The University of Osaka
Institutional Knowledge Archive

Title	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
Author(s)	吉田, 則裕; 肥後, 芳樹; 神谷, 年洋 他
Citation	情報処理学会論文誌. 2007, 48(3), p. 1431-1442
Version Type	VoR
URL	https://hdl.handle.net/11094/50191
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

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

吉田則裕[†] 肥後芳樹[†] 神谷年洋[††]
楠本真二[†] 井上克郎[†]

コードクローンとは，互いに一致または類似したコード片（ソースコードの断片）を持つコード片を意味し，ソフトウェア保守を困難にしている要因の 1 つとされている．たとえば，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すべてに対して，修正の是非を検討する必要がある．コードクローンを取り除く方法として，リファクタリングの適用が考えられる．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．しかし，クローンセット（互いに一致または類似したコード片の集合）に含まれるコード片と周辺のコード片間に依存関係が存在すると，リファクタリングが困難になる場合がある．本稿は，クローンセット間の依存関係を利用したリファクタリング支援手法を提案する．まず，異なるクローンセットに含まれるコード片間の依存関係に着目し，そのような依存関係を持つコード片の集合をチェーンドクローンセットと定義する．そして，メトリクスを用いてチェーンドクローンセットの特徴を判定し，適用可能なリファクタリングパターンを提示する．最後に，リファクタリング支援ツールとして実装することで，いくつかのオープンソースソフトウェアに適用し，有効性の評価を行う．

On Refactoring Support Based on Code Clone Dependency Relations

Norihiro Yoshida,[†] Yoshiki Higo,[†] Toshihiro Kamiya,[††]
Shinji Kusumoto[†] and Katsuro Inoue[†]

Code clone is a set of code fragments identical or similar to each other. It is generally said that code clone is one of the factors that make software maintenance more difficult. If we modify one of them, it is necessary to determine whether or not we have to modify the others. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. However, there are dependency relations between code fragments belonging to the different clone sets (a clone set is an equivalence class of code clones), and it is difficult to apply refactoring to such code clones. In this paper, we propose a refactoring support method by using dependency relations. At first, we focus on dependency relations between code fragments belonging to the different clone set, and we define "chained clone set" as such code fragments. Then, we define the metrics for providing an appropriate refactoring pattern to each "chained clone set". Finally, we present the "chained clone set" refactoring support tool that we have developed, together with some case studies.

1. はじめに

ソフトウェア保守を困難にする要因の 1 つとして，コードクローンが指摘されている[1)~5)]．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，または類似したコード片（ソースコードの断片）を意味する．たとえば，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すべてに対して，修正の是非を検討する必要がある．このような作業に要するコストが，特に大規模ソフトウェアの保守において問題となる．

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[6)] 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技術のことである．Fowler は，リファクタリングを検討すべき箇所に現れる特徴を Bad Smell と呼び，その代表例としてコードクローン（Duplicated Code）をあげている．また，コードクローンを単一のモジュールに集約する手法として，"Pull Up Method" や "Extract Method"，"Extract

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†† 産業技術総合研究所
National Institute of Advanced Industrial Science and Technology

SuperClass” 等のリファクタリングパターンを紹介している[6]．

これまでに，我々はコードクローン検出ツール CCFinder[7] およびリファクタリング支援環境 Aries[8] を開発してきている．CCFinder は，ソースコードに字句解析と正規化処理を行うことで得られたトークン列の同値性に基づいてコードクローン検出を行う．CCFinder の特徴は，表現上の差異があるコードクローンを検出できること，および百万行単位のソースコードであっても実用時間で解析できることである．Aries は，CCFinder の出力情報を基に，リファクタリングに適した単位（e.g. クラス，メソッド単位）でクローンセット（互いに一致または類似したコード片の集合）を検出し，さらにメトリクスで特徴付けすることでリファクタリングパターンの提示を行う．

これまでに，Aries を用いて様々なソースコードを解析した結果，異なるクローンセットに含まれるコード片間に依存関係が存在する場合が確認されている．たとえば，クローンセット S_a に 2 つのメソッド m_{a1}，m_{a2} が含まれ，同様にクローンセット S_b に 2 つのコード片 m_{b1}，m_{b2} が含まれるときに，メソッド m_{a1} がメソッド m_{b1} を呼び出し，メソッド m_{a2} がメソッド m_{b2} を呼び出しているという場合である．

Aries は，上述した呼出関係の解析は行っていないため，ユーザは自らクローンセット S_a と S_b 間の呼び出し関係を把握する必要がある．もし，ユーザが S_a に対して m_{a1} と m_{b1}，m_{a2} と m_{b2} の呼び出し関係を考慮せずに集約を試みると，呼び出し関係が保存されない可能性がある．有効なリファクタリング支援を行うためには，クローンセット S_a と S_b は，まとめてユーザに提示するべきであると考えられる．

本稿では，クローンセット間の依存関係を利用したリファクタリング支援手法を提案する．まず，異なるクローンセットに含まれるコード片間の依存関係に着目し，そのような依存関係を持つコード片の集合をチェーンドクローンセットと定義する．そして，チェーンドクローンセットの特徴に応じて，適用可能なリファクタリングパターンを提示するためのメトリクスを定義する．最後に，提案手法をリファクタリング支援ツールとして実装し，2 つのオープンソースソフトウェアに適用することで有効性の評価を行う．

2. コードクローン

2.1 コードクローンの定義

あるトークン列中に存在する 2 つの部分トークン列 α，β が等価であるとき，α と β は互いにクローンであるという．またペア（α，β）をクローンペアと呼ぶ．α，β それぞれを真に包含するいかなるトークン列も等価でないとき，α，β を極大クローンと呼ぶ．また，クローンの同値類をクローンセットと呼ぶ．ソースコード中でのクローンを特にコードクローンという[9]．

2.2 コードクローン検出ツール：CCFinder

CCFinder はプログラムのソースコード中に存在する極大クローンを検出し，その位置をクローンペアのリストとして出力する．検出されるコードクローンの最小トークン数はユーザが前もって設定できる．

CCFinder のコードクローン検出手順は以下の 4 つの STEP からなる．

STEP1（字句解析）：ソースファイルを字句解析することによりトークン列に変換する．入力ファイルが複数の場合には，個々のファイルから得られたトークン列を連結し，単一のトークン列を生成する．

STEP2（変換処理）：実用上意味を持たないコードクローンを取り除くこと，および，些細な表現上の違いを吸収することを目的とした変換ルールによりトークン列を変換する．たとえば，この変換により変数名は同一のトークンに置換されるので，変数名が付け替えられたコード片もコードクローンであると判定することができる．

STEP3（検出処理）：トークン列の中から指定された長さ以上一致している部分をクローンペアとしてすべて検出する．

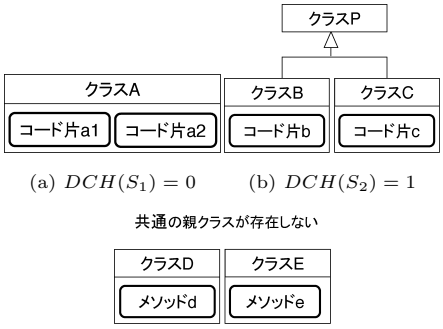
STEP4（出力整形処理）：検出されたクローンペアについて，ソースコード上での位置情報を出力する．

2.3 リファクタリング支援環境：Aries

リファクタリング支援環境 Aries は，CCFinder の出力情報を基に，リファクタリングに適した単位（e.g. クラス，メソッド単位）でクローンセットを検出し，さらにメトリクスで特徴付けすることでリファクタリングパターンの提示を行う．

リファクタリングに適した単位とは，ソースコード上の構造的なまとまりのことである．クローンセットに含まれるコード片が構造的なまとまりを持っているなら，容易に集約することができる．現在，Aries は Java 言語を対象として実装されているため，用いる構造的なまとまりは以下の 12 種類である．

宣言	:	class { }, interface { }
メソッド	:	メソッド本体，コンストラクタ，スタティックイニシャライザ



(a) $DCH(S_1) = 0$ (b) $DCH(S_2) = 1$

(c) $DCH(S_3) = \infty$

図 1 DCH メトリクスの算出例
Fig. 1 Example of DCH metric calculation.

文 : if, for, while, do, switch, try, synchronized

クローンセットの特徴付けに用いるメトリクスの 1 つとして，分散度メトリクス **DCH(S)** [8] について説明する．クローンセット S はコード片 f_1, f_2, $\cdots$, f_n を含んでいるとする．クラス 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, $\cdots$, 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位（最も深い階層）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．このとき，

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

と表される．直観的には，$DCH(S)$ メトリクスはクローンセット 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．**図 1** (a)〜(c) は，それぞれクローンセットに含まれる 2 つのコード片に対して，$DCH(S)$ メトリクスを算出した例である．$DCH(S)$ の値は，すべてのコード片が 1 つのクラス内に存在する場合は 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 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 ∞ とする（図 1 (c)）．このメトリクスは，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．

$DCH(S)$ メトリクスにより，クローンセット S のコード片を集約したモジュールを置くことができるクラス（集約先）を特定することができる．たとえば，$DCH(S)$ メトリクスの値が 1 の場合は，そのクローンセットが存在するクラスの親クラスに集約できることが分かる．また，$DCH(S)$ メトリクスの値が ∞ の場合は，分析対象内に集約先位置になるクラスが存在しないため，クラスの作成，もしくは継承関係のないクラスへの集約を検討するべきであることが分かる．

3. 提案手法

3.1 本研究の動機

リファクタリングを行う際は，“保守性を悪化させるコード（Bad Smell）の特定” と “外部的振舞いを変更させず保守性を向上させる修正作業” を行う必要がある．

すでに述べたように，保守性を悪化させるコードとしてコードクローンが指摘されている．門田らは，コードクローンを含むモジュールの方が保守コストが大きくなる事例を紹介している[10]．彼らが紹介した事例では，より行数の多いコードクローンを含むモジュールほど，保守コストが大きくなっていた．

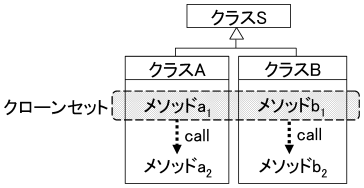
検出したコードクローンを容易にリファクタリングするためには，次の 2 つの条件を満たしていることが望ましい．

（条件 1） クローンセットに含まれるコード片が構造的なまとまりを持っている．

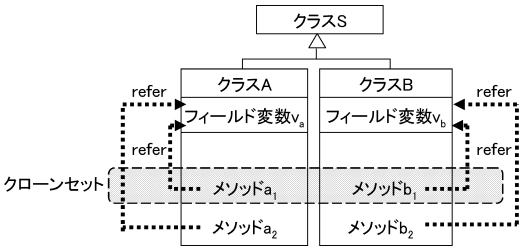
（条件 2） クローンセットに含まれるコード片と周辺のコード間に結合が少ない．

（条件 1）については，2.3 節で述べている．（条件 2）の例として，メソッド呼び出しを含む場合（**図 2** (a)）や，フィールド変数を使用（参照，代入）している場合（図 2 (b)）があげられる．図 2 (a) では，メソッド a_1 と b_1 がコードクローンとなっており，点線矢印で表すようにメソッド a_1 が a_2 を呼び出し，メソッド b_1 が b_2 を呼び出している．図 2 (b) では，メソッド a_1 とメソッド a_2 はフィールド変数 v_a を参照しており，メソッド b_1 とメソッド b_2 はフィールド変数 v_b を参照している．このような周辺のコードと結合度が高いコードクローンを集約することは困難である．なぜなら，集約作業を行う際に，メソッド間の呼び出し関係やメソッドとフィールド変数の関係に不整合が起きないよう考慮する必要があるからである．

ここまでに述べたように，コード量が大きいコードクローンにかかる保守コストや，周辺のコードと結合が多いコードクローンに対するリファクタリングの難しさが問題になりやすい．我々はコードクローン分析を行う中で，そのようなコードクローンのパターンを見つけた．それは，異なるクローンセット間に依存関係が存在する場合である（**図 3** (a)）．図 3 (a) は，クラス A, B 間にまたがる 3 つのクローンセットを表している．これら 3 つのクローンセットは，他のクローンセットとの間にメソッド呼び出し関係やフィールド変



(a) メソッド呼び出し



(b) フィールド変数の使用

図 2 周辺のコードと結合している例
Fig. 2 Examples of codes are coupled each other.

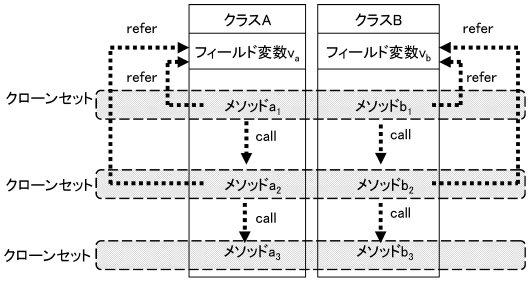
数の参照関係が存在する．図 3 (a) のクラス A，B のように，類似したクラスには，“Extract SuperClass” パターンの適用を検討すべきと指摘されている[6]．クラス A，B に “Extract SuperClass” パターンの適用した結果が図 3 (b) である．

図 3 (a) から図 3 (b) へのリファクタリングを支援する方法として，図 3 (a) 中に含まれる 3 つのクローンセットを同時に提示することが考えられる．

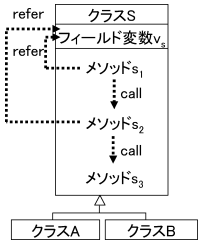
その理由の 1 つ目は，1 つのクローンセットを対象としたリファクタリングは，呼び出し関係が原因で困難な場合があるからである．たとえば，図 3 (a) のクローンセットの 1 つ（メソッド a_1 とメソッド b_1）に対して集約を試みると，親クラスに新たに作成したメソッド s_1 から子クラスのメソッド a_2 と b_2 を呼び出すことができなくなる（図 3 (c)）．この問題を解決するためには，子クラスのメソッドに対応する抽象メソッドを親クラスに追加する必要がある．3 つのクローンセットを同時にリファクタリングする際にはこのような工夫は必要ない．

2 つ目の理由は，図 3 (a) のようなクローンセットの組合せを手作業で見つけ出すことは難しいからである．なぜなら，大規模ソフトウェアは大量のクローンセット含んでいることが多いため，それらのすべてに対して呼び出し関係の有無を確認することは困難だからである．また，呼び出し関係の理解支援にはコールグラフ（Call Graph）がよく用いられるが，頂点数（メソッド数）が膨大になりやすい大規模ソフトウェアへの適用は現実的でない．

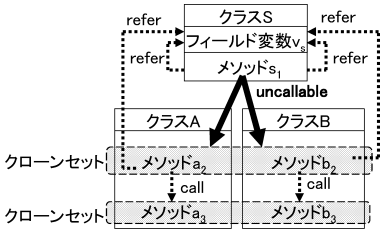
本稿では，図 3 (a) のようなクローンセットの組合せを “チェーンドクローンセット” と呼び，“チェーンドクローンセット” に対するリファクタリング支援手法を提案する．



(a) リファクタリング前



(b) リファクタリング後（3 つ同時）



(c) リファクタリング後（1 つのみ）

図 3 異なるクローンセット間に依存関係が存在する例
Fig. 3 Example of clone sets that have dependency relations.

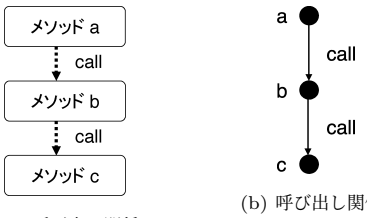
3.2 チェーンドクローン

チェーンドクローンを定義するための準備として，メソッドチェーンを定義する．メソッドの集合が与えられたとき，それらメソッド間の依存関係を表す有向グラフが連結グラフになるなら，そのメソッドの集合をメソッドチェーンと定義する．

ここで扱う依存関係は，以下の 2 種類である．

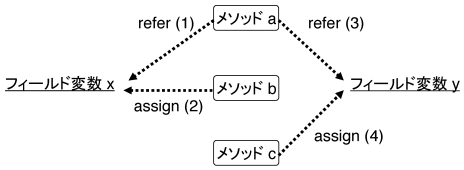
(1) メソッドの呼び出し関係
(2) 同一フィールド変数の共有（参照または代入）

図 4 (a) は呼び出し関係を含むメソッドチェーンの例である．この例では，メソッド a が b を，メソッド b が c を呼び出している．また，図 4 (b) は，図 4 (a) のメソッドチェーンの依存関係をラベル付き有向グラフで表したものである．有向辺に付属しているラベル

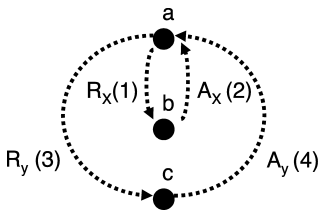


(a) 呼び出し関係　(b) 呼び出し関係を表すグラフ

図 4 呼び出し関係と呼び出し関係を表すグラフ
Fig. 4 Method chain whose dependency relations are method invocations.



(a) フィールド変数の使用



(b) フィールド変数の共有関係を表すグラフ

図 5 フィールド変数の使用と共有関係を表すグラフ（括弧内の数字は対応する関係を表す）
Fig. 5 Chained method whose dependency relations are sharing variables.

“call” は，依存関係の種類がメソッドの呼び出し関係であることを表している．たとえば，メソッド a が b を呼び出しているとき，有向辺 (a, b) を引き，ラベル “call” を付ける．なお，1 つのメソッドが同一のメソッドを 2 回呼び出している場合は，それらメソッド間に呼び出し関係を表す有向辺を 2 本追加する．

図 5 (a) はフィールド変数を使用しているメソッドチェーンの例である．図 5 (a) の有向辺に付属しているラベル “refer” はフィールド変数の参照を表しており，“assign” はフィールド変数への代入を表している．図 5 (a) のメソッド a と b は，フィールド変数 x を共有しており，メソッド a，c はフィールド変数 y を共有している．また，図 5 (b) は，図 5 (a) のメソッドチェーンに含まれるメソッド間の共有関係をラベル付き有向グラフで表したものである．有向辺に付属しているラベル “$R_x(refer)$” はフィールド変数 x への参照による共有関係，ラベル “$A_y(assign)$” は，フィールド変数 y への代入による共有関係を表している．たとえば，メソッド a が変数 x を参照し，かつメソッド b が変数 x を使用（参照または代入）しているとき，有向辺 (a, b) を引き，ラベル “R_x” を付ける．また，メソッド c が変数 y に代入し，かつメソッド a が変数 y を使用しているとき，有向辺 (c, a) を引き，ラベル “A_y” を付ける．

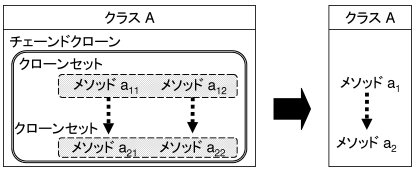


図 6 ケース 1
Fig. 6 Case 1.

次に，メソッドチェーンを用いてチェーンドクローンを定義する．2 つのメソッドチェーンが互いにチェーンドクローンとなるのは，各メソッドチェーンが持つ依存関係のグラフが同形であり，対応する頂点（メソッド）が同一クローンセットに含まれ，対応する辺（依存関係）のラベルは等しいときである．また，互いにチェーンドクローンであるメソッドチェーンの同値類を，チェーンドクローンセットと呼ぶ．

3.3 チェーンドクローンセットに対するリファクタリング

ここでは，チェーンドクローンセットに対して考えられるリファクタリングについて説明するため，適用可能なリファクタリングパターンが異なる 4 つのケースを紹介する．

ケース 1 は，チェーンドクローンセットが 1 つのクラスに包含されている場合である．ケース 1 では，その 1 つのクラス内にすべてのクローンセットを集約可能である．**図 6** は，ケース 1 のチェーンドクローンセットに対するリファクタリングの例である．互いにクローンであるメソッド a_{11} とメソッド a_{12} を集約しメソッド a_1 とし，同様に互いにクローンであるメソッド a_{21} とメソッド a_{22} を集約しメソッド a_2 としている．

ケース 2 は，チェーンドクローンセットが以下の 2 つの条件を満たす場合である．

- チェーンドクローンセットに含まれるメソッドは，すべて兄弟クラスに属する．
- 各メソッドチェーンは，それぞれ 1 つのクラスに包含されている．

ケース 2 は，“Pull Up Method” パターンを適用することで，リファクタリングできる．つまり，兄弟クラスにまたがって存在するクローンセットを，親クラスに集約することでリファクタリングできる．**図 7** (a)

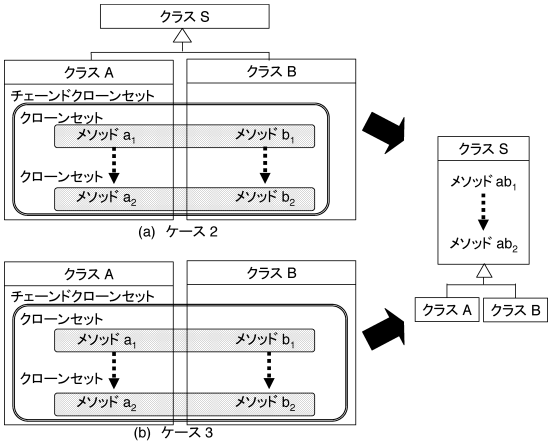


図 7 ケース 2 とケース 3
Fig. 7 Case 2 and Case 3.

は，ケース 2 に対するリファクタリングの例である．この例では，兄弟クラスであるクラス A，B にまたがって存在する 2 つのクローンセットを，親クラス S に作成したメソッドに集約している．具体的には，クラス A のメソッド a_1 とクラス B のメソッド b_1 を集約し，親クラス S のメソッド ab_1 とし，同様にクラス A のメソッド a_2 とクラス B のメソッド b_2 を集約し，親クラス S のメソッド ab_2 としている．

ケース 3 は，チェーンドクローンセットが以下の 2 つの条件を満たす場合である．

- チェーンドクローンセットに含まれるメソッドを持つクラスは，いずれも共通の親クラスを持たない．
- 各メソッドチェーンは，それぞれ 1 つのクラスに包含されている．

ケース 3 は，“Extract SuperClass” パターンを適用することで，リファクタリングできる．つまり，チェーンドクローンセットに含まれるメソッドを持つクラスに対して，共通の親クラスを作成し，クラス間をまたがって存在するクローンセットを，新たに作成した親クラスに集約することでリファクタリングできる．図 7 (b) は，ケース 3 に対するリファクタリングの例である．この例では，まず共通の親クラスを持たない 2 つのクラス A，B に，共通の親クラス S を作成している．その後，ケース 2 と同様に兄弟クラスとなったクラス A，B にまたがって存在する 2 つのクローンセットを，親クラス S に作成したメソッドに集約している．

ケース 4 は，チェーンドクローンセットが以下の条件を満たす場合である．

- 各メソッドチェーンは，複数のクラスにまたがって存在する．つまり依存関係が複数のクラス間にまたがっている．

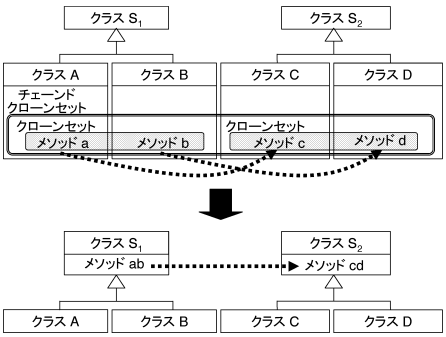


図 8 ケース 4
Fig. 8 Case 4.

ケース 4 は，チェーンドクローンセット単位でリファクタリングできない場合である．だが，チェーンドクローンセットを複数のクローンセットとして扱い，それぞれの親クラスに集約することでリファクタリングできるため，クローンセット単位でのリファクタリングを検討すべきである．**図 8** は，ケース 4 にクローンセット単位でのリファクタリングを適用した例である．この例では，兄弟クラスであるクラス A，B にまたがって存在する 2 つのクローンセットを，それぞれの親クラスに作成したメソッドに集約している．具体的には，クラス A のメソッド a とクラス B のメソッド b を集約し，親クラス S_1 のメソッド ab とし，同様にクラス C のメソッド c とクラス D のメソッド d を集約し，親クラス S_2 のメソッド cd としている．

3.4 チェーンドクローンセットの分類

前節の 4 つのケースのように，チェーンドクローンセットを分類する．前節の 4 つのケースには，それぞれ適合するための条件があった．それらは，次の 2 つである．

C1 チェーンドクローンセットに含まれるメソッドが所属するクラス間の関係についての条件

C2 メソッドチェーンに含まれるメソッドが所属するクラス間の関係についての条件

ここでのクラス間の関係とは，クラス階層上の関係のことである．クラス間の関係は，次に 3 つに分類できる．

R1 同一クラス

R2 共通の祖先クラスを持つクラス

R3 共通の祖先クラスを持たないクラス

条件の種類とクラス間の関係を組み合わせることにより，チェーンドクローンセットを**表 1** のように分類できる．

表 1 チェーンドクローンセットの分類
Table 1 Categorization of chained clone sets.

C1 \ C2	R1	R2	R3
R1	分類 1	分類 4	
R2	分類 2		
R3	分類 3		

各分類に属するチェーンドクローンセットに対して，以下に示すリファクタリングを行うことできると考えられる．

分類 1 前節のケース 1 である．図 6 の例のように，チェーンドクローンセットを包含しているクラスに，すべてのクローンセットを集約することができる．

分類 2 前節のケース 2 である．図 7 (a) の例のように，“Pull Up Method” パターンを適用できる．

分類 3 前節のケース 3 である．図 7 (b) の例のように，“Extract SuperClass” パターンを適用できる．

分類 4 前節のケース 4 である．図 8 から分かるように，チェーンドクローンセット単位でのリファクタリングを行うことができないが，クローンセット単位でのリファクタリングを検討すべきである．

3.5 チェーンドクローンセットの分類を目的としたメトリクス

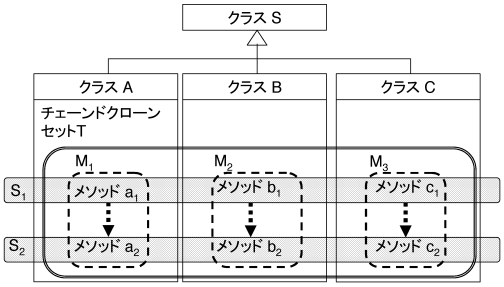
ここでは，チェーンドクローンセットの分類を行うためのメトリクスを 2 つ提案する．1 つは C1 を評価するメトリクス，もう 1 つは C2 を評価するメトリクスである．これらメトリクスは，メソッド間のクラス階層上における関係を表す．この関係は，2.3 節で述べた $DCH(S)$ メトリクス（クローンセット S に含まれる各コード片間のクラス階層内における最大の距離）を用いて表すことができる．

まず，$DCH(S)$ メトリクスを用いて，C1 を表す **DCHS(T)** メトリクスを定義する．チェーンドクローンセット T を n 個のクローンセット $S_1, S_2, \cdots, S_n$ に分割する（すなわち，$S_1 \cup S_2 \cup \cdots \cup S_n = T$，$S_i \cap S_j = \emptyset$，$1 \leq i \leq n$，$1 \leq j \leq n$，$i \neq j$）．さらに，クローンセット S_i には，複数のメソッドが含まれるとする．このとき，$DCHS(T)$ メトリクスの定義は以下のようになる．

$$DCHS(T) = max\{DCH(S_1), \cdots, DCH(S_n)\}$$

同様に $DCH(S)$ メトリクスを用いて，C2 を表す **DCHD(T)** メトリクスを定義する．チェーンドクローンセット T 中には，n 個のメソッドチェーン $M_1, M_2, \cdots, M_n$ が含まれるとする．さらに，メソッドチェーン M_i には，複数のメソッドが含まれるとする．このとき，$DCHD(T)$ メトリクスの定義は以下のようになる．

$$DCHD(T) = max\{DCH(M_1), \cdots, DCH(M_n)\}$$



クローンセット $S_1 = \{a_1, b_1, c_1\}$, $S_2 = \{a_2, b_2, c_2\}$
メソッドチェーン $M_1 = \{a_1, a_2\}$, $M_2 = \{b_1, b_2\}$, $M_3 = \{c_1, c_2\}$

$DCHS(T) = \max\{DCH(S_1), DCH(S_2)\} = 1$
$DCHD(T) = \max\{DCH(M_1), DCH(M_2), DCH(M_3)\} = 0$

図 9 提案するメトリクスの算出例
Fig. 9 Example of proposed metrics calculation.

図 9 は，提案する 2 つのメトリクスの算出例である．ここでは，分類 2 のチェーンドクローンセットを例として用いる．このチェーンドクローンセットには，クラス A，B，C にまたがって 2 つのクローンセット S_1，S_2 が存在する．各クローンセットについてそれぞれ $DCH(S_1)$，$DCH(S_2)$ を求めると，クラス A，B，C は共通の直接の親クラス S を持っているため両者とも 1 になる．$DCHS(T)$ の値は，これらの最大値の 1 である．また，このチェーンドクローンセットには，3 つのメソッドチェーン M_1，M_2，M_3 が含まれている．各メソッドチェーンについてはそれぞれ $DCH(M_1)$，$DCH(M_2)$，$DCH(M_3)$ を求めると，各メソッドチェーンはそれぞれ 1 つのクラスに包含されているためすべて 0 になる．よって，$DCHD(T)$ の値は，これらの最大値の 0 である．

3.6 実　　装

提案手法を Aries のコンポーネントの 1 つとして実装した．具体的には，Aries に対して，以下の 3 つの機能を追加した．

(F1) チェーンドクローンセットの検出機能
(F2) 提案したメトリクスの算出機能
(F3) チェーンドクローンセットおよびメトリクス値の表示機能

(F1) は，まず CCFinder および Aries を用いて各クローンセットが含むコード片を検出する．次に，それらコード片を対象に，メソッド呼び出し関係と変数

の共有関係を表すグラフを構築する．その後，構築したグラフに含まれる部分グラフから同形グラフを検出することにより，チェーンドクローンセットを検出する．(F2) は，Aries の $DCH(S)$ メトリクスを計算する機能を拡張した．(F3) を実現するために，チェーンドクローンセットを閲覧するビューを追加した．

4. ケーススタディ

4.1 概　　要

提案手法の有効性を確かめるため，ケーススタディを行った．具体的には，以下の 2 つを確認した．

- クローンセット単位の検出と比較して，検出できたチェーンドクローンセットの規模が大きいか．
- クローンセット単位でのリファクタリングと比較して，容易にリファクタリングできているか．

なお，この章におけるチェーンドクローンセットは，極大チェーンドクローンセット☆を指す．

適用対象は，次の 2 つのオープンソースソフトウェアである．

- ANTLR 2.7.4 [11]（4.7 万行，285 クラス）
- JBoss 3.2.6 [12]（64 万行，3,364 クラス）

ANTLR は，3 つのプログラミング言語（Java，C#，C++）に対応したコンパイラ・コンパイラである．JBoss は，J2EE アプリケーションサーバである．

4.2 チェーンドクローンセットの検出

前述の 2 つのソフトウェアに対し，提案手法に基づくチェーンドクローンセットの検出，および従来手法に基づくクローンセットの検出を行った．提案手法ではメソッド単位のコードクローンのみを扱うため，従来手法に基づく検出でもメソッド単位のクローンセットのみを対象とした．また，CCFinder が検出するコードクローンの最小トークン数は 30 に設定した．

検出結果の比較を行うために，2 つの評価基準として，メソッド数と，メソッド行数を用いる．ここで，メソッド数は，検出単位ごと（クローンセットごとやチェーンドクローンセットごと）に含まれるメソッド数を求め，総クローンセット数や各分類に属するすべてのチェーンドクローンセット数で除算した値とした．メソッド行数は，検出単位ごとに最長メソッドの行数を求め，総クローンセット数や各分類に属するすべてのチェーンドクローンセット数で除算した値とした．

設定した評価基準に基づいて，クローンセットの検出結果（**表 2**）と分類 1，2，3 に属したチェーンドクローンセットの検出結果（**表 3** (a)）を比較する．

表 2 クローンセットの検出結果
Table 2 Detection result of clone sets.

検出数		メソッド数		メソッド行数	
ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
152	377	2.3	2.4	10.8	6.63

まず，ANTLR では分類 2 に属したチェーンドクローンセットのメソッド数やメソッド行数がきわめて大きかった．分類 2 のメソッド数はクローンセットの 8.3（19/2.3）倍，メソッド行数は 5.0（54.0/10.8）倍であった．一方，分類 1，3 に属したチェーンドクローンセットのメソッド数は両者ともクローンセットの 1.7（4.0/2.3）倍，メソッド行数はそれぞれ 2.6（27.7/10.8）倍，3.2（35.0/10.8）倍であった．分類 2 に属するチェーンドクローンセットの多くは，**図 10** のような，Java，C#，C++に対応した出力を行う箇所から検出された．これら言語に対応した出力処理は類似しており，大量のコードクローンを含んでいた．

次に，JBoss の結果を見てみると，分類 1，2，3 に属したチェーンドクローンセットのメソッド数はそれぞれクローンセットの 1.8 倍〜2.8 倍，メソッド行数がそれぞれ 2.4 倍〜3.2 倍であった．これらの結果から，チェーンドクローンセットの規模がクローンセットに比べて大きいことが分かる．続いて，分類 1，2，3 に属したチェーンドクローンセットの検出結果（表 3 (a)）と，分類 4 に属したチェーンドクローンセットの検出結果（表 3 (b)）を比較する．3.4 節で述べたように，表 3 (a) で示した分類 1，2，3 は，チェーンドクローンセット単位でのリファクタリングはできるが，表 3 (b) で示した分類 4 はチェーンドクローンセット単位でのリファクタリングはできない．分類 1，2，3 と比較して分類 4 に属するチェーンドクローンセットは少ないことが分かる．特に，ANTLR からは検出されなかった．一方，JBoss では分類 4 のチェーンドクローンセットが検出された．これらのメソッド数，メソッド行数は，分類 1〜3 に比べて数倍の大きさになっている．

4.3 チェーンドクローンセットに対するリファクタリングの例

ANTLR から検出されたすべてのチェーンドクローンセットと JBoss から検出された 8 つのチェーンドクローンセットを対象として，提案手法に基づくリファクタリングを行った．ここでは，それらの中から 2 つの例を紹介する．

まず，図 10，**図 12** で示すチェーンドクローンセットに対し，提案手法により提示されたリファクタリングパターンを適用できることを確認した．図 10 は分

☆ 与えられたチェーンドクローンセットを真に包含するいかなるチェーンドクローンセットも存在しないとき，そのチェーンドクローンセットを極大チェーンドクローンセットと呼ぶ．

表 3 チェーンドクローンセットの検出結果
Table 3 Detection result of chained clone set.

(a) 分類 1，2，3

分類	検出数		メソッド数		メソッド行数	
	ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
1	3	16	4.0	5.8	27.7	16.2
2	6	17	19	4.5	54.0	17.1
3	1	13	4.0	6.8	35.0	21.5

(b) 分類 4

検出数		メソッド数		メソッド行数	
ANTLR	JBoss	ANTLR	JBoss	ANTLR	JBoss
0	4	—	19	—	54.5

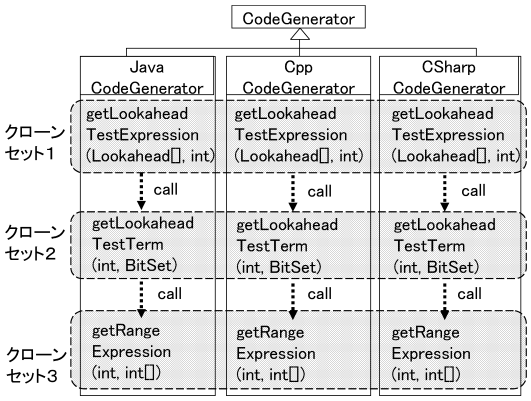


図 10 ANTLR から検出されたチェーンドクローンセットの例
Fig. 10 Example of a chained clone set in ANTLR.

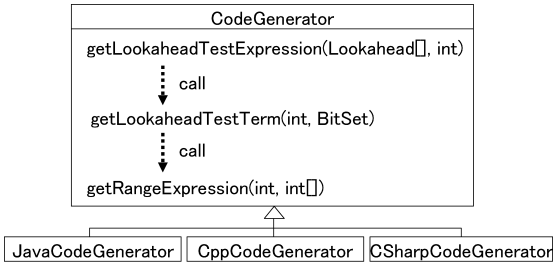


図 11 図 10 のチェーンドクローンセットをリファクタリングした例
Fig. 11 Example of refactoring of a chained clone set of Fig. 10.

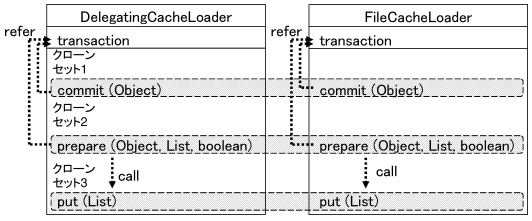


図 12 JBoss から検出されたチェーンドクローンセットの例
Fig. 12 Example of a chained clone set in JBoss.

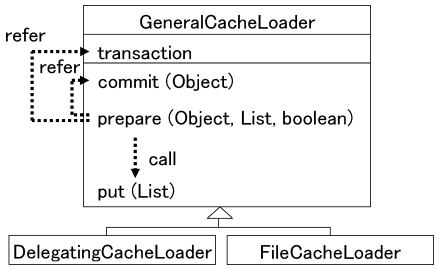


図 13 図 12 のチェーンドクローンセットをリファクタリングした例
Fig. 13 Example of refactoring of a chained clone set of Fig. 12.

類 2 であるから，“Pull Up Method” パターンを適用した．その結果，**図 11** のようになった．ANTLR パッケージ中の examples ディレクトリ以下にあるすべてのテストケース（計 86 ファイル，文法ファイル）を用いて回帰テストを行い，外部的振舞い（出力結果）が変化していないことを確認した．また，図 12 は分類 3 であるから，“Extract SuperClass” パターンを適用した．その結果，**図 13** のようになった．JBoss パッケージの testsuite ディレクトリ以下にある全テストケース（計 65 ファイル，JUnit フレームワーク[13] を使用）を用いて回帰テストを行い，外部的振舞いが変化していないことを確認した．

さらに，従来手法に基づいて，チェーンドクローンセットを構成するクローンセットに対し集約を試みると，工夫が必要となる場合があることを確認した．具体的には，図 10，図 12 からそれぞれクローンセットを 1 つ選び，従来手法により提示されたリファクタリングパターンの適用をした．まず，図 10 のクローンセット 1 に対し集約を試みると，提示されたリファクタリングパターンの適用に加えて，クローンセット 2 のメソッドに対応する抽象メソッドを CodeGenerator クラスに追加する必要があった．なお，クローンセット 2 に対して集約を試みた場合も同様であることが確

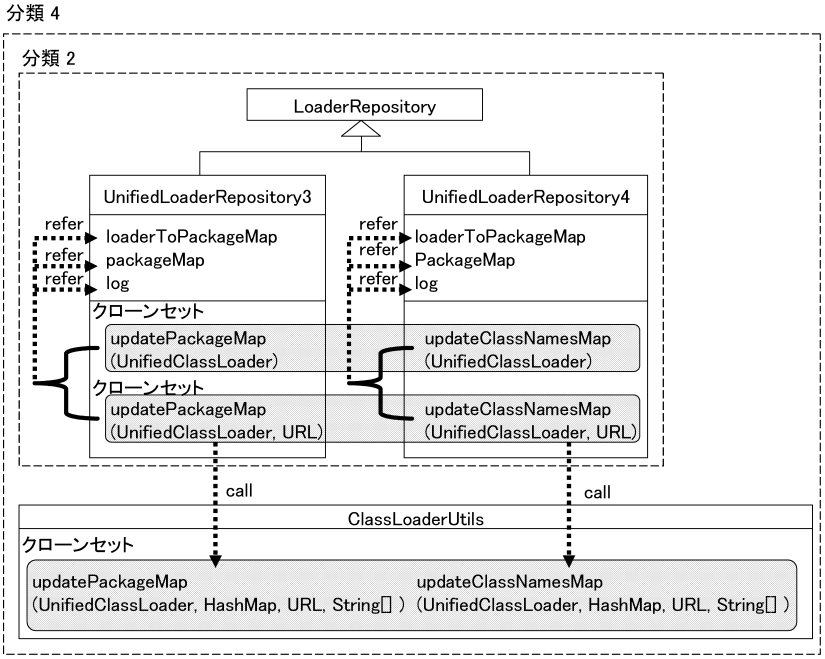


図 14 分類 4 のチェーンドクローンセットの例
Fig. 14 Example of Category 4.

認できた．また，図 12 のクローンセット 2 を新たに作成した親クラスに集約を試みると，提示されたリファクタリングパターンの適用に加えて，クローンセット 2 のメソッドに対応する抽象メソッドを親クラスに追加する必要があった．

4.4 考　　察

ここでは，今回のケーススタディに基づいて，提案手法の妥当性・制限等について考察を行う．

(1) 対象ソフトウェア　ケーススタディでは，Java 言語で開発された 2 つのオープンソースソフトウェアを対象として有効性の確認を行った．オブジェクト指向型言語であれば，Java 言語以外で開発されたソフトウェアであっても適用可能であると考えられる．なぜなら，Java 言語以外のオブジェクト指向型言語で開発されたソフトウェアであっても，リファクタリングや依存関係解析を適用可能だからである．今後，Java 言語以外で開発されたソフトウェアや商用ソフトウェア等，様々な種類のソフトウェアを対象に有効性の評価を行う必要がある．

(2) 被支援者の知識，経験　今回のケーススタディにおけるリファクタリング作業は，すべて著者が行った．そのため，手法の詳細を知らない人やリファクタリング経験が少ない人に対しても，有効な支援が行えるかどうか評価する必要がある．たとえば，一般の開発者に本稿のツールを使用してもらい，リファクタリングにかかった時間を計測することで効率を評価するということが考えられる．

(3) 対象とするコード片や依存関係の種類　提案手法は，メソッドより小さい単位のコード片からなるクローンセットは対象としていない．また，メソッドの呼び出し関係および変数の共用による依存関係のみを扱っているため，その他のデータ依存関係や制御依存関係を対象としていない．今後，対象とするコード片や依存関係を増やすことで検出可能なチェーンドクローンセットの規模を大きくし，有効性の評価を行う必要がある．

(4) 分類 4 のチェーンドクローンセットへの支援　JBoss から分類 2 のチェーンドクローンセットを内包する分類 4 のチェーンドクローンセットが検出された（**図 14**）．提案手法は，これに対しクローンセット単位でのリファクタリングを提示するが，内包された分類 2 のチェーンドクローンセットに “Pull Up Method” リファクタリングパターンを適用可能である．このような場合は，内包されたチェーンドクローンセットに対してリファクタリングパターンの提示を行うべきであると考えられる．

5. 関連研究

CCFinder や Balazinska ら[2] の手法を用いることにより，クローンセットの検出を行うことはできる．本稿の手法では，チェーンドクローンセット単位でのリファクタリングを提示することにより，大規模なリファクタリングを実現することができた．また，CCFinder が検出するクローンセットには容易にリファクタリングできないものが含まれており，Balazinska らの手法も同様と考えられる．本稿では，それら容易にリファクタリングできないクローンセットを組み合わせることで容易にリファクタリングできる場合があることを示し，それらクローンセットに対するリファクタリング手法を提案した．

Komondoor ら[4] の手法は，プログラムスライシング技術を用いて，ソースコードからプログラム依存グラフを構築し，そのグラフ上で同形である箇所をコードクローンとして検出している．よって，本稿で用いている CCFinder が検出できないコードクローン（一部の文の出現順序が異なっている reodered clone 等）を検出することができる．しかし，プログラム依存グラフの構築にかかる計算コストは非常に大きいため，大規模ソフトウェアへの適用は現実的でない．本稿の手法は，CCFinder が検出したコードクローンに対して，メソッド呼び出し関係と変数の利用関係のみを解析しているため，ケーススタディで示した規模のソフトウェアに適用可能である．実際に，ANTLR を対象としたチェーンドクローンセットの検出を約 1 分 4 秒で行うことができた☆．

6. まとめ

本稿では，クローンセットに含まれるメソッド間の依存関係に着目し，チェーンドクローンセットを定義した．そして，チェーンドクローンセットに対しリファクタリングパターンを提示するためのメトリクスを提案した．最後に，提案手法をリファクタリング支援ツールとして実装し，2 つのオープンソースソフトウェアに適用することで，有効性の評価を行った．有効性の評価として，チェーンドクローンセットの規模がクローンセットと比べて大きいこと，およびチェーンドクローンセットのリファクタリングがクローンセット単位のリファクタリングと比べて容易であることを確認した．

今後の課題としては，様々なソフトウェアを対象とした有効性の評価，チェーンドクローンセットの中に異なる分類のチェーンドクローンセットが包含されている場合への対処，対象とする依存関係やコード片の拡大があげられる．

謝辞 本研究は一部，文部科学省リーディングプロジェクト「e-Society 基盤ソフトウェアの総合開発」，日本学術振興会の科研費（課題番号：17200001）の支援を受けている．

☆ 実行環境：CPU Xeon 2.80 GHz，メモリ 2.5 GB，OS FreeBSD 6.1-RELEASE

参考文献

1) Baker, B.S.: A Program for Identifying Duplicated Code, *Proc. Computing Science and Statistics*, Vol.6, pp.49–57 (1992).
2) Balazinska, M., Merlo, E., Dagenais, M., Lague, B. and Kontogiannis, K.: Advanced clone-analysis to support object-oriented system refactoring, *Proc. Working Conference on Reverse Engineering* (*WCRE2000*), pp.98–107 (2000).
3) Baxter, I., Yahin, A., Moura, L., Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. International Conference on Software Maintenance* (*ICSM98*), pp.368–377 (1998).
4) Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, *Proc. International Static Analysis Symposium* (*SAS2001*), pp.40–56 (2001).
5) Krinke, J.: Identifying Similar Code with Program Dependence Graphs, *Proc. Working Conference on Reverse Engineering* (*WCRE2001*), pp.301–309 (2001).
6) Fowler, M.: *Refactoring: improving the design of existing code*, Addison Wesley (1999).
7) Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multi-linguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
8) Higo, Y., Kamiya, T., Kusumoto, S. and Inoue, K.: ARIES: Refactoring Support Environment Based on Code Clone Analysis, *Proc. International Conference on Software Engineering and Applications* (*SEA2004*), pp.222–229 (2004).
9) 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.47–54 (2001).
10) 門田暁人，佐藤慎一，神谷年洋，松本健一：コードクローンに基づくレガシーソフトウェアの品質の分析，情報処理学会論文誌，Vol.44, No.8,

pp.2178–2188 (2003).
11) ANTLR: http://www.antlr.org
12) JBoss: http://www.jboss.org
13) JUnit: http://www.junit.org

(平成 18 年 2 月 3 日受付)
(平成 18 年 12 月 7 日採録)

吉田　則裕（学生会員）

平成 16 年九州工業大学情報工学部知能情報工学科卒業．平成 18 年大阪大学大学院博士前期課程修了．現在，同大学院博士後期課程 1 年．コードクローン分析の研究に従事．人工知能学会会員．

肥後　芳樹（正会員）

平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．現在，日本学術振興会特別研究員．コードクローン分析・リファクタリング支援の研究に従事．

神谷　年洋

平成 8 年大阪大学基礎工学部情報工学科中退．平成 13 年同大学大学院博士課程修了．同年科学技術振興事業団研究者．平成 17 年産業技術総合研究所研究員．博士（工学）．オブジェクト指向関連技術，ソフトウェア保守（メトリクス，コードクローン），認知科学に関する研究に従事．電子情報通信学会，電気学会，IEEE 各会員．

楠本　真二（正会員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年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助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．電子情報通信学会，IEEE，JFPUG，PM 各会員．

井上　克郎（正会員）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．工学博士．平成 14 年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教授．ソフトウェア工学の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．