修士学位論文

題目

コードクローンに含まれるメソッド呼び出しの変更度合の分析

指導教員
井上 克郎 教授

報告者
工藤 良介

平成 25 年 2 月 5 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 24 年度 修士学位論文

コードクローンに含まれるメソッド呼び出しの変更度合の分析

工藤 良介

内容梗概

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，主にコピーアンドペーストによって生成される．コードクローンとなっているコード片の 1 つに修正すべき欠陥が発見された場合，コードクローンとなっている残りのコードすべてについて修正の必要を検討しなければならない．このことから，多くの開発者にとってコードクローンは「長期的に，保守性を悪化させる可能性がある」と認識されている．このため，ソースコード中からコードクローンを検出するツールの開発や，コードクローンを解消するための手法に関する研究が数多く行われている．しかし，コードクローンとなっているコードが持つ特徴についての研究はあまり行われておらず，どのような理由でコピーアンドペーストが行われ，どのようなコードが複製されているのかは，はっきりとわかっていない．

本研究では，コードクローンが持つ特徴を明らかにすることを目的として，コードクローンに関するメトリクスの分析を行う．本研究では，コードクローンの出現位置やコードクローンに含まれる識別子の数といった既存の特徴に加えて，コードクローンとなっているコード片の特徴を知るために，コードクローンに含まれる「重要なメソッド呼び出し」を調査する.「重要なメソッド呼び出し」とは，メソッドの主要な処理を表現していると思われるメソッド呼び出しのことである．

具体的な調査手順は次の通りである．まずは Java で書かれたオープンソースソフトウェアを収集し，コードクローン検出ツール CCFinder を用いてコードクローンのデータを取得した．次に得られたコードクローンのデータから，識別子の変更度合,「重要なメソッド呼び出し」の数などのメトリクスを計算した．そして，クローンのメトリクス値に対して統計処理を行い，分析した．

分析の結果,「重要なメソッド呼び出し」は「重要でないメソッド呼び出し」に比べて名前の修正が加えられにくいことがわかった．このことから，コードがコピーされ修正が加えられても，そのコードが行う主要な処理に変更が加えられることは少ないと考えられる．また，コードクローンになっている部分となっていない部分で「重要なメソッド呼び出し」の割合

を調べた時，両者に有意な差は見られなかった．このことから，コードの中で特に「重要なメソッド呼び出し」を多く含む部分がコピーされているわけではないと考えられる.「重要なメソッド呼び出し」が変更されていないコードクローンの多くは，同一パッケージ内部，あるいは類似した名前のクラスなどに出現しており，特定の処理をクラス間，メソッド間で共有する方法がないために作られているのだと考えられる．一方,「重要なメソッド呼び出し」の変更度合が高いコードクローンは同一の制御構造の繰り返しが多く，制御構造の再利用であると考えられる.「重要なメソッド呼び出し」が変更されていないコードクローンは全体の 87%を占めていることから，コードクローンの多くがメソッド呼び出しによって実現されている処理の再利用であると考えられる.

主な用語

コードクローン
重要なメソッド呼び出し
識別子の変更度合

目 次

1　はじめに

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る．コードクローンは主にコピーアンドペーストによって生成される．互いにクローンとなっているコード片の集合（クローンセット）に含まれる，あるコード片にバグが存在した場合，そのクローンセットに含まれる他のコード片すべてについて不具合の有無を判定し，修正しなくてはならない [19]．このことから，多くの開発者にとってコードクローンは「長期的に，保守性を悪化させる可能性がある」と認識されている．このため，ソースコード中からコードクローンを検出するツールの開発や，コードクローンを解消するための手法に関する研究が数多く行われている [8, 16]．しかし，コードクローンとなっているコードが持つ特徴についての研究はあまり行われておらず，どのような理由でコピーアンドペーストが行われ，どのようなコードが複製されているのかは，はっきりとわかっていない．コピーされるコードの特徴や，コピーされる理由が明らかになれば，開発者が目的のコードクローンを検出する支援となることが期待される．

本研究では，コードクローンが持つ特徴を明らかにすることを目的として，コードクローンに関するメトリクスの分析を行う．メトリクスには，メソッド呼び出しや変数などの識別子の数，対応する識別子の名前が変更される割合などが含まれるが，本研究では，これらの既存の特徴に加えて，コードクローンとなっているコード片の特徴を知るために，コードクローンに含まれる「重要なメソッド呼び出し」を調査する.「重要なメソッド呼び出し」とは，メソッドの主要な処理を表現していると思われるメソッド呼び出しのことで,「メソッドの最後のメソッド呼び出し」,「戻り値がない，あるいは戻り値を利用していないメソッド呼び出し」,「メソッドと同じ action を行なっているメソッド呼び出し」などがこれに該当する [22]．コードクローンに含まれる「重要なメソッド呼び出し」の情報から，コードクローンがメソッド内の主要な処理を意図的にコピーしたものであるかを判定する．

具体的な調査手順は次の通りである．まずは Java で書かれたオープンソースソフトウェアを収集し，コードクローン検出ツール CCFinder を用いてコードクローンのデータを取得した．次に得られたコードクローンのデータから各メトリクスを計算した．そして，クローンのメトリクス値に対して統計処理を行い，分析した．

分析の結果,「重要なメソッド呼び出し」は,「重要でないメソッド呼び出し」に比べて名前の修正が加えられにくいことがわかった．このことから，コードがコピーされ修正が加えられても，そのコードが行う主要な処理に変更が加えられることは少ないと考えられる．また，コードクローンになっている部分と，コードクローンを含んでいるメソッド全体で「重要メソッド呼び出しの割合」を調べた時，両者に有意な差は見られなかった．このことから，コードの中で特に「重要なメソッド呼び出し」を多く含む部分がコピーされているわけ

ではないと考えられる．呼び出されているメソッドの名前について分析を行った結果，名前に write や remove を含むメソッドは「重要なメソッド呼び出し」によって呼び出されることが多く，名前に to を含むメソッドは「重要でないメソッド呼び出し」によって呼び出されることが多いことがわかった.「重要なメソッド呼び出し」が変更されていないコードクローンの多くは，同一パッケージ内部，あるいは類似した名前のクラスなどに出現しており，特定の処理をクラス間，メソッド間で共有する方法がないために作られているのだと考えられる．一方,「重要なメソッド呼び出し」の変更度合が高いコードクローンは同一の制御構造の繰り返しが多く，制御構造の再利用であると考えられる.「重要なメソッド呼び出し」が変更されていないコードクローンは全体の 87%を占めていることから，コードクローンの多くがメソッド呼び出しによって実現されている処理の再利用であると考えられる．

以下 2 章では本研究の背景について述べる．3 章では調査方法について述べる．4 章では調査の結果を示しそれに対する考察を述べる．最後に 5 章では本研究のまとめと今後の課題について述べる．

2　背景

本研究の背景として，分析の対象となるコードクローン及び，コードクローンを検出するツールである CCFinder について説明する．

2.1　コードクローン

コードクローンとは，あるプログラムにおいて，互いに類似したコード片の組あるいはコード片の集合のことである．ソースコードがある程度以上の長さで偶然一致する確率は低く [15]，多くの場合，コードクローンは開発者がソースコードを意図的に複製して再利用することによって生じる [18]．ソースコードの再利用においては，複製したコード片をそのままの形で複製先で利用できるとは限らないため，再利用先の環境に合わせて，複製したコード片を適切に変更する必要がある．そのため，コードクローンとなっているコード片同士は，互いに完全一致するとは限らず，何らかの差異を含むことがある．Bellon らは，この差異の種類に着目してコードクローンを以下の 3 つのタイプに分類している [2]．

タイプ 1 空白やタブの有無などのコーディングスタイルを除き，完全一致のコードクローン

タイプ 2 変数名やメソッド名などのユーザ定義名，また型名などの一部の予約語のみが異なるコードクローン

タイプ 3 タイプ 2 の変更に加え，文の挿入や削除などが行われたコードクローン

2.1.1　コードクローン検出の目的

開発者にとって有用なコードクローンは，その検出の目的によって変わってくる．その目的の例を示す．

- バグのあったコード片に類似したコード片を検出する．[20]
- リファクタリングが容易なコードクローンを検出する．[5]
- 一貫性のない類似コードをバグ候補として検出する．[10, 13]
- 一貫性のあるコードをまとめて変更する．[3, 11, 12]
- 横断的関心事の実装を検出する． [4]

2.1.2 コードクローン検出ツール CCFinder

コードクローンを自動的に検出するためのツールは，数多く提案されている [1, 7, 21]．コードクローンの検出とはソースコードの比較であり，行，字句解析におけるトークン，抽象構文木，プログラム依存グラフなど，様々な比較の基準が用いられている．これらの検出ツールのうち，実用的なものとしては，トークン単位でコードクローンを検出する CCFinder がある．CCFinder は，与えられたソースファイル集合からコードクローンを検出して，コードクローンの位置情報を出力する．また，CCFinder は 100 万行規模の大規模なソースコードに対して，数分から数時間でコードクローンを検出できるほか，C/C++，Java，COBOL，Fortran など多言語に対応していることが特徴である．CCFinder によって検出されるクローンは，ソースコード中に登場する識別子名やリテラルを「パラメータ化」することでその違いを吸収したクローン（Parameterized Clone）であり，タイプ 1 及びタイプ 2 のクローンがこれに該当する．

2.2 関連研究

Sridhara ら [22] は,「メソッドの概要を説明するコメントの内容としてふさわしい」プログラム文を抽出し，それをコメントに変換する技術を提案した．3 章で述べる「重要なメソッド呼び出し」を決定する手順は，このプログラム文を抽出する手法をもとにしている．Sridhara は，自動で生成されたコメントがメソッドの要約として適切であるかどうかについて Java 経験者にアンケートを行い，それが十分適切であると評価された，と述べている．コメントの内容がメソッド呼び出しを元に作られていることから，この手法に定義されたプログラム文の選択基準が，メソッド呼び出しの選択として妥当であると考えた．

3　調査方法

本研究では，実用的なコードクローン検出ツールである CCFinder から得られるコードクローンについてメトリクスを計算し，分析する．これによってコードクローンとなっているコードが持っている特徴を明らかにする．

3.1　CCFinder

今回使用した CCFinder のバージョンは 7.2.4.0 である．また，30 トークン以上の長さのコードクローンを検出対象とした．これは CCFinder のデフォルト値である．

3.2　調査対象

調査対象としてプログラミング言語 Java で記述された 6 つのオープンソースソフトウェアを選択した．6 つのソフトウェアについて簡潔に説明する．

Derby Java で実装された関係データベース管理システムのソフトウェア．Engine，Network Server，Network Client およびツール群から構成されるが，本研究では Engine 部分を実装しているソースコードを対象とした．[6]

h2 Java プラットフォーム上で動く，ACID リレーショナルデータベース．[9]

jtunes Java で書かれた itunes の代替ツール．[14]

Tomcat Java Servlet や Java Server Pages（JSP）を実行するためのサーブレットコンテナ．[23]

XXL 高度な問い合わせ処理機能と，それを実装するための豊富なインフラストラクチャを含む Java ライブラリ．[24]

zk Java で書かれた AjaxWeb アプリケーションフレームワーク．[25]

CCFinder でこれらのソフトウェアからコードクローンを検出した後，RNR の値が 0.5 以下となるクローンセットを調査対象から除外している．RNR とはコードの非繰り返し率のことであり，コードに同じ並びの繰り返しが多いと，この値が低くなる．RNR の値が低いコードは変数の並びや if 文などの制御文の繰り返しになっていることが多い．このようなコードクローンは開発者にとっては興味がないものであることが経験的にわかっているため対象から除外した．6 つのソフトウェアから検出されたクローンセットの数，RNR の値によるフィルタリング後のクローンセットの数について表 1 に示す．

3.3 コードクローン情報に関する前提条件

本研究では，CCFinder が出力するクローンを調査する．CCFinder が検出するクローンはタイプ 1 もしくはタイプ 2 のクローンであるが，どちらのタイプであるかは判定されていない．以下に CCFinder から得られる情報についての前提条件を示す．CCFinder が検出するものと同種のクローン，すなわちタイプ 1 およびタイプ 2 のクローンを検出するツールであり，かつ，下記の前提条件を満たすことができるツールであれば，同等の調査が可能である．

- コードクローン情報は，クローンセット (Clone Set) と呼ばれる単位で出力される．各クローンセットは，互いに類似しているコード片の集合である．
- クローンセットに含まれる各コード片はソースコード上で連続である．各コード片は，ファイル名，開始行番号，開始桁位置，終了行番号，終了桁位置の組によって指定される．
- クローンとなっているコード片は,「パラメータ化」という正規化操作を適用すると，同一の長さのトークン列となる．ここでのパラメータ化とは，Java などのプログラミング言語の文法において，識別子（Identifier）およびリテラル（Literal），すなわち，開発者が自由に定義することができる変数名や型名，値の出現に対して，それらをすべて無名のトークンに置換する操作を意味する．たとえば，２つの代入文 i = i + 1 ; と z = x + y ; が与えられたとすると，これらをパラメータ化した結果は，同一の式 $p = $p + $p ; となる．つまり，これら２つの代入文は，タイプ 2 のクローンであるといえる．

表 1: 各ソフトウェアから検出されたクローンセット数

ソフトウェア名	総クローンセット数	フィルタリング後残ったクローンセット数
Derby	2302	1384
h2	1315	569
jtunes	1324	675
Tomcat	3518	1962
XXL	832	509
zk	338	229

3.4 識別子の変更

CCFinder によって検出されるタイプ 1 及びタイプ 2 のクローンはコードごとのトークンの数が等しく，またトークンの種類の並びも一致することが保証されている．よってコード間で「同じ位置に存在する識別子」を決定することができる．この「同じ位置に存在する識別子」の名前を比べた時に，クローンセット内のすべてのコードで名前が一致する場合この識別子を「名前が変更されていない識別子」，それ以外の場合は「名前が変更された識別子」と定義する．また，メソッド呼び出しや変数などについても同様に「同じ位置に存在するメソッド呼び出し」の名前を比べた時に，クローンセット内のすべてのコードで名前が一致する場合このメソッド呼び出しを「名前が変更されていないメソッド呼び出し」，それ以外の場合は「名前が変更されたメソッド呼び出し」と定義する．

3.5 識別子の変更度合

コードクローンとなっているコード間で識別子の名前を比較し,「名前が変更された識別子」の数を「すべての識別子」の数で割った値を「識別子の変更度合」と定義する．また，メソッド呼び出しや変数などについても同様に「名前が変更されたメソッド呼び出し」の数を「すべてのメソッド呼び出し」の数で割った値を「メソッド呼び出しの変更度合」というように定義する．

たとえば，図 1 の 2 つのコード片がクローンとして検出された場合を考える．図 1 中のコード 1，コード 2 について，これらのコードに含まれる識別子についてまとめると表 2 になる．表 2 から，識別子の数は 9，メソッド呼び出しの数は 2，変数の数は 6 となり,「名前が変更された識別子」の数は 3,「名前が変更されたメソッド呼び出し」の数は 1,「名前が変更された変数」の数は 2 となるので,「識別子の変更度合」は 33%,「メソッド呼び出しの変更度合」は 50%,「変数の変更度合」は 33%となる．

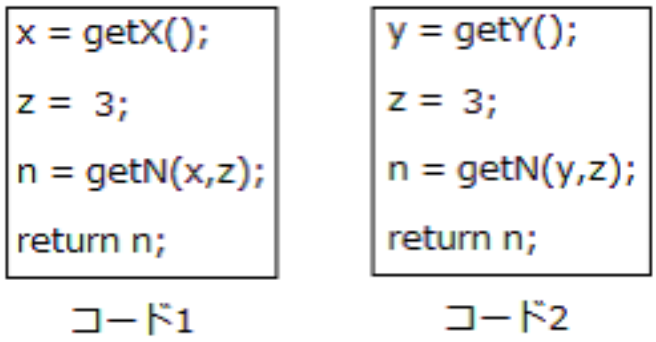


図 1: コードクローン

表 2: コードクローンに含まれる識別子

コード 1	コード 2	種類	変更の有無
x	y	変数	あり
getX	getY	メソッド呼び出し	あり
z	z	変数	なし
3	3	リテラル	なし
n	n	変数	なし
getN	getN	メソッド呼び出し	なし
x	y	変数	あり
z	z	変数	なし
n	n	変数	なし

3.6 重要なメソッド呼び出しの分析

Sridhara ら [22] は,「メソッドの概要を説明するコメントの内容としてふさわしい」プログラム文を抽出し，それをコメントに変換する技術を提案した．プログラム文には様々なものがあるが，一般的な代入文についてはコメントには反映されることはなく，メソッド呼び出しを主な情報源として用いている．本研究では，コードクローンに含まれるメソッド呼び出しのうち，それが出現しているメソッドの概要を説明するような呼び出しを「重要なメソッド呼び出し」とする.「重要なメソッド呼び出し」と判定する基準は次の通りである．

- メソッドの末尾に出現するプログラム文．多くのメソッドでは，何らかの準備を行ってから実行する命令のほうが重要である．
- 戻り値が使われていないか，戻り値の型が void であるメソッド呼び出し文は，副作用を持つメソッドを呼び出しているため重要である．
- メソッドと同じ意味の操作をするプログラム文．たとえば compile というメソッド中で compileRegex というような同一の動詞を持つメソッド呼び出しは，実装の詳細を表現しているため重要である．

上記の基準だけでは，たとえば実行ログを記録するためのメソッドなどをはじめとする，本来の機能ではない横断的関心事 [17] も重要なメソッドとして選定されてしまう．そのため，[22] では，上記に加えていくつかの例外ルールが定義されている．本研究では [22] の記述に従って，条件を満たすようなメソッド呼び出しを抽出する処理を実装した．[22] では Java の匿名内部クラスなどの扱いが定義されていなかったため，本研究では独自に判定基準を作成した.「重要なメソッド呼び出し」の判定処理は，Java のバイトコードを解析することで行った．

3.6.1 バイトコードにおけるプログラム文の認識

解析はメソッド単位で行う．メソッドの中に宣言された内部クラスやそのメソッドは，独立したクラス，メソッドに所属しているものとして扱い，内部クラスのメソッドについてもそれぞれ解析を行う．コードクローンが内部クラスの定義を含んでいるときは，外側のメソッドにおける重要なメソッド呼び出しと，内部クラスのメソッド定義における重要なメソッド呼び出しを両方とも取り扱うことになるが，重要なメソッド呼び出しの計算そのものは各メソッドについて独立に行う．本研究で取り扱うプログラム文は， バイトコード上で「Java 仮想マシンのオペランドスタックを使ってデータの授受を行う命令群」として認識する．Java 仮想マシンはスタックマシンとして定義されており，１つの式に登場する中間結果の値は一時的にスタックに保存される．例として，以下にメソッドを呼び出す式文を示す．

stream.write(buffer.toByteArray());

この式文は，２つのメソッド呼び出しを持っており，toByteArray というメソッド呼び出しの戻り値を write メソッドの引数として用いている． stream 変数が OutputStream 型の変数，buffer が ByteArrayOutputStream 型の変数であるとき，上記の式文は次のような 4 命令からなるバイトコードにコンパイルされる．実際のバイトコードでは各変数やメソッドの名前は数値で表現されるが，ここでは説明のため，名前で表記する．

1: ALOAD stream
2: ALOAD buffer
3: INVOKEVIRTUAL ByteArrayOutputStream.toByteArray()[B
4: INVOKEVIRTUAL OutputStream.write([B)V

このバイトコードは，次のように実行される．

1. ALOAD 命令によって stream 変数に格納されたオブジェクト参照を読み込み，オペランドスタックに乗せる．

2. ALOAD 命令によって buffer 変数に格納されたオブジェクト参照を読み込み，オペランドスタックに乗せる．

3. INVOKEVIRTUAL 命令は，オペランドスタックの一番上にあるオブジェクト参照を使って，ByteArrayOutputStream クラスに定義された toByteArray メソッドを呼び出す．命令末尾の “ [B ” というのは，戻り値が byte 型の配列であることを表現しており，戻り値がオペランドスタックに乗せられる．

4. 最後の INVOKEVIRTUAL 命令は，引数に byte 型の配列を取ることが命令に表現されているので，スタックの一番上をその引数として認識し，その下にある（1 番目の命令でスタックに乗せられた）オブジェクト参照に対して write メソッドを呼び出す．戻り値は V すなわち void 型であるので，write 命令の実行後，オペランドスタックは空になる．

このように，１つの文の実行に必要な変数の値やメソッドの戻り値がオペランドスタックに配置される．一方，文と文の間では，ソースコードに記載されているようにローカル変数などを用いてデータのやり取りを行うため，オペランドスタックは使われない．そのため，オペランドスタックを用いてデータの授受を行っている範囲を１つの文として認識することが

できる．また，オペランドスタックは，仮想マシンのモデルからスタックと呼ばれてはいるが，どの命令がスタックの何番目のデータを読み書きするかが静的に決定されている．そのため，通常の変数に対するデータ依存関係と同様の手法でスタック上のデータ依存関係を抽出することができる．

3.6.2 重要なメソッド呼び出しの認識

バイトコード解析を用いて，Sridhara の論文 [22] で定義された重要なプログラム文を，次のような方法で認識する．

- メソッドの末尾に出現するプログラム文は，戻り値を持たない RETURN 命令の直前に配置された命令か，または戻り値を返す ARETURN, IRETURN, LRETURN, FRETURN, DRETURN 命令となる．いずれの場合も，その命令の実行に使われているデータを提供しているすべてのバイトコード命令を取り出し，その中に含まれるメソッド呼び出しを取り出す．これらのメソッドの集合を，以降の処理では ending と呼ぶ．

- 戻り値の型が void であるメソッド呼び出し文は，INVOKEVIRTUAL をはじめとする INVOKE 命令群のシグネチャから認識できる．また，戻り値が使われていない呼び出しについても，スタックに積まれた戻り値を破棄する命令によって認識することができる．以降の処理では，これらのメソッドの集合を void-return と呼ぶ．

- メソッドと同じ意味の操作をするプログラム文は，INVOKE 命令群が持つ呼び出し先のメソッド名と，解析対象となっているメソッドのメソッド名について，名前を CamelCase のルールに従って単語単位に分割し，先頭単語が一致することによって判定している．これらのメソッドの集合を，以降では same-action と呼ぶ．

これら ending，void-return，same-action を基本集合として，次のような加工を行っていく．

- void-return と same-action メソッド集合から，コンストラクタと，名前が get/set から始まるメソッド，文字列の加算に使用される StringBuilder クラスおよび StringBuffer クラスの append メソッドを除外する．これらは計算の主要な処理である（すなわち ending に含まれている）場合以外は，重要とはみなさない．

- 例外ハンドラの中に記述されている命令は，例外処理であって重要なメソッド呼び出しではないと考えられるので，それらを取り除く．この処理は，例外が発生しないと仮定した（通常の命令による制御の流れだけを取り出した）制御フローグラフによって，メソッドの先頭の命令から到達可能な範囲のメソッド呼び出しだけを残し，他を集合から除外することで実現した．

- メソッドの中では，if (x != null) というように，あるオブジェクトが存在するときだけ処理を実行することがしばしばある．そこで，ある変数が null であること，または null でないことが実行条件となるようなメソッド呼び出しを除外する．この処理は，IFNULL 命令および IFNONNULL 命令による条件分岐によってのみ到達できる範囲のメソッド呼び出しを除外することで実現した．

- ロギングや例外処理に関連すると思われるメソッドを除外する．すなわち，メソッド名に log, trace, error, debug, exception, close が含まれるとき，そのメソッド呼び出しを集合から取り除く．

- 各メソッド呼び出しに対してデータを供給するメソッド呼び出しを取り出す．ending, void-return, same-action の各メソッド呼び出し処理で使用しているローカル変数に関するデータ依存関係をただ 1 度だけ辿り，その変数に値を代入するプログラム文を認識し，そこに含まれるメソッド呼び出しを抽出する．ただし，void-return および same-action に対して行ったのと同様に，コンストラクタ，get/set から始まるメソッド，文字列の結合に関するメソッド呼び出しは，この候補から除外する．残ったメソッド呼び出しを data-facilitating と呼ぶ．

- ending, void-return, same-action, data-facilitating の各メソッド呼び出しの条件分岐にメソッド呼び出しが使われていれば，それを重要メソッドに含める．具体的には，制御依存関係解析を用いて，各メソッドの実行条件となるすべての（推移的な制御依存関係を持つ）条件分岐命令を取得し，それらの条件分岐の値に使われたメソッド呼び出し命令を取り出す．ここでも，get/set 等のメソッド呼び出しは候補から除外する．残ったメソッド呼び出しの集合を controlling と呼ぶ．

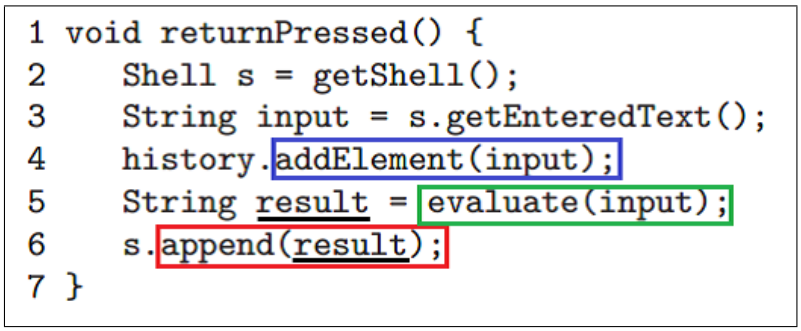
この一連の処理を経て求められた，ending, void-return, same-action, data-facilitating, controlling の 5 つのメソッド呼び出し集合の和集合が,「重要なメソッド呼び出し」となる．

3.6.3 重要なメソッド呼び出しの例

コード例を用いて，重要なメソッド呼び出しについて説明する．

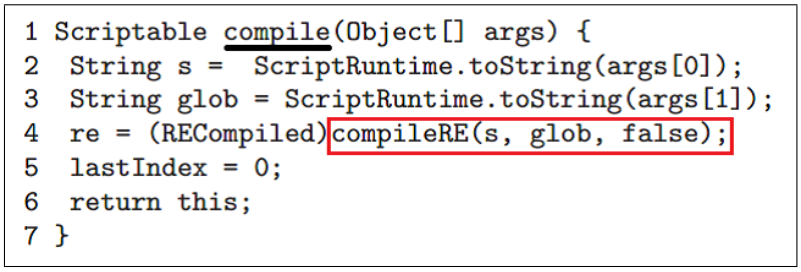
ending 図 2 の 6 行目の append(result) は returnPressed メソッドの末尾にあたるため ending の「重要なメソッド呼び出し」となる．

void-return 図 2 の 4 行目の addElement(imput) は戻り値が利用されていないため void-return の「重要なメソッド呼び出し」となる．



```
1 void returnPressed() {
2    Shell s = getShell();
3    String input = s.getEnteredText();
4    history.addElement(input);
5    String result = evaluate(input);
6    s.append(result);
7 }
```

図 2: ending，void-return，data-facilitating のコード例 [22]



```
1 Scriptable compile(Object[] args) {
2  String s =  ScriptRuntime.toString(args[0]);
3  String glob = ScriptRuntime.toString(args[1]);
4  re = (RECompiled)compileRE(s, glob, false);
5  lastIndex = 0;
6  return this;
7 }
```

図 3: same-action のコード例 [22]

data-facilitating 図 2 の 6 行目の append(result) は ending の「重要なメソッド呼び出し」である．このメソッド呼び出しの引数に使われている変数 result に関するデータ依存関係を辿ると，5 行目の evaluate(input) メソッドの戻り値が result に代入されている．このため，evaluate(input) は data-facilitating の「重要なメソッド呼び出し」となる．

same-action 図 3 の 4 行目の compileRE(s,glob,false) は compile メソッドの中に存在している．所属しているメソッドと同じ compile という action を行うため，compileRE(s,glob,false) は same-action の「重要なメソッド呼び出し」となる．

controlling 図 4 の 5 行目の delete() は ending の「重要なメソッド呼び出し」である．4 行目の equals(”interrupt”) は delete() の実行条件となる条件分岐に使用されているため，controlling の「重要なメソッド呼び出し」となる．

3.7 調査するメトリクス

CCFinder によって得られたコードクローンについてメトリクスを計算する．用いるメトリクスについて説明する．

```
1 void actionPerformed(ActionEvent e) {
2   String cmd = e.getActionCommand();
3   if (cmd != null) {
4      if (cmd.equals("interrupt"))
5         exportFile.delete();
6   }
7 }
```

図 4: controlling のコード例 [22]

3.7.1 識別子情報のメトリクス

コードクローンに含まれる識別子について以下のメトリクスを計算する．

- コードに含まれる識別子の数（*Id*），コードに含まれる「名前が変更された識別子」の数（*ChangedId*），コードに含まれる「名前が変更されていない識別子」の数（*UnchangedId*），コードに含まれる「識別子の変更度合」（*ChangedIdRatio*），の 4 つの値を計算する．これらはすべてクローンセット内の 1 コードあたりの数を計算する．さらに，識別子を変数（*Valuable*），メソッド（*Method*），型（*Type*）に分類し，それぞれについて上の 4 つの値を計算する（*ChangedValue*，*UnchangedValue*，*UnchangedValueRatio* など）．

- クローンセットに含まれる「重要なメソッド呼び出し」の数（*ImportantCallSiteInCloneSet*）を計算する．クローンセットに含まれるコード間で「同じ位置にあるメソッド呼び出し」が，あるコードでは「重要なメソッド呼び出し」であるが，別のコードでは「重要なメソッド呼び出し」ではない，という可能性が存在する．例えば，compileRegex というメソッドの呼び出しは，compile メソッドに含まれれば same-action の「重要なメソッド呼び出し」となるが，compile などの文字列を含まないメソッドの中へとコピーされると「重要なメソッド呼び出し」ではなくなる．そのため，クローンセット内のある 1 コードに含まれる「重要なメソッド呼び出し」の数（*ImportantCallSite*）を調べただけでは不十分であるので，クローンセット内のすべてのコードに含まれる「重要なメソッド呼び出しの総数」を計算することとした．クローンセット内の i 番目のコードに含まれる「重要なメソッド呼び出し」の数を *ImportantCallSite(i)*，クローンセットに含まれるコードの数を n とすると

$$ImportantCallSiteInCloneSet = \sum_{i=1}^{n} ImportantCallSite(i)$$

となる．

- クローンセットに含まれるメソッド呼び出しの数（*CallSiteInCloneSet*）を計算する．後に「重要なメソッド呼び出し」の割合を計算するため,「重要なメソッド呼び出し」と同様に，クローンセット内のすべてのコードに含まれるメソッド呼び出し（*CallSite*）の総数を計算する．クローンセット内の　i番目のコードに含まれるメソッド呼び出しの数を *CallSite(i)*，クローンセットに含まれるコードの数を n とすると

$$CallSiteInCloneSet = \sum_{i=1}^{n} CallSite(i)$$

となる．

- メソッド呼び出しのうち,「重要なメソッド呼び出し」であるものの割合（*ImportantCallSiteRatio*）を計算する.「重要なメソッド呼び出しの総数」を「メソッド呼び出しの総数」で割ることで計算される．

$$ImportantCallSiteRatio = \frac{ImportantCallSiteInCloneSet}{CallSiteInCloneSet}$$

- クローンセットに含まれるコードが存在しているメソッド全体に含まれるメソッド呼び出しの数（*CallSiteInMethodInCloneSet*），及び「重要なメソッド呼び出し」の数（*ImportantCallSiteInMethodInCloneSet*），割合（*ImportantCallSiteRatioInMethod*）を計算する．クローンセット内の　i番目のコードが存在するメソッドに含まれるメソッド呼び出しの数を *CallSiteInMethod(i)*，クローンセット内の　i番目のコードが存在するメソッドに含まれる「重要なメソッド呼び出しの数」を *ImportantCallSiteInMethod(i)*，クローンセットに含まれるコードの数を n とすると

$$CallSiteInMethodInCloneSet = \sum_{i=1}^{n} CallSiteInMethod(i)$$

$$ImportantCallSiteInMethodInCloneSet = \sum_{i=1}^{n} ImportantCallSiteInMethod(i)$$

$$ImportantCallSiteRatioInMethod = \frac{ImportantCallSiteInMethodInCloneSet}{CallSiteInMethodInCloneSet}$$

となる．この値を計算するのは，クローンになっているコードが，クローンが存在するメソッド全体のコードに比べて「重要なメソッド呼び出し」の割合が多くなっているかどうかを調査するためである．

- ソースコードに宣言されているメソッドを，コードクローンとなっているコードを含んでいるメソッド（*HasCloneMethod*）と含んでいないメソッド（*NotHasCloneMethod*）

に分類し，それぞれの分類のメソッドについて「メソッド呼び出しの数（*CallSiteInMethod*）」「重要なメソッド呼び出しの数（*ImportantCallSiteInMethod*）」「重要なメソッド呼び出しの割合（*ImportantCallSiteRatioInMethod*）」を計算する．

- クローンセット内のすべてのコードに含まれるメソッド呼び出しを「重要で名前が変更されていない（*Important-Unchanged*）」「重要で名前が変更された（*Important-Changed*）」「重要でなく名前が変更されていない（*Unimportant-Unchanged*）」「重要でなく名前が変更された（*Unimportant-Changed*）」の 4 種類に分類し，それぞれの数を計算する．

- さらに，名前が変更されていないメソッド呼び出しについて，呼び出し先のメソッドの属するクラスやパッケージを調べ，比較して以下の 6 つに分類し，それぞれの数を計算する．

  1. 属するクラスもパッケージも同じで，それを JRE ライブラリが提供するもの（*JRE*）．具体的にはパッケージ名に”java”，”javax”，”sun”を含むものを JRE ライブラリが提供するものとした．
  2. 属するクラスもパッケージも同じで，1 に含まれないもの（*NotJRE*）．
  3. 属するクラスは違うが，パッケージは同じもの（*SamePackage*）．
  4. 属するパッケージは違うが，クラスは同じもの（*SameClass*）．
  5. 属するクラスもパッケージも違うが，クラス名に共通した接頭語または接尾語がつくもの（*SimilarClass*）．
  6. 属するクラスもパッケージも違い，5 に含まれないもの（*Different*）．

4　調査結果・考察

オープンソースソフトウェア 6 つを対象に CCFinder を適用してコードクローンを抽出し，そのメトリクス値に対する分析を行った．対象ソフトウェアの名称とバージョンを表 3 に示す．また，対象としたコードクローンはトークン数が 30 以上のものである．

4.1　メソッド呼び出しの分析

分析を行うにあたり，以下の 2 つの点を調査項目として設定した．

- コードがコピーアンドペーストによって複製される際，コピー前とコピー後で処理の内容が変わることはあるのか．
- コードがコピーアンドペーストによって複製される際，コピーされるのはメソッドの主要な処理を行なっている部分なのか．

これらを明らかにするため，メソッド呼び出しについて以下の分析を行った．

4.1.1　重要なメソッド呼び出しと重要でないメソッド呼び出しの比較

各ソフトウェアから検出されたクローンセットについて,「重要なメソッド呼び出しの変更度合 (A)」と「重要でないメソッド呼び出しの変更度合 (B)」が得られている．各ソフトウェアの（A) および（B) のヒストグラムを示す（表 5〜表 16）．

この 2 つの値のクローンセットあたりの平均値に差があるかどうかを検定した．これらの値は正規分布に従わないため Wilcoxon の順位和検定を用いた．帰無仮説は「値 (A) と値 (B) の平均値に差はない」，対立仮説は「値 (A) と値 (B) の平均値は異なる」となる．また，有意水準は 1 %とした．この検定結果について表 4 に示す．

表 3: 調査対象のソフトウェアの情報

ソフトウェア名	バージョン	Java ファイル数	Java ファイルの総行数	メソッド数
Derby	10.9.1.0	1445	549911	21653
h2	1.3.168	500	135114	7184
jtunes	(2009.12.12)	519	134243	7296
Tomcat	7.0.27	1242	348856	16916
XXL	1.0	633	178230	7914
zk	6.5.0	406	77772	4851

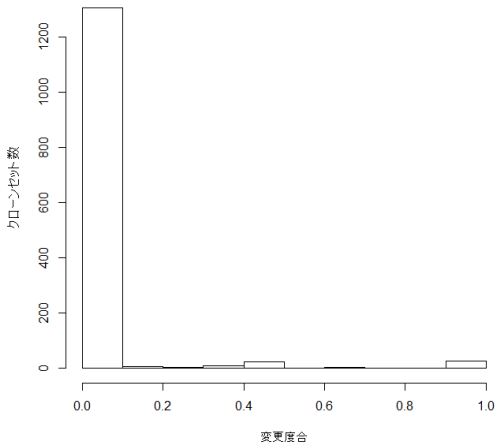


図 5: Derby の (A) のヒストグラム

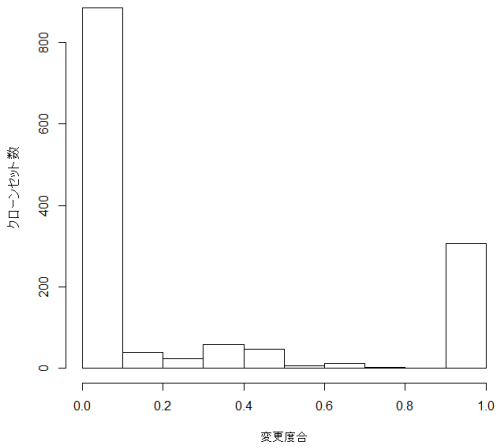


図 6: Derby の (B) のヒストグラム

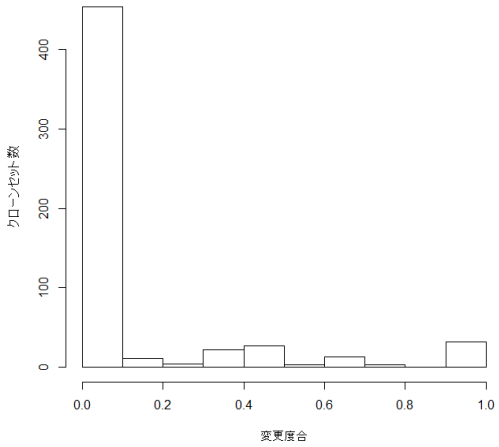


図 7: h2 の (A) のヒストグラム

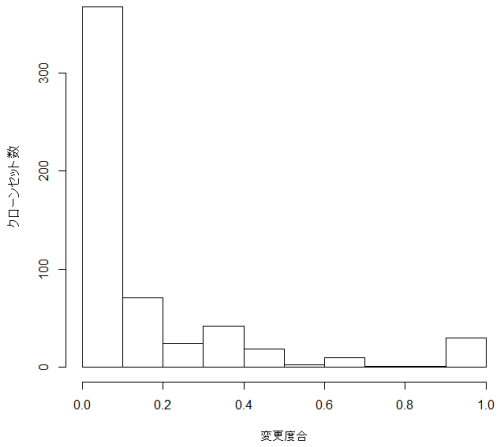


図 8: h2 の (B) のヒストグラム

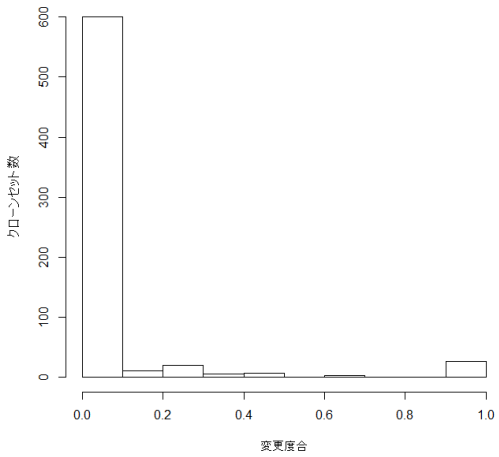


図 9: jtunes の (A) のヒストグラム

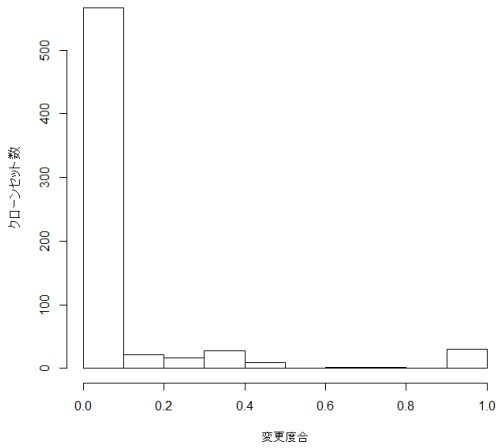


図 10: jtunes の (B) のヒストグラム

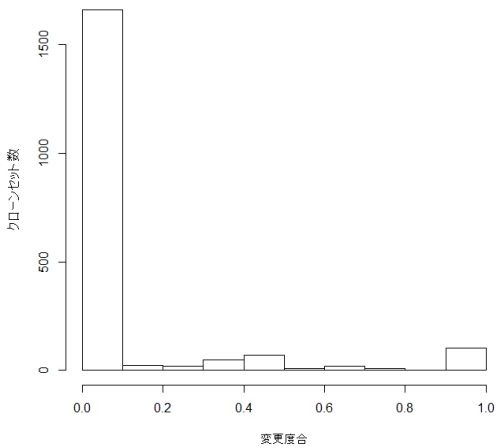


図 11: Tomcat の (A) のヒストグラム

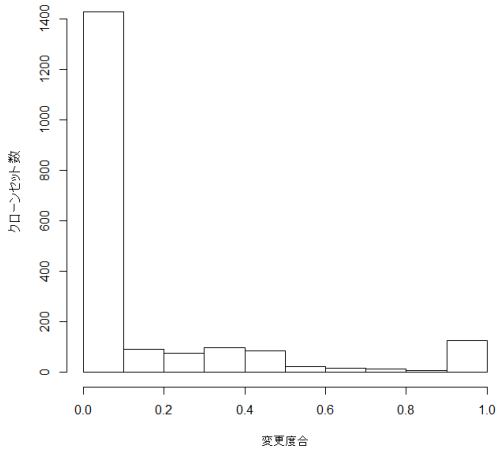


図 12: Tomcat の (B) のヒストグラム

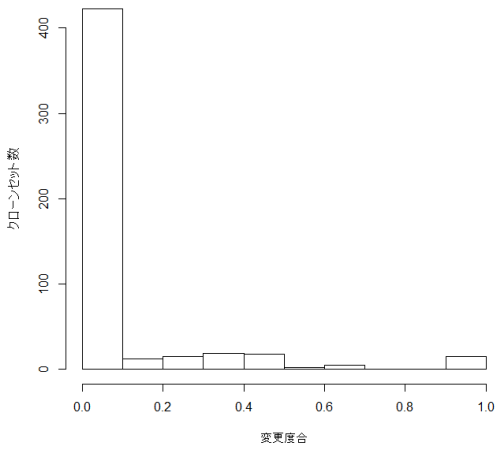


図 13: XXL の (A) のヒストグラム

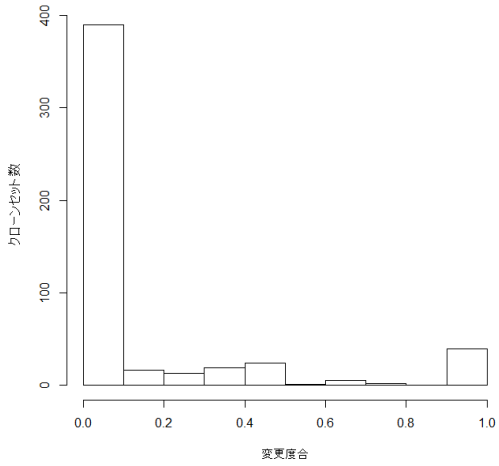


図 14: XXL の (B) のヒストグラム

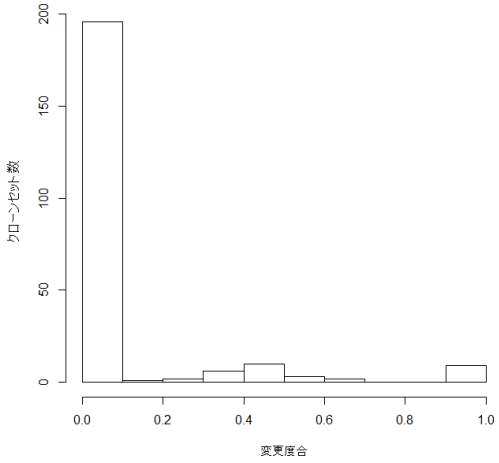


図 15: zk の (A) のヒストグラム

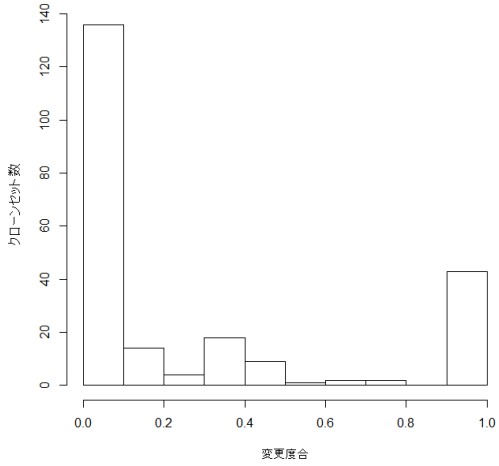


図 16: zk の (B) のヒストグラム

表 4 に示すように 6 つのソフトウェアすべてで (B) の平均値が (A) の平均値を上回り，そこに統計的な差があることがわかった．このことから,「重要なメソッド呼び出し」の名前は保存されやすく，コードがコピーされてもその主要な処理が変わることは少ないと考えられる．

4.1.2 クローン部分とクローンが含まれるメソッド全体でのメソッド呼び出しの比較

各ソフトウェアから検出されたクローンセットについて,「クローンとなっているコードの重要なメソッド呼び出しの割合 (C)」と「クローンが存在しているメソッド全体での重要なメソッド呼び出しの割合 (D)」が得られている．この 2 つの値の 1 クローンセットあたりの平均値に差があるかどうかを検定した．これらの値は正規分布に従わないため Wilcoxon の順位和検定を用いた．帰無仮説は「値 (C) と値 (D) の平均値に差はない」，対立仮説は「値 (C) と値 (D) の平均値は異なる」となる．また，有意水準は 1 %とした．この検定結果について表 5 に示す.

表 5 に示すように，4 つのソフトウェアで帰無仮説は棄却されず, (C) と (D) の平均値に差があるということはできない．つまり，コードがコピーアンドペーストで複製される際,「重要なメソッド呼び出し」を多く含む部分が狙ってコピーされているわけではない．このことから，メソッド内の主要な処理のみをコピーしているわけではないと考えられる．

4.2 呼び出されるメソッドの分析

次に，どのようなメソッドが「重要なメソッド呼び出し」によって呼び出され，どのようなメソッドが「重要でないメソッド呼び出し」によって呼び出されるのか調査を行った．

表 4: (A) と (B) の平均値の差の検定

ソフトウェア名	p 値	帰無仮説	(A) の平均値	(B) の平均値
Derby	2.2e-16 より小さい	棄却	3.4%	27.3%
h2	4.4347e-6	棄却	12.1%	14.6%
jtunes	0.0057	棄却	6.1%	8.2%
Tomcat	2.2e-16 より小さい	棄却	9.5%	14.1%
XXL	0.0032	棄却	8.0%	13.6%
zk	4.967e-10	棄却	8.7%	26.5%

4.2.1 メソッドの分類

まず，各ソフトウェアのソースコード中で宣言されているメソッドを「常に重要なメソッド呼び出しによって呼び出される」,「常に重要でないメソッド呼び出しによって呼び出される」,「どちらの呼び出しにも呼び出される」,「一度も呼び出されない」の 4 種類に分類した．なお，ここでは解析対象のソースファイル内のメソッド及びメソッド呼び出しのみを対象としている．このため，解析対象以外のパッケージなどから呼び出されるように設計されているメソッドは「一度も呼ばれない」に分類される．ソフトウェアごとの結果を表 6 に示す．数字はそれぞれに分類されるメソッドの数を表す．表 6 からわかる通り,「常に重要」,「常に重要でない」の項目に比べ「どちらにも呼ばれる」の値は少なくなっている．このことから,「重要なメソッド呼び出し」と「重要でないメソッド呼び出し」がそれぞれ呼び出すメソッドはある程度傾向が分かれていると推測される．

4.2.2 メソッド名に使用される単語の調査

次に「常に重要なメソッド呼び出しで呼ばれる」メソッドと「常に重要でないメソッド呼び出しで呼ばれる」メソッド，それぞれのメソッド名でよく使用されている単語について調査を行った．方法としては，メソッド名を CamelCase に従い単語に分割し，先頭で使用されている単語を集計して頻度順に並べ替えることで調査した．なお，調査するのは頻度の上位 10 単語とした．まずは「常に重要なメソッド呼び出しで呼ばれる」メソッド,「常に重要でないメソッド呼び出しで呼ばれる」メソッドそれぞれの結果を表 7，表 8 に示す．

表 7，表 8 を見ると get や is などの単語がどちらの表にも含まれている．一方でのみよく使用される単語を示すため，2 つの表に共通して存在する単語を除去したものが表 9，表 10 になる．なお，ここでの「共通して存在する単語」とは，ソフトウェアが同じかどうかを考慮しない．例えば表 7 の Tomcat の 10 番目と表 8 の zk の 9 番目に do という単語が存在する．このように異なるソフトウェア間で共通する単語も除去の対象となる．共通する単語を

表 5: (C) と (D) の平均値の差の検定

ソフトウェア名	p 値	帰無仮説	(C) の平均値	(D) の平均値
Derby	0.0013	棄却	15.6%	14.9%
h2	0.0538	棄却されない	36.4%	36.5%
jtunes	0.0730	棄却されない	43.3%	41.8%
Tomcat	5.868e-07	棄却	35.7%	34.6%
XXL	0.4451	棄却されない	35.2%	35.0%
zk	0.4197	棄却されない	27.0%	27.0%

除いたあとの表を見ると，表 9 からは write や remove，表 10 からは close，to などが複数のソフトウェアに出現している．write や remove は,「書き出す」「除去する」という動作で戻り値を利用することは少なく，void-return に該当することが多いため表 7 に表れたと推測される．to は，toString などのように与えられたデータを別のかたちに変換するメソッドで使用されやすく，変換後のデータが戻り値として利用されていたため「重要なメソッド呼び出し」とならず，表 10 に表れたと推測される．close は，3 章で定義したようにこの単語が使われたメソッドの呼び出しは ending 以外では「重要なメソッド呼び出し」の候補から除去されるため，その多くが「重要でないメソッド呼び出し」と分類され，表 10 に表れたと考えられる．

4.3 コード閲覧による調査

次に，コードクローンの特徴を調べるため,「重要なメソッド呼び出し」が変更されていないクローンセットと「重要なメソッド呼び出し」が多く変更されているクローンセットをそれぞれ各ソフトウェアから 5 例ずつ計 30 例をランダムピックアップし，実際にコードを閲覧して調査した．調査を行ったクローンセットの情報について表 12〜表 23 に示す．表中の CloneID とは，クローンセットに割り振られる識別番号のことである．

4.3.1 「重要なメソッド呼び出し」が変更されていないクローンセット

各ソフトウェアの「重要なメソッド呼び出し」が変更されていないクローンセットの数（*NoChange*）と割合（*NoChangeRatio*）を表 11 に示す．「重要なメソッド呼び出し」が変更されていないクローンセットに含まれるコードを調べた結果,「別パッケージの同名クラス」や「別クラスの同名メソッド」など，完全に同一ではないが，存在するパッケージ名，クラス名，メソッド名のいずれかが同じであるコードがクローンセットとなっていることが多かっ

表 6: メソッドの分類

ソフトウェア名	常に重要	常に重要でない	どちらにも呼ばれる	一度も呼ばれない
Derby	4930	5126	1312	16064
h2	4142	6663	1995	7568
jtunes	2162	4131	647	3781
Tomcat	4202	7435	1645	9158
XXL	1938	2963	691	4303
zk	1411	1845	538	2875

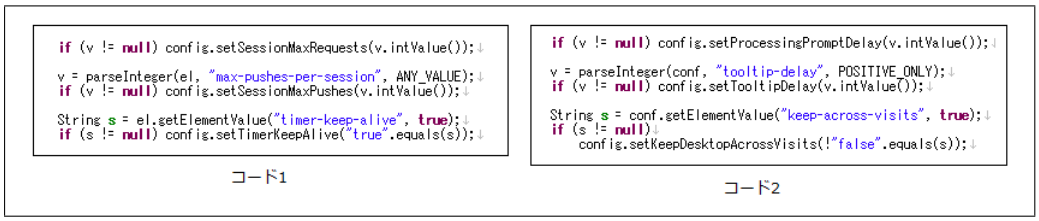


図 17: 「制御構文の繰り返し」のコード例

た．例えば，Tomcat の，AjpNioProcessor クラスの process メソッドと，AjpAprProcessor クラスの process メソッドから「重要なメソッド呼び出し」が変更されていないクローンセットが検出された．調査した 30 例のうち 22 例がこれに該当した．

4.3.2 「重要なメソッド呼び出し」の変更度合が高いクローンセット

「重要なメソッド呼び出し」の変更度合が高いクローンセットを調べた結果，制御構文を繰り返し記述しているコードが多く確認された．これは，開発者にとって興味がないと思われる種類のクローンセットであるが，繰り返し回数が少ないため RNR によってフィルタリングされなかったと考えられる．図 17 はコード 1，コード 2 ともに zk の ConfigParser クラスから検出されたコードであり，この 2 つのコードがクローンセットとなっている．この例では，制御構文が繰り返し現れて，それぞれの制御構文の中で戻り値のないメソッドを呼び出している．このメソッド呼び出しは void-return の「重要なメソッド呼び出し」である．このメソッド呼び出しで呼び出しているメソッドがクローン間で異なるため,「重要なメソッド呼び出し」の変更度合が高くなっている．このような，制御構文の繰り返しで記述されたクローンセットが多数確認され，調査した 30 例中 18 例がこれに該当した．また，残った 12 例中 7 例は，戻り値のないメソッドを繰り返し呼び出しているコードであった．これらのコードクローンは開発者が検出したいコードクローンである可能性は低い．このことから，「重要なメソッド呼び出し」の変更度合が高いクローンセットをフィルタリングすることで，開発者の興味のないコードクローンを除去し，目的のコードクローンを検出する手助けとなることが期待される．

4.4 妥当性

- 本研究は Java で記述されたソースコードについてのみ調査を行った．文法規則の異なる他の言語では違った傾向が見られるかもしれない．調査結果をより一般的なものとするために，対象の言語を増やす必要がある．

- また，本研究では分析するデータが正規分布に従わないため，平均値の差の検定に

Wilcoxon の順位和検定（マン・ホイットニーのU検定）を用いている．Wilcoxon の順位和検定はノンパラメトリックな手法である．ノンパラメトリックな手法は母集団の分布などの前提を必要とせず，広く適用できるがパラメトリックな検定と比較すると検定力が低いという弱点がある．しかし，今回の調査ではコードクローンの検出数が最も少なかった zk でも 229 個という十分な数の標本が得られており，信頼性に問題はないと考えられる．

- 本研究で対象にしたコードクローンは，CCFinder が抽出対象としている正規化しトークン列が一致するソースコード断片である．他のコードクローン検出ツールが取り出すコードクローンについては，本研究の結果があてはまるとは限らない．

- 本研究では，コードクローンのサンプルを取り出して調査を行った．サンプル数の増加などによって，結果が変わる可能性はある．

- 「重要なメソッド呼び出し」の抽出方法は，論文 [22] を参考にして本研究のために作成したものである．[22] で評価されている重要なメソッド呼び出しとは実装の差異による違いが存在する可能性がある．

表 7: 常に「重要なメソッド呼び出し」で呼ばれるメソッドで使われる単語

	1	2	3	4	5	6	7	8	9	10
Derby	get	update	is	read	init	add	write	check	set	bind
h2	get	add	is	read	write	check	remove	parse	init	create
jtunes	add	decode	get	create	paint	remove	is	draw	install	init
Tomcat	get	add	is	remove	create	write	jj	parse	run	do
XXL	get	write	read	update	open	remove	has	is	add	next
zk	get	add	is	parse	remove	on	render	new	set	has

表 8: 常に「重要でないメソッド呼び出し」で呼ばれるメソッドで使われる単語

	1	2	3	4	5	6	7	8	9	10
Derby	get	set	close	is	new	setup	has	to	make	find
h2	get	set	close	is	add	create	debug	read	log	to
jtunes	get	set	create	is	name	contains	close	add	to	read
Tomcat	get	set	is	close	create	find	log	to	read	add
XXL	get	set	close	is	to	has	print	read	index	size
zk	get	set	is	to	add	new	resolve	append	do	parse

表 9: 常に「重要なメソッド呼び出し」で呼ばれるメソッドでのみ頻繁に使われる単語

	1	2	3	4	5	6	7	8	9	10
Derby		update			init		write	check		bind
h2					write	check	remove		init	
jtunes		decode			paint	remove		draw	install	init
Tomcat				remove		write	jj		run	
XXL		write		update	open	remove				next
zk					remove	on	render			

表 10: 常に「重要でないメソッド呼び出し」で呼ばれるメソッドでのみ頻繁に使われる単語

	1	2	3	4	5	6	7	8	9	10
Derby			close			setup		to	make	find
h2			close				debug		log	to
jtunes					name	contains	close		to	
Tomcat				close		find	log	to		
XXL			close		to		print		index	size
zk				to			resolve	append		

表 11: 「重要なメソッド呼び出し」が変更されていないクローンセット

ソフトウェア名	*NoChange*	総クローンセット数	*NoChangeRatio*
Derby	1308	1384	94.5%
h2	452	570	79.3%
jtunes	599	675	88.7%
Tomcat	1661	1962	84.7%
XXL	423	510	82.9%
zk	196	229	85.6%
合計	4639	5330	87.0%

表 12: Derby の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
239	org.apache.derby.iapi.store.raw	D_ContainerKey	diag_detail	59	67
	org.apache.derby.impl.store.raw.data	D_BaseContainerHandle	diag_detail	111	119
597	org.apache.derby.impl.sql.execute	InsertResultSet	updateAllIndexes	1858	1903
	org.apache.derby.impl.sql.execute	InsertResultSet	emptyIndexes	2337	2382
1207	org.apache.derby.impl.sql.compile	InListOperatorNode	generateListAsArray	583	632
	org.apache.derby.impl.sql.compile	CoalesceFunctionNode	generateExpression	213	262
1618	org.apache.derby.impl.sql.catalog	DataDictionaryImpl	createDictionaryTables	8107	8130
	org.apache.derby.impl.sql.catalog	DataDictionaryImpl	createDictionaryTables	8121	8144
	org.apache.derby.impl.sql.catalog	DataDictionaryImpl	createDictionaryTables	8140	8163
2223	org.apache.derby.iapi.services.classfile	ClassHolder	put	161	177
	org.apache.derby.iapi.services.classfile	ClassHolder	put	168	184

表 13: h2 の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
17	org.h2.value	ValueLobDb	getTraceSQL	320	330
	org.h2.value	ValueLob	getTraceSQL	656	666
315	org.h2.store	PageStore	getPage	749	781
	org.h2.store	PageStore	getPage	788	820
662	org.h2.jdbc	JdbcDatabaseMetaData	getImportedKeys	1042	1080
	org.h2.jdbc	JdbcDatabaseMetaData	getExportedKeys	1113	1152
735	org.h2.fulltext	FullTextLucene	init	513	538
	org.h2.fulltext	FullText	init	863	887
1264	org.h2.command.dml	Merge	getPlanSQL	212	232
t	org.h2.command.dml	Inser	getPlanSQL	192	212

表 14: jtunes の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
85	jtunes	PopupButton	init	77	118
	com.explodingpixels.widgets	PopupButton	init	63	108
228	jtunes	TitlePanel	TitlePanel	161	219
	com.explodingpixels.macwidgets	TitlePanel	TitlePanel	167	225
411	com.flagstone.transform.util	FSTextConstructor	decodeAWTFont	826	891
	com.flagstone.transform.util	FSTextConstructor	decodeAWTFont	930	995
582	com.flagstone.transform	FSSoundStreamHead	encode	505	554
	com.flagstone.transform	FSSoundStreamHead2	encode	505	554
1069	com.bric.swing	FadingPanel	captureStates	284	301
	com.bric.swing	FadingPanel	captureStates	302	319

表 15: Tomcat の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
1371	org.apache.jasper.tagplugins.jstl.core	Import	doTag	176	239
	org.apache.jasper.tagplugins.jstl.core	Import	doTag	308	370
1958	org.apache.el.parser	ELParse	Compare	465	581
	org.apache.el.parser	ELParse	Compare	504	620
2026	org.apache.coyote.http11.upgrade	UpgradeNioProcessor	readSocket	119	160
	org.apache.coyote.ajp	AjpNioProcessor	readSocket	328	362
2158	org.apache.coyote.ajp	AjpNioProcessor	process	144	218
	org.apache.coyote.ajp	process	AjpAprProcessor	157	231
2669	org.apache.catalina.servlets	WebdavServlet	doPropfind	577	583
	org.apache.catalina.servlets	WebdavServlet	doLock	1023	1029
	org.apache.catalina.servlets	WebdavServlet	doLock	1046	1052
	org.apache.catalina.servlets	WebdavServlet	doLock	1076	1082

表 16: XXL の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
143	xxl.core.spatial.points	Points	universeDouble	239	254
	xxl.core.spatial.points	Points	universeFloat	267	282
533	xxl.core.indexStructures	ORTree	remove	673	707
	xxl.core.indexStructures	MTree	remove	831	863
629	xxl.core.collections.containers	UnmodifiableContainer	main	245	251
	xxl.core.collections.bags	UnmodifiableBag	main	212	218
	xxl.core.collections.bags	ListBag	main	337	343
	xxl.core.collections.bags	ListBag	main	367	373
	xxl.core.collections.bags	ListBag	main	389	395
	xxl.core.collections.bags	DynamicArrayBag	main	570	576
	xxl.core.collections.bags	DynamicArrayBag	main	599	605
	xxl.core.collections.bags	ContainerBag	main	326	332
	xxl.core.collections.bags	ContainerBag	main	359	365
	xxl.core.collections.bags	ArrayBag	main	409	415
	xxl.core.collections.bags	ArrayBag	main	438	444
642	xxl.core.collections.bags	ListBag	main	321	363
	xxl.core.collections.bags	DynamicArrayBag	main	555	595
698	xxl.applications.xml	Common	getSampleTree	237	259
	xxl.applications.xml	Common	getSampleTree	249	271

表 17: zk の「重要なメソッド呼び出し」が変更されていないクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
0	org.zkoss.zk.xel.impl	Utils	evaluateComposite	127	147
	org.zkoss.zk.xel.impl	Utils	evaluateComposite	151	171
61	org.zkoss.zk.ui.sys	JsContentRenderer	renderValue	165	229
	org.zkoss.zk.ui.sys	JsContentRenderer	renderValue	176	239
252	org.zkoss.zk.ui.http	ZumlExtendlet	handleError	166	181
	org.zkoss.zk.ui.http	DHtmlLayoutServlet	handleError	254	270
275	org.zkoss.zk.ui.http	DHtmlLayoutServlet	handleError	260	279
	org.zkoss.zk.ui.http	DHtmlLayoutPortlet	handleError	312	330
336	org.zkoss.zk.au.http	AuUploader	parseRequest	361	384
	org.zkoss.zk.au.http	AuDropUploader	parseRequest	305	329

表 18: Derby の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
181	org.apache.derby.impl.store.raw.data	StoredPage	updatePageHeader	1928	1934
	org.apache.derby.impl.sql.compile	NormalizeResultSetNode	generate	664	670
704	org.apache.derby.impl.sql.conn	GenericLanguageConnectionContext	lookupCursorActivation	1157	1170
	org.apache.derby.impl.sql.compile	ResultColumnList	updateOverlaps	2659	2669
975	org.apache.derby.impl.sql.compile	ResultColumn	columnTypeAndLengthMatch	1262	1279
	org.apache.derby.iapi.types	DataTypeDescriptor	isExactTypeAndLengthMatch	957	969
1378	org.apache.derby.impl.sql.catalog	XPLAINScanPropsDescriptor	setStatementParameters	116	137
	org.apache.derby.impl.sql.catalog	XPLAINResultSetDescriptor	setStatementParameters	130	151
1999	org.apache.derby.iapi.types	SQLTime	encodedTimeToString	908	913
	org.apache.derby.iapi.types	SQLDate	dateToString	823	828

表 19: h2 の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
40	org.h2.value	Value	convertTo	555	560
	org.h2.value	Value	convertTo	818	823
68	org.h2.value	Transfer	writeValue	482	495
	org.h2.store	Data	writeValue	646	659
117	org.h2.value	CompareMode	getCollator	168	172
	org.h2.tools	ConvertTraceFile	convertFile	157	161
340	org.h2.store	Data	getValueLen	907	919
	org.h2.store	Data	getValueLen	918	930
1167	org.h2.command	Parser	parseCreate	3840	3850
	org.h2.command	Parser	parseCreateTrigger	4141	4151

表 20: jtunes の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
252	jtunes	AngleSliderUI	installUI	206	211
	jtunes	AngleSliderUI	uninstallUI	306	311
405	com.flagstone.transform.util	FSTextConstructor	defineShape	692	698
	com.flagstone.transform.util	FSTextConstructor	defineShape	701	707
548	com.flagstone.transform	FSText	encode	609	628
	com.flagstone.transform	FSSound	encode	548	565
1009	com.flagstone.transform	FSColorTransform	decode	868	876
	com.flagstone.transform	FSColorTransform	decode	878	886
1239	com.bric.math	MathG	testEverything	61	235
	com.bric.math	MathG	testEverything	73	247

表 21: Tomcat の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
330	org.apache.tomcat.util.buf	MessageBytes	equals	299	307
	org.apache.tomcat.util.buf	MessageBytes	equalsIgnoreCase	320	328
	org.apache.tomcat.util.buf	MessageBytes	startsWith	380	388
1531	org.apache.jasper.compiler	Validator	visit	133	153
	org.apache.jasper.compiler	Validator	visit	147	167
1837	org.apache.jasper.compiler	Generator	generateCustomStart	2307	2316
	org.apache.catalina.deploy	ApplicationParameter	toString	110	118
3157	org.apache.catalina.deploy	WebXml	merge	1668	1736
	org.apache.catalina.deploy	WebXml	merge	1689	1757
3315	org.apache.catalina.core	NamingContextListener	containerEvent	371	427
	org.apache.catalina.core	NamingContextListener	containerEvent	379	436

表 22: XXL の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
54	xxl.core.xml.relational.sax	XMLWriterHandle	endElement	109	116
	xxl.core.xml.relational.sax	XMLWriterHandle	endElement	110	118
509	xxl.core.io	ByteArrayConversions	main	592	600
	xxl.core.io	ByteArrayConversions	main	600	608
552	xxl.core.cursors.visual	ControllerJPanel	init	243	250
	xxl.core.cursors.visual	ControllerJPanel	init	260	267
670	xxl.applications.xml.storage	PersistentDOMTest	testNode	139	141
	xxl.applications.xml.storage	PersistentDOMTest	testNode	174	176
781	xxl.applications.io	RAFTest	makeRAFTest	110	113
	xxl.applications.io	RAFTest	makeRAFTest	116	118

表 23: zk の「重要なメソッド呼び出し」が多く変更されているクローンセット例

CloneID	パッケージ名	クラス名	メソッド名	開始行	終了行
75	org.zkoss.zk.ui.sys	ConfigParser	parse	368	374
	org.zkoss.zk.ui.sys	ConfigParser	parseClientConfig	644	651
120	org.zkoss.zk.ui.select.impl	ComponentIterator	buildRootCtx	231	238
	org.zkoss.zk.ui.select.impl	ComponentIterator	buildRootCtx	239	246
168	org.zkoss.zk.ui.metainfo	PageDefinition	preInit	901	912
	org.zkoss.zk.ui.metainfo	PageDefinition	preInit	914	925
214	org.zkoss.zk.ui.impl	PageImpl	sessionWillPassivate	1087	1099
	org.zkoss.zk.ui.impl	PageImpl	sessionDidActivate	1109	1122
288	org.zkoss.zk.ui	HtmlBasedComponent	service	617	622
	org.zkoss.zk.ui	HtmlBasedComponent	service	622	627

5　まとめ

本研究ではコードクローンとなっているコードが持つ特徴を調査するため，コードクローンのメトリクスを計測し，分析を行った．その結果,「重要なメソッド呼び出し」はそうでないメソッド呼び出しに比べて名前の修正が加えられにくいこと，コードクローンになっている部分となっていない部分で「重要なメソッド呼び出しの割合」に有意な差はないこと，名前に write や remove を含むメソッドは「重要なメソッド呼び出し」によって呼び出されることが多く，名前に to を含むメソッドは「重要でないメソッド呼び出し」によって呼び出されることが多いことがわかった．また,「重要なメソッド呼び出しの変更度合」の高いコードクローンには制御構造の繰り返しによって記述されたコードが多いことがわかった．「重要なメソッド呼び出し」が変更されていないコードクローンの多くは，同一パッケージ内部，あるいは類似した名前のクラスなどに出現しており，特定の処理をクラス間，メソッド間で共有する方法がないために作られているのだと考えられる．一方,「重要なメソッド呼び出し」の変更度合が高いコードクローンは同一の制御構造の繰り返しが多く，制御構造の再利用であると考えられる.「重要なメソッド呼び出し」が変更されていないコードクローンは全体の 87%を占めていることから，コードクローンの多くがメソッド呼び出しによって実現されている処理の再利用であると考えられる．

今後の課題としては，対象ソフトウェアの数や種類を増やしたり，対象言語を増やすことで今回の分析結果の一般性を確認することが挙げられる．また，今回は呼ばれるメソッドの分析はそのメソッド名についてのみに留まった．呼ばれるメソッドについて調査する項目を広げることで「重要なメソッド呼び出し」で呼ばれやすいメソッドの特徴がより見えるかもしれない．

謝辞

本研究の全過程を通して，常に適切な御指導及び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いたします.

本研究において，随時適切な御指導及び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准教授に深く感謝いたします.

本研究において，終始適切な御指導及び御助言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石尾隆助教に深く感謝いたします.

本論文を作成するにあたり, 適切な御指導および御助言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眞鍋雄貴特任助教に深く感謝いたします.

本論文を作成するにあたり, 数多くの御指導および御助言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伊達浩典氏に深く感謝いたします.

本論文を作成するにあたり, 様々な御指導および御助言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鹿島悠氏に深く感謝いたします.

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] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. *In Proceedings of the 14th International Conference on Software Maintenance (ICSM 1998)*, pp. 368–377, 1998.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591, 2007.

[3] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. *In Proceeding of the International Conference on Software Maintenance (ICSM 2005)*, pp. 27–36, 2005.

[4] M. Bruntink, A. Deursen, R. Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concerns. *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, pp. 804–818, 2005.

[5] E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored?: A case study of three OSS projects. *In Proceedings of the 5th Workship on Refactoring Tools (WRT 2012)*, pp. 1–7, 2012.

[6] Apache Derby. http://db.apache.org/derby/.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. *In Proceedings of the 15th International Conference on Software Maintenance (ICSM 1999)*, pp. 109–118, 1999.

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

[9] h2. http://www.h2database.com/html/main.html.

[10] Y. Hayase, Y. Yong Lee, and K. Inoue. A criterion for filtering code clone related bugs. *In Proceedings of the 2008 Workchip on Defects in Large Software System (DEFECTS 2008)*, pp. 37–38, 2008.

[11] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. *In Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC 2007)*, pp. 262–269, 2007.

[12] T. Hon and G. Kiczales. Fluid aop join point models. *In Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, System, Languages, and Applications (OOPSLA 2006)*, pp. 712–713, 2006.

[13] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. *In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pp. 55–64, 2007.

[14] jtunes. http://sourceforge.jp/projects/sfnet_jtunes-online/.

[15] E .Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy and paste. *In Proceedins of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pp. 78–87, 2010.

[16] 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.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pp. 220–242, 1997.

[18] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. *In Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2004)*, pp. 83–92, 2004.

[19] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo, and J. Hudepohl. Assessing the bene ts of incorporating function clone detection in a development process. *In Proceedings of the 13th International Conference on Software Maintenance (ICSM 1997)*, pp. 314–321, 1997.

[20] J. Li and M. D. Emst. CBCD: cloned buggy code detector. *In Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pp. 310–320, 2012.

[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] G. Sridhara, E. Hill, D. Muppaneni, and L. Pollick. Towards automatically generating summary comments for java methods. *In Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pp. 43–52, 2010.

[23] Apache Tomcat. http://tomcat.apache.org/.

[24] XXL. http://dbs.mathematik.uni-marburg.de/home/research/projects/xxl/.

[25] zk. http://www.zkoss.org/.