社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

共同開発環境におけるコーディングスタイルの変更に関する調査

小倉　直徒†　　柗本　真佑†　　畑　　秀明††　　楠本　真二†

† 大阪大学大学院 情報科学研究科　〒 565–0871 大阪府吹田市山田丘 1–5
†† 奈良先端科学技術大学院大学 情報科学研究科　〒 630–0192 奈良県生駒市高山町 8916–5
E-mail: †{n-ogura,shinsuke,kusumoto}@ist.osaka-u.ac.jp, ††hata@is.naist.jp

あらまし　ソースコード内における，インデントの種類や演算子前後のスペースの有無，変数名の命名規則といった記法に関するルールを，コーディングスタイル（以下スタイル）と呼ぶ．スタイルはプログラムの本質に影響を与えないことから，正解/不正解のはっきりした問題ではなく，開発者ごとのプログラミング経験や文化が反映された一種の「好み」の問題であるといえる．しかしながら，複数の開発者が参加する共同開発環境では，その一貫性や統一作業が重要となる．本研究では，スタイル統一が容易な開発環境実現の予備調査として，共同開発環境におけるスタイルの実情を調査する．調査では 4 種類の OSS を対象として，ソフトウェア内でのスタイルの一致度を算出し，この値に基づいて 4 つの調査項目について議論する．

キーワード　Java，コーディングスタイル，リポジトリマイニング，スタイルの不一致度

1. ま え が き

ソースコード内における，インデントの種類（スペース/タブ）や演算子前後のスペースの有無，変数名の命名規則といった記法に関するルールを，コーディングスタイル（以下スタイル）と呼ぶ．このスタイルは，プログラムの持つ本質的な振る舞いには一切影響を及ぼすことはないものの，開発者に対するコードの視認性や可読性に強い影響を与える [1]．スタイルの種類は多様であり，統合開発環境 Eclipse では 300 種類以上のスタイル設定項目が用意されている．

スタイルはプログラムの本質に影響を与えないことから，正解/不正解のはっきりした問題ではなく，開発者ごとのプログラミング経験や文化が反映された一種の「好み」の問題であるといえる．例えば，分岐命令（if や for）直後のスペースを挿入するか/否かや，ブロックの開始を表す括弧 { を分岐命令と同じ行に書くか/次の行に書くかは，好みの分かれやすいスタイルである．また，スタイルはプログラミング環境の進化に伴って変化しているといえる．横幅 80 文字で強制折り返しというスタイルは，パンチカードの物理的な横幅制限から引き継いだものであるが，近年のディスプレイ解像度の増加に伴ってその値は大きくなりつつある．実際に Google による Java のコーディング規約 [2] では，一行の最大文字数は 100 と指定されている．

このようにスタイルは正解のない好みの問題ではあるものの，複数の開発者が参加する共同開発環境ではその一貫性や統一作業が重要となる．スタイルの不一致はコードの可読性の低下を招くだけでなく，改行の位置が好みと異なる/スペースを挿入したくなるといった，本質的なプログラミング作業時の思考の妨げに繋がるとも考えられる．一般的に，スタイルはコーディング規約等で規定されるが，全ての開発者にそのルールを遵守させることは容易ではない．また，スタイルの種類が多岐に渡ることから，全種類のスタイルについて規定することも現実的ではない．近年では，IDE やフォーマッタ (注1) 等のツールの登場しており，ローカル環境内で一貫したスタイルを適用することは容易になった．しかし，ツールごとのスタイル規定値が異なるため，複数開発者の間での一貫性の確保はやはり容易ではない．

本研究では，スタイルの統一が容易な開発環境実現の予備調査として，共同開発環境におけるスタイルの実情を調査する．調査では 4 種類の OSS を対象として，ソフトウェア内でのスタイルの不一致度を算出し，この値に基づいて 4 つの調査項目について議論する．本研究の貢献は以下のとおりである．

- ソースコードからスタイル特徴を抽出する手法を提案した．
- ソフトウェアの開発過程においてスタイルの違反と修正が繰り返し行われていることを明らかにした．
- 違反の多いスタイルの種類を特定した．

2. ス タ イ ル

2.1 スタイルの定義

本研究では，コンパイル後の生成物に影響を及ぼさない，ソースコード内の表現のことを，スタイルと定義する．スタイルには，トークン間の空白の有無，識別子名の命名や順序，あるい

(注1)：http://checkstyle.sourceforge.net/

はプログラム構文などが含まれる．大別して以下に分類した．

a） トークン同士の区切り要素

- スペース: 括弧の前後にスペースを挿入するか等
- インデント: クラスのボディでインデントするか，インデントのスペースの数等
- 改行: if のあとに改行を挿入するか等

b） 識別子の命名規則と宣言順序

- 命名規則: キャメルケースかスネークケースか，ハンガリアンを用いるか等
- 宣言順序: メソッド宣言が先か変数宣言が先か等

2.2 本研究で対象とするスタイル

スタイルの設定項目はエディタやフォーマッタなどのツールによって異なる．それらで設定可能な項目を全て調査し対象とするのは現実的ではない．そのため本研究では，Eclipse で設定可能な空白の挿入に関するもののみを対象とする．対象となる設定項目は 161 個存在する．空白の挿入に関する設定で指定可能な値は `insert` または `do_not_insert` の 2 値のみである．`insert` はトークン間に空白を挿入することを示し，`do_not_insert` は空白を挿入しないことを示す．以降は `insert` を空白あり，`do_not_insert` を空白なしと表す．

対象とする項目の一部を表 1 に示す．“空白なしの例”は空白なしの設定でフォーマットした場合，“空白ありの例”とは空白ありの設定でフォーマットした場合の，Java ソースコードの一部を示している．

2.3 関 連 研 究

スタイルの研究は長年に渡って多数行われている．特に抽象構文木を用いてソースコードをフォーマットする手法が数多く提案されている [3] [4] [5]．Spinellis らは長期間の開発におけるスタイル違反の変化について調査した．43 年間にわたる Unix の開発において，後述されるスタイルの不一致度が年々減少していることを明らかにした [6]．Buse らは開発者の感じるソースコードの可読性はスタイルによって変化することを明らかにした．また全ての開発者にとって最適なスタイルは存在しないことを報告した [7]．Bacchelli らは違反スタイルの発見がコードレビューの重要な役割の 1 つであることを報告した [8]．すなわちスタイルの統一のために手間を要しているといえる．Caliskan らはソースコードから開発者を特定する手法を提案した [9]．この手法では，スタイルと抽象構文木の特徴は開発者ごとに異なることを利用している．

2.4 SI 度

スタイルが統一していないことの指標として **SI(Style Inconsistency) 度** [6] を用いる．SI 度とは，あるスタイル設定の取りうる値が a または b で，i 番目のスタイル設定が適用された箇所の出現数をそれぞれ a_i および b_i としたとき，i 番目のスタイルの $\mathrm{SI_i}$ 度は，

$$\mathrm{SI_i} = \frac{min(a_i, b_i)}{a_i + b_i}$$

で表される．また，全てのスタイル項目における $\mathrm{SI_{all}}$ 度は，

$$\mathrm{SI_{all}} = \frac{\sum_{i=1}^{n} min(a_i, b_i)}{\sum_{i=1}^{n} a_i + b_i}$$

で表される．Eclipse の空白に関する設定項目は 161 個あるため，i は $1 \leqq i \leqq 161$ である．ただし後述するようにスタイルの抽出において，スタイルの項目を新規に定義するため，実際は 161 個よりも多くなる．SI 度は 0 から 0.5 までの値をとり，値が大きいほどスタイルは不統一であることを示す．例えばある特徴名において，特徴値が空白ありである特徴の数が 10 で，空白なしである特徴の数が 2 のとき，その特徴名 i の $\mathrm{SI_i}$ 度は $2/12 = 0.16$ となる．空白ありとなしそれぞれの出現数が同数であるとき，$\mathrm{SI_i}$ 度は 0.5 となり，その値は SI 度の上限である．

3. スタイル特徴の抽出手法

3.1 スタイル特徴

スタイル特徴とは，スタイル設定によってソースコードが変化するときの，以下の要素の組み合わせである．

- ソースコード上の位置
- スタイル特徴名
- スタイル特徴値

スタイル特徴名は Eclipse のスタイルで定義された設定可能な値の名前である．スタイル特徴値は空白ありまたは空白なしの 2 値である．スタイル特徴値が空白ありの場合は，位置の示すトークンの前または後に空白が存在することを示し，空白なしの場合は空白が存在しないことを示す．

3.2 手法の概要

本手法の概要を図 1 に示す．本手法の入力は，Java のソースコードである．出力は入力ソースコードから抽出したスタイル特徴の集合である．

3.3 Step1: 字句解析・構文解析

入力されたソースコードを，トークン単位の文字列に分割する．また，構文解析により，分割したトークンが抽象構文においてどの要素に属するかを判定する．

3.4 Step2: スタイル特徴名のマーキング

抽出対象に該当する抽象構文のトークンを探し，そのトークン前後のトークン間に対して，スタイル特徴名をマーキングする．

3.5 Step3: スタイル特徴値の抽出

スタイル特徴名をマーキングされたトークン間に空白が存在するか判別し，空白の有無をスタイル特徴値として抽出する．抽出したスタイル特徴値を，トークンの位置とスタイル特徴名を組み合わせてスタイル特徴として出力する．

ただし以下に該当する場合，上記とは異なる処理を行う．それぞれの例として図 2 に示す．

a） 同一の空白に複数の特徴名がマーキングされた場合

図 2a のように，複数のスタイル設定が同一の空白の有無に変化を及ぼす場合がある．このとき，複数の特徴名が空白にマーキングされる．複数のスタイル設定のうち，1 つでも特徴値が空白ありなら空白が挿入されるため，新たな特徴名 `a|b` を定義し，そのトークンからは新たな特徴名の特徴のみを抽出する．ただし空白が存在しない場合，特徴名 `a` および `b` それぞれの値を，空白なしとして抽出する．

表 1: 空白のスタイル設定の例

スタイル名	空白なしの例	空白ありの例（崗は該当箇所）
space_before_opening_brace_in_type_declaration	**class** HelloWorld{	**class** HelloWorld␣{
space_before_opening_paren_in_method_declaration	**void** main(String[] args)	**void** main␣(String[] args)
space_after_opening_paren_in_method_declaration	**void** main(String[] args)	**void** main(␣String[] args)
space_before_opening_bracket_in_array_type_reference	**void** main(String[] args)	**void** main(String␣[] args)

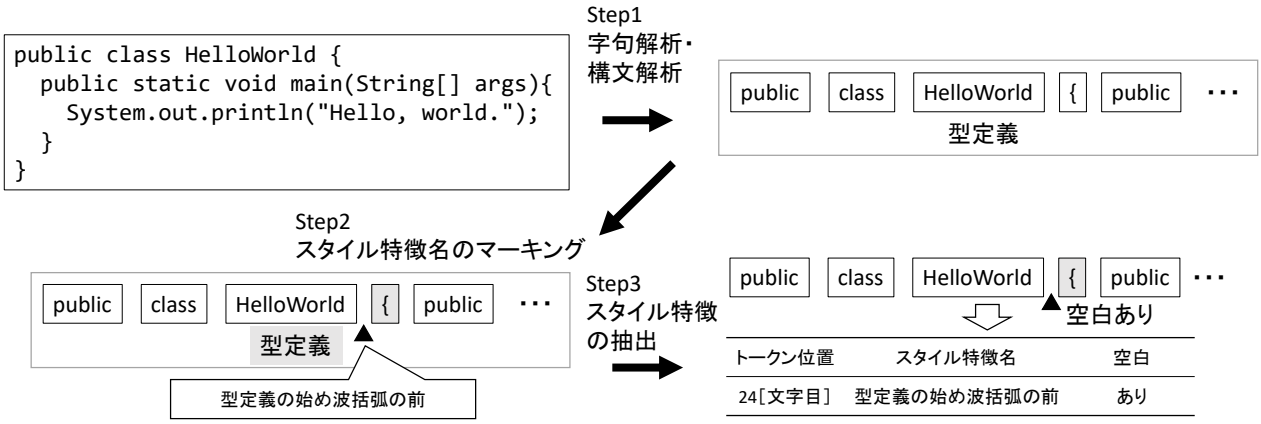
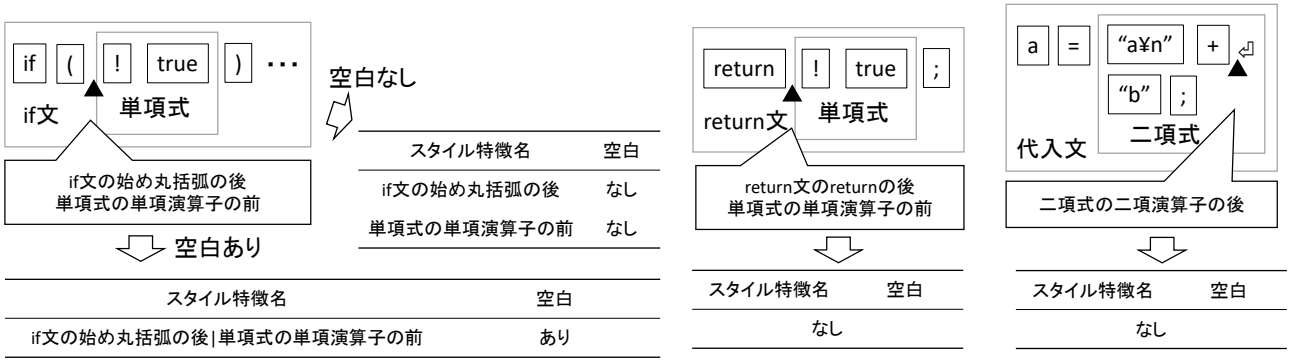


図 1: 手法の概要



(a) 同一の空白に複数の特徴名がマーキングされた場合

(b) 構文上必ず存在する空白に特徴名がマーキングされた場合

(c) 行頭または行末のトークンの空白の場合

図 2: スタイルの抽出に関する場合分け

b） 構文上必ず存在する空白に特徴名がマーキングされた場合

図 2b のように，構文上必ず空白を挿入しなければならないことがある．例えば，return 文において `return` キーワードの後に単項演算子が続く場合は，トークンの区切りとして必ず空白を挿入しなければならない．そのような場合，マーキングされた空白からはスタイル特徴を抽出しない．

c） 行頭または行末のトークンの空白の場合

図 2c のように，行内の最初のトークンの前や，最後のトークンの後にトークン間からは，スタイル特徴を抽出しない．これらのトークン間には改行が含まれるが，改行は最大桁指定など空白とは異なる設定に基いて挿入されるためである．

3.6 Spinellis らの手法との違い

Spinellis らは C 言語のソースコードから様々なメトリクスを取得するツール *cqmetrics*(注2) を公開しており，この中にはスタイル特徴を抽出する機能が存在する．このツールには以下の欠点が存在する．

- 構文解析を行わないため，抽出されるスタイル特徴の特徴名は構文上の要素名を含まない．例えば，for 文と if 文の始め丸括弧から同一のスタイル特徴名の特徴を抽出する．
- 3.5 節で挙げた場合分けの処理を行っていない．そのため，誤った特徴を抽出している場合がある．例えば，単項演算子の前には空白を挿入しないという規約に従っている場合でも，図 2b と図 2a における単項演算子の前の空白の有無は異なるため，スタイルが不一致であると判定される．

2 つの欠点により，SI_{all} 度が本手法よりも高く検出されている可能性がある．

(注2)：https://github.com/dspinellis/cqmetrics

4. 研究目的と調査項目

4.1 研究目的

本研究の目的は，ソフトウェア開発においてスタイルの統一にかかる労力を明らかにすることである．研究目的を達成するため，以下の調査項目を設定した．

4.2 調査項目

4.2.1 調査項目 1

Spinellis らは C 言語で実装された Unix におけるスタイルの違反は漸減していると報告している [6]．この知見が Java ソフトウェアに対しても当てはまるのかを確認するため，複数の Java ソフトウェアを対象に開発過程における SI_{all} 度の変化を調査する．

4.2.2 調査項目 2

ソフトウェア開発において，スタイル違反がどの程度発生しているかを調査する．指標として，版管理上の全コミットのうち，スタイル違反を含むコミットの割合を算出する．

4.2.3 調査項目 3

ソフトウェア開発において，スタイル修正がどの程度発生しているのかを調査する．調査項目 2 と同様に，指標としてスタイル修正を含むコミットの割合を算出する．

4.2.4 調査項目 4

どのようなスタイルの違反が多いのかを調査する．違反の多いスタイルをコーディング規約等のドキュメントにおいて強調することで，今後のスタイル違反が少なくなることが期待される．

5. 実　　験

5.1 準　　備

3 章で述べた手法をもとにソースコードからスタイル特徴を抽出するツールを，*Format Feature Extractor* として実装した．本ツールはウェブサイト上 (注3) で公開している．本ツールの制限として，抽出可能なスタイル特徴は Eclipse で設定可能な空白に関するスタイル特徴のみである．

5.2 実験対象

本実験は複数の開発者によって開発された 4 つの Java のソフトウェアを対象に行った．対象ソフトウェアは git で版管理されており，リポジトリは GitHub から取得した．対象ソフトウェアの概要を表 2 に示す．“行数”は最新リビジョンでの Java ソースコードの行数を表す．

5.3 実験手順

5.3.1 調査項目 1 に対する実験の手順

4 つのソフトウェアの開発過程における SI_{all} 度の変化を調査する．master または trunk ブランチに属する全リビジョンを対象として，各リビジョンの全ソースコードからスタイル特徴を抽出し，SI_{all} を算出する．また同時に，対象リビジョンの Java のソースコード行数も計測する．

5.3.2 調査項目 2 に対する実験の手順

本実験では，開発過程においてスタイル違反を含むコミットがどの程度行われたかを明らかにする．

実験ではまず対象ソフトウェアにそのソフトウェアで規定された正しいスタイル設定の集合を決める．この正しいスタイル設定の集合は，コーディング規約等のドキュメントから生成することも可能であるが，全てのスタイルについて詳細に決められているとは限らない．そのため，本実験では各リビジョンで抽出された特徴のうち，過半数を占める特徴値を正しいスタイルと定義する．このスタイル集合をもとに，Java のソースコードに追加された行に違反スタイルが含まれているかを判定する．

対象とするコミットは，1 つ以上の Java ソースコードが変更されたコミットである．

5.3.3 調査項目 3 に対する実験の手順

本実験では実験 2 で検出されたスタイル違反のうち，後のコミットでスタイル修正されたものを対象とする．スタイル修正のコミットとは，空白のみの変更が行われた行において，正しいスタイル設定の集合に違反したスタイル特徴が全て修正されたものを含むコミットのことである．すなわち，同一のスタイル特徴名を持つスタイル違反を全て修正した行が存在した場合，スタイル修正が行われたとみなす．

5.3.4 調査項目 4 に対する実験の手順

最新リビジョンを対象に各特徴名 i の SI_i 度を算出する．ただし出現数が 100 以上のスタイルのみを対象とする．

5.4 結果と考察

5.4.1 調査項目 1 に対する実験の結果と考察

各ソフトウェアのリビジョンごとの SI_{all} 度と LOC の推移を図 3 に示す．開発過程の後半において，Guava と JUnit4 の SI_{all} 度は減少傾向にあるが，log4j と SpirngFramework では変化に傾向はない．また LOC について，log4j 以外は増加の傾向にあるが，log4j は増加の傾向にあるとはいえない．

既存研究 [6] では 1973 年から 2015 年までの 37 年間という長期間に渡って SI_{all} 度の変化を調査した．1995 年以前は SI_{all} 度とその変化は大きいが，1995 年以降の SI_{all} 度は小さくその変化も小さい．理由として，最近はエディタやフォーマッタ等のツールの充実により，スタイルの統一が容易になったことが挙げられる．また，スタイルを統一させることの重要性が十分広まったことが考えられる．

本実験で対象としたソフトウェアは全て 1995 年以降に開発が開始されており，Unix における SI_{all} 度の変化が小さい時期である．そのため，本実験で得られた結果も，SI 度が明らかに

表 2: 対象ソフトウェアの概要

ソフトウェア	開発期間 [年]	リビジョン数	開発者数	行数
Guava	7	4,136	114	374,603
log4j	15	3,275	21	59,780
JUnit4	15	2,115	159	39,722
SF(注3)	8	13,312	207	18,134

(注3)：http://sdl.ist.osaka-u.ac.jp/~n-ogura/2016_format-feature-extractor

(注3)：Spring Framework

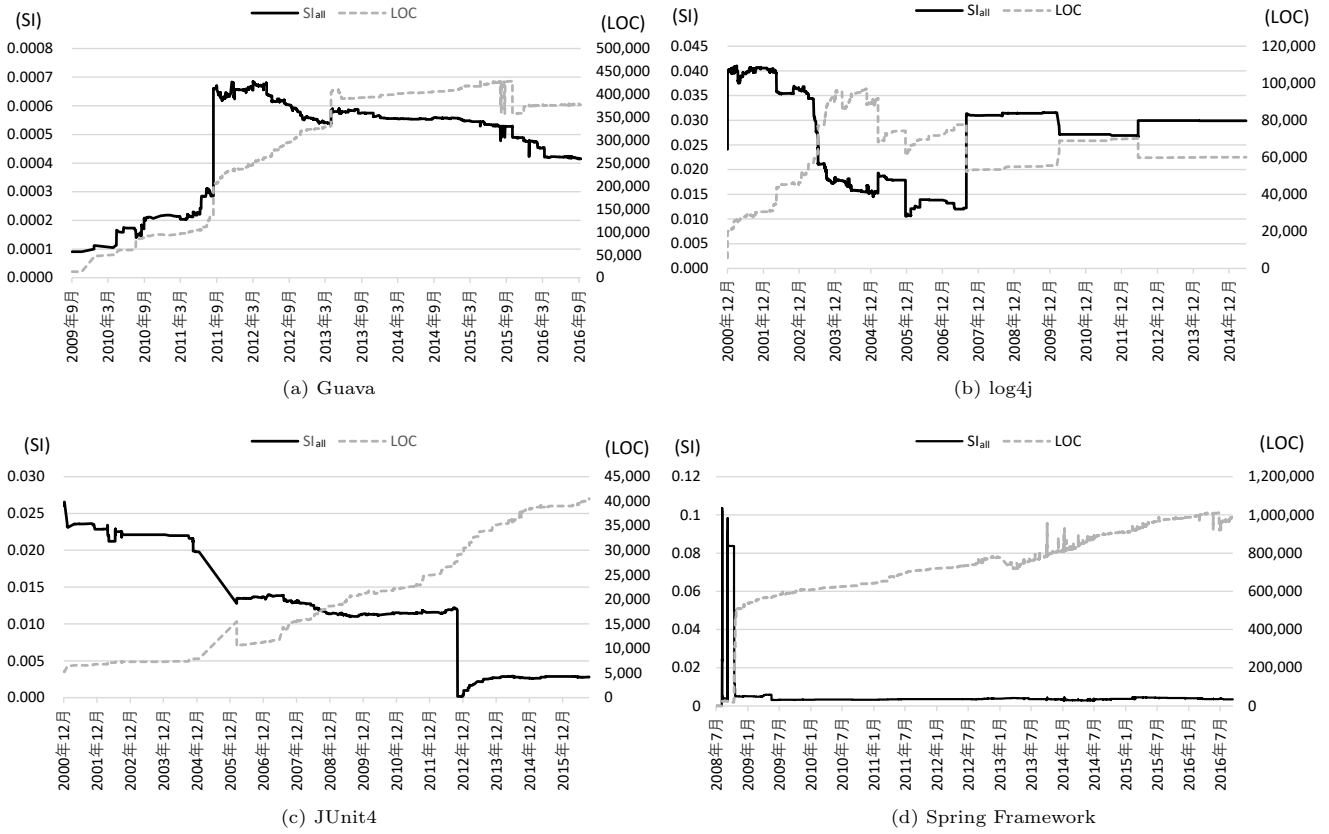


図 3: 各ソフトウェアにおける SI_{all} 度と LOC の変化

減少しているとはいえない結果となった．

JUnit4 では 2012 年 11 月に SI_{all} 度が非常に小さくなっている．この期間において新しいコーディング規約を導入しており，全ての Java ファイルに対してフォーマッタを適用したため，SI_{all} 度が小さくなったと思われる．

5.4.2 調査項目 2 に対する実験の結果と考察

実験の結果を表 3 に示す．“スタイル違反を含むコミット数”とは，“対象コミット”のうちそのコミットで追加された行にスタイル違反のスタイル特徴を含んでいる数である．log4j では対象コミットのうち半分がスタイル違反を含んでおり，ソフトウェアの保守に大きな影響を及ぼしている可能性がある．一方，SI_{all} 度が開発過程において一貫して小さかった Guava は，スタイル違反を含むコミットも少ない．

5.4.3 調査項目 3 に対する実験の結果と考察

本実験の結果を表 4 に示す．“スタイル修正を含むコミット数”とは，“対象コミット”のうちそのコミットでスタイル修正のみが行われた行を含んでいる数である．log4j ではスタイル修正が多数行われており，スタイル違反を放置するのではなく修正しようとする実情が明らかになった．Guava はスタイル修正を含むコミット数が小さく，スタイル違反がソフトウェア開発に与える影響は小さい．

5.4.4 調査項目 4 に対する実験の結果と考察

最新リビジョンにおいて最も SI_{all} 度の大きかった log4j を対象に実験を行った．表 5 に違反されやすいスタイルの一覧を示す．この表では，ソースコード全体の中で 100 回以上出現しているスタイルのうち，SI_i 度が高い上位 10 件のスタイルのみが示されている．

ほとんどのスタイルが SI_i 度 0.1 以上（すなわち，10 件中 1 件が違反）であり，これらのスタイルは開発者ごとの好みが分かれやすいスタイルであるといえる．特に 2 番目の if 直後の空白は出現数が 1 万以上でありながら，SI_i 度が 0.48 と極めて高く，log4j の中ではほぼ一貫性がないスタイルだと考えられる．

また，違反されやすい構文要素の種類も確認できる．具体的には，配列初期化に関するスタイル（#1，#3，#7，#10）や，二項演算子に関するスタイル（#6，#8）などである．特に二項演算子に関するスタイルは，前述の if 直後の空白と同様に出

表 3: スタイル違反を含むのコミット数

ソフトウェア	対象コミット数	スタイル違反を含むコミット数	割合 [%]
Guava	3,510	166	4.7
log4j	2,198	1,123	51.1
JUnit4	1,321	306	23.1
SF	10,210	1,688	16.5

表 4: スタイル修正のコミット数

ソフトウェア	対象コミット数	スタイル修正を含むコミット数	割合 [%]
Guava	3,510	11	0.3
log4j	2,198	199	9.1
JUnit4	1,321	51	3.9
SF	10,210	162	1.6

表 5: log4j における違反の多いスタイル（出現数が 100 以上の上位 10 件のみ）

#	スタイル名 (i)	SI_i 度	出現数	最頻値	空白ありの例（菌は該当箇所）
1	space_before_closing_brace_in_array_initializer	0.50	132	-	`int[]{1,2,3␣}`
2	space_before_opening_paren_in_if	0.48	11,687	空白あり	`if␣(i>0)`
3	space_after_opening_brace_in_array_initializer	0.46	114	空白なし	`int[]{␣1,2,3}`
4	space_before_opening_paren_in_for	0.36	233	空白なし	`for␣(i=0;i<10;i++){`
5	space_before_opening_paren_in_catch	0.33	360	空白なし	`}catch␣(Exception e){`
6	space_after_binary_operator	0.22	3,207	空白あり	`i=i+␣1`
7	space_before_opening_brace_in_array_initializer	0.22	179	空白あり	`int[]␣{1,2,3}`
8	space_before_binary_operator	0.21	3,470	空白あり	`i=i␣+1`
9	space_after_closing_paren_in_cast	0.17	373	空白あり	`(short)␣10`
10	space_after_comma_in_array_initializer	0.08	324	空白あり	`int[]{1,␣2,␣3}`

現数と SI_i 度が高い．

これら違反されやすいスタイルについては，コーディング規約等で開発上のルールとして明記するか，何かしらのツールによって一貫性を保つべきであると考える．別の解釈としては，log4j の中ではスタイルの一貫性にあまり関心がなく，スタイルの不一致が放置されやすいといった可能性もある．しかしながら，1 節でも述べたとおりスタイルの不一致は可読性や視認性の低下のみならず，プログラミング作業時の思考の妨げにつながると考えられる．また，OSS においては他の開発者の参入の妨げになるとも考えられる．スタイルの一貫性をどのようなツールでどのように確保するかに関しては，今後の検討課題である．

6. 妥当性への脅威

4 つのソフトウェアを対象に実験を行った．異なるソフトウェアを対象に実験を行えれば，得られる結果も異なる．

また，本実験のためにスタイル特徴を抽出するツールを開発した．本ツールに不具合が含まれていて正しい実験が行えていない可能性がある．本ツールはソースコードとともにウェブサイト上で公開しており，本実験は再現可能である．

また，本ツールで抽出可能なスタイルは，Eclipse で設定可能なトークン前後の空白に関するスタイルのみである．インデント幅や改行などのスタイル設定は抽出できず，そのようなソースコードの変化も検出できない．したがって本実験で得られたスタイル違反や修正のコミット数は，他のスタイルを検出対象に加えた場合よりも小さい．

7. あとがき

本研究では，スタイル統一が容易な開発環境実現の予備調査として，共同開発環境におけるスタイルの実情を調査した．

結果，SI_{all} 度の推移はソフトウェアによって異なり，推移に一般性はなかった．また，全てのソフトウェアにおいてスタイルの違反と修正が行われており，開発者は本質的な振る舞い以外の変更に労力が割かれていることが分かった．違反されやすいスタイルは配列初期化や二項演算子に関するスタイルであった．共同開発においては，スタイルの統一を支援する何らかの仕組みが必要であるといえる．

今後の課題として，空白以外のスタイル特徴を抽出可能とすることや，本実験をより多くのソフトウェアへ適用し一般性を向上すること，スタイルの違反や修正は開発者ごとに特徴があるのかに関する調査が挙げられる．版管理上でスタイルの統一を支援する開発環境の実現も課題である．

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究 (S)（課題番号：JP25220003），および文部科学省研究費補助金若手研究（B）（課題番号：JP26730155）の助成を得て行われた．

文　献

[1] P.W. Oman and C.R. Cook, “A taxonomy for programming style,” Proceedings of the 1990 ACM Annual Conference on Cooperation, pp.244–250, CSC ’90, ACM, New York, NY, USA, 1990. http://doi.acm.org/10.1145/100348.100385

[2] https://google.github.io/styleguide/javaguide.html

[3] R.D. Cameron, “An abstract pretty printer,” IEEE Softw., vol.5, no.6, pp.61–67, Nov. 1988.

[4] D.C. Oppen, “Prettyprinting,” ACM Trans. Program. Lang. Syst., vol.2, no.4, pp.465–483, Oct. 1980.

[5] L.F. Rubin, “Syntax-directed pretty printing-a first step towards a syntax-directed editor,” IEEE Transactions on Software Engineering, vol.9, no.undefined, pp.119–127, 1983.

[6] D. Spinellis, P. Louridas, and M. Kechagia, “The evolution of c programming practices: A study of the unix operating system 1973–2015,” Proceedings of the 38th International Conference on Software Engineering, pp.748–759, ICSE ’16, ACM, New York, NY, USA, 2016.

[7] R.P.L. Buse and W.R. Weimer, “Learning a metric for code readability,” IEEE Trans. Softw. Eng., vol.36, no.4, pp.546–558, July 2010.

[8] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” Proceedings of the 2013 International Conference on Software Engineering, pp.712–721, ICSE ’13, IEEE Press, Piscataway, NJ, USA, 2013.

[9] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, “De-anonymizing programmers via code stylometry,” Proceedings of the 24th USENIX Conference on Security Symposium, pp.255–270, SEC’15, USENIX Association, Berkeley, CA, USA, 2015.