論　文

開発者メトリックスに基づくソフトウェア信頼性の分析

松本　真佑[†*a)]　亀井　靖高[†**]　門田　暁人[†]　松本　健一[†]

Analyzing Software Reliability Based on Developer Metrics

Shinsuke MATSUMOTO[†*a)], Yasutaka KAMEI[†**], Akito MONDEN[†], and Ken-ichi MATSUMOTO[†]

あらまし　ソフトウェアの信頼性に影響を及ぼす要因として，ソフトウェアプロダクトの特徴から算出されたメトリックスを用いた信頼性の分析が数多く行われている．本論文ではプロダクトそのものの特性ではなく，プロダクトを作成した開発者の特性（開発者メトリックス）に基づいたソフトウェア信頼性の分析を行う．用いる開発者メトリックスは開発者ごとの変更行数やコミット回数などと，モジュールごとの開発にかかわった開発者の数などである．本論文の分析は以下の四つの仮説に基づく．仮説 1a：バグの混入のさせやすさには個人差がある．仮説 1b：バグの混入のさせやすさは開発者の特性（変更行数やコミット回数など）から判断できる．仮説 2：多くの開発者が変更したモジュールにはバグが混入されやすい．仮説 3：開発者メトリックスは fault-prone モジュールの判別に役立つ．Eclipse プロジェクトから収集したメトリックスデータを用いた分析の結果，すべての仮説が支持され，開発者のバグの混入のさせやすさには少なくとも 5 倍以上の個人差があること，及び多くの開発者が関与したモジュールほど，わずかではあるがバグが混入されやすいことが明らかとなった．

キーワード　開発者メトリックス，ソフトウェア信頼性，バグ混入率，人的要因

1. まえがき

ソフトウェアの品質管理や品質向上を目的として，ソフトウェアプロダクトの特徴を表す尺度（メトリックス）の算出方法が数多く提案されている．一般的にはプロダクトの特徴を表すメトリックスとして，ソースコード行数や Cyclomatic の複雑度，オブジェクト指向メトリックスなどの静的メトリックス [1] が広く用いられている．規模が大きく複雑なモジュールほど理解が難しく品質が低下しやすいことが指摘されており [2], [3]，このような特徴に基づいてテスト計画立案などの品質改善活動が実施される [4], [5]．静的メトリックスのほかにもバージョンアップ時の差分情報から算出された変更メトリックス [6] や，モジュールの依存関係ネットワークから算出された構造メトリックス [7] などが提案されている．

† 奈良先端科学技術大学院大学情報科学研究科，生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, 8916–5 Takayama, Ikoma-shi, 630–0192 Japan
* 現在，神戸大学大学院
** 現在，カナダ Queen's 大学
a) E-mail: shinsuke-m@is.naist.jp

しかし不具合の混入といったソフトウェアの信頼性を低下させる要因は，プロダクト自体の特徴のみならず，プロダクトを作成した開発者の特性に依存する部分も，少なからず存在する．例えば規模や複雑さが同じモジュールでも，経験の浅い開発者が作ったモジュールほど不具合が混入される可能性が高いといえる．また多くの開発者によって変更が加えられたモジュールほど，個々の思考過程の違いや機能の実装手段の混在により意図の読み違いを誘発しやすく，結果として不具合が混入されやすくなると考えられる．このような不具合が混入されやすいモジュールに対しては慎重にレビューやテストを実施することにより，効率良く信頼性を確保することが期待できる．

従来，開発者個々の特性（経験年数やプログラミングスキルなど）によるコーディング速度やデバッグ効率の違いについては数多くの指摘があり [8], [9]，プログラマの行動を計測するシステムも提案されている [10]．その一方で，開発者個々の活動の計測と計測結果に基づく管理に対する批判も存在する [11], [12]．個人の活動の計測という行為そのものが意図せずとも個人の評価につながるという指摘もあり [11]，これらの計測，及び計測結果に基づく分析は避けられてきた．しかし

 ⓒ（社）電子情報通信学会 2010

ながら，肥大化の一途をたどりつつも短期間かつ限られたコストの中で高い信頼性が求められる近年のソフトウェア開発においては，効率的な信頼性の確保が求められていることも事実である．また，本研究は [11] の批判が必ずしも正しいわけではなく，人的要因計測と人事評価を切り離す工夫が可能であると考える（詳細は **5.** のとおり）．そのため，信頼性を低下させ得る要因については定量的な計測・分析に基づいた，正しい理解を得ることが必要であると考える．

本論文では開発者個々の特性とソフトウェア信頼性の関係に対する理解を得ることを目的として，開発者に着目したメトリックスに基づいた四つの仮説について分析を行う(注1)．用いる開発者メトリックスは開発者ごとの変更行数やコミット回数などと，モジュールごとの開発にかかわった開発者の数などである．具体的な仮説としては，仮説 1a：バグの混入のさせやすさに個人差がある，に関してまず分析を行い，この仮説が支持された場合に，仮説 1b：バグの混入のさせやすさは開発者の特性（変更行数やコミット回数など）から判断できる，について分析する．更に仮説 2：多くの開発者が変更したモジュールにはバグが含まれやすい，及び仮説 3：開発者メトリックスがバグを含んでいる可能性の高いモジュール（fault-prone モジュール）の判別に有効である，を実験的に確かめる．分析及び実験の対象としては開発プロダクトの規模が大きく，数多くの開発者が参加している Eclipse の開発プロジェクトから収集したメトリックスデータを用いる．

仮説 1a や仮説 1b，仮説 2 に関しては経験的には正しいと理解されている部分もあり，それを前提とした研究もある [14], [15]．しかし，具体的にどの程度の関係の強さがあるかは明らかにされておらず，また実際のソフトウェア開発データに対する分析の報告事例は少ない．特に Eclipse のような高い能力をもった開発者が集う巨大なソフトウェア開発プロジェクトで，信頼性に対する人的要因の影響があるかについては調査されていない．プロセス改善や新技術の導入を開発現場に促進・説得するためには，経験的な理解ではなく，定量的かつ客観的なデータに基づく科学的な評価結果を示す必要があることが指摘されている [16]．本論文の貢献は，これら経験的に知られた人的要因の影響に対して，一般公開されたソフトウェア開発プロジェクトデータから調査を行い，定量的な分析に基づいた客観的な知見を得たことにある．

なお，本論文ではバグという言葉をソースコード中に含まれる fault（欠陥または不具合）の意味で用い，信頼性の高さを他の信頼性研究 [17], [18] と同様にモジュールに含まれるバグの少なさにより判断する．信頼性は ISO/IEC9126-1 では，その副特性の一つとして成熟性（ソフトウェアが潜在的欠陥を含んでいない程度 [19]）が定義されている．本論文では，信頼性確保のためには，潜在的欠陥を発見・除去して成熟性を確保することが先決であるという考えから，狭義の信頼性として成熟性を取り扱う．

以降，**2.** では本論文における開発者メトリックスの説明と，開発者メトリックスに基づく四つの仮説について述べ，**3.** では分析対象のデータと仮説の分析手段について説明する．**4.** で分析と実験の結果について述べ，**5.** で結果に対する考察を行い，**6.** で関連研究について述べ，**7.** でまとめと今後の課題について述べる．

2. 開発者メトリックスと信頼性に関する仮説

本章では本論文で用いる開発者メトリックスについてと，開発者メトリックスに基づいた四つの仮説について説明する．

2.1 開発者メトリックス

本論文で用いる開発者メトリックスには，観点の違いにより以下の 2 種類のメトリックスが存在する．

- 個々の開発者に付随する開発者メトリックス
- 個々のモジュールに付随する開発者メトリックス

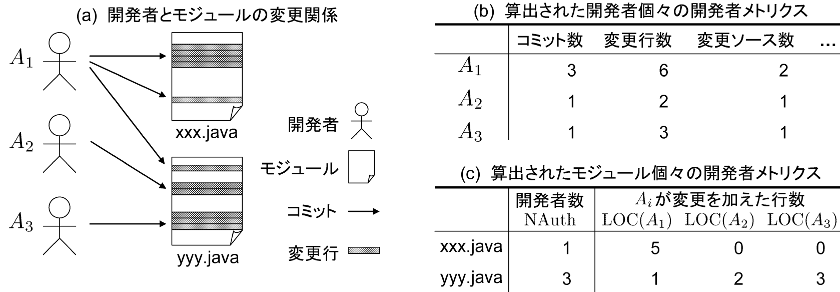
2 種類の開発者メトリックスの例を図 1 に示す．図 1 の上側の図 (a) が開発者と変更されたソースコードの関係を表す．表 (b) が図 (a) をもとに算出された開発者個々のメトリックスを表し，表 (c) が図 (a) をもとに算出されたモジュール個々のメトリックスを表す．開発者個人に着目する仮説 1a と仮説 1b に関しては，開発者個々に付随するメトリックスを用い，モジュールごとの分析を行う仮説 2 と仮説 3 に関しては，モジュール個々に付随するメトリックスを用いる．

2.2 開発者メトリックスに基づく仮説

仮説 1a：バグの混入のさせやすさには個人差がある．

開発者個々のバグの混入のさせやすさに個人差があるかを定量的に明らかにする．従来，コーディング速度やデバッグ効率の個人差については様々な指摘がされており [8], [9]，バグの埋め込みやすさについても個

(注1)：本論文は筆者らの研究会原稿 [13] をもとにデータの再収集，仮説の追加，及び分析の再検討を行い，論文としてまとめたものである．



(b) 算出された開発者個々の開発者メトリクス

	コミット数	変更行数	変更ソース数	…
A_1	3	6	2	
A_2	1	2	1	
A_3	1	3	1	

(c) 算出されたモジュール個々の開発者メトリクス

	開発者数 NAuth	A_i が変更を加えた行数 LOC(A_1)	LOC(A_2)	LOC(A_3)
xxx.java	1	5	0	0
yyy.java	3	1	2	3

図 1 開発者メトリックスの例
Fig. 1 An example of developer metrics.

人差が少なからず存在すると考えられる．この仮説 1a が正しいとすれば，重点的にテストすべきバグ混入率の高い開発者の担当部分を特定することが可能となる．本論文では開発者のバグの埋め込みやすさを表す指標として，1 コミット当りいくつのバグを埋め込んでいたか（以降，バグ混入率）の値を用いる．ソフトウェア進化の分析事例 [20] や可視化研究 [21] などでも，変更行為の 1 単位としてコミットが用いられており，本論文でもこれらを踏襲し単位コミットに基づく分析を行う．なお，このバグ混入率は，バグを埋め込まずにソースコードを作成/変更できる能力であるととらえることができる．

仮説 1b：バグの混入のさせやすさは開発者の特性から判断できる．

仮説 1a の結果よりバグの混入率に個人差があると確認された場合に限り，開発者個々のバグ混入率を個々の特性により判断可能かどうかを分析する．本論文では開発者の特性を表すメトリックスとして開発者個々の，1. コミット数，2. 変更行数，3. 変更したモジュールの数，4. 変更したパッケージの数，5. 前バージョンでのバグの埋込数の五つのメトリックスを用いる．各メトリックスの算出例は図 1 の図 (a) と表 (b) に示すとおりである．これらのうち，コミット数や変更行数，変更モジュール数などは開発者の生産能力やプロジェクトに対するアクティビティの高さを反映しているととらえることができる．これらの特性によりバグの混入のさせやすさを判断できれば，バグが混入されていやすい部分をモジュールの開発担当からある程度推定し，信頼性の改善活動に役立てられる可能性がある．

仮説 2：多くの開発者が変更したモジュールにはバグが混入されやすい．

この仮説の根拠としては，多くの開発者によって変更されたモジュールには，個々の思考過程の違いや複数の機能の実装手段が混在しやすく，理解容易性の低下や意図の読み違いによりバグを埋め込んでしまう確率が高くなると考えられるためである．また逆に 1 人の開発者により作成されたモジュールの場合，理解がしやすくバグ混入率が下がると考えられる．モジュールの規模や変更行数が混入バグ数と高い正の相関があることは数多くの指摘があり [22], [23]，変更を加えた開発者の数と混入バグ数の関係についても分析を行う．この仮説 2 が正しいことが明らかとなれば，開発人数（変更者の数）の多いモジュールを重点的にテストすることで，ソフトウェア信頼性の向上が見込める．

仮説 3：開発者メトリックスは fault-prone モジュールの判別に役立つ．

テストの効率化を目的として，バグが含まれている可能性の高いモジュール（fault-prone モジュール）の判別問題に関する研究が行われている [23], [24]．fault-prone モジュール判別は，統計的な判別モデルを用いてバグが混入されているモジュールをテストの前に特定し，テスト工数を適切に割り振ることでテストの効率化や信頼性の向上を図る手段である．一般に fault-prone モジュールの判別に用いるモデルへの入力としては，ソフトウェアプロダクトから算出されたソースコードメトリックス（静的メトリックス [1] や変更メトリックス [6] など）が用いられる．本論文ではこれら従来のソースコードメトリックスに加え，開発者の特徴から算出されたメトリックス（開発者メトリックス）を加えた実験を行い，開発者メトリックスが fault-prone モジュールの判別精度の向上に寄与するかを確かめる．この仮説 3 は，仮説 1 と 2 の結果をメトリックス活用方法の一つである，fault-prone モジュール予測に役立てられることを確かめるためのものである．

具体的な開発者メトリックスとしては，モジュールごとの変更を加えた開発者の数：NAuth と，ある開発者 A_i の変更を加えた行数：LOC(A_i) の 2 種類を用いる．このうち，LOC(A_i) は個々のモジュールについて，変更を加えた開発者の数 (NAuth) だけ存在するメトリックスとなる ($i = 1..\mathrm{NAuth}$)．図 1 の図 (a)，及び表 (c) に二つの開発者メトリックスの算出例を示す．

3. 分析方法

3.1 分析対象のデータセット

実験には統合開発環境 Eclipse のバージョン 3.x 系列の三つのマイナーバージョン (ver.3.00，ver.3.10，ver.3.20) から収集したメトリックスデータを用いる．Eclipse を分析対象とする理由は，Eclipse は数多くの開発者が参加する大規模なプロジェクトであり，複数の開発者を対象とした分析が可能であるためである．

データの収集条件を表 1 に，収集したデータの概要を表 2 に示す．なお，本論文では 1 ソースコードファイルを一つの意味をもった機能の固まりである 1 モジュールとみなす．

収集したモジュール単位のメトリックスは本論文で提案する開発者メトリックス，及び静的メトリックス，変更メトリックスの三つである．各メトリックスの名称，及びその概要を表 3 に示す．各メトリックスの収集期間は表 4 に示すとおりであり，マイナーバージョンのリリース日を一つの区切りとした．これはリリースが開発の一つの区切りであり，分析対象の全モジュールが一時的にではあるものの開発完了の状態にあるということ，及びその期間が主観によらない客観的な区切りであると考えられるためである．また，各期間はそれぞれ 1 年程度とそろっており，期間の長さが分析結果に与える影響をある程度抑えることができていると考えられる．

開発者メトリックスとしては，変更を行った開発者の数：NAuth と，ある開発者 A_i の変更した行数：LOC(A_i) の 2 種類を収集した．静的メトリックスは Eclipse Metrics Plugin(注7) を用いてソースコードファ

表 1 分析対象データの収集条件
Table 1 Conditions of data collection.

項目	内容
対象プロジェクト	Eclipse Platform(注2)
対象ソースコードファイル	各バージョンリリース時に含まれていたソースコード(注3) すべて
CVS リポジトリ	MSR 2008 が公開しているリポジトリ(注4)
BTS リポジトリ	MSR 2008 が公開している Bugzilla データ(注4)

表 2 データセット概要
Table 2 Summary of dataset.

	ver.3.00	ver.3.10	ver.3.20
ブランチタグ名	R3_0	R3_1	R3_2
ブランチ作成日	2004/6/25	2005/6/28	2006/6/30
開発者数	69	66	72
モジュール数	8,313	9,663	11,525
変更ありモジュール数	7,080	9,428	8,950
バグありモジュール数	2,986	3,302	2,506
コミット回数	61,366	53,302	45,441
混入バグ数	6,351	7,667	4,772
1 コミット当りのバグ混入率	10.7%	17.4%	14.5%

表 3 モジュール単位のメトリックス
Table 3 Source code metrics for each module.

	名称	概要
静的メトリックス	TLOC	総行数
	MLOC	実行行数
	PAR	パラメータ数
	NOF	フィールド数
	NOM	メソッド数
	NORM	オーバライドしたメソッド数
	NSC	サブクラス数
	NSF	静的フィールドの数
	NSM	静的メソッドの数
	NBD	最大ネスト数
	VG	Cyclomatic の複雑度
	DIT	継承の深さ
	LCOM	凝集性欠如の度合(注5)
	WMC	VG の総和
	SIX	特殊化指標の平均(注6)
変更メトリックス	Codechurn	変更行数（追加行数 + 削除行数）
	LOCAdded	追加行数
	LOCDeleted	削除行数
	Revisions	改訂回数
	Age	作成されてからの経過時間（日）
	BugFixes	バグ修正が行われた回数
	Refactorings	リファクタリングが行われた回数
開発者メトリックス	NAuth	変更を加えた開発者の数
	LOC(A_i)	開発者 A_i が変更した行数

(注2)：http://www.eclipse.org/projects/project_summary.php?projectid=eclipse.platform
(注3)：http://archive.eclipse.org/eclipse/downloads/index.php
(注4)：http://msr.uwaterloo.ca/msr2008/challenge/index.html
(注5)：クラスのメソッドが互いに関連している程度であり，値が小さいほどクラス設計の凝集度（メソッドの強度）が高いことを表す．$LCOM = \{1/NOF \times \Sigma_i^{NOF}(u(F_i) - NOM)\}/(1 - NOM)$．$F_i$: i 番目のフィールド，u: フィールド F_i にアクセスしているメソッド数
(注6)：メソッドの継承やオーバライド関係などのクラス設計の複雑さを表す．$SIX = (NORM + DIT)/NOM$．
(注7)：http://eclipse-metrics.sourceforge.net

表 4 各メトリックス・各バージョンの収集時期と収集方法
Table 4 Term and approach of each metric and of each version.

	ver.3.0	ver.3.1	ver.3.2
静的メトリックス	ver.3.0 リリース時のソースコードから計測	ver.3.1	ver.3.2
変更メトリックス	ver.2.1*〜ver.3.0 の変更履歴から計測	ver.3.1〜ver.3.2	ver.3.1〜ver.3.2
開発者メトリックス	ver.2.1〜ver.3.0 の変更履歴から計測	ver.3.1〜ver.3.2	ver.3.1〜ver.3.2
SZZ によるバグ数	ver.2.1〜ver.3.0 に混入されたバグすべて	ver.3.1〜ver.3.2	ver.3.1〜ver.3.2

∗ ver.2.1: ブランチ R3_0 の直前ブランチに該当，ブランチタグ = R2_1，ブランチ作成日 = 2003/3/29

イルから 15 種類のメトリックスを収集した．変更メトリックスはバージョン管理ツールのリポジトリにおけるコミットログから，Moser らの提案する変更メトリックス [6] のうち代表的な（最大値や平均値を除く）メトリックス 7 種類を収集した．

モジュールごとのバグの有無と開発者ごとの混入バグ数に関しては，Śliwerski らの提案する SZZ アルゴリズム [25] を用いて収集した．SZZ アルゴリズムはバージョン管理システムに記録されたソースコードのコミット日時，コミット者，コメントなどの情報と，バグ管理システムに記録された報告バグの登録日時，コメント，バグ ID などの情報を比較することにより，いつ，どのファイルに，だれがバグを混入させたかを特定する方法である．

3. 2 各仮説の分析方法

3. 2. 1 仮説 1a と仮説 1b の分析方法

仮説 1a の分析手段としては，まずバージョン全体での全開発者のバグ混入率 $P(A_i)$ の度数の分布を確認する．更に，どのバージョンでも同程度のバグ混入率をもつ開発者に着目し，そのバグ混入率を比較することにより個人差の有無を確認する．これは度数分布の確認のみでは，バグ混入率の偶然のばらつきによって個人差があるとみなしてしまう可能性があるためである．例えばコミット数の少ない開発者の場合，わずかな混入バグ数の増減によってバグ混入率の値が大きくばらけやすい．このような偶然のばらつきを個人差とみなしてしまうことを避けるために，バージョンごとに開発者個々のバグ混入率のばらつきを考慮に入れて分析を行う．

更に仮説 1a が支持，すなわちバグ混入率に個人差があると確認できた場合のみ，バグ混入率が開発者の特性と相関があるかを調べる．開発者個々の特性を表す指標として **2.** の仮説 1b で述べた五つのメトリックスと開発者のバグ混入率それぞれの単相関係数を算出し，無相関検定により相関の有無を確かめる．

3. 2. 2 仮説 2 の分析方法

仮説 2 を確かめるために，1. 偏相関係数と 2. 重回帰分析を用いた分析を行う．偏相関係数を用いることによりバグ数と開発者数の間の相関の有無を確かめ，重回帰分析を用いることにより開発者数が他のメトリックスと比較してどの程度強い関係をもつかを調べる．

偏相関係数を用いる理由としては，バグ数と開発者数の単相関係数のみを調べた場合，他の変数の寄与による擬相関が検出される可能性があるためである．例えばモジュールの規模（行数）は，バグ数と正の相関をもっており [22]，また開発者数とも正の相関があると考えられる．このような交絡変数が存在する場合，バグ数と開発者数の間が実際には無関係であっても見かけ上の相関（擬似相関）が得られてしまう．そこで本論文では単相関係数による分析に加え，偏相関係数 [26] を用いた分析を行う．偏相関係数は他の変数による寄与を排除した相関のことであり，他変数の値が同一という条件のもとでのある二つの変数間の相関と言い換えることが可能である．偏相関係数を用いることで，上記の擬似相関の問題を避けることが可能である．

ただし，偏相関係数の一つの問題として，すべての変数を寄与排除変数として従属変数 X と目的変数 Y の偏相関係数を調べた際には，X がある変数 A に寄与し，更に A が Y に寄与するといった間接的な関係を確認できなくなるという点が挙げられる．例えば，変更を行った開発者が増加することで，（思考の流れが混在し）プログラムの複雑さが増加，その結果不具合が混入するといった事象は，複雑さメトリックスを除外変数として投入した際には確認できない．しかしこのような，ある変数を媒介して X と Y が関係する，すなわち開発者数とバグ数が間接的な相関関係にあるという分析結果も，現場にとっては役に立つ知見であるといえる．

そこで本論文では寄与排除変数として，二つの行数系メトリックス（総行数，変更行数）のみを用いる．企業などの開発組織においては，ソースコードの信頼性を表す一つの尺度として，バグ数そのものよりもバグ数を行数で正規化したバグ密度が一般的に扱われて

おり，行数がバグ数と強い関係にあることが知られている．行数系メトリックスを排除変数とすることで，バグ密度の考えに従った相関の強さを調べることが可能である．

重回帰分析とは，目的変数と目的変数に関連する複数の説明変数の関係を一次式で表現する手法の一つである．バグ数を目的変数とする重回帰モデルを構築し，推定された各説明変数の標準偏回帰係数を比較することで，バグ数の増加に対する寄与の強さをメトリックス間で比較することが可能である．重回帰モデルの構築に用いるメトリックスとしては，開発者数に加え，偏相関係数の分析と同様に総行数と変更行数の二つを比較対象として用いる．

3.2.3　仮説 3 の fault-prone モジュールの判別実験

仮説 3 を確かめるために，開発者メトリックス，静的メトリックス，変更メトリックスのそれぞれを利用する/しない場合の組合せ計 7 通りの fault-prone モジュール判別実験を行い，各メトリックスの有無による判別精度の変化を比較する．判別モデルとしては一般にバグの判別に用いられる線形判別分析 [27]，ロジスティック回帰分析 [28]，分類木 [29] の三つを用い，バージョン N のモジュールデータに基づいて，バージョン N＋1 のモジュールのバグの有無について判別実験を行う（ver.3.00 から ver.3.10，及び ver.3.10 から ver.3.20 の 2 通り）．

判別結果の評価指標としては Lessmann らの提案する fault-prone モジュール判別問題の比較実験フレームワーク [30] に従い，ROC 曲線 (Receiver Operating Characteristics Curve) の曲線下面積 (AUC：Area Under the Curve) を用いる．ROC 曲線は横軸に False Positive Rate，縦軸に True Positive Rate をとったときの判別結果の変化を表す曲線である．AUC 値は曲線下の面積のことであり，ROC 曲線に対する AUC 値とは判別結果がどの程度擬陽性率を抑えつつ陽性率を確保できるかを表す指標となる．ROC 曲線とそれに対応する AUC 値の例を図 2 に示す．ROC 曲線に対する AUC 値の値域は [0-1] をとり，値が 1 に近いほど判別精度が高くランダムに判別を行った場合におよそ 0.5 の値をとる．この AUC 値は，従来 fault-prone モジュール判別の評価尺度として用いられていた F1 値 [31] や第 1 種過誤，第 2 種過誤と異なり，単一指標での評価が可能，連続値で得られる判別結果を 2 値判別へ区切る際のしきい値に依存しない，データ全体のバグあり/なしモジュールの割合に依存しない結果が得られる，といった利点がある [30]．

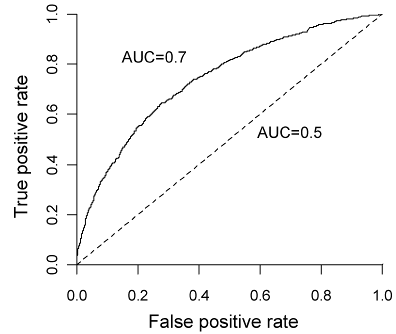


図 2　ROC 曲線と対応する AUC 値の例
Fig. 2　An example of ROC curve and its AUC value.

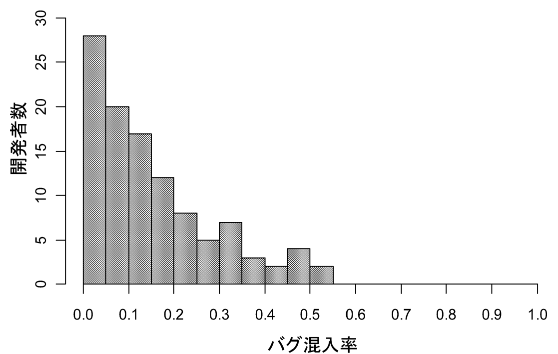


図 3　平均バグ混入率のヒストグラム
Fig. 3　Histogram of average bug injection rate.

4. 実 験 結 果

4.1　仮説 1a：バグの混入のさせやすさには個人差がある

全バージョンにおける全開発者（108 人）の平均バグ混入率のヒストグラムを図 3 に示す．モジュール全体での平均バグ混入率は 17.4%（1 コミット当り混入されるバグ数が約 0.17 個）であり，開発者の全体の約半数はバグ混入率 0〜20%の区間に分布している．一方でバグ混入率が 30%以上の開発者は 17%（18 人）存在しているほか，バグ混入率が 50%を超える開発者も存在しており，バグ混入率にある程度の個人差があることが読み取れる．

次に開発者個々のバージョンごとのバグ混入率を分析する．開発者及びバージョンごとのバグ混入率を図 4 に示す．ここではすべてのバージョンで 1 回以上の変更を加えた開発者 39 人のみを抽出している．破線は全モジュールの平均バグ混入率 (17.4%) を表す．開発者

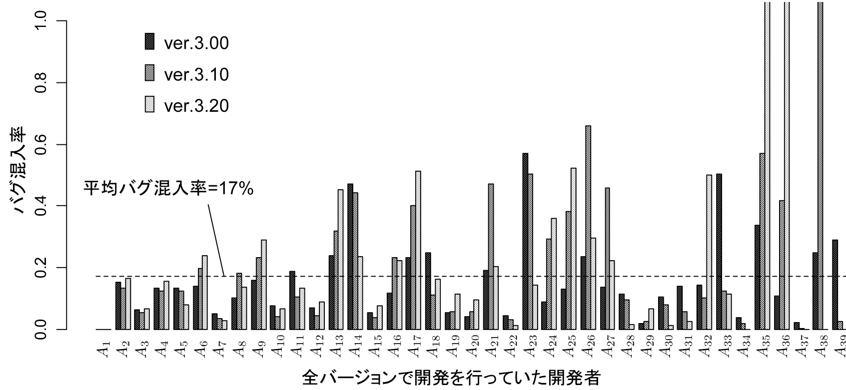


図 4　各開発者のバージョンごとのバグ混入率
Fig. 4　Bug introduction rate for each developer on each version.

のソート基準は 3 バージョンのバグ混入率の変動係数（相対的なばらつき度合）の昇順であり，左側の開発者ほどバージョンごとのバグ混入率のばらつきが小さい．なお，開発者のラベルについては個人名を特定できないようにソート順にインデックスを割り振られている．

図より，すべてのバージョンでバグ混入率が低い開発者 (A_1，A_3，A_7，A_{10}，A_{12}) や，高い開発者 (A_{13-14}，A_{17}) が確認できる．このうち A_{12} と A_{14} は三つのバージョンすべてで 1,000 回以上のコミットを行っていた開発者であるが，それぞれの平均バグ混入率は 0.07 と 0.38 であり，5 倍以上の個人差があることが分かる．また，Friedman 検定を用いて開発者ごとのバグ混入率の差を検定したところ有意な差が確認できた (p 値 = 1.29E-05)．これらのことから，仮説 1a は支持されたといえる．

なお上記の結果は，1 コミット当りのバグ数をバグ混入率として用いた結果であるが，単位行数当りのバグ数を用いても同様の傾向が得られた．1,000 行当りの混入バグ数に基づく結果を付録の図 A·1 と図 A·2 に示す．

4. 2　仮説 1b：バグの混入のさせやすさは開発者の特性から判断できる

開発者個々の五つの特性それぞれとバグ混入率との相関係数を表 5 に示す．表中の「*」は有意水準 5%の無相関検定により相関が有意にあったケースを指す．なお，ver.3.00 の前バージョンでのバグ混入率は，ver.3.00 がバージョン 3 系列の第一リリースバージョンであるため算出不可である．

表より，前バージョンでのバグ混入率は現バージョンでのバグ混入率と正の相関をもっていることが分かる．

表 5　開発者ごとの特性とバグ混入率の相関
Table 5　Single correlation coefficients between bug introduction rate and developer features.

	ver.3.00	ver.3.10	ver.3.20
コミット数	−0.07	−0.19	−0.18
変更行数	−0.05	0.09	−0.14
変更ソースコード数	0.00	−0.14	−0.22
変更パッケージ数	0.35*	−0.06	−0.08
前バージョンでのバグ混入率	算出不可	0.47*	0.32*

*：有意水準 5%の無相関検定により有意に相関あり

これは前バージョンで多くのバグを混入させていた開発者は，次のバージョンでも多くのバグを混入させる可能性が高いことを意味する．一方，変更行数，変更ソースコード数，コミット数に関してはほぼ無相関となっている，変更パッケージの数に関しては ver.3.00 でのみ正の相関をもっているが，他のバージョンでは有意な相関は見られない．すなわち，バグ混入率はコミット数や変更行数などのアクティビティの高さや，変更ソースコード数や変更パッケージ数といったプロジェクトにどの程度幅広く関与しているかといった値では一概に判断できないといえる．

これらの結果からバグの混入のさせやすさは前バージョンでのバグ混入率からのみある程度推測可能であり，仮説 1b は支持されたといえる．

なお，単位行数当りのバグ数をバグ混入率として用いた結果も同様に，前バージョンでのバグ混入率のみ有意な相関が見られた．結果を付録の表 A·1 に示す．

4. 3　仮説 2：多くの開発者が変更したモジュールにはバグが混入されやすい

4. 3. 1　偏相関係数による分析結果

バグ数と変更した開発者の数 (NAuth) の単相関係

表 6　混入バグ数と開発者数の単/偏相関係数
Table 6　Single/Partial correlation coefficients between the number of bugs and the number of developers.

	ver.3.00	ver.3.10	ver.3.20
単相関係数	0.303*	0.389*	0.303*
偏相関係数	0.092*	0.166*	0.104*

*：有意水準 5%の無相関検定により有意に相関あり

表 7　バグ数を目的変数とする重回帰モデルの標準偏回帰係数
Table 7　Standard partial regression coefficients of linear regression models with the number of bugs as dependent variable.

	ver.3.00	ver.3.10	ver.3.20
開発者数：NAuth	0.101	0.164	0.124
総行数：TLOC	0.279	0.091	0.141
変更行数：Codechurn	0.276	0.537	0.383

数及び，総行数 (SLOC) と変更行数 (Codechurn) の影響を取り除いたときの偏相関係数を算出し，その関係の強さを確かめる．バグ数と開発者数の単相関係数と偏相関係数を表 6 に示す．表中の「*」は，有意水準 5%の無相関検定により相関が有意にあったケースを指す．

有意水準 5%の無相関検定により，すべてのバージョンについてバグ数と開発者数は有意に正の相関をもっていた．具体的な値としては，その相関の強さはどのバージョンでも 0.1 程度と弱く，開発者数の増加はバグ数の多さに対して弱く寄与しているといえる．

4. 3. 2　重回帰分析による分析結果

混入バグ数を目的変数とした重回帰モデルを構築し，三つのメトリックス（開発者数，総行数，変更行数）の標準偏回帰係数を比較する．表 7 に三つのメトリックスの標準偏回帰係数をバージョンごとに示す．

バグ数に対して最も強く寄与しているのはどのバージョンにおいても変更行数であった．開発者数の標準偏回帰係数はどのバージョンでも 0.10〜0.16 であり，開発者数はバグ数を目的変数とする重回帰モデルの説明変数として寄与していることが分かる．特に ver.3.10 では開発者数の標準偏回帰係数は総行数よりも大きく，これは総行数よりも開発者数の方がバグの増加に強く寄与しているとみなすことができる．

これらの結果から，同規模・同変更量であれば開発者の数が多いほどわずかではあるがバグが含まれやすくなり，またバージョンによっては開発者数は行数よりも強く寄与するといえる．つまり仮説 2 は支持されたと考えられる．

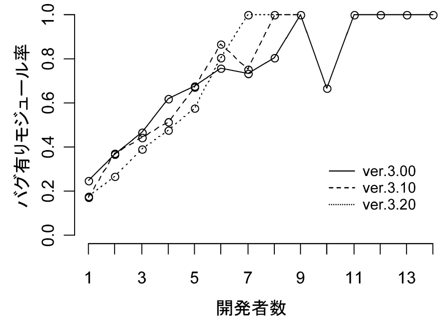


図 5　各開発者数におけるバグありモジュールの割合
Fig. 5　Percentage of faulty modules for each number of developers.

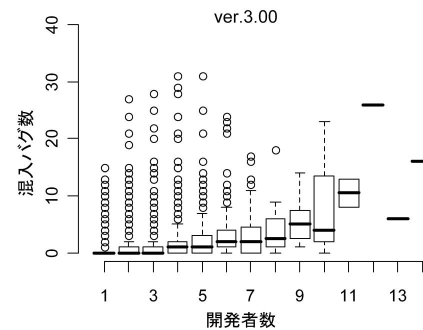


図 6　各開発者数で層別したときの混入バグ数（ver.3.00）
Fig. 6　Number of bugs for each number of developers (ver.3.00).

4. 3. 3　担当開発者数ごとの分析結果

上記の分析により開発者数とバグ数に正の相関があることが確認できたが，担当開発者数ごとの具体的なバグ数について更に詳細な分析を行う．図 5 に，開発者数ごとのバグを一つ以上含んでいるソースコードの割合を示す．図より，どのバージョンについても開発者数が 6 人を超えるソースコードには約 7 割以上の確率でバグが含まれていることが分かる．更に具体的なバグ数を調べるために，ver.3.00 において各開発者数で層別したときの各ソースコードの混入バグ数を箱ひげ図の形で図 6 に示す（紙面の都合上，ver.3.10, ver.3.20 については割愛）．図より，開発者数 1〜5 人のソースコードのほとんど（80%以上）がバグ数 3 個以下となっている．一方，開発者数 6 人以上からは徐々に混入バグ数の多いソースコードが増加しており，開発者数 6 人のソースコードには平均 4.2 個のバグが混入している．この傾向は ver.3.10, ver.3.20 でも同様であった．

これらの結果から，6 人以上の開発者が関与したソー

表 8 fault-prone モジュール判別の結果（ROC 曲線の AUC 値）
Table 8 Result of fault-prone module detection (AUC of ROC curve).

#	メトリックス 静的	変更	開発者	ver.3.00 から ver.3.10 を判別 LDA	LRA	CT	ver.3.10 から ver.3.20 を判別 LDA	LRA	CT	平均ランク
1	o			0.732	0.739	0.697	0.720	0.722	0.650	6.7
2		o		0.818	0.838	0.738	0.893**	0.894*	0.771	3.5
3			o	0.832	0.846	0.657	0.815	0.833	0.660	5.2
4	o	o		0.825	0.849	0.653	0.818	0.842	0.713	4.8
5	o		o	0.820	0.834	0.738	0.888*	0.894**	0.771	3.3
6		o	o	0.861**	0.876**	0.767**	0.883	0.887	0.774**	2.0
7	o	o	o	0.853*	0.872*	0.767**	0.887	0.890	0.774**	2.0

LDA：線形判別分析，LRA：ロジスティック回帰分析，CT：分類木
**：最も精度の高い組合せ，*：2 番目に精度の高い組合せ

スコードには平均で約 4 個のバグが混入しており，また 7 割以上の確率で一つ以上のバグが混入するということから，関与した開発者が 6 人以上という点はテストやレビューの計画を立てる際に注意すべき一つの基準であるといえる．

4.4 仮説 3：開発者メトリックスは fault-prone モジュールの判別に役立つ

3 種類のメトリックスの組合せ 7 通りを用いて，次バージョンの fault-prone モジュールの判別実験を行った際の結果（ROC 曲線の AUC 値）を表 8 に示す．メトリックスの列の「o」は該当するメトリックスを判別に用いた場合を指しており，一番下の行がすべてのメトリックス（表 3 に示すメトリックスのすべて）を用いた場合の結果を表す．表中の平均ランクとは各モデルごとに判別精度の順位付けを行った際の平均の順位を表しており，値が小さいほど精度が高いメトリックスの組合せであることを意味する．AUC 値の横の「**」と「*」は，ある判別モデルを用いた際に判別精度が最も高かったメトリックスの組合せと，2 番目に精度が高かったメトリックスの組合せを表す．

まず，各メトリックス単体で判別を試みた場合に着目すると，変更メトリックス（2 行目）が最も精度が高く，次いで開発者メトリックス（3 行目）であり，従来用いられていた静的メトリックス（1 行目）は最も精度が低かった．また開発者メトリックスを加える場合と加えない場合で平均ランクを比較すると，静的メトリックスは 6.7（1 行目）から 3.3（5 行目），変更メトリックスは 3.5（2 行目）から 2.0（6 行目），静的と変更メトリックスの組合せは 4.8（4 行目）から 2.0（7 行目）と，開発者メトリックスを加えることによる精度の改善が確認できる．開発者メトリックスとの組合せにより判別精度の向上が確認できたため，仮説 3 は支持されたといえる．

5. 考　　察

5.1 研究の制約と貢献

本論文では開発者の特性と信頼性の関係の理解を目的として，開発者個々の特性に基づいた分析を行ったが，これらの計測手段及び分析方法を用いて開発者個人の評価を行うことは適切ではない．ソフトウェア開発において個人の活動を計測することに対する批判は数多く存在する [11], [12]．ソフトウェア開発のような知識生産では，生産性やバグの混入率のような指標のみでは個人の能力を判断できない [11] ことがその理由の一つである．

例えば，仮説 1b で示した A_{14} の場合，そのバグ混入率はすべてのバージョンで平均よりも高かった（図 4 参照）．しかし，これは A_{14} の開発担当パッケージの難易度の高さに起因している可能性もあり，A_{14} がバグを埋め込みやすい開発者と考えるべきではない．特に仮説 1b でのバグ混入率は前バージョンと次バージョンで正の相関をもつという結果については，開発担当の難易度が原因である可能性もあり，また開発者の学習効果も存在するため，多くのバグを混入させた開発者のすべてが次バージョンでも多くのバグを混入させるとは断言できない．更に A_{14} は全バージョンで 1,000 回以上のコミットを行っていた．分析対象とした Eclipse のようなオープンソースソフトウェアの開発では，ある程度バグが混入していても頻繁にリリースするべきである [32] という指摘もあり，この観点では A_{14} は Eclipse 開発プロジェクトに大きく貢献していた開発者であるともいえる．

このように開発者の能力は表層的な指標のみで判断することは難しく，また数値化しにくい部分も多いため，これらの指標に基づいた開発者の評価を行うことは不適当である．

なお，本論文では Austin の批判 [11] が必ずしも正しいわけではなく，人的要因の計測という行為を人事考課につなげない工夫が可能であると考える．例えば，人的要因の計測と分析，それに基づく改善活動を，人事権とは独立した品質保証部門のみが実施，利用するといった品質改善プロセスの工夫を行うことにより，開発者特性の計測と人事評価のつながりを排除することができると考える．

本論文の貢献は開発者個々の評価手段ではなく，定量的な分析に基づいたソフトウェア開発に対する以下の理解を得たことにある．

- バグ混入率はプログラミングスキルやコーディング速度と同様に個人差がある．
- バグ混入率は前バージョンと現バージョンで正の相関をもつ．
- 規模や複雑さと同様に，変更を加えた開発者数はバグ数と正の相関をもつ．
- 開発者メトリックスは fault-prone モジュールの判別精度向上に寄与する．

これらの知見に基づいた信頼性確保のための活動としては以下のような点が考えられる．

- テスト計画立案の際に，バグを混入しやすい（新人の）開発者の作成部分に対して，バグを混入しにくい（ベテランの）開発者の 5 倍以上のバグ混入を見込んだテスト工数の割当を行う．
- 前バージョンでバグ混入率が高かった開発者の担当部分には重点的にテストを行う．ただし，この開発者個々のバグ混入率を利用する際には慎重な判断が必要である．この指標値は品質保証部門などの開発チームから独立した部門が品質改善や品質管理のためのみに役立てる．
- 多くの開発者が関与しているモジュールに対しては，開発担当者割当の再検討を行う．若しくは重点的にレビューやテストを実施する．特に 6 人以上の開発者が関与したモジュールには，より重点的なテスト工数の割当を行うべきである．
- fault-prone モジュールの判別を行う際には，一般に用いられている静的メトリックスや変更メトリックスのみならず，開発者メトリックスを加えることで判別精度の向上が見込まれる．

5.2 一般的な開発形態に対する考察

本論文では OSS の開発データを題材として分析を行ったが，得られた結果は以下の理由により，一般的な開発組織による開発形態に対してもある程度の妥当性を確保できていると考えられる．

分析対象とした Eclipse 開発プロジェクトではコミッターとして認証された開発者以外は，開発リポジトリに変更を加えられない運営方式をとっている．そのため本研究で扱ったすべての開発者は Eclipse 開発プロジェクトに認められた開発者であり，高い実装能力，コーディング技術をもっていると推測できる．このような高い技術をもった開発者集団の中でも，バグ混入率の個人差や信頼性に対する人的要因の影響が少なからずあったことから，一般企業のような新人開発者からベテラン開発者まで幅広い人材がそろう組織では，本研究で得られた結果以上の人的要因が働くと考えられる．したがって，仮説 1a で得られた 5 倍以上の個人差が発生するという結果は，企業においてはそれ以上の差が発生する可能性がある．また，fault-prone モジュールの判別を従来の開発形態に導入する際には，開発者メトリックスが本論文で示した結果以上に精度改善に役立つ可能性がある．

6. 関連研究

ソースコードなどのソフトウェアプロダクトの特徴から計測されたメトリックスに基づいた信頼性に対する研究が多数行われている [22], [23], [33]〜[37]. 特にモジュールの規模に基づいた信頼性の分析は，これまで数多くの報告がなされている [22], [23], [33]. Gaffney [33] は規模の増大によりバグ数が増加することを指摘しており，Koru ら [22] は規模が大きいモジュールほど混入バグ数は多いが，単位規模当りのバグ数（バグ密度）は規模の小さいモジュールほど多いことを指摘している．規模の他にも複雑さに基づいた分析 [34], [35] や，オブジェクト指向言語のメトリックスに基づいた分析 [36], [37] などが行われている．これらの研究は，規模や複雑さといったプロダクトの特徴と信頼性の関係の分析であるが，プロダクトの開発プロセス，特に開発者に着目した分析事例は少ない．

開発者の特性に基づく分析結果の報告事例として Schröter ら [38] の研究がある．Schröter らはバグ管理システムからのデータマイニング方法を示し，その分析結果の一部として本論文の仮説 1a に該当するバグ混入率には個人差があること，及び仮説 1b の一部に該当するバグ混入率と変更ファイル数の間には関連がないことを確かめている．この結果は本論文の結果とも一致する．Schröter らの分析結果は単一バージョンの分析結果である一方，本論文では複数のバージョ

ンで分析を行っており，また変更ファイル数のみならずコミット数や変更行数，変更モジュール数などの複数の指標と混入バグ数の関連を調べている点で異なる．

過去のバージョンのメトリックスデータを用いて，次期バージョンにバグが残存するかの判別を試みる fault-prone モジュール判別に関する研究も数多く実施されている [6], [23], [24]．Moser ら [6] はバージョンアップ時の差分情報を用いた変更メトリックスを提案し，判別精度の向上を試みている．本論文の仮説 2 と仮説 3 で用いた変更メトリックスはこの研究の提案に基づく．Moser らの提案する変更メトリックスには変更を加えた開発者の数というメトリックスが含まれているが，実験では 17 種の変更メトリックス（変更行数やコミット数など）すべてを加えたときの判別精度の改善を示しており，開発者数そのものが混入バグ数の増加に寄与しているか（本論文の仮説 2 に該当），またどの程度 fault-prone 判別の精度向上に寄与しているか（仮説 3 に該当）については明らかにされていない．

fault-prone モジュール判別に開発者の情報を用いた事例として，Weyuker ら [39] による研究がある．この研究では開発者の増加がバグの増加と関連をもつかという本論文の仮説 2 に関連する分析を行っている．分析の結果として Weyuker らは，バグの増加は様々な要因によるものであり，開発者数の増加は主たる原因にはならないと結論づけている．しかし，Weyuker らの分析方法は従来の静的メトリックスを用いた fault-prone モジュール判別モデルに対して，開発者に関するメトリックスを加えた際の判別精度の変化が小さいことから得たものである．この分析手段は判別モデルの精度改善幅の確認による間接的な方法であり，開発者数と混入バグ数が直接的にどのような関係をもつか（もたないか）については言及されていない．一方，本論文では開発者数とバグ数の偏相関係数を調べており（仮説 2），バグ数と強い相関をもつ [6], [33] とされる行数と変更行数の影響を取り除いている点，及びその結果として開発者数とバグ数には少なからず正の相関があることを確かめている点で異なる．更に fault-prone モジュール判別に対する開発者メトリックスの寄与の分析（仮説 3）においては，複数の判別モデル，及び複数のメトリックスの組合せについて詳細に実験している点で異なる．

7. むすび

本論文では開発者個々の特性と，ソフトウェア信頼性の関係に対する理解を得ることを目的として，開発者メトリックスに基づいた四つの仮説について分析を行った．

Eclipse プロジェクトのデータを用いた分析の結果，バグ混入率には 5 倍以上の個人差があること，及び多くの開発者が変更を加えたモジュールほどバグが混入されやすいことが分かった．また開発者メトリックスを用いた fault-prone モジュールの判別実験により，開発者メトリックスが判別精度の向上に寄与することが確認できた．

本論文では Eclipse プロジェクトのみを対象とした分析を行ったが，人的要因は開発プロジェクトに依存する部分も大きい．結果の一般性を向上させるために他のプロジェクトでも同様の分析を行う予定である．また，開発者数とバグ数に正の相関があることは確かめられたが，その具体的な因果関係の有無の調査や，IT スキル標準 (ITSS) に基づくインタビューなどにより収集可能な，開発者の定性的な能力を考慮に入れた調査は今後の課題である．

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．また特別研究員奨励費（課題番号：21・8995）の研究助成を受けて行われた．

文　献

[1] S.G. Crawford, A.A. McIntosh, and D. Pregibon, “An analysis of static metrics and faults in C software,” J. Syst. Softw., vol.5, no.1, pp.37–48, 1985.

[2] K.S. Lew, T.S. Dillon, and K.E. Forward, “Software complexity and its impact on software reliability,” IEEE Trans. Softw. Eng., vol.14, no.11, pp.1645–1655, 1988.

[3] J.C. Munson and T.M. Khoshgoftaar, Software metrics for reliability assessment, McGraw-Hill, 1996.

[4] R.B. Grady, “Successfully applying software metrics,” Comput. J., vol.27, no.9, pp.18–25, 1994.

[5] A.H. Watson and T.J. McCabe, “Structured testing: A testing methodology using the cyclomatic complexity metric,” Technical Report, Nat’l Inst. of Standards and Technology, 1996.

[6] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” Proc. Int’l Conf. on Softw. Eng. (ICSE ’08), pp.181–190, 2008.

[7] T. Zimmermann and N. Nagappan, “Predicting defects using network analysis on dependency graphs,” Proc. Int’l Conf. on Softw. Eng. (ICSE ’08), pp.531–540, 2008.

[8] H. Sackman, W.J. Erikson, and E.E. Grant, “Ex-

ploratory experimental studies comparing online and offline programming performance," Commun. ACM, vol.11, no.1, pp.3–11, 1968.

[9] D.E. Egan, "Individual differences in human-computer interaction," in Handbook of Human-Computer Interaction, pp.543–568, Elsevier Science, 1988.

[10] Y. Takada, K. Matsumoto, and K. Torii, "A programmer performance measure based on programmer state transitions in testing and debugging process," Proc. Int'l Conf. on Softw. Eng. (ICSE '94), pp.123–132, 1994.

[11] R.D. Austin, Measuring and managing performance in organizations, Dorset House Publishing Company, 1996.

[12] T. Demarco and T. Lister, Peopleware: Productive projects and teams, 2nd ed., Dorset House Publishing Company, 1999.

[13] 桝本真佑，亀井靖高，門田暁人，松本健一，"開発者メトリクスを用いたソフトウェア信頼性の分析," ソフトウェア工学の基礎 XVI，日本ソフトウェア科学会 (FOSE '09)，pp.207–214, 2009.

[14] M.E. Nordberg, III, "Managing code ownership," IEEE Softw., vol.20, no.2, pp.26–33, 2003.

[15] 岩間　太，中村大賀，"コーディングスタイルに基づくメトリクスを用いたソースコードからの属人性検出," ソフトウェア工学の基礎 XV，日本ソフトウェア科学会 (FOSE '08)，pp.129–134, 2008.

[16] 松本健一，井上克郎，鶴保征城，鳥居宏次，"学と産の連携による基盤ソフトウェアの先進的開発：5. 産官学連携によるエンピリカルソフトウェア工学の実践データに基づく実証的アプローチ," 情報処理，vol.49, no.11, pp.1257–1264, 2008.

[17] N.E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Trans. Softw. Eng., vol.25, no.5, pp.675–689, 1999.

[18] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting fault incidence using software change history," IEEE Trans. Softw. Eng., vol.26, no.7, pp.653–661, 2000.

[19] 山田　茂，ソフトウェア信頼性モデル—基礎と応用（実践ソフトウェア開発工学シリーズ），日科技連，1994.

[20] D.M. German, "Using software trails to reconstruct the evolution of software: Research articles," J. Softw. Maintenance and Evolution, vol.16, no.6, pp.367–384, 2004.

[21] D.M. German and A. Hindle, "Visualizing the evolution of software using softchange," Int'l J. Softw. Eng. and Knowledge Eng., vol.16, no.1, pp.5–22, 2006.

[22] A.G. Koru, D. Zhang, K.E. Emam, and H. Liu, "An investigation into the functional form of the size-defect relationship for software modules," IEEE Trans. Softw. Eng., vol.35, no.2, pp.293–304, 2009.

[23] N. Nagappan, L. Williams, J. Hudepohl, W. Snipes, and M. Vouk, "Preliminary results on using static analysis tools for software inspection," Proc. Int'l Symp. on Softw. Reliability Eng. (ISSRE '04), pp.429–439, 2004.

[24] A.E. Hassan, "Predicting faults using the complexity of code changes," Proc. Int'l Conf. on Softw. Eng. (ICSE '09), pp.16–24, 2009.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," Proc. Int'l Conf. on Mining Softw. Repositories (MSR '05), pp.1–5, 2005.

[26] R.A. Fisher, "The distribution of the partial correlation coefficient," Metron, vol.3, pp.329–332, 1924.

[27] R.A. Fisher, "The use of multiple measurements in taxonomic problems," Annals Eugenics, vol.7, Part II, pp.179–188, 1936.

[28] D.W. Hosmer and S. Lemeshow, Applied logistic regression, Wiley, New York, 1989.

[29] L. Breiman, J. Friedman, R. Olshen, and C. Stone, Classification and regression trees, Wadsworth, California, 1984.

[30] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," IEEE Trans. Softw. Eng., vol.34, no.4, pp.485–496, 2008.

[31] J.L. Herlocker, J.A. Konstan, L.G. Terveen, and J.T. Riedl, "Evaluating collaborative filtering recommender systems," ACM Trans. Information Systems, vol.22, no.1, pp.5–53, 2004.

[32] E.S. Raymond, The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary, O'Reilly and Associates, 1999.

[33] J.E. Gaffney, "Estimating the number of faults in code," IEEE Trans. Softw. Eng., vol.10, no.3, pp.584–585, 1984.

[34] N.F. Schneidewind and H.-M. Hoffmann, "An experiment in software error data collection and analysis," IEEE Trans. Softw. Eng., vol.5, no.3, pp.276–286, 1979.

[35] N.E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," IEEE Trans. Softw. Eng., vol.26, no.8, pp.797–814, 2000.

[36] L.C. Briand, J. Wüst, J.W. Daly, and D.V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," J. Syst. Softw., vol.51, no.2, pp.245–273, 2000.

[37] R. Subramanyam and M.S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," IEEE Trans. Softw. Eng., vol.29, no.4, pp.297–310, 2003.

[38] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk...," Proc. Int'l Symp. on Empirical Softw. Eng. (ISESE '06), pp.18–20, 2006.

[39] E.J. Weyuker, T.J. Ostrand, and R.M. Bell, “Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models,” Empirical Softw. Eng., vol.13, no.5, pp.539–559, 2008.

付　　録

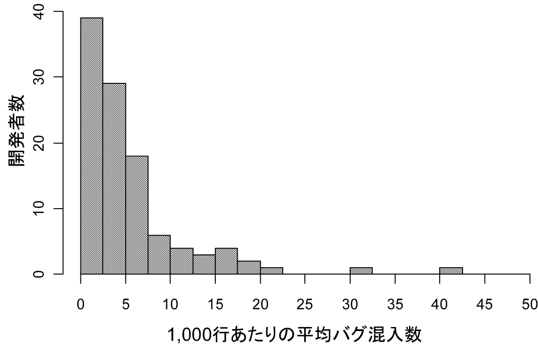


図 A·1　平均バグ混入率のヒストグラム（単位行数）
Fig. A·1　Histogram of average bug injection rate (per 1,000 lines).

表 A·1　開発者ごとの特性とバグ混入率（単位行数）の相関
Table A·1　Single correlation coefficients between bug introduction rate (per 1,000 lines) and developer features.

	ver.3.00	ver.3.10	ver.3.20
コミット数	−0.17	−0.14	−0.21
変更行数	0.13	−0.09	−0.21
変更ソースコード数	0.44*	0.08	0.01
変更パッケージ数	0.05	−0.17	−0.17
前バージョンでのバグ混入率	算出不可	0.53*	0.37*

*：有意水準 5%の無相関検定により有意に相関あり

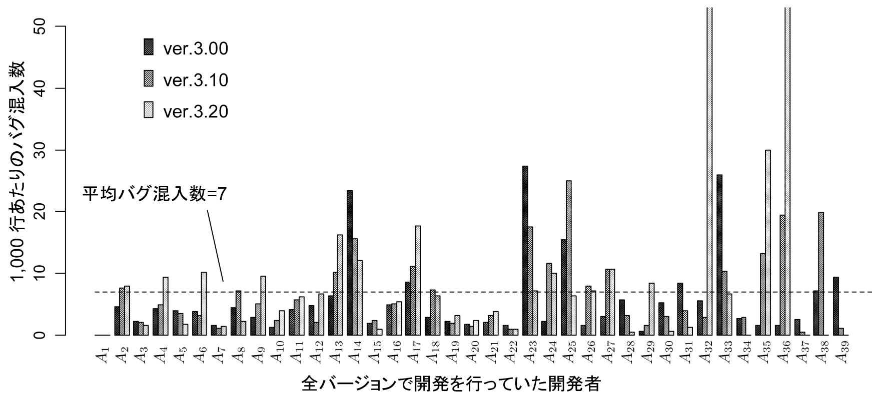


図 A·2　各開発者のバージョンごとのバグ混入率（単位行数）
Fig. A·2　Bug introduction rate (per 1,000 lines) for each developer on each version.

（平成 21 年 11 月 30 日受付，22 年 3 月 25 日再受付）

松本　真佑　（正員）

平 18 京都産業大・理卒．平 21 日本学術振興会特別研究員 (DC2)．平 22 奈良先端科学技術大学院大学情報科学研究科博士後期課程了．同年神戸大学大学院システム情報学研究科特命助教．博士（工学）．エンピリカルソフトウェア工学，特にソフトウェアメトリックスの研究に従事．情報処理学会，IEEE，ACM 各会員．

亀井　靖高　（正員）

平 17 関西大・総合情報卒．平 21 奈良先端科学技術大学院大学情報科学研究科博士後期課程了．同年日本学術振興会特別研究員 (PD)．平 22 カナダ Queen’s 大学博士研究員．博士（工学）．ソフトウェアメトリックス，マイニングソフトウェアリポジトリの研究に従事．情報処理学会，IEEE 各会員．

門田　暁人　（正員）

平 6 名大・工・電気卒．平 10 奈良先端科学技術大学院大学情報科学研究科博士後期課程了．同年同大学助手．平 16 同大学助教授．平 19 同大学准教授．平 15〜16 Auckland 大学客員研究員．博士（工学）．ソフトウェアメトリックス，ソフトウェアプロテクション，ヒューマンファクタの研究に従事．情報処理学会，日本ソフトウェア科学会，IEEE，ACM 各会員．

松本　健一　（正員）

昭 60 阪大・基礎工・情報工学卒．平元同大大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平 5 奈良先端科学技術大学院大学助教授．平 13 同大学教授．工博．エンピリカルソフトウェア工学，特に，プロジェクトデータ収集/利用支援の研究に従事．情報処理学会，日本ソフトウェア科学会，ACM 各会員，IEEE Senior Member.