Vol.2015-SE-189 No.3
2015/7/22

商用 Android ソフトウェア開発環境における コードレビュー統計の実証的研究

島垣　潤二[†1,a)]　亀井　靖高[†2,b)]　鵜林　尚靖[†2,c)]

概要：コードレビューシステム Gerrit の普及により様々なオープンソースプロジェクトはコードレビューの場を Web に移してきた．Gerrit はレビューやコード改編の記録をアクセスできる形で残すため，その情報を利用したソフトウェア不具合の研究が近年多くなっている．オープンソースデータを使った研究によると，開発履歴から得られるメトリクスで将来の不具合の発生箇所をある程度説明できることがわかっている．本研究では商用 Android ソフトウェア開発履歴データを用いて過去の研究結果との比較を行った．既存研究で提案されるメトリクスのうち，オンライン上の議論に関するメトリクスは将来の不具合の数を高い精度で説明することができなかった ($p \simeq 0.19$)．一方，開発プロセスの違い，特にオフライン上での議論の存在，を考慮したメトリクスは $p \leq 10^{-4}$ という高い信頼性で不具合数を関係づけることができた．

キーワード：コードレビュー, Android, Gerrit

1. はじめに

ソフトウェアの不具合の傾向を開発履歴から得られるデータで説明する研究は広く行われている．例えば，文献 [4] では，コミットの履歴の複雑性を示す Entropy を提案し，用いることで，Entropy 値から不具合の発生する確率を予測することを試みている．文献 [2] ではファイルにおける修正頻度を開発者ごとに計測し Major(コンポーネント内で 5%以上のコミットを作った開発者の数) , Minor(Major 以外の開発者の数), Ownership(最多コミット作成者のコミットの割合) などの開発者に関するメトリクスを定義することで，ファイルに触れる開発者の数とそこでの不具合の発生率には正の相関関係があることを示している．

開発履歴から計測するメトリクスは実装工程の完了後に得られる一方で，実装工程中のコミットのレビューの過程に着目した研究も近年多い．文献 [1] はコードレビューを開発プロセスに導入している企業において，開発者がどういう視点でコードレビューを行っているかを調査し，開発者がレビューで不具合を発見することを必ずしも第一目的としていないことがわかった．文献 [5], [6] ではオープンソースプロジェクトのレビューシステムに蓄積される履歴から計測可能な (レビュー) メトリクスと不具合の発生率の関係性を調査している．

しかしながら，我々の知るところ，現在までのところ企業のレビューメトリクスと将来の不具合を関係づける研究はなされていない．本研究では Sony Mobile Communications, Inc. における Android スマートフォンの開発データを用いて文献 [5], [6] の追実験を行う．また，企業における開発の特性を反映するためのレビューメトリクスを新規に導入する．組み立てたモデルでコンポーネント [*1] 単位でレビューメトリクスと不具合の関係性を調べ，[5], [6] のリサーチクエスションを実験的に再検証する．

2. 想定する開発環境

本研究で扱うソフトウェアプロジェクトにおいて採用されているソースコード管理システムの Gerrit コードレビュー，および開発プロセスの *release branch* 手法について説明する．

2.1 Gerrit コードレビュー

Gerrit コードレビュー [*2] は Android Open Source

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，Sony Mobile Communications Japan, Inc.
Presently with 東京都港区港南 1-8-15 W ビル
[†2] 現在，九州大学
Presently with 福岡県福岡市西区元岡 744 番地
[a)] Junji.Shimagaki@sonymobile.com
[b)] kamei@ait.kyushu-u.ac.jp
[c)] ubayashi@ait.kyushu-u.ac.jp

[*1] 本研究では 1git レポジトリを 1 コンポーネントと扱う．
[*2] http://code.google.com/p/gerrit/ 本研究ではバージョン 2.8.11 を使用

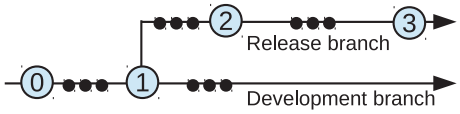

図 1 主開発ブランチ (Development) とリリースブランチ (Release)．● はコミット群，直線はブランチを表す．0 が調査開始地点，1 がリリースブランチの作成地点，2 がソフトウェアリリース地点，3 がポストリリース地点．

Project[*3] に初めて導入されて以来，多くのオープンソースプロジェクトや企業のソフトウェアプロジェクトに採用されてきた．[*4] 古典的なレビューと比較した Gerrit の特徴として，Web ブラウザ上でコードの校閲を行い開発者同士が意見を交換できること，また Git をベースにしているためパッチセットのダウンロードやアップロードが SSH, HTTP プロトコルなどを経由して容易に行うことができること，そしてユーザーグループによる高度なアクセス権限の設定ができることなどがあげられる．

レビュアーはコミットに対してコーディングに関する採点 `Code-Review` を $-2 \sim +2$，また動作確認など仕様に関わる採点 `Verified` を $-1 \sim +1$ のスコアでつけられる．通常，これらで `Code-Review+2` と `Verified+1` の両方が集まることで初めてブランチにコミットを `Submit` ができて，その後コミットは Git レポジトリに登録される [10]．

開発者は何らかの理由によりコードのレビューをスキップして `Submit` を行うために，コードの作者自身がレビューのスコアをつけることがシステム上はできる．このような “Self-Review” はプロジェクトによっては時折発生するものの，第三者によるチェックが行われないためコードレビューシステムを導入している以上推奨される方法ではない．しかしながら，現実的に不具合修正コミットの動作確認を行うための環境構築が困難であったり，また他の要因で開発者自身が動作確認を行うのが最適とみなされる場合には `Verified+1` が作者自身に押されることがある．

2.2 Release branch

大規模ソフトウェア開発においてプロジェクトの中〜後期に release branch(主開発ブランチから派生した特定リリース向けのブランチの呼称) を作ることは特定の不具合群を修正するうえで有効な手段とみなされている [9]．Android Open Source Project は Google Inc. 社内からのコミットと社外の Contribution(開発コミュニティの “貢献” の意でコミットを指す) から成り立っている一方，OS メジャーアップデートの前には**図 1** のように release branch が作成され，そこにはマイルストーンに向けた社内コミットのみを受け付け，それ以外のコミットが入らないような体制が取られている [8]．Sony Mobile Communications, Inc. も類似するプロセスに従い開発を行っている．

本研究では先行研究 [5], [6] と同様，上述の特性を活かしてリリース前の不具合の数（以下 defects.prior）とリリース後の不具合の数（以下 defects.post）を次のように計測する．defects.prior は図 1 の地点 1 と地点 2 の間のコミット群，defects.post は地点 2 と地点 3 の間のコミット群とする．

3. ケーススタディデザイン

初めにリサーチクエスションの設定と調査に必要なシステム，データについて議論する．その後，本研究で用いる線形回帰モデルと各変数の扱いについて説明する．

3.1 リサーチクエスション (RQ)

先行研究 [6] と同様の 3 つの RQ で，異なる観点でレビューに関するメトリクスを計測し，defects.post と関係の強いメトリクスを追実験により調べる．

RQ1: レビューカバレッジと defects.post に関係はあるか

Gerrit を導入していても，レビューを回避してコミットすることは技術的には可能である．しかしレビュアーの目に触れないコミットが増えてしまえば，不具合の混入が増える可能性が高まる．そこで，コードレビューを経てないコミットの数と defects.post に関係があるかを調べる．

RQ2: レビューの積極性と defects.post に関係はあるか

レビューシステムを通過するためだけに実質的なレビューを行わず Approval をつけていては効果的なレビューが行われているとはいい難い．レビュアーがコミットに潜在する不具合を見つけるため積極的なレビュー活動を行うことで不具合の数は減らせると考えられる．ここでは人々のレビューの積極性を数値化し，それらと defects.post との関係を調べる．

RQ3: レビュアーの能力と defects.post に関係はあるか

コミットに存在する不具合を見つけるためにはコードに対する知識が必要不可欠である．そういった知識はコミット数に比例して高くなるという研究がある [2]．そこでレビュアーの能力をコミットの全体に対する割合で評価することで，個々のコミットへのレビューを点数付けて defects.post の関係について調べる．

3.2 調査対象

本研究・調査は Sony Mobile Communications, Inc. により開発された Android スマートフォンソフトウェアプロジェクトの開発データの一部を抽出して行った．プロジェクトの概要を**表 1** に載せる．それぞれ，本研究の Android ソフトウェアプロジェクト全体に関するデータ，そのうち研究の対象となったコンポーネントのデータ，そして比較のために先行研究 [6] で使われたプロジェクト Qt v5.1 の

[*3] https://source.android.com/
[*4] http://blogs.collab.net/git/why-gerrit-is-important-for-enterprise-git

表 1　対象プロジェクトの概要

	プロジェクト全体	本研究の対象	Qt v5.1
コミット数	≃ 20,000	≃ 10,000	7,106
# Authors	-	≃ 1,000	422
# Components	500 ∼ 1,000	220	1,337
# Reviewers	-	≃ 1,000	348
レビューカバレッジ	-	81%	96%

- 会社の秘匿情報に関るため公開できない数値

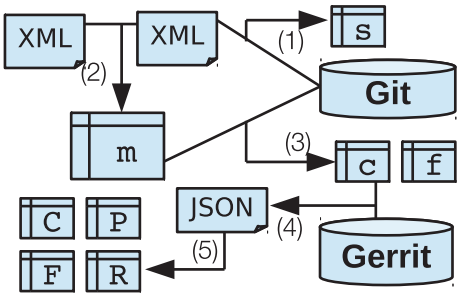

図 2　データ抽出の流れ

データを掲載する．

3.3　データ取得に使うツール

ここでは研究結果の再現性を高めるために Git, Gerrit からどのようにデータを取得するかを説明する．なお，本研究のために開発したスクリプトの一部を github に公開する．[*5]

3.3.1　Repo, Git

git-repo[*6] は Android のために開発された，多数の Git で構成されるシステムを簡易に扱うためのツールである．システム構成は単一の `manifest.xml` という XML ファイルにレポジトリ URL，ダウンロード先のパス，そしてチェックアウトする revision などで記述される．得られた情報を用いて git-repo で Git レポジトリをダウンロードを行った．[*7] そして個々の Git レポジトリには GitPython [*8] でアクセスし，データの取得と加工を行った．

3.3.2　Gerrit

Gerrit は Web インターフェースの他に [7] の研究にも用いられたように，REST API によるデータアクセスを提供している．本研究では実装の容易化のために pygerrit [*9] を用いて REST API アクセスをラップして取得した JSON データを Python データ構造に変換した．

3.4　データ（基本データ）

RQ を議論するために必要なメトリクスは Git, Gerrit から抽出されたデータ（以下，基本データ）で計算できる．ここでは基本データの中身を**表 2** で，取得の流れを**図 2** を用いて説明する．

ソフトウェアリリース時のソースコードの状態を調べるために，`manifest.xml` に記述されたソースコードをダウンロードする．LOC を計測して `s` を得る [図内 (1)]．これまでにどういうコミットが投入され，コードが進化してきたかを知るために，過去の `manifest.xml` も読んで共通に存在するコンポーネントに関して新旧の revision を抽出する [(2),m]．2 つの revision が異なるときは 1 つ以上のコミットが間に存在しているので，ダウンロード済みの git データからコミット情報に関する `c`，ファイル情報に関する `f` を得る [(3)]．

次に個々のコミットがどういう実装工程を経たかを知るために Gerrit から情報を取得する．`c` の revision はコミットの唯一の ID となるのでそれをキーに Gerrit にクエリーを送ることで該当のコミットに関する情報が JSON オブジェクトで得られる [(4)]．オブジェクトをコミットに関する情報 `C`，ファイルに関する `F`，パッチセットに関する `P`，そしてレビューに関する `R` にデータ分割した [(5)]．得られたデータの例として表 2 の `P`(Gerrit patch sets) を説明する．`P` の ID は Gerrit ID と Patchset number の 2 つなので，それら ID を組み合わせたものが 1 レコードとなる．[*10] レコード内のメトリクスには Committer, Author, Uploaded time がある．

3.5　データのフィルタリング

Git データを取得するとき，調査の対象外となるコミットやコンポーネントが存在するので，どのようなフィルタリングを行ったかを説明する．

(1) git コミットのうち merge コミットはブランチと別ブランチ上のコミット群を結合する作業で発生するコミットであり，それ自体はコードを改編することはない [*11] ので，それらは除外する．

(2) コンポーネントのうち，Java/C/C++のソースコードを含まないコンポーネントは除外する．Android では特に XML や設定用ファイルがソースコードの多くを占めており，そのようなファイルは行数が長い一方レビューによってヒューマンエラーを減らすことは比較的難しい．今回はコーディング上の不具合を防ぐためのレビューに着目するため，Android の主要言語以外は除外することにした．

(3) (2) にあてはまらなくとも調査対象期間内に Java/C/C++にソースコードの進化が見られなかったコンポーネントは除外する．

(4) Gerrit review メッセージのうち，Continuous Integra-

*5 https://github.com/yiu31802/gerrit2csv/releases/tag/1.0
*6 https://gerrit.googlesource.com/git-repo
*7 本研究では `manifest.xml` をプログラム的に扱うための repo-manifest も用いている．repo-manifest は https://github.com/sonyxperiadev にて 2015 年に公開予定
*8 https://pypi.python.org/pypi/GitPython/
*9 https://github.com/sonyxperiadev/pygerrit
*10 ある Change (Gerrit ID=500) が 3 つの Patch set を経た場合には 500/1, 500/2, 500/3 という 3 つのレコードが存在する
*11 その限りではないが，例外は少数なので考慮しない

表 2　開発履歴を計測する基本データ

	Content	ID	Metrics
m	Manifests comparison	Project	Path, Left revision, Right revision
s	Code snapshot	Project	LOC (Lines of code)
c	Git log commits	Revision	Project, Committer, Author
f	Git log files	Revision, File name	Churn
C	Gerrit changes	Gerrit ID	Project, Number of patches, Uploaded/Closed time
P	Gerrit patch sets	Gerrit ID, Patchset number	Committer, Author, Uploaded time
F	Gerrit files	Gerrit ID, Patchset number, File name	Churn
R	Gerrit reviews	Gerrit ID, Patchset number, Index	Reviewer, Score, Verified, Message, Time

tion ツールである Jenkins [*12] や Buildbot[*13] が残した自動テスト結果などのメッセージは除外する．理由は，本研究の対象が開発者の行ったレビュー活動だからである．

3.6　プロダクトメトリクス

開発工程を終えたプロダクトに関するメトリクスをプロダクトメトリクスと呼ぶ．**表 3** にメトリクスをまとめる．Name 列がメトリクス名，Source 列にどの基本データテーブルを組み合わせて計算できるかを説明する．

size (コンポーネント行数の総和)

s の LOC を Project ごとに足し合わせる．

entropy (コミット履歴の複雑性)

CF を結合したテーブルの Churn（変更行数）をファイルごとに足しあわせて，コンポーネント c のファイル i の総変更行数 l_i と全ファイル変更行数 $L_c = \sum_i l_i$，そして割合 $p_i = l_i/L_c$ を定義することでコンポーネント c のエントロピー $H(c) = \sum_i^{L_c} \frac{p_i \log_2 p_i}{\log_2 L_c}$ [4] を計算する．

churn (コミット変更行数の総和)

各コミットの Churn を足し合わせる．

major (主要開発者の数)

[2] の定義に従い，コミット数が一定以上 [*14] の開発者の数を計測する．

minor (上記以外の開発者の数)

一定以下だがコミット実績のある開発者の数を計測する．

ownership (主要開発者の占有率)

major によるコミットの全体に対する割合を計測する．

defects.prior (過去の不具合の数)

2.2 で述べたとおり図 1 の地点 1 と地点 2 の間のコミット数をコンポーネントごとに数える．

3.7　レビューメトリクス

開発工程中のレビューに関するメトリクスをレビューメトリクスと呼ぶ．同じく表 3 にまとめる．

[RQ1] review.coverage (レビューカバレッジ)

cCfF を結合したテーブルでコンポーネントごとに全体のコミット数に対して Gerrit システムを経由したコミット数の割合を計測する．

[RQ1] churn.coverage (同じく行数の割合)

コミット数を行数に変換して同様の割合を計測する．

[RQ2] patches.sd (パッチセット間の変更量のバラつき)

コミットの Patch set 1 から Submit されるまでの最終 Patch set の間にレビューによってどれだけファイル修正量の変遷があったかというレビューの効能を $\sigma_{\text{patch.}} = \sigma_L/\mu_L$ で定義する．これは各 Patch set における総変更行数 L をに対しての偏差 σ_L を取り，変更行数による影響を無くすため平均値 $\mu_L = \langle L \rangle$ で割ったものである．各コンポーネントを代表する値として総和を取る．

[RQ2] self.approval/verification (自分で+2/Verified をつけたコミット数)

他人によるレビューをスキップし自分自身で Code-Review+2 をつけたコミットの数，Verified+1 をつけたコミットの数をコンポーネントごとに計測する．

[RQ2] review.window　(レビュー所要時間)

コミットが Gerrit にアップロードされてから Submit されるまでの時間を Review window と定義し，コミットごとに計測する．コンポーネントの代表値としては中央値を取る．

[RQ2] hastily (短いレビュー時間であったコミット数)

コードの行数に対してレビューの時間が極端に短いコミットの数を計測する．

[RQ2] times.discuss (レビューコメントの回数)

コミットに対して作者以外が行ったコメントの回数を計測する．代表値には中央値を取る．

[RQ2] length.discuss (同コメントの長さ)

同じくコメントの長さを計測する．

[RQ2] no.discuss (議論が行われなかったコミット数)

コメントのやりとりがなかったコミット数を計測する．

[RQ3] exp.involve (Expert の参加率)

先の Major を熟練開発者 (Expert) とみなし，そのような開発者がコードレビューもしくはコードの owner としての参加しているコミットの数を計測する．

[RQ3] typical.exp (レビュアーの Expert レベル)

全レビュアーの過去に作成したコミット数を計測し，そ

[*12] https://wiki.jenkins-ci.org/display/JENKINS/Gerrit+Trigger
[*13] http://buildbot.readthedocs.org/en/v0.8.6/manual/cfg-changesources.html
[*14] 全コミット数の 5%以上とした

表 3　プロダクト・レビューメトリクス

Name	Source	RQ1	RQ2	RQ3
size$^{(*)}$	s	$1_{\diamond}$	-	$1_{\diamond}$
entropy$^{(*)}$	CF	-	-	1
churn$^{(\circ)}$	CF	$1_{\diamond}$	†	1
defects.prior$^{(*)}$	C	$5^{***}_{\diamond}$	$5^{***}_{\diamond}$	$5^{***}_{\diamond}$
ownership$^{(\circ)}$	C	†	†	†
minor$^{(*)}$	C	$3_{\diamond}$	‡	†
major$^{(\circ)}$	C	†	†	†
total$^{(*)}$	C	$1^{*}_{\diamond}$	$1^{**}_{\diamond}$	†
review.coverage$^{(*)}$	cC	1		
churn.coverage$^{(\circ)}$	cCfF	†		
⋆ patches.sd	CFP		$1^{***}_{\diamond}$	
⋆ self.verification	CR		$3^{*}_{\diamond}$	
self.approval$^{(\circ)}$	CR		$1_{\diamond}$	
review.window$^{(\circ)}$	C		-	
hastily$^{(*)}$	C		$1_{\diamond}$	
⋆ times.discuss	CR		-	
length.discuss$^{(*)}$	CR		-	
no.discuss$^{(*)}$	CR		†	
exp.involve$^{(*)}$	CR			$3^{**}_{\diamond}$
typical.exp$^{(\circ)}$	CR			$1_{\diamond}$

† Disregarded during clustering analysis
‡ Disregarded during redundancy analysis
- Not used because of low ρ^2
$\diamond$ Log transformation
$*$ Statistical significance ($*$ 0.10, $**$ 0.05, $***$ 0.01)
$^{(\circ)}/^{(*)}$ Weak/Strong significance level appeared in [6]
⋆ New metrics introduced in this work

れを Expert レベルとみなしレビュアーの “Expert” 能力値の中央値を計測する．

3.8　線形回帰モデル

統計モデルには線形回帰モデルを使う．線形回帰モデルでは予測したい値 y_i とそれらを説明する説明変数群 x_{ij} を用いて次の式で表現する．

$$y_i = \sum_{j=0}^{M} \beta_j x_{ij} \qquad (1)$$

ここで $1 \leq i \leq I$ はレコード番号で I は総レコード数，M は説明変数の数，そして係数 β_j となる．式 (1) は線形結合だけでなく，多次項 x^n を新たな変数として定義することで関数のコブを *knots*(節) で表現することもできる．近似曲線の *overfitting* を避けるためあまりに多くの節を割り当てるのはよくないが，最適な自由度は $d.f. = I/15$ で求められる [3].

我々の研究対象に同モデルをあてはめると y_i がコンポーネント番号 i における defects.post の数，x_{ij} がコンポーネント番号 i のメトリクス j の値，I がデータセットにおけるコンポーネントの総数となる．

表 4　レビューカバレッジの統計量

	Min	1st Qu.	Med.	Mn.	3rd Qu.	Max
r_N	0.015	0.716	1.000	0.813	1.000	1.000
r_L	0.001	0.708	1.000	0.792	1.000	1.000

3.9　説明変数の Log/Logit 変換

線形回帰モデルは変数の正規分布を仮定している [3]. Shapiro-Wilk 検定結果で正規分布でない可能性が高い (> 0.95) ときには，分布を正規分布に近づけるため Log 変換を行う．また $0 \leq x \leq 1$ のような変数については Logit 変換を行うこともできる．しかし，今回の研究に限っては結果に影響を与えなかったので，Logit 変換は行わない．

3.10　強相関の説明変数の排除

相関の強すぎる説明変数を同時に式 (1) に入れると，互いの作用を打ち消し合ってよいモデルが得られない．そこで本研究では Spearman 法階層型クラスタリングを行って，2 変数間のそれぞれの相関係数をクラスター図でプロットする．しきい値 $\rho = 0.7$ を超えるものについては代表値 1 つを除いて排除することにする．

また，同様に Spearman の冗長度の確認を R の `hmisc::redun` を用いて計算してそこで出された変数も排除の候補にする．こちらの冗長度は 2 変数間を語る相関係数と異なり，全ての説明変数を考慮したものになる．変数が他の変数によって線形式で表せされ，その線形モデルと実測値の間の R^2 がしきい値 0.9 を超える場合には冗長とみなす．

3.11　モデルの構築と評価

排除されずに残った変数に対して defects.post との相関性を Spearman ρ^2 順位相関係数表を作ることで測り，高いものから順に自由度を割り当てて，式 (1) の β_j を最小二乗法で求める．構築されたモデルは β_j の値がどれだけ 0 から離れているかで評価される．具体的には β_j は t 分布に従うと考えられるので，β_j を t 値に変換することで統計的優位性を判断する．

4.　ケーススタディ結果

ここではケーススタディの結果，リサーチクエスションに対する答え，そして結果の考察を行う．

4.1　RQ1: コードレビューカバレッジと defects.post 修正の数に関係性はあるか

各コンポーネントのメトリクス計測結果を **表 4** に示す．大半のコミットがレビューされていることがわかる．[*15]

[*15] レビューされていないコミットの例として，他のブランチからマージコミットにより流入したコミットがある

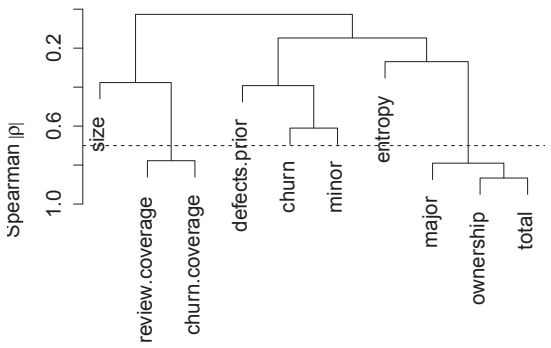

図 3 RQ1: Spearman 法階層型クラスタリング

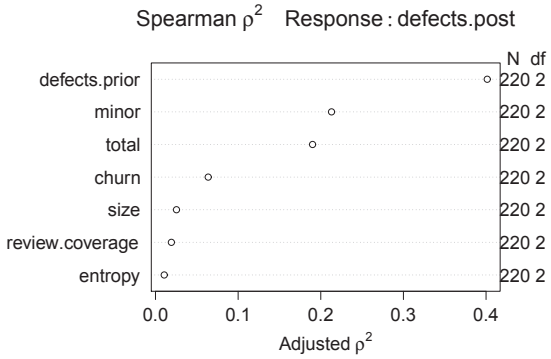

図 4 RQ1: Spearman ρ^2 順位相関係数表

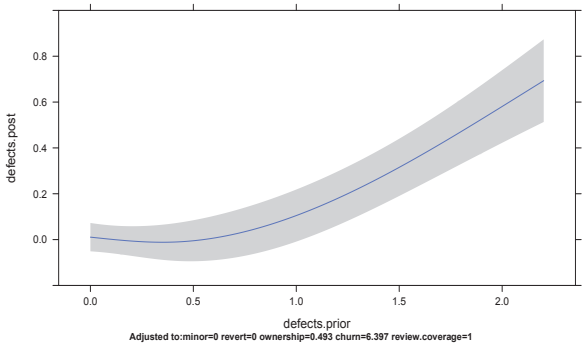

図 5 RQ1: defects.prior に対する defects.post

4.1.1 説明変数の補正と選択

Shapiro-Wilk 検定の結果，全てのメトリクスが非正規分布という結果であったので，Log 変換できるものは変換を行った．そして Spearman 法階層型クラスタリングの結果を**図 3** に載せる．右のクラスターにおいては [6] と同様に total を採用する．冗長とみなされた変数はなかった．最後に順位相関係数表を**図 4** にまとめる．相関の最も強い

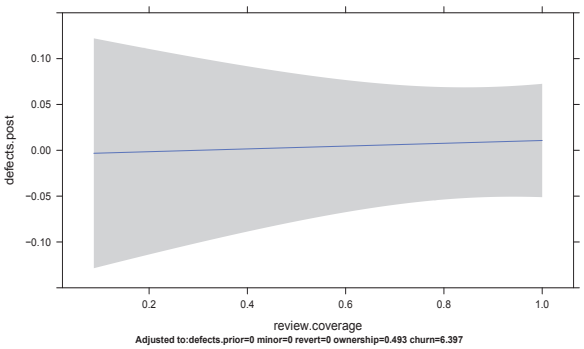

図 6 RQ1: review.coverage に対する defects.post

表 5 RQ1: モデル構築の結果

Name	Coef.	S.E.	t	$Pr(>\|t\|)$
defects.prior (x^5)	0.57	0.14	4.07	10^{-4}
minor	-0.01	0.04	-0.29	0.77
total	0.09	0.05	1.88	0.06
churn	-0.17	0.01	-1.46	0.15
size	-0.17	0.01	-1.28	0.20
review.coverage	0.03	0.08	0.42	0.67
entropy	0.01	0.08	0.09	0.93

defects.prior には自由度 5 を，その次の minor には自由度 3 を，そして残りの変数に自由度 1 ずつを割り当てた結果を表 3 にまとめる．

4.1.2 モデル構築の結果

defects.prior と review.coverage に対する defects.post のプロットと結果を**図 5**, **図 6**, **表 5** に示す．

defects.prior は極めて小さい信頼区間と高い t 値を持ち，defects.post との間には明白な相関関係が見受けられる．一方，review.coverage は同じような比例関係は読み取れず，信頼区間も大きいことから統計的優位性を持ち合わせていないと考えられる．他の説明変数は total が唯一 0.06 の優位性を持つ．

4.1.3 考察

RQ1 においては review.coverage から defects.post を関係づけることはできなかった．その理由は本研究データも，文献 [6] の Qt 5.1 データと同様に平均レビューカバレッジレートが 80%を超えていて極めて高いためであると考える．これらのシステムのように Gerrit レビューシステムが開発プロセスの根幹をなしている場合には，必然的にレビューカバレッジレートは 100%に近付くため，レビューカバレッジの小さいコンポーネントの情報が圧倒的に少なくなる． [6] でもコードレビューカバレッジだけでリリース後の不具合を予測するのは難しいことは結論付けられている．また統計的にも分布の偏りが大きい review.coverage のような値を説明変数として使うことは推奨しないか，もしくは他の変数と組み合わせた変換が必要であるという提案もある [3].

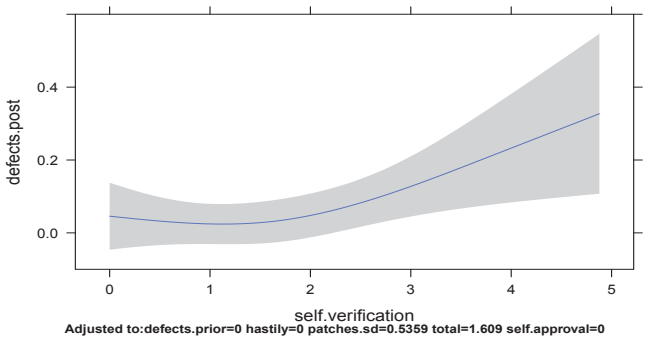

図 7 RQ2: self.verification に対する defects.post

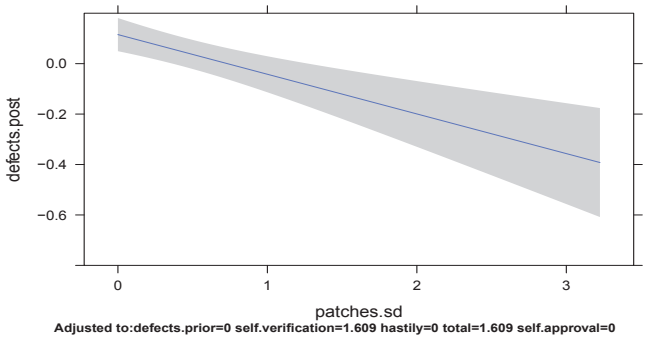

図 8 RQ2: patches.sd に対する defects.post

表 6 RQ2: モデル構築の結果

Name	Coef.	S.E.	t	$Pr(>\|t\|)$
self.approval	0.02	0.03	0.78	0.43
self.verification	-0.03	0.04	-0.71	0.48
self.verification (x^3)	0.11	0.06	1.88	0.07
hastily	-0.06	0.05	-1.32	0.19
patches.sd	-0.16	0.04	-4.16	10^{-4}

4.2 RQ2: レビューの積極性と defects.post に関係性はあるか

4.2.1 説明変数の補正と選択

4.1.1 と同様に Log 変換を行い変数の選択を行った．結果を表 3 にまとめる．RQ2 特有の説明変数の多くは冗長，また，重複とみなされず排除の対象にならなかった．また，RQ1 でも扱った変数の統計量は表では割愛している．

4.2.2 モデル構築の結果

self.verification，patches.sd に対する defetcts.post の結果を**図 7**, **図 8**, **表 6** に示す．

図 7 からは self.verification の割合が高くなるほど defects.post が上昇する明白な比例関係が読み取れる．また，図 8 からはパッチの変遷が大きくなるほど defects.post が大きくなることがわかる．その他，defects.prior は 4.1.2 と同様の関係性があったが，self.approval, hastily からは特筆すべき特徴は見いだせなかった．

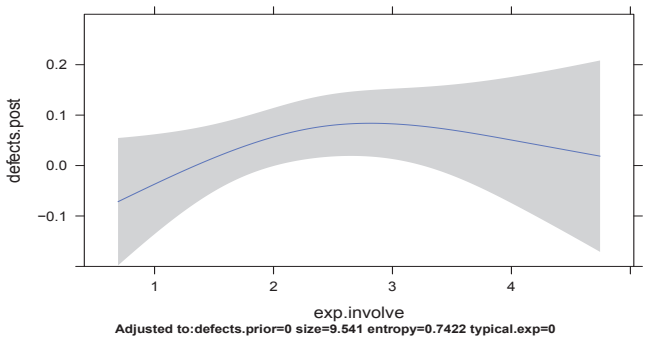

図 9 RQ3: typical.exp に対する defects.post

4.2.3 考察

RQ2 で取り扱ったメトリクスでは self.verification と patches.sd が defects.post に強く関係していることがわかった．これらのメトリクスは [6] で取り扱いがなかったので簡単に比較はできないが，企業内プロジェクトならではの特徴が表れていると推測できる．

Sony Mobile Communications, Inc. では Gerrit においてユーザーグループを厳密に管理して，開発者のコードに対する責任を明確にしている．情報の公開はできないが，本データセットにおける self.approval の割合は self.verification を大きく下回る．それは開発プロセスにおいて self.approval を非推奨としているので，特に重要な不具合修正の場合には他人によるレビューが必須である．一方，self.verification は 2.1 で述べた背景もありそれほど厳しく禁じられてはいない．しかしながらそれに甘んじてしまうと動作確認が個人の開発環境に依存してしまうのは否めなく，事実いくつかの不具合は検知出来なかったのかもしれない．

patches.sd が与えた defects.post に対する強い負の相関，また他の discuss 変数の影響の少なさは，Gerrit に残らないオフラインでの開発者同士の議論の結果と推測できる．オープンソースプロジェクトが様々な国の機関や個人の貢献によって発展するという特性上，人々はオンライン上のコミュニケーションに強く依存しなくてはいけない．それが功を奏して Gerrit には多くの議論の結果が残ることになるので文献 [6] では議論に関するメトリクス，特に no.discuss からは defects.post に対する強い比例関係を見出すことができた．一方，Sony Mobile Communications, Inc. のように開発者が地理的に近い環境で開発を行う場合にはオンラインの議論が必ずしも最適ではなく，直接会って行うこともある．patches.sd はそういった水面下で行われた議論の結果をうまく抽出できたように見える．

4.3 RQ3: レビュアーの能力と defects.post に関係性はあるか

4.3.1 説明変数の補正と選択

RQ3 の変数に関する結果を表 3 にまとめる．typical.exp

表 7　RQ3: モデル構築の結果

Name	Coef.	S.E.	t	$Pr(>\|t\|)$
exp.involve	0.11	0.05	2.10	0.03
exp.involve (x^3)	-0.10	0.06	-1.74	0.08
typical.exp	-0.02	0.03	-0.85	0.40

が ownership,major,minor と強い相関を示したが RQ の変数を優先選択した．

4.3.2　モデル構築の結果

exp.involve, typical.exp に対する defetcts.post の結果を**図 9**, **表 7** に示す．“Expert” のレビューへの参加の割合が高まると最初は defects.post は増加するものの，途中から defects.post は減少した．なお，typical.exp からは統計的優位な結果が得られなかった．

4.3.3　考察

結果を 2 つの領域に分けて考察してみる．まず，値が低い領域の exp.involve 値はそのコンポーネントのコミット数に等しいことが多い．なぜなら，総コミット数が少なければ 1 つでもコミットを作成する開発者は Major とみなされるので，当然，全てのコミットは exp.involve なコミットとなる．これを踏まえると，値の低い領域では総コミット数が増えるほど不具合が含まれる可能性が高くなるという影響がそのまま表れたと考えられる．

次に値の高い領域においては，開発者が Expert とみなされるには複数のコミットを作成する必要がある．そのため，ここでは多くの経験を積んだ開発者によるコミットが増えれば不具合が含まれる可能性が下がるという影響が文献 [6] と同様に表れたと考えられる．

5.　本研究の制約

本研究では単一のソフトウェアプロジェクトにおいての分析であったので，結果の信頼性を高めるために複数のプロジェクトを比較する必要がある．

6.　おわりに

本研究の発見 企業におけるコードレビューデータでソフトウェア不具合数の説明を試みた．オープンソースプロジェクトに関する研究結果と幾つか異なる結果が見いだせた．よりよい関係性を示すために，企業の開発特性を取り入れたメトリクスの導入が必要であった．

データに関して 本研究は Sony Mobile Communications, Inc. のデータを使用して行われたが，本研究成果と同社が販売するいかなる商品の品質には関係はない．

謝辞 内容に関してソニーモバイル（株）田代真樹氏，松尾道人氏，ソニー（株）田中俊彰氏に助言を頂いた．また統計モデルの構築に関して東京大学・土松隆志氏に助言を頂いた．本研究の一部は，日本学術振興会 科学研究費補助金 (若手 A:課題番号 15H05306) による助成を受けた．ソニーモバイル（株）に本研究の機会を頂き感謝の意を表する．

参考文献

[1] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of the Intl. Conf. on Software Engineering*, ICSE, pages 712–721, 2013.

[2] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proc. of the ACM SIGSOFT Symposium and the European Conf. on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, 2011.

[3] Frank Harrell. *Regression modeling strategies : with applications to linear models, logistic regression, and survival analysis*. Springer, 2001.

[4] A.E. Hassan. Predicting faults using the complexity of code changes. In *Proc. of the Intl. Conf. on Software Engineering*, ICSE, pages 78–88, 2009.

[5] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. The impact of code review coverage and code review participation on software quality: A case Study of the Qt, VTK, and ITK projects. In *Proc. of the Working Conf. on Mining Software Repositories*, MSR, pages 192–201, 2013.

[6] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, page To appear, 2015.

[7] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In *Proc. of the Working Conf. on Mining Software Repositories*, MSR, pages 45–48, 2013.

[8] Android Open Source Project. Codelines, branches, and releases. https://source.android.com/source/codelines.html.

[9] Android QAEP service. Android for msm project. https://www.codeaurora.org/xwiki/bin/QAEP/WebHome.

[10] Gerrit Code Review v2.11. Gerrit code review - a quick introduction. https://gerrit-documentation.storage.googleapis.com/Documentation/2.11/intro-quick.html.