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

信学技報
TECHNICAL REPORT OF IEICE.

コードクローンメトリクスとプロジェクトデータ間の相関分析 —同一仕様プロジェクト群を対象として—

肥後　芳樹† 　村上　寛明† 　楊　　嘉晨† 　柗本　真佑† 　楠本　真二†

三宅　武司†† 　藤波　崇志†† 　石橋　　昭†† 　星野　　隆††

† 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　〒 565-0871 大阪府吹田市山田丘 1-5
†† 日本電信電話株式会社 ソフトウェアイノベーションセンタ　〒 108-0075 東京都港区港南 2-13-34
E-mail: †{higo,h-murakm,jc-yang,shinsuke,kusumoto}@ist.osaka-u.ac.jp,
††{miyake.takeshi,fujinami.takashi,ishibashi.akira,hoshino.takashi}@lab.ntt.co.jp

あらまし　ソースコード中のコードクローンの存在は，複数箇所への同時変更を要求することから，ソースコードの保守性を低下させる要因の 1 つといわれている．しかしその一方で，コードクローンに対しては同時に変更が行われることはあまりなく，保守性の低下を招いているとは必ずしもいえないとの研究報告もいくつかされている．本稿では，同一の仕様に基づく 9 つのプロジェクトについて，コードクローンメトリクス（コードクローンの数および重複度）とバグ数等のプロジェクトデータ間の相関分析を行った結果を報告する．

キーワード　コードクローン，ソフトウェア品質とコードクローンの関係，相関分析

1. はじめに

コードクローン（以降，クローン）の存在はソースコードの保守作業を困難にするといわれており，これまでにさまざまな研究が行われてきている [11], [15], [19], [21]．クローンが保守作業に対して悪影響を与える原因の 1 つとして複数箇所への同時変更が挙げられる．あるクローンを変更した場合には，それと対応するクローンも変更する必要があるとされている．もし，同時に変更が行われなかった場合には，ソースコード中に一貫性の欠如が発生してしまい，それが後にバグとしてソフトウェアに不具合をもたらしてしまう．これまでに同時変更の観点からクローンの存在がソースコードの保守性に与える悪影響について，いくつか調査が行われてきている [1], [4], [6], [8]〜[10], [18], [20] ．これまでの調査により，全てのクローンが保守作業に悪影響を与えているわけではないが，バグの原因となるクローンも確かに存在しているということがわかってきた．しかし，調査はオープンソースソフトウェアに対して行われたものが多く，企業で開発されたソフトウェアに対する分析事例は多くない．

現在，著者らの研究グループでは，クローンの存在がソフトウェア開発に与える影響について研究を行っている．本稿では，クローンのメトリクス（クローンの数および重複度）とプロジェクトデータの間の相関関係の分析結果を報告する．本稿における調査対象は，異なるベンダーで開発された 9 つのプロジェクトであり，それらは同一の仕様に基づくものである．同一の仕様に基づいた複数のプロジェクトを対象にすることにより，その実装の違い（クローンの量の違い）が，プロジェクトデータに及ぼす影響を調査できる．

2. コードクローン

この章では，クローンの種類ならびにクローンに関する既存調査について述べる．

2.1 種　　類

クローンは，その類似度に基づいて一般的に以下の 3 種類に分類される．

TYPE-1　空白，タブ，改行文字，コメント等のプログラムの振る舞いに影響を与えないソースコード中の要素を除いて完全に同一のクローン．

TYPE-2　変数名や関数名等のユーザ定義名の違いやリテラルの違いのような字句単位での差異を含むクローン．

TYPE-3　字句単位よりも大きな違いを含むクローン．

これまでにさまざまな検出手法が提案されているが，クローンの定義は手法ごとに異なる．そのため，異なる手法を利用して同一ソースコードからクローンを検出した場合，その検出結果は異なる．本研究で利用したクローン検出技術については 3.2 で述べる．

2.2 既存調査

Inoue らは，クローン間の変数の対応付けに着目したバグ検出手法を考案した [6]．コピーアンドペーストによりクローンが生成されたあと，変数名がペースト後の文脈に合わせて変更されたとしても，クローン間では変数の対応付けが保たれている場合が多い．そのため，クローン間で変数の対応付けが保たれていない場合はバグの可能性があるとして，そのようなクローンを検出するツール CloneInspector を開発した．CloneInspector を企業で開発された 2 つの携帯端末関係のソフトウェアに適用したところ，68 のクローンが検出され，それらの中の 26 がバグを含んでいた．

門田らは，COBOL で記述されたソフトウェアに対してクローンとソースファイルの改版数の関係を調査した [22]．その結果，80%以上がクローンになっているソースファイルや，200 行以上のクローンを含むソースファイルは，他のソースファイルに比べて改版数が多くなる傾向であることがわかった．

コピーアンドペーストを用いた開発が望ましい場合もあるとの報告もされている [7]．例えば，新しい機能を追加する場合に，機能を追加する周辺のコードのクローンを作成し，そこに新しい機能を作成する．新しい機能を追加した部分の動作が安定してくると，その部分をコピー元へと移す．このような手順で開発を行うことにより，新しい機能追加に起因するバグの発生を稼働中のシステムにおいて抑えることができる．Kapser らは 2 つのオープンソースソフトウェアについて調査を行い，71%のクローンはソフトウェアの保守性に良い影響を与えていたと報告している．

山中らは，クローンの変更管理システムを作成し，実プロジェクトへの適用を行った [14]．彼らの変更管理システムは，バージョン管理システムに登録されている最新バージョンのソースコードとその前のバージョンのソースコードからそれぞれクローンを検出し，その 2 つの検出結果において対応が取れなかったクローンを変更が行われたクローンとして開発者に通知する．ある企業のソフトウェア開発プロジェクトにシステムを適用した結果，適用期間中に開発者が把握していなかったクローンに対して変更が行われ，その情報が開発者に電子メールにより何度か通知された．開発者はその情報を元にクローンの集約や位置情報の把握を行うことができ，システムの有用性が確認できた．

Chatterji らは，43 人の大学院生を対象として，クローン情報を利用したバグの原因箇所特定に関する実験を行った [3]．実験では，バグの原因箇所を 1 つ見つけた後にクローン情報を利用してその他のコード片もチェックするやり方は効率的に他のバグ原因箇所を見つけることができた．その一方で，バグの原因箇所を見つける前に，クローン情報を利用してコード片を閲覧していく方法では効率的にバグの原因箇所を見つけることができなかった．

Zhan らは 10 年以上保守されている通信系のソフトウェアの開発者 21 人に対して，クローンの生成に関する調査を行った [13]．その結果，クローンを生成する理由として，既存コードの変更によるバグの混入を防ぐ，1 つのモジュールにするのが難しいという理由に加えて，開発時間の制限上しかたなくクローンを生成したという組織的な理由や，他人のコードをコピーして変更を行うことで自分自身のコーディングスキルを上達させたいという個人的な理由があることがわかった．

Göde らは，C もしくは Java で記述された 3 つのオープンソースソフトウェアを対象として，クローンに対して行われる変更について調査を行った [4]．彼らの調査では，約 88%のクローンは生成された後に全く変更がされず，クローンに対して行われる変更のうちの約 15%が不注意による一貫性の欠如を引き起こしていたという結果であった．

著者らの研究グループでは，クローンに対する変更の頻度とクローンではない部分に対する変更の頻度を調査した [20]．調査対象は C/C++もしくは Java で記述された 15 のオープンソースソフトウェアである．調査の結果，クローンはそうでない部分に比べて変更される頻度が少ないという結果であった．

3. 実験の設定

本章では，対象プロジェクトやクローン検出，プロジェクトデータ等の実験に関する設定について記述する．

3.1 対象プロジェクト

本研究の対象は，国内のある企業（以降，発注元）が仕様を作成し，ベンダー 9 社（以降，発注先 A∼I）が開発したソフトウェアである．発注先の 9 社はそれぞれ完全に独立して開発した．これらのソフトウェアは同一仕様に基づいて開発された，実装の異なるソフトウェアである．

個々の発注先はそれぞれテストを行い，テストをパスした最終版のソースコードが発注元に納品された．全ての発注先は Java を用いて実装を行なった．最終版のソースコード規模は約 15,000∼24,000 ステップ数である．発注元でも納品されたソースコードに対してテストを行った．以降，発注先が行ったテストを発注先テスト，発注元が行ったテストを受入テストと呼ぶ．発注先テストと受入テストはそれぞれ以下の 3 つのテストからなる．

単体テスト (UT) 各モジュール単体での動作を検査するためのテスト．

結合テスト (IT) モジュール間の引数や返り値の受け渡し，データベースを介したデータのやり取り等を検査するためのテスト．

システムテスト (ST) システムに対して実際と同じような入力を与え，システム全体が要求された仕様通りに動作するかを検査するためのテスト．

3.2 コードクローン検出

本研究では字句単位のクローン検出を行った．字句単位のクローン検出手法は他の検出手法に比べて以下の特徴を持つ [2], [11], [19]．

- 検出の速度が速い．
- 検出結果が膨大になる傾向にある．つまり，漏れなくクローンを検出することは得意だが，検出結果には大量の誤検出が混じる傾向がある．

- 完全に同一なクローン（TYPE-1）や，変数名やリテラル等が異なるクローン（TYPE-2）を検出する．TYPE-3 クローンの検出は得意ではない．

著者らの研究グループでは，字句単位の長所を残しつつ短所を改善することを目的として，クローン検出ツール CloneGear を開発している．CloneGear は字句単位のクローン検出ツールであり，現在のところ，C/++，Java，Python，PHP，JavaScript に対応している [5]．CloneGear は以下の特徴を持つ．

特徴 1　連続した変数宣言文等のプログラムの繰り返し部分からのクローン検出を行わない．この特徴により，人間が確認する必要のない重複部分からのクローン検出を抑制できる．

特徴 2　コード片に字句単位よりも大きな差異があったとしても，その周辺領域が類似していればクローンとして検出する．この特徴により，TYPE-3 クローンの検出能力を向上できる．

特徴 1 は，著者らが過去に提案したソースコードの繰り返し部分を折りたたんだ上でクローンを検出する技術 [16] を利用して実現している．特徴 2 は，Smith-Waterman アルゴリズム [12] を利用することで実現している．いずれについても著者らが過去にその有効性を実験により示している [16], [17].

本研究ではクローンのメトリクスとして，クローン数と重複度を用いた．定義を以下に示す．

[クローン数]　対象ソースファイルもしくは対象プロジェクト全体に含まれるクローンの数．0 以上の整数値をとる．

[重複度]　対象ソースファイルもしくは対象プロジェクト全体に含まれる字句のうち，いずれかのクローンに含まれている字句の割合である．クローンが全く無い場合に最小値 0%，全ての字句がクローンになっている場合に最大値 100%をとる．

対象プロジェクトに対して表 1 に示す 6 つの設定を用いてクローンを検出した．設定の二項目を以下に示す．

最小字句数　クローンとして検出されるコード片の最小の大きさを表す．最小字句数が小さいほど漏れのない検出を行えるが，検出結果には検出の必要がないコードも含まれるようになる．

最大ギャップ数　クローンの内部に許容する不一致コードの量である．最大ギャップ数を大きくするほど，内部に大きい不一致コードがあったとしても 1 つのクローンとして検出できるが，検出結果には検出する必要のないクローンも含まれるようになる．最大ギャップ数は字句数ではなく，文の数である．

各プロジェクトのクローン数および重複度を表 2 に示す．この表に示すように，最小字句数が小さいほどクローン数が多く重複度が高くなり，ギャップを許容する設定の場合は更にそれらが大きくなっていることがわかる．

表 1　クローン検出の設定

検出設定 ID	最小字句数	最大ギャップ数	検出対象クローン
50T0G	50	0	TYPE-12
100T0G	100	0	TYPE-12
150T0G	150	0	TYPE-12
50T2G	50	2	TYPE-123
100T2G	100	2	TYPE-123
150T2G	150	2	TYPE-123

3.3　プロジェクトデータ

対象プロジェクトでは，発注元および発注先においてさまざまなデータが記録されていた．本研究では，定量的な値を持つ以下のデータを利用した．

規模に関するデータとして以下のものを用いた．

[ステップ数]　ソースファイルの行数を基にした値である．ただし，空行やコメント行は除く．

開発コストに関するデータとして以下のものを用いた．

[開発期間]　発注先が開発に要した期間を表す．

[開発工数]　発注先が開発に要した工数（人月）を表す．

[規模／週]　発注先の開発における週あたりの実装規模を表す．

テストに関するデータとして以下のテスト件数を用いた．このデータは，発注先テストにおける各テストのテスト件数である．なお，発注元が納品された各ソースコードに対して行った受入テストは同一であるため，受入テストの件数は本研究では利用しない．

[発注先 UT・IT・ST 件数]　発注先が UT・IT・ST を行うために作成したテストの件数を表す．

テストに関するデータとして以下のテスト密度も用いた．テスト密度は，発注先テストにおける各テストのテスト件数をその時のソースコードのステップ数で割った値である．

[発注先 UT・IT・ST 密度]　発注先が行った UT・IT・ST テストにおけるテスト密度を表す．

バグに関するデータとして以下のバグ数を用いた．発注先テストの各テストに加えて，受入テストの各テストについてもバグ数のデータを利用した．

[発注先 UT・IT・ST バグ数]　発注先が行った UT・IT・ST テストにおいて発見されたバグの数を表す．

[受入 UT・IT・ST バグ数]　発注元が行った UT・IT・ST テストにおいて発見されたバグの数を表す．

バグに関するデータとして，以下のバグ密度も用いた．バグ密度は，発注先テストの各テストにおいて，発見されたバグの数をその時のソースコードのステップ数で割った値である．

[発注先 UT・IT・ST バグ密度]　発注先が行った UT・IT・ST テストにおけるバグ密度を表す．

3.4　相関係数の計算

表 2 で示したクローンメトリクスと，3.3 で示した各プロジェクトデータの相関係数を計算した．具体的には，クローンメトリクスの降順で並べたプロジェクトの順が，プロジェクトデータの降順で並べたプロジェクトの順とどの程度同じであるのかを Spearman の順位相関係数を計算することで調査した．クローン検出の設定が 6 つあり，各検出結果についてクローン数および重複度の 2 つのメトリクスがある．そのため，各プロジェクトデータに対して 12 の相関係数が得られることになる．

4.　実験の結果

クローンメトリクスと各プロジェクトデータの相関係数を表 3 に示す．TYPE-12 の列はギャップを許容しない検出，つまり TYPE-1 と TYPE-2 のクローンを検出した場合の相関係数の

表 2　クローン数と重複度

発注先	50T0G		100T0G		150T0G		50T2G		100T2G		150T2G	
	クローン数	重複度	クローン数	重複度	クローン数	重複度	クローン数	重複度	クローン数	重複度	クローン数	重複度
A	187	8.1%	35	3.1%	10	1.3%	436	12.3%	96	7.5%	40	4.4%
B	3,296	37.1%	1,689	31.1%	885	26.7%	5,145	43.1%	3,022	38.7%	1,855	33.8%
C	561	16.2%	162	11.3%	71	8.0%	1,336	22.4%	422	18.3%	202	13.7%
D	937	20.2%	249	12.8%	97	7.6%	1,681	26.2%	506	19.1%	169	10.1%
E	965	16.2%	274	11.5%	110	7.9%	1,692	22.1%	616	18.1%	280	13.1%
F	493	16.7%	206	13.6%	128	11.2%	748	20.6%	304	17.3%	177	14.6%
G	305	13.5%	80	8.4%	33	5.4%	537	19.6%	178	15.0%	45	6.7%
H	1,086	26.3%	371	19.0%	164	13.9%	1742	34.1%	708	29.4%	283	19.5%
I	1.281	27.2%	569	24.2%	255	20.2%	1,973	32.1%	943	29.5%	312	17.9%

範囲を表している．一方，TYPE-123 はギャップを許容した検出，つまり TYPE-1，TYPE-2，および TYPE-3 のクローンを検出した場合の相関係数を表している．それぞれについて，字句数 50，100，および 150 を用いた場合の 3 つの相関係数が計算されており，この表で示すのはそれらの中央値である．この表において，相関の範囲が-0.7 以下もしくは 0.7 以上の値（強い相関）には二重下線を引いており，-0.4 以下もしくは 0.4 以上の値（ある程度の相関）には単下線を引いている．以下の 3 つのプロジェクトデータがクローンメトリクスと強い相関があることがわかる．括弧内の数値は強い相関があったクローンメトリクスを表す．

- 開発規模（クローン数，重複度）
- 規模／週（重複度）
- 受入 IT バグ数（クローン数，重複度）

ある程度の相関を持つプロジェクトデータは以下の 4 つである．

- 開発期間（クローン重複度）
- 発注先 IT 密度（クローン数，重複度）
- 受入 UT バグ数（クローン数，重複度）
- 発注先 UT バグ密度（クローン数，重複度）

5. 考　　察

この章では，4. で示した実験の結果を元に，クローンメトリクスと各プロジェクトデータとの関係およびクローンとプロジェクトの品質との関係について考察を行う．

5.1　プロジェクトデータとの相関について

強い相関があるとした 3 つのプロジェクトデータ，およびある程度の相関があるとした 4 つのプロジェクトデータについて，なぜそのような結果になったのかを考察した．

開 発 規 模

クローン数が多いおよび重複度が高いほど，開発規模が大きいという結果であった．クローンになっているコードが同一機能であると仮定すれば，同一機能が複数存在している（1 つのモジュールとしてまとめられていない）プロジェクトほど規模が大きいということであり，この相関は当然の結果といえる．

規 模 ／ 週

重複度が高いほど，規模／週が大きいという結果であった．クローンの生成理由が主にコピーアンドペーストと仮定すれば，単位時間あたりの開発規模は大きくなるため，この相関は当然の結果といえる．

受入 IT バグ数

クローンの数が多いおよび重複度が高いほど，受入 IT バグ数が多いという結果であった．受入テストにおいてバグが多いということは，発注先ではバグを取り切れていなかったということである．この結果から，著者らは以下の 3 つの可能性があると考えた．

- クローンを多く含む実装はテストの実施が難しく十分にテストが行えなかった．
- クローンを多く発生させた発注先は開発能力が高くなく，発注先テストを十分に行えなかった．
- コピーアンドペーストにより生成したコードはテストしなくても大丈夫だろうという思い込みから十分にテストを行わなかった．

開 発 期 間

クローン数が多いおよび重複度が高いほど，開発期間が短く

表 3　クローンメトリクスと各プロジェクトデータとの相関

プロジェクトデータ	クローン数		重複度	
	TYPE-12	TYPE-123	TYPE-12	TYPE-123
開発規模	0.75	0.75	0.73	0.68
開発期間	-0.13	-0.13	-0.26	-0.40
開発工数	-0.03	-0.03	-0.12	-0.08
規模／週	0.57	0.58	0.67	0.75
発注先 UT 件数	-0.20	-0.20	-0.08	-0.13
発注先 IT 件数	-0.20	-0.18	-0.37	-0.25
発注先 ST 件数	0.03	0.00	0.00	-0.12
発注先 UT 密度	-0.22	-0.22	-0.13	-0.15
発注先 IT 密度	-0.40	-0.38	-0.50	-0.37
発注先 ST 密度	0.00	-0.02	-0.03	-0.10
発注先 UT バグ数	-0.37	-0.32	-0.33	-0.18
発注先 IT バグ数	0.08	0.11	-0.06	0.09
発注先 ST バグ数	-0.01	0.11	-0.03	0.28
受入 UT バグ数	0.38	0.42	0.25	0.38
受入 IT バグ数	0.74	0.80	0.68	0.80
受入 ST バグ数	0.03	0.03	-0.03	-0.05
発注先 UT バグ密度	-0.62	-0.62	-0.53	-0.47
発注先 IT バグ密度	0.00	0.02	-0.07	0.05
発注先 ST バグ密度	-0.06	0.03	-0.07	0.13

なるという結果であった．クローンがコピーアンドペーストにより生成されたことを仮定すると，コピーアンドペーストによる開発は，開発期間の短縮に寄与しているといえる．

発注先 IT 密度

クローン数が多いおよび重複度が高いほど，発注先 IT 密度が低いという結果であった．この結果は，発注先 IT 件数がクローンメトリクスと負の相関があることとも一致する．クローンが存在しているほど，発注先 IT テストが十分に行われなかったことを表す結果である．

受入 UT バグ数

クローン数が多いおよび重複度が高いほど，受入 UT において多くのバグが見つかっているという結果であった．発注先 UT バグ数は反対の結果になっているため，クローンが多く存在しているほど，発注先 UT において見逃したバグが多いことを示唆しているといえる．

発注先 UT バグ密度

クローン数が多いおよび重複度が高いほど，発注先 UT バグ密度が小さいという結果であった．発注先 UT バグ数が小さいことからも，クローンが多く存在しているほど，発注先 UT が十分に行われたなかったといえる結果である．

以上のことから，クローンの存在がプロジェクトに与える影響について下記のようにまとめることができる．

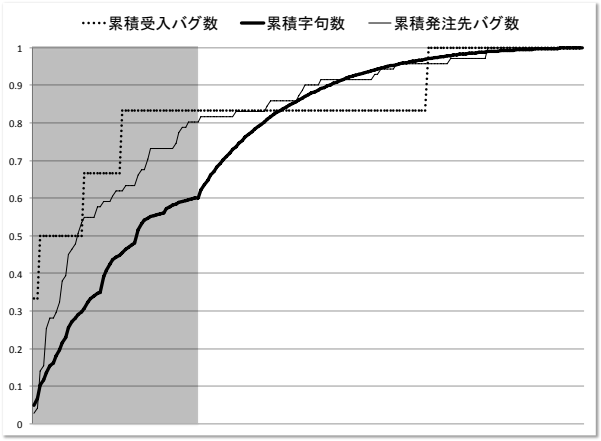
- コピーアンドペーストを多用して開発を行うことにより，開発期間を短縮させることができる．
- しかしその反面，テスト件数が少ない・テスト密度が小さくなるという傾向もあり，結果として発注先テストにおいてバグを見つけづらくなる．

5.2 プロジェクトの品質との相関について

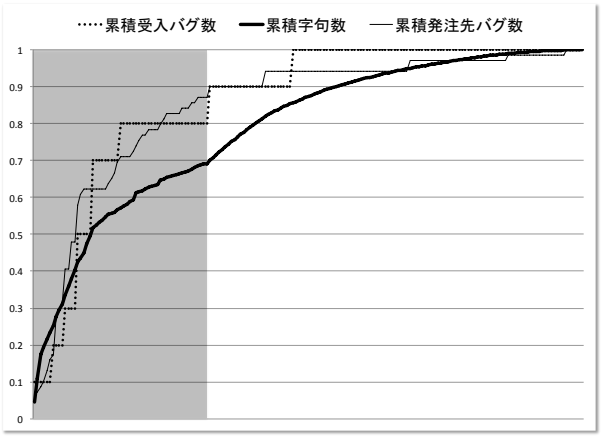
発注元では，受入テストで見つかったバグ数に応じて，納品された 9 つのプロジェクトを高品質，並品質，低品質の 3 つに分類していた．高品質には発注先 A および F，低品質には発注先 I が分類された．なお，高品質に分類されたプロジェクトの受入テストにおける発見バグ数は 10 以下，低品質に分類されたプロジェクトの受入テストにおける発見バグ数は約 50 である．

これら 3 つのプロジェクトに対して，設定 50T2G の検出結果を用いて，発注先テストおよび受入テストにおいて，クローンが検出されたソースファイルからバグが見つかったか調査した．図 1 では，それら 3 つのプロジェクトのファイルが X 軸に並べられている．並び順は各ファイルが所有するクローンの数の降順である．つまり，左に配置されたファイルほど多くのクローンを持っており，右に配置されたファイルにはクローンは存在しない．このグラフは，累積字句数，累積発注先バグ数，および累積受入バグ数を表している．

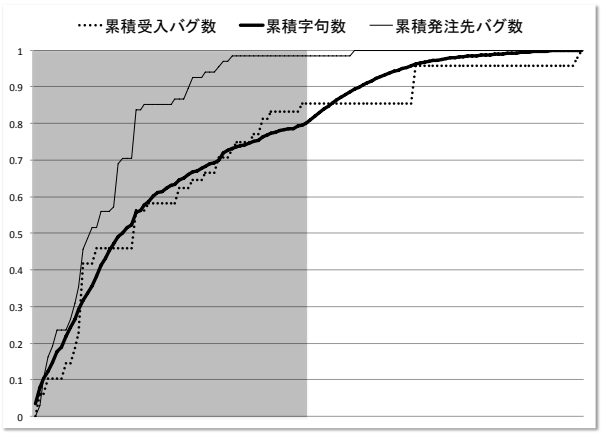
高品質であった発注先 A および発注先 F では，累積発注先バグ数および累積受入バグ数の曲線が，累積字句数よりも上にあり，クローンが存在しているファイルにバグが多く存在していた傾向にあることがわかる．また，低品質であった発注先 I では，累積発注先バグ数について同様の傾向が見られた．しかし，累積受入バグ数の曲線は累積字句数の曲線とほぼ同じであり，クローンが存在しているファイルに受入テストで見つかったバグが偏って存在していたわけではなかった．



(a) 発注先 A（高品質）



(b) 発注先 F（高品質）



(c) 発注先 I（低品質）

図 1　発注先 A，F，I の累積字句数と累積発注先バグ数および累積受入バグ数の関係．X 軸のソースファイルはクローン数の降順で整列されている．クローン数が同じソースファイルは字句数の降順で整列されている．ハイライト部分がクローン数が 1 以上の範囲である．

図 1 により，プロジェクトが高品質であっても低品質であってもクローンの情報を利用することにより，注力してテストを行わなければならないソースファイルをある程度特定できることを示唆している．しかしながら，クローンの情報を使うだけ

では，注力してテストすべきモジュールを特定するには不十分であることもわかる．今後は，注力すべきモジュールを特定するモデルをクローン情報やその他のプロダクトメトリクスおよびプロセスメトリクスを利用して構築していく予定である．

6. おわりに

本研究では，同一の仕様に基づいて作成された 9 つのプロジェクトに対して，そのプロジェクトデータとコードクローンメトリクス（コードクローン数および重複度）の相関を調査した．調査の結果，コードクローンが多く存在する（コピーアンドペーストを多用する）プロジェクトでは開発期間が短くなる傾向にあること，およびコードクローンが多く存在するとテスト件数やテスト密度が小さくなっておりバグを発見しづらくなる傾向があることがわかった．また，高品質に分類されたプロジェクト，低品質に分類されたプロジェクトのソースファイルのクローンの数と発見されたバグの数を調査し，クローンが存在するファイルからはバグが多く存在している傾向にあることもわかった．

今後は，コードクローン情報を利用してテスト工程において注力すべきモジュールを自動的に特定するためのモデルの作成を行っていく予定である．

文　献

[1] L. Barbour, F. Khomh, and Y. Zou. An Empirical Study of Faults in Late Propagation Clone Genealogies. *Journal of Software: Evolution and Process*, 25(11):1139–1165, 2007.

[2] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[3] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft. Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, pages 20–29, 2011.

[4] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 311–320, 2011.

[5] Y. Higo. Clonegear. https://github.com/YoshikiHigo/CloneGear.

[6] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of Finding Inconsistently-changed Bugs in Code Clones of Mobile Software. In *Proceedings of the 6th International Workshop on Software Clones*, pages 94–95, 2012.

[7] C. J. Kapser and M. W. Godfrey. ”Cloning Considered Harmful” Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[9] A. Lozano and M. Wermelinger. Assessing the Effect of Clones on Changeability. In *Proceedings of the 24th International Conference on Software Engineering*, pages 227–236, 2008.

[10] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1227–1234, 2012.

[11] D. Rattan, R. Bhatia, and M. Singh. Software Clone Detection: A Systematic Review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[12] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[13] W. Zhao, Z. Xing, X. Peng, and G. Zhang. Cloning Practices: Why Developers Clone and What Can Be Changed. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance*, pages 285–294, 2012.

[14] 山中, 崔, 吉田, 井上, 佐野. コードクローン変更管理システムの開発と実プロジェクトへの適用. **情報処理学会論文誌**, 54(2):883–893, 2013.

[15] 神谷, 肥後, 吉田. コードクローン検出技術の展開. **コンピュータソフトウェア**, 28(3):28–42, 2011.

[16] 村上, 堀田, 肥後, 井垣, 楠本. ソースコード中の繰り返し部分に着目したコードクローン検出ツールの実装と評価. **情報処理学会論文誌**, 54(2):845–85, 2013.

[17] 村上, 堀田, 肥後, 井垣, 楠本. Smith-Waterman アルゴリズムを利用したギャップを含むコードクローン検出. **情報処理学会論文誌**, 55(2):981–993, 2014.

[18] 肥後, 楠本. コード修正履歴情報を用いた修正漏れの自動検出. **情報処理学会論文誌**, 54(5):1686–1696, 2013.

[19] 肥後, 楠本, 井上. コードクローン検出とその関連技術. **電子情報通信学会論文誌** *D*, J91-D(6):1465–1481, 2008.

[20] 堀田, 佐野, 肥後, 楠本. 修正頻度の比較に基づくソフトウェア修正作業量に対する重複コードの影響に関する調査. **情報処理学会論文誌**, 52(9):2788–2798, 2011.

[21] 堀田, 肥後, 楠本. 生成防止，分析効率化，不具合検出を中心としたコードクローン管理支援技術に関する研究動向. **コンピュータソフトウェア**, 31(1):14–29, 2014.

[22] 門田, 佐藤, 神谷, 松本. 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**, 44(8):2178–2187, 2003.