特別研究報告

題目

クラス階層内のコードクローン間距離を評価するメトリクス DCH の改良

指導教員
楠本 真二 教授

報告者
佐野　由希子

平成 20 年 2 月 19 日

大阪大学 基礎工学部 情報科学科

平成 19 年度 特別研究報告

クラス階層内のコードクローン間距離を評価するメトリクス DCH の改良

佐野　由希子

内容梗概

ソフトウェアの保守作業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．コードクローンとは，ソースコード中に同一，または類似したコード片を持つコード片のことであり,「重複コード」とも呼ばれている．もし，あるコードクローンにバグが見つかった場合，そのコードクローンすべてに同じ修正を行うかどうかを確認する必要がある．

コードクローンに対する集約を支援するためのメトリクスの 1 つとして DCH がある．DCH はクラス階層内におけるコードクローン間距離を評価する．例えば，対象コードクローンが 1 つのクラス内に存在する場合はそのクラス内で集約することができ，共通する親クラスを持つ複数の子クラス内に存在する場合はその共通の親クラスに引き上げることによって集約を行える．しかし，それ以外の場合は，コードクローンを集約することは難しい．

本研究では，メトリクス DCH の改良を行う．従来の DCH は，1 つのクローンセット (コードクローンの同値類) 全体から計測されるため，クローンセット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散らばっている場合の区別がつかない．後者は集約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．本手法で改良した DCH を用いれば前者と後者の区別を簡単に行うことができる．また，改良した DCH を計測するツールを試作し，オープンソースソフトウェアに対して実験を行った．実験の結果，実際のソースコードから従来の DCH よりも多くの集約可能なコードクローンを検出できることを確認した．

主な用語

コードクローン
ソフトウェア保守
リファクタリング

目 次

1　まえがき

近年，ソフトウェアシステムの大規模化，複雑化に伴い，ソフトウェアの保守に要するコストが増加してきている．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．それらの多くは，既存システムに対する変更や拡張時における「コピーアンドペースト」による安易な機能的再利用の際に発生する．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ついて修正を行うかどうかを検討しなければならない．このような作業は，特に大規模ソフトウェアでは非常に手間のかかる作業である．

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 [1] 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 [1] の中で，リファクタリングを検討すべき箇所の代表例としてコードクローン（重複したコード）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（ループやメソッド，クラスなど）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，比較的容易に 1 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 1 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．

そのようなコストを削減するための手法の 1 つに，コードクローン検出ツール CCFinder[2] とリファクタリング支援環境 Aries[3] が挙げられる．CCFinder の特長は，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，リファクタリングパターンの決定支援を目的としたメトリクスの中に DCH がある．

DCH はクラス階層内のコードクローン間距離を評価するメトリクスである．基本的に，この距離は近い方がリファクタリングを行いやすい．しかし，クローンセット全体での距離を評価するため，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散らばっ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と思われる．

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．具体的には，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 を設定する．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．

以降 2 節では，コードクローンに対する諸定義，リファクタリング支援環境 Aries について説明する．3 節では，従来の DCH の問題点と，その問題点を解決する手法の提案を行う．4 節では，3 節で提案した手法を用いて行った適用実験について説明し，その考察を 5 節で行う．最後に 6 節で本研究のまとめと今後の課題について述べる．

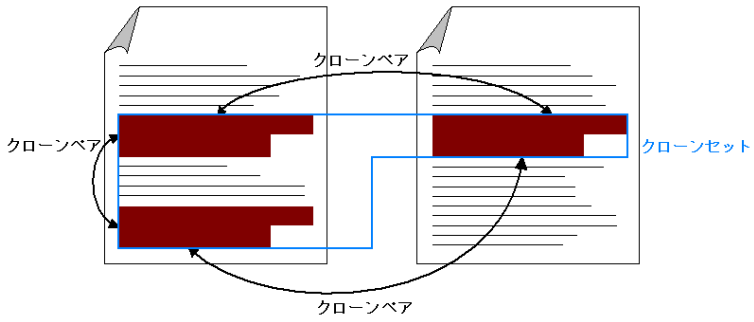

図 1: クローンペアとクローンセット

2 準備

2.1 コードクローン

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（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 [4].

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 [2][5][6].

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．

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある．例えば，ユーザインターフェース処理を記述するコードなどである．

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．

複数のプラットフォームに対応したコード

複数の OS(Linux, FreeBSD, HP-UX や AIX など) や CPU(i386 系, amd64 系, alpha や sparc64 など) に対応したソフトウェアは，各プラットフォーム用のコード部分に重複した処理が存在する傾向が強い．

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，3 つのコード片がコードクローンとなっていた場合，その 1 つに対して修正を行うと，他の 2 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とする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 1 つ 1 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．

このようなコードクローンによる問題に対処する方法としては，以下の 2 つが考えられる [4]．

コードクローン情報の文書化

コードクローンに関する情報が文書化され，断続的に保守されている場合には，コードクローンに対する変更は幾分やさしくなる．しかし，すべてのコードクローンに対

する情報を常に最新に保つ作業は非常に手間がかかるため，現実的に困難である場合が多い．

コードクローンの自動検出

コードクローンを形式的に定義し，コードクローンをプログラムテキスト中から自動的に検出するためのさまざまな手法が提案され，検出システムが開発されてきている．

その手法の 1 つであるコードクローン検出ツール CCFinder[2] について 2.4 節で述べる．

このコードクローンの自動検出により，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 2 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 1 つとして，2.3 節で述べるリファクタリングがある．

2.3 リファクタリング

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 [1]．Fowler は文献 [1] 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不吉な匂いと呼び，その代表例としてコードクローン（重複したコード）を挙げている．コードクローンを取り除く手法として，様々な対処方法がある．その内,“ Pull Up Method ”と“ Extract Method ”については 4.1 節と 4.2 節で後述する．

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．

2.4 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 [4].

様々な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検出可能である．

実用的に意味の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の検出を防ぐことができる．
- 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できる．

2.5 CCShaper

リファクタリングを目的としている場合，ただ極大クローンを抽出するよりも，その中のループやメソッド，クラスなどの構造的なまとまりを持った部分を抽出する方が望ましい．CCShaper[7][8] は CCFinder の検出したコードクローンから，そのように構造的なまとまりを持った部分を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．

2.6 Aries

Aries[3] は Java 言語を対象としたリファクタリング支援環境である．CCFinder が検出したコードクローンから，CCShaper を用いて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．Aries が計測するメトリクスには次節で後述する DCH や 4.2 節で後述する NSV も含まれる．

現在，Aries は 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

なお，Aries は解析部と GUI 部とに分かれている．解析部は対象のソースファイルと最小一致トークン数を入力とし，コードクローン情報ファイルを出力する．そして，GUI 部は入力されたコードクローン情報ファイルを視覚的に表示するという構成である．図 2 に GUI 部の Main Window を示す．Main Window の CLONE SET LIST から興味のあるクローンセットを選べば，そのクローンセットのより詳しい情報を Clone Set Viewer（図 3）で見ることができる

この Aries の GUI 部を用いて，次のようにリファクタリングを行える．

ステップ 1

NRV/NSV SELECTOR と CLONE UNIT SELECTOR を用いて，4.2 で後述するメトリクス NSV 等の計算に用いる変数の種類（自クラスのフィールド変数，親クラスのフィールド変数，ローカル変数，this 変数など）とコードクローンの構造的なまとまりの単位を選ぶ．

ステップ 2

METRIC GRAPH を用いて，各メトリクスの値の上限や下限を設定し，条件に合うクローンセットを絞り込む．絞り込まれたクローンセットは CLONE SET LIST に表示される．

ステップ 3

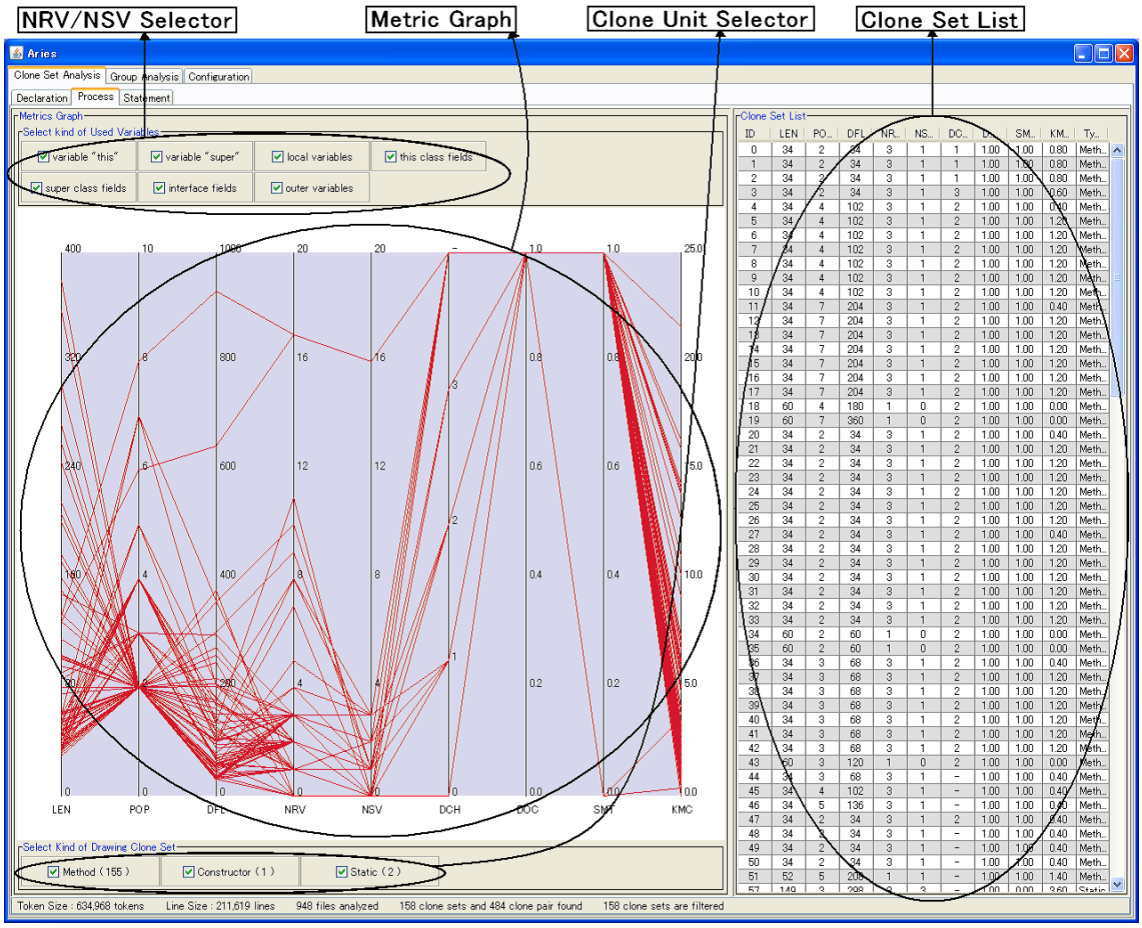

図 2: Main Window

CLONE SET LIST は，各メトリクスについて昇順や降順に並び替えることができる．この機能を用いて，ユーザは興味のあるクローンセットを選ぶ．

ステップ 4

CLONE SET VIEWER を用いて，ステップ 3 で選んだクローンセットの詳しい情報を見ながら，ユーザはこのコードクローンをリファクタリングするかどうか決定する．

2.7 DCH

メトリクス DCH(S) は文献 [3] 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．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，…，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

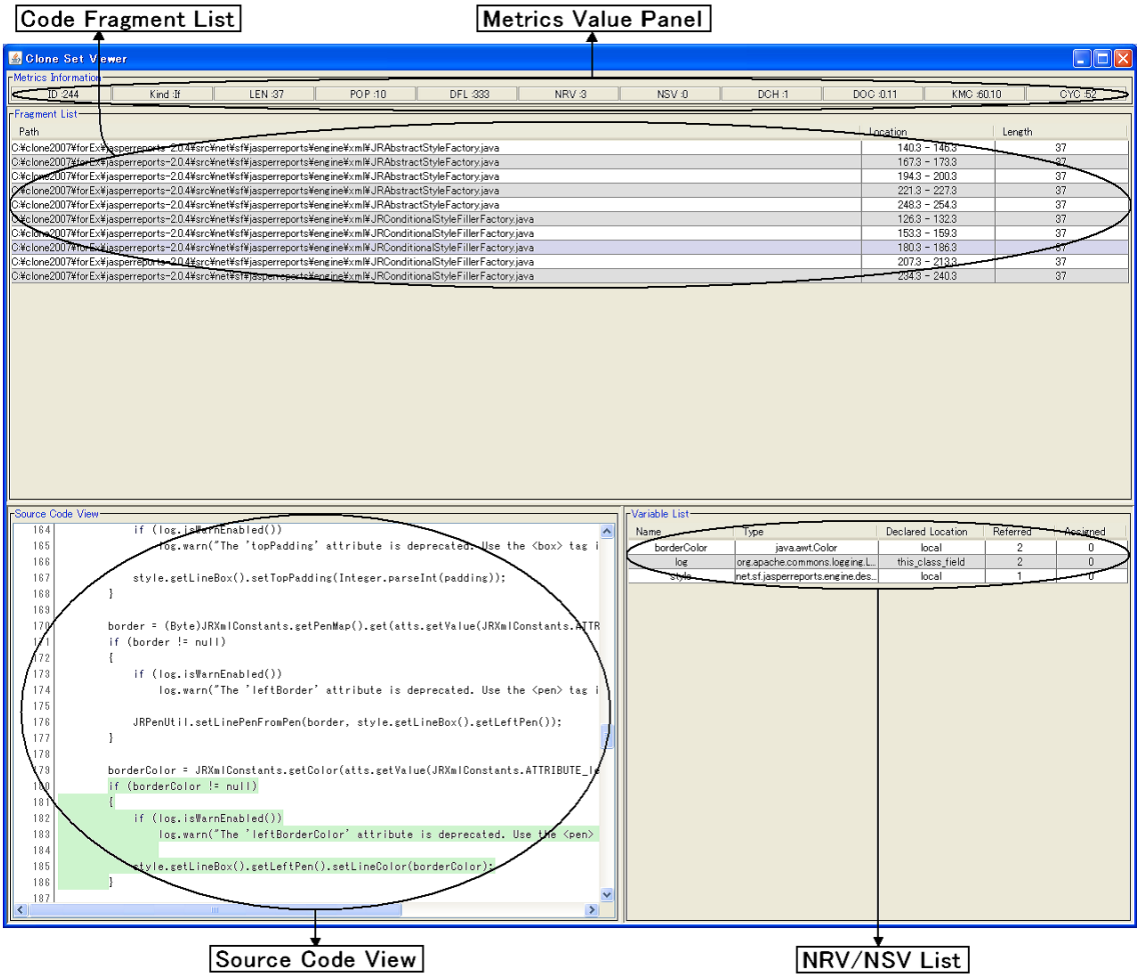

図 3: Clone Set Viewer

図 4: DCH=0

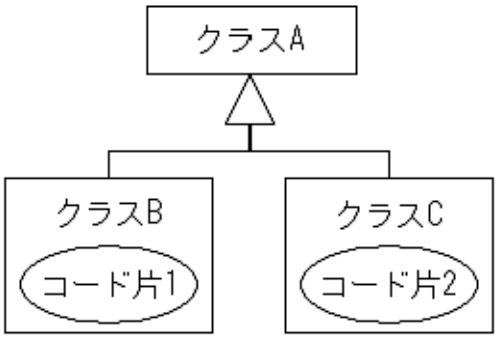

図 5: DCH=1

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,

$$DCH(S) = maxD(C_1, C_p), \cdots, D(C_n, C_p)$$

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 4），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 5）．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 6）．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) 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．

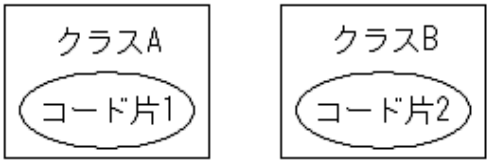

図 6: DCH=∞

2.8 関連研究

2.8.1 コードクローンの集約

コードクローンの集約とは，互いにコードクローンになっているコード片群を 1 つのモジュール (関数やメソッド，アスペクトなど) にまとめることである．コードクローンを集約することにより，コードクローンによって引き起こされる問題を軽減することができる．

Baxter らのツール CloneDr は，コードクローンを検出すると同時に，そのコードクローンを集約するための雛形も出力する [5]．この雛形を用いることによって，集約を行うにはどのようにソースコードを修正すればよいのかを知ることができる．

Balazinska らは，メソッド単位のコードクローンに対する集約支援手法を提案している [9]．この手法では，集約の際に重要な情報となるコードクローン間の違いを提供する．この違いを用いることによって，各コードクローンを集約するのか容易かどうかを判断することができる．また，コードクローンとその周囲との結合度も解析する．結合が低い場合は，容易にコードクローンを他の部分に移動させることができるが，結合が強い場合は，移動させることは困難である．

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 [10][11]．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による記述と，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．実際に，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されている [12][13].

2.8.2 コードクローンの修正支援

前節では，コードクローンの集約支援手法を紹介したが，すべてのコードクローンが集約可能あるいはすべきというわけではない．川口らや Kim らは，オープンソースソフトウェアの複数のバージョンに対して，コードクローンの出現と消失を調査し，次のことを報告し

ている [14][15].

- あるバージョンでコードクローンになったが，その後異なった修正が加えられたことにより，コードクローンでなくなる場合がある．
- 長期間存在するコードクローンは，プログラミング言語に適切な抽象化機構が存在しないなど，集約を行わない理由が存在する．

Kapser らは，コードクローンを集約すべきではない状況をいくつか紹介している [16]．例えば，新しいハードウェアのドライバを作成する場合，既存のドライバから再利用可能な部分をコピーアンドペーストすることが有益であるとしている．既存の正しく動作しているドライバと，新しく作成しているドライバ間のコードクローン部分を集約することは，その際に不具合を混入してしまう危険があるため，そのような集約は行うべきではないとしている．

Balazinska らは，コードクローン間の差異によって，集約の困難さが異なると報告している [17]．Balazinska らの実験結果では，コーディングスタイルなどの表面的な違いのみを含むコードクローンは，利用している変数の型が違うなどの意味的な違いを含むコードクローンに比べ，リファクタリングを容易に行えるという結論であった．

上述のように，すべてのコードクローンが集約に向いているわけではない．Toomim らは，あるコードクローンを修正すると，それと対応するすべてのコードクローンに対して，同様の修正を自動的に施すエディタを開発している [18]．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 [19]．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，変数名や関数名等もまったく同じである，完全一致のコードクローンに対してしか適用することができない．

泉田らは，CCFinder のオプションを適切に設定することによって，入力コード片と対象ソースコード間のコードクローンのみを高速に検出できると報告している [20]．適切に入力コード片を与えることによって，同時に修正すべき箇所を得ることができる．また，佐々木らも，CCFinder の検出したコードクローン情報をソースコード中にコメントとして付加することによって，同時に修正すべき箇所を容易に特定することができると報告している [21].

Mann はコピーアンドペーストの履歴を保存することが有益であると主張している [22]．コピーアンドペーストの履歴をたどることにより，任意のコード片の出自を知ることができ，またこれらは同時に修正すべき候補となりうるとも述べている．開発者は 1 時間に約 4 回のコード片単位でのコピーアンドペーストを行う [23]，及びコピーアンドペースト後に，各コード片には異なった修正がしばしば加えられる [24]，との報告もあることから，コピーアンドペーストの履歴を追うことにより，ソースコードのみを用いたコードクローン検出手

法では検出することのできない，同時に修正すべき箇所を検出することができると考えられる．

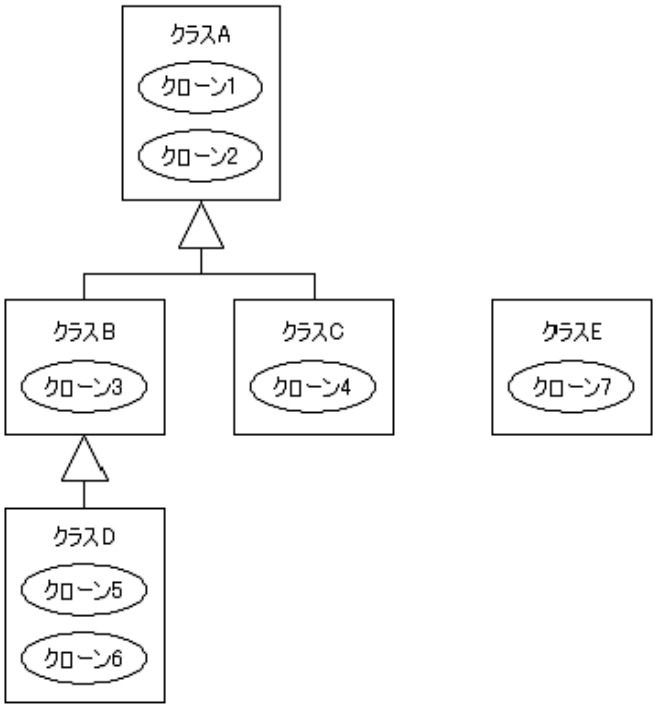

図 7: クローンセット S

3 提案手法

3.1 DCH の問題点

今まで DCH は 1 つのクローンセットにつき，1 つの値を計測していた．このやり方だと，クローンセット内の一部のコード片のために DCH の値が ∞ になってしまったり，大きくなってしまったりする可能性がある．例えば，クローンセット S 内のコードクローン，クローン 1〜クローン 7 が，図 7 のようにクラス A〜クラス E に分布している場合を考える．なお，図中の三角形はクラスの親子関係を表している．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．クローン 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ため，$DCH(S) = \infty$ となる．もしクローンセット S 内にクローン 7 がなければ $DCH(S) = 2$ となるのに，クローン 7 というたった 1 つのコード片のために DCH の値が ∞ になってしまうのである．

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等，細かなリファクタリング要求に応えにくいためである．

3.2 提案手法の概要

そこで本研究では，1 つのクローンセットに対して 1 つ以上の DCH を設定する手法を提案し，その手法を実現するツールの作成を行う．この手法では，クローンセット内のコード片のサブセットごとに DCH を設定する．この手法を用いることによって，より多くのコード片をリファクタリングできると考えられる．

本手法では，クローンセット内のコードクローンでサブセットを作成する．サブセットは，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，…というように作成する．このように分類し，各サブセットごとに DCH の値を設定することで，1 つのクローンセットに対して 1 つ以上の DCH を計測することができる．

この分類のアルゴリズムを下記に示す．

- 全てのクローンセットについて以下を行う．

  1. クローンセット内のコードクローンを，存在するクラスごとに 1 つのグループとする．このとき，複数のコードクローンを含むグループを $DCH = 0$ のサブセットとする．
  2. 1. で作成したグループの全てについて以下を行う．
     (a) そのグループ内のコードクローンが存在するクラスの親クラスをクラス P，h=1 とする．
     (b) クラス P を 0〜h 階層上の親クラスとするクラスに存在するコードクローンのグループを探し，それらを $DCH = h$ のサブセットとする．ただし，$DCH \leq h$ でまったく同じ組み合わせのサブセットが既に作成されていた場合を除く．
     (c) クラス P に親クラスがあれば，クラス P の親クラスをクラス P とし，h の値を 1 増やして (b)〜(c) を繰り返す．

従来の手法では $DCH(S) = \infty$ となってしまっていた図 7 のクローンセット S も，本手法を用いれば，

- $DCH = 0$ のサブセットは (クローン 1, クローン 2)，(クローン 5, クローン 6)
- $DCH = 1$ のサブセットは (クローン 1, クローン 2, クローン 3, クローン 4)，(クローン 3, クローン 5, クローン 6)
- $DCH = 2$ のサブセットは (クローン 1, クローン 2, クローン 3, クローン 4, クローン 5, クローン 6)

- $DCH = \infty$ のサブセットは (クローン 1, クローン 2, クローン 3, クローン 4, クローン 5, クローン 6, クローン 7)

のようになる．

3.3 実装

本手法を実現するために作成したツールが DCHchecker である．DCHchecker の入力として，Aries の解析部が出力するコードクローン情報ファイルを用いるため，Aries と同じく Java 言語で書かれたプログラムのみを対象としている．Aries の出力するコードクローン情報ファイルには，対象プログラムのソースファイルやクラス，コードクローンの情報が書かれている．

DCHchecker の実行結果は図 8 のようになる．cloneSetID，cloneID は，それぞれクローンセット，コードクローンに振った通し番号である．各クローンセットごとに，DCH の値と，それに対応するサブセットに含まれるコードクローンの ID を表示している．複数の DCH が計測されているクローンセットの存在も確認できる．

また，図 9 のように，各コードクローンのソースコードを表示することも可能である．

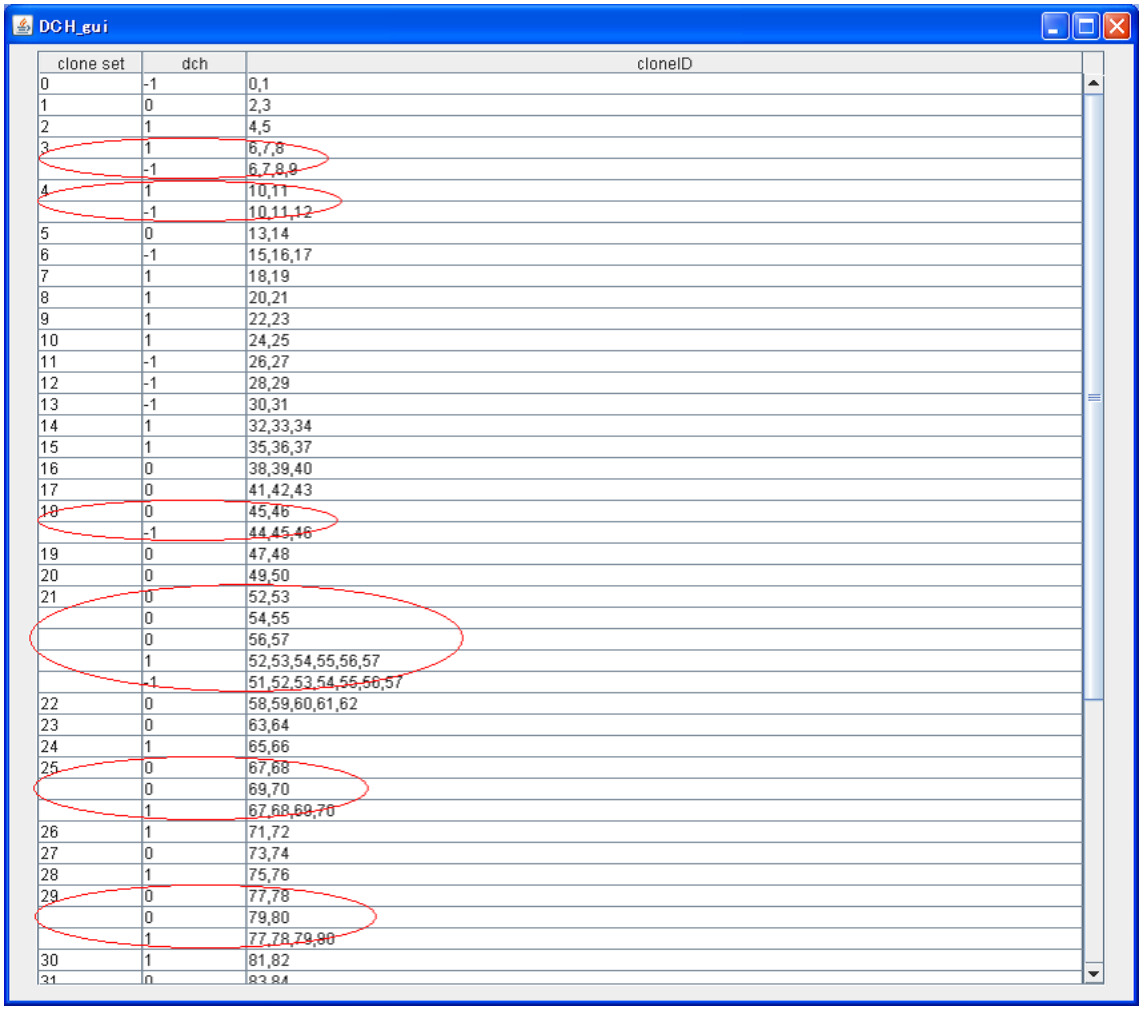

図 8: DCHchecker

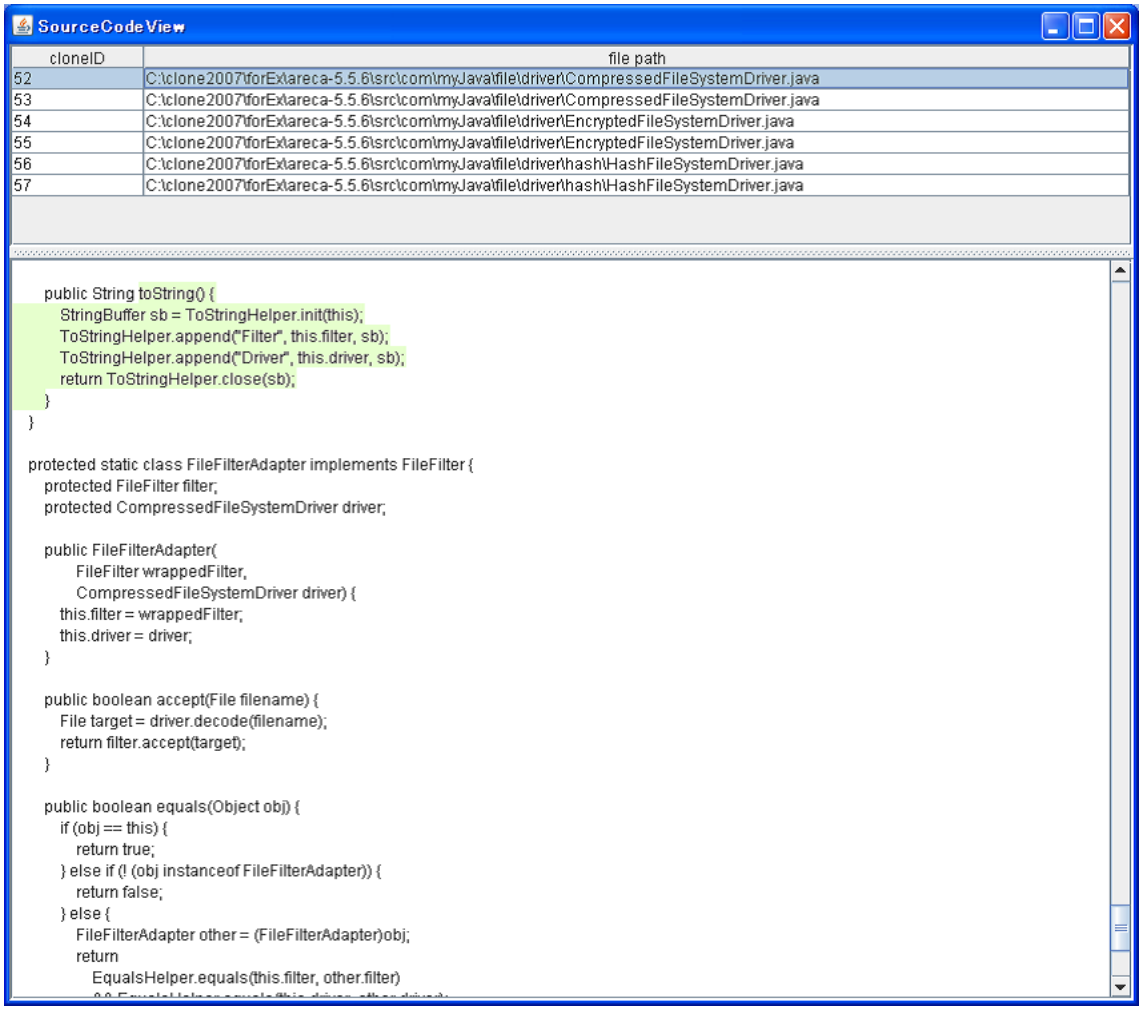

図 9: DCHchecker の SourceCodeView

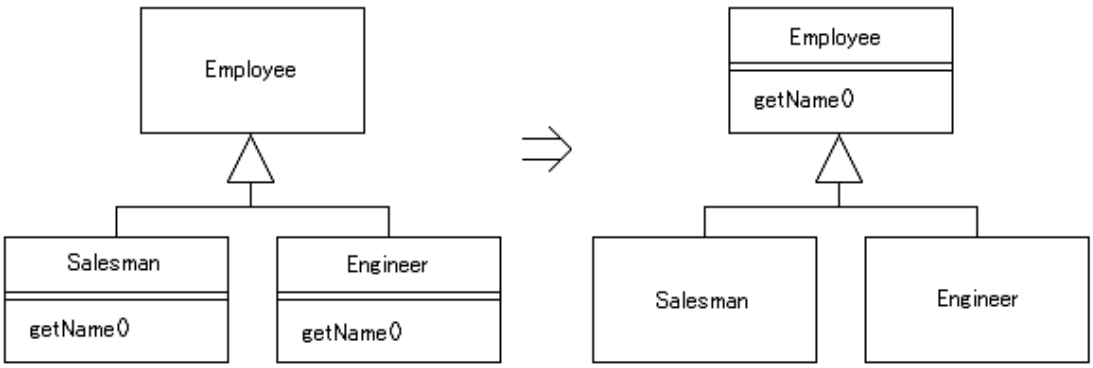

図 10: Pull Up Method

4 実験

DCH を既存の手法で計測した場合と本研究で提案した手法で計測した場合とで，リファクタリングできるコードクローンのコード片の数にどれだけの違いがあるのか調べるため実験を行った．様々なリファクタリングパターンが提案されているが，本研究では ”Pull Up Method ”と ”Extract Method ”の 2 つのパターンのリファクタリングについて実験を行った．

4.1 Pull Up Method

”Pull Up Method ”とは，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 10 のように，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によってリファクタリングを行うことができる．

上記より，”Pull Up Method ”を行うには，対象がメソッドであり，重複したメソッドを持つクラスが共通の親クラスを継承している必要がある．よって，”Pull Up Method ”を用いてリファクタリングを行うための条件は以下のようになる．

1. 対象となる単位はメソッド本体
2. DCH の値が 1 以上

4.2 Extract Method

”Extract Method ”とは，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 11 のように，抽出箇所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ること

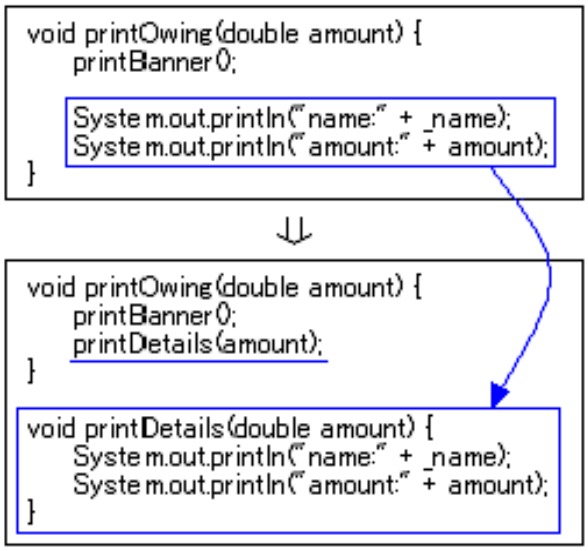

図 11: Extract Method

によって，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．このとき，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．

なお，コード片を新たなメソッドとして再定義することになるため，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．もし抽出部分の内部において外部定義変数を参照している場合は，その変数を引数として与えなければならない．また，抽出部分の内部において外部定義変数に対して代入を行っている場合は，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させる必要がある．抽出部分における外部定義変数への代入が 1 つの場合は単に return 文を用いて返すだけでよいが，複数あった場合はリファクタリングを容易には行えない．

そこで，文献 [3] にて定義されているメトリクス NSV(S) を用いる．クローンセット S は n 個のコード片 f_1，f_2，…，f_n を含んでおり，コード片 f_i では t_i 個の外部定義の変数に対して代入を行っているとする．このとき，メトリクス NSV(S) は次の式で表される．

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

直観的には，クローンセット S に含まれる各コード片内で代入が行われている外部定義変数の平均数を表している．

”Extract Method ”はメソッド内のコード片に対して適用されるので，対象は文単位である．また，全てのコード片が同一クラス内に存在し，且つ，コード片内における外部定義変数への代入が 1 回以下の場合，容易に ”Extract Method ”によるリファクタリングを行

表 1: 実験対象のプログラム

	ファイル数	行数
Apache Ant(v1.6.0)	650	189,915
Areca backup(v5.5.6)	323	55,179
Eclipse Checkstyle Plug-in(v4.4.0)	206	43,349
JasperReports-Java Reporting(v2.0.4)	948	211,619
jEdit(v4.2)	394	140,665
JFreeChart(v1.0.9)	538	194,470
Jimm-Mobile Messaging(v0.5.1)	56	27,166
Robocode(v1.5.2)	209	48,717
XUI RIA Framework(v1.0.4)	439	76,770
ZK-Simply Ajax and Mobile(v3.0.2)	961	132,597

うことができる．よって，”Extract Method ”を用いてリファクタリングを行うための条件を以下に設定する．

1. 対象となる単位は文単位
2. DCH の値が 0
3. NSV(S) の値が 1 以下

4.3 実験対象

本実験では，Apache Ant(v1.6.0) と，Sourceforge にて公開されているオープンソースの Java 言語で記述されたソフトウェア 9 種を対象とした．使用したプログラムのプログラム名とソースコードの規模を表 1 に示す．

4.4 実験方法

実験には，リファクタリング支援環境 Aries と本研究で作成したツール DCHchecker を用いた．なお，Aries は 1 つのクローンセットにつき，1 つの DCH の値を計測する．CCFinder の最小一致トークン数は 30 トークンと設定した．また，リファクタリングが行いやすいコードクローンを抽出するため，トークンの正規化に関するオプションではメンバ変数の正規化とリテラルの正規化のみを選択した．

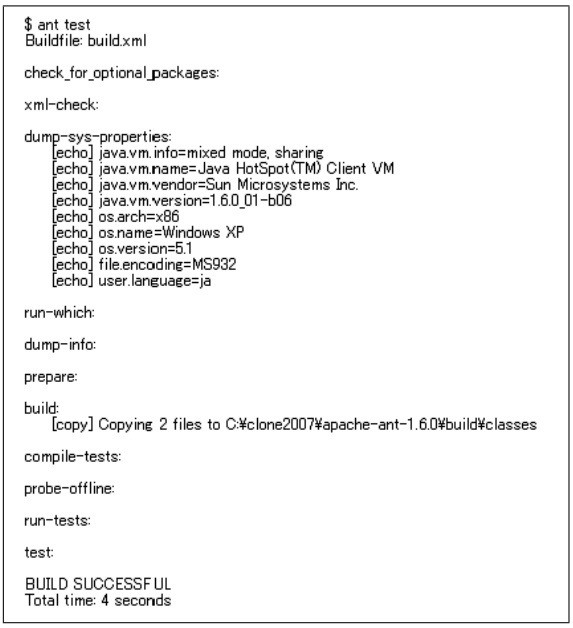

図 12: 通常の Ant

図 13: リファクタリング後の Ant

まず，実験対象のプログラムから 4.1 節と 4.2 節で述べた，”Pull Up Method ”によるリファクタリングの条件と ”Extract Method ”によるリファクタリングの条件に合うクローンセットを Aries でそれぞれ絞り込み，そのコード片数を計測する．次に，DCHchecker でも同様に，条件に合うコードクローンのコード片数を計測することによって，新たにどの程度の数のコード片をリファクタリングできるようになったかを調べる．

Apache Ant については DCHchecker で新たに発見されたコードクローンに対して実際にリファクタリングを行い，Ant のテストケースを用いてその動作が変わらないことを確認した．テストケースを実行した際の出力結果を図 12，図 13 に示す．

なお，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．

4.5 実験結果

実験結果を表 2 に示す．実験対象のプログラムを Aries と DCHchecker 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 DCHchecker で計測したコード片数から Aries で計測したコード片数を引いた値を示している．

表 2 より，Aries で検出された ”Pull Up Method ”，”Extract Method ”の条件に合うコードクローンの合計がそれぞれ 766 個，787 個だった．それに対し，DCHchecker を用いた場合

表 2: リファクタリング可能なコード片の数

	Pull Up Method			Extract Method		
	Aries	DCH checker	差分	Aries	DCH checker	差分
Apache Ant(v1.6.0)	65	68	3	68	72	4
Areca backup(v5.5.6)	37	48	11	30	44	14
Eclipse Checkstyle Plug-in(v4.4.0)	9	9	0	20	24	4
JasperReports-Java Reporting(v2.0.4)	292	309	17	143	205	62
jEdit(v4.2)	3	8	5	153	155	2
JFreeChart(v1.0.9)	262	307	45	176	275	99
Jimm-Mobile Messaging(v0.5.1)	0	0	0	28	28	0
Robocode(v1.5.2)	17	17	0	60	60	0
XUI RIA Framework(v1.0.4)	34	34	0	43	47	4
ZK-Simply Ajax and Mobile(v3.0.2)	47	47	0	66	72	6
合計	766	847	81	787	982	195

は，”Pull Up Method ”では Aries を用いたときよりも 81 個多い 847 個，”Extract Method ”では 195 個多い 982 個検出されている．つまり，DCHchecker を用いた場合は，Aries のみを用いた場合に比べて，”Pull Up Method ”は約 10.6 %，”Extract Method ”は約 24.8 %多くのコードクローンを検出することができた．

5　考察

本節では，本研究で提案した手法がリファクタリングを行う上で有効であるかについて考察していく．

表 2 より，Aries と DCHchecker とで差が出ないものもあるものの，大抵のケースでは DCHchecker を用いて検出したコード片数が Aries を用いて検出したコード片数を上回った．リファクタリング可能なコード片の検出数が多いほど，リファクタリングする箇所や方法の選択肢が広がる．そうすると，より効果的なリファクタリングを選択できるので，本手法はリファクタリングを行う上で有効であるといえる．

対象プログラムの規模と，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．また，Sourceforge にて公開されているオープンソースのソフトウェアについては，開発規模（開発人数，開発期間）についても調査した．その結果を表 4 に示す．ただし，開発期間に示しているのは Sourceforge に各ソフトウェアが登録された日付である．なお，どちらの表も ”Pull Up Method ”と ”Extract Method ”の差分の合計数の少ない順に並び替えてある．

表 3 を見ると，行数の多いものは差分も多い傾向がある．10 万行以上の規模のプログラムについては，Aries と DCHchecker とでコード片 6 個分以上の差が出ている．規模が 10 万行未満のプログラムは，”Areca backup ”のように多くの差が出るものもあれば，”Jimm-Mobile Messaging ”と ”Robocode ”のように差が出ないものもあるなど，まちまちであった．ファイル数と差分には関連性はみられなかった．

表 4 より，開発規模は従来の手法と本手法との差にあまり関係ないと思われる．開発人数にも開発期間にも，差分との関連性は見えてこない．

対象プログラムによって，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．また，対象プログラムの行数が多いほど，差分が大きくなる傾向があることもわかった．よって，行数の多いプログラム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．

表 3: プログラムの規模との比較

	Pull Up Method	Extract Method	ファイル数	行数
Jimm-Mobile Messaging(v0.5.1)	0	0	56	27,166
Robocode(v1.5.2)	0	0	209	48,717
Eclipse Checkstyle Plug-in(v4.4.0)	0	4	206	43,349
XUI RIA Framework(v1.0.4)	0	4	439	76,770
ZK-Simply Ajax and Mobile(v3.0.2)	0	6	961	132,597
Apache Ant(v1.6.0)	3	4	650	189,915
jEdit(v4.2)	5	2	394	140,665
Areca backup(v5.5.6)	11	14	323	55,179
JasperReports-Java Reporting(v2.0.4)	17	62	948	211,619
JFreeChart(v1.0.9)	45	99	538	194,470

表 4: 開発規模との比較

	Pull Up Method	Extract Method	開発人数	開発期間
Jimm-Mobile Messaging(v0.5.1)	0	0	8	2004.01.30
Robocode(v1.5.2)	0	0	3	2001.10.06
Eclipse Checkstyle Plug-in(v4.4.0)	0	4	4	2003.05.03
XUI RIA Framework(v1.0.4)	0	4	21	2003.03.18
ZK-Simply Ajax and Mobile(v3.0.2)	0	6	13	2005.11.09
jEdit(v4.2)	5	2	146	1999.12.06
Areca backup(v5.5.6)	11	14	1	2006.07.03
JasperReports-Java Reporting(v2.0.4)	17	62	10	2001.09.25
JFreeChart(v1.0.9)	45	99	8	2000.11.27

6　あとがき

本研究では，コードクローンのリファクタリングを支援するメトリクス DCH を改良する手法を提案し，その手法を実現するためのツールを実装した．従来の DCH は各クローンセットごとに 1 つの値をとったのに対して，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．これにより，クローンセット内の一部のコード片のみが離れたクラス階層に存在する場合でも，メトリクス DCH を用いたリファクタリング可能なコード片の検出を行えるようになった．

更に，実装したツールを用いて，実際のソフトウェアに対して適用実験を行った．その結果，差が出ないソフトウェアもあったものの，本手法を用いた方が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．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

その他の楠本研究室の皆様のご協力に心より感謝致します.

また，本研究に至るまでに，講義，演習，実験等でお世話になりました情報科学科の諸先生方にこの場を借りて心から御礼申し上げます.

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] 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, Jul 2002.

[3] 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, pp. 186–195, Jul 2005.

[4] 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.

[5] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. *Proc. of International Conference on Software Maintenance 98*, pp. 368–377, Mar 1998.

[6] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. ichiMatsumoto, and H. Kudo. Software analysis by code clones in open source software. *Journal of Computer Information Systems*, Vol. XLV, No. 3, pp. 1–11, Apr 2005.

[7] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K.Inoue. On software maintenance process improvement based on code clone analysis. *Proc. of the 4th International Conference on Product Focused Software Process Improvement*, pp. 185–197, Dec 2002.

[8] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローン解析に基づくリファクタリングの試み. 情報処理学会論文誌, Vol. 45, No. 5, pp. 1357–1366.

[9] M. Balazinska, E. Merlo, M. Dagenais, B.Lag ¨ue, 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, Nov 2000.

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

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

[12] 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, Sep 2003.

[13] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 4, pp. 267–292, Jul 2006.

[14] 川口真司, 松下誠, 井上克郎. 版管理システムを用いたクローン履歴分析手法の提案. 電子情報通信学会論文誌 D, Vol. J89-D, No. 10, pp. 2279–2287, Oct 2006.

[15] M. Kim, V. Sazawal, D. Notkin, and G.C. Murphy. An empirical study of code clone genealogies. *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 187–196, Sep 2005.

[16] C. Kapser and M.W. Godfrey. “ cloning considered harmful ” considered harmful. *Proc. of the 13th Working Conference on Reverse Engineering*, pp. 19–28, Oct 2006.

[17] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. *Proc. of the 6th IEEE International Symposium on Software Metrics*, pp. 292–303, Nov 1999.

[18] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 173–180, Sep 2004.

[19] E. Duala-Ekoko and M.P. Robillard. Tracking code clones in evolving software. *Proc. of the 29th International Conference on Software Engineering*, pp. 158–167, May 2007.

[20] 泉田聡介, 植田泰士, 神谷年洋, 楠本真二, 井上克郎. ソフトウェア保守のための類似コード検索ツール. 電子情報通信学会論文誌, Vol. 86-D-I, No. 12, pp. 906–908, Dec 2003.

[21] 佐々木亨, 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. プログラム変更支援を目的としたコードクローン情報付加ツールの実装と評価. 電子情報通信学会論文誌, Vol. 87-D-I, No. 9, pp. 868–870, Sep 2004.

[22] Z.A. Mann. Three public enemies: Cut, copy, and paste. *IEEE Computer*, Vol. 39, No. 7, pp. 31–35, Jul 2006.

[23] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. *Proc. of 2004 International Symposium on Empirical Software Engineering*, pp. 83–92, Aug 2004.

[24] M. Balint, T. Girba, and R. Marinescu. How developers copy. *Proc. of the 14th IEEE International Conference on Program Conprehension*, pp. 56–68, Jun 2006.