B-06

クラス階層構造を利用したリファクタリング支援手法の改良

Improvement of Refactoring Assistance Based on Class Hierarchy

佐野 由希子 † **肥後 芳樹** † **楠本 真二** †
Yukiko SANO Yoshiki HIGO Shinji KUSUMOTO

1 まえがき

ソフトウェア保守を困難にしている 1 つの要因としてコードクローンが指摘されている．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る．例えば，あるコード片にバグが含まれていた場合，そのコード片のコードクローン全てについて同様の修正を行うかどうかを検討しなければならない．

ソフトウェア保守性を改善する技術の 1 つとして，リファクタリングがある．リファクタリングとは，ソフトウェアの外部的振舞いを変化させることなく，内部の構造を改善する作業のことである [1]．コードクローンを対象としたリファクタリングでは，ソースコード中からリファクタリングに適したコードクローンを検出し，適切なリファクタリングパターンを用いて，単一のモジュールに集約する．

我々はこれまでに，コードクローンに対してどのようなリファクタリングパターンを用いるのが適切か判断する手法を提案している [2]．その手法の中に，メトリクス DCH がある．DCH はクラス階層内において，コードクローン同士がどれだけ離れているかを表す．一般的に，コードクローンはクラス階層において広く分散しているよりも，近い位置に集まっている方がリファクタリングを行いやすい．しかし，DCH はクローンセット[*1]単位で計測されるため，クローンセット内の一部のコード片のみが離れて存在する場合と，全てのコード片がクラス階層のさまざまな位置に分布している場合の区別がつかない．後者はリファクタリングを行うことが難しいが，前者については離れた位置に存在しているコード片以外は容易に集約することができる．

そこで本研究では，このメトリクス DCH を改良する手法の提案と，その手法を実現するツールの試作を行った．また，実際のソフトウェアを対象に適用実験を行った結果，従来の手法よりも多くのリファクタリング可能なコードクローンを検出できることを確認した．

2 従来手法

2.1 メトリクス DCH

メトリクス DCH(S) は文献 [2] にて下記のように定義されている．

クローンセット S はコード片 f_1，f_2，…，f_n を含んでいるとする．C_i はコード片 f_i を含んでいるクラスとする．もしクラス C_1，C_2，…，C_n が共通の親クラスを持つ場合は，その共通の親クラスの中で，クラス階層的に最も下に位置するクラスを C_p で表すとする．また D（C_k,C_h） はクラス C_k と C_h のクラス階層における距離を表すとする．この時，

$$DCH(S) = max\{D(C_1, C_p), \cdots, D(C_n, C_p)\}$$

と表される．直観的には，メトリクス DCH(S) はクローンセット S に含まれる各コード片間のクラス階層内における最大の距離を示す．例えば，S 中の全てのコード片が 1 つのクラス内に存在する場合は DCH(S) の値は 0（図 1(a)），あるクラスとその直接の子クラス内に存在する場合は DCH(S) の値は 1 となる（図 1(b)）．例外的に，コードクローンが存在するクラスが共通の親クラスを持たない場合は DCH(S) の値は ∞ とする（図 1(c)）．なお，このメトリクスは，JDK のクラスライブラリ等の修正不可能なクラスを除外したクラスを対象として計算される．

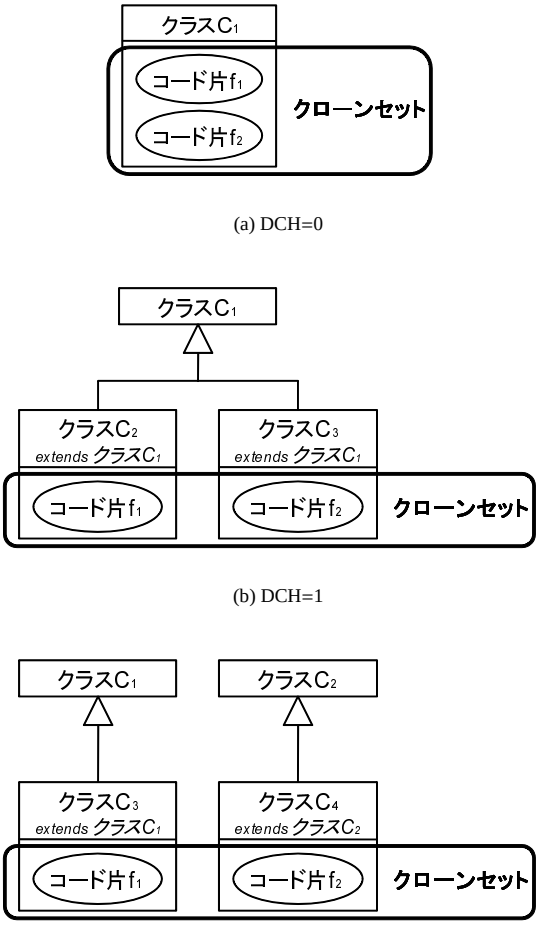

図 1 コード片の位置と DCH

この DCH の値から，コードクローンをどこにリファクタリングできるかが分かる．例えば，DCH=0 ならコードクローンの存在するクラス内に，DCH≥1 なら各コードクローンの存在するクラスに共通する親クラス内に集約を行うことができる．ただし，DCH=∞ の場合は，集約を行うことが難しい．

2.2 問題点

従来の DCH はクローンセット単位で計測されていた．この方法では，DCH

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

[*1] クローンセットとは，互いにコードクローンとなっているコード片の集合である．

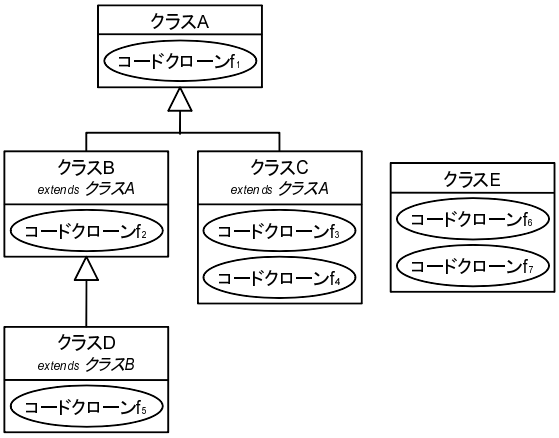

図 2　クローンセット S の例

```
1 for (すべてのクローンセット){
2   group[i]←NULL
3   for (クローンセット内のすべてのコードクローン){
4     group[i]←(group[i],クラスi内のコードクローン)
5   }
6   for (group[i]){
7     DCH(group[i])←0
8   }
9   for (group[i]){
10    P←group[i]の存在するクラスの親クラス
11    h←1
12    G←NULL
13    while(P≠NULL){
14      for (group[j]){
15        if (group[j]のクラスがクラスPの0～h階層下の子クラス)
16          G←(G,group[j])
17        end if
18      }
19      if (Gと同じ組み合わせのサブセットに対してDCHが設定されていない)
20        DCH(G)←h
21      end if
22      P←Pの親クラス
23      h←h+1
24    }
25  }
26 }
```

図 3　アルゴリズム

の値が大きくなってしまう場合がある．クローンセット S に含まれているコードクローンを f_1，f_2，…，f_7 とし，これらが図 2 のように分布している場合を考える．この場合，クラス B はクラス D の親クラス，クラス A はクラス B とクラス C の親クラスである．f_6 と f_7 の存在するクラス E は，他のどのクラスとも共通する親クラスを持っていない．もしクローンセット S 内に f_6 と f_7 がなければ $DCH(S) = 2$ となるが，f_6 と f_7 のために DCH の値が ∞ になってしまう．

しかし，それはリファクタリングを行う上において好ましくない．同一クラス内に存在するクローンのみを集約したい，より多くのコード片を集約したい等の，細かなリファクタリング要求に応えにくいためである．

3　提案手法

3.1　概要

従来手法の問題点を解決するため，本研究では，クローンセット内のコードクローンをサブセットに分類し，そのサブセットごとに DCH を計測する手法を提案する．サブセットは全ての組み合わせで作成するのではなく，DCH の値が 0 となるサブセット，DCH の値が 1 となるサブセット，…のように DCH の値毎に作成する．この手法を用いることによって，より多くのリファクタリング可能なコード片を検出し，より柔軟にリファクタリングの候補を提示することができる．

3.2　アルゴリズム

提案手法のアルゴリズムを図 3 に示す．3～5 行目では，同一クラス内に存在するコードクローンを 1 つの group として分類している．そのため，各 group 内のコードクローンの集合は DCH=0 のサブセットとなるので，6～8 行目で各 group に対する DCH を 0 と設定している．9～25 行目では DCH が 1 以上のサブセットを探索している．まず，ある group が存在するクラスの 1 階層上の親クラスを基準とし，そのクラスを共通の親クラスとする DCH=1 のサブセットを探索する．その後，基準としたクラスの 1 階層上の親クラスを新たな基準とし，探索するサブセットの DCH の値も 1 増やして繰り返す．これをすべてのグループに対して行う．

3.3　従来手法との比較

この提案手法を使うことにより，従来の手法では $DCH(S) = \infty$ となってしまっていた図 2 のクローンセット S の DCH が，以下のように計測される．

- $DCH = 0$ のサブセット
  - $\{f_3, f_4\}$
  - $\{f_6, f_7\}$
- $DCH = 1$ のサブセット
  - $\{f_1, f_2, f_3, f_4\}$
  - $\{f_2, f_5\}$
- $DCH = 2$ のサブセット
  - $\{f_1, f_2, f_3, f_4, f_5\}$
- $DCH = \infty$ のサブセット
  - $\{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}$

この結果から集約の一例として，$DCH = 2$ のサブセット $\{f_1, f_2, f_3, f_4, f_5\}$ の 5 つのコードクローンをクラス A に集約し，更に，$DCH = 0$ のサブセット $\{f_6, f_7\}$ の 2 つのコードクローンもクラス E に集約することができる．そうすると，7 つあったコードクローンを 2 つに減らすことができる．

4　実験

リファクタリング支援環境 Aries[2] の出力するコードクローン情報を入力とし，各クローンセットのサブセットごとの DCH を出力とするツールを実装した．このツールを用いて，提案手法の有効性を調べるための実験を行った[*2]．対象ソフトウェアのクローンセット数を l，クラス階層の階層数を m，クローンセット内の group 数を n とすると，アルゴリズムの計算時間は $O(lmn^2)$ である．しかし，10 万行規模のソフトウェアでもクローンセット数は数百個程度，階層数は数個程度，group 数は数個から十数個程度であるため，ツールの実行時間はさほど長くはなく，数秒以内で処理が終了した．

従来手法を用いた場合と提案手法を用いた場合において，リファクタリング可能と判断されるコードクローンの数にどの程度差が出るかを計測した．表 2，表 3 に表しているオープンソースソフトウェア 10 種を実験の対象とし，それらのソフトウェアに後述するリファクタリングパターン “Pull Up Method” と “Extract Method” の 2 つを適用した．なお，Aries で計測するコードクローンの最小一致トークン数は 30 トークン (5,6 行程度) 以上と設定した．

Pull Up Method

ある親子クラス関係が存在した場合に，子クラスに存在するメソッドを親クラスに引き上げる手法である．図 4 のよ

[*2] 実験を行った環境は CPU が Pentium4 2.66GHz，主記憶容量が 2.00GB，OS が Windows XP/Professional SP2 である．

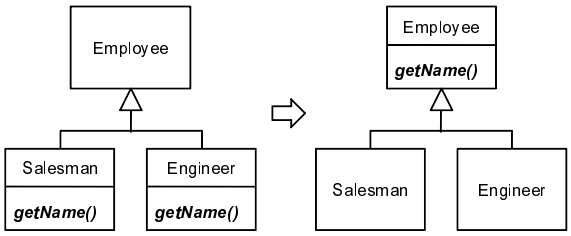

図 4　Pull Up Method

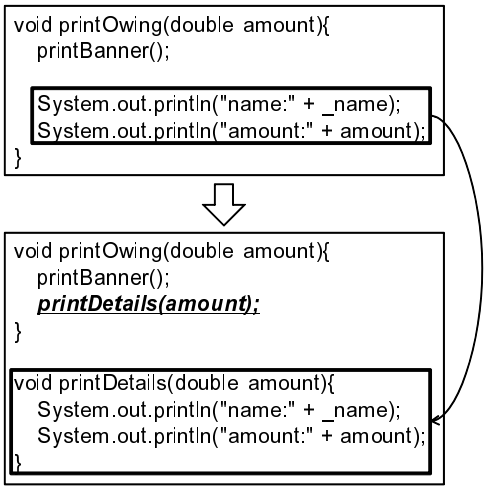

図 5　Extract Method

うに複数のクラスに重複したメソッドが存在し，それらのクラスが共通の親クラスを持っている場合は，そのメソッドを親クラスに引き上げることでリファクタリングを行うことができる．

Extract Method

長すぎるメソッドや複雑な処理の一部分を新たなメソッドとして抽出する手法である．図 5 のように，抽出部分を新たなメソッドとして定義し，抽出箇所は定義したメソッドへの呼び出しに置き換える．これをコードクローンに適用すれば，重複したコード片をリファクタリングすることができる．特に，全てのコードクローンが同一のクラス内に存在する場合は容易にリファクタリングが可能である．

“Pull Up Method”，“Extract Method” を用いてリファクタリングを行うための条件については，文献 [2] を参照されたい．

4.1　実験結果

実験結果を表 2 に示す．この表は，実験対象のソフトウェアを従来手法と提案手法のそれぞれに適用し，リファクタリングパターン “Pull Up Method” と “Extract Method”，それぞれの条件に一致するコードクローンのコード片数を計測した結果である．差分には提案手法で計測したコード片数から従来手法で計測したコード片数を引いた値を示している．なお，表中の記号 a〜j は，表 1 と対応している．

表 2 より，提案手法を用いた場合，従来手法を用いた場合よりも “Pull Up Method” では 81 個，“Extract Method” では 195 個多くのコード片が検出されている．

4.2　考察

表 2 より，従来手法と提案手法とで差が出ないものもあるが，多くの場合において提案手法を用いて検出したコード片数が，従来手法を用いて検出したコード片数を上回った．

対象ソフトウェアの規模と，表 2 における “Pull Up Method”，“Extract Method” の差分とを比較するため，表 3 を示す．この表中の記号 a〜j も，表 1 と対応している．また，“Pull Up”，“Extract” は，それぞれ “Pull Up Method”，“Extract Method” の略である．表 3 より，行数の多いものは差分も多い傾向がある．10 万行以上の規模のソフトウェアについては，従来手法と提案手法とでコード片 6 個分以上の差が出ている．規模が 10 万行未満のソフトウェアは，かなりの差が出るものもあれば，まったく差が出ないものもあった．また，ファイル数と差分には関連性は見られなかった．

本手法を用いて新たに見つかったリファクタリング可能なコードクローンを調査した結果，同じディレクトリ内のファイル同士に存在しているものが多いことが判明した．同じディレクトリ内のファイルは似た機能を実装していることが多いので，これらのコードクローンは意味的に似たものである可能性が高い．例えば，4 つのコードクローンが

- org\apache\tools\ant\taskdefs\optional\ejb\BorlandGenerateClient.java
- org\apache\tools\ant\taskdefs\optional\XMLValidateTask.java
- org\apache\tools\ant\taskdefs\Property.java
- org\apache\tools\ant\util\ClasspathUtils.java

の 4 つのファイルに分布するクローンセットは，従来手法では DCH=∞ となったが，提案手法では taskdefs ディレクトリ内のファイルに分布する 3 つのコードクローンは，DCH=1 でリファクタリング可能と判断された．4 つのファイルの内，ClasspathUtils.java ファイルは ant のクラスパスに関する機能を提供しているが，それ以外の taskdefs ディレクトリ内の 3 つのファイルは，どれも ant のタスクの動作を定義している．よって，この 3 つのファイルに分布するコードクローンは意味

表 1　実験対象のソフトウェア

a	Apache Ant(v1.6.0)
b	Areca backup(v5.5.6)
c	Eclipse Checkstyle Plug-in(v4.4.0)
d	JasperReports-Java Reporting(v2.0.4)
e	jEdit(v4.2)
f	JFreeChart(v1.0.9)
g	Jimm-Mobile Messaging(v0.5.1)
h	Robocode(v1.5.2)
i	XUI RIA Framework(v1.0.4)
j	ZK-Simply Ajax and Mobile(v3.0.2)

表 2　リファクタリング可能なコード片の数

	Pull Up Method			Extract Method		
	従来手法	提案手法	差分	従来手法	提案手法	差分
a	65	68	3	68	72	4
b	37	48	11	30	44	14
c	9	9	0	20	24	4
d	292	309	17	143	205	62
e	3	8	5	153	155	2
f	262	307	45	176	275	99
g	0	0	0	28	28	0
h	17	17	0	60	60	0
i	34	34	0	43	47	4
j	47	47	0	66	72	6
合計	766	847	81	787	982	195

表 3　ソフトウェアの規模との比較

	Pull Up	Extract	ファイル数	行数
a	3	4	650	189,915
b	11	14	323	55,179
c	0	4	206	43,349
d	17	62	948	211,619
e	5	2	394	140,665
f	45	99	538	194,470
g	0	0	56	27,166
h	0	0	209	48,717
i	0	4	439	76,770
j	0	6	961	132,597

的に似ている可能性が高いといえる．このようなコードクローンは集約してよいと思われる．

対象ソフトウェアによって，その差分の量は大きく変わるものの，従来の手法を用いるよりも本手法を用いた方が多くのリファクタリング可能なコード片を検出できた．更に，新たに検出されたリファクタリング可能なコード片は，リファクタリングを行ってよいものである可能性が高いことも分かった．また，対象ソフトウェアの規模が大きいほど，差分が大きくなる傾向があるので，規模の大きいソフトウェアに本手法を用いれば，リファクタリング可能なコード片を多く検出するのに役立つと考えられる．ただし，本手法はリファクタリング可能なコード片を検出することを目的としており，実際にリファクタリングすべきかどうかは人間が見て判断する必要がある．

5　関連研究

コードクローンを集約する技術には次のようなものがある．

Baxter らのツール CloneDr は，構文解析をして作成した抽象構文木を用いてコードクローンを検出する [3]．更に，検出したコードクローンを集約するための雛形も出力する．この雛形によって，どのようにソースコードを修正すれば集約を行えるのかが分かる．

Balazinska らは，メソッド単位で検出したコードクローンに対する集約支援手法を提案している [4]．この手法では，コードクローンに関する 2 点に焦点を当てている．1 つ目はコードクローン間の違いで，これを用いることによって，各コードクローンの集約が容易かどうかを判断することができる．2 つ目はコードクローンとその周囲との結合度である．結合度が低ければ，コードクローンを他の部分に移動させるのは容易だが，結合度が高ければ，移動させるのは困難である．

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，大きい単位でのコードクローン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 [5][6]．XVCL を用いることで，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には，使用されているプログラミング言語による記述と，そのメタコンポーネントがどのようにしてコンパイル可能なクラスに展開されるかを記述した XVCL の命令文が含まれている．この XVCL を用いることによって，多くのコードクローンを集約できたとの報告もなされている [7][8].

6　あとがき

本研究では，メトリクス DCH を改良する手法を提案した．従来の手法ではクローンセットごとに DCH を設定していたのに対し，本手法ではクローンセット内のコードクローンをサブセットに分類し，そのサブセットごとに DCH を設定している．

また，本手法の方がリファクタリング可能なコードクローンを多く検出できることを実験で確認した．

7　謝辞

本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，文部科学省科学研究費補助金基盤研究 (A)(課題番号:1720001)，(C)(課題番号:20500033)，および若手研究 (スタートアップ)(課題番号:19800022) の助成を得た．

参考文献

[1] M. Fowler: “Refactoring: improving the design of existing code”, Addison Wesley (1999). (児玉公信, 友野晶夫, 平澤章, 梅澤真史　訳 (2000)『リファクタリング　プログラミングの体質改善テクニック』. ピアソン・エデュケーション).

[2] 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, **J88-D-I**, 2, pp. 186–195 (2005).

[3] I. Baxter, A. Yahin, M. A. L. Moura and L. Bier: “Clone detection using abstract syntax trees”, Proc. of International Conference on Software Maintenance 98, pp. 368–377 (1998).

[4] M. Balazinska, E. Merlo, M. Dagenais, B., K. Kontogiannis：“Advanced clone-analysis to support object-oriented system refactoring”, Proc. of the 7th IEEE International Working Conference on Reverse Engineering, pp. 98–107 (2000).

[5] “Xml-based variant configuration language-technology for reuse”. http://xvcl.comp.nus.edu.sg/.

[6] S. Jarzabek: “Effective Software Maintenance and Evolution: Reused-based Approach”, CRC Press Taylor and Francis (2007).

[7] S. Jarzabek and L. Shubiao: “Eliminating redundancies with a “ composition with adaptation ” meta-programming technique”, Proc. of ESECFSE’03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 237–246 (2003).

[8] S. Jarzabek and S. Li: “Unifying clones with a generative programming technique: a case study”, Journal of Software Maintenance and Evolution: Research and Practice, **18**, 4, pp. 267–292 (2006).