Vol.2014-SE-186 No.3
2014/11/13

オープンソース開発でのコード修正における コーディング規約違反の変化に関する調査

織田　泰輔[1,a)]　阿萬　裕久[2]　佐々木 隆志[2]　川原 稔[2]

概要：本稿は静的解析ツールで検出可能なコーディング規約違反に着目し，実際のオープンソース開発においてどういった規約違反が多く検出され，その中でどういった規約違反は結果的に解消されていくのか，あるいは逆に解消されずにそのまま残っていくのかを調査している．調査としては，Java 言語で開発されている 2 種類のオープンソースソフトウェアに対して，それらのリポジトリへのコミットごとに静的解析ツール Checkstyle を適用し，Sun Java コーディング規約に対する違反情報を収集している．分析の結果，二つのソフトウェアにおいて六つの違反項目が共通して多く検出されることが分かり，それらの変化（改善）の傾向から開発の過程でプログラマがどういった点に注意を払っていたかを考察している．

1. はじめに

ソフトウェア開発において，ソフトウェアシステムの実装，即ちコーディングは最も基本的な作業である．上流工程における要求定義及び設計が高品質なシステムを生み出す上で極めて重要であることは言うまでもないが，そこでは，それらの下流に位置する実装が正確かつ効率的に行われることが前提となっている．仕様や設計からソースコードを自動生成できる，あるいは既存のソフトウェア部品の再利用と再設定で対応できるという場合は問題ないが，実際にはそうでない場合，つまり，人手によるコーディング作業を避けて通れない場合が多いというのが現実である．

大規模なシステムになれば，複数の技術者が分担してコーディング作業を行うのは一般的であるが，その際にプログラムの書き方（流儀，スタイルを含む）[1] は無視できない重要な観点である．コードレビューはソースコードの内容確認を含む有用な品質保証活動である [2] が，その際には他人の書いたソースコードを読むことになり，書き方の不統一（個人差）はレビュー作業により多くの手間と時間をかけさせる結果となる．あわせて，プログラム内容の誤解や誤りの見落としを行ってしまうリスクも高まると考えられる．つまり，“読みやすい” コードを書くことは，間接的にはコード品質の効率的な維持・向上につながるといえる [3], [4]．このことは特定の企業内の閉じた開発だけでなく，オープンソースソフトウェア開発においてもいえることである．実際，オープンソースソフトウェア開発の場合，新機能の追加や不具合の修正が行われる際にはパッチがリポジトリへ反映される前にパッチ開発者とは別の人間によるレビューを必要としているプロジェクトも多い．

客観的に “読みやすいコードとは何か” を定義するのは難しいが，本稿ではあらかじめ定められた規則，即ちコーディング規約に従って書かれたソースコードには一定レベル以上の読みやすさ（可読性）があるものと考えることにする．これまで，現場での経験も踏まえてさまざまなコーディング規約が提唱・活用されており [5]，それらへの準拠を自動的に確認する静的解析ツールもいくつかある．コーディング規約によっては 100 を超える多数の規約（ルール）が定義されていることも珍しくなく [5], [6]，実際にそういった静的解析ツールを使用すると，プログラマの意図しない規約違反（警告）が多く検出されることもある．その場合，それら全てに対応するようコードを書き換えるべきか，あるいは内容によってはそこまで気にしないでよいものか判断に迷う場面も考えられる．そこで本稿では，実際のオープンソースソフトウェア開発プロジェクトにおける開発過程において，どういった規約違反が多く見られ，その中でどういったものは結果的に改善されていくのかといった観点から実データの収集と分析を行い，コーディング規約と静的解析ツールの有効活用に向けた基礎データの報告を行う．

以下，2 章でコーディング規約とその利用について概説する．そして，3 章で二つのオープンソース開発プロジェ

[1] 愛媛大学大学院理工学研究科
Graduate School of Science and Engineering, Ehime University, Matsuyama, Ehime 790–8577, Japan
[2] 愛媛大学総合情報メディアセンター
Center for Information Technology, Ehime University, Matsuyama, Ehime 790–8577, Japan
[a)] oda@se.cite.ehime-u.ac.jp

クトに対するデータ収集と分析結果を示し，4 章で関連研究についてふれる．最後に 5 章で本稿のまとめと今後の課題について述べる．

2. コーディング規約とその利用

2.1 ソースコードの可読性

ソースコードの可読性とは，そのソースコードを書いた人以外がその内容を読む際の “読みやすさ” であり，他者が記述内容や処理の流れをどれほど容易に理解できるかを表している．ソフトウェア開発においては，コードの可読性を可能な限り高めておくことが重要である．一般にソフトウェア開発では，いったんシステムが動作するようになったとしても，それで全てが完了（完成）ということはない．その後もユーザからの要望により機能の追加や変更が必要になったり，新たに見つかった不具合を除去するための修正が必要になったりする．その際にはソースコードの内容を確認することになるが，その場合に過去に他の技術者が書いたコードを理解して修正しなければならないこともある．なお，自分が過去に書いたソースコードであっても，時間の経過とともに詳細な内容を忘れてしまっている場合もある．いずれの場合でも，コードの可読性が高ければ高いほど，各コードの動作や処理内容をより早く理解し，効率良く保守作業を行うことができる．

2.2 コーディング規約

前述したように，コードの可読性を一意に評価するのは難しい．プログラミングの作法といったかたちでの提唱はいくつか知られている [1], [3] が，それらが絶対的な存在というわけでもない．実際には開発組織ごとにコーディング規約（コーディング標準）を定め，それに準拠するように努めるというのが現実解であると思われる．つまり，必ずしも各個人によって最適な書き方ではないかもしれないが，一定のルールを定めておくことでコーディングにおける個人差を可能な限り排除し，無用なばらつきを抑制することで一定の可読性を維持しようというものである．

改めて，コーディング規約とはプログラム作成時の規則や作法のことである．これはプログラミング言語の文法とは異なり，ある機能を実現する際に複数の書き方が可能な場合，どのような書き方を採用するかを約束事として決めたものである．あらかじめコーディング規約を決めておくことで，可読性を一定水準に保つことができる．コーディング規約は組織により独自に設けられていることが多い．例えば，Java 言語について次のものを含め多くの規約が存在している：

- Sun Java コーディング規約 [7]
- Google Java Style[8]
- 永和システムマネジメント Java コーディング標準 [9]
- Acroquest Technology Java コーディング規約 [10]
- 富士通 Java 標準化規約 [11]，等

2.3 コーディング規約チェックツール

前述したコーディング規約では，ソースコードの見た目に関するルールだけでなく，変数名の命名規則やコメント記述に関するルールといったものも含まれ，項目数が 100 を超えるものも珍しくない．そういった多数のルールについて，手作業で一つ一つ確認していくのは非現実的であり，非効率でもある．そのため，コーディング規約への準拠を自動的に確認するツールを使用するのが一般的である．実際，数多くのツールが販売されていたり，無償で公開あるいはオープンソース化されていたりする．例えば，次のようなツールがある：

- Checkstyle[12]
- PMD[13]
- CX-Checker[14]
- Jtest[15]，等.

例えば，Checkstyle を用いて Sun Java コーディング規約への準拠を確認したい場合，**図 1** のようにして規約違反を検出できる．

2.4 規約違反への対応

実際にコーディング規約チェックツールを使用すれば，いくつかの規約違反（警告）が検出されることになる．それらは場合によってはプログラマが意識していなかったような指摘事項も含まれているかもしれない．あらかじめ特

```
$ java -jar checkstyle-5.7-all.jar
                          -c sun_checks.xml XXX.java

Starting audit...
XXX.java:0:  Missing package-info.java file.
XXX.java:14:1:  Utility classes should not have a
public or default constructor.
XXX.java:15:1:  '{' は前の行にあるべきです。
XXX.java:17:5:  Javadoc コメントがありません。
XXX.java:18:5:  '{' は前の行にあるべきです。
XXX.java:22:  Line has trailing spaces.
XXX.java:23:5:  Javadoc コメントがありません。
XXX.java:23:29:  Parameter args should be final.
XXX.java:24:  Line has trailing spaces.
XXX.java:24:5:  '{' は前の行にあるべきです。
XXX.java:36:  Line has trailing spaces.
XXX.java:56:9:  '}' は同一行にあるべきです。
Audit done.
```

図 1 Checkstyle を用いて XXX.java の Sun Java コーディング規約への準拠を確認する手順（実際は 1 行）と結果の例

Fig. 1 Example of checking the coding style in XXX.java by Checkstyle with Sun Java Coding Rule.

表 1　分析対象の概要
Table 1　Summary of target open source products.

	SQuirrel SQL Client	Eclipse Checkstyle Plug-in
データ収集期間	2001 年 11 月 13 日 ～ 2014 年 7 月 7 日	2003 年 5 月 11 日 ～ 2014 年 2 月 15 日
最新コミットの Java ファイル数	4,454	430
コミット回数	6,459	882

定のコーディング規約が定められており，その規約へ準拠することが求められている場面であれば，全ての違反に対してそれが解消されるようにコードを修正しなければならない．ところが，特定のコーディング規約が決まっているわけではない場面，あるいは規約は決まっているがツールがそれ以外のルールについてもサポートしていて違反（警告）を表示している場合，そのような指摘項目についてプログラマはどこまで気にして対応したらよいのかという疑問もある．

そこで本項では，ツールの使用・未使用に関わらず，実際のソースコードではどういった規約違反が多く検出される傾向にあるのか，そして，どういった規約違反は開発が進むにつれて結果的に解消される傾向にあるのかを定量的に分析することを考える．そのような傾向をつかむことができれば，ソースコードの書き方という観点について，経験的に重要と考えられているポイントが明確になると思われる．そうすることで，静的解析ツールは単なるコーディング規約への準拠チェックに留まらず，書き方に対する指摘を通じた品質の維持・向上にも役立つと考えられる．

以上のことから，本稿では規約違反に着目し，実プロジェクトからデータ収集を行い，

- 実際にどういった規約違反が多く検出されるのか？
- そのような中でどういった規約違反は開発の経過とともに解消されていくのか？

という観点での分析を行っていく．

3. 分析

3.1 対象

本稿では**表 1** に示す二つの著名なオープンソースソフトウェアを分析対象とした．これらを選んだ主な理由は (1) Java 言語で開発されている，(2) 10 年以上にわたって開発・保守されており，十分な機能と安定性を有している，(3) バージョン管理システム Git で管理されている，という三つである．(1) は分析対象とするコーディング規約を一つに絞るためである．今回は著名でありツールも無償で利用できるということから Sun Java コーディング規約を Checkstyle によって確認することとした．(2) は可能な限り著名で長期にわたって開発されているソフトウェアを対象とすることで，本稿での分析結果の一般性を高めることが狙いである．(3) は，分析に際して多数のチェックアウトを行う必要があるため，リポジトリのコピーを容易に入手可能であるというのが理由である．

3.2 データ収集

本稿ではデータ収集を以下の手順（1）～（4）で行った（**図 2**）．

(1) リポジトリ（コピー）の取得
本分析では，各コミットにおけるスナップショットに対してコーディング規約への準拠を確認するため，多数回のチェックアウトを実施することになる．そのため，あらかじめリポジトリのローカルコピーを作成しておく．Git リポジトリの場合，標準でそのクローンを生成できるようになっている．

(2) 各コミットにおけるスナップショットの取得
各コミットで変更が施された Java ファイルを取得する．ただし，開発の途中で名前が変更されたり，削除されたりするファイルも存在するため，ここではデータ収集期間の最終日において存在している Java ファイルのみを対象とする．

(3) 各コミットに対するコーディング規約チェックの実施
(2) で取得した Java ファイルに対して Checkstyle によるコーディング規約違反の検出を行う．

(4) コミット間での規約違反・警告の差分抽出
コミット間で新たに検出された規約違反，並びに修正後に解消された規約違反を差分データとして記録する．

3.3 結果と考察

SQuirrel SQL Client 及び Eclipse Checkstyle Plug-in における各コミットでのコーディング規約違反件数の推移を**図 3** 及び **図 4** にそれぞれ示す．

図 3 及び 図 4 から分かるように，いくつかのコミット時に一時的に多数の違反が検出されるという現象が見られた．詳細な追跡まではできていないが，該当するコミットで多くのコード追加が行われていたことが主な原因として考えられる．あわせて共通で見られた現象として，そのような急激な違反数の増加が発生した後，その次のコミットではそれらのほとんどが解消されていたことも興味深い．特定の種類の違反に限られる話かもしれないが，どちらのプロジェクトでもプログラムの書き方に対して何らかの制御・調整が行われた可能性があると考えられる．そういっ

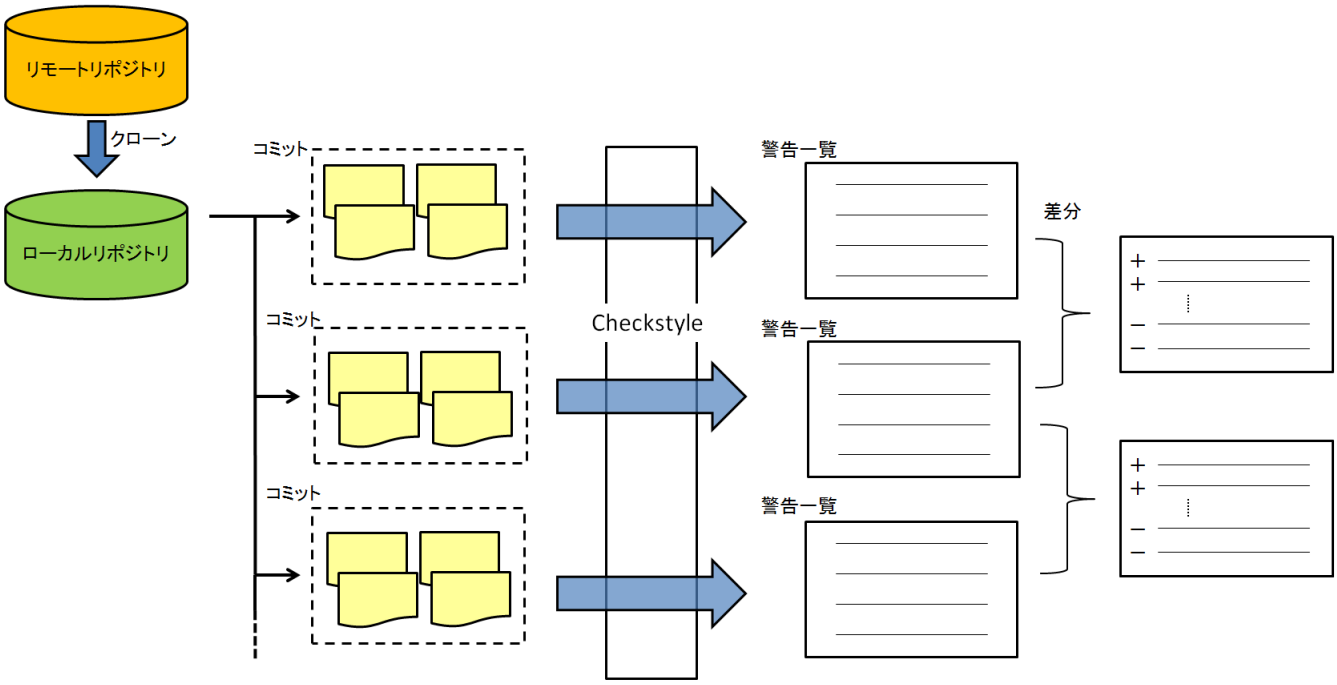

図 2 データ収集の手順
Fig. 2 Procedure of data collection.

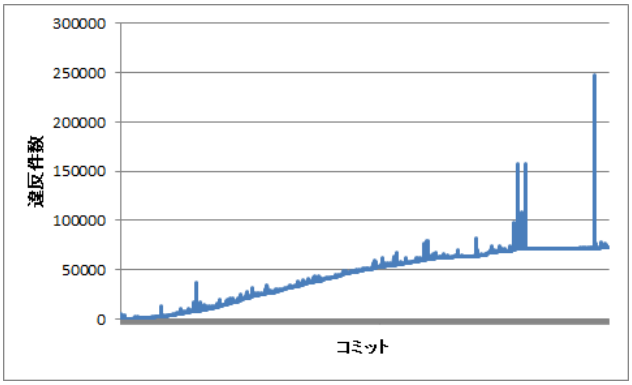

図 3 SQuirrel SQL Client における各コミットでのコーディング規約違反件数の推移
Fig. 3 Changes in the number of coding standard violations for SQuirrel SQL Client.

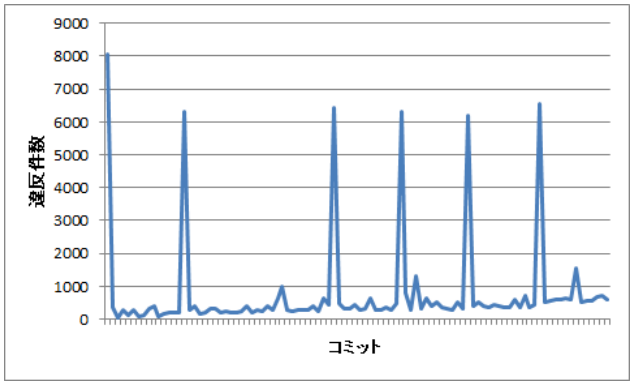

図 4 Eclipse Checkstyle Plug-in における各コミットでのコーディング規約違反件数の推移
Fig. 4 Changes in the number of coding standard violations for Eclipse Checkstyle Plug-in.

た突発的な増減を除くと，違反件数は緩やかな増加傾向にあるが，開発が進むにつれて安定しているようにも見える．このことは，ソースファイルの総数が関係しているものと思われる．

次に，各ソフトウェアにおける規約違反の内容について見ていく．**表 2** に各ソフトウェアにおいて検出された規約違反の総数と最新版までに解消された違反総数，並びに両者の比（解消率）を示す．同表の (A) 列では，コミット間での比較において初めて登場した規約違反の合計数を表している．一方 (B) 列は，逆に前のコミット時点では提示されていたが，その次のコミットでは提示されなくなった規約違反，つまり “解消” されたものの合計数を表している．そして，(B)/(A) でもってどれだけの規約違反が結果的に解消されたのかを示している．

二つのソフトウェアは開発言語は同じ Java であるが，開発プロジェクトやドメインは異なっており，検出された規約違反の種類数にも差が見られた．解消率にも差があり，SQuirrel SQL Client の方がより多くの種類・数の違反が検出されたにも関わらず，より高い解消率（96%）となった．今回の結果だけで Eclipse Checkstyle Plug-in におけ

表 2　検出された違反の種類数と件数
Table 2　Numbers of detected violations.

対象ソフトウェア	種類数	(A) 新規に検出された違反総数	(B) 最終的に解消された違反総数	解消率 (B)/(A)
SQuirrel SQL Client	61	1,971,216	1,895,446	96%
Eclipse Checkstyle Plug-in	37	51,851	45,523	88%

る解消率 88% が低いかどうかを一概に決定することは難しいが，少なくとも SQuirrel SQL Client の方がコーディング規約違反になるようなプログラムの書き方にはより敏感に対応されていることがうかがえる．

各ソフトウェアで検出された件数の多い違反（上位 10 件）のデータを **表 3** 及び **表 4** に示す．

実際，各ソフトウェアにおいて，検出された上位 10 件の違反でもって全体の大部分を占めていた．SQuirrel SQL Client では全体の約 85%，Eclipse Checkstyle Plug-in では約 92% をそれぞれ占めていた．それらのうち以下に示す 6 件は共通の違反項目であった．

(1) No. 4：“行が 80 文字を超えています。”
1 行を構成する文字数が 80 を超えてしまっている，つまり，長い行になっている場合に指摘される規約違反である．
行が長いということは，それだけ行全体を見渡しにくくなるため，コードの可読性を下げる要因の一つと考えられる．あわせて，内容確認の際に誤りを見落とすリスクも考えられ，その意味ではバグ混入のリスクを高めているともいえる．ただし，80 文字という閾値が妥当であるかどうかは議論の余地があると思われる．

(2) No. 5：“Parameter ＜変数名＞ should be final.”
メソッドの引数がそのメソッド内で読み込みにしか使われないにもかかわらず，その引数が final 宣言されていない場合に指摘される規約違反である．
その場合，final 宣言を行っておくことで無用な誤りの混入を防止できる効果がある．

(3) No. 30: “Javadoc コメントがありません。”
メソッドに対して Javadoc コメントが書かれていない場合に指摘される規約違反である．
Javadoc コメントはそのメソッドのプログラママニュアルの生成にも使用される有用なコメントであり，この記述がないことはその後のプログラムの再利用性に悪影響を及ぼす恐れがある．

(4) No. 3：“Line has trailing spaces.”
その行の末尾に空白が入っている場合に指摘される規約違反である．
そのような空白の存在は，ソフトウェアの動作に直接的に影響しないかもしれないが，エディタによってはコーディング作業で無用なキー操作を要する場合があり，その存在を気にするプログラマも少なくない．なお，複数行にまたがった文字列リテラルの場合は，実行結果に差が生じる可能性があるため注意が必要である．

(5) No. 28：“名前 ＜変数名＞ はパターン ＜正規表現＞ に一致しなければなりません。”
これは変数名が所定の命名規則に従っていない場合に指摘される規約違反である．＜正規表現＞の部分には例えば “`[a-z][a-zA-Z0-9]*`” といったパターンが入る．その場合，下線で始まる変数名（例えば `_add` 等）が使われていた場合に指摘されることになる．
変数が所定の命名規則に従っていない場合，他のプログラマがその変数の意味合いや働きを誤解してしまう可能性がある．つまり，そのような変数の存在はレビューやコード修正の際に無用な混乱を引き起こしかねない．

(6) No. 53：“＜値＞ はマジックナンバーです。”
プログラム中に数値や文字列を直接記述している場合，いわゆる “マジックナンバー” が使われている場合に指摘される規約違反である．
マジックナンバーは保守性の低下を招くといわれており，実際，他のプログラマがコードの変更を行う際に，一部の定数のみを修正し忘れたり，誤った値に置き換えてしまったりする危険性がある．そのような誤りを未然に防止するため，マジックナンバーとなるような値はあらかじめ定数として定義しておくことが望ましい．

以上の六つの規約違反が二つのソフトウェアに共通して多く見られた項目である．つまり，これらが実際に多く登場している規約違反の代表例であるといえる．結果的に (4) は開発環境に依存しているところがあった可能性は否定できないが，残りの五つは単なるプログラムの書き方（スタイル）という問題ではなく，複数人での共同開発においては気にすべき項目であるように思われる．実際，85% 以上の違反は結果的に解消されているため，開発時に大きな支障が起こったとは考えにくいが，この種の規約違反は新規コードの追加といったことが原因で現実に起こりやすかったものと推察される．

さらに個別の解消率について見ていく．

SQuirrel SQL Client では平均の解消率が 96% であったことを鑑みると，違反 (1) と (4) がこれを下回っていた．つまり，“長すぎる行（80 文字を超える）” と “行末の空白” が比較的解消されずに残っていることがわかる．この

表 3　SQuirrel SQL Client において検出数の多かった違反
Table 3　Top 10 violations in SQuirrel SQL Client.

No.	違反件数	解消件数	解消率	内容	備考
4	414022	390340	94%	行が 80 文字を超えています。	(1)
63	313784	286871	91%	'{' は前の行にあるべきです。	
30	204211	202834	99%	Javadoc コメントがありません。	(3)
5	202942	202778	99%	Parameter ＜変数名＞ should be final.	(2)
3	161025	142957	89%	Line has trailing spaces.	(4)
40	126496	125025	99%	＜記号＞ の後にホワイトスペースがありません。	
28	69717	69705	99%	名前 ＜変数名＞ はパターン ＜正規表現＞ に一致しなければなりません。	(5)
53	68616	68507	99%	＜値＞ はマジックナンバーです。	(6)
35	65845	65845	99%	＜変数名＞ には @param タグが必要です。	
34	55173	54978	99%	＜記号＞ の前にホワイトスペースがありません。	

表 4　Eclipse Checkstyle Plug-in において検出数の多かった違反
Table 4　Top 10 violations in Eclipse Checkstyle Plug-in.

No.	違反件数	解消件数	解消率	内容	備考
4	14551	12507	86%	行が 80 文字を超えています。	(1)
5	8085	7136	88%	Parameter ＜変数名＞ should be final.	(2)
30	5557	4947	89%	Javadoc コメントがありません。	(3)
3	5412	4756	88%	Line has trailing spaces.	(4)
57	3937	3338	85%	'}' は同一行にあるべきです。	
12	3019	2711	90%	変数 ＜変数名＞ は private とし、アクセッサメソッドを持つべきです。	
28	2703	2432	90%	名前 ＜変数名＞ はパターン ＜正規表現＞ に一致しなければなりません。	(5)
53	1852	1633	88%	＜数値＞ はマジックナンバーです。	(6)
62	1361	1198	88%	Missing package-info.java file.	
50	1192	1055	89%	＜クラス名＞ のクラス情報が取得できません。	

ことから，これらは開発を進めていくにあたって，特に大きな問題とはならなかったとも考えられる．

Eclipse Checkstyle Plug-in においても同様に，違反 (1) “長すぎる行（80 文字を超える）” の解消率は全体での平均を下回っていた．つまり，こちらのプロジェクトでも行の長さについては，少なくとも 80 文字という閾値の下では大きな問題とならなかったと推察される．

逆に，違反 (2)，(3)，(5) 及び (6)，即ち “メソッドパラメータが final 化されていない”，“Javadoc がない”，“変数名が命名規則に従っていない” 及び “マジックナンバーを使っている” という四つの規約違反はいずれのソフトウェア開発においても比較的高い割合で解消されている．実際のところ，プログラマが今回のようなコーディング規約への準拠を確認したかどうかは不明なため，単純に指摘を受けて改善したというような結論は出せないが，少なくとも最終的には “指摘されないようなコード” に改善されている．つまり，そのままではなく，そのように改善する方が望ましいと考えて行動したものと推察される．それゆえ，これらの規約違反は多くのプログラマにとっても起こりやすく，かつ，重要なものであると考えられる．

今回は，以上の傾向が現実にあるということを確認できたことが本稿の貢献であると考える．ただし，バグ修正との関係については分析できていないため，その点に関しては今後の課題としたい．あわせて，違反 (1) “長すぎる行（80 文字を超える）” に対する解消率が比較的低いことについても，さらなる解析が必要であると考えている．具体的には，多くのプログラマが “行の長さ” をさほど気にしていないという結果なのか，あるいは “80 文字” という基準が現実に即していないためなのかが明確になっていない．この点についても追加での分析が必要であると考えている．

4. 関連研究

Takai ら [16] は静的解析ツールによって検出された違反件数に関するメトリクスをいくつか提案し，開発過程におけるそれらの推移に着目した研究を行っている．文献 [16] では CX-Checker[14] を使って MISRA-C に違反する箇所を検出し，プロジェクトにおける開発状況がどのように進捗しているかを考察している．開発過程におけるコーディング規約違反の変化に着目している点は本研究と共通して

いるが，本研究では違反の内容に着目した分析を行っており，その点が Takai らの研究とは異なっている．

Mizuno ら [17] は静的解析ツールによる警告メッセージに対してテキストマイニング技術を適用し，フォールト潜在モジュール予測を行うという研究を行っている．文献 [17] では PMD [13] を使って Eclipse BIRT プロジェクトのコード解析を行い，そこでの警告メッセージに対するテキストマイニングを行ってフォールト潜在モジュール予測実験を行い，良好な結果を得ている．静的解析ツールによるコーディング規約違反の内容に着目している点は本研究も同じであるが，本研究ではどういった違反が実際に多く検出され，その中でどういったものが多く改善されているのかを分析している点が異なる．

Boogerd ら [18] は MISRA-C への違反とフォールト検出との関係について実証的な研究を行っている．文献 [18] によると，一部の規約違反に関してはフォールト潜在箇所との関連性が見てとれるが，そうでない規約も多く見られ，結果としては適切なルールの選択が必要であるとも述べられている．使用しているコーディング規約や対象は異なるが，文献 [18] で主張されている観点は本研究での結果とも関連している．今回は，プログラマが結果としてどういった規約違反を犯しやすいか，またどういったものは結果的に修正されていくのかを分析するに留まったが，今後，フォールト潜在性との関係についても分析していく必要があると考える．

5. おわりに

本稿では，ソースファイルにおけるコーディング規約違反に着目し，実際のオープンソースソフトウェア開発において，どういった規約違反が多く検出され，また，その中でどういったものは開発が進む中で解消されていくのか，あるいはそのまま残っていくのかという観点から定量的な調査を行った．

10 年以上にわたって開発・保守されている著名な二つのオープンソースソフトウェア SQuirrel SQL Client 及び Eclipse Checkstyle Plug-in についてデータ収集と分析を行ったところ，次の六つの規約違反が実際に多く登場することがわかった：

(1) “行が 80 文字を超えています。”
(2) “Parameter ＜変数名＞ should be final.”
(3) “Javadoc コメントがありません。”
(4) “Line has trailing spaces.”
(5) “名前 ＜変数名＞ はパターン ＜正規表現＞ に一致しなければなりません。”
(6) “＜値＞ はマジックナンバーです。”

このうち，(2)，(3)，(5) 及び (6) については，比較的高い割合で最終的には解消される傾向にあることが分かった．実際のところ，プログラマがコーディング規約への準拠を確認したかどうかは不明なため，単純に指摘を受けて改善したというような結論は出せないが，少なくとも最終的には “指摘されないようなコード” に改善されていることが分かった．つまり，そのままではなく，そのように改善する方が望ましいと考えて行動したものと推察される．それゆえ，これらの規約違反は多くのプログラマにとっても起こりやすく，かつ，重要なものであると考えられる．

一方，(1) “長すぎる行（80 文字を超える）” については，どちらのソフトウェアにおいても解消されずに残っている案件が比較的多く見られた．この現象についてはさらなる解析が必要であると考えられる．具体的には，多くのプログラマが “行の長さ” をさほど気にしていないという結果なのか，あるいは “80 文字” という基準が現実に即していないためなのかが明確になっていない．今後，この点についてさらなる分析を行っていく必要があると考えられる．また，バグ修正との関係についても分析できていないため，その点も今後の課題としたい．

参考文献

[1] Kernighan, B. W. and Pike, R.: *The Practice of Programming*, Addison Wesley, Indianapolis, Indiana (1999).
（福崎 俊博 訳：プログラミング作法，アスキー出版局 (2000)）．
[2] Wiegers, K. E.: *Peer Reviews in Software*, Addison Wesley, Indianapolis, Indiana (2002).
（大久保 雅一 訳：ピアレビュー，日経 BP 社 (2004)）．
[3] Boswell, D. and Foucher, T.: *The Art of Readable Code*, O’Reilly Media, Sebastopol, CA (2012).
（角 征典 訳：リーダブルコード，オーム社 (2012)）．
[4] 縣 俊貴：良いコードを書く技術，技術評論社 (2011).
[5] 森崎雅稔：最新 Java コーディング作法，日経 BP 社 (2011).
[6] MISRA-C 研究会：組込み開発者におくる MISRA-C:2004，日本規格協会 (2006).
[7] Sun: Sun Java コーディング標準，Oracle（オンライン），入手先 ⟨http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html⟩（参照 2014-10-12）．
[8] Google: Google Java Style, Google (online), available from ⟨http://google-styleguide.googlecode.com/svn/trunk/javaguide.html⟩ (accessed 2014-10-12).
[9] 平鍋健児：Java コーディング標準，永和システムマネジメント（オンライン），入手先 ⟨http://www.objectclub.jp/community/codingstandard/CodingStd.pdf⟩（参照 2014-10-12）．
[10] Acroquest Technology: Java コーディング標準，Acroquest Technology（オンライン），入手先 ⟨http://www.acroquest.co.jp/webworkshop/javacordingrule/Acroquest_JavaCodingStandard_6_7.pdf⟩（参照 2014-10-12）．
[11] 富士通：Java 標準化規約，富士通（オンライン），入手先 ⟨http://jp.fujitsu.com/solutions/sdas/technology/develop-guide/2-regulation.html⟩（参照 2014-10-12）．
[12] Project, C.: Checkstyle, Checkstyle Project (online), available from ⟨http://checkstyle.sourceforge.net/⟩ (accessed 2014-10-12).
[13] PMD: PMD, InfoEther (online), available from

⟨http://pmd.sourceforge.net/⟩ (accessed 2014-10-12).
[14] Sapid Project: CX-Checker — ルールのカスタマイズが可能な C 言語コーディングチェッカ, 名古屋大学, 愛知県立大学, 南山大学（オンライン）, 入手先 ⟨http://cxc.sapid.org/⟩（参照 2014-10-12）.
[15] TechMatrix: 静的解析 — Jtest, TechMatrix（オンライン）, 入手先 ⟨https://www.techmatrix.co.jp/quality/jtest/staticanalysis/index.html⟩（参照 2014-10-12）.
[16] Takai, Y., Kobayashi, T. and Agusa, K.: Software Metrics based on Coding Standards Violations, *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, IEEE, pp. 273–278 (2011).
[17] Mizuno, O. and Nakai, M.: Can Faulty Modules be Predicted by Warning Messages of Static Code Analyzer?, *Advances in Software Engineering*, Vol. 2012, No. 924923, pp. 1–8 (2012).
[18] Boogerd, C. and Moonen, L.: Assessing the value of coding standards: An empirical study, *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, IEEE, pp. 277–286 (2008).