修士学位論文

題目

コーディングスタイルの自動変換ツール
～共同開発時におけるコーディングスタイルの統一と各開発者の好みの両立を目指して～

指導教員
楠本　真二　教授

報告者
小倉 直徒

平成 29 年 8 月 17 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻

平成 29 年度 修士学位論文

コーディングスタイルの自動変換ツール
～共同開発時におけるコーディングスタイルの統一と
各開発者の好みの両立を目指して～

小倉 直徒

内容梗概

ソースコード内のインデント幅や括弧前後の空白の有無といった記法のルールを，コーディングスタイル（以下スタイル）という．スタイルは開発者ごとに好みが異なるため，共同開発においてしばしば問題となる．一般的にはスタイルの規約を定めることにより，プロジェクト内でのスタイルの統一が図られている．しかし好みと異なるスタイルを強制されることにより，開発者のソースコードの理解が阻害される恐れがある．また，コードレビューによってスタイルの規約違反を見つけ修正を行なうなど，開発者はスタイルの統一のために多大な労力を割いている．

本研究では，共同開発におけるプロジェクトのコーディングスタイルを統一しつつ，各開発者好みのスタイルを利用できるツールを提案する．本ツールは，ソースコードを手元に取得する際は開発者好みのスタイルに，ソースコードをコミットする際はプロジェクトの共通スタイルに自動的に変換する．本ツールを提案するにあたり，実際のソフトウェア開発において行われたスタイルの変更を調査し，スタイルの違反や修正に要する開発者の労力を明らかにした．また，ツールの有用性を評価するため性能評価実験および有用性評価実験を実施し，本ツールがソフトウェア開発において有用であることを示した．

主な用語

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，版管理システム

目 次

図 目 次

表 目 次

1 はじめに

ソースコード内における，インデントの種類（スペースまたはタブ）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．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，あるいは次の行に書くかは，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 文字での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ため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 やスタイルフォーマッタ[1]等のツールが登場しており，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．

本研究の目的は，複数の開発者が参加する共同開発環境において，以下 2 点を達成することにある．

- 版管理システム上のソースコードのスタイルを統一
- 開発時には各開発者が好みのスタイルを利用可能

この目的を達成するために，版管理システム上でスタイルの自動変換を行うツールを提案し実装する．提案ツールは開発者が版管理システムにソースコードをコミットする際に，自動的にソースコードのスタイルを変換する．提案ツールに開発者好みのスタイルと，プロジェクトの共通スタイルを設定しておくことで，ソースコードを手元に取得する際は開発者好みのスタイルに，ソースコードをコミットする際はプロジェクトの共通スタイルに自動的に変換される．

予備調査として，上記に挙げたようなスタイルの違いが実際のソフトウェア開発においてどの程度

[1] http://checkstyle.sourceforge.net/

発生しているかを調査する．さらに，ツールを用いた際の応答速度の低下度合いの調査，及び被験者実験による提案ツールの有用性評価を行う．

本研究の貢献は以下のとおりである．

スタイルに関する予備調査

- 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．

ツールの提案と評価

- スタイルの統一が容易な開発環境を実現するためのツールを提案した．
- 提案ツールが現実的な時間で動作することを確認した．
- 提案ツールが開発者の作業に適したスタイルを提供することを確認した．

本論文では，2 章で準備としてスタイルについて述べる．3 章ではソースコードからスタイルを抽出する手法について述べる．4 章では予備調査の目的と調査項目について述べる．5 章では予備調査の手順とその結果および考察について述べる．6 章では予備調査の結果を受けて，提案するツールの詳細について述べる．7 章では提案ツールの評価として性能評価および被験者実験について，その手順と結果および考察について述べる．8 章では本論文に関連する研究を紹介する．9 章では本研究における妥当性に対する議論について述べる．最後に，10 章では本研究のまとめと今後の研究課題について述べる．

2 準備

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．

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

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 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** [3] 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，

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

表 1: 空白のスタイル設定の例

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

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

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

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 $1 \leq i \leq 161$ 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 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，SI_{i} 度は 0.5 となり，その値は SI 度の上限である．

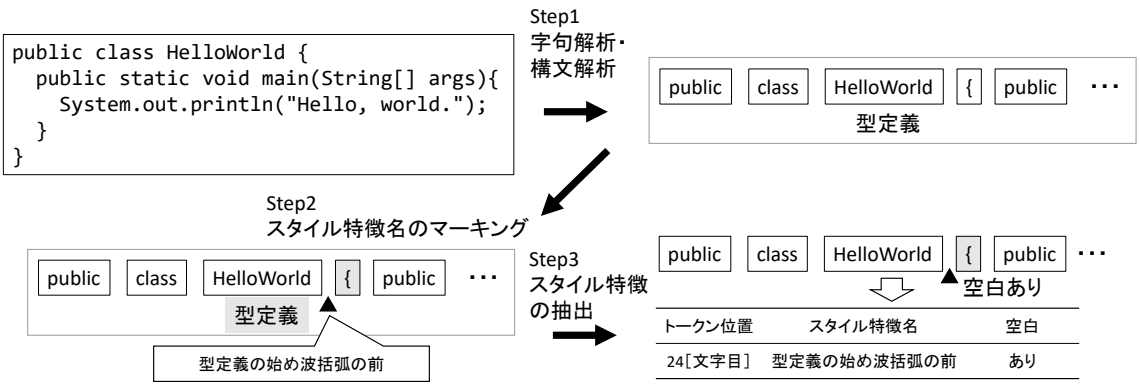


図 1: 手法の概要

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

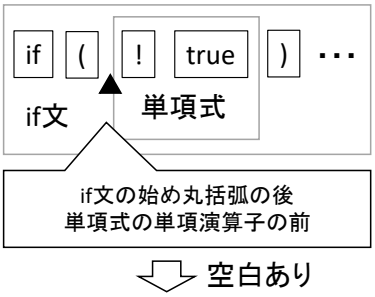
本手法の概要を図 1 に示す．本手法の入力は，Java のソースコードである．出力は入力ソースコードから抽出したスタイル特徴の集合である．

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．

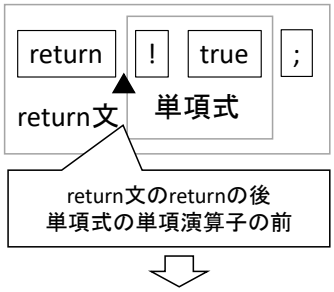


空白なし

スタイル特徴名	空白
if文の始め丸括弧の後	なし
単項式の単項演算子の前	なし

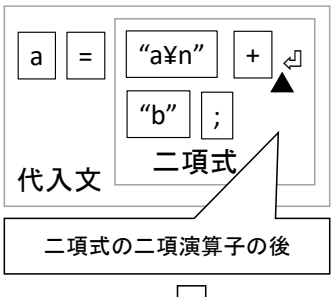
スタイル特徴名	空白
if文の始め丸括弧の後|単項式の単項演算子の前	あり

(a) 同一の空白に複数の特徴名がマーキングされた場合



スタイル特徴名	空白
	なし

(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



スタイル特徴名	空白
	なし

(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

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．それぞれの例を図 2 に示す．

同一の空白に複数の特徴名がマーキングされた場合　図 2(a) 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．

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 2(b) 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．

行頭または行末のトークンの空白の場合　図 2(c) 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[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 2(a) と図 2(b) 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．

[2]https://github.com/dspinellis/cqmetrics

4 予備調査

4.1 調査の目的

予備調査の目的は，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．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: SI 度の変化

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 [3]．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: スタイル違反のコミットの割合

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: スタイル修正のコミットの割合

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: 違反の多いスタイル

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．

4.3 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[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．

4.4 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．

[3] http://sdl.ist.osaka-u.ac.jp/~n-ogura/2016_format-feature-extractor

4.5 実験手順

4.5.1 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．

4.5.2 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．

4.5.3 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．

4.5.4 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．

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
SpringFramework	8	13,312	207	18,134

4.6 結果と考察

4.6.1 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 [3] 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．

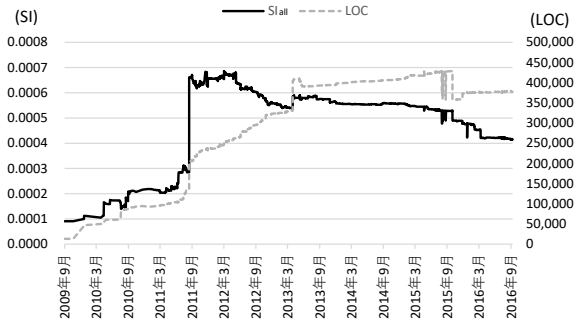
JUnit4 では 2012 年 11 月に SI_{all} 度が非常に小さくなっている．この期間において新しいコーディング規約を導入しており，全ての Java ファイルに対してフォーマッタを適用したため，SI_{all} 度が小さくなったと思われる．

4.6.2 調査項目 2 に対する実験の結果と考察

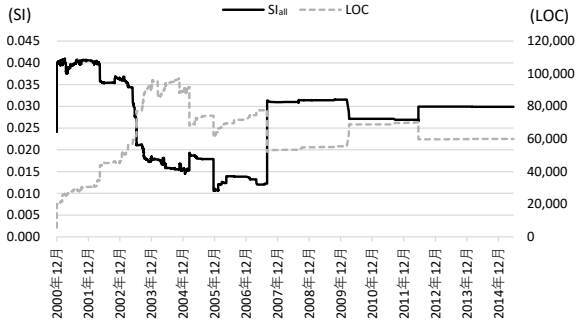
実験の結果を表 3 に示す．“スタイル違反を含むコミット数” とは，“対象コミット” のうちそのコミットで追加された行にスタイル違反のスタイル特徴を含んでいる数である．log4j では対象コミットのうち半分がスタイル違反を含んでおり，ソフトウェアの保守に大きな影響を及ぼしている可能性がある．一方，SI_{all} 度が開発過程において一貫して小さかった Guava は，スタイル違反を含むコミットも少ない．

表 3: スタイル違反を含むのコミット数

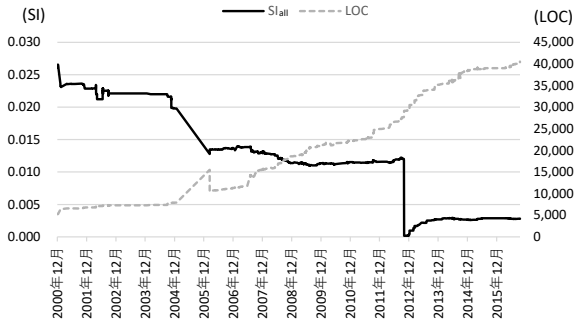
ソフトウェア	対象コミット数	スタイル違反を含むコミット数	割合 [%]
Guava	3,510	166	4.7
log4j	2,198	1,123	51.1
JUnit4	1,321	306	23.1
SpringFramework	10,210	1,688	16.5



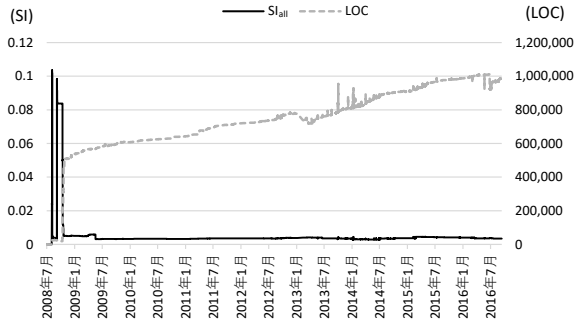
(a) Guava



(b) log4j



(c) JUnit4



(d) Spring Framework

図 3: 各ソフトウェアにおける SI_{all} 度と LOC の変化

4.6.3 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．

4.6.4 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

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
SpringFramework	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	空白ありの例（␣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	`if␣(i>0)`
3	space_after_opening_brace_in_array_initializer	0.46	114	`int[]{␣1,2,3}`
4	space_before_opening_paren_in_for	0.36	233	`for␣(i=0;i<10;i++){`
5	space_before_opening_paren_in_catch	0.33	360	`}catch␣(Exception e){`
6	space_after_binary_operator	0.22	3,207	`i=i+␣1`
7	space_before_opening_brace_in_array_initializer	0.22	179	`int[]␣{1,2,3}`
8	space_before_binary_operator	0.21	3,470	`i=i␣+1`
9	space_after_closing_paren_in_cast	0.17	373	`(short)␣10`
10	space_after_comma_in_array_initializer	0.08	324	`int[]{1,␣2,␣3}`

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．

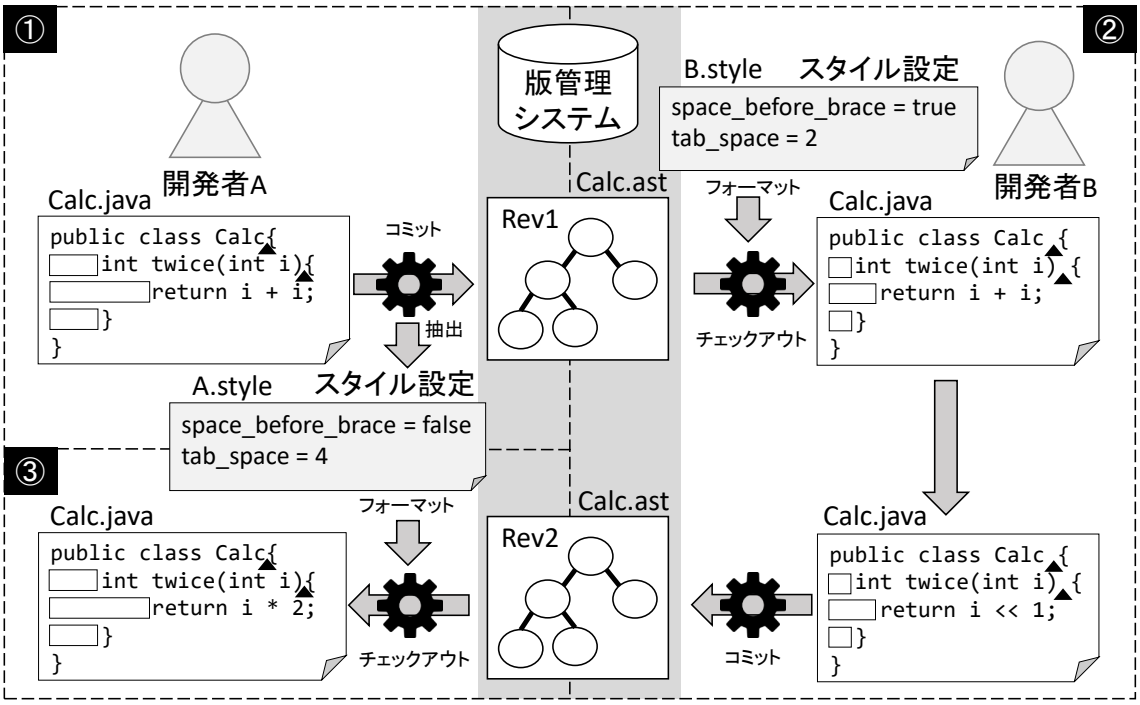


図 4: 手法の概要

5　コーディグスタイルの自動変換ツール

5.1　提案の目的

開発者は各々好みのスタイルを用いるほうが開発者は快適に開発作業をできると考えられる．しかし，事前調査で明らかになったとおり，スタイルが不統一なプロジェクトが多く存在しており，保守に影響を及ぼす．そのため，複数の開発者が参加する共同開発では，スタイルの不統一を調整する何らかの仕組みが必要である．

そこで本研究では，スタイルを統一し，同時に開発者好みのスタイルで編集できる環境を提案する．

5.2　動作の概要

本ツールの動作の概要を図 4 に示す．図中のソースコードの▲と□は 2 人の開発者 A と B の間でのスタイルの好みの差を表す．以下に共同開発における本ツールの動作例について説明する．

開発者 A が版管理システムにソースコードを登録（図 4 中 1）　開発者 A は，初めてツールを利用する開発者である．開発者 A の実装した Calc.java を本ツールを適用した版管理システムに登録する際，本ツールによって自動的に構文情報のみが版管理システムに登録される．同時に Calc.java のスタイルを解析し，開発者 A の好みのスタイル設定 A.style を生成する．

開発者 B が版管理システムからソースコードを取得し修正する（図 4 中 2）　開発者 A が登録した Calc.java を開発者 B が取得する場合を考える．開発者 B はすでに本ツールを利用しており，開発者 B の好みのスタイル設定 B.style を有している．開発者 B が Calc.java を取得するコマンドを実行すると，本ツールは自動的にスタイル設定 B.style を Calc.ast に適用し，開発者 B 好みのスタイルである Calc.java が生成される．

開発者 B は取得した Calc.java に修正を加える．図 4 では，twice メソッドを書き換えている. この際，開発者 B は好みのスタイルで修正することができる. 修正済みの Calc.java を再び版管理システムに登録する際，本ツールによって構文情報のみが登録される.

開発者 A が Calc.java を取得（図 4 中 3）　開発者 A は，開発者 B が修正した Calc.java を取得する. この際，開発者 A は開発者 A の好みのスタイル設定 A.style を有しているため, 本ツールによって Calc.ast に自動的に A.style が適用され，Calc.java が生成される．

5.3 実装

本研究では提案ツールを Java で実装した．また版管理システムは Git を前提としており，本ツールは Git の拡張機能を利用する．Git には属性という機能があり，ファイルの拡張子によってステージング（ファイルを版管理システムに登録する）時やチェックアウト（版管理システムからファイルを取得する）時の動作を変更することができる．この機能は一般的に OS 間で異なる改行コードの統一などで用いられている．

プログラムのソースコードに “スタイル変換対象”という属性を付与し，ステージング時やチェックアウト時にスタイルを変更するプログラムをソースコードに適用する．チェックアウト時には開発者好みのスタイルを適用することで，開発者の作業するソースコードは開発者好みのスタイルになっている．一方ステージング時にはプロジェクト共通のスタイルを適用することで，他の開発者と共有するソースコードは常に同じスタイルになっている．

5.4 利用方法

スタイルファイルの作成　本ツールの動作には，スタイルを定義したファイルが必要である．このファイルは，Eclipse のフォーマッタの設定から出力できる．またリポジトリ内でスタイルファイルを配布しているプロジェクトもある．これらより取得したスタイルファイルを，ホームディレクトリ下にコピーする．開発者好みのスタイル設定は `user.xml` というファイル名で，プロジェクトの共通のスタイル設定は `project.xml` というファイル名とする．

ツールのインストール　本ツールはウェブ上で公開している[1]．本ツールをウェブ上から取得後，開発者のホームディレクトリ下にインストールする．

[1] https://sdl.ist.osaka-u.ac.jp/~n-ogura/2017_jdtformatter

Git リポジトリの設定　Git リポジトリ内に以下の内容の.gitatttributes ファイルを作成する．

```
*.java filter=format
```

このファイルにより，Git リポジトリは拡張子が java であるファイルに format 属性を付与する．また，以下のコマンドを実行することで，format 属性のファイルに適用するステージング時とチェックアウト時の処理を指定する．

```
git config filter.format.clean ~/jdtformatter/bin/jdtformatter project.xml
git config filter.format.smudge ~/jdtformatter/bin/jdtformatter user.xml
```

6 評価実験

6.1 2 つの実験の概要

本ツールの評価を行うために，以下の 2 種類の実験を実施した．

性能評価実験 提案ツールが実用的な時間で処理できることを確かめるために，ツールを用いた際の性能に対する影響を調査する．

有用性評価実験 本ツールの有用性を被験者実験により調査する．被験者には開発におけるマージ作業タスクを与え，ツールありとなしの場合を比較することにより，その有用性を確かめる．さらに，アンケートを実施しツールを用いた際の使用感についてインタビューを行う．

6.2 性能評価実験

6.2.1 概要

提案ツールを用いることで，版管理システムへコミットする際の処理時間の増加（性能の低下）が予想される．本実験では，この性能の低下度合いが実用的な範囲で収まるかを確かめる．

実験では，実際のソフトウェアの開発過程で行われたコミットを手元の計算機で再現する．この各コミットに対して，ツールを用いた場合と用いない場合の両方の処理時間を計測し，その性能の低下度合いを確認する．実験対象としたソフトウェアは JUnit4 であり，実験に用いた計算機の性能は表 6 の通りである．

6.2.2 実験結果

性能評価実験の結果を箱ひげ図の形で図 5 に示す．図では 5 秒を超える外れ値を除外している．ただし，ツールありにおける最大値以上の外れ値は省略している．ツールを用いない場合，ほぼ全てのコミットは 1 秒以内に処理を終えることができていた．対し，ツールを用いた場合は，ツールを用いない場合と比べて約 17 倍程度の時間の増加が確認できた．しかしながら，ツールを用いた場合でも全体の内，約 75%は 2.5 秒以内に完了することが分かった．

表 6: 実験に用いた計算機の性能

項目名	値
OS	Windows 10 Pro
プロセッサ	Intel Xeon CPU E5-2620 v3 （2 プロセッサ）
メモリ	32GB
ストレージ	SSD 512GB （PCI-Express 接続）

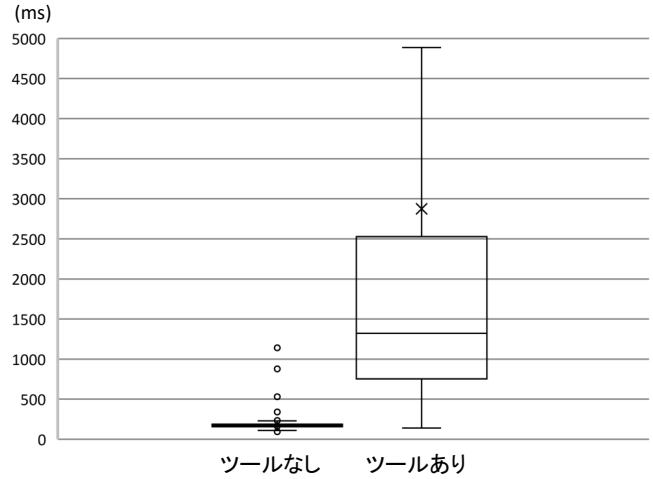


図 5: 性能評価実験の結果：コミットに要する時間の比較

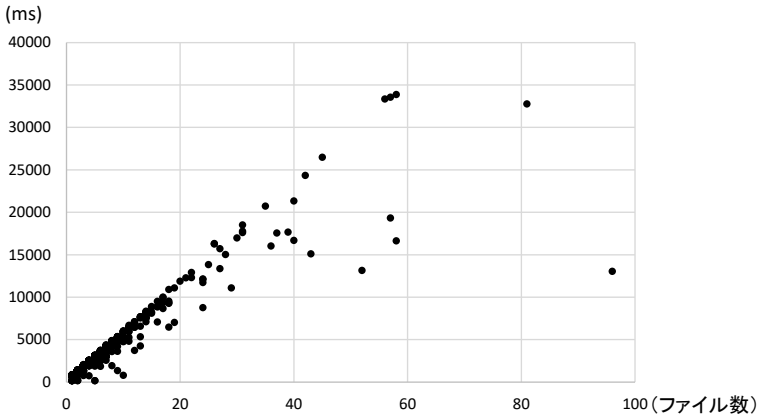


図 6: ツールありにおけるファイル数に対するコミットに要する時間

最も速度が悪化したケースは図中の外れ値のコミットであり，約 1.1 秒から約 178.4 秒に増加していた．このコミットでは約 305 個のファイルが同時にコミットされていた．

コミット対象のファイル数とコミットに要する時間に相関があるのか調査した結果を，図 6 に示す．ファイル数の最大は 488 個であったが，図では 100 個を最大値としている．コミット対象のファイル数が多いほど性能の低下度合いが大きい傾向があることを確認できる．

6.2.3 議論

提案ツールを用いた場合でもほとんどのコミットは 5 秒以内に完了することが確認できた．実際の開発においてコミットは，開発作業のある程度の区切りの際に行う作業であり，作業時間における割合は極めて小さい．そのため，5 秒という時間は実用的の範囲内であると考える．

また，コミット対象のソースコードファイル数が多く，行数が増加するほど性能が低下することが

確認できた．これは対象のファイル数が多いほど，構文木の生成処理とスタイルの変換処理が増えるためであると考えられる．近年のソフトウェア開発現場では，可能な限りコミットを小さく，かつ頻度を多くすることが一つのプラクティスとして推奨されている [4]．そのため，ファイル数や行数の増加に伴う性能低下の影響はさらに小さくなると考えられる．

6.3 有用性評価実験

6.3.1 概要

提案ツールの有用性を確認するために被験者実験を行う．実験では，被験者に対して手動によるマージ作業タスクを実施してもらう．マージ作業とは第三者の開発したソースコードの修正パッチを，修正対象となるコード本文に適用する作業のことを指す．マージ作業は一般的なソフトウェア開発現場でも用いられる行為であり，修正内容に対するコードレビューやコードインスペクション [5] の効果を得ることができる．

実験では，8 種類のマージ作業を対象に，提案ツールを用いた場合と用いない場合を比較することで，ツールの有用性を確かめる．評価には以下 3 つの尺度を用いる．

- タスク完了に要する作業時間
- 修正結果のコードに含まれる構文ミスの数
- 修正結果のコードに含まれるスタイルミスの数

時間は開発速度への影響を確認するための尺度であり，構文ミス数とスタイルミス数は開発作業の正確さへの影響を確認するための尺度である．構文ミスは Java 文法上のエラーであり，構文エラーの要因となる記述ミスのことである．スタイルミスは，修正対象のコードのスタイルと一致しない表現上の誤りのことであり，後の保守コストの増加の要因となるミスのことである．

さらに，上記の速度や正確性では計測が困難な定性的なツールの使用感を確かめるために，8 つのタスク完了後にインタビューを実施する．

6.3.2 被験者

被験者は以下を前提とする．

- 本ツールの対象とするプログラミング言語である Java を習熟している
- 本ツールの対象とする版管理システムである git を習熟している

上記を満たす被験者として，本研究室の修士 2 年 4 人，修士 1 年 2 人の合計 6 人を被験者とした．

6.3.3 実験手順

本実験は実際の開発現場での利用を想定し，被験者による手動でのマージ作業をタスクとする．マージ作業では，修正対象となるソースコードファイルと，ユニファイド Diff 形式で表されたパッチファイルの 2 つのファイルが与えられる．被験者は，パッチに書かれた修正内容どおりに修正対象のソースコードを修正する．開発現場では，他の開発者の修正作業を自分のプロジェクトに取り込む作業である，マージコマンドを用いてパッチの適用が行われる．一般的にはマージ作業にはマージツールを用いるが，複数人が同一の箇所を修正するなどした際，手動での作業が求められる．すなわち，本実験でのタスクは開発現場での利用に則っているといえる．ただし，パッチファイルから修正対象コードへのソースコードのコピー&ペーストは禁止とした．

マージ作業タスクの作成には，実際の開発で発生したバグ修正のデータセットである Defect4J[6] を用いた．タスクとして用いたバグ修正の概要を表 7 に示す．これらのバグ修正は，理解のしやすい小規模のプロジェクトで，かつ作業時間が長すぎずまた変更行に複数のスタイルが含まれているものである．削除行数および追加行数は，パッチファイルによって行う作業の大きさを表す．

被験者 6 人を無作為に 2 群に分け，それぞれ被験者 A 群および被験者 B 群とした．被験者 A 群は奇数 ID タスクの（#1，#3，#5，#7）に対してツールありで行い，偶数 ID タスク（#2，#4，#6，#8）に対してツールなしで行った．逆に被験者 B 群は偶数 ID タスクをツールありで，奇数 ID タスクをツールなしで行った．偶数と奇数でツール有無を切り分けた理由としては，タスク実施順序による慣れや学習効果を排除するためである．また被験者を 2 群に分けた理由は，被験者ごとの開発スキルの差の影響を排除するためである．

ツールありでタスクを行う際には，被験者の好みのスタイルにフォーマットされたソースコードファイルとパッチファイルを被験者に与える．この各被験者の好みのスタイルは，被験者自身が開発したプロジェクトから抽出したものを用いる．一方で，ツールなしでタスクを行う際には，被験者の好みのスタイルから 40 個無作為に選択し逆の設定を行ったスタイルを用いてソースコードファイル

表 7: タスクの概要

タスク ID	バグ修正 ID	プロジェクト	バグ修正日時	削除行数	追加行数
#1	lang13	Commons Lang	2012 年 2 月 29 日	2	24
#2	lang15	Commons Lang	2011 年 11 月 18 日	5	18
#3	lang17	Commons Lang	2011 年 7 月 15 日	20	13
#4	lang19	Commons Lang	2011 年 7 月 3 日	4	21
#5	lang32	Commons Lang	2010 年 2 月 7 日	11	18
#6	math16	Commons Math	2012 年 11 月 26 日	13	35
#7	math18	Commons Math	2012 年 9 月 30 日	6	10
#8	math28	Commons Math	2012 年 7 月 31 日	16	24

とパッチファイルをフォーマットした．これは，共同開発でありがちな好み以外のスタイルを強制させられるケースを再現するための手順である．

なお，スタイルの無作為抽出 40 個という値は事前調査の結果から導出した値である．この事前調査では，JUnit4 の主要開発者のコードスニペットから抽出したスタイルと，JUnit4 全体のスタイルを比較した場合に，48 個のスタイル設定の違いがあることが分かった．すなわち，実際のソフトウェア開発プロジェクトにおいて 40 個のスタイルが好みと異なる状況は妥当であると考えられる．

被験者の普段用いている Java の統合開発環境と，被験者の好みのコーディングスタイルと Eclipse の標準スタイル設定との差を表 8 に示す．提案ツールは開発環境に依存しないため，被験者の任意の開発環境で実験を行った．また，Eclipse を利用している被験者のうち，半数以上は Eclipse の標準スタイル設定と何らかの差を持つ．

6.3.4 結果と考察

作業時間 作業時間に関する実験結果を表 9 に示す．各セルは被験者ごとの各タスクに要した時間を秒単位で記したものである．表下部の行 “ツールなしの合計” 以下は，表の上部を元にツールの有無で各タスクの所要時間を比較した結果である．例えば，左上のセル 1,897 秒とはタスク#1 に対してツールなしでタスクを実施した被験者 B 群の値を合計した値であり，その下のセル 1,455 秒は同タスクに対してツールありで実施した被験者 A 群の値を合計したものである．また，表最下部の “ツール効果” はツール有無の行を元に得られた結果であり，ツールを用いることで作業時間が減少したか増加したかを表している．Pos は 10%以上の時間の減少が確認できた肯定的な結果を，Neg は 10%以上の時間の増加が確認できた否定的な結果を意味する．

正規分布を仮定し対応のある T 検定を行うと，$p = 0.38$ であった．よって提案ツールを用いることによる時間削減効果は確認できなかった．ツールの有無によるタスクごとの作業時間を比較すると，肯定的な結果が 2 タスク，否定的な結果が 1 タスクであり大きな差は見られなかった．これは，ツールの効果よりもタスクそのものの難易度が結果に影響していると考えられる．もう一つの要因としては，全ての被験者が学生であるため，スタイルへのこだわりが低かったことが挙げられる．長

表 8: 被験者の概要

被験者名	被験者群	統合開発環境	Eclipse のスタイルとの差
被験者 1	A	Eclipse	0
被験者 2	A	IntelliJ IDEA	3
被験者 3	A	Eclipse	12
被験者 4	B	Eclipse	2
被験者 5	B	Eclipse	14
被験者 6	B	Eclipse	0

い開発経験を持つ実務者を対象とした実験，及びタスクの難易度調整は今後の課題である．

構文ミス数　表 10 に構文ミス数に関する実験結果を示す．

ツール有無で比較すると，タスク 8 個のうち 6 個でミス数の減少が確認できた．一方，ツールなしの方がミスが少ないタスクは 1 個のみであった．正規分布を仮定し対応のある T 検定を行うと，$p = 0.17$ であった．構文ミスの数はいずれのタスクにおいても 0 個〜4 個程度と少なく，また実験に用いた被験者数およびタスク数が少なく有意差は確認できなかった．

被験者ごとに着目すると，2 名は全てのタスクにおいてミス数が 0 個であった．一方，被験者 1 はツールを用いることで大幅な構文ミス数の減少が確認できた．また被験者 5 と 6 は絶対数が少ない者の，被験者 1 と同様の傾向が確認できた．ミスのしやすさには個人差があるものの，ツールを用いた際の肯定的な効果が確認できる．

スタイルミスの数　表 11 にスタイルミス数の実験結果を示す．表下部 “ツール効果” は，表 9 と同様，10%以上の減少/増加が確認できたケースを表す．

ツールを用いることで肯定的な結果（ミス数の減少）が確認できたケースはタスク 8 個中 4 個であった．正規分布を仮定し対応のある T 検定を行うと，$p = 0.23$ であった．これらの結果から，スタイルミスに対する本ツールの効果は確認できなかった．しかしながら，本ツールを利用した場合には，コミット時にスタイルが自動的に修正されるため，スタイルミス自体はプロジェクトの保守コストに影響はしない．

インタビュー　被験者には，タスク完了後にインタビューを実施した．タスクは交互にスタイルが変化していたことを明かした上で，スタイルの違いによる被験者の評価を収集した．

表 9: 各被験者および各タスクの作業時間（秒）

被験者名	被験者群	#1	#2	#3	#4	#5	#6	#7	#8
被験者 1	A	526	517	321	534	416	516	135	312
被験者 2	A	302	431	442	622	412	793	205	712
被験者 3	A	627	631	336	653	409	538	139	507
被験者 4	B	455	525	336	476	325	425	174	331
被験者 5	B	1,028	972	596	1,005	602	519	239	833
被験者 6	B	414	373	222	347	304	262	104	440
ツールなしの合計		1,897	1,579	1,154	1,809	1,231	1,847	517	1,531
ツールありの合計		1,455	1,870	1,099	1,828	1,237	1,206	479	1,604
ツール効果		Pos	Neg	—	—	—	Pos	—	—

被験者 6 人全員が，好みでないスタイルのタスク群について，好みのスタイルのタスク群よりも作業がしにくいと答えた．また，パッチファイルのスタイルを正確にソースコードに適用できているかについては，好みのスタイルのタスク群は作業に自信がある一方，好みでないスタイルのタスク群は間違いがある可能性があると答えた．

スタイルの違いがプロジェクトの参加の意欲には影響しないという回答が多かった．プロジェクト内におけるスタイルの統一は気にならないという意見があった．一方で，エディタのフォーマッタを日常的に用いているため，フォーマッタ用の設定ファイルをプロジェクトは配布するべきであるという意見もあった．

表 10: 各被験者および各タスクの構文ミスの数

被験者名	被験者群	#1	#2	#3	#4	#5	#6	#7	#8
被験者 1	A	0	3	0	2	0	3	0	2
被験者 2	A	0	0	0	0	0	0	0	0
被験者 3	A	3	1	0	0	0	0	0	2
被験者 4	B	0	0	0	0	0	0	0	0
被験者 5	B	0	0	0	0	0	2	0	1
被験者 6	B	0	0	0	0	1	0	1	0
ツールなしの合計		0	4	0	2	1	3	1	4
ツールありの合計		3	0	0	0	0	2	0	1
ツール効果		Neg	Pos	-	Pos	Pos	Pos	Pos	Pos

表 11: 各被験者および各タスクのスタイルミスの数

被験者名	被験者群	#1	#2	#3	#4	#5	#6	#7	#8
被験者 1	A	1	2	3	6	4	3	0	5
被験者 2	A	0	0	0	0	0	0	0	6
被験者 3	A	0	0	0	3	0	1	0	6
被験者 4	B	2	3	1	7	2	4	0	1
被験者 5	B	32	1	5	3	0	0	2	1
被験者 6	B	0	2	3	0	0	2	0	1
ツールなしの合計		34	2	9	9	2	4	2	17
ツールありの合計		1	6	3	10	4	6	0	3
ツール効果		Pos	Neg	Pos	—	Neg	Neg	Pos	Pos

6.4 議論

被験者実験では，タスクの作業を通して，本ツールによる効果を定量的および定性的に評価した．定量的評価では，提案ツールによりある一定の効果が見えるものの，その結果には統計的に有意な差があるとはいえない．また，定性的評価では，すべての被験者から作業がしやすいという意見が得られたため，本ツールの有用性は被験者実験により確認されたといえる．

また本実験を実施するにあたり，被験者の普段のコーディングスタイルが，Eclipse の標準のスタイル設定からどの程度乖離しているかを調査した．被験者の半数以上が，標準の設定とは乖離した好みのスタイルがあるにも関わらず，誰ひとりとして Eclipse の設定を編集していないことが分かった．このことから，被験者らはスタイルに対するこだわりは小さいといえる．

7 関連研究

様々な開発プロジェクトにおいて，コーディングスタイルを規定した文章が存在しており，一般公開されている [7, 8, 9, 10, 11, 12, 13, 14]．当然ながら，各プロジェクトの規定スタイルには様々な違いが存在する．これらは，スタイルが正解/不正解のはっきりした問題ではないこと，及び個人によって好みの差がある一方で共同開発において無視できない問題であることを示唆しているといえる．また，本研究で対象としたような空白関係のスタイルは Java では 300 種類以上存在しており，全項目を完全に規定し全開発者に遵守させることは容易ではない．提案ツールは，ソースコード集合からのスタイルの抽出機能を備えており，スタイル規定の困難さを低減させることが可能である．また，コミット時のスタイル変換機能により，全開発者でのスタイルの遵守という問題も同時に解決できる．

プロジェクト内でのスタイルの統一を目的としたツールが数多く存在する [15, 16]．さらに，Eclipse や Netbeans，IntelliJ IDEA 等の一般的な IDE では，標準でスタイル修正ツールが搭載されており，容易に利用可能である．また，学術分野においてもスタイル修正ツールや手法がいくつか提案されている [17][18][19]．これらのツールの多くは，抽象構文木を用いてコードをフォーマットするものであり，我々の提案ツールと共通する点は多い．一方で，スタンドアローン環境での動作を前提としたツールが多く，あくまで開発者自身のローカル環境でのスタイル修正に留まる．提案ツールは，版管理システムのクライアントとして動作しており，ローカル環境だけでなく共同開発環境でのスタイルの統一が可能という観点で長所を持つ．

ツールによる自動修正ではなく，開発者自身にスタイルを遵守させる仕組みが提案されている．Lim らの提案する Style Avatar は，スタイル違反の有無をアバターの表情としてフィードバックするシステムである [20]．学生を用いた実験では 30%の違反が減少したことを報告している．Prause らは，開発者がプロジェクトのスタイルを守る動機づけとしてゲーミフィケーションを提案している [21]．アジャイルソフトウェア開発者を対象とした実験により，効果的にスタイルを守らせることができると報告している．これらと提案ツールの違いは，開発者が能動的にスタイルを修正するか，ツールが自動的にスタイルを修正するかという点にある．すなわち提案ツールは自動化が可能という点で利点があるが，これらの手法との併用も可能である．

スタイルの実情を調査する研究が多数行われている．Spinellis らは 43 年間にわたる Unix の開発履歴を対象に，長期開発におけるスタイル違反の変化について調査している [3]．本研究の予備調査で用いた SI 度は，Spinells の提案に基づいた尺度である．調査の結果として，ソースコード全体でのスタイルの不一致度は年々減少しており，開発コミュニティ内での合意形成の成功を明らかにしている．また，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ている [22]．Li らは，学生を対象にスタイルに関する意識を調査している [23]．多くの学生は規約を重要だと考えている一方，従わない傾向があると報告した．

スタイルには個人差があるという特徴を利用し，ソースコードやバイナリから開発者を特定する手法が提案されている．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 [24]．この

手法ではスタイルの分析は自動で行っており，1600 人以上の開発者において 92%以上の正確さで開発者を特定できている．Rosenblum らはスタイルの特徴がコンパイル後のバイナリにも表れてることを報告した [25]．またバイナリから開発者を特定する手法を提案している．Hayes と Offutt は，20 人の開発者のソースコードを目視で分析し，開発者によってスタイルに特徴があることを報告している [26]．また，熟練の開発者は学生と比較してスタイルに強いこだわりがあることを明らかにした．提案ツールは版管理上での利用を前提としているため，開発者を特定するという状況は発生しない．しかしながら，提案ツールはスタイルの自動抽出機能が内包されており，これらの研究の前処理に活用できる可能性がある．

スタイルの違いが与える可読性や理解容易性への影響を調査した研究が存在する．Hofmeister らは，識別子の長さが開発者の理解に影響するのかについて調査している [27]．結果として，識別子名を省略するよりも長い識別子名を用いたほうが理解が早いことを報告している．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 [28]．

版管理システムを拡張し，コードの本質的な情報とデザインを分離するというアイデアもいくつか存在する．畑らは Java のソースコードを解析し，より小さい構文要素毎にファイルを分割し記録する版管理システム Historage を提案している [29] ．Historage は履歴情報のマイニングの前処理として有用である一方で，エンドユーザとなる開発者が直接利用する仕組みではない．提案ツールは開発者が直接利用し利益を享受できるという点で異なる．

8 妥当性への脅威

予備調査 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．

提案ツールの評価 性能が異なるコンピュータで実験すると，コミットに要する時間は異なる．また対象としたソフトウェアは JUnit4 のみであり，他のソフトウェアを対象とすると異なる結果が得られる．

被験者は学生のみであり，共同開発の経験も少ない．被験者の能力に差がある可能性があるが，本研究科の修士を対象としており，全員同じカリキュラムでプログラミングを学習したため，能力は同じであると考えられる．

9 おわりに

本研究では，スタイル統一が容易な開発環境を実現するためのツールを提案した．提案するにあたり，予備調査として実際のソフトウェア開発においてスタイルの統一に関する実情を調査した．結果，SI_{all} 度の推移はソフトウェアによって異なり，調査したソフトウェアにおいては一般性はないといえる．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．

予備調査の結果より，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．そこで版管理システム上で動作するスタイル統一支援ツールを提案した．提案ツールは，開発者が版管理システムを用いて他の開発者とソースコードを交換する際，自動的にソースコードのスタイルを変換する．提案ツールに開発者好みのスタイルと，プロジェクトの共通スタイルを設定しておくことで，ソースコードを手元に取得する際は開発者好みのスタイルに，ソースコードを他の開発者にコミットする際はプロジェクトの共通スタイルに変換される．この変換作業は自動的に行われ開発者は意識する必要がない．また提案ツールは既存の版管理システムに容易に導入することができる．

提案ツールが実際の開発において有用であることを確認するため，性能評価実験と有用性評価実験を行った．性能評価実験では，提案ツールを導入した際のコミットに要する時間を計測した．提案ツールを導入した場合，導入していない場合と比較しコミットに要する時間は増加したものの，ほとんどのコミットは 5 秒以内に完了する．そのため，現実的な時間で動作することがいえる．また被験者実験では，作業時間や正確に作業できたかを評価した．結果，提案ツールを導入した場合作業時間が小さくなる傾向が見られたものの，統計的な差はなかった．作業における被験者のミスをスタイルミスと構文ミスに分け，それぞれのミスの数を用いて評価した．スタイルミスでは提案ツールによる差はなかった．一方，構文ミスは効果のあったタスクが 8 個中 6 個であり，提案ツールに対して肯定的な効果が確認できた．作業完了後被験者にインタビューを実施した．結果，全ての被験者はツールのないタスクにおいて作業を困難に感じていたことが分かった．定性的および定量的評価により，本ツールの有用性を確認できた．

今後の研究課題としては，提案ツールの有用性をより多くの被験者で確認することや，実際の開発での導入が挙げられる．またソースコードに含まれるスタイル以外の内容の分離，すなわちドキュメントやコメント，修正履歴などの情報を別のファイルとして保存・管理することなどが挙げられる．

謝辞

本研究を行うにあたり，常に励まして頂き，暖かく見守っていただきました 楠本 真二 教授に心より感謝申し上げます．

本研究に対して新しい視点からのご意見を頂き，研究室生活の中で相談に乗っていただきました 肥後 芳樹 准教授 に心より感謝申し上げます．

本研究を行うにあたり，日頃より熱心なご指導ご鞭撻を頂き，活発な議論に応じていただきました 柗本 真佑 助教に心より感謝申し上げます．

日々の研究室生活において，互いに切磋琢磨し，高め合い，ときに白熱した議論を繰り広げました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の 下仲 健斗 氏，同 中島 弘貴 氏，同 山田 悠斗 氏，同 山本 将弘 氏に心より感謝いたします．日々の研究室生活の中心となり，様々な役職で我々の生活を支えてくださりました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 1 年の 有馬 諒 氏，同 佐々木 美和 氏，同 谷門 照斗 氏，同 松尾 裕幸 氏，同 山田 涼太 氏に心より感謝いたします．

短い間でしたが，日々の議論を通じて共に成長を実感し合いました大阪大学基礎工学部情報科学研究科 4 年の 田中 紘都 氏，同 土居 真之 氏，同 松本 淳之介 氏，同 林 純一 氏に心より感謝いたします．

また，本研究に関して多くのご助言を頂くとともに，様々な面において親切なご助力，ご協力を頂きました楠本研究室の皆様に心より感謝いたします．

最後に，本研究に至るまでに，講義やセミナー等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます．

参考文献

[1] Paul W. Oman and Curtis R. Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM Annual Conference on Cooperation*, CSC '90, pages 244–250, 1990.

[2] Google. Google java style guide. Available online "`https://google.github.io/styleguide/javaguide.html`" (August 2017).

[3] Diomidis Spinellis, Panos Louridas, and Maria Kechagia. The evolution of c programming practices: A study of the unix operating system 1973–2015. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 748–759, 2016.

[4] M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.

[5] J. Barnard and A. Price. Managing code inspection information. *IEEE Software*, 11(2):59–69, 1994.

[6] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[7] L. W. Cannon et al. Recommended c style and coding standards. Available online "`http://sunland.gsfc.nasa.gov/info/cstyle.html`" (January 2006).

[8] The FreeBSD Project. style – kernel source file style guide, dec. 2015. freebsd manual pages: style(9). Available online "`https://www.freebsd.org/cgi/man.cgi?query=style&sektion=9`" (August 2017).

[9] Georgios Gousios and Diomidis Spinellis. Conducting quantitative software engineering studies with alitheia core. *Empirical Software Engineering*, 19(4):885–925, 2014.

[10] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1982.

[11] Trevor Misfeldt, Jim Shur, and Patrick Thompson. *Elements of C++ Style*. Cambridge University Press, New York, NY, USA, 2004.

[12] R. Stallman et al. Gnu coding standards. Available online "`http://www.gnu.org/prep/standards/`" (August 2017).

[13] Inc Sun Microsystems. Code conventions for the java programming language. Available online "`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`" (August 2017).

[14] Scott W. Ambler, Alan Vermeulen, and Greg Bumgardner. *The Elements of Java Style*. Cambridge University Press, New York, NY, USA, 1999.

[15] Checkstyle. Available online "`http://checkstyle.sourceforge.net/`" (August 2017).

[16] Code beautify. Available online "`https://codebeautify.org/`" (August 2017).

[17] Robert D. Cameron. An abstract pretty printer. *IEEE Software*, 5(6):61–67, 1988.

[18] Dereck C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[19] L.F. Rubin. Syntax-directed pretty printing-a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, 9(2):119–127, 1983.

[20] Jin-Su Lim, Jeong-Hoon Ji, Yun-Jung Lee, and Gyun Woo. Style avatar: A visualization system for teaching c coding style. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1210–1211, 2011.

[21] Christian R. Prause and Matthias Jarke. Gamification for enforcing coding conventions. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 649–660, 2015.

[22] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, 2013.

[23] Xiaosong Li and Christine Prasad. Effectively teaching coding standards in programming. In *Proceedings of the 6th Conference on Information Technology Education*, SIGITE '05, pages 239–244, 2005.

[24] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC '15, pages 255–270, 2015.

[25] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who wrote this code? identifying the authors of program binaries. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS '11, pages 172–189, 2011.

[26] Jane Huffman Hayes and Jeff Offutt. Recognizing authors: An examination of the consistent programmer hypothesis. *Software Testing, Verification and Reliability*, 20(4):329–356, 2010.

[27] Johannes Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. In *Proceedings of the 24th Conference on Software Analysis, Evolution and Reengineering*, SANER '17, pages 217–227, 2017.

[28] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[29] 畑秀明, 肥後芳樹, and 楠本真二. リポジトリマイニング可能なコードクローン版管理システムの提案. 情報処理学会論文誌, 54(2):894–902, 2013.