コードクローンとの位置関係に基づく欠陥混入傾向の調査

中山 直輝[1,a)]　吉田 則裕[2,b)]　藤原 賢二[1,c)]　飯田 元[1,d)]　高田 光隆[2,e)]　高田 広章[2,f)]

概要：コードクローンは，ソースコード中に存在するコード片のうち，同一プロジェクト内に同一または類似した部分を持つものを指し，主に開発者が行うコピーアンドペーストによって混入する．コピーアンドペーストによるコード片の再利用は，欠陥混入の原因となる恐れがあることから，コードクローン検出ツールを利用して欠陥を発見する手法が提案されている．それらの手法の多くはコードクローン内に存在する欠陥を対象としているが，コードクローンに起因する欠陥はコードクローン外にも存在するため，開発者はコードクローン内外の両方を検査する必要がある．しかし，コードクローン内外のコード片における欠陥混入傾向の差異は明らかになっておらず，コードクローン外を効率的に検査する手法も確立されていない．そこで本研究では，オープンソースソフトウェアの開発履歴を用いて，コードクローン内外における欠陥混入傾向をコード片とコードクローンの位置関係に基づき定量的に調査した．その結果，コードクローン近傍のコード片において欠陥が多く混入する傾向が見られた．

キーワード：コードクローン, ソフトウェアリポジトリマイニング, 実証的ソフトウェア工学

Investigation into the Defect Occurrence Based on Positional Relationship between Code Clone and Code Fragment

Abstract: Code clone is a duplicate code fragment in the source code of software, and usually generated when developer conducts copy-and-paste. Since reuse of code fragment by copy-and-paste is considered to be a possible reason of introducing or spreading defects, methods of code review based on code clone analysis have been proposed by several previous researches. Although most of them can detect only the defect within cloned code fragment, it is necessary for developers to review both cloned and non-cloned code fragments because clone related defects might exist in non-cloned code fragment as well. However, there is no clarification regarding the tendency of defect occurrence within cloned code fragment or non-cloned code fragment. Moreover, method of code review for efficiently detecting defects in non-cloned fragment has not been established yet. For addressing these issues, this research quantitatively investigated the tendency of defect occurrence within cloned code fragment or non-cloned code fragment based on the positional relationship between code clone and code fragment by using development histories of open source software. The investigation result shows that defects tend to be involved in code fragments neighboring code clones.

Keywords: Code Clone, Mining Software Repositories, Empirical Software Engineering

1. はじめに

コードクローンは，ソースコード中に存在するコード片のうち，同一プロジェクト内に同一または類似した部分を持つものを指し，主に開発者が行うコピーアンドペーストによって混入する [1]．コードクローンの存在は，ソフトウェアの品質に悪影響を与えるといわれている [2]．例えば，開発者があるコード片を編集すると、そのコードクローンに対しても同様の編集 (以降，同時編集とする) を

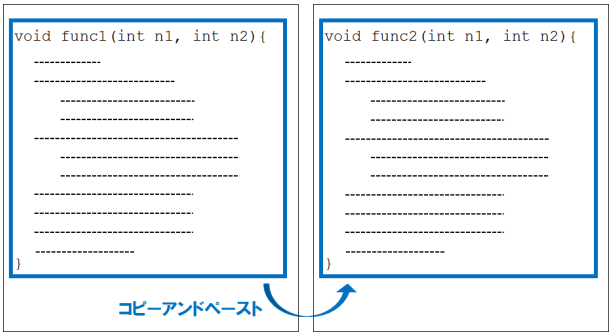
[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan
[2] 名古屋大学
Nagoya University, Nagoya, Aichi 464-8601, Japan
[a)] nakayama.naoki.my7@is.naist.jp
[b)] yoshida@ertl.jp
[c)] kenji-f@is.naist.jp
[d)] iida@itc.naist.jp
[e)] mtakada@nces.is.nagoya-u.ac.jp
[f)] hiro@ertl.jp

行う必要性が高く，その同時編集に漏れがあると，新たに欠陥を混入させてしまう恐れがある．そのため，開発者はすべてのコードクローンに対して同時編集の是非を検討しなければならないが，ソフトウェアが大規模な場合，全てのコードクローンを手作業で探すことは非常に困難な作業となる．そこで，これまでにソースコード中からコードクローンを自動で検出するコードクローン検出ツールが数多く開発されている [3], [4], [5], [6], [7]．それらのツールを用いることでコードクローンを効率的かつ正確に保守することができる [8]．

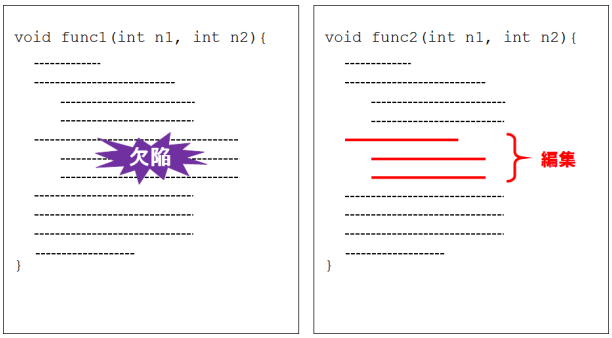
近年，コードクローン検出ツールを利用して，コードクローンに起因する欠陥を発見する手法が提案されている [5], [9]．コードクローンに起因する欠陥として，コードクローンに対する一貫していない修正や，コピー先プログラムとの不整合が原因で混入する欠陥が挙げられる．

コードクローン検出ツールを用いた多くの欠陥検出手法はコードクローン内（検出されたコード片の先頭行から最終行の範囲内）に存在する欠陥を対象としているが，コードクローンに起因する欠陥がツールの検出するコードクローンの外（検出されたコード片の先頭行から最終行の範囲外）に存在する場合もある．その例として，同時編集の漏れが原因で混入した欠陥がコードクローンの外に存在する例を図 1 に示す．図 1(a) は，ある開発者がコピーアンドペーストによって関数 func1 を再利用して関数 func2 を作成した例を示している．図 1(b) は，別の開発者が図 1(a) のコピー先の関数 func2 だけを編集した結果，本来は同時に編集されるべきであったコピー元の関数 func1 に編集漏れが生じ，その漏れが欠陥となった例である．そして図 1(c) は、図 1(b) において欠陥が混入したソースコードに対し，ツールによるコードクローン検出を行った結果を示している．開発者がコピー先の関数のみを編集した結果，2 つの関数の類似度が下がり，検出ツールは再利用した関数全体をコードクローンとして検出していない．ただし，部分的に類似しているコード片はコードクローンとして検出されるため，結果として関数 func1 に混入した欠陥はツールが検出するコードクローンの外に存在している．このように，コードクローンに起因する欠陥はツールが検出するコードクローンの外に存在する場合もあるため，コードクローンに着目して欠陥の検査を行う際は，コードクローン内外の両方に対して欠陥の有無を確認する必要があると考えられる．しかし，コードクローン内外のコード片における一般的な欠陥混入傾向は著者らが知る限り明らかになっておらず，コードクローン外を効率的に検査する手法も確立されていない．

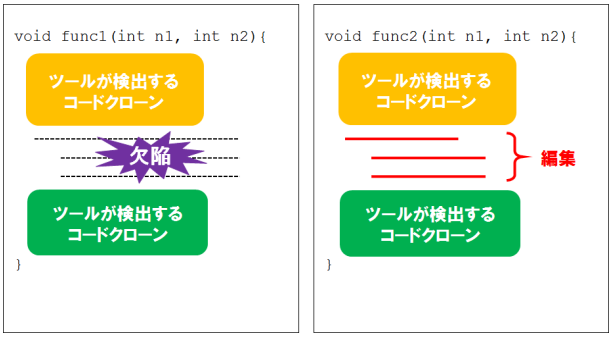
そこで本研究では，オープンソースソフトウェアの開発履歴を用いて，コードクローン内外における欠陥混入傾向をコード片とコードクローンの位置関係に基づき定量的に調査する．1 つ目の調査では，コードクローン内のコード



(a) コピーアンドペーストによる関数の再利用



(b) 同時編集の漏れに起因する欠陥の混入



(c) コードクローン検出の結果

図 1　コードクローンに起因する欠陥がコードクローンの外に存在する例

Fig. 1　Example of clone related defects' occurring within non-cloned code fragment.

片とコードクローン外のコード片の欠陥率を比較する．2 つ目の調査では，欠陥とコードクローンの距離に着目し，コードクローン外のコード片においてコードクローンから近いコード片と遠いコード片で欠陥混入数に違いがあるかどうかを分析する．本研究では，欠陥とコードクローンの距離を，欠陥を含む最終行からコードクローンの先頭行までの行数（欠陥がコードクローンより上側に混入した場合）もしくはコードクローンの最終行から欠陥を含む先頭行までの行数（欠陥がコードクローンより下側に混入した場合）とする．なお，欠陥の先頭行および最終行は，版管理シス

テムおよび欠陥管理システムに記録された情報から算出した（3.1.2 項参照）．

以降，2 章では本研究の関連研究について述べ、3 章では具体的な調査手法について述べる．4 章では調査結果を示し、5 章では調査結果に対する考察を述べる．最後に，6 章で今後の課題を示して本稿をまとめる．

2. 関連研究

本章では，本研究の関連研究としてコードクローンと欠陥の関連性を定量的に調査した既存研究について述べる．これまでに，さまざまな研究者によってコードクローンと欠陥の関連性に関する定量的調査が行われている [10], [11], [12], [13].

Sajnani らは，Java で開発された OSS を対象に，コードクローン内外の欠陥密度を比較する調査を行っている [13]. この調査における欠陥密度とは，ソースコード 1000 行あたりに含まれる欠陥数のことである．欠陥検出には，Java のソースコードから欠陥となる可能性が高いコード片を検出する FindBugs というツールを用いている．コードクローン検出には，字句解析に基づき関数単位のコードクローンを検出する独自のツール [14] を用いている．調査の結果，31 プロジェクト中 26 プロジェクトにおいて，コードクローン内よりもコードクローン外の方が欠陥密度が高いことが確認された．

Rahman らは，OSS のスナップショット全体におけるコードクローン率と欠陥におけるコードクローン率を比較する定量的調査を行っている [12], [15]．OSS の版管理システムおよび欠陥管理システムにおける開発履歴とコードクローン検出ツールである DECKARD を用いて欠陥およびコードクローンの位置情報を取得し，それらの位置情報に基づきコードクローン率を計算して比較している．コードクローン率とは，あるコード片の総行数に対するコード片に含まれるコードクローンの総行数の割合である．調査の結果，調査対象の全プロジェクトにおいて，欠陥のコードクローン率よりもスナップショット全体のコードクローン率が高いことが確認され，コードクローンは欠陥の主な原因ではないと結論づけている．

ここで，前述の 2 つの調査の問題点および本研究との違いについて述べる．Sajnani らの調査で用いられた FindBugs は，最大 50%の確率で欠陥を誤検出することが Hovemeyer らによって報告されており [16]，調査結果の信頼性が高いとはいえない．また，関数単位のコードクローンを検出して調査を行っている点で本研究とは調査内容が異なる．Rahman らの調査は，コードクローン内外の欠陥率を比較していないため本研究とは動機が異なる．さらに，コードクローンに関連する欠陥をコードクローン内に存在する欠陥に限定している点で，コードクローンと欠陥の関連性に関する調査として妥当性が十分にあるとはいえない．

3. 調査手法

本研究では，ソフトウェアの開発履歴を用いてコードクローン内外のコード片における欠陥混入傾向を定量的に調査する．本章では，その調査手法について述べる．図 2 に調査手法の概要を示す．

本研究では，コード片とコードクローンの位置関係に着目し，以下の 2 つのリサーチクエスチョンを設定した．

RQ1 コードクローン内とコードクローン外では，コードクローン内の方が欠陥率が高いか．

RQ2 コードクローン外では，コードクローンに近いコード片で欠陥が多く混入するか．

コードクローンに着目した欠陥検出に関する既存研究では，コードクローン内に存在する欠陥を OSS のソースコード中から検出している [5], [9]．このような背景から，コードクローン内にはコードクローンに起因する欠陥が多く存在し，コードクローン外よりも欠陥率が高くなると考えられるため RQ1 を設定した．また，コードクローンの外かつ近辺のコード片には 1 章の図 1 で示したような欠陥が存在する可能性があり，その影響でコードクローン近辺は欠陥が多く混入すると考えられるため RQ2 を設定した．この 2 つのリサーチクエスチョンに関する調査の結果として，欠陥が混入しやすいコード片とコードクローンの位置関係が明らかになれば，開発者はそれらの指標に基づいて欠陥が混入しやすいコード片から優先的に検査することができ，効率的に欠陥を探すことができると考えられる．

本研究では，調査対象のソフトウェアで使用されている版管理システムおよび欠陥管理システムの開発履歴に基づ

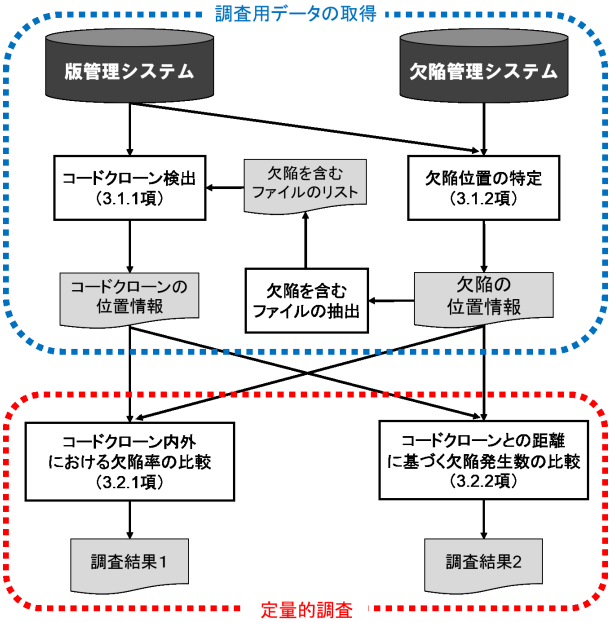


図 2 調査手法の概要
Fig. 2 Overview of investigation method.

き調査用データの取得を行う．調査用データとは，欠陥とコードクローンの位置情報である．本研究における欠陥とは，版管理システムで管理されているコミットのうち，欠陥管理システムで報告された欠陥修正が実施されたコミット（以降，欠陥修正コミットとする）において変更されたコード片のことを指す．そして，取得した調査用データを用いてリサーチクエスチョンごとに定量的調査を行う．以下に各調査の概要を示す．

調査 1　コードクローン内外における欠陥率の比較

調査 2　コードクローンとの距離に基づく欠陥混入数の比較

調査 1 では RQ1 に取り組むため，欠陥とコードクローンの両方を含むファイルを対象にコードクローン内外の欠陥率を比較する．調査 2 では RQ2 に取り組むため，調査 1 で対象としたファイルに存在する全てのコードクローンについてコードクローンと欠陥の距離を求め，その距離ごとに欠陥混入数を比較する．1 章で述べたとおり，本研究では欠陥とコードクローンの距離を，欠陥を含む最終行からコードクローンの先頭行までの行数（欠陥がコードクローンより上側に混入した場合）もしくはコードクローンの最終行から欠陥を含む先頭行までの行数（欠陥がコードクローンより下側に混入した場合）とする．

以降，3.1.1 項では，コードクローンの位置情報を取得する手法（コードクローン検出）について述べ，3.1.2 項では，欠陥の位置情報を取得する手法について述べる．3.2.1 項では，調査 1 に関する具体的な分析手法について述べ，3.2.2 項では，調査 2 に関する具体的な分析手法について述べる．

3.1　調査用データの取得

本研究で行う調査用データの取得手法は，Rahman らの手法に基づいている [12], [15].

3.1.1　コードクローン検出

本研究では，コードクローン検出ツールである DECKARD を用いてコードクローンの位置情報を取得する．DECKARD を用いる理由は，文の編集が行われたコードクローンも検出可能であり，大規模なソフトウェアにおいて現実的な時間で検出できるためである．DECKARD の設定は，最小一致トークン数 *1，類似度の閾値 *2，ストライド *3 をそれぞれ 50，1.0，2 とし，この設定を Rahman らの調査 [12], [15] に基づき Conservative と呼ぶ．調査 1 では DECKARD の設定の違いが調査結果に与える影響を分析するため，類似度の閾値を 0.99 に変更した調査を行う．なお，最小一致トークン数とストライドは変更せず，この設定を Rahman らの調査 [12], [15] に基づき Liberal と呼ぶ．さらに調査 1 では，コードクローン検出ツールの違いが結果に与える影響を分析するため，CCFinder を用いた調査も実施する．CCFinder の設定は，最小一致トークン数が 50，類似度の閾値が 1.0 の初期設定を用いる．

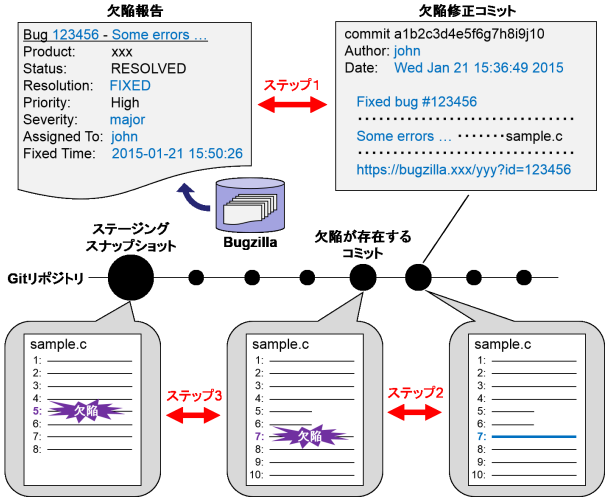


図 3　欠陥の位置情報を取得する手順の概要

Fig. 3　Process for specifying the location of defect.

3.1.2　欠陥位置の特定

調査対象のソフトウェアで使用されている版管理システムおよび欠陥管理システムにおける開発履歴に基づき，欠陥の位置情報を取得する．本研究では，版管理システムとして Git*4，欠陥管理システムとして Bugzilla*5 を採用しているソフトウェアを対象に調査を行う．以下，Git と Bugzilla の情報を用いて欠陥の位置情報を取得する具体的な手順を述べる．図 3 は手順の概要である．

ステップ 1　Śliwerski らの SZZ アルゴリズム [17] に従い，Git リポジトリに存在するコミットの中から欠陥修正コミットを抽出し，Bugzilla の欠陥報告との対応付けを行う．なお，誤った対応付けを避けるために Bachmann らのフィルタリング手法 [18] を適用し，欠陥報告における欠陥修正日時が欠陥修正コミットの作成日の前後 7 日間以内であるものだけを採用する．*6 さらに，対

*1　最小一致トークン数とは，検出されるコードクローンのトークン数の閾値を表し，トークン数がこの値より小さいコード片はコードクローンとして検出されない．この値が小さいと，より小規模なコード片がコードクローンとして検出されるため，連続する関数宣言といったコピーアンドペースト以外の要因で混入するコードクローンも多く検出される [12], [15].

*2　コードクローンとして検出されるコード片同士の類似度は 0 から 1 の実数値で表され，この閾値を 1 より小さく設定するとコードクローンの検出数および誤検出数が増加する．Jiang らの調査によると，類似度の閾値が 0.99 を下回ると検出量が急激に増加し，誤検出の量も増えることが分かっているため，0.99 より小さい値での調査は行わない．

*3　ストライドとは，抽象構文木上で隣り合う部分木をマージする際の範囲を表している．Jiang らの調査によると，この値を 2 に設定するとコードクローンの検出数が最大となるため，この値は 2 で固定する．

*4　http://git-scm.com/

*5　http://www.bugzilla.org/

*6　フィルタリングを行う理由は，一般的に開発者は欠陥修正コミットを作成した数分から数時間以内に欠陥管理システムで欠陥修正報告を行うためである [18]．Eclipse プロジェクトでは，Bachmann らが特定した欠陥修正コミットのうち，全体の 92%において時

応する欠陥報告において欠陥の深刻度（Severity）が‘enhancement’に設定されているものも調査対象から除外する．[*7]

ステップ 2　ステップ 1 で抽出した欠陥修正コミットについて，それぞれの変更内容に基づき欠陥の位置情報を取得する．本研究では，欠陥修正コミットにおいて変更されたコード片を欠陥とする．例えば，欠陥修正コミットを r，その直前のコミットを $r-1$ とすると，r で変更された $r-1$ に存在するコード片が欠陥となるため，r と $r-1$ の間で UNIX の Diff コマンドを適用して欠陥の位置情報を取得する．[*8]

ステップ 3　本研究では，欠陥が存在するコミット（以降，欠陥コミットとする）のスナップショットに対してコードクローン検出を行い，欠陥とコードクローンの位置関係を特定する．しかし，全ての欠陥コミットについてコードクローン検出を行うと計算時間が膨大になる恐れがあるため，Rahaman らの手法 [12], [15] に基づき，月の初めのスナップショット（以降，ステージングスナップショットとする）を対象に調査を行う．具体的には，ステップ 2 で特定した欠陥がステージングスナップショットにおいて何行目に該当するかを UNIX の Diff コマンドによって求める．例えば，ステージングスナップショットを ss とし，ss における欠陥の行番号を l_{ss}，$r-1$ における欠陥の行番号を l_{r-1} とする．そして，ss と $r-1$ のスナップショット間で，l_{ss} の上側に文が m 行追加，n 行削除されたとすると，l_{ss} は以下の式で求められる．

$$l_{ss} = l_{r-1} - m + n$$

なお，ss と $r-1$ の間で新しく追加された行は，ss に存在しないコード片であるため調査対象から除外する．図 4 に示した例の場合，$r-1$ における sample.c の 7 行目に欠陥が存在し，ss と $r-1$ の間で欠陥の上側に文が 3 行追加，1 行削除されている．この場合の ss における欠陥の位置（行番号）は $l_{ss} = 7-3+1 = 5$ となる．

3.2　位置情報に基づく定量的調査手法

3.1 節で抽出した調査用データを用いて行う 2 つの調査について，具体的な調査手法を以下に述べる．

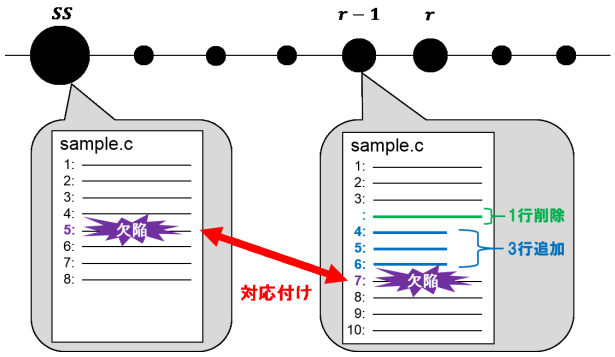


図 4　ステージングスナップショットへの対応付けの例
Fig. 4　Example of mapping the location of defect to staging snapshot.

3.2.1　調査 1：コードクローン内外における欠陥率の比較

調査 1 では，コードクローン内外の欠陥率を比較することで RQ1 に取り組む．まず調査対象のプロジェクトにおける各ステージングスナップショットからコードクローンと欠陥の両方が存在するファイルを抽出し，ファイルごとにコードクローン内外の欠陥率を計算する．ここで，各ファイルのソースコード中に存在する行の行番号の集合を S，欠陥に該当する行の行番号の集合を D，コードクローンに該当する行の行番号の集合を C，コードクローン外に存在する行の行番号の集合を $\bar{C}$ とし，コードクローン内外の欠陥率をそれぞれ R_{clone}，$R_{nonclone}$ とすると，欠陥率の計算方法は以下のようになる．

$$R_{clone} = \frac{|D \cap C|}{|C|} \qquad R_{nonclone} = \frac{|D \cap \bar{C}|}{|\bar{C}|}$$

例えば，調査対象のソースコードの行数が 500 行であり，そのうちコードクローンに該当するコード片（コードクローン内）が 50 行，コードクローンでないコード片（コードクローン外）が 450 行あるとする．そして，コードクローン内において欠陥に該当する行の総数が 2 行，コードクローン外において欠陥に該当する行の総数が 5 行であるとすると，コードクローン内外の欠陥率はそれぞれ以下のように求められる．

$$R_{clone} = \frac{2}{50} = 0.04 \qquad R_{non-clone} = \frac{5}{450} = 0.01$$

以上の方法で各調査対象ファイルにおける欠陥率を求めた後，コードクローン内外の欠陥率に統計的有意差があるかどうかを検定する．

3.2.2　調査 2：コードクローンとの距離に基づく欠陥混入数の比較

調査 2 では，コードクローンと欠陥の距離を求め，その距離ごとに欠陥混入数を比較することで RQ2 に取り組む．なお，本研究では，調査対象の欠陥をコードクローンから最も近い欠陥（以降，最近傍欠陥とする）に限定して調査を行う．この調査では，コードクローンと欠陥の両方が存

間差が 1 日以内であったことから，7 日間を超える時間差が存在する場合は，欠陥修正コミットと欠陥報告の対応付けが不適切である可能性が高い．

[*7] なぜなら，Bugzilla において深刻度が‘enhancement’に設定されている場合，それは開発者に対する機能の追加や改善の依頼を意味するため，対応するコミットは欠陥修正コミットでないと考えられるからである．

[*8] なお，Diff コマンドの出力結果のうち，r と $r-1$ の間で変更または削除された行のみを欠陥とみなし，新しく追加された行は欠陥の直接的な原因でないと考えて調査対象から除外する．また，全体がコメント文のみで構成されている行も調査対象から除外する．

在するファイルのうち，上側および下側に欠陥が存在するコードクローンを調査対象とし，それらのコードクローンと最近傍欠陥の距離を求める．本研究ではコードクローンから 101 行以上離れたコード片でコードクローンに起因する欠陥が混入する可能性は低いと考え，コードクローンからの距離が 100 行以内のコード片のみを調査対象とする．そして，コードクローンとの距離ごとにコード片の欠陥混入数を求め，コードクローンに近いコード片と遠いコード片において欠陥混入数に差があるかどうかを検定する．コードクローンに近いコード片と遠いコード片を区別する際は，それぞれの行数の比が 10 対 90，20 対 80，30 対 70，40 対 60，50 対 50 になるように分割した．例えば，コード片を 10 対 90 に分割する場合，コードクローンとの距離が 1 行から 10 行までのコード片をコードクローンから近いコード片，コードクローンとの距離が 11 行から 100 行までのコード片をコードクローンから遠いコード片とする．このようにして調査対象のコード片を分割した後，コードクローンに近いコード片と遠いコード片の欠陥混入数に統計的有意差があるかどうかを検定する．なお，この調査ではコードクローンの上側と下側について別々に調査を行う．

4. 調査結果

4.1 調査対象

本研究では，3 章で示した調査手法に基づいて OSS を対象とした定量的調査を行った．対象のプロジェクトは，Gimp，Evolution，Nautilus，Apache httpd，Gedit の 5 プロジェクトである．全てのプロジェクトは C 言語で開発されており，このうち Gimp，Evolution，Nautilus，Apache httpd の 4 プロジェクトは，Rahman らの調査対象と同様である [12], [15]．表 1 に調査対象プロジェクトについてまとめる．表 1 におけるファイル数および LOC とは，それぞれ 2014 年 6 月のステージングスナップショットにおける C ソースファイル (拡張子が ‘.c’ のファイル) の総数および総行数を示しており，欠陥報告数は対象期間中に Bugzilla で報告された欠陥修正の数を示している．対象期間は，各プロジェクトにおいて最初に C ソースファイルが作成された日から 2014 年 6 月までとする．

本研究ではソースコードの自動生成ツールによって生成されたファイル（以降，自動生成ファイルとする）を調査対象から除外した．具体的には，各ソースコードの先頭から 100 行目までに ‘generated’ という単語が存在するファイルを全て抽出し，抽出したそれぞれのファイルが自動生成ファイルであるかどうかを手作業で確認した．‘generated’ という単語の存在を確認した理由は，自動生成ファイルに ‘/* This file is generated by ... */’ といったコメント文が存在する場合が多いためである．

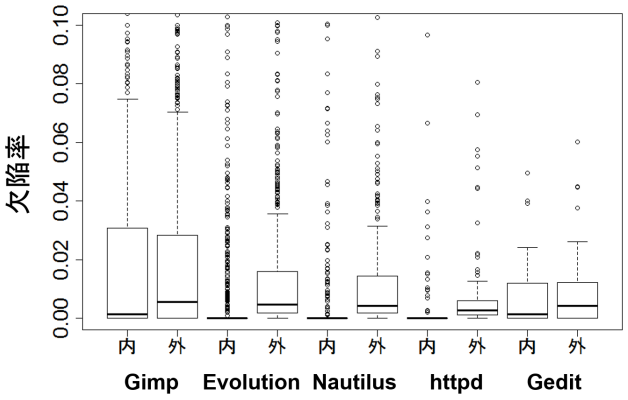


図 5 コードクローン内外における欠陥率の分布（DECKARD, Conservative）

Fig. 5 Distribution of defect rate within cloned code fragment and non-cloned code fragment. (DECKARD, Conservative)

4.2 調査 1：コードクローン内外における欠陥率の比較

3.2.1 項で述べた手法に従い，各プロジェクトにおいてコードクローン内外の欠陥率を比較した．この調査では，コードクローン検出ツールの設定の違いが調査結果に与える影響も検証するため，DECKARD の設定は Conservative と Liberal の 2 通りで調査を行った．表 2 は，それぞれの設定における調査対象プロジェクトの詳細である．表 2 におけるファイル数とは，欠陥とコードクローンの両方が存在するファイルのうち自動生成ファイルを除外したファイルの数を表している．スナップショット数とは，対象ファイルが存在するステージングスナップショットの総数を表し，欠陥修正コミット数とは各ステージングスナップショットに対応する欠陥修正コミットの総数を表している．

図 5 は，DECKARD の設定を Conservative とした場合のコードクローン内外の欠陥率の分布を示している．縦軸は欠陥率を表しており，それぞれの箱ひげ図はプロジェクトごとのコードクローン内外における欠陥率の分布を示している．図 5 より，全プロジェクトにおいてコードクローン内の中央値よりもコードクローン外の中央値の方が高い傾向が見られた．表 3 は，コードクローン内外の中央値に統計的有意差があるかどうかをプロジェクトごとに検定した結果である．Kolmogorov-Smirnov 検定（以降，K-S 検定とする）を行った結果，コードクローン内外の欠陥率の分布に正規性が認められなかったため，有意差の検定には Wilcoxon の順位和検定を用いた．表 3 において p 値が黄色くハイライトされているプロジェクトは，Wilcoxon の順位和検定で統計的有意差が確認されたプロジェクトである（以降で示す検定結果の表においても同様である）．表 3 より，Gimp，Evolution，Nautilus，Apache httpd の 4 プロジェクトにおいて有意水準 5%で統計的有意差が確認された．つまり，DECKARD の設定が Conservative の場

表 1　調査対象プロジェクトの概要
Table 1　Investigated projects.

	ファイル数	LOC	欠陥報告数	対象期間
Gimp	1539	870K	4253	1997 年 11 月〜2014 年 6 月
Evolution	753	499K	17485	1998 年 1 月〜2014 年 6 月
Nautilus	141	122K	4408	1997 年 1 月〜2014 年 6 月
httpd	291	235K	2317	1999 年 7 月〜2014 年 6 月
Gedit	95	65K	2065	1998 年 4 月〜2014 年 6 月

表 2　調査対象ファイルの詳細 (DECKARD)
Table 2　Details of studied files within each project.(DECKARD)

	ファイル数		スナップショット数		欠陥修正コミット数		コードクローン数	
	Conservative	Liberal	Conservative	Liberal	Conservative	Liberal	Conservative	Liberal
Gimp	1104	1156	138	138	627	653	72529	107272
Evolution	838	911	99	100	498	538	27152	46600
Nautilus	371	418	105	106	328	349	9057	16254
httpd	113	122	61	64	99	106	4185	7308
Gedit	75	90	42	48	62	76	4998	7206

表 3　コードクローン内外における欠陥率の差の検定結果（DECKARD, Conservative）
Table 3　Statistical significance between defect rates within cloned code fragment and non-cloned code fragment.（DECKARD, Conservative）

	中央値（内）	中央値（外）	p 値
Gimp	0.00141	0.00549	4.71e-06
Evolution	0	0.00475	< 2.20e-16
Nautilus	0	0.00420	< 2.20e-16
httpd	0	0.00269	< 2.20e-16
Gedit	0.00126	0.00427	1.84e-01

表 4　コードクローン内外における欠陥率の差の検定結果（DECKARD, Liberal）
Table 4　Statistical significance between defect rates within cloned code fragment and non-cloned code fragment.（DECKARD, Liberal）

	中央値（内）	中央値（外）	p 値
Gimp	0.00358	0.00558	2.35e-03
Evolution	0	0.00495	< 2.2e-16
Nautilus	0	0.00437	< 2.2e-16
httpd	0	0.00283	3.43e-14
Gedit	0	0.00593	3.05e-02

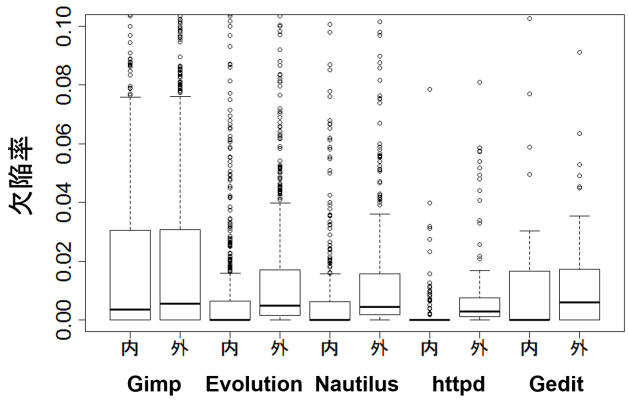


図 6　コードクローン内外における欠陥率の分布（DECKARD, Liberal）
Fig. 6　Distribution of defect rate within cloned code fragment and non-cloned code fragment.（DECKARD, Liberal）

合，この 4 プロジェクトではコードクローン外の欠陥率がコードクローン内の欠陥率よりも高いといえる．

図 6 に，DECKARD の設定を Liberal にした場合のコードクローン内外の欠陥率の分布を示す．図 6 においても，全プロジェクトにおいてコードクローン内の中央値よりもコードクローン外の中央値の方が高い傾向が見られた．そこで，コードクローン内外の中央値に統計的有意差があるかどうかを検定した結果を表 4 に示す．K-S 検定の結果，コードクローン内外の欠陥率の分布に正規性が認められなかったため，有意差の検定には Wilcoxon の順位和検定を用いた．表 4 より，全てのプロジェクトにおいて有意水準 5%で統計的有意差が確認された．したがって，DECKARD の設定が Liberal の場合，全プロジェクトでコードクローン外の欠陥率がコードクローン内の欠陥率よりも高いといえる．

次に，コードクローン検出ツールの違いが調査結果に与える影響を検証するため，CCFinder を用いて同様の調査を行った．その結果，DECKARD を用いた場合と同様に全プロジェクトでコードクローン内の中央値よりもコードクローン外の中央値の方が高い傾向が見られ，有意水準 5%で統計的有意差も確認された．つまり，CCFinder を用いた場合も全プロジェクトでコードクローン外の欠陥率がコードクローン内の欠陥率よりも高いといえる．なお，検定結果に関する詳細なデータは文献 [19] を参照されたい．

表 5　コードクローンから近いコード片と遠いコード片における欠陥混入数の差の検定結果
Table 5　Statistical Significance between defects' occurring within code fragment near code clone and far from code clone.

	上側 (p 値)					下側 (p 値)				
	10：90	20：80	30：70	40：60	50：50	10：90	20：80	30：70	40：60	50：50
Gimp	1.31e-03	9.69e-08	8.99e-12	6.79e-16	2.04e-15	8.88e-06	1.07e-07	1.02e-10	3.44e-12	3.10e-13
Evolution	3.53e-02	3.43e-04	7.51e-04	1.01e-02	1.53e-05	4.41e-05	4.29e-03	9.82e-04	1.99e-04	1.07e-02
Nautilus	3.29e-05	1.52e-09	1.47e-13	4.25e-13	1.95e-09	1.10e-03	9.66e-07	7.08e-09	1.38e-08	1.31e-09
httpd	2.47e-01	1.24e-01	6.75e-01	6.95e-01	6.75e-01	4.70e-02	5.90e-01	7.74e-01	7.29e-01	9.50e-01
Gedit	7.07e-01	3.41e-01	6.67e-02	3.88e-03	1.41e-03	8.72e-03	1.37e-01	6.12e-01	5.23e-01	7.65e-01

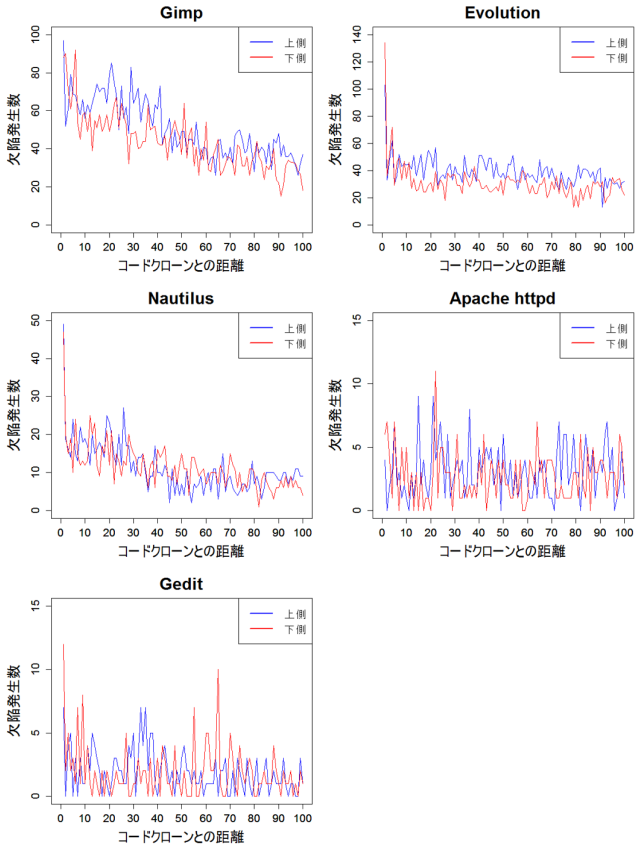


図 7　コードクローンとの距離に基づく欠陥混入数の分布
Fig. 7　Distribution of defects' occurring within in non-cloned code fragment based on its distance to the code clone.

4.3　調査 2：コードクローンとの距離に基づく欠陥混入数の比較

3.2.2 項で述べた手法に従い，コード片の欠陥混入数をコードクローンとの距離ごとに調査した．調査対象は，調査 1 で DECKARD の設定を Conservative とした場合の調査対象ファイル（表 2）に存在するコードクローンのうち，その上側または下側に欠陥が存在するコードクローンである．なお，コードクローンの上側と下側についてはそれぞれ分けて調査を行った．図 7 に，コードクローンとの距離ごとにコード片の欠陥混入数を示す．各グラフの横軸はコードクローンとの距離，縦軸は欠陥混入数を表しており，青いグラフがコードクローンの上側，赤いグラフがコードクローンの下側を示している．本研究では，コードクローンからの距離が 100 行を超えるコード片にコードクローンに起因する欠陥が存在する可能性は低いと考え，コードクローンからの距離が 100 行以内のコード片を調査対象とした．図 7 では，Gimp，Evolution，Nautilus，Gedit の 4 プロジェクトにおいて，コードクローンとの距離が大きくなるにつれて欠陥混入数が減少する傾向が見られた．そこで，コードクローンから近いコード片と遠いコード片において欠陥混入数に統計的有意差があるかどうかを Wilcoxon の順位和検定を用いて検定した．調査対象のコード片をコードクローンから近いコード片と遠いコード片に分割する際は，それぞれの行数の比が 10 対 90，20 対 80，30 対 70，40 対 60，50 対 50 になるように分割し，それぞれの場合で検定を行った．例えば分割比が 10 対 90 の場合，コードクローンとの距離が 1 行から 10 行までのコード片と 11 行から 100 行までのコード片に統計的有意差があるかどうかを検定した．検定結果を表 5 に示す．表 5 の各値は検定における p 値を示しており，統計的有意差が存在する場合は黄色でハイライトされている．表 5 より，Gimp，Evolution，Nautilus の 3 プロジェクトにおいて統計的有意差が確認された．具体的には，この 3 プロジェクトの上側と下側の全ての分割パターンにおいてコードクローンに近いコード片の欠陥混入数が遠いコード片の欠陥混入数よりも高く，統計的有意差が存在した．また Gedit では，上側の 40 対 60，50 対 50 および下側の 10 対 90 の分割パターンにおいて統計的有意差が存在し，コードクローンから近いコード片の方が遠いコード片よりも欠陥混入数が高かった．一方 Apache httpd では，ほぼ全ての分割パターンにおいて統計的有意差が存在しなかった．

5.　考察

5.1　調査結果 1：コードクローン内外における欠陥率の比較

調査 1 の結果，全てのプロジェクトにおいてコードクローン外の欠陥率がコードクローン内の欠陥率より高く、そのうち Gimp，Evolution，Nautilus，Apache httpd の 4 プロジェクトにおいて統計的有意差が存在した．また，

DECKARD の設定を Liberal にした場合と CCFinder を用いた場合は Gedit においても統計的有意差が存在した．したがって，コードクローン内外ではコードクローン外の方が欠陥率が高く，コードクローン検出ツールやその設定の差異が調査結果に与える影響は小さいと考えられる．

コードクローン内外の欠陥密度を比較した Sajnani らの既存研究 [13] においても同様の傾向が確認されていることから，一般的にコードクローン内よりもコードクローン外の方が欠陥率が高い傾向にあると考えられる．その理由として，コピーアンドペーストによって再利用されるコード片は過去に十分なテストが行われたものや熟練者によって開発された高品質なプログラムであることが多く，その結果としてコードクローン内の欠陥率が低下した可能性が挙げられる．

以上より、RQ 1 について、コードクローン内外ではコードクローン外の方が欠陥率が高い傾向があり，開発者はコードクローンに着目してコードクローン外のソースコードを優先的に検査することで効率的に欠陥を見つけられると考えられる．ただし、Gedit のようにコードクローン内外の欠陥率に統計的有意差が存在しないプロジェクトも存在するため、全てのプロジェクトにおいて欠陥の検査を効率化できるとはいえない．

5.2 調査結果 2：コードクローンとの距離に基づく欠陥混入数の比較

調査 2 の結果，Gimp，Evolution，Nautilus，Gedit の 4 プロジェクトにおいてコードクローンに近いコード片の欠陥混入数が遠いコード片の欠陥混入数よりも高く，統計的有意差が存在した．すなわち，この 4 プロジェクトではコードクローンに近いコード片に欠陥が多く存在し，コードクローンから遠ざかるにつれて欠陥混入数が減少する傾向があるといえる．特に Gimp，Evolution，Nautilus の 3 プロジェクトでは，全ての分割パターンで欠陥混入数に統計的有意差が存在したため，この傾向が強いと考えられる．Gedit では，前述の 3 プロジェクトのように全ての分割パターンに統計的有意差が存在することはなかったが，対象範囲をおおよそ半分に分割するパターンにおいて統計的有意差が存在したため，コードクローンから遠ざかるにつれて欠陥混入数が減少していると考えられる．Gedit の一部の分割パターンにおいて統計的有意差が存在しなかった理由として，調査対象のコードクローン数が他のプロジェクトに比べて著しく少ないことが挙げられる．つまり，コードクローンから遠ざかるにつれて欠陥混入数が減少する傾向は存在するが，調査対象のコードクローンが少なく欠陥混入数に統計的有意差が現れなかった可能性がある．ゆえに，もし Gedit において上側および下側に欠陥が存在するコードクローンの情報を更に取得することができれば，Gimp，Evolution，Nautilus と同様に統計的有意差が確認される可能性があると考えられる．一方で，Apache httpd では前述のような統計的有意差は存在せず，一部の分割パターンではコードクローンから遠いコード片の方が欠陥混入数が多い傾向も確認された．したがって，コードクローンに近いコード片において欠陥が多く混入する傾向は，すべてのプロジェクトにおいて一般的に見られるものではないといえる．

以上より，RQ2 について，コードクローン外のコード片ではコードクローンに近いコード片で欠陥が多く混入するため，開発者はコードクローンに近いコード片を優先的に検査することで効率的に欠陥を探すことができると考えられる．

5.3 妥当性について

本研究で行った定量的調査の妥当性について以下に考察する．

第 1 に，調査 2 ではコードクローン外の欠陥として最近傍欠陥のみに着目しているため，調査対象になっていない欠陥が存在する点が挙げられる．本研究では，コードクローンに関連する欠陥がコードクローンから離れたコード片に存在する可能性は低いと考え，コードクローンの最近傍にある欠陥のみを対象とした．また，本研究における欠陥とは欠陥修正コミットで修正された各コード片であり，1 つの欠陥修正コミットに複数の修正箇所が存在する場合はそれぞれを別の欠陥として扱っている．ゆえに，全ての欠陥を対象として欠陥混入数を比較すると，各欠陥修正コミットにおける修正箇所の数が調査結果に影響を与えると考えられるため最近傍欠陥のみに着目した．しかし，本研究ではステージングスナップショットに対して調査を行っているため，1 つのソースコードに複数の欠陥修正コミットに関する欠陥が存在する場合，調査の対象にならない欠陥修正コミットが存在することになる．したがって，今後はステージングスナップショットを用いず，欠陥修正コミットごとに調査を行う必要があると考えられる．

第 2 に，調査結果がコードクローン検出ツールの違いや検出ツールの設定の違いに影響する可能性が挙げられる．本研究ではコードクローン検出ツールとして DECKARD と CCFinder を用いたが，これら以外のツールを用いて同様の調査を行うと結果が本研究とは異なる可能性がある．また，DECKARD の設定には本研究で用いた 2 つの設定以外にも様々なパターンが考えられるため，その設定の違いが結果に影響を与える可能性も考えられる．しかし，調査 1 で行った DECKARD および CCFinder を用いた調査では，すべてにおいて共通の結果が得られたため，コードクローン検出ツールの違いやその設定の違いが調査結果に与える影響は小さいと考えられる．一方で，調査 2 ではこのような検証を行っておらず，本研究の全体を通してコードクローン検出ツールの違いやその設定の違いの影響が小

さいとはいえないため，調査 2 においても検証する必要があると考えられる．

6. おわりに

本研究では，OSS のソースコードにおけるコード片の欠陥混入傾向をコードクローンとの位置関係に基づいて定量的に調査した．コードクローン内外の欠陥率を比較した結果，5 プロジェクト中 4 プロジェクトにおいてコードクローン外の方が欠陥率が高い傾向を確認した．また，コード片の欠陥混入数をコードクローンとの距離ごとに比較した結果，5 プロジェクト中 4 プロジェクトにおいて，コードクローンに近いコード片で欠陥が多く混入する傾向が見られた．したがって，開発者がコードクローンに着目して欠陥の検査を行う際，コードクローンの外かつ近辺のコード片から優先的に検査することで効率的に欠陥を探すことができると考えられる．しかし，両調査において前述の傾向が確認されなかったプロジェクトも存在したことから，一般的に全てのプロジェクトで同様の傾向が存在するとは言えず，プロジェクトごとに本研究の調査手法を用いた欠陥混入傾向の確認が必要であると考えられる．

今後の課題は，コードクローンとの距離に着目した欠陥混入数の調査において，コードクローン検出ツールの違いが結果に与える影響の検証および欠陥修正コミットごとの調査を行い，調査結果の信頼性を確認することである．

謝辞　本研究は, JSPS 科研費 26730036 の助成を受けた．

参考文献

[1] 肥後芳樹，吉田則裕：コードクローンを対象としたリファクタリング，コンピュータソフトウェア，Vol. 28, No. 4, pp. 43–56 (2011).
[2] Kapser, C. J. and Godfrey, M. W.: “Cloning Considered Harmful”Considered Harmful: Patterns of Cloning in Software, *Empirical Software Engineering*, Vol. 13, No. 6, pp. 645–692 (2008).
[3] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670 (2002).
[4] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, *Proc. ICSE*, pp. 96–105 (2007).
[5] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Trans. Softw. Eng.*, Vol. 32, No. 3, pp. 176–192 (2006).
[6] Baxter, I. D., Yahin, A., Moura, L., Sant’Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. ICSM*, pp. 368–377 (1998).
[7] Komondoor, R. and Horwitz, S.: Using Slicing to Identify Duplication in Source Code, *Proc. SAS*, pp. 40–56 (2001).
[8] Chatterji, D., Carver, J., Kraft, N. A. and Harder, J.: Effects of cloned code on software maintainability: A replicated developer study, *Proc. WCRE*, pp. 112–121 (2013).
[9] Jiang, L., Su, Z. and Chiu, E.: Context-based Detection of Clone-related Bugs, *Proc. ESEC-FSE*, pp. 55–64 (2007).
[10] Xie, S., Khomh, F. and Zou, Y.: An Empirical Study of the Fault-proneness of Clone Mutation and Clone Migration, *Proc. MSR*, pp. 149–158 (2013).
[11] Juergens, E., Deissenboeck, F., Hummel, B. and Wagner, S.: Do Code Clones Matter?, *Proc. ICSE*, pp. 485–495 (2009).
[12] Rahman, F., Bird, C. and Devanbu T., P.: Clones: What is that smell?, *Proc. MSR*, pp. 72–81 (2010).
[13] Sajnani, H., Vaibhav, S. and Cristina V., L.: A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code, *Proc. SCAM*, pp. 21–30 (2014).
[14] Sajnani, H. and Cristina V., L.: A parallel and efficient approach to large scale code clone detection, *Proc. IWSC*, pp. 46–52 (2013).
[15] Rahman, F., Bird, C. and Devanbu, P.: Clones: What is That Smell?, *Empirical Software Engineering*, Vol. 17, No. 4-5, pp. 503–530 (2012).
[16] Hovemeyer, D. and Pugh, W.: Finding Bugs is Easy, *SIGPLAN Not*, Vol. 39, No. 12, pp. 92–106 (2004).
[17] Śliwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, *Proc. MSR*, pp. 1–5 (2005).
[18] Bachmann, A. and Bernstein, A.: Data Retrieval, Processing and Linking for Software Process Data Analysis, Technical Report IFI-2009.0003b, Department of Informatics, University of Zurich (2005).
[19] 中山直輝：コードクローンとの位置関係がコード片の欠陥発生傾向に与える影響の調査，奈良先端科学技術大学院大学情報科学研究科 修士論文, NAIST-IS-MT1351080, (オンライン), 入手先 ⟨http://sdlab.naist.jp/pman3/pman3.cgi?DOWNLOAD=117⟩ (2015).