接問い合わせる必要がある．

5　ニーズと今後の展望

本章では，コードクローンのリファクタリングに関するニーズと今後の研究の展望について述べる．

5.1　リファクタリング候補の提示に関して

まずリファクタリング候補となるコードクローンの提示については，複数のファイルに散在しているコードクローンをユーザに示すのが望ましい．バグ修正や機能追加などのためにソースコードを閲覧することにより，1 つのファイル内に存在しているコードクローンはユーザにとって発見することは難しくない．実際に，1 つのクラス内で閉じたリファクタリング (メソッドの抽出など)，はソースコード修正の際に同時に行われることが多いとの報告もある [24]．よって，ユーザが独力では見つけることが難しい複数のファイルに散在するコードクローンをユーザに示す

ことが有益であろう．

また，そのような複数のファイルに存在するコードクローンを除去するには，コードがクラスをまたいで移動するため，リファクタリング後のコードの様子(クラス階層やメソッドのオーバーライドに関する情報等) をユーザに提供することが重要である．そのような情報を提示することにより，ユーザはそのリファクタリングの可否を効率的に決定することができる．このような情報提示には可視化が有効であろう．あるクラスが 3 つの派生クラスを持つ状況を考える．もし，3 つのクラス全てがコードクローンを所有している場合は，そのコードクローンを共通の親クラスに引き上げることは有益なリファクタリングかもしれない．しかし，2 つのクラスのみがコードクローンを所有している場合は，同様のリファクタリングを行うのは不適切かもしれない．しかし，肥後らの位置メトリクスは後者と前者の区別をすることはなく，両者において同じメトリクス値を示す．このような場合，コードクローンを持つクラス群のクラス図等を描画し，コードクローンを持つクラスを強調表示する可視化により，そのような細かい状況の違いを瞬時に把握することができるであろう．

コードクローンをその特徴に基づいて細かく分類し，その分類毎に適用可能なリファクタリングパターンを定めておくことも有益であろう．例えば，クラス内に存在するのか共通の親クラスを持つクラス間に存在するのか，抽出した場合の返り値の必要性，コードクローン間の差異部分に関する情報等を基に分類を行うことが考えられる．特に，経験の浅いプログラマにとっては，コードクローンの特徴を入力するのみで適切なリファクタリング方法を提示する手法は大きな助けになる．しかし，このような分類の構築には，多くの経験，知識，実証的な実験等を必要とするため，単一の研究グループのみで行うには難しいかもしれない．例えば，分類を作成する枠組みを作成しそれを Web システム等を用いて公開することにより，集合知的に構築するやり方もあるだろう．

これまでの研究により，全てのコードクローンがソフトウェア開発および保守に悪影響を与えるわけではないことがわかっている [2] [8]．また，一部のコードクローンは，開発が進むに従って異なる修正が加えられ，コードクローンではなくなる場合もあることが報告されている [19]．このような，将来コードクローンではなくなるものをリファクタリングしてしまうと，かえって開発や保守に支障をきたす場合がある．つまり，開発や保守に悪影響を与えると思われるコードクローンを事前に特定する手法が必要である．

各プログラミング言語に特化したリファクタリング支援手法も有益であろう．たとえば，C++言語は Java 言語や C#言語とは異なり，クラスの多重継承を許容している．よって，C++言語においては，リファクタリング前のクラス階層を壊すことなく，コードクローンを除去するためだけの共通の親クラスを定義することができる．

5.2 振る舞いの保証について

コードクローンの除去により振る舞いが変化していないことを確認するには，回帰テストを行う必要がある．リファクタリング支援ツールは，リファクタリングが回帰テストに与える影響も考慮すべきかもしれない．たとえば，メソッドを定義する場所が子クラスから親クラスに移動するなどした場合，子クラスに対する単体テストケースの一部は，親クラスのテストケースとして実施するのが妥当になるかもしれない(テストケース自体は子クラスに対して実施できるとしても，テストされるべき対象は子クラスではなく親クラスである，ということになるかもしれない)．手動でテストケースを作成するのは，人的および時間的に大きなコストを必要とするため，自動的に必要なテストケースを生成する手法があれば，リファクタリングの大きな助けとなる．また，コードクローンをリファクタリングするための事前条件や事後条件の定式化も必要であろう．これにより，振る舞いを変えることなく除去可能なコードクローンを自動的に特定でき，またそのソースコードを自動変換することも可能になる．リファクタリングの事前条件と事後条件については，研究が始まりつつある．しかし，現在のところ，3 章で紹介したリファクタリングパターンについては，メソッドの引き上げのみが対象となっており [22]，他のコードクローンを除去するパターンについ

ても，事前条件および事後条件の特定が望まれる．

5.3 リファクタリング支援手法の普及について

4章で紹介したように，コードクローンのリファクタリングに関する手法はいくつか提案されてはいるが，それらはまだ広く使われてはいない．それを実現するためには，ユーザが普段利用しているプログラミング環境 (例えば，Eclipse や VisualStudio などの統合開発環境) において，利用可能な機能として提供されることが早道である．さらに，リファクタリング支援をウィザード形式で実現すれば，ユーザはより容易にコードクローンを除去できるであろう．しかし，そのような事柄は，研究というよりも実装や開発という面が強く，研究者にとっては敷居が高い．提案した手法を実用的なツールとして開発および提供するための社会的な枠組みが必要であると著者は感じている．

5.4 ソースコード以外のプロダクトに関して

3章で紹介したように，コードクローンに対するリファクタリングは，クラス階層の変更を伴う場合がある．よって，このようなリファクタリングを行うことにより，設計書など，ソースコード以外のプロダクトとの乖離が発生する場合がある．このような乖離が何度も起こることにより，ソースコードと設計書はまったく対応がとれなくなってしまう．よって，リファクタリングを行った際に，関連するプロダクトも自動的あるいは半自動的に変更を行う，若しくは変更の必要性を通知する等の支援手法が必要である．

5.5 リファクタリング以外の支援

コードクローンの中には，プログラミング言語に適切な抽象化機構が存在しない等の理由により，リファクタリングが難しいものが存在する [19]．このようなコードクローンについては，漏れのない修正支援を行う必要がある．Toomim らは，あるコードクローンを修正すると，それと対応する全てのコードクローンに対して同様の修正を施すエディタを開発している [29]．Duala-Ekoko らも同時修正を行うエディタを Eclipse のプラグインとして開発している [5]．しかし，現段階では，これらのエディタはユーザの入力を，対応する各コードクローンに対してそのまま反映させるため，異なる変数名を用いているコードクローンについては適用することができない．どのようなコードクローンに対するどのような修正であっても適用可能なツールが望まれている．

6 おわりに

本稿では，コードクローンを除去するためのリファクタリング方法，コードクローンの検出技術，およびコードクローンに対するリファクタリング支援手法を紹介した．コードクローンの存在に悩んでいる実務者の方，およびこれからコードクローンのリファクタリングに関する研究を始めようとしている方の一助になれば幸いである．

謝辞 本稿の執筆に当たり多数の助言を頂いた，はこだて未来大学の神谷年洋准教授に感謝する．

参 考 文 献

[1] Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, in *Proc. of International Conference on Software Maintenance 98*, 1998, pp. 368–377.

[2] Bettenburg, N., Shang, W., Ibrahim, W., Adams, B., Zou, Y. and Hassan, A.: An Empirical Study on Inconsistent Changes to Code Clones at the Pelease Level, in *Proc. of the 16th Working Conference on Reverse Engineering*, 2009, pp. 85–94.

[3] Blue Edge Bulgaria: SimScan, http://blue-edge.bg/download.html.

[4] CCFinderX, http://www.ccfinder.net/.

[5] Duala-Ekoko, E. and Robillard, M. P.: Tracking Code Clones in Evolving Software, in *Proc. of the 29th International Conference on Software Engineering*, 2007, pp. 158–167.

[6] Fowler, M.: *Refactoring: improving the design of existing code*, Addison Wesley, 1999.

[7] Gabel, M., Jiang, L. and Su, Z.: Scalable Detection of Semantic Clones, in *Proc. of the 30th International Conference on Software Engineering*, 2008, pp. 321–330.

[8] Göde, N. and Koschke, R.: Frequency and Risks of Changes to Clones, in *Proc. of the 33rd International Conference on Software Engineering*, 2011.

[9] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎: コードクローンを対象としたリファクタリング支援環境, 電子情報通信学会論文誌, Vol. J88-D-I, No. 2(2005), pp. 186–195.

[10] 肥後芳樹, 楠本真二: プログラム依存グラフを用いたコードクローン検出法の改善と評価, 情報処理学会論文誌, Vol. 51, No. 12(2010), pp. 2149–2168.

[11] 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, 電子情報通信学会論文誌 *D*, Vol. J91-D, No. 6(2008), pp. 1465–1481.

[12] 堀田圭佑, 肥後芳樹, 楠本真二: プログラム依存グラフを用いた Template Method パターン適用によるコードクローン集約支援, 情報処理学会研究報告 2011-SE-171, No.14, 2011, pp. 1–8.

[13] Jarzabek, S.: XML-Based Variant Configuration Language - Technology for Reuse, http://xvcl.comp.nus.edu.sg/.

[14] Jarzabek, S.: *Effective Software Maintenance and Evolution: Reused-based Approach*, CRC Press Taylor and Francis, 2007.

[15] Jarzabek, S. and Li, S.: Unifying Clones with a Generative Programming Technique: a Case Study, *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 4(2006), pp. 267–292.

[16] Jarzabek, S. and Shubiao, L.: Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique, in *Proc. of ESEC-FSE'03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003, pp. 237–246.

[17] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-based Detection of Code Clones, in *Proc. of the 29th Intenational Conference on Software Engineering*, 2007, pp. 96–105.

[18] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7(2002), pp. 654–670.

[19] Kim, M., Sazawal, V., Notkin, D. and Murphy, G. C.: An Empirical Study of Code Clone Genealogies, in *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 187–196.

[20] 大阪大学井上研究室: ICCA, http://sel.ist.osaka-u.ac.jp/icca/.

[21] 政井智雄, 吉田則裕, 松下誠, 井上克郎: テンプレートメソッドの形成に基づく類似メソッド集約支援, 日本ソフトウェア科学会 ソフトウェア工学の基礎 XVII, 2010, pp. 125–130.

[22] Mens, T., EetVelde, N. V., Demeyer, S. and Janssens, D.: Formalizing Refactorings with Graph Transformations, *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 17, No. 4(2005), pp. 247–276.

[23] Mens, T. and Tourwe, T.: A Survey of Software Refactoring, *IEEE Transactions on Software Engineering*, Vol. 30, No. 2(2004), pp. 126–139.

[24] Murphy-Hill, E., Parnin, C. and Black, A. P.: How We Refactor, and How We Know It, in *Proc. of the 31th International Conference on Software Engineering*, 2009, pp. 287–297.

[25] Semantic Designs: CloneDR, http://www.semdesigns.com/Products/Clone/.

[26] Simian, http://www.harukizaemon.com/simian/.

[27] Software Composition and Modeling Laboratory, University of Alabama: CeDAR: Clone Detection, Analysis, and Refactoring, http://www.cis.uab.edu/softcom/cedar/.

[28] Tairas, R. and Gray, J.: Get to Know Your Clones with CeDAR, in *Proc. of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 817–818.

[29] Toomim, M., Begel, A. and Graham, S.: Managing Duplicated Code with Linked Editing, in *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004, pp. 173–180.

[30] 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(2007), pp. 1431–1442.