社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

コードクローンの長さとソフトウェア信頼性の関係の分析

左藤 裕紀† 亀井 靖高† 上野 秀剛† 門田 暁人†
川口 真司† 名倉 正剛† 松本 健一† 飯田 元‡

†‡奈良先端科学技術大学院大学情報科学研究科 〒 630 - 0192 奈良県生駒市高山町 8916-5

E-mail: †{hiroki-sa, yasuta-k, hideta-u, akito-m, kawaguti, nag, matumoto}@is.naist.jp, ‡iida@itc.naist.jp

あらまし ソースコード中の重複コード列であるコードクローンによって，ソフトウェアの保守工数が増大するといわれている．一方で，コードクローンがソフトウェアの信頼性に与える影響については明確にはわかっていない．本稿では，コードクローンとソフトウェアの信頼性の関係を分析するために，コードクローンの長さに着目した．開発プロジェクトのリポジトリを分析し，ソースコードファイルに含まれるコードクローンの長さによってファイルを分類し，それぞれのファイル群のバグ密度を計測した．その結果，短いコードクローンを含んでいるファイル群のバグ密度は大きく，長いコードクローンを含んでいるファイル群のバグ密度は小さいことがわかった．さらに各ソースコードの長さにバグ密度が影響を受けると考え，ソースコード行数にしたがってファイルを分類し，それぞれのファイル群においてバグ密度がどのように変化するのかを調べた．その結果，ソースコード行数が比較的長い場合において前述の傾向がより強くなることがわかった．

キーワード コードクローン，コードクローンの長さ，ソフトウェア信頼性，バグ密度

An analysis of relationship between code clone length and software reliability

Hiroki SATO† Yasutaka KAMEI† Hidetake UWANO† Akito MONDEN†
Shinji KAWAGUCHI† Masataka NAGURA† Ken-ichi MATSUMOTO† Hajimu IIDA‡

†‡Nara Institute of Science and Technology 8916-5 Takayama, Ikoma-shi, Nara, 630-0192 Japan

E-mail: †{hiroki-sa, yasuta-k, hideta-u, akito-m, kawaguti, nag, matumoto}@is.naist.jp, ‡iida@itc.naist.jp

Abstract A code clone that is duplicated code section in source files makes software maintenance more difficult. However, to our knowledge, no study has analyzed how the code clone influences software reliability. In this paper, we focused on the length of the code clone in order to analyze the relation between software reliability and code clone. We analyzed a repository of a project, classified source code files according to the length of the code clone included in the files, and measured the bug density of each classified file group. As a result, it was clarified that (1) the bug density of the file group that contained a short code clone was high, and (2) the bug density of the file group that contained a long code clone was low. In addition, because we had thought the bug density was influenced from the length of each source code file , we classified the files by measuring lines of source code and examined how much the bug density of each file group changed. We observed that the result (1), (2) became more pronounced when lines of source code was comparatively large.

Keyword Code clone, Length of code clone, Software reliability, Bug density

1. まえがき

近年，ソフトウェアの大規模化に伴ってソフトウェア保守の重要性が高まっている．ソフトウェア保守を困難にする要因の一つとしてコードクローンが注目されている[9]．コードクローンとはソースコード中に含まれる重複，または類似したコード断片のことである．コードクローンはソフトウェアの構造を複雑にし，ソフトウェア保守性を低下させる一因であるといわれている[3][6]．コードクローンを含むソースコードに不具合が見つかった場合，関係する他の全てのコードクローンを調査し，それぞれに対して同様の修正を施すことになる可能性が高いためである．

一方で，コードクローンがソフトウェアの信頼性に与えている影響は明らかではない．例えば，優良なコードをコピーすることによって発生したコードクローンはソフトウェアの信頼性を向上させることがある．一方で，不良なコードをコピーすることによって発生したコードクローンはソフトウェアの信頼性を低下させる可能性がある．

ソフトウェアの信頼性に影響するコードクローンに関するメトリクスとしては，コードクローンの長さや数などが考えられる．本研究では，コードクローン

の長さに着目し，ソフトウェアの信頼性にどのような影響を与えているか調査する．そして，コードクローンの長さとソースコードに含まれているバグの数を分析し，どの程度の長さのコードクローンを含んでいるファイルにバグが多いのかを明らかにする．

まず，長いコードクローンを含むソースコードはバグを含む可能性が低い，という仮説を立てた．そして仮説を検証するため，ケーススタディとして Eclipse プロジェクトを対象に，ソースコードとバグ履歴を分析した．

2. コードクローンとその検出方法

2.1. コードクローンの定義

コードクローンとはソースコード中の重複したコード列のことである．これはソフトウェアの開発中や保守作業中に，次の要因によって生じる[3].

- コード列のコピー&ペースト
- コードジェネレータによるコード生成
- 特定のコーディングスタイルの利用
- パフォーマンスを向上させるための，意図的なコード記述の繰り返し
- 偶然の一致

コピー&ペーストによって作られた場合には，コピー後のコード列に部分的な変更を加えることも多く，一般的にそのような変更を加えられた類似コード列の組も，コードクローンと見なす.

コードクローンの多くはデバッグや機能拡張の際に同時に更新しなければならず，ソフトウェアの保守工数を増大させている．このため，保守工程においてはコードクローンを検出することが望ましい.ただし，どのようなコード列をコードクローンと見なすかの定義は，検出方法やツールごとに異なる[1][2][3][9].

2.2. コードクローン検出方法

コードクローン検出方法はいくつか提案されているが[3][9]，本節では，神谷らが提案したトークンベースのコードクローン検出方法について述べる[9]．この方法はプログラミング言語の構文規則に基づき，ソースコードをトークン列に変換し，比較する．これによって，コード列中の空白やコメント，インデントが異なったり，あるいは変数名等が書き換えられた場合でも，コードクローンを抽出することができる．この方法を実装したツール CCFinder は Java や C++等のプログラミング言語で記述されたソフトウェアからコードクローンを検出できる．トークンベースのコードクローン検出手順の概要を次に示す.

(i) 字句解析

入力として与えられたソフトウェアに含まれる全てのソースコードを，プログラミング言語の構文規則に従ってトークンに分割する．ソースコード中の空白やコメントは無視される.

(ii) トークン変換

型，変数，定数に属するトークンはそれぞれ同一のトークンに置き換えられる．この置き換えにより，型名，変数名，定数だけが異なるコード列の組をコードクローンとして検出することができる.

(iii) マッチングおよびフォーマッティング

変換後のトークン列に含まれる全ての部分列の集合から，同一の部分列の組を探し出してコードクローンとして検出する．検出にあたっては Suffix Tree マッチングアルゴリズム[7]を用いることで，ソフトウェアのサイズ n に対して $O(n)$の計算量で検出できる．最後に，検出されたトークン列の位置情報を元のソースコード上の行番号に変換し，コードクローンとして出力する.

3. コードクローンの長さとソフトウェア信頼性の関係の分析

3.1. コードクローンのソフトウェア信頼性への影響

2.1節で述べたように，一般的にコードクローンによってソフトウェアの保守工数が増大するといわれている．しかし，コードクローンの存在がソフトウェアの信頼性を低下させるとは，一概に言い切れない.

例えば，優良なコードをコピーすることによって発生したコードクローンはソフトウェアの信頼性を向上させることがある．逆に，不良なコードをコピーすることによって発生したコードクローンは，ソフトウェアの信頼性を低下させる.

このようにソフトウェアの信頼性への影響は，どのようなコードをどれだけコピーしてコードクローンを生み出したかに起因する．しかしソースコード中にコードクローンが含まれることによるソフトウェア信頼性への影響については，明確にはわかっていない．これらの関係を導くことができれば，コードクローン含有の状況をソフトウェア信頼性の評価のための指針として利用できる.

3.2. 仮説

我々は，コードクローンの長さ，数，複雑さなどのメトリクスを計測することにより，コードクローンによるソフトウェア信頼性への影響を把握することが可能であると考える．本研究では，コードクローンの長さに着目し，ソフトウェア信頼性にどのような影響を与えるか調査する．まず本節では，ソースコードに含まれるコードクローンの長さとバグの関係について考える.

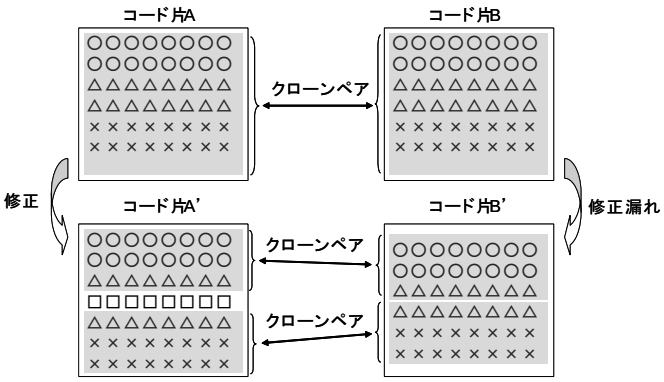

図 1 修正漏れによる短いコードクローンの発生例

いま，複数のソースコード中に存在する長いコードクローンを修正する場合を考える．すべてのソースコードに対して変更が漏れなく行われていれば，修正後の各ソースコードには，長いコードクローンが存在し続ける．もしそうでなければ，修正漏れにより，複数の短いコードクローン片に分割される．分割されたそれぞれのコードクローン片は，コピー元のコードの一部分に対しての，コードクローンになる．その様子を図 1 に示す．

このように，長いコードクローンを修正することにより，複数の短いコードクローンが発生することがある．従って，もし長いコードクローンがそのままソースコード中に存在するのならば，コードクローンの部分のソースコードを変更していないか，あるいはすべてのソースコードのコードクローンに対して変更が漏れなく行われているかのいずれかであるといえる．前者はソースコードに不備がなく変更する必要がない場合であり，後者は変更への対応が十分に行われている場合であると考えることができる．以上より，次の仮説を導く．

(仮説) 長いコードクローンを含むソースコードは，短いコードクローンを含むソースコードよりも，バグを含む可能性が低い．

本研究では，この仮説を検証するために，実際の開発プロジェクトを分析した．次節では，分析の方針について述べる．

3.3. 分析方針

前節で述べた仮説の検証のために行った，開発プロジェクト分析の方針を示す．

まず，(1) 開発プロジェクトのある時点での全ソースコードを分析し，コードクローンを検出する．そしてどのような長さのコードクローンを含むかによって，それぞれのファイルを分類する．次にその時点以降に，(2) 開発プロジェクト全体，および (1) で分類したファイル群のそれぞれに起因するバグがどの程度含まれているかを算出する．最後に (1) と (2) の結果を比較し (3) ソースコードに含まれるコードクローンの長さと，バグ密度を比較する．それぞれの詳細を，次に示す．

(1) コードクローンの検出とコードクローンの長さによるファイルの分類

ある時点での開発プロジェクト全体のソースコードを分析し，コードクローンを検出する．コードクローン検出には，2.2 節で述べたトークンベースによるコードクローン検出方法を利用する．この方法ではコードクローンとして認識する長さの範囲を指定することにより，設定した範囲の長さのコードクローンが検出できる．ここでは，比較的短い範囲から長い範囲まで連続した複数の範囲を設定し，それぞれの範囲ごとにコードクローンを検出する．そして各範囲において，全ソースコードのうちのいくつのファイルからコードクローンが検出されるかを計測する．なお，後述の (2) において範囲ごとにバグの発生頻度を算出するが，範囲ごとにコードクローンの検出されるファイル数に大きな差異があると，バグの発生頻度の比較が困難になる．そのため検出されるファイル数が極端に少なくならないように，コードクローンの長さの範囲を調節する．

(2) バグ含有状況の分析

開発プロジェクト全体のソースコードの単位行数あたりにバグが含まれる割合を算出する．

まず，開発プロジェクト全体のソースコードの総行数を求める．各ソースファイルにはコメント文や空白行が含まれるが，これらをソースコードの行数から除外する．本稿では，このように計測した各ファイルのソースコード行数を，SLOC と呼び，開発プロジェクトに含まれる全ソースコードについて SLOC の合計 (これを総 SLOC と呼ぶ) を求める．

次に，開発プロジェクトのバグ情報を調べる．この際に，(1) でコードクローンを検出した時点以降で発見されたバグの件数を計測する．

そして，これらの値から，ソースコードの単位行数あたりにバグが含まれる割合を算出する．具体的には，総 SLOC の 1000 行あたりにバグを何件含むかを算出する．これをバグ密度と呼び，次の式で表す．

$$\text{バグ密度} = \frac{\text{全ソースファイルのバグ数}}{\text{全ソースファイルの総SLOC}} \times 1000$$

またさらに (1) で分類したファイル群のそれぞれに起因するバグがどの程度含まれているかを，同様に

表 1 各範囲の設定によりコードクローンが検出されるファイル数

コードクローン長 (トークン)	20~29	30~39	40~49	50~59	60~69	70~79	80~89	90~109	110~139	140~189	190~
ファイル数 (ファイル)	6869	5865	4333	3285	2339	1689	1367	1501	1410	1079	1034

算出する．これは，(1) で分類したそれぞれのファイル群に対して，次の方法で計測する．

まず, 対象とするファイル群の SLOC の合計を算出する．次にバグ情報を調べ，それぞれのバグがどのソースファイルに起因するものかを分類し，対象とするファイル群に起因するバグ数を計測する．そしてそれらの値を利用し，バグ密度を算出する．バグ密度の算出式は次のようになる．

$$\text{バグ密度} = \frac{\text{対象とするファイル群に起因するバグ数}}{\text{対象とするファイル群のSLOCの合計}} \times 1000$$

(3) ソースコードに含まれるコードクローンの長さとバグ密度の比較

(1) と (2) の結果を比較することで，ソースコードに含まれるコードクローンの長さによって，バグ密度がどのような変化を受けるのかを調べる. これにより，ソースコードに含まれるコードクローンの長さとバグの関係を比較することができる．

なお，(1)では，コードクローンとして認識する長さの範囲として複数の範囲を指定する．指定する長さが長い場合は，長いコードクローンが検出されるので，SLOC が小さいソースファイルからはコードクローンを検出しにくくなる．

また，バグ密度は前述の算出式のように，含有されるバグ数を，対象とするソースファイル群の SLOC で除算したものである．したがって 1 つのバグによる影響の大きさが，SLOC の大きさによって異なる．

ソースコードに含まれるコードクローンの長さとバグを含む関係を論じる上では，このような SLOC の大きさによる影響を考慮する必要がある．そこでさらに (3) において，開発プロジェクトに含まれる全ソースファイルを SLOC の大きさごとに分類し，ソースコードに含まれるコードクローンの長さとバグ密度の関係を比較する．

本研究ではケーススタディとして，このような方針に基づいてオープンソースソフトウェアの開発プロジェクトを分析し，3.2 節に述べた仮説を検証した．次章ではこのケーススタディについて述べる．

4. ケーススタディ

4.1. 対象データ

オープンソースの統合ソフトウェア開発環境 Eclipse の開発プロジェクトを対象に，仮説を検証するため，バージョン 3.0 のコードクローン含有状況と，バージョン 3.0 の時点のソースコードに起因したバグ含有状況を分析した．このバージョン 3.0 は 2004 年 6 月 25 日にリリースされ，10635 個の Java ファイルで構成されている．

4.2. 分析手順

本節では，3.3 節に示した分析方針に基づき行った分析手順について述べる．

(1) コードクローンの検出とコードクローンの長さによるファイルの分類

CCFinder[5]を用いて，バージョン 3.0 時点でのソースコードからコードクローンを検出した. なお, 3.3 節 (1) で述べたように，コードクローンとして認識する長さの範囲をトークン単位で設定したが，この際に検出されるファイル数が極端に少なくならないように調整した．具体的には，それぞれの範囲の設定によりコードクローンが検出されるファイル数が 1000 ファイル以上になるように調整した．その結果，表 1 に示す 11 区間に分類した．

(2) バグ含有状況の分析

まず，Eclipse のバージョン 3.0 時点での 10635 個の Java ファイルに対して，SLOC を計算した．この計算には，CCFinder の機能を用いた．なお，CCFinder は，この計算の際に，Java 言語における宣言部分を，コードクローン解析対象外として除外する．

次に，開発プロジェクトのバグ情報を調べた．本稿では，Gyimothy らが行った方法[8]で，各バージョンの各ファイルにおいてバグの有無を調べた．

まず, Eclipse プロジェクトが運用している障害管理ツール Bugzilla[4]から，バグ報告とそれに対応する修正履歴を収集した．次に，バグの報告がどのファイルに対して行われたものかを特定するために，バグ対応時に修正パッチが適用されたファイル名を，収集した修正履歴から取得した．そして，どの時期にバグが含まれていたのかを特定するために，バグが報告された日と，修正パッチが適用された日を取得した．これにより，全バージョンに対しての報告と，修正が行われた日時を，障害管理ツールの履歴情報から抽出することができた．最後に， バージョン 3.0 のリリースされた日時以降に，バージョン 3.0 を対象に報告，修正されたバグをバージョン 3.0 に起因するバグであるとみ

なして抽出した．

上記手順によって各ファイルに適用された修正パッチの数を求め，バグ数とした．そして，3.3節(2)の式を用いて各ファイルのバグ密度を求めた．

(3) ソースコードに含まれるコードクローンの長さとバグ密度との比較

(1) で分類した11区間の範囲に含まれるファイル群ごとにバグ密度を算出し，コードクローンとして認識する長さの範囲とバグ密度の関係を分析した．

さらに，3.3 節に述べたように，SLOC の大きさごとにファイルを分類して，同様の分析を実施した．ここでは SLOC が，500 以上，100 以上 500 未満，100 未満の 3 段階にファイルを分類し，各ファイル群ごとにコードクローンとして認識する長さの範囲とバグ密度の関係を分析した．

5. 結果と考察

5.1. コードクローンの長さとバグ密度の関係

全ファイル群に含まれるコードクローンの長さとバグ密度の関係を図 2 に示す．図 2 は，各ファイル群が含むコードクローンの長さを横軸に示し，各ファイル群のバグ密度を縦軸に示している．図 2 より，コードクローンが含まれるファイル群のバグ密度は，Eclipse 全体の平均的なバグ密度より小さく，信頼性が高いといえる．これは，コードクローン検出対照外となるファイル群に含まれているバグ数が，ファイルのサイズと比較して少なくなかったためである．

また，コードクローンの長さとバグ密度の関係には負の相関関係が見られた．この結果より，長いコードクローンを含むファイル群はバグ密度が小さいといえ，長いコードクローンが含まれることによってソフトウェアの信頼性は向上することがうかがえる．さらに，短いコードクローンが含まれることによってソフトウェアの信頼性は低下することもうかがえる．

SLOC の大きさごとに 3 段階に（500 以上， 100 以上 500 未満，100 未満）ファイルを分類し，同様の分析を実施した結果を，それぞれ図 3，図 4，図 5 に示す．

図 3 から，SLOC が 500 以上の比較的規模の大きいファイルにおいて，短いコードクローンを含んでいるほど，バグ密度が小さい傾向があることがわかった．つまり，短いコードクローンを含むことは，信頼性低下の原因となり，長いコードクローンを含むことは信頼性向上につながると考えられる．

一方で，SLOC が 100 以上 500 未満のファイルが含むコードクローンの長さと，バグ密度の大小には関係

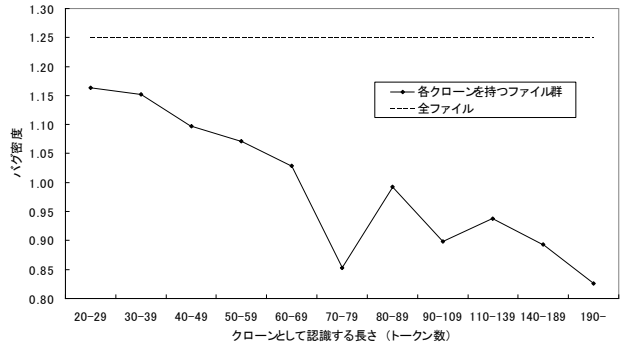

図 2 バグ密度とコードクローンの長さの関係（全ファイル）

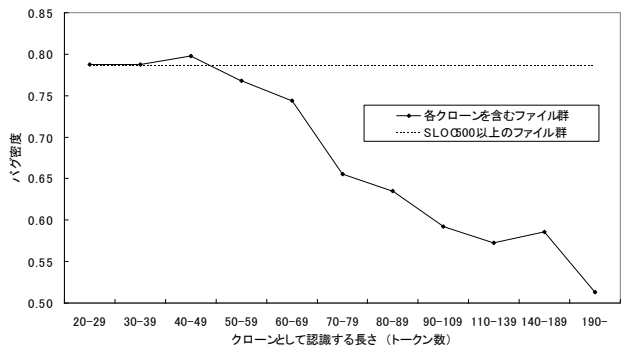

図 3 バグ密度とコードクローンの長さの関係（SLOC 500 以上）

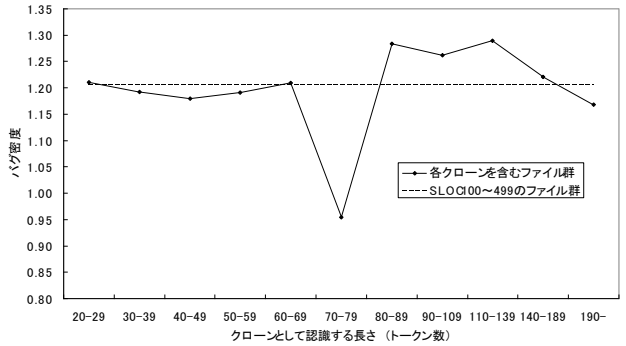

図 4 バグ密度とコードクローンの長さの関係（SLOC 100~500）

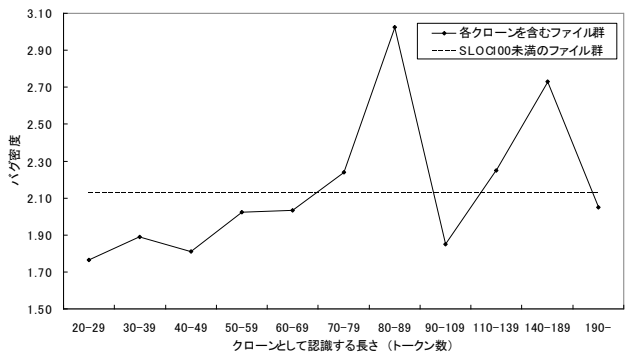

図 5 バグ密度とコードクローンの長さの関係（SLOC 100 未満）

表 2　ファイル間クローンの占める割合 *P*

		コードクローン長 (トークン)		
		20~29	140~189	190~
SLOC (行)	500~	0.844	0.392	0.616
	100~499	0.933	0.669	0.802
	1~99	0.966	0.934	0.976

が見られなかった．このことから，SLOC が 100 以上 500 未満のファイルにおいては，コードクローンの長さとバグ密度の間には関係がないと考えられる．

さらに SLOC が 100 未満の規模の小さいファイルにおいては，SLOC が 500 以上の比較的規模の大きいファイルとは逆の傾向が見られた．規模の小さいファイルでは，長いコードクローンを含むことが信頼性低下の原因となっていることがうかがえる．

5.2. ファイル間クローンに着目した分析

コードクローンとして検出された類似する 2 つのコード列の組（以降，クローンペア）には，(i) 2 つのコード列の両方が同一のソースコードに存在するファイル内クローンペアと，(ii) 2 つのコード列がそれぞれ異なるソースコードに存在するファイル間クローンペアが存在する[10]．どちらのクローンペアもプログラムの保守性を下げる要因となるが，ファイル間クローンペアは類似の処理を行うコード断片が 2 つのファイルにまたがるため，ファイル内クローンペアと比べてファイル間の結合度を増大させ，信頼性をより低下させる可能性がある．

そこで 4.2 節で述べた分析手順に対して，本稿ではさらにファイル間クローンに着目し，分析を行った．そして，SLOC の大きさによって分類したファイル群ごとに，ファイル間クローンの占める割合 *P* を次の式によって求めた．

$$P = \frac{\text{ファイル群におけるファイル間クローンのトークン数の総和}}{\text{ファイル群におけるクローンのトークン数の総和}}$$

その結果の一部を表 2 に示す．表 2 より SLOC の値が小さいほど P の値が大きくなっていることが確認できる．また，図 3，図 4，図 5 より，各ファイル群の平均的なバグ密度はそれぞれ 0.786，1.206，1.772 となっている．このように SLOC の値が小さいほど，平均的なバグ密度が大きくなっている．これらより，ファイル間クローンの割合がバグ密度に大きく影響を及ぼしていると考えられる．

6. むすびに

本稿ではコードクローンの長さとソフトウェア信頼性の関係について分析した．そして統合開発環境である Eclipse を対象に実験を行い，短いコードクローンが含まれるファイル群のバグ密度は大きいこと，長いコードクローンが含まれるファイル群のバグ密度は小さいことを明らかにした．

今後の課題としては，コードクローンに起因するバグの発生についての特徴をさらに分析する．そのために，コードクローンの長さだけではなく，ファイル間クローンとファイル内クローンについてもバグ密度とどういった関係を示すのかを詳しく調査する.そして，コードクローンによるバグ密度の予測モデルを作成する予定である．

謝　辞

本研究の一部は，文部科学省「次世代 IT 基盤構築のための研究開発」の委託に基づいて行われた．

文　献

[1] B. S. Baker, “A program for identifying duplicated code.24th Symposium on the Interface,” Computing Science and Statistics, pp.49-57, 1992.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lague and K. A. Kontogiannis, “Measuring clone based reengineering opportunities,” In Proc. Int'l Symposium on Softw. Metrics, pp.292-303, 1999.

[3] I. D. Baxter , A. Yahin, L. Moura, M. Sant’Anna and L. Bier, “Clone detection using abstract syntax trees,” In Proc. Int’l Conf. on Softw. Maintenance, pp.368-377, 1998.

[4] Bugzilla, https://bugs.eclipse.org/bugs.

[5] CCfinder, http://www.ccfinder.net/index-j.html.

[6] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Trans. Software Engineering, Vol.20, No.6, pp.652-686, 1994.

[7] D. Gusfield, “Algorithms on Strings, Trees and Sequences,” Cambridge University Press, pp.89-180, 1997.

[8] T. Gyimothy, R. Ferenc, I. Siket, “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” IEEE Trans. Software Engineering, Vol.31, No.10, pp.897–910, 2005.

[9] T. Kamiya, S. Kusumoto and K. Inoue, “CCFinder:A multi-linguistic token-based code clone detection system for large scale source code,” IEEE Trans. Software Engineering, Vol.28, No.7, pp.654-670, 2001.

[10] 門田暁人，佐藤慎一，神谷年洋，松本健一, “コードクローンに基づくレガシーソフトウェアの品質の分析,” 情報処理学会論文誌, Vol.44, No.8, pp.2178-2188, 2003.