ソフトウェア変更時のコーディング規約違反と不具合の共起傾向の調査

田口 健介[1]　名倉 正剛[2]　高田 眞吾[3]

概要：ソフトウェア開発において，レビューやテストといった品質保証活動を効率的に行うため，ソフトウェアの変更に対する不具合予測手法の研究が行われている．既存手法の多くは教師あり学習により不具合予測を実施する手法である．これらの手法の利用には学習のためのコストが大きい．一方で近年では教師なし学習による不具合予測手法も提案されている．しかし予測精度が高い方法を利用するためには，ある程度の学習時間が必要になる．そこで，我々は異なったプロジェクト間で共通的に利用できるメトリクスとして，コーディング規約違反に基づくメトリクスに着目した．ソフトウェアのコーディングスタイルには，ソフトウェア開発者間の共通的な慣例として一般的に守られるべきスタイルが存在する．それらのスタイルに基づく規約に対する違反をメトリクスとして利用できれば，学習コストをかけずに不具合予測を実施できるのではないかと考えた．そこで本論文では，事前調査として実施したコーディング規約違反と不具合の共起傾向を調査した結果を報告する．

1. 序論

ソフトウェア開発では，レビューやテストといった品質確保のための活動を，効率的に実施する必要がある．このため，重点的に実施すべきモジュールを早期に予測する不具合予測手法の研究が行われている [1]．不具合予測とは，過去のソフトウェア開発の履歴情報やソースコードから取り出された特徴を利用し，ソフトウェアの変更に対して不具合が混入していそうなソフトウェアモジュールやファイルを予測することである．予測結果を利用することで，効率的なレビューやテストを実施できるようになり，リリース後のソフトウェアの不具合を減らすことができる．

そして，ソフトウェアプロジェクトで過去に実施された変更について変更前後のソースコードと不具合発生の関係を学習することにより，プログラム変更時の不具合発生を予測する予測手法が，古くから提案されている [2][3][4][5][6][7]．これらの手法では，まずソースコードや開発履歴情報を，構成管理システム等のソフトウェアリポジトリから取得する．そして取得した情報から，変更作業の前後の変化に対する不具合発生有無をラベル付けすることにより，教師あり学習を実施する．具体的には，ソースコードやその構造の変化パターンや，開発履歴情報を利用して得られたメトリクス値の変化を，実際のバグ発生有無によってラベル付けし，分類器に学習させる．それにより構築された学習モデルに基づき，開発されたソフトウェアに対してメトリクスを抽出し，不具合を予測する．プロジェクトの進行とバグ発生有無の関係性は，対象プロジェクトの属性（開発チームの構成，プロジェクトの規模，開発期間等）に影響を受ける．そのためこれらの既存手法では，対象プロジェクトの十分な量のプロジェクトデータを教師データとして学習させることで，不具合予測を実施する．したがってこれらの手法の利用には学習のためのコストがかかり，変更が多く実施されているプロジェクトでないと適用が難しい．

一方で，変更時のコミットに含まれる変更差分情報を分類することによる，教師なし学習による不具合予測手法も提案されている [8][9]．これらの手法は，教師データが不要であり学習コストが低い．しかし，変更前後のメトリクス変化から不具合予測を実施する手法 [8] は，メトリクス変化の特徴が対象プロジェクトに依存するためそれほど予測精度が高くなく，ソースコード片自体を分類することで予測する手法 [9] では，分類にある程度の処理時間を要する．

対象プロジェクトに依存せずに，異なったプロジェクト間で共通的に利用できるメトリクスを利用できれば，これらの問題を解決できる．そこて我々は，そのためのメトリクスとして，コーディングスタイルに対する規約違反に基づくメトリクスに着目した．ソフトウェアのコーディング規約には，プロジェクト個別に定められたものだけでなく，ソフトウェア開発者間の共通的な慣例として一般的に守ら

[1] 慶應義塾大学大学院理工学研究科
[2] 南山大学理工学部
[3] 慶應義塾大学理工学部

© 2018 Information Processing Society of Japan

れるべきスタイルに基づく規約が存在する．それらの規約が守られない場合に，プロジェクトに依らずバグ発生との相関を示すことができれば，それを利用することで，適用対象のプロジェクトに対する学習なく，それほど多くの処理時間を要さずに不具合予測を実施できる可能性がある．

このような考えに基づき，事前調査としてオープンソースソフトウェアリポジトリを対象に，コミット前後のコーディング規約違反変化とバグ発生の関連を分析した．Java で開発されたプロジェクトを対象に分析し，コーディング規約の検査には Java コーディング規約検査ツール Checkstyle [10] を利用した．本論文では，分析結果を報告する．

2. 関連研究

変更前後のソースコードと不具合発生の関係を学習することによって，不具合発生を予測する手法は，古くから提案されている．それらは，ソフトウェア開発プロジェクトで過去に実施された変更前後のソースコードと不具合発生の関係を教師データとして学習することで，変更実施に対する不具合発生の可能性を予測する [2][3][4]．また，変更前後のメトリクス変化と不具合発生の関係を利用した予測手法 [5][6] や，変更前後のソースコード構造の変化と不具合発生の関係を利用した予測手法 [7] も提案されている．

学習コスト削減のために教師データを利用せずに，変更時のコミットに含まれる変更差分情報を分類し，その分類結果に従って不具合発生を予測する手法も，近年提案されている．Yang らの手法 [8] は，ソフトウェア変更差分に関するメトリクス [11] を分類し，不具合が混入しているコミットを予測する．また近藤らの手法 [9] は，変更ソースコード片自体を分類し，不具合が混入しているコミットを予測する．後者は近藤らの論文でも考察されているように前者より予測精度が高いが，ソースコード片の分類にある程度の処理時間を要する．

3. コーディング規約違反増加量の不具合予測への利用

3.1 コーディング規約

プログラムの設計や実装に対する属人性を排除するために，プロジェクト開発ではコーディング規約が規定されることが多い．コーディング規約とは，ソフトウェア開発における，コーディングにあたっての統一的なガイドラインである．プロジェクト全体のメンバでの統一的な認識に基づくコーディング規約を定め，遵守することにより，プロジェクト内でのメンバ間でのプログラムの理解を促進し，保守性や移植性を高めることが可能になる．C 言語の場合，特にメモリをはじめとするハードウェアへの直接的な制御が可能であるため，可読性や保守性の低下がプログラムの安全性に直接的に影響する場合が多い．このため，組み込みソフトウェアである車載製品向けの安全性を議論するための業界団体 [*1] では，ソフトウェア設計のための標準規格 （MISRA-C）が制定されている．この標準規格は改訂されており，守る義務があったり必須であったりする規約や，守ることが推奨される規約を定義している．コーディング規約は非常に多いため，すべてに対して正しく理解し，正しく規約に遵守することが難しいため，組織や，開発グループで守るべきコーディング基準を定めて，それを規約として遵守しているケースも多い [12]．

3.2 Checkstyle によるコーディング規約の検査

開発成果物であるプログラムがコーディング規約に準拠して記述されているかどうかを検査するためのツールが開発されており，Java 言語を対象としたものとして Checkstyle [10] がある．これはプログラムの静的解析によってコーディングスタイルが規約に準拠しているかどうかを検査するためのツールである．155 種類のコーディング規約が用意されており [*2]，ユーザはそのうちから必要な規約のセットを選択し，準拠しているかを検査できる．

規約によっては，検査実施の際に具体的な値を必要とするものも存在する（例えば，一行あたりの文字数に関する規約での，具体的な文字数など）．Checkstyle の実装では，XML の設定ファイル内にそれぞれのコーディング規約の利用をタグとして設定できるようにしている．実際にコーディング規約を策定する際には，規約策定に必要な具体的な値を，タグに対する属性値として記述する．

3.3 不具合予測への利用に対する基本アイデア

3.1 節で述べたように，コーディング成果物の属人性が高いことは成果物の品質を下げる．そして実際に，C 言語による開発プロジェクトを対象にした分析によって，ソフトウェアのコーディングスタイルに基づく規約（3.1 節で述べた MISRA-C）に対する違反とバグ含有率に相関があるという調査結果も示されている [13]．また，3.2 節で述べたような Java 言語のプログラムに対するコーディング規則のうち，識別子名とスコープの長さなどのメトリクスと不具合発生には関連があることも報告されている [14]．我々も，コーディングスタイルに基づく規約への違反と不具合の発生には関連があるのではないかと考える．

3.2 節では 155 種類のコーディング規約があることを述べたが，すべてを遵守することは現実的ではない．数多くのコーディング規約からどのコーディング規約を遵守するかは次の要因により決定されると考える．

(1) 属人的要因に依存する場合
コーディングスタイルによっては特定の開発者のみに外れ値的に利用される場合がある [15]．一部のコーディング規約についての遵守は属人的に決定される．

*1 MISRA Consortium (https://www.misra.org.uk/)
*2 checkstyle ver.8.11 (2018/8/9 閲覧)

© 2018 Information Processing Society of Japan

(2) プロジェクトに依存する場合
3.1 節で述べたように，組織内での後工程での保守性を考え，組織やプロジェクトに依存し，遵守するコーディング規約が決定される．

(3) エキスパートの共通的な慣例に依存する場合
3.1 節で述べたように，エキスパート開発者であれば可読性を考えて，守るべきコーディング規約を遵守している場合がある．これについては，組織や開発プロジェクトに依存しない．それらがオープンソース／フリーソフトウェアコミュニティなどによってガイドラインとして規定されている場合もある [16][17][18].

前述のように，教師なし学習による不具合予測手法 [8] では，メトリクス変化の特徴が対象プロジェクトに依存するためそれほど予測精度が高くない．一方，ソースコード片自体を分類することで予測する手法 [9] では，ソースコード片の分類にある程度の処理時間を要する．仮に対象プロジェクトに依存せずに，異なったプロジェクト間で共通的に利用できるメトリクスを利用できれば，これらの問題を解決できる可能性がある．上記の (3) で挙げたコミュニティに規定されたガイドラインには，品質低下を防ぐためのコーディング規約が規定されている．そこて我々は，異なったプロジェクト間で共通的に利用できるメトリクスとして，それらの規約違反に基づくメトリクスに着目した．

3.2 節では，Checkstyle を利用することで 155 種類のコーディング規約に対する準拠を検査できることを述べた．Java に対する検査ツールである Checkstyle は，Google によるガイドライン [17] と Sun Microsystems によるガイドライン [18] への準拠を検査するための設定ファイルを公開している．それらに共通して含まれるコーディングスタイルに対する規約を中心に，ソフトウェア変更時の規約違反増加量とソフトウェアの品質に有意な相関があることを明らかにできれば，組織や開発プロジェクトに依存しないメトリクスとして，不具合予測に利用できる．

このような考えに基づき次章以降では，オープンソースソフトウェアプロジェクトを対象に，規約違反増加量とソフトウェア品質の関係を調査した結果について述べる．

4. 調査方法

4.1 概要

コーディング規約違反増加量とソフトウェア品質の関係を，オープンソフトウェア開発プロジェクトを対象に，次の 2 種類の調査を実施した．

調査 1: ソフトウェア品質に影響を与える可能性のあるコーディング規約違反の特定
まずプロジェクトの各コミットを対象に，コミット前後で Checkstyle で検出されるコーディング規約違反の増加量を，メトリクス値として取得した．次に，各コミットとそのコミット以降に現れるバグ履歴を分類し，各コミットによってそれ以降にバグが発生したかどうかを判別した．バグが発生した場合，該当のコミットを，バグ発生の原因になったコミットとして扱う．そして，バグ発生の原因になったコミットと原因にならなかったコミットで，コーディング規約違反の増加量に有意差がないか調査した．
なお，この調査は 4 個のオープンソースソフトウェアプロジェクトを対象に実施した．また，コーディング規約違反については，3.2 節に挙げた 155 種類のすべての規約を対象にした．

調査 2: 調査 1 で特定したコーディング規約違反に対するメトリクス値を利用した不具合予測性能の調査
調査 1 で，バグ発生の原因になったコミットとならなかったコミットで増加量に有意差のあったコーディング規約違反を利用して，調査 1 で利用したものとは別のプロジェクトを対象に，不具合予測を実施した．
まず調査 1 で全プロジェクトで有意差のあったコーディング規約違反から，コミュニティのガイドライン [17] [18] を利用し，3 つのコーディング規約を選択した．この選択の過程については調査の結果として後述する．
そして選択したコーディング規約を用い，調査 1 とは別の 4 個のオープンソースソフトウェアプロジェクトを対象に，各コミットに対するコーディング規約違反の増加量を取得した．この増加量を教師なし学習により分類し，バグ発生原因になるかどうかを予測させた．

調査 1 では，特定の 4 プロジェクトに対する性質を明らかにすることを目的に，調査を実施した．そして品質に影響を与える可能性が高いとして導出されたコーディング規約違反に対して，調査 2 で他の 4 プロジェクトに対してクロスプロジェクトで利用可能性を調査した．なお，これらの合計 8 プロジェクトは，Commit Guru[19] で公開されている Java プロジェクトを利用した．Commit Guru はオープンソースソフトウェアのリポジトリを解析したデータセットを提供する Web サービスであり，各コミットに対するソフトウェアメトリクスとコミットメッセージの情報を取得できる．

本章では，まずコーディング規約違反の増加量を表すメトリクスを定義し，そのあとで調査手順の詳細を述べる．

4.2 コーディング規約違反メトリクス

本節では，調査手順においてコーディング規約違反の増加量を表すために本研究で定義したコーディング規約違反メトリクスについて述べる．

コーディング規約違反メトリクスは，あるコミットの前後で増加したコーディング規約違反の増加量として定義する．コミットに関連するファイルが複数存在する場合は，それぞれのファイルに対する増加量の総和として定義する．

© 2018 Information Processing Society of Japan

いま，あるコミットに関連するファイル群 F を次の (1) 式として定義する．

$$F = \{F_1, F_2, F_3, \ldots, F_n\} \quad (1)$$

コーディング規約 R_x に対するそのコミットでのコーディング規約違反増加量を，コーディング規約違反メトリクス $Violation_{R_x}$ として，(2) 式のように定義する．

$$Violation_{R_x} = \sum_{i=1}^{n} \phi(V_{R_x}(F_{i_{aft}}) - V_{R_x}(F_{i_{bef}})) \quad (2)$$

$$\phi(x) = \begin{cases} 0 & (x < 0 \text{の場合}) \\ x & (\text{それ以外}) \end{cases}$$

ここで，$V_{R_x}(F_i)$ は，ファイル F_i のコーディング規約 R_x に対する規約違反量である．そして，$F_{i_{bef}}$ はファイル F_i のコミットによる変更前の状態のファイルを，$F_{i_{aft}}$ はコミットによる変更後の状態のファイルを示す．コーディング規約違反メトリクスはコミットによる規約違反の増加量を表す．このため，あるコミットで規約違反の増加量 $V_{R_x}(F_{i_{aft}}) - V_{R_x}(F_{i_{bef}})$ が負値であった場合，(2) 式ではその値を 0 として計算している．

なおコーディング規約違反の増加量は，そのコミットで編集操作が行われた編集量に影響を受ける可能性がある．そこで，各コミットでの編集量（追加行数 LA，削除行数 LD）を利用し正規化した正規化コーディング規約違反メトリクス $NormV_{R_x}$ を，(3) 式のように定義する．

$$NormV_{R_x} = \frac{\sum_{i=1}^{n} \phi(V_{R_x}(F_{i_{aft}}) - V_{R_x}(F_{i_{bef}}))}{\sum_{i=1}^{n}(LA_{F_i} + LD_{F_i})} \quad (3)$$

$$\phi(x) = \begin{cases} 0 & (x < 0 \text{の場合}) \\ x & (\text{それ以外}) \end{cases}$$

ここで，LA_{F_i}，LD_{F_i} はそれぞれ対象コミットでのファイル F_i に対する追加行数，削除行数を表す．

4.3 調査 1: ソフトウェア品質に影響を与える可能性のあるコーディング規約違反の特定

不具合とコーディングスタイルの規約違反の共起傾向を明らかにするため，調査対象の各プロジェクトに対し次の手順で調査した．

(1) 対象プロジェクトの Commit Guru データセットから，コミット ID の一覧を取得する．
(2) 各コミット ID に対応したコミットログに現れる各ソースファイルに対して，コミットによる変更前後のソースファイルを取得する．
(3) (2) で取得したソースファイルに対して，Checkstyle により検査を実施する．この際に，3.2 節で挙げた 155 種類すべてのコーディング規約に対して検査を実施した．コーディング規約が属性値を必要とする場合は，Checkstyle で推奨されるデフォルト値を利用した．
(4) (3) で Checkstyle により検査した実行結果から，それぞれのコーディング規約に対してコーディング規約違反メトリクスを算出する．
(5) (2) から (4) を，(1) で取得したすべてのコミットに対して実施する．
(6) Commit Guru データセットから各コミット ID を取得する．Commit Guru では，コミットメッセージに含まれる単語によって不具合修正コミットを検出し，そのコミットが示す修正箇所のコードを最初に追加したコミットに対して，不具合発生と修正の関連をラベル付けして記録している．このラベルを利用し，不具合を発生させたコミットのコミット ID を特定する．
(7) (2) から (5) で算出した，コミット ID ごとの各規約に対するコーディング規約違反メトリクスに，(6) で取得した情報により不具合との関連をラベル付けする．
(8) 全コミット ID に対して，コーディング規約ごとに不具合有無によってコーディング規約違反メトリクスに有意差があるかどうかを，マン・ホイットニーの U 検定により検定した．その際に，帰無仮説 H_0 を「2 群のメトリクス値に差がない」とし，有意水準を 5% に設定した．そして，この帰無仮説が棄却されたコーディング規約を，不具合有無によってコーディング規約違反メトリクスに有意差を生じる規約として記録した．

表 1　調査 1 の対象プロジェクト

プロジェクト名	開発者数（人）	開発期間（日）	コード行数（行）	総コミット数	不具合混入率（%）
ndk	220	3,397	1,164,140	10,560	16.6
hadoop	214	3,183	5,361,034	17,746	14.7
presto	266	1,947	1,260,484	11,958	20.1
netty	392	3,447	508,650	8,595	15.6

この手順に従い，Commit Guru で公開されている 4 つの Java プロジェクトを調査した．プロジェクトの詳細を**表 1** に示す．なお 表 1 のコード行数は開発期間の最後のコミット時点での行数である．また不具合混入率は，総コミット数に対する不具合を含むコミットの割合である．

4.4 調査 2: 調査 1 で特定したコーディング規約違反に対するメトリクス値を利用した不具合予測性能の調査

調査 1 で特定した不具合有無によりコーディング規約違反メトリクスに有意差を生じる規約群に対して，その利用可能性を他のプロジェクトに対して適用し調査した．

(1) 調査 1 で全プロジェクトで有意差のあったコーディング規約違反から，コミュニティのガイドライン [17] [18] を利用し，コーディング規約を選択する．
(2) 調査 1 で利用したプロジェクトとは異なるプロジェクトに対して，(3) から (7) を実施する．
(3) 対象プロジェクトの Commit Guru データセットから，コミット ID の一覧を取得する．

© 2018 Information Processing Society of Japan

表 2 調査 2 の対象プロジェクト

プロジェクト名	開発者数（人）	開発期間（日）	コード行数（行）	総コミット数	不具合混入率（%）
camel	532	3,981	2,617,924	31,289	20.5
fabric8	140	2,457	128,722	13,470	20.3
log4j	21	5,314	75,652	3,276	16.7
tomcat	36	4,339	808,074	19,103	27.5

(4) 各コミット ID に対応したコミットログに現れる各ソースファイルに対して，コミットによる変更前後のソースファイルを取得する．
(5) (4) で取得したソースファイルに対して，Checkstyle により検査を実施する．この際に，(1) で選択したコーディング規約に対して検査を実施した．コーディング規約が属性値を必要とする場合は，Checkstyle で推奨されるデフォルト値を利用した．
(6) (5) で Checkstyle により検査した実行結果から，それぞれのコーディング規約に対してコーディング規約違反メトリクスを算出する．
(7) (4) から (6) を，(3) で取得したコミットに対して繰り返し実施する．繰り返しの都度，各コーディング規約違反メトリクスについて分類器により 2 つのクラスタに分類する．分類した結果コーディング規約違反メトリクスが大きい方を，不具合を含むクラスタとみなす．

(7) のクラスタ分類には，分類器として Deep Embedded Clustering[20] を利用した．これは，ディープニューラルネットワークを利用した，教師なし分類器である．

この手順に従って，Commit Guru で公開されているプロジェクトのうち，調査 1 で利用したプロジェクトと異なる 4 つの Java プロジェクトに対して調査を行った．実際に用いたプロジェクトの詳細を **表 2** に示す．

5. 結果

5.1 調査 1 について

調査対象にした 4 つのプロジェクトの各コミットに対して，コーディング規約違反メトリクスを算出し，有意差があるかどうかを検定した．4.2 節で述べたように，コーディング規約違反メトリクス $Violation_{R_x}$ を利用した場合，各コミットでの編集量の影響を受け，有意差が生じる規約が存在しなかった．そこで正規化コーディング規約違反メトリクス $NormV_{R_x}$ を利用して有意差の検定を実施した．

4 つのプロジェクトすべてにおいて正規化コーディング規約違反メトリクス $NormV_{R_x}$ に有意差が存在したコーディング規約は，11 種類であった．各プロジェクトに対して，11 種類のコーディング規約それぞれについて調査したコーディング規約違反メトリクスを， **表 3** に示す．なお，この表では，コーディング規約違反メトリクス $Violation_{R_x}$ と正規化コーディング規約違反メトリクス $NormV_{R_x}$ について，不具合に関連しないコミット（不具合関連無），不具合に関連するコミット（不具合関連有）での平均値を算出している．コーディング規約違反メトリクス $Violation_{R_x}$ については， 表 3 にあるように，どのプロジェクトにおいても不具合関連の有無で明確な差がなかった．一方，正規化コーディング規約違反メトリクス $NormV_{R_x}$ については，有意差が生じた．こちらについては参考に， 表 3 に p 値を載せた．なお表に掲載したコーディング規約については，p 値がすべてのプロジェクトにおいて有意水準 5% 未満に収まっている．

表 4 コーディング規約違反の有意差による順位付け

コーディング規約名	ndk	hadoop	presto	netty	Google	Sun
FinalParameters	9	1	1	2	×	○
EmptyLineSeparator	4	4	2	8	○	×
JavadocType	3	6	6	3	×	○
LineLength	10	2	9	4	○	○
MissingCtor	1	10	3	7	×	×
JavadocMethod	7	3	8	5	○	○
OuterTypeFilename	2	8	4	9	○	×
WriteTag	5	5	7	6	×	×
AnnotationOnSameLine	8	9	10	1	×	×
ImportControl	6	7	5	10	×	×
MissingSwitchDefault	11	11	11	11	○	○

5.2 調査 2 について

5.1 節で述べた結果に基づき，各プロジェクトごとにコーディング規約を有意差の大きい順 （p 値の小さい順）に順位付けした結果を， **表 4** に示す．この表の「Google」列「Sun」列は，それぞれコミュニティのガイドライン [17] [18] に含まれている規約かどうかを表す．

まず，調査 2 の不具合予測に利用するコーディング規約を選択した．表 4 に示すように，FinalParameters の規約 （代入されないメソッド引数に，final 修飾子を付加する）に対する規約違反が，4 プロジェクト中 3 プロジェクトで，不具合との関連について他の規約より比較的高い相関を示した．そこで，FinalParameters に対する規約違反は，不具合との関連に特に高い相関を示す可能性があると考え選択した．また，ガイドライン [17] [18] に共通に含まれる規約は，LineLength（一行に記述する文字数に対して制限する），JavadocMethod（protected/public のメソッドに Javadoc コメントを記述する），MissingSwitchDefault（switch 文に必ず default 句を記述する）であった．これらの規約に対する違反箇所のうち，JavadocMethod に対する違反を含むコミット差分を確認すると，テストケースのメソッドを追加する際に Javadoc コメントを記述しないことが多く，そのことと不具合の発生が関連づいたことによるものであった．そこでこの規約を除いた LineLength と MissingSwitchDefault の 2 つの規約を選択した．

この結果として，調査 2 では FinalParameters，LineLength，MissingSwitchDefault の 3 つのコーディング規約を利用して，不具合予測を実施した．4.4 節の手順 (7) に示したように，クラスタの分類をインクリメンタルに実施した．まず，それぞれのコーディング規約ごと，および 3

© 2018 Information Processing Society of Japan

表 3 調査 1 の結果

コーディング規約名	ndk					hadoop				
	$Violation_{R_x}$		$NormV_{R_x}$			$Violation_{R_x}$		$NormV_{R_x}$		
	不具合関連無	不具合関連有	不具合関連無	不具合関連有	p 値	不具合関連無	不具合関連有	不具合関連無	不具合関連有	p 値
FinalParameters	$5.48 \times 10^{\ 2}$	$7.63 \times 10^{\ 2}$	$6.88 \times 10^{\ 7}$	$9.85 \times 10^{\ 5}$	$2.79 \times 10^{\ 2}$	1.97×10^{1}	3.76×10^{1}	$9.48 \times 10^{\ 3}$	$2.12 \times 10^{\ 2}$	$7.19 \times 10^{\ 207}$
EmptyLineSeparator	$1.52 \times 10^{\ 2}$	$2.75 \times 10^{\ 2}$	$1.99 \times 10^{\ 7}$	$4.73 \times 10^{\ 5}$	$3.08 \times 10^{\ 3}$	7.72×10^{0}	1.38×10^{1}	$4.66 \times 10^{\ 3}$	$8.54 \times 10^{\ 3}$	$4.42 \times 10^{\ 144}$
JavadocType	$1.03 \times 10^{\ 2}$	$2.12 \times 10^{\ 2}$	$1.36 \times 10^{\ 7}$	$3.39 \times 10^{\ 5}$	$3.00 \times 10^{\ 4}$	1.52×10^{0}	2.77×10^{0}	$5.81 \times 10^{\ 4}$	$1.50 \times 10^{\ 3}$	$2.46 \times 10^{\ 130}$
LineLength	$1.42 \times 10^{\ 2}$	$2.22 \times 10^{\ 2}$	$1.75 \times 10^{\ 7}$	$3.82 \times 10^{\ 5}$	$3.57 \times 10^{\ 2}$	5.30×10^{0}	9.32×10^{0}	$4.87 \times 10^{\ 3}$	$9.24 \times 10^{\ 3}$	$1.94 \times 10^{\ 185}$
MissingCtor	$8.42 \times 10^{\ 3}$	$1.69 \times 10^{\ 2}$	$1.12 \times 10^{\ 7}$	$2.95 \times 10^{\ 5}$	$2.94 \times 10^{\ 4}$	1.07×10^{0}	2.07×10^{0}	$5.44 \times 10^{\ 4}$	$1.25 \times 10^{\ 3}$	$9.14 \times 10^{\ 98}$
JavadocMethod	$2.58 \times 10^{\ 2}$	$4.45 \times 10^{\ 2}$	$3.48 \times 10^{\ 7}$	$5.45 \times 10^{\ 5}$	$2.07 \times 10^{\ 2}$	1.65×10^{1}	3.12×10^{1}	$1.03 \times 10^{\ 2}$	$1.99 \times 10^{\ 2}$	$5.83 \times 10^{\ 172}$
OuterTypeFilename	$8.11 \times 10^{\ 3}$	$1.59 \times 10^{\ 2}$	$1.11 \times 10^{\ 7}$	$2.67 \times 10^{\ 5}$	$2.96 \times 10^{\ 4}$	1.65×10^{0}	3.26×10^{0}	$6.02 \times 10^{\ 4}$	$1.43 \times 10^{\ 3}$	$5.29 \times 10^{\ 117}$
WriteTag	$1.01 \times 10^{\ 2}$	$1.91 \times 10^{\ 2}$	$1.36 \times 10^{\ 7}$	$2.66 \times 10^{\ 5}$	$1.71 \times 10^{\ 2}$	1.39×10^{0}	2.52×10^{0}	$5.63 \times 10^{\ 4}$	$1.47 \times 10^{\ 3}$	$5.18 \times 10^{\ 133}$
AnnotationOnSameLine	$4.99 \times 10^{\ 3}$	$1.48 \times 10^{\ 2}$	$7.34 \times 10^{\ 8}$	$1.93 \times 10^{\ 5}$	$2.73 \times 10^{\ 2}$	1.08×10^{1}	1.98×10^{1}	$1.00 \times 10^{\ 2}$	$1.57 \times 10^{\ 2}$	$1.73 \times 10^{\ 106}$
ImportControl	$7.90 \times 10^{\ 3}$	$1.38 \times 10^{\ 2}$	$1.11 \times 10^{\ 7}$	$1.93 \times 10^{\ 5}$	$1.82 \times 10^{\ 2}$	1.67×10^{0}	3.31×10^{0}	$6.09 \times 10^{\ 4}$	$1.45 \times 10^{\ 3}$	$2.78 \times 10^{\ 117}$
MissingSwitchDefault	$1.44 \times 10^{\ 2}$	$6.41 \times 10^{\ 2}$	$9.10 \times 10^{\ 7}$	$2.63 \times 10^{\ 6}$	$4.39 \times 10^{\ 2}$	$3.41 \times 10^{\ 2}$	$6.67 \times 10^{\ 2}$	$5.57 \times 10^{\ 6}$	$3.40 \times 10^{\ 5}$	$2.48 \times 10^{\ 12}$

コーディング規約名	presto					netty				
	$Violation_{R_x}$		$NormV_{R_x}$			$Violation_{R_x}$		$NormV_{R_x}$		
	不具合関連無	不具合関連有	不具合関連無	不具合関連有	p 値	不具合関連無	不具合関連有	不具合関連無	不具合関連有	p 値
FinalParameters	5.21×10^{0}	2.70×10^{1}	$2.17 \times 10^{\ 2}$	$3.96 \times 10^{\ 2}$	$1.27 \times 10^{\ 85}$	6.64×10^{0}	3.02×10^{1}	$2.96 \times 10^{\ 2}$	$3.17 \times 10^{\ 2}$	$9.40 \times 10^{\ 69}$
EmptyLineSeparator	1.39×10^{0}	7.85×10^{0}	$6.26 \times 10^{\ 3}$	$1.11 \times 10^{\ 2}$	$1.43 \times 10^{\ 81}$	2.15×10^{0}	8.50×10^{0}	$6.94 \times 10^{\ 3}$	$7.74 \times 10^{\ 3}$	$1.72 \times 10^{\ 51}$
JavadocType	$6.40 \times 10^{\ 1}$	2.84×10^{0}	$1.94 \times 10^{\ 3}$	$3.67 \times 10^{\ 3}$	$7.70 \times 10^{\ 73}$	$4.66 \times 10^{\ 1}$	1.89×10^{0}	$1.05 \times 10^{\ 3}$	$2.23 \times 10^{\ 3}$	$1.37 \times 10^{\ 68}$
LineLength	5.99×10^{0}	3.51×10^{1}	$4.69 \times 10^{\ 2}$	$6.86 \times 10^{\ 2}$	$2.23 \times 10^{\ 56}$	6.92×10^{0}	3.92×10^{1}	$6.29 \times 10^{\ 2}$	$6.64 \times 10^{\ 2}$	$1.66 \times 10^{\ 68}$
MissingCtor	$1.91 \times 10^{\ 1}$	$8.37 \times 10^{\ 1}$	$7.51 \times 10^{\ 4}$	$1.48 \times 10^{\ 3}$	$6.91 \times 10^{\ 78}$	$3.24 \times 10^{\ 1}$	1.10×10^{0}	$8.11 \times 10^{\ 4}$	$1.43 \times 10^{\ 3}$	$3.16 \times 10^{\ 52}$
JavadocMethod	3.13×10^{0}	1.57×10^{1}	$1.60 \times 10^{\ 2}$	$2.51 \times 10^{\ 2}$	$5.06 \times 10^{\ 59}$	4.78×10^{0}	1.85×10^{1}	$2.45 \times 10^{\ 2}$	$2.38 \times 10^{\ 2}$	$9.45 \times 10^{\ 67}$
OuterTypeFilename	$4.86 \times 10^{\ 1}$	2.20×10^{0}	$1.44 \times 10^{\ 3}$	$2.68 \times 10^{\ 3}$	$3.58 \times 10^{\ 73}$	$7.49 \times 10^{\ 1}$	2.54×10^{0}	$1.16 \times 10^{\ 3}$	$2.00 \times 10^{\ 3}$	$6.76 \times 10^{\ 51}$
WriteTag	$6.38 \times 10^{\ 1}$	2.83×10^{0}	$1.93 \times 10^{\ 3}$	$3.67 \times 10^{\ 3}$	$1.44 \times 10^{\ 70}$	$4.35 \times 10^{\ 1}$	1.72×10^{0}	$1.04 \times 10^{\ 3}$	$2.06 \times 10^{\ 3}$	$9.85 \times 10^{\ 65}$
AnnotationOnSameLine	3.20×10^{0}	1.42×10^{1}	$1.68 \times 10^{\ 2}$	$2.18 \times 10^{\ 2}$	$2.80 \times 10^{\ 55}$	4.41×10^{0}	1.94×10^{1}	$2.21 \times 10^{\ 2}$	$2.10 \times 10^{\ 2}$	$4.19 \times 10^{\ 71}$
ImportControl	$4.86 \times 10^{\ 1}$	2.20×10^{0}	$1.39 \times 10^{\ 3}$	$2.68 \times 10^{\ 3}$	$7.44 \times 10^{\ 73}$	$7.57 \times 10^{\ 1}$	2.56×10^{0}	$1.17 \times 10^{\ 3}$	$2.01 \times 10^{\ 3}$	$1.30 \times 10^{\ 50}$
MissingSwitchDefault	$1.41 \times 10^{\ 2}$	$7.27 \times 10^{\ 2}$	$9.31 \times 10^{\ 5}$	$1.29 \times 10^{\ 4}$	$6.46 \times 10^{\ 13}$	$4.73 \times 10^{\ 2}$	$1.55 \times 10^{\ 1}$	$1.04 \times 10^{\ 4}$	$1.83 \times 10^{\ 4}$	$3.36 \times 10^{\ 8}$

表 5 コーディング規約ごとの不具合予測正解率

コーディング規約名	camel	fabric8	log4j	tomcat
FinalParameters	72.5%	70.1%	55.4%	60.7%
LineLength	66.1%	68.3%	50.4%	53.3%
MissingSwitchDefault	70.8%	69.1%	52.2%	56.2%
すべて利用した場合	77.8%	76.3%	60.5%	68.7%
(適合率)	60.1%	53.8%	64.1%	40.3%
(再現率)	51.2%	49.0%	30.8%	28.4%
(F 値)	0.553	0.513	0.605	0.333

つすべてのコーディング規約を利用した場合に予測結果がどの程度の正解率を示すかを，**表 5** に示す．なおこの結果は，手順 (7) によってすべてのコミットを分類し終えた時点での分類の正解率である．また，すべてのコーディング規約を利用した場合については，適合率，再現率，F 値の各評価指標についても併せて記載している．なおクラスタ分類に利用した分類器（Deep Embedded Clustering）は非決定的に予測結果を導出するため，表内のすべての値は 10 回予測を行った際の平均値を利用している．

また，4.4 節の手順 (7) に示したように，クラスタの分類をコミットの順番にインクリメンタルに実施した．その際の解析に利用した累積コミット数と，正解率の関係を，**図 1** に示す．なお，表 5 と同様に，すべての値は 10 回予測を行った際の平均値を利用している．log4j については 3000 コミット程度しか存在しなかったため，3000 コミットで終了している．また他のプロジェクトについても，累積コミット数を増やしても大きな変化がなかったため，6000 コミットまでの結果を載せている．

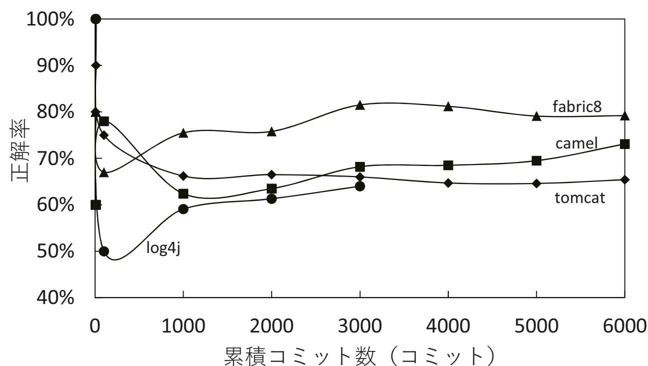

図 1 累積コミット数と正解率の関係

6. 考察

6.1 コーディング規約違反と不具合発生との関連

調査 1 の結果から，対象にした 4 プロジェクトすべてにおいて，11 種類のコーディング規約に対する違反が不具合発生に関連していることが明らかになった．

これらのコーディング規約違反のうち，調査 2 で対象として選択した FinalParameters と LineLength の各規約について，正規化コーディング規約違反メトリクスが大きかったコミットを，presto プロジェクトを対象に観察した．

まず，FinalParameters の規約違反が発生する状況について観察した．不具合発生に関連するコミットのうち，FinalParameters に対する正規化コーディング規約違反メトリクスの大きい 10 コミットの開発履歴を参照したところ，次のような特徴がみられた．

- コミットをした開発者は，FinalParameters 規約違反

© 2018 Information Processing Society of Japan

表 6 LineLength の違反コードの主な分類

#	コード差分に含まれる特徴の箇所数（カッコ内はそのうち不具合に関連する箇所）				
	変数名が長い	文字列が長い	メソッドチェーン	メソッド引数が多い	メソッドのネスト呼出し
1	1 (0)	1 (1)	0	0	0
2	0	0	7 (0)	0	0
3	1 (0)	0	0	3 (2)	1 (0)
4	0	0	0	0	8 (5)
5	0	0	3 (0)	2 (2)	0
6	4 (0)	4 (0)	0	3 (1)	6 (0)

を繰り返していた（10 件）
- コミットにより新しいメソッドが追加され，その中に FinalParameters 規約違反を含んでいた（8 件）
- コミットをした開発者は，変更数で比較した場合に，開発者全体の上位 10 % に含まれている（7 件）

さらに，LineLength の規約違反が発生する状況について観察した．不具合発生に関連するコミットのうち，LineLength に対する正規化コーディング規約違反メトリクスの大きい 6 コミットの開発履歴を参照したところ，LineLength の違反につながる主な特徴を分類できた（**表 6**）．なお，不具合に対応する修正コミットにおいて該当箇所が変更されていた場合，その数を内数で示している．

観察を実施したプロジェクトが 1 つのプロジェクトだけであり，この特徴が他のプロジェクトに当てはまるかどうかは確認する必要があるが，観察した 2 つのコーディング規約に対して規約違反を含むコードが次の特徴を持つ場合に，不具合に関連する傾向があった．
- FinalParameters 規約違反
  日常的に FinalParameters 規約違反を繰り返している比較的アクティブな開発者によって，新しいメソッドが追加された時に不具合に関連する．
- LineLength 規約違反
  行に含むメソッド呼び出しが長くなった結果として LineLength 規約違反が生じた場合，不具合に関連する．

6.2 不具合発生予測性能について

調査 2 では 3 種類のコーディング規約を選択し，不具合予測を実施した．表 5 に示したように，3 種類のコーディング規約を利用することによって 60% ∼ 80% の正解率で，教師なし学習による不具合予測を実施可能である．この調査は，調査 1 で相関を導出したプロジェクトとは別のプロジェクトに対して試行している．したがって，コーディング規約違反と不具合発生との関連が対象プロジェクトに依存する度合いが低く，教師なし学習での不具合予測の際に分類の基準として利用できることが分かった．また 表 5 によると，利用するコーディング規約の種類を増やすことにより不具合予測の正解率が高くなっている．このため，今回利用したコーディング規約以外にも不具合との関連に有意差があるコーディング規約を追加することにより，より精度の高い不具合予測が可能である．

本研究と同様にメトリクス分類により教師なし学習で不具合予測を実施する既存研究 [8] で実装したツールでは，不具合予測の適合率は 10% ∼ 50% 前後，再現率は 70% 前後と報告されている．対象プロジェクトが同一でないため純粋に比較できないが，表 5 を参照すると，本研究で提案した不具合予測の手法は適合率で上回り，逆に再現率で下回っている．我々は今回提案した方法を利用して，コミット時に不具合予測を開発者に指摘するツールを，今後構築する予定である．開発時に指摘内容を開発者がチェックすることを考えると，適合率が高い（不具合と指摘された箇所が，実際の不具合箇所である可能性が高い）方が，再現率が高い（実際の不具合箇所のうち，不具合と指摘された箇所の割合が高い）ことより優先される．このため，コミット時に不具合を指摘するという目的に対しては，我々の手法のほうが優れている．

さらに 図 1 に示したように，解析した累積コミット数の少ない段階でも，多数のコミットを利用したときと同程度の十分に高い正解率で不具合予測を実施できている．教師なし学習での予測の際に，データが少ない状態でも十分な予測性能を確保できると言える（ただし log4j の結果にあるように，コミット数が少ない段階ではノイズデータの影響を大きく受ける）．なお， 図 1 では，あるコミットを実施した場合にそれ以前のコミットでの予測正否と累積して正解率を算出した結果を示している．実際に開発者が利用する際には，あるコミット実施に対する予測正否のみが重要であり，過去の予測正否は関係がない．この評価については，今後方法を含めて再検討する必要がある．

6.3 妥当性への脅威

6.3.1 外的妥当性

4.2 節で述べた (2) 式および (3) 式では，あるコミットでの規約違反の増加量 $V_{R_x}(F_{i_{aft}}) - V_{R_x}(F_{i_{bef}})$ が負値であった場合に，その値を 0 としている．これはコーディング規約違反増加が不具合に関連するという前提のもと，あるコミットにコーディング規約違反の増加したファイルと減少した別のファイルが関連している場合に，前者による増加量が不具合を引き起こす可能性を示唆していることを，後者が打ち消してしまう可能性が存在するためである．同様のことは，ファイル内の変更にも当てはまる．すなわち，あるコミットに関連したファイル内の一部でコーディング規約違反が増加し，別の一部で減少していた場合に，規約違反増加量に減少分の影響が現れる．したがって，本来観測されるべき規約違反増加量より少ない値が観測される可能性があり，その場合は不具合予測に影響を与える．

また，4.3 節で述べた調査 1 の手順 (3) で Checkstyle により検査を実施する際に，コーディング規約が属性値を必要とする場合は，Checkstyle で推奨されるデフォルト値を

© 2018 Information Processing Society of Japan

利用した．例えば，3.2 節で述べた行の長さに対しての具体的な文字数であれば，Checkstyle が公開する設定ファイルでは，Sun Microsystems によるガイドラインにはデフォルト値の 80 文字，Google によるガイドラインには 100 文字が与えられている．どちらのコーディングスタイルに準拠するのが妥当かは本研究のスコープ外であるため議論しないが，デフォルト値を利用して検査を実施したことが，規約違反の増加量に影響を与える可能性がある．

さらに調査 1 でのプロジェクトの選択には，コード行数や利用ドメインに特にバイアスをかけていないが，プロジェクト数が少なく，ソフトウェアプロジェクトの一般的な結果を示せていない可能性がある．

6.3.2 内的妥当性

本研究の調査には Commit Guru に記録されたバグ発生と修正の関連をラベル付け結果を利用した．このラベル付けはコミットメッセージに対して特定のキーワードが含まれるかどうかにより実施されている．したがって不具合を修正したコミットのコミットメッセージがそのキーワードを含まない場合は，不具合を見落としている可能性がある．

7. 結論

本研究では，ソフトウェア開発時のコーディング規約違反と不具合の共起傾向を調査した．その結果，ソフトウェア変更に対する特定のコーディング規約に関する違反の増加に対して，不具合の発生と関連が高いことが分かった．

今後の課題としては，より多くのプロジェクトを利用して関連性を確認することと，予測性能の評価方法の再検討が挙げられる．また，本研究で得られた傾向を利用し，不具合が含まれる可能性の高いコードがコミットされた場合に指摘する手法を，提案する予定である．

謝辞

本研究の成果の一部は，科研費基盤研究 (C)17K00110，2018 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] 畑 秀明，水野 修，菊野 亨：不具合予測に関するメトリクスについての研究論文の系統的レビュー，コンピュータソフトウェア， Vol. 29, No. 1, pp. 106–117 (2012).

[2] Aversano, L., Cerulo, L. and Grosso, C. D.: Learning from bug-introducing changes to prevent fault prone code, *Proc. of the Ninth int'l workshop on Principles of software evolution (IWPSE'07)*, pp. 19–26 (2007).

[3] Jiang, T., Tan, L. and Kim, S.: Personalized defect prediction, *Proc. of the 28th IEEE/ACM Int'l Conference on Automated Software Engineering (ASE'13)*, pp. 279–289 (2013).

[4] Kim, S., Whitehead, E. and Zhang, Y.: Classifying Software Changes: Clean or Buggy?, *IEEE Trans. on Software Engineering*, Vol. 34, No. 2, pp. 181–196 (2008).

[5] Zimmermann, T., Premraj, R. and Zeller, A.: Predicting Defects for Eclipse, *Proc. of the Third Int'l Workshop on Predictor Models in Software Engineering (PROMISE'07)*, pp. 9– (2007).

[6] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K., Adams, B. and Hassan, A. E.: Revisiting common bug prediction findings using effort-aware models, *Proc. of the 2010 IEEE Int'l Conference on Software Maintenance (ICSM'10)*, pp. 1–10 (2010).

[7] Wang, S., Liu, T. and Tan, L.: Automatically learning semantic features for defect prediction, *Proc. of the 38th Int'l Conference on Software Engineering (ICSE'16)*, pp. 297–308 (2016).

[8] Yang, X., Lo, D., Xia, X., Zhang, Y. and Sun, J.: Deep Learning for Just-in-Time Defect Prediction, *Proc. of the 2015 IEEE Int'l Conference on Software Quality, Reliability and Security (QRS '15)*, pp. 17–26 (2015).

[9] 近藤将成，森 啓太，水野 修，崔 銀惠：深層学習によるソースコードコミットからの不具合混入予測，情報処理学会論文誌，Vol. 59, No. 4, pp. 1250–1261 (2018).

[10] checkstyle: checkstyle - Checkstyle 8.11, checkstyle (online), available from ⟨http://checkstyle.sourceforge.net/⟩ (accessed 2018-08-09).

[11] Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A. and Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance, *IEEE Trans. on Software Engineering*, Vol. 39, No. 6, pp. 757–773 (2013).

[12] 独立行政法人情報処理推進機構：【改訂版】組込みソフトウェア開発向けコーディング作法ガイド［C 言語版］Ver.2.0，独立行政法人情報処理推進機構（IPA）技術本部ソフトウェア高信頼化センター（SEC） (2014).

[13] Boogerd, C. and Moonen, L.: Evaluating the relation between coding standard violations and faultswithin and across software versions, *Proc. of the 2009 6th IEEE Int'l Working Conference on Mining Software Repositories (MSR'09)*, pp. 41–50 (2009).

[14] 阿萬裕久，天嵜聡介，佐々木隆志，川原 稔：変数名とスコープの長さ及びコメントに着目したフォールト潜在性に関する定量的調査，ソフトウェアエンジニアリングシンポジウム 2015 論文集，pp. 69–76 (2015).

[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，日本ソフトウェア科学会 FOSE 2008 ソフトウェア工学の基礎 XV，pp. 129–134 (2008).

[16] Free Software Foundation, Inc.: GNU Coding Standards, Free Software Foundation, Inc. (online), available from ⟨https://www.gnu.org/prep/standards/standards.html⟩ (accessed 2018-08-09).

[17] Google, Inc.: Google Java Style Guide, Google, Inc. (online), available from ⟨http://google.github.io/styleguide/javaguide.html⟩ (accessed 2018-08-09).

[18] Sun Microsystems, Inc.: Code Conventions for the Java Programming Language (archive), Oracle Corp. (online), available from ⟨http://www.oracle.com/technetwork/java/codeconvtoc-136057.html⟩ (accessed 2018-08-09).

[19] Rosen, C., Grawi, B. and Shihab, E.: Commit Guru: Analytics and Risk Prediction of Software Commits, *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, pp. 966–969 (2015).

[20] Xie, J., Girshick, R. B. and Farhadi, A.: Unsupervised Deep Embedding for Clustering Analysis, *Proc. of the 33rd Int'l Conference on Machine Learning (ICML'16)*, Vol. 48, pp. 478–487 (2016).

© 2018 Information Processing Society of Japan