Bring Your Own Coding Style

Naoto Ogura*, Shinsuke Matsumoto*, Hideaki Hata† and Shinji Kusumoto*
*Graduate School of Information Science and Technology, Osaka University, Japan
{n-ogura, shinsuke, kusumoto}@ist.osaka-u.ac.jp
†Graduate School of Information Science, Nara Institute of Science and Technology, Japan
hata@is.naist.jp

***Abstract*—Coding style is a representation of source code, which does not affect the behavior of program execution. The choice of coding style is purely a matter of developer preference. Inconsistency of coding style not only decreased readability but also can cause frustration during programming. In this paper, we propose a novel tool, called StyleCoordinator, to solve both of the following problems, which would appear to contradict each other: ensuring a consistent coding style for all source codes managed in a repository and ensuring the ability of developers to use their own coding styles in a local environment. In order to validate the execution performance, we apply the proposed tool to an actual software repository.**

***Index Terms*—coding style, style feature, style inconsistency, software repository, Git**

I. Introduction

Coding style is a textual representation of source code, which does not affect the behavior of program execution. Examples of coding style include the type of indentation (i.e., tabs or spaces), the existence of a whitespace character around arithmetic operators, and naming conventions of variables and methods. Although coding style has no effect on the program behavior itself, it does have a significant influence on readability and maintainability for developers [1][2]. In a practical programming environment, a wide variety of styles must be defined in various locations of source code. In Eclipse, over 300 detailed items of coding style configuration, such as whether a space is inserted before an open parenthesis in an if statement, are provided.

The choice of coding style is purely a matter of developer preference [3], which has generally evolved from his/her programming experience. For example, the following choices are known to vary depending on the developer: whether a whitespace is inserted after `if`/`for` keywords and method names, whether an open brace "{" is inserted onto a new line or the same line. The general coding style changes according to the evolution of the programming environment. The style limitation of 80 characters per line originally comes from the physical limit of IBM's punch card [4]. However, the limit is being relaxed as monitor resolution improves. Currently, Google's Java coding convention[1] sets the limit to 100.

Unifying on a common coding style for all source codes is a significant activity in a co-development environment [5][6]. Inconsistency of coding style not only decreases readability, but also hinders developer concentration. A developer may feel frustration by being forced to use a style that he/she does not usually use. From the perspective of understanding software evolution, a history of style modifications usually becomes "noise" in a software repository. When a developer contributes both bug fix and style fix in a single commit, the resulting problem is referred to as the problem of "tangled code changes" [7].

Despite the fact that a style is defined as a coding convention for each project, it is impossible to force all of developers to follow the convention for the entire source code. An IDE and style formatter[2] can be used to unify coding style in a *local environment*. However, ensuring a consistent style within a project (i.e., *remote environment*) is still difficult because default style configurations are different for individual tools.

In this paper, we tackle the following challenges, which would appear to contradict each other:

- Ensuring a consistent coding style for all source codes managed in a repository.
- Developers can use their own coding styles (and own tools) in a local environment.

In order to achieve the above challenges, we propose a novel tool, called StyleCoordinator, which provides bidirectional coding style conversion between individual and project-defined styles. The tool provides a style-conversion feature and is executed by a standard Git client. The tool formats to a project-defined style when a developer commits source codes to a repository. This behavior solves the first challenge. The second challenge is solved by formatting to an individual style when a developer pulls from a repository. In addition, the tool supports automatic extraction of style preference from written source codes. This feature helps to reduce the cost of tool installation.

The main contributions of this paper are:

- We formulated specific cases of coding style as style features.
- We improved an existing extraction method of style features.
- We developed a tool for overcoming coding style frustration in a co-development software project.

The tool is still an early prototype in the sense that it only covers one of five categories of coding styles. However, we believe that our concept may be one of a peaceful solution to *holy wars* [8] about coding preferences.

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

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

II. Coding Style

A. Classification of Coding Styles

We define a representation of a source code as a *coding style*, which has no impact on program behavior. The coding style can be broadly classified as follows, The items are sorted in ascending order of influence on program compilation.

- Separation of tokens
  (e.g., space before open parenthesis in an if statement)
- Indentation
  (e.g., space or tab, number of indentation spaces)
- Name convention
  (e.g., camel case or snake case or kebab case)
- Declaration order
  (e.g., alphabetical order or public method first)
- Sugar syntax
  (e.g., if-else statement or ternary operator)

Most of the common IDEs provide customization of each category of coding style. In this paper, we focus only on the first category, separation of tokens, because this category includes a greater number of detailed items as compared to other categories. Eclipse defines 161 items for the category. All of the detailed items may be difficult to completely describe as coding conventions. We consider automated style conversion for the category to be practical for use in actual development.

B. Style Features

In this paper, we define a *style feature* as a set of information related to coding style. The style feature is composed of the following three attributes.

- Type (e.g., *space-before-open-paren-in-if-statement*)
- Location (e.g., line 30)
- Value (e.g., presence)

The above example means that a whitespace before open parenthesis in if statement in present in line 30. For the style category considered (separation of tokens), the value can only be presence or absence. The style feature exists in various locations of source code. Automatic inference of style configuration can be enabled by collecting style features from the developer's own written source codes.

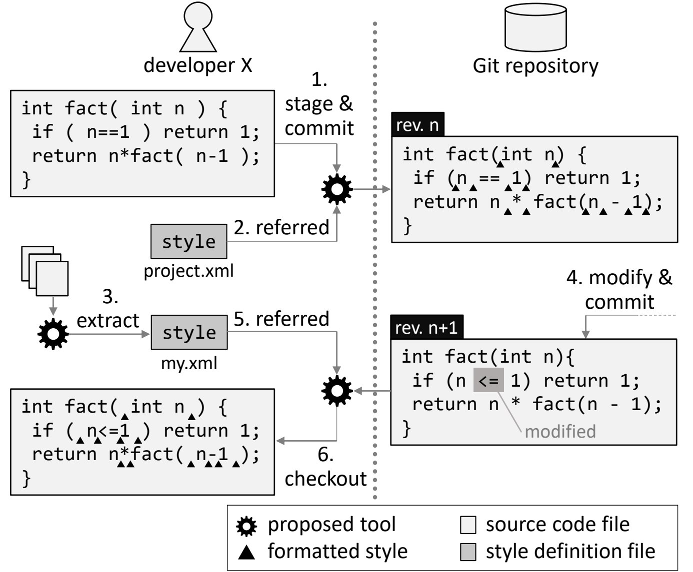
C. Style Inconsistency

Spinellis et al. [9] proposed a degree of *style inconsistency* (SI) to show how a source code includes inconsistencies of coding styles. Let a and b be values of a style feature (i.e., presence and absence), and let i be an ID of a style feature. Moreover, let a_i be the number of occurrences of a for style feature i. The degree of style inconsistency of style feature i, denoted by SI_i, is defined as

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

Let n be the total number of types of style feature ($1 \leq i \leq n$). Then, the overall degree of style inconsistency for all source codes, denoted by SI_{all}, is defined as

$$SI_{all} = \frac{\sum_i^n min(a_i, b_i)}{\sum_i^n a_i + b_i}$$




```
int fact( int n ) {
 if ( n==1 ) return 1;
 return n*fact( n-1 );
}
```

```
int fact(int n) {
 if (n == 1) return 1;
 return n * fact(n - 1);
}
```

```
int fact(int n){
 if (n <= 1) return 1;
 return n * fact(n - 1);
}
```

```
int fact( int n ) {
 if ( n<=1 ) return 1;
 return n*fact( n-1 );
}
```

Fig. 1: Overview of the StyleCoordinator

SI_i takes a value from 0 to 0.5. A larger value indicates that a higher degree of style inconsistency exists. For example, when a style type *space-before-open-brace-in-if-statement* occurs in ten instances, and only two of the ten instances include a space, and SI_i takes a value of 0.2 (2/10). Here, SI_i becomes maximum (0.5) when the number of occurrences of presence and absence are exactly the same.

III. StyleCoordinator

A. Overview

We propose a tool, StyleCoordinator, that solves the following seemingly contradictory challenges: ensuring a consistent coding style for all source codes managed in a repository and ensuring that developers can use their own coding styles in a local environment.

Figure 1 shows an overview of the processing flow of the tool. The tool is working on Git as software configuration management (SCM). Using a standard Git feature, called Git attribute, the tool executed on both stage and checkout operations in a local environment. Here, we introduce three use cases of using the tool.

- *Stage and commit*: Developer X uses our tool for the first time. When X stages and commits his/her written source code (step 1), the tool refers a style configuration file, which is provided as a common convention of the project (step 2). The source code is formatted based on the configuration and is stored to a repository. In this case, X's unusual coding style is formatted to the popular coding style. Thus, the first challenge, unifying the coding style in a repository, is solved.
- *Extract a style configuration file*: The tool helps to make a style configuration file by extracting style features from their written source code (step 3). This extraction is based on the concept of SI. In other words, a major

coding style within the developer's code is assumed to be his/her preferred style. The concrete method of the style extraction is described in the next Section.

- *Checkout*: Another developer modifies and commits the source code (step 4). When X checkouts from a repository, the tool refers the extracted configuration file (step 5) and transforms the source code to his/her own style (step 6). Thus, the second challenge is solved.

B. Style Feature Extraction

1) Problem of Existing Technique: In a co-development environment, the key to dealing with individual coding styles is detailed and accurate style feature extraction. Spinellis published a metrics measurement tool, called cqmetrics[3], in a study on SI's [9]. The tool supports style feature extraction as one of many functions of metrics measurement. However, the tool cannot follow syntactic information, because the extraction is based only on token analysis. Therefore, extractable types of style feature are coarse and incomplete.

Here, we illustrate the problem using a specific example. The black triangle (▲) represents a style feature that cannot be accurately extracted by cqmetrics.

```
if▲(▲n > 0▲) return▲;
  a1 c      c        b2
for▲(▲int i = 0▲; i < n▲; i▲++▲) {
   a1 c           b1       b1   c   c
  doSomething▲(▲i▲)▲;
             a2 c c b2
```

The tool does not distinguish between control flow keywords (e.g., `if`, `for`, and `while`). As such, both a_1's illustrated in the example are regarded have the same coding style as *space-before-open-paren-in-control-statement*. On the other hand, a space before an open parenthesis in method invocation (a_2) cannot be extracted. Every type of space before a semicolon (b_1 and b_2) is undistinguished because the tool also ignores the context of the program statement. In addition, there are many cases (c) in which a style feature is neglected.

2) Proposed Style Extraction Technique: In order to improve the existing extraction method, we propose a method by which to extract fine-grained style features based on syntax analysis. Figure 2 illustrates an overview of the processing flow. The input of the method is a set of Java source codes, and the output is a set of style features.

Step1: Search space characters that are candidates of a style feature. The tool filters some spaces that cannot be syntactically omitted (e.g., `public▲void`, and `int▲i`).

Step2: Generate an abstract syntax tree by conducting token analysis and syntax analysis.

Step3: Extract all style features from the candidates by checking the syntax tree.

Based on the above discussion, we implemented an improved extraction tool of style feature[4]. The tool is a sub-

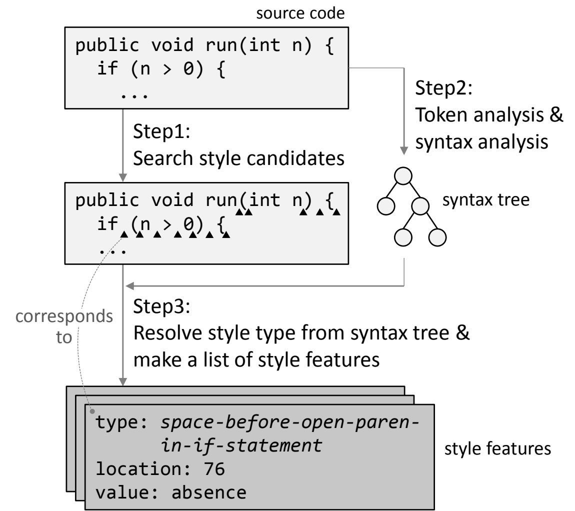


Fig. 2: Processing flow of the proposed style extraction

```
<profile kind="CodeFormtterProfile">
 <setting
    type="insert-space-before-open-paren-
          in-if-statement"
    value="insert"/>
 <setting
    type="insert-space-before-open-paren-
          in-for-statement"
    value="insert"/>
 <setting
    type="insert-space-before-open-paren-
          in-method-invocation"
    value="do not insert"/>
 ...
</profile>
```

Fig. 3: Example of a style configuration file

component of StyleCoordinator. Although the tool currently supports Java, the basic concept of extraction can be applied to many programming languages.

C. Style Configuration

A file for style configuration is shown in Figure 3. The file is written in XML. Each individual `setting` tag corresponds to a style feature. A space is inserted or omitted based on the `value` attribute for various locations corresponding to a style type specified in the `type` attribute. This file is compatible with a style configuration file used on Eclipse. Therefore, the configuration file of StyleCoordinator can be easily customized using Eclipse's style configuration GUI.

D. Implementation and Usage

StyleCoordinator is written in Java and published on our website[5]. The tool provides style transformation and style extraction. In order to cooperate with Git, we use a Git standard feature, called gitattributes, which enables customization

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

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

[5]http://sdl.ist.osaka-u.ac.jp/~n-ogura/2018_style-coordinator

of the pre-processor of the stage and checkout operations. In general, the feature is used to enforce the correct line endings policy.

Here, we describe the tool usage on a specific repository. First, a custom filter `scformat` must be defined in a gitattributes file. Pre-processing for all java files is enabled by this definition.

```
$ echo '*.java filter=scformat' >> \
   .git/info/gitattributes
```

Next, we need to define a preferred style file in a local environment. The definition is provided by automatic style extraction or Eclipse GUI. In this introduction, the preferred style is designated my.xml and the project-defined style is designated project.xml, which should be managed in a Git repository because the file is shared in a project.

Finally, we define the actual pre-processing behavior for the defined filter. The proposed tool is specified to execute at stage and checkout operations (i.e., filter.format.smudge and filter.format.clean).

```
$ git config filter.scformat.smudge \
   stylecoordinator my.xml

$ git config filter.scformat.clean \
   stylecoordinator project.xml
```

IV. Evaluation

A. Style Inconsistency in Actual Projects

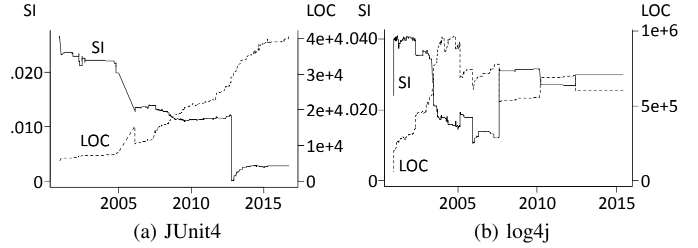
1) Purpose and Method: The purpose of the study is to survey style inconsistency during the evolution of Java projects, whereas Spinellis's study focused on the evolution of the C language.

We calculate changes of SI_{all} from a Java repository. The proposed tool is used for the SI calculation. The calculation is conducted for all source codes within the master and trunk branches. The subject projects are JUnit4 and log4j. An overview of the projects is shown in Table I. The development histories for over 15 years are stored in GitHub.

2) Result and Discussion: Figure 4 shows the changes of SI_{all} and lines of code (LOC). From Figure 4(a), JUnit4 has increasing LOC. Although SI_{all} was higher at the beginning of the project, the value has been decreasing over time. In particular, SI_{all} decreased drastically in 2012 by applying a style formatter for all source codes. A new coding convention was discussed at the time[6]. Even though the convention was introduced, SI_{all} has been increasing slightly with increasing LOC.

From Figure 4(b), log4j has a similar tendency as JUnit4 between 2000 and 2007 when the project was in early development. After that, SI varies around 0.03. In 2007, 2010 and 2012, SI and LOC were drastically changed because the project performed branch merging, feature addition, and feature deletion.

[6]https://github.com/junit-team/junit4/issues/426



(a) JUnit4 (b) log4j

Fig. 4: Change of SI_{all} and LOC for each project

In summary, style inconsistency tends to occur in early development and to be affected by changing LOC. With the software evolution, the inconsistency cannot be solved even if a style formatter is applied. Some developers may get frustrated by the inconsistency. We conclude that a tool to support a consistent coding style may be helpful in a co-development environment.

B. Performance Evaluation

1) Purpose and Method: Using the proposed tool, the execution performance of Git operation is expected to decrease because of token and syntax analysis. In this experiment, we confirm whether the performance decrease is practical.

All stage and commit operations that were executed in actual software are reproduced. We measured and compared the execution times for every operation with and without the proposed tool. The subject project is JUnit4, which includes 2,115 revisions during years 2000 through 2015.

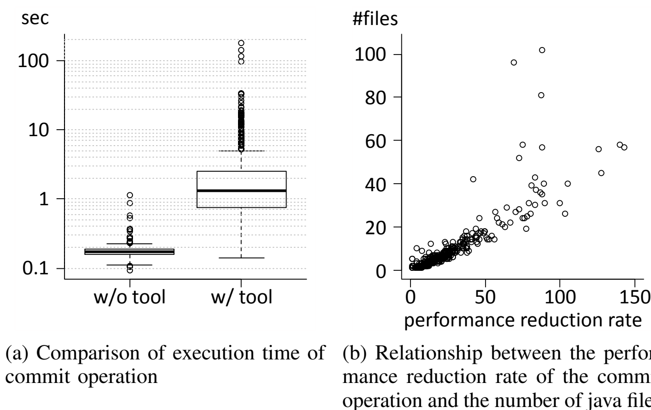
2) Result: Figure 5(a) shows the performances with and without the proposed tool. The distribution of the execution time is shown as a box plot. The y-axis shows the execution time in log scale.

Most original commits were accomplished within a single second. In contrast, using the proposed tool, the execution time was increased approximately 17-fold. However, we found that 75% of commits were accomplished within 2.5 seconds. In the worst case, the execution time increased from 1.1 seconds to 178.4 seconds, which was the longest time with and without the proposed tool. In this case, 305 files were committed at the same time in order to apply a new coding style in 2012. The drastic style change is also shown in Figure 4(a).

Figure 5(b) illustrates the relationship between the rate of performance reduction and the total number of java files. The x-axis indicates the rate of performance reduction obtained using our tool, and the y-axis indicates the number of files. Note that a few cases, in which the number of java files exceeds 100, are omitted. We can confirm that the performance is decreased as the number of java files increases.

TABLE I: Summary of subject projects

project name	years	# revisions	# developers	LOC
JUnit4	15	2,115	159	39,722
log4j	15	3,275	21	59,780



(a) Comparison of execution time of commit operation

(b) Relationship between the performance reduction rate of the commit operation and the number of java files

Fig. 5: Results of the performance evaluation experiment

3) Discussion: In summary, the most commits were accomplished within 5 seconds when the proposed tool was used as a pre-processor. In a practical situation, the percentage of the commit time during development is extremely small because the commit is executed after a certain task is completed. We believe that the performance decrease by the proposed tool may be acceptable.

We also found that the execution performance decreases by committing multiple files simultaneously. The reason for this is that the construction of the syntax tree requires more time as the number of total program statements increases. In recent software development, it is recommended that all commits should be kept smaller [10]. This practice is known to be key to continuous integration. Therefore, we believe that the influence of the performance reduction may be smaller with the spread of agile practices.

V. Related Work

A number of software projects have published coding guidelines[7,8,9] that include some specific rules of coding style. Of course, there are many differences between these styles. This means that coding style is not only a preference of the project but is also a non-trivial problem that cannot be neglected in a co-development environment.

Some researchers have conducted a survey of coding styles in practical environments. Spinellis et al. [9] have studied long-term changes in style inconsistency in Unix for 43 years. They concluded that the project successfully achieved a consensus of coding style because the inconsistency decreased year by year. Bacchelli and Bird [11] reported that code improvements, such as checking that source code follows code convention, are important motivation for code review. Li et al. [12] studied students' opinions on coding styles in programming courses. From these studies, it is difficult to follow the coding convention for all developers.

[7]https://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/

[8]https://www.gnu.org/prep/standards/

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

Various style formatters have been published[10] and many popular IDEs also include these formatters by default. Moreover, in the academic field, a number of style formatters have been proposed [3][13][14][15]. These tools have some features in common with the proposed tool. However, the tools can only work in a stand-alone or local environment. The proposed tool can ensure consistent style in both local and remote environments by execution with Git operation.

VI. Conclusion

In this paper, we proposed a tool for use in development environments to easily maintain style consistency. The tool includes an improved algorithm of style extraction. We conducted an experiment to evaluate performance of the tool. From the evaluation, the performance decrease by the tool may be acceptable.

In the future, we intend to extend to cover other categories of coding styles such as indentation and name convention. To deal with these categories, we need to consider further style extraction and inconsistency criteria. In order to confirm the usefulness of the tool, we are going to conduct qualitative evaluation with industrial practitioners.

Acknowledgment

This study was supported by JSPS/MEXT KAKENHI Grant Numbers JP25220003 and JP26730155.

References

[1] P. W. Oman and C. R. Cook. A taxonomy for programming style. In *Proc. Annual Conf. Cooperation*, pages 244–250, 1990.

[2] A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *Proc. Working Conf. Mining Software Repositories*, pages 146–156, 2015.

[3] T. Parr and J. Vinju. Towards a universal code formatter through machine learning. In *Proc. Int'l Conf. Software Language Engineering*, pages 137–151, 2016.

[4] IBM Archives: 1928, January 2018.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proc. Int'l Symp. Foundations of Software Engineering*, pages 281–293, 2014.

[6] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *Proc. Int'l Conf. Software Maintenance*, pages 277–286, 2008.

[7] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proc. Working Conf. Mining Software Repositories*, pages 121–130, 2013.

[8] D. Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[9] D. Spinellis, P. Louridas, and M. Kechagia. The evolution of c programming practices: A study of the unix operating system 1973–2015. In *Proc. Int'l Conf. Software Engineering*, pages 748–759, 2016.

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

[11] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. Int'l Conf. Software Engineering*, pages 712–721, 2013.

[12] X. Li and C. Prasad. Effectively teaching coding standards in programming. In *Proc. Conf. Information Technology Education*, pages 239–244, 2005.

[13] D. C. Oppen. Prettyprinting. *ACM Trans. Programming Languages and Systems*, 2(4):465–483, 1980.

[14] L. F. Rubin. Syntax-directed pretty printing - a first step towards a syntax-directed editor. *IEEE Trans. Software Engineering*, 9(2), 1983.

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

[10]https://codebeautify.org/javaviewer