同一仕様プロジェクトを対象とした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

肥後 芳樹[1]　柗本 真佑[1]　楠本 真二[1]　三宅 武司[2]　藤波 崇志[2]　石橋 昭[2]　星野 隆[2]

概要：ソースコード中のコードクローンの存在は，ソフトウェアの保守性を悪化させる 1 つの要因であるといわれている．しかしその一方で，コードクローンに対して同時に変更が行われることはあまりなく，コードクローンの存在が問題を引き起こすことは少ないという調査報告もある．また，コードクローンはソフトウェア開発に有用であると報告している研究もある．現在著者らはコードクローンの存在がソフトウェア開発や保守作業に与える影響について研究を行っている．この論文では，コードクローンメトリクスとテストケース数や検出バグ数といったプロジェクトデータ間の相関分析の結果を報告する．この分析対象は 9 つのウェブシステムであり，同一の仕様に基いて異なる組織によって開発された．つまり，これらは機能が同一であり実装が異なるシステムである．この 9 つのシステムを対象とすることで，実装の違いとプロジェクトデータ間の関係を調査できる．調査の結果，コードクローンが多く存在しているプロジェクトほどコードのサイズが大きく実装の速度が早いことがわかった．コードクローンが多く存在しているほど結合テストでバグを見逃す傾向であった．多くのファイルがコードクローンを共有している場合は結合テストは良く実施されていた．しかし，そのような場合は，単体テストでバグを見逃す傾向であった．

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いる．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．

クローンに関する問題の支援として，これまでにさまざまな研究が行われている [18], [25], [27], [29]．また，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], [5], [7], [10], [11], [14], [26], [28] ．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．

著者らは，クローンの存在がどの程度ソフトウェア開発や保守に影響をおよぼすのか，また，クローンの存在は本当に技術的負債となるのかを調査したいと考えている．しかし，これらの調査を厳密に行うことは非常に難しい．1 つのシステムを異なった 2 つの方法で開発および保守する必要がある．一方は存在しているクローンに対してなにも対処することなく開発及び保守を行うシステム，他方はクローンをできる限り 1 つのモジュールにまとめた上で開発及び保守を行うシステムである．これら 2 つを比較すればクローンの影響を調べることができる．クローンの影響が明らかになれば開発者はクローンを集約すべきかどうかを判断できる．しかし，このような調査を行うことは現実的ではない．

この論文では，コードクローンメトリクスとさまざまなプロジェクトデータの相関分析の結果を報告する．調査対象は 9 つのウェブシステムであり，それらは全て異なった組織により開発されたものである．これらのウェブシステムを調査対象として選定した理由は，それらが同一の仕様に基いて開発されたシステムであるからである．つまり，それらは同一の機能を有しているが実装が異なるシステムである．これらのシステムの実装の違い (クローンメトリクスの違い) とプロジェクトデータを比較することで，クローンがソフトウェア開発および保守に与える影響を考察する．

以降，本論文は 2 章でクローンに関する説明を行い，3

[1] 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
[2] 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

章で実験の設定について述べる．4 章では実験の結果を報告し，5 章では実験の妥当性について述べる．最後に 6 章で本論文をまとめる．

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，コメント等のソフトウェア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 [18], [20]．例えば CCFinder のような字句単位の検出手法では，一定数以上連続して重複する字句の列がクローンとして検出される [8]．CloneDR のような抽象構文木を用いた検出では，まず対象ソースコードを解析することにより抽象構文木が生成され，同じ形状を持つ部分木がクローンとして検出される [2]．つまり，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 節で述べる．

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 [7]．コピーアンドペーストによりクローンが生成された後，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，そのうちの 26 がバグを含んでいた．

Monden 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 [15]．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 [9]．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 [23]．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この変更管理システムを適用した結果，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．開発者はその情報に基いてクローンの把握や集約を行うことができた．

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 [4]．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 [24]．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 つのモジュールに集約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 [5]．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．

堀田らは，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 [28]．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．

Tsunoda らは，バグを含むモジュールの予測モデルにクローンメトリクスを組み込む場合は，複数種類の検出結果を利用したほうがモデルの予測精度が良くなることを実験により示した [12]．彼らは CCFinderX，PMD s Copy/Paste Detector，Simian，Nicad の 4 つのソフトウェアそれぞれを 2 つの設定を用いてクローン検出を行い，8 種類の検出結果を得た．それぞれのクローン検出結果から対象ソースコードの重複度を算出し，それを予測モデルに組み込むことにより評価を行った．いずれかのクローンメトリクスを組み込んだ 8 種類の予測モデルに加え，すべてのクローンメトリクスを組み込んだ予測モデル，いずれのクローンメトリクスも組み込まない予測モデルの，計 10 種類の予測モデルの比較評価を行った．その結果，全てのクローンメトリクスを組み込んだ予測モデルが最も精度が高いという結果であった．単一のクローンメトリクスを組み込んだ予測モデルの中には，クローンメトリクスを組み込まない予測モデルに比べて精度が劣るものも存在した．

Saini らは，クローンになっているメソッドとそうでないメソッドの間での種々のメトリクスの値が異なるのかを調査した [21]．調査対象のメトリクスは McCabe の複雑度 [13] や Halstead のソフトウェアサイエンス [6] のようによく利用される伝統的なメトリクスから，引数の数やループの数などのようにプログラム要素の数を単純にカウントしたメトリクスなどさまざまであり，計 27 種が計測された．その結果，メソッドのサイズはクローンになっているメソッドのほうが約 29%小さくなるという傾向であったが，他のメトリクスの場合はほとんど差がないという結果であった．

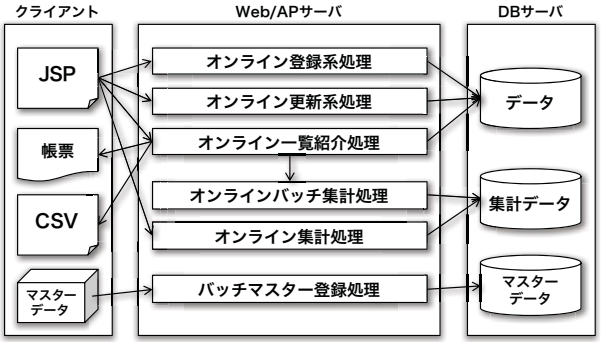
3. 実験の設定

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．

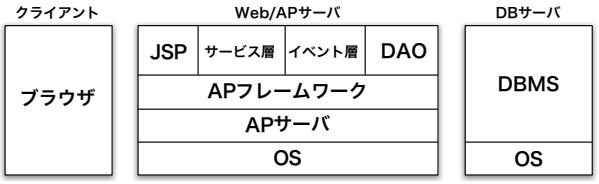
3.1 対象プロジェクト

図 1 は対象プロジェクトの概要を表している．このプロジェクトは Web ベースの在庫管理システムである．オンラインのトランザクション処理を実現するために J2EE が利用されている．このシステムの画面数は 11，および帳票数は 2 であり，32 種類の処理を持つ．ファンクションポイント数は 113FP である．画面処理 (クライアントサイド) は JSP，業務処理 (サーバサイド) は Java を用いて開発された．

このシステムは実験用プロジェクトであり，ある組織が仕様を作成し，異なる 9 つの組織が開発した．以降，仕様を作成した組織を発注元，開発を行った組織をベンダーと呼ぶ．9 つのベンダー（V_A〜V_I）は，それぞれ独立して開発を行った．このシステム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．なお，開発時の条件をできる限り揃えるために，システム開発経験が浅い新人や突出したスキルを持つ有識者はこのシステムの開発には加わっていない．



(a) アーキテクチャ



(b) 階層構造

図 1　対象システムの概要

個々のベンダー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．この実験の対象は Java で記述された業務処理部分である．最終版のソースコード規模は約 20,000〜44,000 行 (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，ベンダーが行ったテストをベンダーテスト，発注元が行ったテストを受入テストと呼ぶ．ベンダー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 [3], [18], [27]．

- 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（T PE-1）や，変数名やリテラル等が異なるクローン（T PE-2）を検出する．T PE-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[*1]．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．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 類似したコード片が字句単位よりも大きな差異を含んでいたとしても，その全体を 1 つのクローンとして検出できる．この特徴により，T PE-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 [17] を利用して実現している．特徴 2 は，Smith-Waterman アルゴリズム [22] 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がどれくらい存在しているかの指標として以下の 3 つのメトリクスを用いた．対象システムのソースコード規模が異なるため，いずれも正規化された値を持つメトリクスである．

DOC (Density Of Clones) 1,000 行あたりのクローンの数を表すメトリクスである．対象システムから検出されたクローンの数を対象システムのキロ行数で除算することにより求める．

ROC (Ratio Of Clones) 対象システムを構成する字句のうち，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．

DORF (Density of Related Files) 1,000 行あたりのクローンを共有しているファイルペア数を表す．対象システムにおいてクローンを共有しているファイルペア数を対象システムのキロ行数で除算することにより求める．

図 2 は 3 つのメトリクスの計測例を表す．この例では 3 つのソースファイルがあり，ABC の 3 つのクローンのグループが検出されている．ABC に含まれる各クローンの字句数は，20，30，10 である．この例では，各メトリクスは図 2 の下部に示す式で計算される．

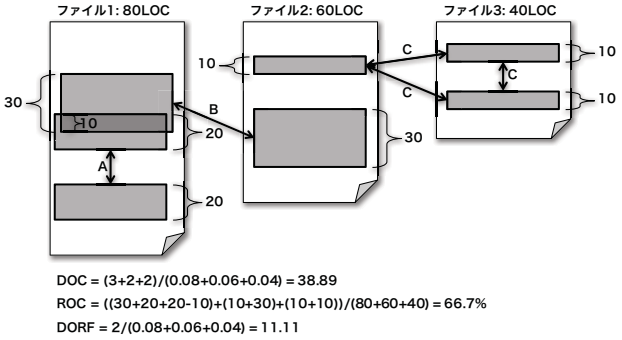


図 **2** クローンメトリクスの計算例

各プロジェクトに対して，著者らは 3 つの設定を用いてクローンの検出を行った．設定の違いは検出する最小のクローンの大きさであり，それぞれ 50 字句，100 字句，150 字句以上のクローンを検出するように設定した．クローンの検出結果は誤検出と見逃しが少ないほど良いとされるが，誤検出と見逃しはトレードオフの関係にある．誤検出を少なくするような設定を利用してクローン検出を行えば見逃しが増えてしまい，見逃しを少なくするような設定で検出を行えば誤検出が増えてしまう．

各設定は以下の意図により用いた[*2]．

50 字句 字句単位のクローン検出でよく利用されるしきい値である．このしきい値を用いた場合，クローンの検出ツールは見逃しは少ないが誤検出が多くなる傾向にある [8], [19].

100 字句 誤検出を減らす目的で用いたしきい値である．見逃すクローンは多くなってしまう．

150 字句 できる限り誤検出を行わない目的で用いたしきい値である．さらに見逃すクローンは多くなる．

表 1 は上記の 3 つの設定を用いて検出したクローンから算出したクローンメトリクスの値を表している．すべてのメトリクスにおいて，大きなしきい値を用いるほどメトリクス値が小さくなっていることがわかる．

3.3 プロジェクトデータ

対象プロジェクトでは，発注元およびベンダー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．なお，これらのデータの利用にあたり，クローンメトリクスと高い相関があるという仮定があるわけではない．利用可能なデータを多く利用してその中から相関があるデータを見つけることが本

[*1] https://github.com/YoshikiHigo/CloneGear

[*2] 50 字句は，クローン検出の研究でしばしば用いられる値である [8], [19]．100 字句および 150 字句は，著者らのこれまでのクローン検出の経験に基づく値である．企業で開発されたソフトウェアはオープンソースソフトウェアに比べてクローンの量が多い傾向がある．新規でクローン検出を行う場合は 50 字句をしきい値としてクローン検出を行い，非常に多くのクローンが検出された場合には設定を 100 字句に変更して再度クローン検出を行う，というのが通常の検出プロセスである．

表 1　クローンメトリクス値の算出結果

ベンダー	しきい値の字句数 50			しきい値の字句数 100			しきい値の字句数 150		
	DOC	ROC	DORF	DOC	ROC	DORF	DOC	ROC	DORF
V_A	11.66	12.3%	3.69	2.57	7.5%	0.59	1.07	4.4%	0.27
V_B	61.17	43.1%	5.24	35.93	38.7%	2.64	22.06	33.8%	1.75
V_C	24.85	22.4%	6.64	7.85	18.3%	1.77	3.76	13.7%	0.89
V_D	20.92	26.2%	10.56	6.30	19.1%	3.46	2.10	10.1%	0.95
V_E	30.18	22.1%	10.01	10.99	18.1%	3.03	4.99	13.1%	1.78
V_F	14.60	20.6%	2.81	5.93	17.3%	1.00	3.45	14.6%	0.60
V_G	17.38	19.6%	9.58	5.76	15.0%	3.50	1.46	6.7%	1.49
V_H	33.52	34.1%	7.95	13.62	29.4%	3.79	5.45	19.5%	2.23
V_I	32.86	32.1%	3.86	15.71	29.5%	2.78	5.20	17.9%	2.42

実験のアプローチである．

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＞　ベンダーが開発に要した期間を表す．

＜開発工数＞　ベンダーが開発に要した工数（人月）を表す．

＜規模／週＞　ベンダー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，ベンダーテストにおけるテスト件数である．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．

＜ベンダー **UT・IT・ST** 件数＞　ベンダーが単体テスト，結合テスト，システムテスト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，ベンダーにおけるテストの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．

＜ベンダー **UT・IT・ST** 密度＞　ベンダーが行った単体テスト，結合テスト，システム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．ベンダーテストに加えて，受入テストについてもバグ数のデータを利用した．

＜ベンダー **UT・IT・ST** バグ数＞　ベンダーが行った単体テスト，結合テスト，システム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＞　発注元が行った単体テスト，結合テスト，システム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，ベンダーテストにおいて，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．

＜ベンダー **UT・IT・ST** バグ密度＞　ベンダーが行った単体テスト，結合テスト，システムテストにおけるバグ密度を表す．

3.4　相関係数の計算

表 1 で示したクローンメトリクスと，3.3 節で示した各プロジェクトデータの相関関係を分析した．具体的には，クローンメトリクスを降順に並べた時のプロシェクトの順が，プロジェクトデータを降順に並べた時の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 3 つあり，各検出結果について 3 つのクローンメトリクス値がある．そのため，各プロジェクトデータに対して 9 つの相関係数が得られることになる．

4.　実験の結果

図 3 は Spearman の順位相関係数 (以降，ρ と表す) の計算結果を表している．各プロジェクトデータに対する各クローンメトリクスにおいて，線分が表示されている．線分の上端と下端はそれぞれ ρ の最大値と最小値を表している．この実験では 3 つの設定を用いてクローン検出を行ったため，各プロジェクトデータにおける各クローンメトリクスの ρ は 3 つ存在する．中央値は $\times$ で示されている．クローンメトリクス名に付随しているアスタリスクの数は，その線分を構成する ρ の p 値が 0.017 以下であった数を表している．0.017 を用いた理由は $(1-0.017)^3 \fallingdotseq 0.95$ となるからである．つまり，3 つの検定結果に 1 つも偶然の結果が含まれていないことが 95%以上の確率であるためには，個々の検定結果は 98.3%以上の確率で偶然でないことが求められるため，0.017 を用いた．各線分には 3 つの ρ の値が含まれるため，アスタリスクの数は最大で 3 となる．本実験では，$|\rho|$ が 0.4 以上のときにプロジェクトデータとクローンメトリクスの間に相関があるとし，相関がある場合をさらに以下のように 3 つに分類した．

強い相関　$0.9 \leq |\rho|$

中程度の相関　$0.7 \leq |\rho| < 0.9$

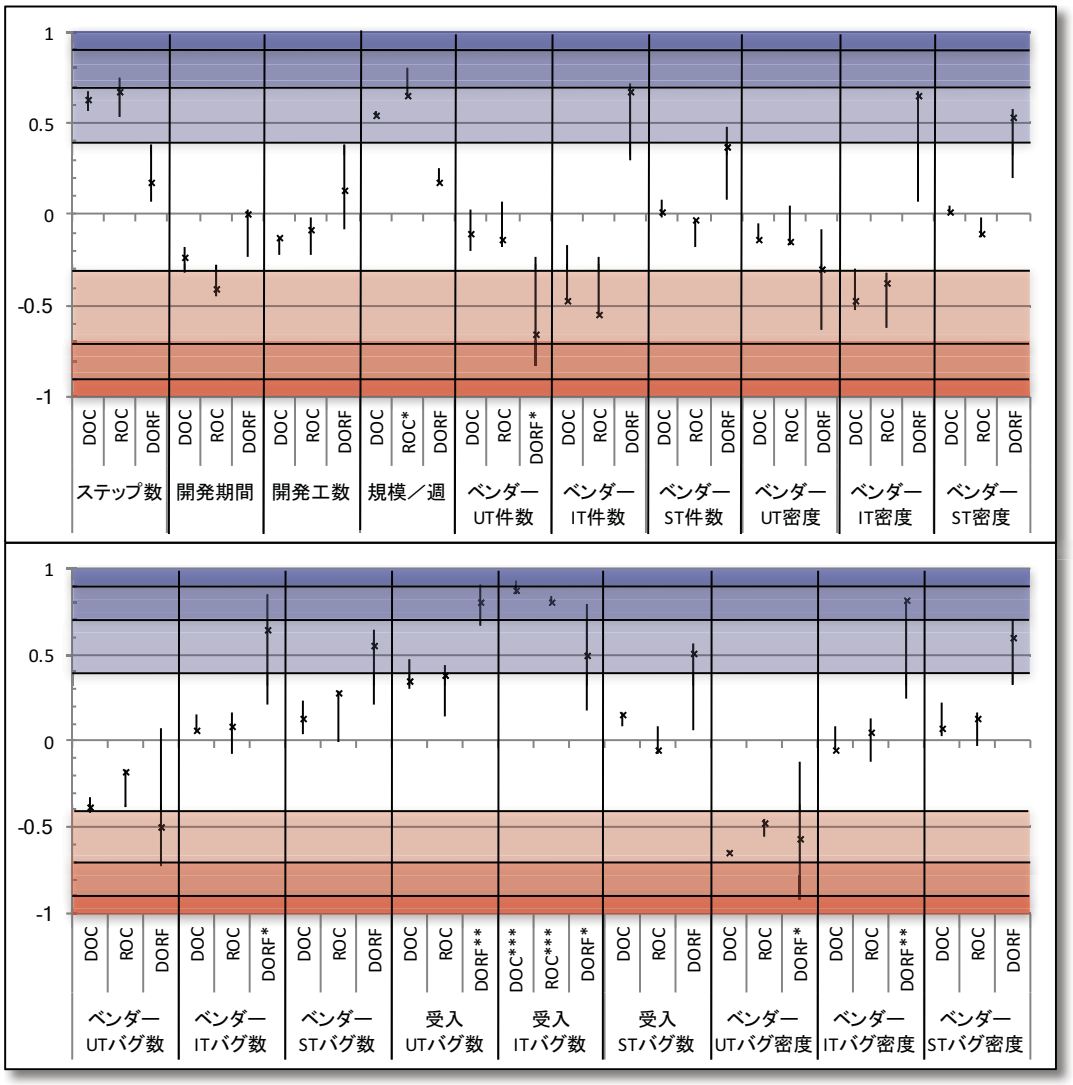


図 3　クローンメトリクスとプロジェクトデータ間の相関係数の計算結果
(アスタリスクの数は p 値が 0.017 以下であった検定の数を表している)

弱い相関　$0.4 \leq |\rho| < 0.7$

図 3 では，正の相関部分は青，負の相関部分は赤で示されており，相関が強いほど濃く強調表示されている．この実験では，7 つのプロジェクトデータが有意水準 0.017 においていずれかのクローンメトリクスと正または負の相関があるという結果であった．

以下のプロジェクトデータはいずれかのクローンメトリクスと有意に強い相関があったものである．括弧内は強い相関があったクローンメトリクスを表す．

- 受入 UT バグ数 (DORF)
- 受入 IT バグ数 (DOC)

以下は有意に中程度の相関があったプロジェクトデータである (有意に強い相関があったプロジェクトデータを除く)．

- 規模／週 (ROC)
- ベンダー UT 件数 (DORF)
- ベンダー IT バグ数 (DORF)
- 受入 IT バグ数 (ROC, DORF)
- ベンダー UT バグ密度 (DORF)
- ベンダー IT バグ密度 (DORF)

以上のことから，ソフトウェア開発プロジェクトとクローンの存在には以下のような関係があると著者らは考えた．

開発速度　規模／週と ROC に相関があった．これはソースコードが重複している割合が高いほど，週あたりの実装規模が大きいことを表している．クローンの生成理由がコピーアンドペーストであると仮定すれば，コピーアンドペーストによるコードの再利用が実装のスピードの向上に貢献していると考えることができる．

結合テスト　受入 IT バグ数と DOC および ROC に相関があった．これは，クローンの量が多いほど，受入結合テストにおいて多くのバグが検出される傾向であることを表している．つまり，クローンの量が多いとベンダーの結合テストにおいてバグを見逃す傾向にあることがわかる．

また DORF がベンダー IT バグ数およびベンダー IT バグ密度と正の相関があった．言葉を変えれば，クローンを共有しているファイルの数が多いと，ベンダーにおける結合テストが良く実施されている傾向であることを表している．

単体テスト　ベンダー UT 件数およびベンダー UT バグ密度と DORF に負の相関があった．多くのファイルがクローンを共有していると，ベンダーにおいて単体テストの達成度が低い傾向にあることを表している．受入 UT バグ数と DORF の正の相関は，ベンダー単体テストにおいて見逃されたバグが受入単体テストで多く見つかったことを表している．

クローンの存在が結合テストと単体テストに与える負の影響は，「テストされたコードをコピーアンドペーストした場合はコピー先のコードは十分にテストしなくても良い」という開発者の先入観が原因ではないかと著者らは考えている．

以上のことから，クローンの検出結果は以下のようにテストの実施状況の確認に利用できると著者ら考えた．

- 多数のクローンが検出された場合やソースコードの大部分がクローンである場合は，結合テストが十分に実施されているかを確認する．
- 多数のファイルがクローンを共有している場合は単体テストが十分に実施されているかを確認する．

5. 実験の妥当性について

本実験では有意水準 0.017 のもとで検定を行った．これは，用いたクローンの検出結果が 3 つあり，その全てが偶然の結果ではないことを 95%担保するための設定である．

また，本実験では 57 の相関係数 (19 のプロジェクトデータと 3 つのクローン検出結果) を算出した．そのため，FDR 法のような多重検定法を用いるのも 1 つの方法であっただろう．

実験対象で利用した 9 つのシステムは，見つかったバグの位置についてはファイル単位の情報のみを含んでいた．そのため，各バグがクローンの内部に存在していたのか，クローンとそうでない部分の境界に存在していたのか，クローンとは関係のない部分に存在していたのかは不明である．

今回の実験は 1 つの対象プロジェクト群にしか適用できていない．そのため今回の実験結果がどの程度他のプロジェクト群に対して当てはまるかは現在のところ不明である．

6. おわりに

本論文では，プロジェクトデータとクローンの相関関係を分析した結果について報告した．分析対象は同一の仕様に基いて開発された 9 つのシステムであり，すべて別の組織によって開発された．実験の結果，以下のことが判明した．

- クローンの量が多いほど，実装スピードが早い傾向にある．
- クローンの数が多くソースコードの重複度が高いほど，ベンダーにおける結合テストで多くのバグが見逃されてる傾向にある．
- 多くのファイルがクローンを共有しているほど，ベンダーにおける単体テスト数が少なくなり検出されるバグの数も減る傾向にある．しかし，興味深いことにベンダーにおける結合テスト数は多くなり検出されるバグの数は多くなる傾向にある．

この結果より，クローンの検出結果はテストの実施状況の確認に利用できると著者らは考えた．具体的には，多数のクローンが検出された場合やソースコードの大部分がクローンだった場合には結合テストの実施状況を確認したり，多くのファイルがクローンを共有していた場合には単体テストの実施状況を確認したり，という作業がテストが不十分な状態でソフトウェア開発が次の工程に移行してしまうことを予防する 1 つの手段となりうることを示した．

今後の研究方針として，著者らは DORF メトリクスについてさらに調査を行うことを計画している．このメトリクスはクローンの量そのものではなく，クローンを共有しているファイルの数を表している．DORF が高い場合は多くのファイルがクローンを共有していることを意味している．著者らは DORF メトリクスがソフトウェアの設計の粗悪さを表しているのではないかと考えており，今後このメトリクスの有用性について研究を行う予定である．

参考文献

[1] Barbour, L., Khomh, F. and Zou, .: An Empirical Study of Faults in Late Propagation Clone Genealogies, *Journal of Software: Evolution and Process*, Vol. 25, No. 11, pp. 1139–1165 (2007).

[2] Baxter, I. D., ahin, A., Moura, L., Sant Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proceedings of the International Conference on Software Maintenance*, pp. 368–377 (1998).

[3] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 577–591 (2007).

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

[5] Göde, N. and Koschke, R.: Fre uency and Risks of Changes to Clones, *Proceedings of the 33rd International Conference on Software Engineering*, pp. 311–320 (2011).

[6] Halstead, M. H.: *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science Inc. (1977).

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

[8] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670 (2002).

[9] 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).

[10] Li, Z., Lu, S., Myagmar, S. and Zhou, .: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).

[11] Lozano, A. and Wermelinger, M.: Assessing the Effect of Clones on Changeability, *Proceedings of the 24th International Conference on Software Engineering*, pp. 227–236 (2008).

[12] Masateru Tsunoda, asutaka Kamei, A. S.: Assessing the Differences of Clone Detection Methods Used in the Fault-prone Module Prediction, *Proceedings of the 10th Internal Workshop on Software Clones*, pp. 15–16 (2016).

[13] McCabe, T. J.: A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308–320 (1976).

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

[15] Monden, A., Nakae, D., Kamiya, T., Sato, S.-i. and Matsumoto, K.-i.: Software Quality Analysis by Code Clones in Industrial Legacy Software, *Proceedings of the 8th International Symposium on Software Metrics*, pp. 87–94 (2002).

[16] Murakami, H., Hotta, K., Higo, ., Igaki, H. and Kusumoto, S.: Folding Repeated Instructions for Improving Token-Based Code Clone Detection, *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 64–73 (2012).

[17] Murakami, H., Hotta, K., Higo, ., Igaki, H. and Kusumoto, S.: Gapped Code Clone Detection with Lightweight Source Code Analysis, *Proceedings of the 21st International Conference on Program Comprehension*, pp. 93–102 (2013).

[18] Rattan, D., Bhatia, R. and Singh, M.: Software Clone Detection: A Systematic Review, *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199 (2013).

[19] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *Proceedings of the 16th International Conference on Program Comprehension*, pp. 172–181 (2008).

[20] Roy, C. K., Cordy, J. R. and Koschke, R.: Comparison and evaluation of code clone detection techni ues and tools: A ualitative approach, *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495 (2009).

[21] Saini, V., Sajnani, H. and Lopes, C.: Comparing Quality Metrics for Cloned and non cloned Java Methods: A Large Scale Empirical Study, *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, pp. 256–266 (2016).

[22] Smith, T. F. and Waterman, M. S.: Identification of Common Molecular Subse uences, *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195–197 (1981).

[23] amanaka, ., Choi, E., oshida, N., Inoue, K. and Sano, T.: Applying Clone Change Notification System into an Industrial Development Process, *Proceedings of the 21th International Conference on Program Comprehension*, pp. 199–206 (2013).

[24] Zhao, W., Xing, Z., Peng, X. and Zhang, G.: Cloning Practices: Why Developers Clone and What Can Be Changed, *Proceedings of the 28th International Conference on Software Maintenance*, pp. 285–294 (2012).

[25] 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，Vol. 28, No. 3, pp. 28–42 (2011).

[26] 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，Vol. 54, No. 5, pp. 1686–1696 (2013).

[27] 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，Vol. J91-D, No. 6, pp. 1465–1481 (2008).

[28] 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，Vol. 52, No. 9, pp. 2788–2798 (2011).

[29] 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，Vol. 31, No. 1, pp. 14–29 (2014).