修士学位論文

題目

コードクローン間の依存関係を利用したリファクタリング支援手法の提案と実現

指導教員
井上克郎 教授

報告者
吉田 則裕

平成 18 年 2 月 13 日

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 ソフトウェア工学講座

平成 17 年度 修士学位論文

コードクローン間の依存関係を利用したリファクタリング支援手法の提案と実現

吉田 則裕

内容梗概

コードクローンとは，同一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一つとされている．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対して，修正の是非を検討する必要がある．ソフトウェア保守性を改善する方法の一つとして，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．コードクローンに対しリファクタリングを行うことにより，保守性を改善することができる．しかし，異なるクローンセット（コードクローンの同値類）に含まれる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．このような場合，これらクローンセットをまとめてリファクタリングすることが可能であるが，リファクタリング技術に詳しく，かつコード片間の依存関係を把握した技術者でなければ，そのようなリファクタリングを行うことは難しい．

本研究では，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に依存関係がある場合に着目し，そのような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が異なることを利用して，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．

また，提案手法をリファクタリング支援ツールとして実装し，複数のオープンソースソフトウェアを対象としたケーススタディを行った．ケーススタディにより，チェーンドクローンセットに対して提示するリファクタリングパターンが妥当であることを確認できた．

主な用語

リファクタリング
ソフトウェア保守
コードクローン
ソフトウェアメトリクス

目 次

1 まえがき

ソフトウェア保守を困難にする要因の一つとして，コードクローンが指摘されている [3, 8, 9, 19, 20]．コードクローンとは，直観的には同一，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する．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対して，修正の是非を検討する必要があり，ソフトウェア保守にかかるコストが増大する．

ソフトウェア保守性を改善する技術の一つとして，リファクタリング [14] 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．Fowler は文献 [14] の中で，リファクタリングを検討すべき箇所にあらわれる特徴を Bad Smell と呼び，その代表例としてコードクローン（Duplicated Code）を挙げている．また，コードクローンを取り除く手法として，“Pull Up Method” や “Extract Method” などのリファクタリングパターンを紹介している．

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．しかし，これらの作業に要するコストは非常に大きい．まず，コードクローンの検出に大きなコストが必要となる．ソフトウェア全体に散布しているコードクローンや，表現上は異なるコード片からなるコードクローンを検出することは難しいからである．次に，検出されたコードクローンの中から，リファクタリングに適したもののみを抽出するコストが必要となる．検出したコードクローンの中には，プログラミング言語における構造単位（for 文単位，メソッド単位，クラス単位）のコード片からなるものと，そのような構造単位ではないコード片からなるものがある．前者は，一つのモジュールに集約可能であるためリファクタリングに適しているが，後者は一つのモジュールに集約することが困難であるためリファクタリングに適していない．よって，前者のみを抽出する必要がある．最後に，リファクタリングパターンの選択に大きなコストが必要となる．適切なリファクタリングを選択するには，対象となるコードクローン，およびそのコードクローンを含むソースコードに対する理解が必要となるからである．

我々はこれらを踏まえ，コードクローン検出ツール CCFinder[18] およびリファクタリング支援環境 Aries[27] を開発してきている．CCFinder の特徴は，ソースコードに対して様々な正規化処理を行っているため，表現上の差異があるコードクローンであっても検出できること，および数百万行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，まず CCFinder が検出したコードクローンから，リファクタリングに適したもののみを抽出する．そして，それらコードクローンをクローンセット（コードクローンの同値類）単位に分類し，その特徴をメトリクスとして提示する．それらメトリクスの中には，リファク

タリングパターンの決定支援を目的としたものがあり，ユーザは提示されたメトリクス値を見ながら，リファクタリングパターンを決定することができる．

Aries を用いて，様々なソースコードを解析したところ，異なるクローンセット間に，メソッドの呼出や変数を共用することによる依存関係が存在する場合があることがわかった．例を挙げると，クローンセット $A = \{a_1, a_2\}$, $B = \{b_1, b_2\}$ が検出され，かつメソッド a_1 からメソッド b_1 に，メソッド b_1 からメソッド b_2 に呼出関係がある場合である．このような場合，Aries は依存関係を考慮せず，各クローンセットに対してそれぞれメトリクスを提示するなどして，リファクタリング支援を行う．しかし，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と，リファクタリングが困難になる場合がある．このような場合，これらクローンセットをまとめてリファクタリングすることが可能であるが，リファクタリング技術に詳しく，かつコード片間の依存関係を把握した技術者でなければ，そのようなリファクタリングを行うことは難しい．

そこで本稿では，コード片間の依存関係に着目することにより，より効果的にコードクローンをリファクタリングする手法を提案する．まず，先程の例の対 $(\{a_1, b_1\}), \{a_2, b_2\})$ のように，依存関係も含めてコードクローンとなっているコード片集合の対をチェーンドクローンペアと定義する．更に，互いにチェーンドクローンペアとなっているコード片集合を同値としたときの同値類をチェーンドクローンセットと呼ぶ．そして，チェーンドクローンセットに対して適切なリファクタリングパターンを提示する手法を提案する．

また，提案した手法の妥当性を確認するためのケーススタディを行った．ケーススタディでは，提案手法を実装したツールを作成し，複数のオープンソースソフトウェアに適用した．その結果，チェーンドクローンセットに対して提示するリファクタリングパターンが妥当であることを確認できた．

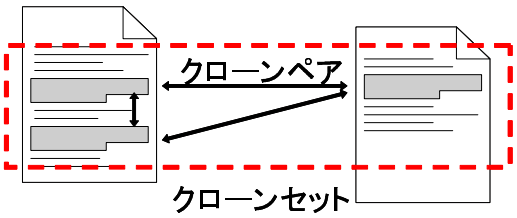


図 1: クローンペアとクローンセット

2　コードクローンが引き起こす問題とリファクタリング技術

2.1　コードクローン

あるトークン列中に存在する二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（図 1）．α，β それぞれを真に包含する如何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，互いにクローンであるトークン列を同値としたときの同値類をクローンセットと呼ぶ（図 1）．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 [25].

コードクローンがソフトウェアの中に作りこまれる，もしくは発生する原因として次のようなものがある [9][18].

既存コードのコピーとペーストによる再利用

近年のソフトウェア設計手法を利用すれば，構造化や再利用可能な設計が可能である．しかし，コードの再利用が容易になったために，現実にはコピーとペーストによる場当たり的な既存コードの再利用が多く行われるようになった．

定型処理

定義上簡単で頻繁に用いられる処理．例えば，給与税の計算や，キューの挿入処理，データ構造アクセス処理などである．

プログラミング言語に適切な機能の欠如

抽象データ型や，ローカル変数を用いられない場合には，同じようなアルゴリズムを持った処理を繰り返し書かなくてはならないことがある．

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある．

コード生成ツールの生成コード

コード生成ツールにおいて，類似した処理を目的としたコードの生成には，識別子名等の違いはあろうとも，あらかじめ決められたコードをベースにして自動的に生成されるため，類似したコードが生成される．

偶然

偶然に，開発者が同一のコード片を書いてしまう場合もあるが，大きなコードクローンになる可能性は低い．

2.2 コードクローンが引き起こす問題

コードクローンは，ソフトウェア保守を困難にする要因の一つと指摘されている．この節では，まずソフトウェア保守の困難さについて述べ，その後にコードクローンがソフトウェア保守作業を困難にする理由を説明する．

2.2.1 ソフトウェア保守とその課題

ソフトウェア保守とは，“納入後，ソフトウェア・プロダクトに対して加えられる，フォールト修正，性能または他の性質改善，変更された環境に対するプロダクトの適応のための改訂” であると定義されている [16]．また，目的毎に次の四つのカテゴリに分けられている．

修正のための保守 (Correction) 発見された問題を修正するために，納入後に実施される，ソフトウェア・プロダクトの対処的改変，

適応のための保守 (Adaptation) 変化した，または変化しつつある環境において，ソフトウェア・プロダクトを続けて使用可能なように維持するために，納入後に実施される，ソフトウェア・プロダクトの改変，

完全化のための保守 (Perfection) 性能または保守性を改善するため，納入後に実施される，ソフトウェア・プロダクトの改変，

予防のための保守 (Prevention) ソフトウェア・プロダクトのなかに潜む，潜在的なフォールトが，効果的なフォールトに転じる前に，それを検出し，修正するために，納入後に実施される，ソフトウェア・プロダクトの改変．

ソフトウェア保守は，ソフトウェアライフサイクルコストの大きな部分を占めており [13]，ライフサイクルを考えるとき，その費用と労力からみて保守は非常に大切な工程である．しかし，ソフトウェア保守における課題は多くあり，Dorfman および Thayer[11] は，保守に

関しては，投資効果が明らかにならないので，常に資源を獲得するための戦いが起きると述べている．資源を巡って争うということが常に存在し，次のソフトウェア・プロダクトに対するコードを作成しながら，将来の納入計画を決め，現在のソフトウェア・プロダクトに対して緊急的なパッチを施すということは難題である．また，ソフトウェアを開発したチームは，ソフトウェアが運用に入ると必ずしも保守を担当するとは限らない．自分の開発したものではない，大規模なソースコードに潜む欠陥を発見しなければならないということは，保守者にとっては非常に難題である．

多くの場合，独立したチームが，ソフトウェアが適切に運用されユーザの変化するニーズを満たすように進化させられていることを確実にするために雇用されているが，保守のアウトソーシング (社外調達) も，主要な産業になりつつある．大企業は，非公開としたいビジネス中核となるソフトウェアを除いては，ソフトウェア保守を含めて，運用をアウトソーシングする．このような背景のために，開発者は通常コードを説明しなければならない場には居合わせないことが多く，変更も文書化されていないことも多い．したがって，保守者は，ソフトウェアに関して制限された理解しか得ることができず，ソースコードを自分で読まねばならない．文献 [22] では保守作業のおおよそ 40%から 60%は，改変すべきソフトウェアの理解に費やされていると指摘されている．したがって，保守作業の効率を高めること，さらにプログラムの理解容易性を高めるということは，ソフトウェア工学における重要な課題の一つとなっている．

2.2.2 コードクローンがソフトウェア保守に与える影響

プログラム中にコードクローンが存在すると，そのプログラムの修正が難しくなる．例えば，三つのコード片がコードクローンとなっていた場合，その一つに対して修正を行うと，他の二つに対しても修正の検討を行う必要がある．

特に大規模ソフトウェアを対象のソフトウェア保守では，大量のソースコード中からコードクローンを探し出し，その一つ一つに対し修正の検討を行うことは，大きなコストがかかる作業である．更に，アウトソーシングなどにより，ソフトウェア開発者とソフトウェア保守者が異なる場合は，コードクローンを探し出す作業がより大きな負担となる．このようなコストは，投資効果の見積もりが難しいソフトウェア保守工程では，非常に大きな問題となる可能性が高い．

2.3 コードクローンが引き起こす問題への対策

前節で述べたコードクローン問題に対処するために，さまざまな自動でコードクローン検出する手法やツールが提案されている（これらの手法やツールについては，2.4 節で詳しく

述べる）．

これにより，ソフトウェアに含まれるコードクローンに対して，以下の二つのことを効率良く実現することができる．

(1) ソースコードの修正前に，その修正部分に対応するコードクローンの有無を確認し，もしあれば修正の検討を行う

(2) コードクローンを集約（除去）する

また，(2) を実現する技術の一つして，2.5 節で述べるリファクタリングがある．なぜなら，2.2.1 節で述べた理由により，多くの場合，保守者が対象となるソフトウェアを十分に把握していないため，コードクローンとなっている箇所を提示するだけでは，リファクタリング技術を活用することは難しい．そこで，我々の講座では，節で述べるコードクローンに基づくリファクタリング支援手法を提案している．この手法 S では，リファクタリングに適したコードクローンの検出や，それらに適切な集約方法の提示を行う．

2.4 コードクローン検出手法

この節では，我々の講座が開発したコードクローン検出ツール CCFinder[18] およびその他の手法の特徴について述べる．

2.4.1 コードクローン検出ツール CCFinder

CCFinder は，単一または複数のファイルのソースコード中から全ての極大クローンを検出し，それをクローンペアの位置情報として出力する．CCFinder の持つ主な特徴は次の通りである．

細粒度のコードクローンを検出

字句解析を行うことにより，トークン単位でのコードクローンを検出する．

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である [25].

様々にプログラミング言語に対応可能

言語依存部分を取り替えることで，様々なプログラミング言語に対応できる．現在は，C/C++，Java，COBOL/COBOLS，Fortran，Emacs Lisp に対応している．またプレーンテキストに対しても，分かち書きされた文章として解析可能となっており，未対応の言語に対しても完全一致判定によるコードクローンはとることができる．

実用的に意味を持たないコードクローンを取り除く

- コードクローンは小さくなればなるほど偶然の一致の可能性が高くなるが，最小一致トークン数を指定することができるため，そのようなコードクローンを検出しないようにできる．
- モジュールの区切りを認識する．

ある程度の違いは吸収可能

- ソースコード中に含まれるユーザ定義名，定数をパラメータ化することで，その違いを吸収できる．
- クラススコープや名前空間による複雑な名前の正規化を行うことで，その違いを吸収できる．
- その他，テーブル初期化コード，可視性キーワード (protected, public，private 等)，コンパウンド・ブロックの中括弧表記等の違いも吸収することができる．

CCFinder のコードクローン検出手順（ソースコードを読み込んで，クローンペア情報を出力する）は大きく四つの過程から成り立っている．

ステップ 1（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．

ステップ 2（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，及び，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．例えば，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．

ステップ 3（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全て検出する．

ステップ 4（出力整形処理）: 検出されたクローンペアについて，元のソースコード上での位置情報を出力する．

実際に，CCFinder によってどのようなコードクローンが検出されるのか例を示す．

図 2 に説明のための Java のソースコードを示す．このソースコードには，互いに似通った二つのメソッドが含まれ，左端には行番号が付されている．ここで，最小一致トークン数を 5 トークンに定め，図 2 のソースコードに対しコードクローン検出を行うと，図 2 中の

```
    1. static void foo() throws RESyntaxException
    2. {
    3.    String a[] = new String [] {"123,400", "abc"};
A1  4.    org.apache.regexp.RE pat =
A1  5.              new org.apache.regexp.RE("[0-9,]+");
A1  6.    int sum = 0;
    7.    for (int i = 0; i < a.length; ++i)
B1  8.    {
B1  9.       if (pat.match(a[i])){
B1 10.          sum += Sample.parseNumber(pat.getParen(0));}
   11.    }
C1 12.    System.out.println("sum = " + sum);
   13. }
   14. static void goo(String [] a) throws RESyntaxException
   15. {
A2 16.    RE  exp = new RE("[0-9,]+");
A2 17.    int sum = 0;
   18.    int i   = 0;
   19.    while (i < a.length)
B2 20.    {
B2 21.       if (exp.match(a[i]))
B2 22.          sum += parseNumber(exp.getParen(0));
   23.       i++;
   24.    }
C2 25.    System.out.println("sum = " + sum);
   26. }
       :
       :
```

図 2: コードクローン検出例

A1 (4 行目-6 行目) とA2 (16 行目-17 行目), B1 (8 行目- 10 行目) とB2 (20 行目-22 行目), そしてC1 (12 行目) とC2 (25 行目) がそれぞれクローンペアとして検出される．それぞれのクローンペアの長さは順に 7, 18, 6 トークンとなっている．見ての通り，A1 と A2 の間，B1 と B2 の間には次のような幾らかの違いが含まれているがコードクローンとして検出可能となっている．

2.4.2 その他のコードクローン検出手法

CCFinder 以外にも，これまでにさまざまなコードクローン検出手法やツールが提案されている [1][3][4][5][8][6][7][9][12][18][19][20][23][21]．それぞれの手法やツールの特徴は次のようになっている．

Covet

文献 [23] で定義された種々の特徴メトリクスの幾つかのメトリクス値を比較することによって，コードクローン検出を行う．現在，試作段階にあり，検出対象言語は，Java である．

CloneDR[9]

抽象構文木 (AST) の節点を比較することによって，コードクローン (類似部分木) の検出を行う．また，部分的に異なっているコードクローンも検出することが可能であり，検出したコードクローンを自動的に等価なサブルーチンやマクロに置き換えることも可能である．検出対象言語は，C/C++，COBOL，Java である．

Dup[3][4][5]

ユーザ定義名のパラメータ化を行った後，行単位の比較によりコードクローンを検出する．マッチングアルゴリズムには，サフィックス木探索 [15] を用いているため線形時間で解析可能である．

Duploc[12]

前処理として，空白やコメント等を取り除いた後，行単位 (のハッシュ値) での表検索を用いた比較によってコードクローンを検出する．また，コードクローンの散布図等の GUI を備えたツールであり，ソースコード参照支援を行う．検出対象言語は，C，COBOL，Python，Smalltalk である．

JPlag[21]

ソースコードを字句解析し，トークン単位での比較を行う．プログラム盗用の検出を目的として開発され，プログラム間の類似率を検出する．検出対象言語は，C/C++，Java である．

Komondoor らの手法 [19]

関数等にまとめるのに適したコードクローンを抽出を目的として，プログラム依存グラフ (PDG) 上での各節点の比較を行うことでコードクローン (同型 (isomorphic) 部分グラフ) を検出する．文字列比較や抽象構文木等を用いた検出方法ではなかった非連続コードクローンや，対応行の順番が異なるクローン，互いに絡みあったクローン等を検出可能である．[19] で作成されたツールの検出対象言語は，C である．

Krinke の手法 [20]

AST や Traditional PDG に似た Fine-grained PDG というグラフ上での類似 (similar) 部分グラフ (同型部分グラフではない) を検出することで，コードクローンが存在すると思しき場所を検出する．試作ツールの検出対象言語は，C である．

SMC[8][6][7]

まず特徴メトリクスによってコードクローンと思しきメソッドに絞り込む．次に絞り込まれたメソッドのペアに対し，表検索を用いることでメソッド単位のコードクロー

ンを検出する．特徴メトリクスによって絞り込まれているため，実用上ほぼ線形時間で解析可能である．また検出されたペアのメソッドは，特徴により 18 種類に分類される．さらにそれぞれの分類については共通メソッドへの書き換え指針が示されている．

MOSS[1]

検出アルゴリズムは公開されていない．JPlag 同様，プログラム盗用の検出を目的として開発された．検出対象言語は，Ada，C/C++，Java，Lisp，ML，Pascal，Scheme である．

いずれの手法，ツールにおいても提案者によってコードクローンの定義が微妙に異なっており，検出されるコードクローンが異なっている．つまり，コードクローンの定義とは検出アルゴリズムそのものによって定義される．Burd ら [10] も，CloneDR，Covet，JPlag，Moss，そして我々の開発した CCFinder を含めた五つのツールを用いて，それぞれ検出されるコードクローンの比較が行っているが，全ての面において他のツールよりも優れているツールはなく，使う場面に応じて，適切なツールを選ぶことが必要となってくると述べている．

2.5 リファクタリング技術

リファクタリングとは “外部からみ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” であると定義されている [14].

Fowler は文献 [14] の中で，リファクタリングを検討すべき箇所にあらわれる特徴を Bad Smell と呼び，その代表例としてコードクローン（Duplicated Code）を挙げている．コードクローンを取り除く手法として，次のような対処方法がある (以下，オブジェクト指向プログラミング言語，特に Java を例にとって述べる).

“Extract Method”

ひとまとめにできるコード片がある場合に，新たなメソッドとして定義し，抽出されたコードを抽出先のメソッドへの呼び出し文に置き換える．特に重複したコードに限ったリファクタリングではないが，重複したコードで最も単純な例は，同一クラス内の複数メソッドに同じ式があるものである (図 3 参照).

“Pull Up Method”

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した場合，それらを親クラスに引き上げる．最も単純なケースは，複数のメソッド本体が全く同じである場合である (図 4 参照)．重複したコードが兄弟クラスに存在した場合には，“Extract Method” を行ってから，“Pull Up Method” を行えばよい.

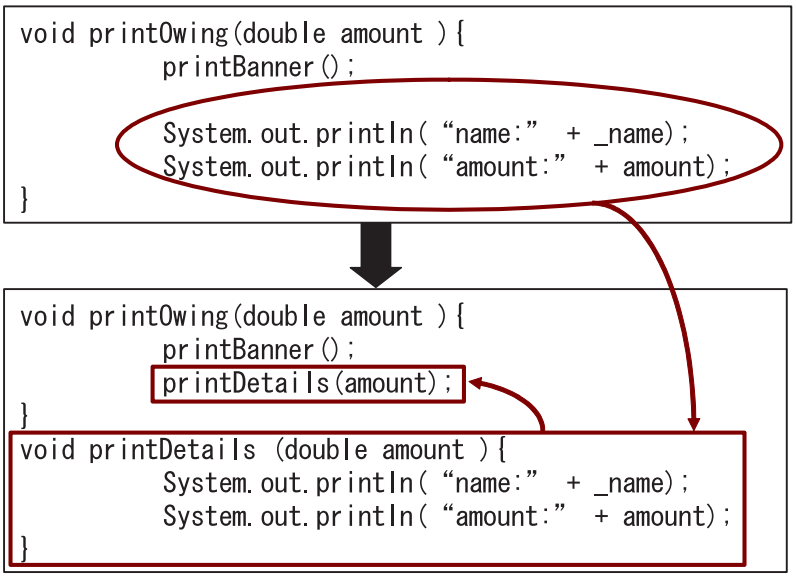


図 3: コードクローンを対象とした “Extract Method”

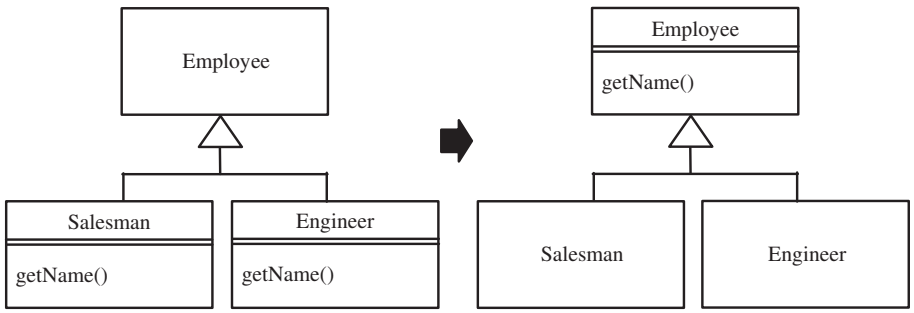


図 4: コードクローンを対象とした “Pull Up Method”

“Extract Class”

二つのクラスでなされるべき作業を一つのクラスで行っている際に，新たにクラスを作って，適当なフィールドとメソッドを元のクラスからそこに移動する．これも特に重複したコードに限ったリファクタリングではないが，全く関係のない複数のクラス間で，重複したコードが見られるときには，メソッド引き上げの代わりに別のクラスとして定義する．

“Extract SuperClass”

似通った特性を持つ複数のクラスがある場合に，新たに親クラスを作成して，共通の特性を移動する．“Extract Class”との違いは，継承するか委譲するかの違いである (図

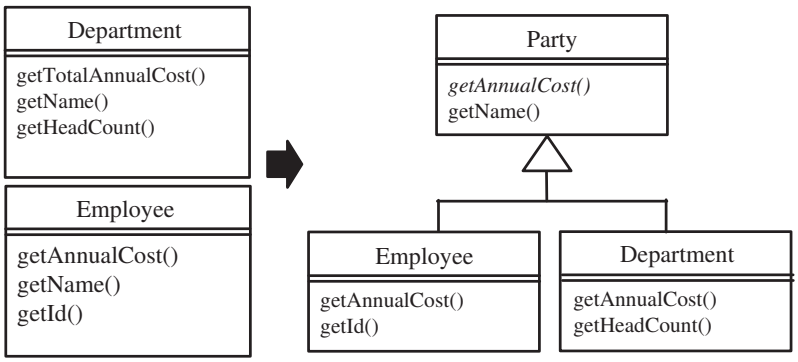


図 5: コードクローンを対象とした “Extract SuperClass”

5 参照).

“Form Template Method”

類似の処理を同じ順序で実行しているが，各処理が異なる場合，各処理を同じシグニチャを持つメソッドとして，親クラスに引き上げる．例えば，よく似た処理を同じ順番で実行しているものの，処理内容が違うには，順序の制御を親クラスに移動し，異なる処理については，元のクラスで行わせる (図 6 参照).

“Consolidate Duplicate Conditional Fragments”

条件式の全ての分岐に同じコード片がある場合，式の外側に移動する (図 7 参照)．条件記述以外にも，例外記述にも適用可能である．例えば，try ブロック内の例外の原因となる文の後，および全ての catch ブロックの中に重複コードがあるときは，`finally` ブロックに移動する．

“Replace Conditional with Polymorphism”

switch 文などは重複したコードを生成しやすくしている．同じような switch 文がある場合は，新たな分岐を追加した際に全ての switch 文を探して似たような変更をしなければならない場合も多く，新たな分岐での処理も他の分岐と比較し類似した処理が並ぶことが多いためである．その中でも特にオブジェクトの種類で分岐していた際には，ポリモーフィズムを利用した “Extract Method”等で対処することができる．

保守プロセスにおいてリファクタリングは，特に機能追加や，バグ修正，コードレビューの際に行うのがよいとされている．いずれもコードの理解が必要な作業であり，リファクタリングを行うことで，コードの理解が深まり，バグが混入しにくくなり，バグを発見しやすくなる．しかし，リファクタリングとは，あくまでもソフトウェアを理解しやすく，変更を容易にするために行うことであり，機能追加とは区別されなければならない．

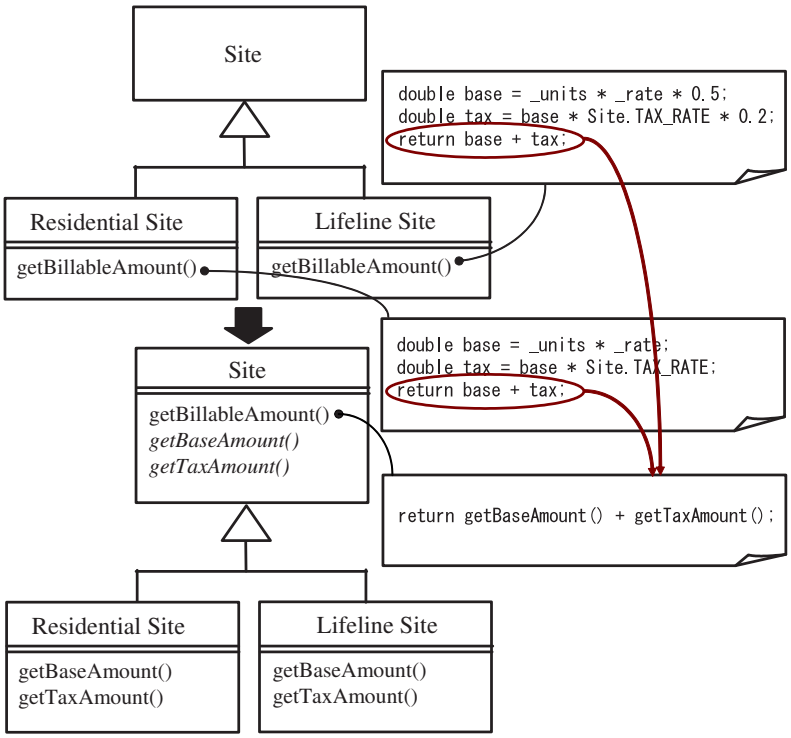


図 6: コードクローンを対象とした “Form Template Method”

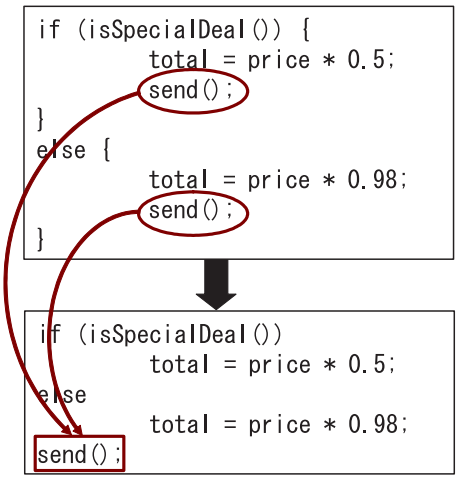


図 7: コードクローンを対象とした “Consolidate Duplicate Conditional Fragments”

2.6 コードクローンに基づくリファクタリング支援手法

2.6.1 リファクタリングに適したコードクローンの検出

これまでに，リファクタリング（集約）に適したコードクローンの検出手法がいくつか提案されている．例えばプログラム依存グラフを用いる手法 [19][20] が提案されている．この

手法は，ソースコードからプログラム依存グラフを構築し，そのグラフ上でのコードクローン検出を行なうことにより，高精度の検出を可能としている．しかし，プログラム依存グラフ構築の時間コストは非常に大きく，実際にソフトウェアの開発現場で適用するのは難しい．また，Mayland ら [23] は特徴メトリクスを用いたコードクローン検出を行なっている．彼らはコードクローン検出対象を，関数・メソッド単位に絞り込み，各関数・メソッドに対して 21 種類のメトリクス値を計測し，それらを定量的に特徴づける．そして，そのメトリクス値の比較により八段階の類似度を用いてコードクローンを検出している．この手法では，コードクローン検出対象を，関数・メソッドに特定しているため，関数・メソッド内部に存在するコードクローンの検出，集約を行なうことは不可能である．

これらの問題を解決する手法として，我々の講座ではプログラミング言語における構造的なまとまりをもったコードクローンを検出する手法を提案している [27]．

プログラミング言語における構造単位（e.g. for 文単位，メソッド単位，クラス単位）でコードクローンとなっているものは，一つのモジュールに集約可能であるため，リファクタリングに適している．

図 8 はその例を示している．図 8 では，A と B の二つのコード片が示されている．A と B それぞれの灰色の部分は，その部分が A と B の間の最大長のコードクローンであることを示している．コード片 A ではいくつかのデータがリスト構造の先頭から順に連続して格納されている．一方コード片 B では，リスト構造の後方から順に連続してデータが格納されている．これら二つのコード片には，リスト構造を扱う共通のロジック（for 文）が含まれているが，コード片の最初と最後には，偶然クローンとなった部分（代入文）も含まれてしまっている．集約を目的とした場合，灰色の部分全体よりも for 文のみをコードクローンとして検出する方が望ましい．

2.6.2 集約支援を目的としたメトリクスの提示

我々の講座では，検出されたリファクタリングに適したコードクローンをクローンセット単位に分類し，その特徴をメトリクスとして提示する手法を提案している [27]．それらメトリクスの中には，リファクタリングパターンの決定支援を目的としたものがあり，ユーザは提示されたメトリクス値を見ながら，リファクタリングパターンを決定することができる．コードクローンに対して適用できるリファクタリングパターンとして，“Extract Method” や “Pull Up Method” が考えられる．

“Extract Method” とはコード片を新たなメソッドとして再定義することであるから，抽出部分は周囲との結合度が低い方がよい．つまり，抽出部分の外側で定義された変数を抽出部分でできるだけ用いていないことが望ましい．もしそのような変数を用いていた場合は，抽出したメソッドの引数として与える，あるいはメソッドの返り値として返す必要がある．

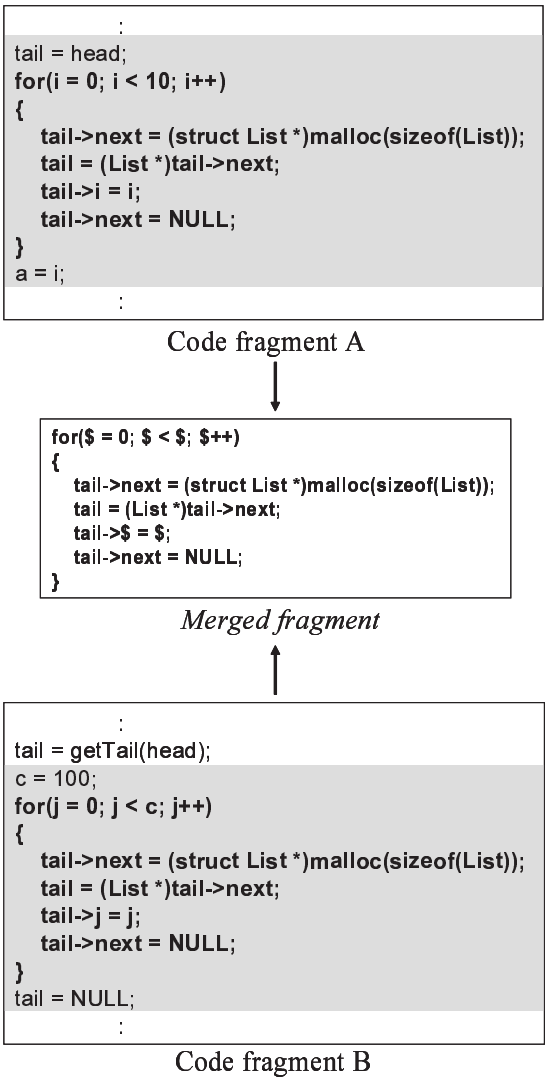
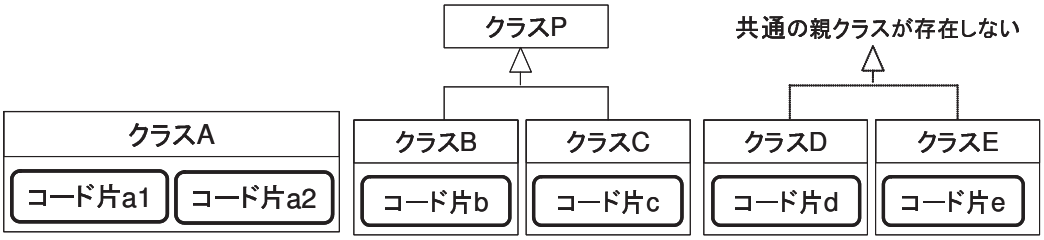


図 8: コードクローン集約の例

抽出部分とその周囲の結合度を計測するために $NRV(S)$ と $NSV(S)$ の二つのメトリクスを定義した．ここでは，クローンセット S はコード片 f_1，f_2，$\cdots$，f_n を含んでおり，コード片 f_i では外部定義の変数 rv_{i_1}，rv_{i_2}，$\cdots$，$rv_{i_{s_i}}$ を参照しており，sv_{i_1}，sv_{i_2}，$\cdots$，$sv_{i_{t_i}}$ に対して代入を行なっているとする．この時 $NRV(S)$ と $NSV(S)$ はそれぞれ次の式で表される．

$$NRV(S) = \frac{1}{n}\sum_{i=1}^{n} s_i, \qquad NSV(S) = \frac{1}{n}\sum_{i=1}^{n} t_i,$$

直観的には，$NRV(S)$ はクローンセット S に含まれる各コード片内で参照されている外部定義変数の平均数を示し，同様に $NSV(S)$ は代入が行なわれている変数の平均数を示す．



(a) DCH = 0　(b) DCH = 1　(c) DCH = ∞

図 9: DCH メトリクスの算出例

一方 “Pull Up Method” は，メソッドを親クラスに引き上げることであるから，メソッドを含むクラスは共通の親クラスを継承している必要がある．そのため，クローンのクラス階層内における分散度を計測する．これについては，メトリクス $DCH(S)$ を定義する．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，

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

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．例えば，S 中の全てのコード片が一つのクラス内に存在する場合は $DCH(S)$ メトリクスの値は 0（図 9(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 9(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 $DCH(S)$ メトリクスの値は ∞ とする（図 9(c)）．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．これにより，分析対象のソフトウェア内に存在するメソッドを修正不可能なクラスに引き上げようとする場合は，$DCH(S)$ メトリクスの値は ∞ となり，そのようなリファクタリングが不可能であることがわかる．

2.6.3　コードクローンに基づくリファクタリング支援環境 Aries

我々の講座では，2.6.1 節，2.6.2 節で述べたリファクタリングに適したコードクローンの検出手法と集約支援手法を実現するために，リファクタリング支援環境 Aries[27] を開発した．Aries は，CCFinder が検出したコードクローンから，リファクタリングに適したもののみを抽出する．そして，それらコードクローンをクローンセット単位に分類し，その特徴

をメトリクスとして提示する．それらメトリクスの中には，リファクタリングパターンの決定支援を目的としたものがあり，ユーザは提示されたメトリクス値を見ながら，リファクタリングパターンを決定することができる．

現在，Aries は Java 言語を対象として実装されている．Java 言語を対象としているため，リファクタリングに適したコードクローンとして，抽出す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

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

Aries の各インターフェースを簡単に紹介する．図 10(a)，10(b) は Aries のスナップショットである．ユーザは図 10(a) の *Main Window* においてリファクタリング対象となるクローンセットの絞り込みを行なうことができる．また個々のクローンセットについてより詳細な情報を *Clone Set Viewer* を用いて得ることができる．

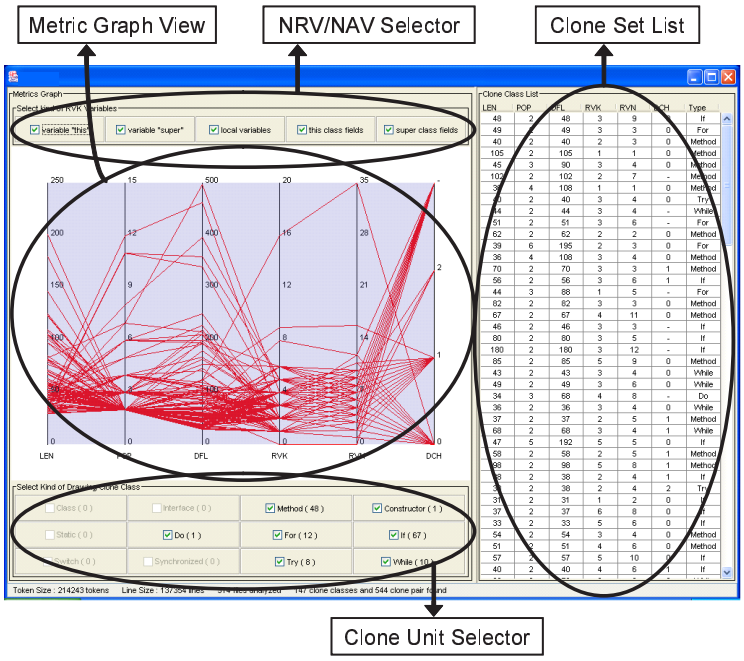
*Metric Graph View*は，各クローンセットに，2.6.2 節で提案したメトリクスに加え，$LEN(S)$, $POP(S)$, $DFL(S)$ [26] の三つのメトリクスを提供する．各メトリクスを簡単な説明を以下に示す．

LEN(S) S に含まれるコード片のトークン数の平均値を表す．

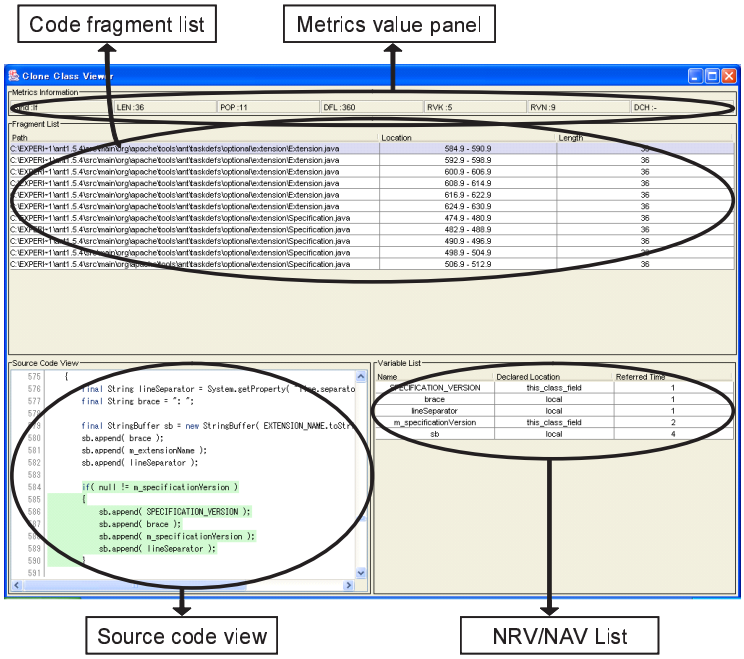
POP(S) S に含まれるコード片の数を表す．

DFL(S) S を再構築した場合に減少するトークン数の予測値を表す．ここでは，再構築とは，S 内のコード片集合から抜き出した共通のロジックを実装するサブルーチンを作り，各コード片をそのサブルーチンの呼び出しに置き換えることである．$DFL(S)$ は，再構築前のコードの大きさ（すなわち，S 内のコード片の大きさの和）から，再構築後のコードの大きさ（すなわち，サブルーチン呼び出しの大きさの和+共通ロジックを実装するサブルーチンの大きさ）を引いたものとして定義される．

次に，図 11 を例にとって *Metric Gragh View* を説明する．*Metric Graph View* では各メトリクスにつき一つの並行座標軸が用意され，各メトリクス値の上限，下限が設定される．図中の網がかかった部分が上限と下限の間を示している．また，各クローンセットにつき，一本の折れ線が描画される．この例では二つのクローンセット S_1 と S_2 が描画されている．図 11(a) では，両クローンセットとも，全てのメトリクス値が上限と下限の間におさまっているので，選択状態となっている．一方，図 11(b) ではメトリクス $DCH(S_2)$ が上限より大

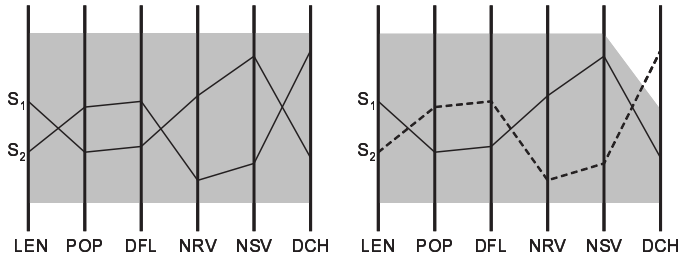


(a) Main Window



(b) Clone Set Viewer

図 10: Aries のスナップショット



(a) 絞り込み前　　(b) 絞り込み後

図 11: Metric Graph を用いた絞り込み

きくなってしまっており，クローンセット S_2 は非選択状態である．*Metric Graph View* を用いたクローンセットの選択，非選択状態は *Clone Set List* に反映される．

NRV/NAV Selector では，ユーザはどの種類の変数をメトリクス $NRV(S)$，$NSV(S)$ の要素としてカウントするのかを決定することができる．変数は以下の六種類に分類されている．

- 自クラスのフィールド変数,
- 親クラスのフィールド変数,
- インターフェースのフィールド変数,
- 自クラスを表す変数 “this”,
- 親クラスを表す変数 “super”,
- ローカル変数.

Clone Unit Selector では，ユーザはどの単位のクローンセットを *Metric Graph View* における選択の対象とするのかを決定することができる．選択できる単位は，前述した 12 種類である．

Clone Set List は *Metric Graph View* において選択状態にあるクローンセットの一覧を表す．このリストに表示されているクローンセットは各メトリクス値に基づいてソートが可能である．また，クローンセットを選択することにより，そのクローンセットに関するより詳細な情報を示す *Clone Set Viewer*（図 10(b)）が表示される．

Metrics Value Panel はそのクローンセットの各メトリクス値を示す.

Code Fragment List はそのクローンセットに含まれるコード片の一覧を示す．各コード片について，そのコード片を含むファイルへのパス，ファイル内での位置（開始行，開始列，終了行，終了列），そのコード片のトークン数が表示される．

Source Code View は *Code Fragment List* において選択されたコード片周辺のソースコードを示す．クローン部分は強調して表示される．

NRV/NAV List は *Code Fragment List* において選択されたコード片内で用いられているが，その外部で定義されている変数の参照，代入回数を示す．

次に，リファクタリングにおける具体的な Aries の利用方法について述べる．ユーザが “Pull Up Method” を行なおうと考えた場合，例えば，以下のような条件が考えられる．

(PC1) 対象となる単位はメソッド本体,

(PC2) $DCH(S)$ の値が 1 以上.

“Pull Up Method” はメソッドに対して行なわれるので，条件 (PC1) が考えられる．また，クローンメソッドを含むクラスが共通の親クラスを継承している必要があることから条件 (PC2) が必要である．Aries を用いてこの絞り込みを行なうには，*Clone Unit Selector* によってメソッド本体に相当するコードクローンのみを選択し，その上で，*Metric Graph View* によって $DCH(S)$ が 1 以上のコードクローンのみを選択する．これにより，“Pull Up Method” を適用できると期待されるクローンセットを *Clone Set List* や *Source Code View* に表示することができる．

次に “Extract Method” を行なうことを考える．“Extract Method” を行なう際の条件としては例えば，以下のものが上げられる．

(EC1) 対象となる単位は文単位,

(EC2) $DCH(S)$ の値が 0,

(EC3) $NSV(S)$ の値が 1 以下,

“Extract Method” とはメソッド内のコード片に対して適用されるので，(EC1) が必要である．また，全てのクローンとなっているコード片が同一のクラス内に存在する場合は容易に集約が可能であると考えられるので，条件 (EC2) を考慮している．クローンの内部において，外部定義変数に対して代入を行なっている場合は，その変数を引数として与え，返り値として返し，メソッドの呼び出し元に反映する必要がある．このような変数が複数あった場合は新たなデータクラスを定義し，そのオブジェクトを介して値を受け渡す必要がある．もしこのような変数が一つの場合は単に return 文を用いて返すだけで良く，容易にリファクタリングを行なうことができるので，条件 (EC3) を考慮している．

Aries を用いてこの絞り込みを行なうには，*Clone Unit Selector* を用いて，文単位 (do, if, for, switch, synchronized, try, while) のコードクローンのみを選択し，その上で，*Metric*

Graph View によって，条件 (EC2)，(EC3) を満たす上限，下限を設定する．これにより，“Extract Method” を適用可能であると期待されるクローンセットを *Clone Set List* や *Source Code View* に表示することができる．

3 提案手法

3.1 本研究の動機

Aries を用いて，様々なソースコードを解析したところ，異なるクローンセット間に依存関係が存在する場合があることがわかった．図 12 は，そのような場合の例である．この図では，クラス A，B にまたがって二つのクローンセットが存在する．一方のクローンセットはメソッド a_1，b_1 を含み，もう一方のクローンセットはメソッド a_2，b_2 を含む．更に，点線矢印で表すように，メソッド a_1 は メソッド a_2 を，メソッド a_1 はメソッド a_2 を呼び出しており，依存関係が二つ存在する．このような場合に Aries は，依存関係を考慮せず各クローンセットに対し個別にリファクタリング支援を行う．例えば，図 12 の場合，二つのクローンセットに対し，個別に “Pull Up Method” パターンの適用を提示する．提示どおりに，二つのクローンセットに対しそれぞれ “Pull Up Method” パターンを適用すると，クローンセットに含まれる全てのメソッドを共通の親クラスであるクラス S に引き上げることにより集約できる．

しかし，実際にリファクタリングを行うユーザから，それらクローンセットは一度にリファクタリングした方が効果的である，という指摘があった．また，図 12 に含まれる二つのクローンセットは，いずれも “Pull Up Method” パターンによりクラス S に集約することができる．つまり，同一パターンにより同一箇所に集約できる．これらの理由により，図 12 に含まれる二つのクローンセットは，個別にリファクタリングパターンを提示するよりも，まとめて一つのリファクタリングパターンを提示した方が効果的であると考えられる．

本稿では，図 12 のようなコードクローンをまとめてリファクタリングする手法を提案する．まず，このようなコードクローンを “チェーンドクローン” として定義する．次に，チェーンドクローンの特徴を計測するメトリクスを定義する．更に，そのメトリクスを用いて，適切なリファクタリングパターンを提示する手法を提案する．

3.2 チェーンドクローン

n 個のクローンセット $C_1, C_2, \cdots, C_n$ が与えられ，各クローンセットに含まれるコード片集合を $F_{c_1}, F_{c_2}, \cdots, F_{c_n}$，ソースコード中で宣言されている変数の集合を VA としたとき，集合 $F = F_{c_1} \cup F_{c_2} \cup \cdots \cup F_{c_n} \cup VA$ を頂点集合，その頂点集合に含まれるコード片間およびコード片と変数間の関係を辺集合とするラベル付き有向グラフ G を作成する．

有向辺に付属しているラベルは，コード片間およびコード片と変数間の関係の種類を表している．ラベルは，以下の五種類である．

C コード片間の呼出関係

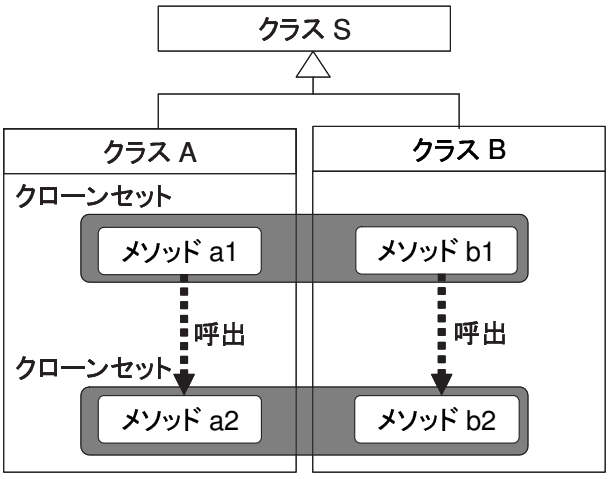


図 12: 異なるクローンセット間に依存関係が存在する例

R 始点のコード片が終点の変数を一回以上参照している

A 始点のコード片が終点の変数に一回以上代入している

更に，このラベル付き有向グラフ G に含まれる二つの部分グラフを α, β（ただし，$V(\alpha) \neq V(\beta), |V(\alpha) \cap F| \geq 2, |V(\beta) \cap F| \geq 2$）とし，この部分グラフに対応するコード片集合を $F_\alpha = V(\alpha) \cap F, F_\beta = V(\beta) \cap F$ とする．

この二つのコード片集合を F_α, F_β が互いにチェーンドクローンになるのは，二つの部分グラフを α, β が同型になり，対応するコード片（F に含まれる頂点）が互いにコードクローンになり，対応する関係（辺）の種類（ラベル）が等しいときである．

また，対 (F_α, F_β) をチェーンドクローンペアと呼び，互いにチェーンドクローンであるコード辺集合を同値であるとし，その同値類をチェーンドクローンセットと呼ぶ．更に，チェーンドクローンペア（F_α, F_β）の各要素（各コード片集合）をそれぞれチェーンと呼ぶ．

なお．与えられたチェーンドクローンセットを真に包含する如何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．

図 13 は，チェーンドクローンセットの例である．図 13 のクローンセット C_1, C_2, C_3, C_4 には，それぞれコード片集合 $F_{c_1} = \{f_{11}, f_{12}, f_{13}\}, F_{c_2} = \{f_{21}, f_{22}, f_{23}\}, F_{c_3} = \{f_{31}, f_{32}, f_{33}\}, F_{c_4} = \{f_{41}, f_{42}, f_{43}\}$ が含まれている．

コード片集合 $F = Fc1 \cup Fc2 \cup Fc3 \cup Fc4$ からなる頂点集合と，コード片の集合 F に含まれるコード片間の依存関係からなる辺集合から，有向グラフ G を作成する（図 15）．

この有向グラフ G の部分グラフ g_1, g_2 は，同型であり，対応する頂点がクローンであり，

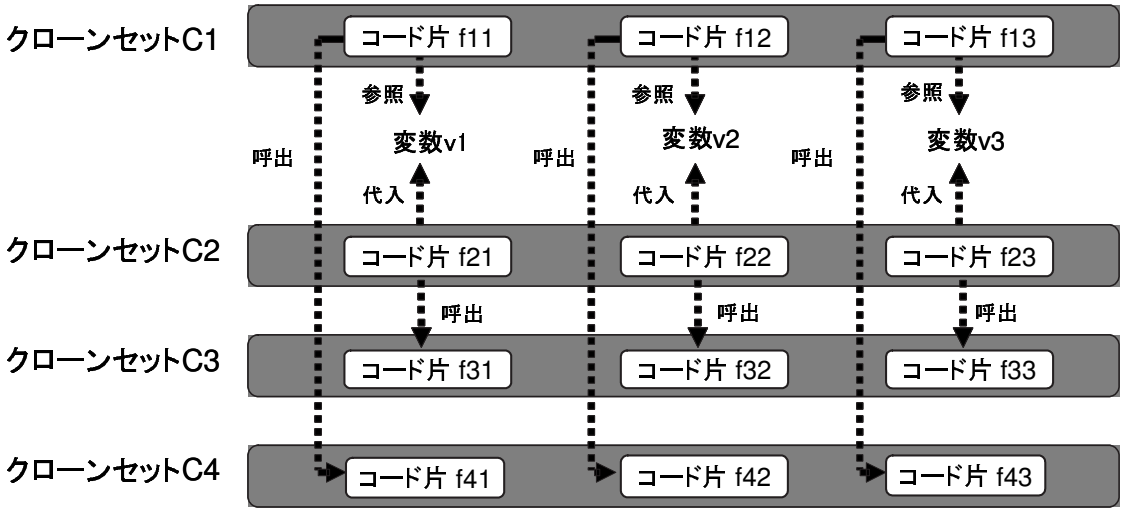


図 13: チェーンドクローンセットの例

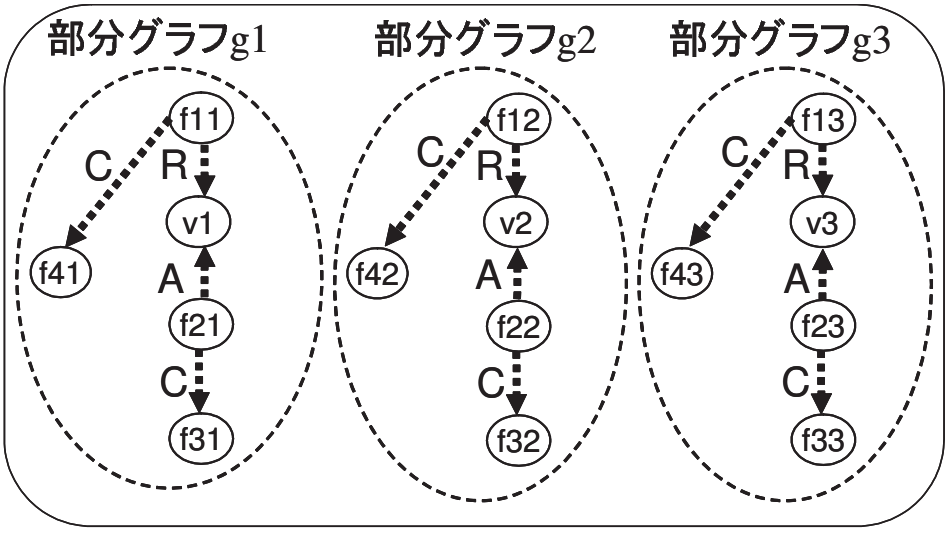


図 14: 図 13 に含まれるコード辺間の依存関係を表すラベル付き有向グラフ

また対応する辺のラベルが等しい．よって，コード片集合 $F_{g_1} = V(g_1) = \{f_{11}, f_{21}, f_{31}, f_{41}\}$ と $F_{g_2} = V(g_2) = \{f_{12}, f_{22}, f_{32}, f_{42}\}$ は，互いにチェーンドクローンであると言える．

また，同様にして，F_{g_1} と $F_{g_3} = V(g_3) = \{f_{13}, f_{23}, f_{33}, f_{43}\}, F_{g_2}$ と F_{g_3} もチェーンドクローンであると言える．また，これらコード片集合の同値類 $CCS = \{F_{g_1}, F_{g_2}, F_{g_3}\}$ はチェーンドクローンセットであると言える．

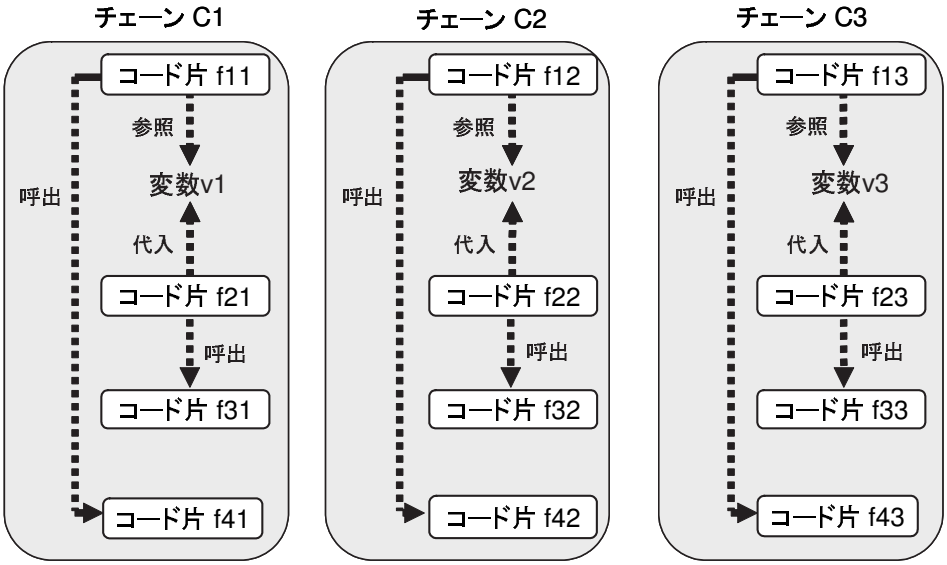


図 15: 図 13 に含まれるチェーン

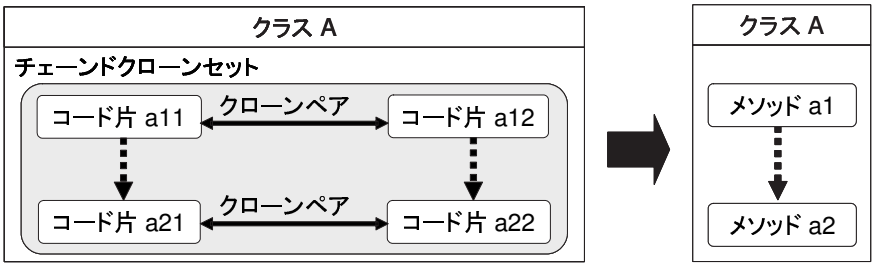


図 16: ケース 1

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

チェーンドクローンセットは，特徴に応じて四つのケースに分けることが出来る．各ケースに対するリファクタリングについて紹介する．

ケース 1 は，チェーンドクローンセットが一つのクラスに包含されている場合である．ケース 1 では，その一つのクラス内にコードクローンを集約可能である．図 16 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コード片 a_{11} とコード片 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コード片 a_{21} とコード片 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二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コード片は，全て兄弟クラスに所属する．

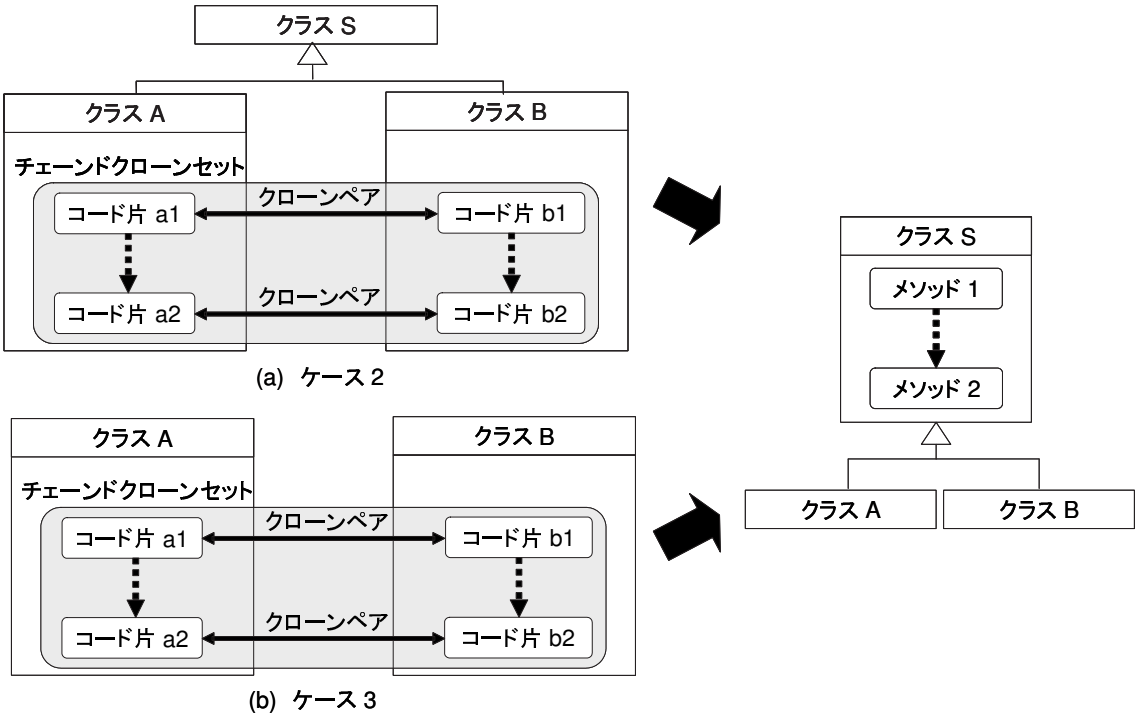


図 17: ケース 2 とケース 3

- 各チェーンは，それぞれ一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ペアを，親クラスに集約することでリファクタリングできる．図 17(a) 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二つのコードクローンを，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 のコード片 a_1 とクラス B のコード片 b_1 を集約し，親クラス S のメソッド 1 とし，同様にクラス A のコード片 a_2 とクラス B のコード片 b_2 を集約し，親クラス S のメソッド 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二つの条件を満たす場合ある．

- チェーンドクローンセットに含まれるコード片を持つクラスは，いずれも共通の親クラスを持たない．
- 各チェーンは，それぞれ一つのクラスに包含されている．

ケース 3 は，“Extract Super Class” 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コード片を持つクラスに，共通の親クラスを

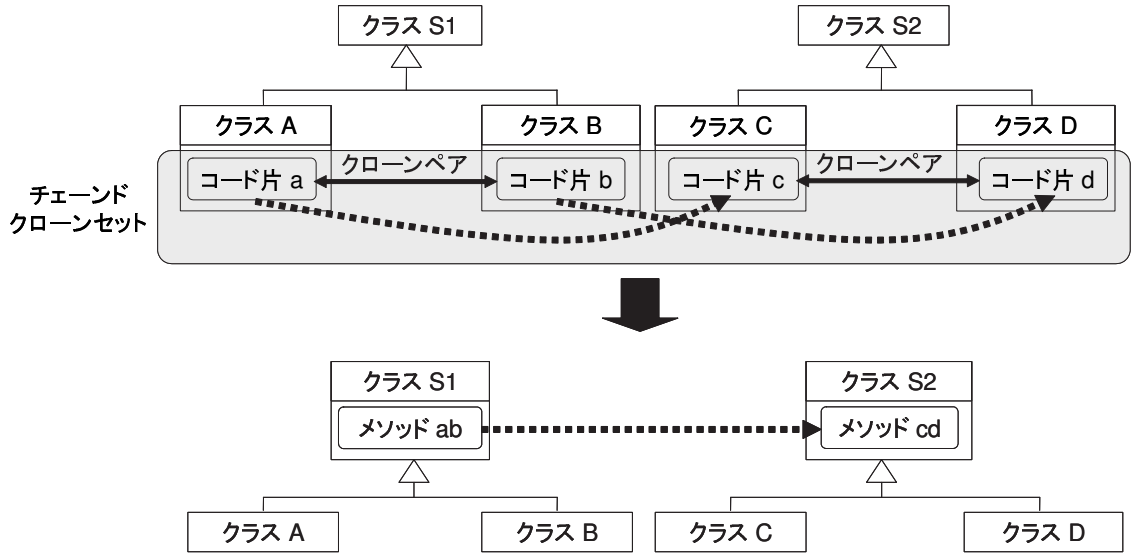


図 18: ケース 4

作成し，クラス間をまたがって存在するクローンペアを，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 17(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二つのクラス 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二つのコードクローンを，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一つの条件を満たす場合である．

- 各チェーンは，それぞれ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．

ケース 4 は，全てのコードクローンをまとめてリファクタリングすることができない場合である．図 18 は，ケース 4 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二つのクローンペアを，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 のコード片 a_1 とクラス B のコード片 b_1 を集約し，親クラス S_1 のメソッド 1 とし，同様にクラス A のコード片 a_2 とクラス B のコード片 b_2 を集約し，親クラス S_2 のメソッド 2 としている．ケース 4 では，チェーンドクローンセットを複数のクローンペアとして扱い，それぞれの親クラスに集約する．つまり，チェーンドクローンセット単位で一度にリファクタリングできない．

3.4 チェーンドクローンセットの分類

前節の四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．前節の四つのケースには，それぞれ適合するための条件があった．これらは，以下の二種類の関係についての条件である．

C1 チェーンドクローンセットに含まれるコード片が所属するクラス間の関係についての条件

C2 チェーンを構成するコード片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．

R1 全ての同一クラス

R2 兄弟クラス

R3 共通の親クラスを持たないクラス

条件の種類と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．

次の三つの分類については，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16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，コードクローン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17(a)の例のように，“Pull Up Method” パターンを適用できる．

表 1: チェーンドクローンセットの分類

<table>
<tr><th>C1 \ C2</th><th>R1</th><th>R2</th><th>R3</th></tr>
<tr><td>R1</td><td>分類 1</td><td rowspan="3">分類 4</td><td rowspan="3"></td></tr>
<tr><td>R2</td><td>分類 2</td></tr>
<tr><td>R3</td><td>分類 3</td></tr>
</table>

- 分類 3

  前節のケース 3 である．図 17(b) の例のように，“Extract Super Class” パターンを適用できる．

- 分類 4

  前節のケース 4 である．図 18 の例のように，全てのコード片をまとめてリファクタリングできない．

3.5 メトリクスを用いたチェーンドクローンセットの自動分類

本稿では，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二つ提案する．一つは C1 を評価するメトリクスと，もう一つは C2 を評価するメトリクスである．これらメトリクスは，与えられたコード片群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 $DCH(S)$ メトリクスによって表すことができる．$DCH(S)$ メトリクスは，与えられたコード片間のクラス階層内における最大の距離を示す．

まず，$DCH(S)$ メトリクスを用いて、C1 を表す $DCHS(CCS)$ メトリクスを定義する。チェーンドクローンセット CCS を n 個のクローンセットの部分集合 $S_1, S_2, \cdots, S_n$ に分割する (すなわち，$S_1 \cup S_2 \cup \cdots \cup S_n = CCS, S_i \cap S_j = \emptyset, 1 \leq i \leq n, 1 \leq j \leq n, i \neq j$)．更に，クローンセットの部分集合 S_i には，m 個のコード片 $F_1, F_2, \cdots, F_m$ が含まれるとする．このとき，$DCHS(CCS)$ メトリクスの定義は以下のようになる．

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

次に，同様に $DCH(S)$ メトリクスを用いて，C2 を表す $DCHS(CCS)$ メトリクスを定義する．チェーンドクローンセット CCS 中には，n 個のチェーン $FC_0, FC_1, \cdots, FC_n$ が含まれるとする．更に，チェーン FC_i には，m 個のコード片 $F_1, F_2, \cdots, F_m$ が含まれるとする．このとき，$DCHD(CCS)$ メトリクスの定義は以下のようになる．

$$DCHD(CCS) \quad = \quad max\{DCH(FC_1), \cdots, DCH(FC_n)\}$$

図 19 は，提案する二つのメトリクスを算出する例である．例として，分類 2 のチェーンドクローンセットを用いている．このチェーンドクローンセットには，クラス A，B，C にまたがって二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 $DCH(S)$ メトリクスを求めると，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．よって，$DCHS(CCS)$ メトリクスは，これら $DCH(S)$ メトリクスの最大値のため，1 となる．また，このチェーンドクローンセットには，三つのチェーン FC_1, FC_2, FC_3 が含まれている．各チェーンについては，それぞれ $DCH(S)$ メトリクスを

求めると，各チェーンはそれぞれ一つのクラスに包含されているため，全て 0 になる．よって，$DCHS(CCS)$ メトリクスは，これら $DCH(S)$ メトリクスの最大値のため，0 となる．

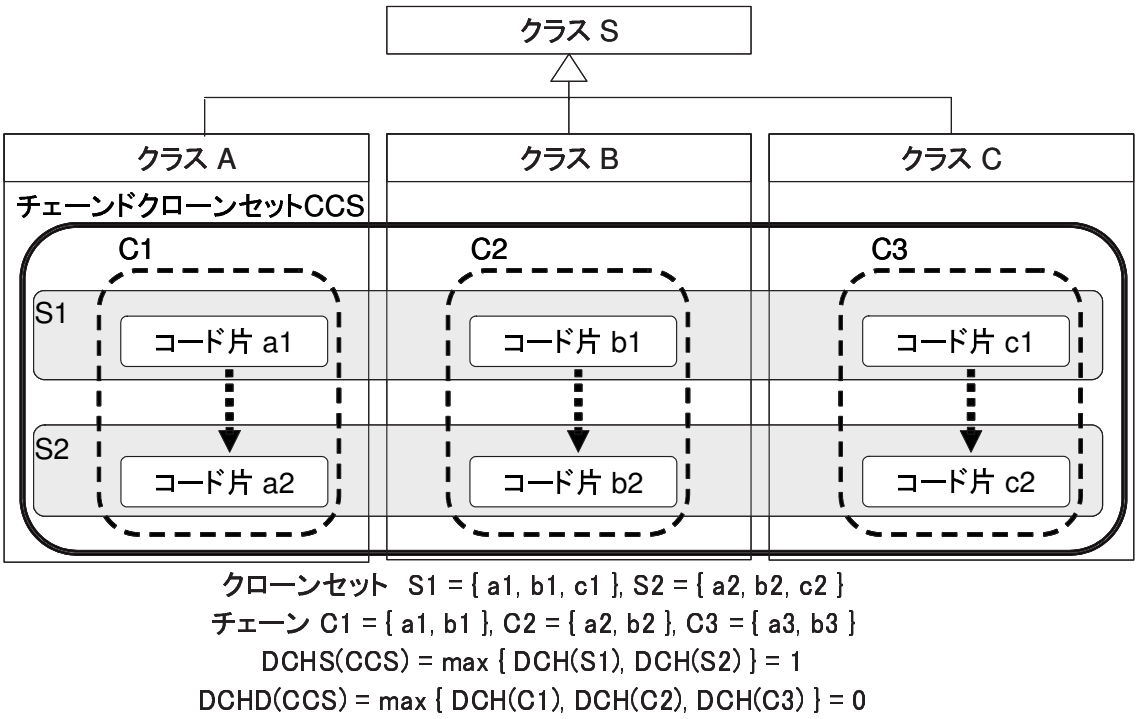


図 19: 提案するメトリクスの算出例

4 実装

4.1 概要

提案手法を Aries の追加機能として実装した．具体的には，Aries に対して，以下の三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，CCFinder および Aries を用いてクローンセットを検出し，そのクローンセット間の依存関係を解析することにより実現した．(F2) の実装は，Aries の $DCH(S)$ メトリクスを計算する機能を拡張することにより行った．また，(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．(F3) の詳細は，次節で詳述する．

4.2 ユーザインタフェース

4.2.1 Chained Clone Set List

図 20 は，*Chained Clone Set List* のスナップショットである．

この画面には，検出されたチェーンドクローンセットの一覧が表示される．各チェーンドクローンセットは，分類毎に表示される．また，各チェーンドクローンセットに対して，3.5 節で提案した $DCHS(CCS), DCHD(CCS)$ メトリクスや，2.6.3 節で紹介した $LEN(S), POP(S), DFL(S)$ メトリクスの値がそれぞれ表示されている．

ユーザはこの画面で，各分類に所属しているチェーンドクローンセットの数や規模を知ることができる．また，関心のあるチェーンドクローンセットを選択すると，*Chained Clone Set View* が開き，そのチェーンドクローンセットの詳細を確認することができる．

4.2.2 Chained Clone Set View

図 21 は，*Chained Clone Set View* のスナップショットである．

この画面には，*Chained Clone Set List* で選択したチェーンドクローンセットの詳細が表示される．この画面は，チェーンドクローンセットに含まれているクローンセットのリストである *Clone Set List*，各コード片とその依存関係をグラフとして表示する *Depedence Graph View* の二つから構成される．

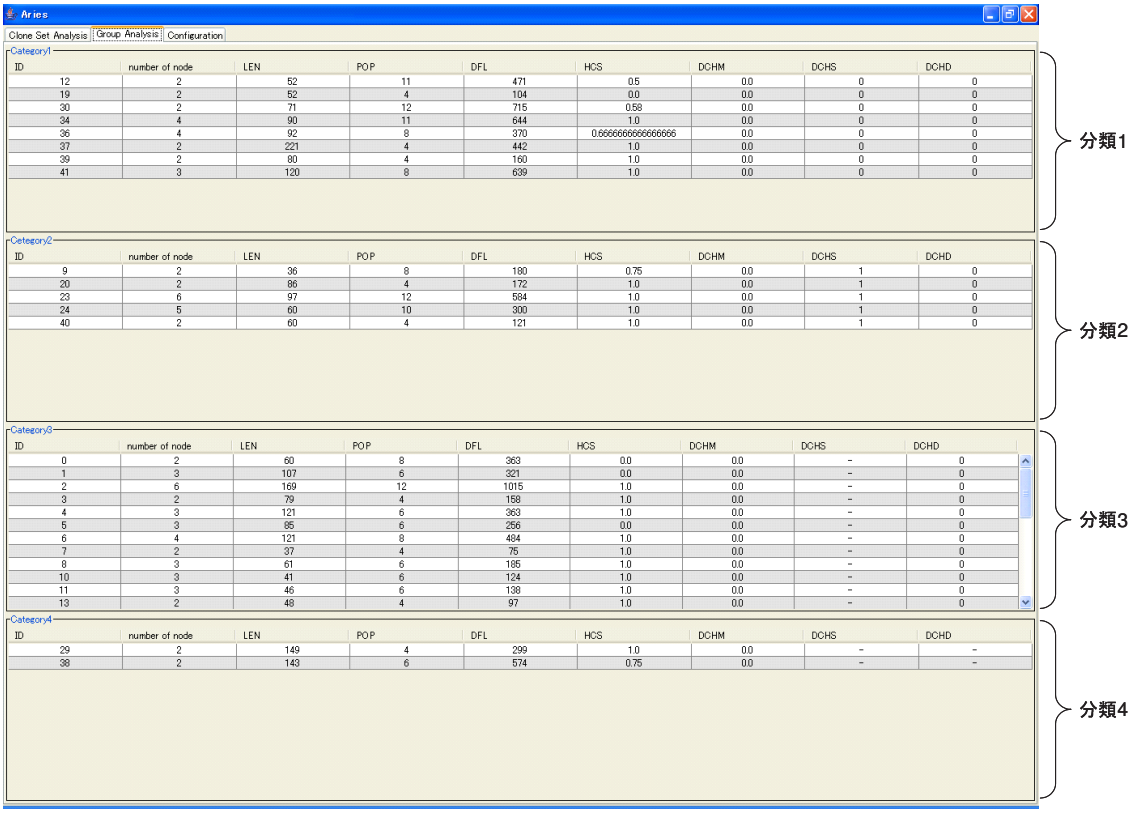


図 20: Chained Clone Set List

ユーザはこの画面で，選択したチェーンドクローンセットにどのクローンセットが含まれているか，またどのような依存関係が含まれているかを確認することができる．また，関心のあるクローンセットを選択すると，*Source Code View* が開き，含まれるコード片を閲覧することができる．

4.2.3 Source Code View

2.6.3 節で紹介した *Source Code View* と同一のものである．ユーザは，*Chained Clone Set View* において選択したクローンセットを構成しているコード片を閲覧することができる．

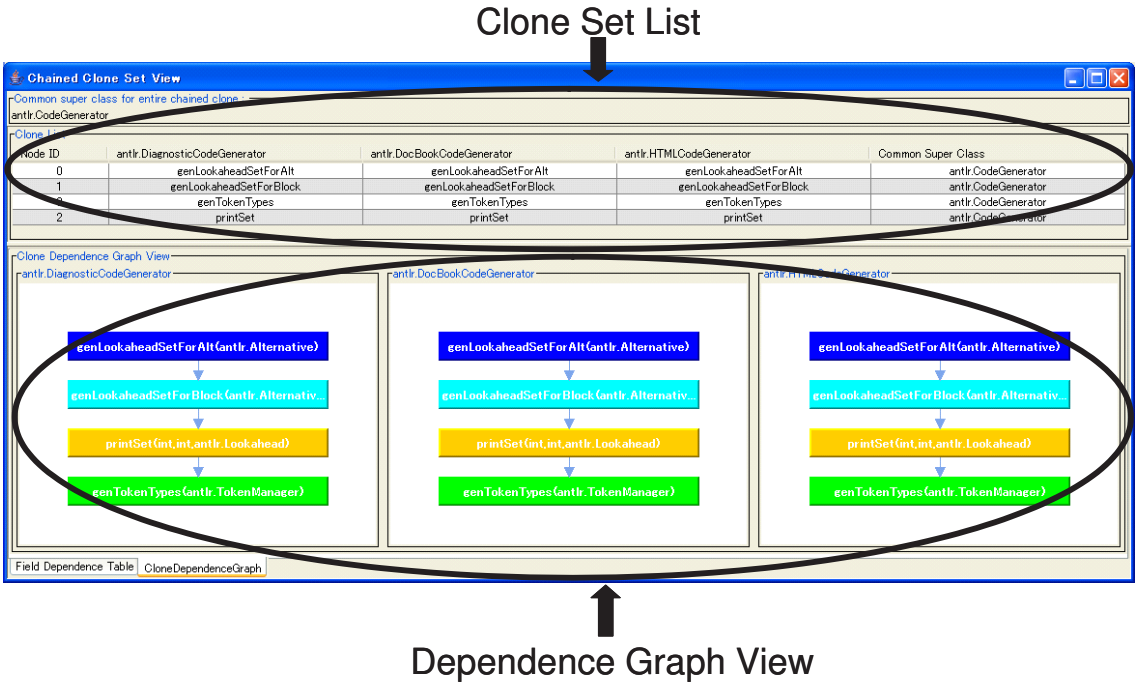


図 21: Chained Clone Set View

5 ケーススタディ

5.1 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．ケーススタディの目的は，次の二つである．

- 実際のソフトウェアにおけるチェーンドクローンセットの数，規模の調査
- 提示するリファクタリングパターンの妥当性の確認

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．また，4 章で述べたように，提案手法の実現には CCFinder を用いている．ケーススタディでは，CCFinder が検出するコードクローンの最小トークン数を 30 トークンに設定した．

対象ソフトウェアは以下の二つである．

- ANTLR 2.7.4[2](4.7 万行，285 クラス）
- JBoss 3.2.6[17](64 万行， 3364 クラス）

ANTLR は，多くの言語（Java，C#，C++等）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．

5.2 チェーンドクローンセットの検出

前述の二つのソフトウェアに対し，チェーンドクローンセットの検出を行った（表 2，表 3）．表中の “コード片数” は，チェーンドクローンセットを構成しているコード片の数である．

まず，ANTLR の結果を見ると，分類 2 のチェーンドクローンセットが多く検出され，また含まれるコード片も多い（表 2）．この原因は，Java，C#，C++に対応した解析器を生成する部分の多くが，コードクローンになっていたことにあると考えられる．具体的には，出力であるパーザーを作成するクラスとして CodeGenerator クラスおよび，その子クラスとして JavaCodeGenerator クラス，CppCodeGenerator クラス，CSharpCodeGenerator クラスを用意しているため，これら三つの子クラス間で大量にコードクローンが存在し，チェー

表 2: チェーンドクローンセットの検出結果（ANTLR 2.7.4）

分類	検出数	コード片数	
		最大	最小
1	6	11	4
2	8	120	6
3	1	4	4
4	0	0	0
計	15	-	-

表 3: チェーンドクローンセットの検出結果（JBoss 3.2.6）

分類	検出数	コード片数	
		最大	最小
1	52	37	4
2	45	35	4
3	36	29	4
4	8	65	4
計	141	-	-

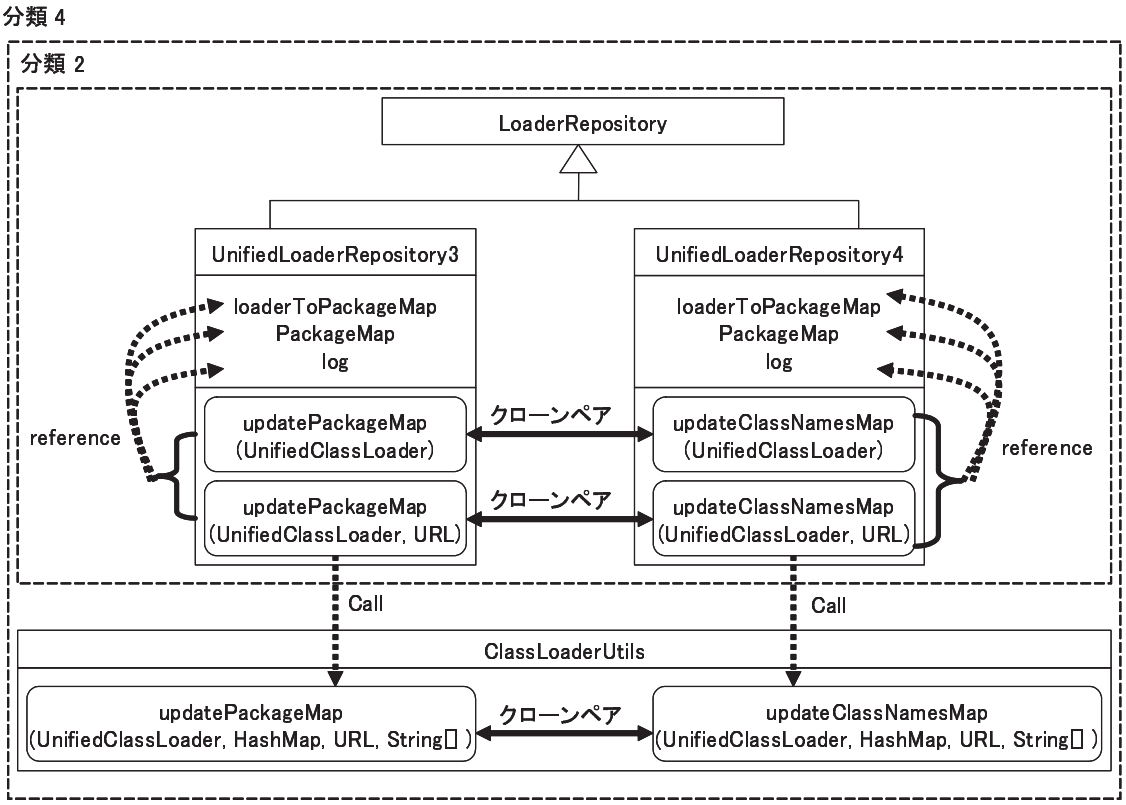


図 22: 分類 4 のチェーンドクローンセットの例

ンドクローンセットを構成していた．

次に，JBoss の結果を見ると，ANTLR の結果と比較して分類 3 のチェーンドクローンセットが多く見つかっている．この原因は，JBoss はいくつかのソフトウェアを内部に含んでおり，そのソフトウェア間でクローンセットが存在するためであると考えられる．また，JBoss では，分類 4 のチェーンドクローンセットが見つかっている（表 3，図 22）．このチェーンドクローンセットの特徴的な点は，内部に分類 2 のチェーンドクローンセットが含まれていることである．

5.3　リファクタリングパターンの適用

提案手法が提示するリファクタリングパターンの妥当性の確認するために，検出されたチェーンドクローンセットに対し，提示されたリファクタリングパターンを適用した．

図 23 は，ANTLR から検出された分類 2 のチェーンドクローンセットである．これに，分類 2 に対応するリファクタリングパターンである “Pull Up Method” パターンを適用することにより，図 24 のようにリファクタリングできた．図 23 と図 24 を比較すると，親子クラ

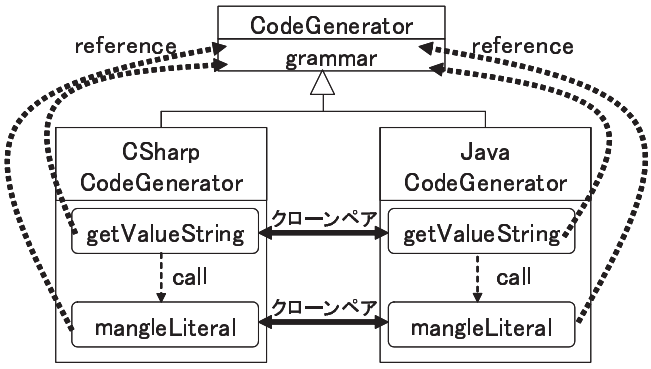


図 23: メソッドで構成される分類 2 のチェーンドクローンセットの例（リファクタリング前）

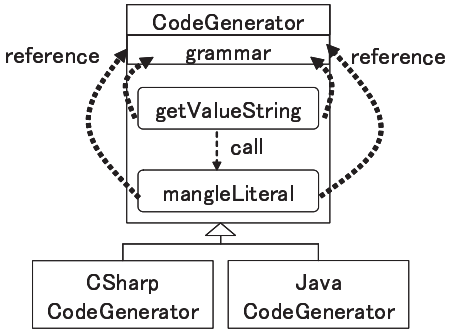


図 24: メソッドで構成される分類 2 のチェーンドクローンセットの例（リファクタリング後）

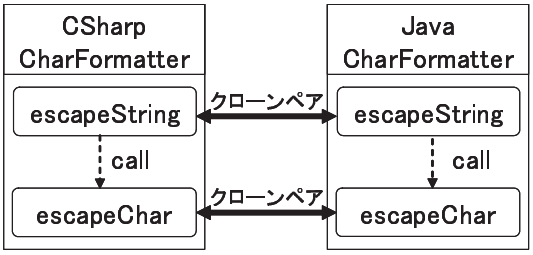


図 25: メソッドで構成される分類 3 のチェーンドクローンセットの例（リファクタリング前）

ス間の結合がなくなっていることがわかる．

図 25，図 26 は，ANTLR から検出された分類 3 のチェーンドクローンセットに対して行ったリファクタリングである．分類 3 に対応するリファクタリングパターンである “Extract Super Class” パターンを適用している．

図 27，図 28 は，ANTLR から検出された分類 2 のチェーンドクローンセットに対して行ったリファクタリングである．分類 2 に対応するリファクタリングパターンである “Pull Up

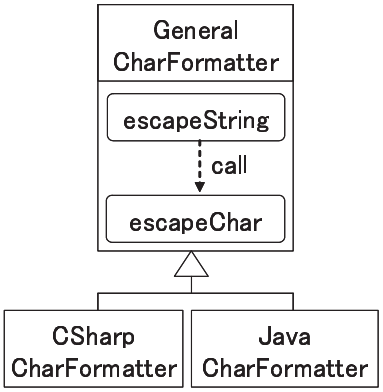


図 26: メソッドで構成される分類 3 のチェーンドクローンセットの例（リファクタリング後）

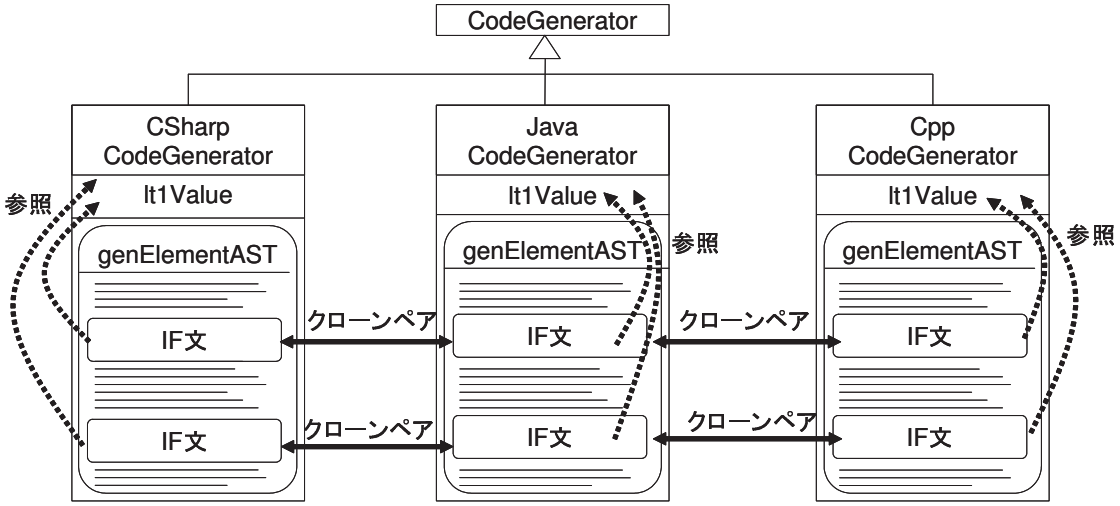


図 27: IF 文で構成される分類 2 のチェーンドクローンセットの例（リファクタリング前）

Method” パターンを適用している．具体的には，まず共通の親クラスである CodeGenerator クラスに newMethod1 メソッドと newMethod2 メソッドを作成する．次に，チェーンドクローンセットを構成している各 IF 文を，CodeGenerator クラス作成した二つのメソッドに集約している．最後に，チェーンドクローンセットを構成していた IF 文が参照していた lt1Value 変数を，共通の親クラスである CodeGenerator クラスに引き上げている．

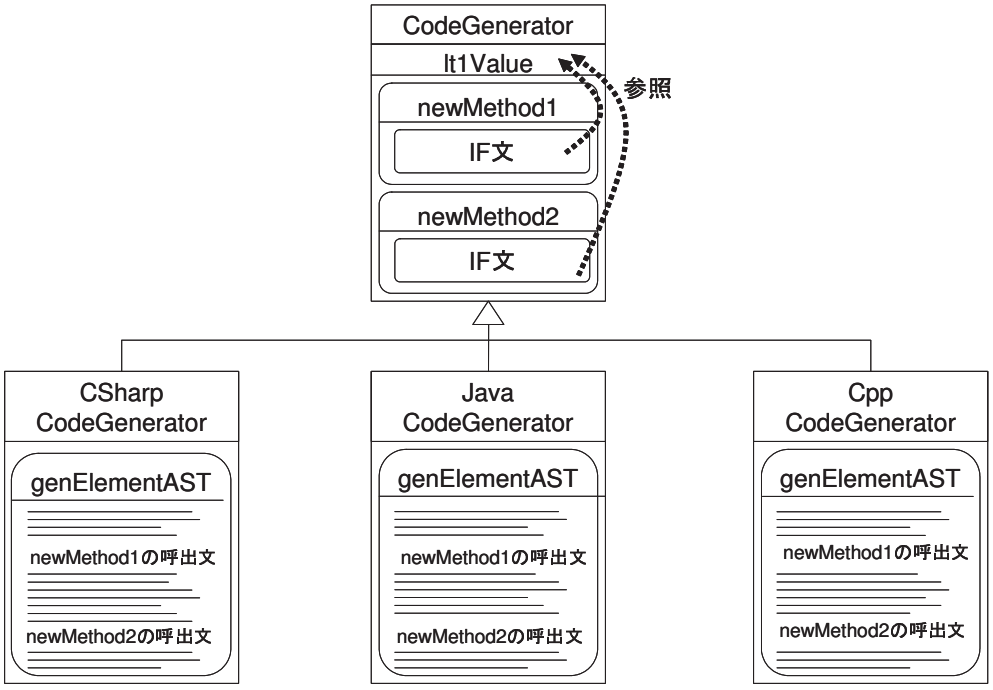


図 28: IF 文で構成される分類 2 のチェーンドクローンセットの例（リファクタリング後）

5.4 考察

5.4.1 チェーンドクローンセットの検出

ケーススタディにおいて，二つのオープンソフトウェアにチェーンドクローンセットが含まれていることを確認できた．

今回は，メソッド呼出および変数の共有による依存関係のみを扱った．これらの依存関係を扱うことにより，最大で 120 個のコード片からなるチェーンドクローンセットを検出することができた．今後，より多くのコード片をチェーンドクローンとして扱うには，二つの方法が考えられる．一つ目は，扱う依存関係の種類を増やす方法である．本稿では，メソッド呼出と変数の共有から生じる依存関係のみを扱ったが，その他のデータ依存関係や制御依存関係を用いることにより，チェーンドクローンの数や規模を大きくすることができると考えられる．

ケーススタディにおいて，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 22）．提案手法では，このチェーンドクローンは分類 4 であるため，チェーンドクローンセット単位のリファクタリングはできないと提示する．しかし，このチェーンドクローンが内包する分類 2 のチェーンドクローンセットは，“Pull Up Method” リファクタリングパターンを用いてリファクタリングすることができる．

このような異なる分類のチェーンドクローンセットを内包するチェーンドクローンセットは，内包するチェーンドクローンセットの情報を提示する方が望ましい．

5.4.2 リファクタリングパターンの適用

検出したチェーンドクローンセットに対し，分類に応じたリファクタリングパターンを適用することができた．また，検出されたチェーンドクローンセットの中には，類似したクラス間に存在する大量のコードクローンがある箇所や，内部に含まれる複数のソフトウェア間のコードクローンがある箇所に含まれるものがあった．このことから，チェーンドクローンセットの検出を行うことより，各ソフトウェア設計上の問題点を明らかにすることが出来たといえる．更に，これらチェーンドクローンセットに対し，分類に応じたリファクタリングパターンを適用すると，それぞれの問題点を改善することができた．

図 27，図 28 は，Gapped クローン（不一致部分を含むコードクローン）[24] でもある．この例のように，検出されたチェーンドクローンセットが Gapped クローンであることがある．よって，文献 [24] で提案されている Gapped クローン検出手法により，少数のチェーンドクローンセットを検出できる．しかし，本研究の提案手法では，チェーンドクローンセットに含まれるコード片に加えて，チェーンドクローンセットに含まれる複数のコード片が共用している変数（例：図 27 の ltValue 変数）を提示できる．“Pull Up Method” や “Extract Method” を行う手順の一環として，チェーンドクローンセットに含まれる複数のコード片が共用している変数 “Pull Up Field” の検討を行う必要があるため，このような変数の提示は有益なリファクタリング支援であると考えられる．

また，5.3 節で述べたファクタリングを行った後，回帰テストを行ったところ，外部的振る舞いを変化させることなくコードクローンを集約できていることが確認できた．

6　むすび

本稿では，ソフトウェア保守を困難にする要因の一つであるコードクローンを効果的にリファクタリングする手法の提案を行った．まずクローンセット間に依存関係が存在する場合に着目し，そのようなコードクローンをチェーンドクローンセットとして定義した．次に，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が異なることを示した．そして，メトリクスを用いることにより，チェーンドクローンセットの特徴に応じたリファクタリングパターンを提示する手法を提案した．

提案手法の妥当性を確認するためのケーススタディを行った．ケーススタディでは，提案手法をリファクタリング支援ツールとして実装し，二つのオープンソースソフトウェアに適用した．その結果，チェーンドクローンセットに対して提示するリファクタリングパターンが妥当であることを確認できた．

今後の課題として，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をすること，扱うリファクタリングパターンを増やすこと，対象とするコードクローンを増やすことが挙げられる．対象とするコードクローンの範囲を広げるためには，扱う依存関係の種類を広げる方法が考えられる．このような手法の拡張に並行して，開発現場においてチェーンドクローンセットに対するリファクタリングを行うことで，保守性を向上できるか評価したい．

また本研究では，リファクタリング対象としてコードクローンを提示したが，コードクローンを提示することにより支援できる作業は他にもある．例えば，バグフィックスや機能追加の際，修正の対象となるコード片のコードクローンを提示することにより，修正漏れを防ぐことができる．そこで今後，リファクタリングの対象を探すことにではなく，コード片の修正作業を支援することに着目して，開発支援を行いたいと考えている．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました 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 井上 克郎 教授に心より深く感謝致します.

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました 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 楠本 真二 教授に心から感謝致します.

本論文を作成するにあたり，適切な御指導，御助言を賜りました 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 松下 誠 助教授に心から感謝致します.

本研究を通して，逐次適切な御指導，御助言を頂きました 産業技術総合研究所 研究員 神谷 年洋 氏に深く感謝致します.

本研究において，様々な御協力を頂きました 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 肥後 芳樹 氏に深く感謝します.

最後に，その他様々な御指導，御助言等を頂いた 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 井上研究室の皆様に深く感謝いたします.

参考文献

[1] A. Aiken. A System for Detecting Software Plagiarism (Moss Homepage). `http://www.cs.berkeley.edu/~aiken/moss.html`.

[2] ANTLR. `http://www.antlr.org`.

[3] B. S. Baker. A Program for Identifying Duplicated Code. In *Proc. of Computing Science and Statistics*, Vol. 6, pp. 49–57, 1992.

[4] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of WCRE 2000*, pp. 86–95, 1995.

[5] B. S. Baker. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal on Computing*, Vol. 26, No. 5, pp. 1343–1362, 1997.

[6] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring Clone Based Reengineering Opportunities. In *Proc. of METRICS '99*, pp. 292–303, 1999.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proc. of WCRE '99*, pp. 326–336, 1999.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of WCRE 2000*, pp. 98–107, 2000.

[9] I.D. Baxter, A. Yahin, L. Moura, M.S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of ICSM '98*, pp. 368–377, 1998.

[10] E. Burd and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proc. of SCAM 2002*, pp. 36–43, 2002.

[11] M. Dorfman and R. H. Thayer. *Software Engineering.* IEEE Computer Society Press, 1997.

[12] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of ICSM '99*, pp. 109–118, 1999.

[13] N. Ford and M. Woodroffe. *Introducing software engineering.* Prentice-Hall, 1994.

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

[15] D. Gusfield. *Algorithms on Strings, Trees, And Sequences.* Cambridge University Press, 1997.

[16] *IEEE Std 1219: Standard for Software Maintenance.* 1997.

[17] JBoss. `http://www.jboss.org`.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[19] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proc. of SAS 2001*, pp. 40–56, 2001.

[20] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of WCRE 2001*, pp. 301–309, 2001.

[21] G. Malpohl L. Prechelt and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038, 2002.

[22] Pigoski T. M. *Encyclopedia of Software Engineering*, Vol. 1, chapter Maintenance. John Wiley & Sons, 1994.

[23] J. Mayland, C. Leblanc, and E. M. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proc. of ICSM '96*, pp. 244–253, 1996.

[24] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On Detection of Gapped Code Clones using Gap Locations. In *Proc. of APSEC 2002*, pp. 327–336, 2002.

[25] 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.

[26] 植田泰士, 神谷年洋, 楠本真二, 井上克郎. 開発保守支援を目指したコードクローン分析環境. 電子情報通信学会論文誌 D-I, Vol. 86, No. 12, pp. 863–871, 2003.

[27] 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 D-I, Vol. 88, No. 2, pp. 186–195, 2005.