Measuring the Stylistic Inconsistency in Software Projects using Hierarchical Agglomerative Clustering

Qing Mi, Jacky Keung
Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong
Qing.Mi@my.cityu.edu.hk,
Jacky.Keung@cityu.edu.hk

Yang Yu
Search Engine Department
Beijing Sogou Technology Development Co., Ltd
Beijing, China
yuyangtx062@sogou-inc.com

ABSTRACT

Background: Although many software engineering methodologies and guidelines are provided, it is common that developers apply their very own programming styles to the source code being produced. These individually preferred programming styles are more comprehensive for themselves, but may well conflict with each other. Thus, the problem of stylistic inconsistency is inevitable during the software development process involving multiple developers, the result is undesirable and that will significantly degrade program readability and maintainability. ***Aims:*** Given limited understanding in this regard, we perform an empirical analysis for the purpose of quantitatively measuring the inconsistency degree of programming style within a software project team. ***Method:*** We first propose stylistic fingerprints, which are represented as a set of attribute-counting-metrics, in an attempt to characterize different programming styles. Then we adopt the hierarchical agglomerative clustering (HAC) technique to quantitatively measuring the proximity of programming style based on six C/C++ open source projects chosen from different application domains. ***Results:*** The empirical results demonstrate the feasibility and validity of our fingerprinting methodology. Moreover, the proposed clustering procedure utilizing HAC algorithm with dendrograms is capable of effectively illustrating the inconsistency degree of programming style among source files, which is significant for future research. ***Conclusions:*** This study proposed an effective and efficient approach for analyzing programming style inconsistency, supported by a sound theoretical basis for dealing with such a problem. Ultimately improving program readability and therefore reduce the maintenance overhead for software projects.

CCS Concepts

•**Software and its engineering** → **Software product lines;** *Automated static analysis;* Empirical software validation;

PROMISE 2016, September 09 2016, Ciudad Real, Spain

DOI: http://dx.doi.org/10.1145/2972958.2972963

Keywords

programming style, stylistic inconsistency, hierarchical agglomerative clustering, empirical software engineering

1. INTRODUCTION

Programming Style can be viewed as a kind of personal habits when writing the source code, which reflects individual's preferences in terms of physical layout (e.g., alignment and indentation), data structure, algorithm, etc. It tends to become ingrained, which is highly identifiable but difficult to quantify and measure.

At present, the majority of software development is conducted by project teams, which range in size from two to over hundreds of developers [21]. Generally, each developer has his/her own programming style due to different experiences and personalities, yet these individually preferred programming styles could well contradict each other. Such divergences are quite likely to degrade readability and maintainability of the entire code base during the process of team development. We refer to this problem as *Stylistic Inconsistency*. Although quantities of advanced methodologies and tools are proposed to prevent the problem [2, 4, 8], it is still highly probable for one or more of the following reasons:

- The development team lacks a clear coding standard. Consequently, a variety of programming styles is imposed on the source code being produced.
- A mature and widely accepted coding standard is provided before the collaboration begins, whereas some developers are unwilling or unable to conform to it.
- Another typical scenario is code reuse, such as the adoption of third-party libraries, which also exacerbates the existing problem.

A considerable number of previous studies have focused in the area of programming style. For instance, investigating the impact of stylistic variants on program comprehension [7, 14, 24], or coding style-based plagiarism detection [3]. However, there is little focus on the measurement of stylistic inconsistency. In order to assist practitioners to better understand the inconsistency degree of programming styles within software projects, we aim to propose an effective and efficient approach in this study. More specifically, we will address the following research questions:

- **RQ1.** How to characterize the programming style quantitatively?

- **RQ2.** How to measure the stylistic inconsistency in software projects?

The rest of this paper is structured as follows. Section 2 proposes stylistic fingerprints to characterize different programming styles. An empirical experiment is then performed to evaluate such fingerprinting methodology. In Section 3, we measure the stylistic inconsistency in software projects making use of hierarchical agglomerative clustering. Section 4 discusses threats to validity. Section 5 presents the related work. We provide conclusions and future work directions in Section 6.

2. STYLISTIC FINGERPRINTING

Programming style is an intuitive concept, which is difficult to define and measure. Although the stylistic features of source code are less flexible as compared to, for instance, handwriting or literary writing, due to limited keywords and structured format, there is still enough room for developers to establish their own distinctive styles.

Inspired by the work of Krsul et al. [12] and Avgustinov et al. [5], we adopt the concept of fingerprints to characterize different programming styles (RQ1). A collection of potentially useful metrics is presented in this section as components to construct stylistic fingerprints as the basis for further analysis.

2.1 Target Programming Language

Theoretically, it is possible to compare different programming styles across different development languages. Nonetheless, the majority of stylistic metrics (e.g., metrics demonstrated in Table 1) are highly correlated to specific languages, which is unexpected before. For instance, the metrics for measuring the preferred form of identifiers (e.g., camelCase and under_score), which seems broadly applicable to most any programming language, but actually not. Some separators, such as dot sign in R and dollar sign in PHP, are not allowed in any other popular languages. In addition, there is generally only one development language adopted in a software project. Therefore, it is pointless and meaningless to make comparisons among different languages, which is the major reason that we solely concentrate on one single development language in this study.

The primary reasons for choosing C/C++ as the target programming language are as follows: 1) According to the TIOBE Index[1], C/C++ is one of the main development languages, which is widely used in both industry and academia; 2) The availability of a substantial number of C/C++ projects hosted on Github[2]. Github is a web-based Git repository hosting service that having 15 million users collaborating across 38 million repositories[3].

In summary, this study is restricted to stylistic concerns of C/C++ source files (with the suffix ".c", ".cc", ".cpp" or ".cxx").

2.2 A Taxonomy for Stylistic Metrics

Considering the large number of stylistic metrics involved in this study, we first propose a classification methodology based on the work from Oman et al. [16, 18] and Krsul et al. [12].

[1]http://www.tiobe.com/index.php/tiobe_index
[2]https://github.com
[3]As of June 2016.

```
1  void BubbleSort(int array[], int n) {
2      for (int i = 1; i < n; i++) {
3          for (int j = 0; j < n - i; j++) {
4              if (array[j] > array[j+1]) {
5                  int temp = array[j+1];
6                  array[j+1] = array[j];
7                  array[j] = temp;
8              }
9          }
10     }
11 }
```

(a)

```
1  void
2  BubbleSort (int array[], int n)
3  {
4    for (int i = 1; i < n; i++)
5      {
6        for (int j = 0; j < n - i; j++)
7          {
8            if (array[j] > array[j + 1])
9              {
10               int temp = array[j + 1];
11               array[j + 1] = array[j];
12               array[j] = temp;
13             }
14         }
15     }
16 }
```

(b)

```
1  // bubble sort algorithm
2  void BubbleSort(int array[], int size) {
3      for (int pass = 1; pass < size; pass++) {
4          for (int index = 0; index < size - pass; index++) {
5              // swapping elements if necessary
6              if (array[index] > array[index+1]) {
7                  // holding variable for swapping
8                  int temp = array[index+1];
9                  array[index+1] = array[index];
10                 array[index] = temp;
11             }  // end of if
12         }  // end of for loop
13     }  // end of for loop
14 }  // end of BubbleSort
```

(c)

```
1  void BubbleSort(int array[], int n) {
2      int i = 1, j = 0;
3      while (i < n) {
4          while (j < n - i) {
5              if (array[j] > array[j+1]) {
6                  int temp = array[j+1];
7                  array[j+1] = array[j];
8                  array[j] = temp;
9              }
10             j++;
11         }
12         i++;
13     }
```

(d)

Figure 1: Style Variants on Bubble Sort Algorithm

- **Programming Format Metrics (PFM)**

  Metrics fall into this category are mainly related to the format (e.g., indentation, alignment and spacing) of the source code. It is necessary for developers to maintain a clear and easy-to-understand physical layout making use of white-space characters (e.g., space, tab, linefeed and carriage return). However, the way the white-space characters being used in the source code is quite different. For instance, program in Figure 1a conforms to Google Style Guides[4], whereas program in Figure 1b follows GNU Coding Standards[5]. Both of the code snippets are lexically identical but significantly different in format. Note that all metrics in this category could be easily changed by the use of formatting tools (e.g., Pretty Printer[6] and Artistic Style[7]). Although fragile to some extent, the format is still a distinctive characteristic in terms of measuring stylistic inconsistency.

- **Programming Readability Metrics (PRM)**

  This kind of metrics is used to represent the degree of program readability, which has no impact on functionality and efficiency of the source code. In this category, we include such metrics as the ones that measuring naming preference, the proportion of comments and line length. These metrics still closely pertaining to the layout of the source code, but are relatively difficult to change by automatic formatting tools. Figure 1c provides a program which implements bubble sort algorithm exactly the same way as the program displayed in Figure 1a, but looks quite different, which serves as a simple example for readability variations.

- **Programming Language Metrics (PLM)**

  Slightly different from the classification as given in Krsul's study [12], this category is designed to involve metrics that are highly correlated with the way developers prefer to implement a particular functionality. For instance, metrics reflecting one's preference of either for, while or do-while loops. As shown in Figure 1a and Figure 1d, both of the code snippets realize the bubble sort algorithm but in completely different ways. It should be noticed that metrics fall into this category may vary considerably when applied to other programming languages.

In conclusion, PFM purely correspond to white-space characters that represent horizontal or vertical space in typography. PRM would not be affected by the use of formatting tools on the source code being analyzed. Both of above-mentioned metrics pertain only to physical layout, which is considered highly influential in viewer's first impression of the program. On the other hand, PLM characterize the way the implementation of certain functionality.

Note that our taxonomy is proposed to provide a clear framework to better assist the further analysis. We do not claim it is flawless. There may be some overlap among the three categories or disagreements about the placement of specific metrics.

[4]https://github.com/google/styleguide
[5]http://www.gnu.org/prep/standards
[6]http://prettyprinter.de
[7]http://astyle.sourceforge.net

2.3 Stylistic Metrics Adopted in Our Study

A wide variety of previous studies (including but not limited to [6, 10, 12, 18, 20]) influence our choice of stylistic metrics, among which the metrics proposed by Krsul et al. [12] for C programs and Ding et al. [10] for Java programs are the primary ones. Nonetheless, metrics used by those work may be ambiguous, contradictory, even outdated. By reference to those studies, we refine and present a total of 60 metrics in Table 1 as components to construct stylistic fingerprints for characterizing different programming styles.

To extract these metrics, we develop a Python program (available on-line[8]) on the basis of cpplint, which is an open source lint-like tool released by Google. CSV-formatted datasets are automatically generated after scanning all the source files involved in the software projects.

2.4 Fingerprinting Evaluation

In this section, an empirical experiment is performed to determine whether our fingerprinting methodology is capable of effectively representing different programming styles.

2.4.1 Evaluation Methodology

Stylistic fingerprint, which is proposed in the previous section, serves as the basis for representing the programming style for an individual or a single document. Before further analysis, an empirical experiment should be conducted to evaluate the feasibility and validity of such methodology. More specifically, we intend to make judgment in terms of:

- Whether the stylistic fingerprints from the same developer stay stable.

  It is noticeable that even the same developer varies his/her own programming style as time goes on. Nevertheless, such style is expected to remain relatively steady under certain conditions (e.g., same programming language and short time-span). Thus, the feature vector should not vary widely for the source code from the same developer.

- Whether the stylistic fingerprints from different developers distinctively different from each other.

  The likelihood of finding two programmers who share exactly the same characteristics of programming style are very low [12]. Thus, if our stylistic fingerprints fail to reflect such distinction, which would mean that the metrics we select before are neither sufficient nor characteristic for further analysis.

To sum up, the effectiveness of our fingerprinting methodology could be verified if the fingerprints are most similar to those calculated from the same developer and dissimilar to those from others. The design of our empirical experiment is inspired by the work of Avgustinov et al. [5], the detailed procedure of which is listed as follows:

1. Collect C/C++ source files from Github.

   We give our preference to small-scale projects with only one developer involved. In addition, source files come from large-scale projects but could be explicitly attributed to one single developer are also acceptable. For each project, header files and source files that no longer than 100 lines are excluded.

[8]https://github.com/robertyuyang/StylisticFingerprinting

Table 1: Stylistic Metrics Adopted in Our Study

Dimension		Metric	Description
Programming Format Metrics (PFM)	Physical Layout of Curly Brackets	PFM01	Percentage of open curly brackets ({) that are alone in a line.
		PFM02	Percentage of open curly brackets ({) that are the first character in a line.
		PFM03	Percentage of open curly brackets ({) that are the last character in a line.
		PFM04	Percentage of open curly brackets ({) that are in the middle of a line.
		PFM05	Percentage of close curly brackets (}) that are alone in a line.
		PFM06	Percentage of close curly brackets (}) that are the first character in a line.
		PFM07	Percentage of close curly brackets (}) that are the last character in a line.
		PFM08	Percentage of close curly brackets (}) that are in the middle of a line.
		PFM09	Percentage of functions with blank line(s) before close curly brackets (}).
	Alignment Style	PFM10	Average white spaces or horizontal tabs to the left side of assignment operators.
		PFM11	Average white spaces or horizontal tabs to the right side of assignment operators.
		PFM12	Average white spaces or horizontal tabs after // or /* symbol.
	Indentation Style	PFM13	Percentage of physical SLOC[a] without indentation.
		PFM14	Percentage of physical SLOC with white spaces for indentation.
		PFM15	Percentage of physical SLOC with horizontal tabs for indentation.
Programming Readability Metrics (PRM)	Comment Style	PRM01	Whether the Copyright message appears at the top of the file.
		PRM02	The ratio of the number of blank lines to the number of non-blank lines.
		PRM03	The ratio of the number of comment lines to the number of non-commentary lines (i.e., pure code lines).
		PRM04	The ratio of the number of pure comment lines to the number of comment lines.
		PRM05	The ratio of the number of code lines with inline comments to the number of comment lines.
		PRM06	The ratio of the number of single-line comments (// symbol) to the number of physical SLOC.
		PRM07	The ratio of the number of multi-line comments (/* and */ pair of symbols) to the number of physical SLOC.
		PRM08	The ratio of the number of TODO comments to the number of non-blank lines.
	Naming Preference	PRM09	Percentage of uppercase characters for function name.
		PRM10	Percentage of lowercase characters for function name.
		PRM11	Percentage of number characters for function name.
		PRM12	Percentage of underscores characters for function name.
		PRM13	Percentage of uppercase characters for variable name.
		PRM14	Percentage of lowercase characters for variable name.
		PRM15	Percentage of number characters for variable name.
		PRM16	Percentage of underscores characters for variable name.
		PRM17	Average function name length.
		PRM18	Average variable name length.
	Line Length	PRM19	Average physical SLOC per function.
		PRM20	Average line length of physical SLOC.
Programming Language Metrics (PLM)	Preference for Looping Structure	PLM01	Percentage of for in total of for, while and do-while loops.
		PLM02	Percentage of while in total of for, while and do-while loops.
		PLM03	Percentage of do-while in total of for, while and do-while loops.
	Preference for Selection Structure	PLM04	Percentage of if-else in total of if-else and switch-case branches.
		PLM05	Percentage of switch-case in total of if-else and switch-case branches.
	Frequency of Selected Keywords in C[b] and C++[c] Programming Languages	PLM06	The ratio of the number of keyword struct to the number of physical SLOC.
		PLM07	The ratio of the number of keyword typedef to the number of physical SLOC.
		PLM08	The ratio of the number of keyword namespace to the number of physical SLOC.
		PLM09	The ratio of the number of keyword using to the number of physical SLOC.
		PLM10	The ratio of the number of keyword goto to the number of physical SLOC.
		PLM11	The ratio of the number of keyword const to the number of physical SLOC.
		PLM12	The ratio of the number of keyword template[d] to the number of physical SLOC.
		PLM13	The ratio of the number of keyword operator[d] to the number of physical SLOC.
		PLM14	The ratio of the number of keyword try[d] to the number of physical SLOC.
		PLM15	The ratio of the number of keyword catch[d] to the number of physical SLOC.

Table 1 Continued

Dimension		Metric	Description
Programming Language Metrics (PLM)	Frequency of Selected Keywords in C[b] and C++[c] Programming Languages	PLM16	The ratio of the number of keyword throw[d] to the number of physical SLOC.
		PLM17	The ratio of the number of keyword explicit[d] to the number of physical SLOC.
		PLM18	The ratio of the number of keyword mutable[d] to the number of physical SLOC.
		PLM19	The ratio of the number of keyword volatile[d] to the number of physical SLOC.
		PLM20	The ratio of the number of casting operators[de] to the number of physical SLOC.
		PLM21	The ratio of the number of keyword auto[df] to the number of physical SLOC.
		PLM22	The ratio of the number of keyword constexpr[df] to the number of physical SLOC.
		PLM23	The ratio of the number of keyword nullptr[df] to the number of physical SLOC.
		PLM24	The ratio of the number of keyword static_assert[df] to the number of physical SLOC.
		PLM25	Whether the Rvalue reference[df] appears in the parameter passing.

[a] Physical SLOC (source lines of code) [15] is a count of lines in the text of source code excluding blank and pure comment lines.
[b] http://en.cppreference.com/w/c/keyword
[c] http://en.cppreference.com/w/cpp/keyword
[d] Only exists in C++ but not C programming language.
[e] The casting operators included in this study are: dynamic_cast, static_cast, const_cast and reinterpret_cast.
[f] Since C++11.

2. Compute stylistic fingerprints for each source file.

   Making use of the tool, we extract stylistic metrics proposed in the previous section for each source file and generate the corresponding CSV-formatted dataset.

3. Measure the (dis)similarity among stylistic fingerprints.

   Given that stylistic fingerprints are actually feature vectors, we use Euclidean distance (Equation 1) as the proximity measure between each pair of fingerprints.

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{1}$$

   Where $\mathbf{x}$ and $\mathbf{y}$ are two vectors of length n.

4. Count the ranking result.

   Based on the results of distance calculation, we count how many fingerprints are ranked as being closest to themselves (i.e., closest to one of the stylistic fingerprints derived from the same developer), second closest to themselves, and so on. If our fingerprinting methodology is not characteristic enough, the ranking outcome should be mostly random.

2.4.2 Evaluation Result

Altogether 408 C/C++ source files of 20 developers are randomly collected from Github. Following the aforementioned experimental procedure, we obtain the bar graph (Figure 2) that demonstrates how often a developer's programming style is ranked as being closest to themselves, second closest to themselves, etc. As an example, the leftmost bar represents the number of stylistic fingerprints that being ranked as most similar to themselves, which is 101.

If our fingerprinting methodology is unable to represent different programming styles (i.e., if fingerprints for the same developer vary widely, or all developers tend to have similar fingerprints), we would expect a random ranking outcome. However, the histogram presented in Figure 2 indicates a positive answer. A clear pattern (i.e., downward trend) can be observed, which is a powerful support for the effectiveness of our fingerprinting methodology.

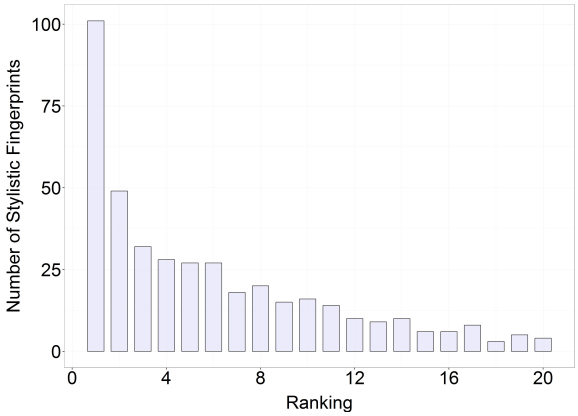


Figure 2: Ranking Result (The Number of Stylistic Fingerprints Being Ranked as First Closest, Second Closest, etc. to Themselves)

3. MEASURING THE STYLISTIC INCONSISTENCY IN SOFTWARE PROJECTS

Results from previous studies have shown that up to 80% of software life-cycle costs are due to maintenance, of which about half is caused by comprehension [1]. Generally, a clear and consistent programming style can significantly improve readability and maintainability of the entire code base. Nonetheless, despite all attempts to maintain it, the programming style during the collaborative process tends to become diversity. Thus, on the basis of stylistic fingerprints proposed before, we provide an effective and efficient approach in this section, which is capable of quantitatively measuring the inconsistency degree of programming style within software projects (RQ2).

Table 2: Projects Used in the Experiment

Project	# of Developers	# of Source Files	Eligible Files[a] #	Eligible Files[a] %	Size (KLOC)	Description
Muduo	27	87	33	37.9310%	6	A C++ non-blocking network library for multi-threaded server in Linux.
LevelDB	14	71	44	61.9718%	16	A fast key-value storage library that provides an ordered mapping.
QXmpp	16	69	55	79.7101%	24	A cross-platform C++ XMPP client and server library.
Bitcoin Core	376	286	195	68.1818%	80	An open source software which enables the use of Bitcoin (an experimental digital currency).
HHVM	416	1178	823	69.8642%	594	An open-source virtual machine designed to execute programs written in Hack and PHP.
CoreCLR	263	2024	1059	52.3221%	1299	The virtual machine component of Microsoft's .NET framework, which manages the execution of .NET programs.

[a]Source files with more than 100 non-blank lines.

3.1 Experimental Setup

3.1.1 Studied Systems

Our experiment is based on six well-established and popular C/C++ open source projects[9], i.e., Muduo[10], LevelDB[11], QXmpp[12], Bitcoin Core[13], HHVM[14] and CoreCLR[15], which are detailed in Table 2. These projects are chosen to involve varying application domains to keep the results generic and widely applicable. Note that only source files with the suffix ".c", ".cc", ".cpp" or ".cxx" are involved in this study.

3.1.2 Level of Granularity

We initially consider to compute stylistic fingerprints for all developers but judged meaningless. For the reason that each source file is usually maintained by more than one developer during team development, it is unreasonable to attribute the file to one arbitrary person. Most importantly, even though the contributors information is publicly available, partitioning source files according to the author is not that significant and cost-effective for our experiment.

Next, we evaluate different levels of granularity, such as component/package, file/class or module/method. Theoretically, the adoption of fine-grained metrics can help to precisely pinpoint the abnormal code block. Nevertheless, it is difficult to obtain meaningful and reliable fingerprints (e.g., too many zeros) with the low data volume. Therefore, we determine to adopt file-level metrics in this study. By the same token, only source files at least 100 lines (exclusive of blank lines) are involved in the experiment.

For each project, our analysis is conducted from three dimensions in accordance with the taxonomy proposed in Section 2.2, namely PFM, PRM and PLM.

3.1.3 Methodology

Cluster analysis is a multivariate method to classify data samples into a set of groups (i.e., clusters) that are relatively homogeneous within themselves and heterogeneous among each other, which is a suitable and powerful technique for addressing RQ2. The rest of the process of our empirical experiment after the calculation of stylistic fingerprints for each source file is detailed as follows.

Step 1: Data Preprocessing.

If variables are measured on different scales, variables with large values would dominate any distance measures. Thus, before further analysis, we scale (i.e., standardize) all the metrics so that they are comparable and contribute equally to the proximity between cases. Our dataset is transformed according to Equation 2.

$$\frac{x_i - center(x)}{scale(x)} \tag{2}$$

Where $center(x)$ is mean and $scale(x)$ is standard deviation of x values.

Step 2: Correlation Analysis.

Making use of the *varclus* function in the *Hmisc* R package, a variable clustering analysis is performed to detect redundancy and collinearity between variables, the result is demonstrated as a hierarchical overview (e.g., Figure 3a and Figure 3b). For the highly correlated variables, we reserve only one from each pair.

Step 3: Clustering Tendency Assessment.

Before applying any method on our dataset, it is necessary to assess the feasibility of clustering analysis. Even if there are no inherent clusters exist in the dataset, clustering methods could blindly partition data and return the corresponding results. Clustering tendency assessment is used to prevent such issue by determining whether a given dataset contains meaningful clusters.

In our experiment, we calculate Hopkins statistic [11] by the *hopkins* function in the *clustertend* R package to evaluate the clustering tendency (i.e., clusterability). Hopkins statistic is known to be a fair estimator of randomness in a dataset by measuring the probability that the dataset is generated by a uniform data distribution, which is defined in Equation 3 as follow:

$$H = \frac{\sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i} \tag{3}$$

Where x_i is the mean nearest neighbor distances in the real dataset and y_i is the mean nearest neighbor distance in the simulated (random) dataset.

[9]The source code being analyzed is extracted from Github in February 2016.
[10]https://github.com/chenshuo/muduo
[11]https://github.com/google/leveldb
[12]https://github.com/qxmpp-project/qxmpp
[13]https://github.com/bitcoin/bitcoin
[14]https://github.com/facebook/hhvm
[15]https://github.com/dotnet/coreclr

Step 4: Determining the Optimal Number of Clusters.

After assessing whether the dataset is clusterable, we then try to identify how many programming styles are there, i.e., to determine the appropriate number of clusters.

In the literature, a wide variety of indices has been proposed to find the optimal number of clusters. Nonetheless, there is no definitive answer. After close consideration and evaluation, we adopt Gap statistic [22] in the experiment, which is one of the most popular techniques outperforming many other methods. The idea of Gap statistic is to compare the within-cluster dispersion to its expectation under an appropriate null reference distribution. It is designed to be applicable to virtually any clustering method.

Step 5: Hierarchical Agglomerative Clustering (HAC).

The two major clustering techniques are partitional and hierarchical methods. A partitional clustering (e.g., k-means) is simply a division of data objects into disjoint subsets. While carrying out a hierarchical clustering, we can obtain a set of nested clusters in the form of a tree, which is better suited to address our research questions. Thus, we adopt agglomerative approach (bottom-up) in this study, which starts with every point being a cluster, and at each step merge with the closest pair of clusters. The procedure can be represented on a tree diagram known as a dendrogram, which displays the order in which the clusters are merged and the distance between clusters at the time of joining.

A number of agglomeration methods (single, complete, Ward, etc.) are available for determining which clusters should be merged at successive steps. Different methods may well result in different solutions. Although quite sensitive to outliers, Ward's method [23], which is one of the most popular methods, is selected as our agglomeration criterion. It first computes the sum of squared distances within clusters, and then aggregate clusters with the minimum increase in the overall sum of squares.

3.2 Experimental Result

According to the experimental procedure, we first scale all numeric metrics in the dataset to have standard deviation 1 and mean 0. After that, we conduct correlation analysis for all projects. As a typical instance, Figure 3a presents the hierarchical overview before variable deduction for project Bitcoin Core in PFM dimension. For the pairs of variables with correlations larger than 0.7, namely PFM1 vs. PFM3, PFM13 vs. PFM14, we keep only one in the model. The corresponding outcome is shown in Figure 3b. Table 3 demonstrates the overall result for variable deduction, which is IDs of excluded metrics in line with those from Table 1.

Table 3: Variable Deduction by Correlation Analysis

Project	PFM	PRM	PLM
Muduo	05, 13	01, 03	15
LevelDB	03, 13	03, 04, 09	-
QXmpp	01, 13	04	07
Bitcoin Core	01, 13	03, 04	-
HHVM	13	04	04
CoreCLR	13	04	04

Next, we assess whether our dataset is clusterable. If it is uniformly distributed, the Hopkins statistic, which is denoted by H, will be around 0.5[16]. According to the documentation of *factoextra* R package, if the value of H is close to 0, the dataset is significantly clusterable. As shown in Table 4, the projects QXmpp, Bitcoin Core, HHVM and CoreCLR are highly clusterable (the H-value is far below the threshold 0.5), while the H-value for the projects Muduo and LevelDB are not that significant.

Table 4: Clustering Tendency Assessment

Project	H-value	Project	H-value
Muduo	0.2079	**Bitcoin Core**	0.1536
LevelDB	0.3148	**HHVM**	0.1037
QXmpp	0.1249	**CoreCLR**	0.0634

Since the dataset is clusterable, we then aim to determine the most appropriate number of clusters (denoted by k) for each project based on Gap statistic. Although it is unnecessary to specify the value of k for hierarchical clustering, we compute the optimal number of clusters in advance for the purpose of better data interpretation. The result is demonstrated in Table 5.

Table 5: The Optimal Number of Clusters

Project	PFM	PRM	PLM
Muduo	1	1	1
LevelDB	1	1	1
QXmpp	10	1	1
Bitcoin Core	3	1	10
HHVM	10	10	4
CoreCLR	5	3	1

Making use of hierarchical agglomerative clustering, we then classify source files into subgroups that have similar stylistic patterns. Figure 3c illustrates the clustering result for Bitcoin Core project[17]. It can be observed that all source files are listed along the horizontal axis, each leaf node corresponds to one source file. The (dis)similarity level between two source files is measured along the vertical axis. The greater the height, the less similar they are. We hide the file names for the reason of clarity.

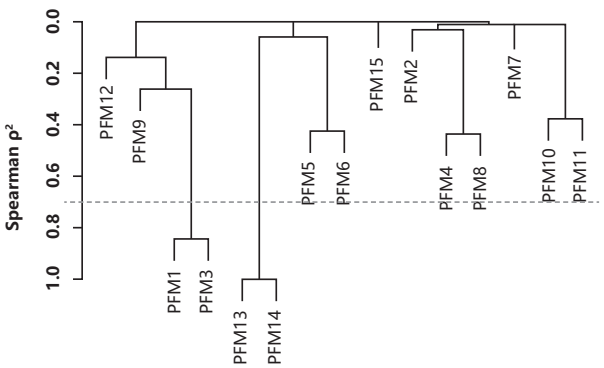
3.3 Discussion

Variable Deduction. As shown in Table 3, metrics that most often excluded by variable clustering analysis are PFM13 (correlated with PFM14), PRM4 (correlated with PRM5) and PLM4 (correlated with PLM5) for each dimension. A deeper analysis of their specific meaning (Table 1) verifies the feasibility of such deduction as the correlations do exist: $PFM13 + PFM14 + PFM15 = 1$, $PRM4 + PRM5 = 1$, $PLM4 + PLM5 = 1$.

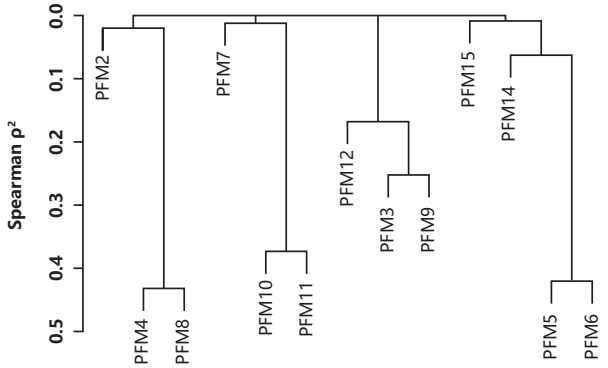
The Impact of Project Scale. Taken together Table 2 and Table 5, the optimal number of clusters seems to increase with the scale of the software project. Considering that large-scale projects tend to have long histories and involve a large number of developers, which could well give rise to the problem of stylistic inconsistency, the conclusion is reasonable and corresponds to reality.

[16]A possible interpretation of the H-value approximately equal to 0.5 is that there is only one programming style present in the software project.

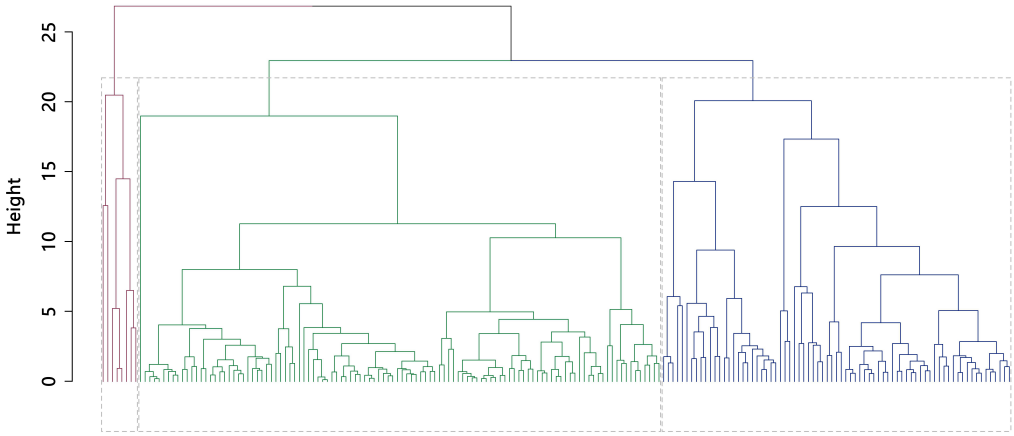
[17]The clustering results for the other projects are omitted here due to space constraints.



(a) Clustered Metrics before Correlation Analysis



(b) Clustered Metrics after Correlation Analysis



(c) Cluster Dendrogram

Figure 3: Results for Project Bitcoin Core in PFM Dimension

The Interpretation of Cluster Dendrograms. In the dendrogram displayed in Figure 3c, we cut the tree diagram at a certain height to obtain k clusters, which indicates the number of distinct types of programming style within the corresponding project teams. Note that the choice of k is somehow subjective even ambiguous, with interpretations depending on the shape and scale of the distribution of data samples and the method used for measuring (dis)similarities, it is only a comparatively better option about the number of clusters. We suggest that practitioners should emphasize more on the (dis)similarities degree among source files, i.e., the data interpretation of the proximity among leaf nodes in the dendrogram.

Manual Analysis. We manually examine four out of the six projects to evaluate the effectiveness of our methodology. Two projects with the higher value of Hopkins statistic (greater than 0.2) and the lower number of clusters, Muduo and LevelDB, perform well at stylistic consistency. Through close investigation, we notice that Windows Coding Style Conventions[18] are widely adopted in project Muduo. Similarly, the majority of source code in project LevelDB well conform to Google Style Guides[4]. On the contrary, the different physical layout of open curly brackets in flow control statement leads to worse performance in PFM dimension for project QXmpp, while project Bitcoin Core suffers from the same issue in terms of indentation style. Its consistency in PRM dimension benefits from unified identifier naming convention of Hungarian Notation[19], while the usage of looping statement varies greatly, which indicates the collaboration of a great number of developers.

4. THREATS TO VALIDITY

Programming style is highly individualistic yet somehow unquantifiable. Although we have made our best to guarantee the quality and comprehensiveness of the stylistic metrics, by no means do we claim that the metrics we have chosen are able to completely characterize different programming styles. On the other hand, our choice of metrics takes into account a variety of factors, some of them are admittedly subjective. This threat may put some limitations on our conclusions, which could be mitigated by further investigation on the essential features of programming style.

Another possible threat to this study is the inaccuracy in metric extraction. Our tool is highly dependent on regular

[18]https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx

[19]https://en.wikipedia.org/wiki/Hungarian_notation

expressions (i.e., patterns used to match character combinations), which inevitably suffers from both false positives and false negatives.

Besides, we consider each source file as the unit to compute stylistic fingerprints, which may be somewhat coarse-grained. If inconsistency occurs only in a small portion of the source code, the file-level metrics could well fail to reflect such problem. We intend to adopt fine-grained metrics (e.g., method-level) in the future study in an attempt to precisely pinpoint the abnormal code block.

Lastly, six well-developed projects from widely differing application areas are adopted to validate our conclusions. However, all the projects involved in this study come from the same source, Github. Therefore, the methodology and the corresponding results may not suitable for other open source projects and industrial projects. This threat to generalization could be mitigated by empirical validations extending to other sources. On the other hand, we biased our selection for well-known and successful projects. It is possible that less popular projects may yield different conclusions. However, it is not believed that the selection of the project has skewed the overall results, yet further study is required to explore this issue.

5. RELATED WORK

Style Analyzer. A number of research efforts have been undertaken to the development of style analyzers. Rees et al. [20] assessed the style of Pascal programs by awarding marks automatically based on 10 metrics chosen on an intuitive basis. Berry et al. [6] designed 11 program characteristics to represent lucidity or understandability of C program by giving a style score to each program. In an attempt to assist analysis, Oman et al. [16] separated stylistic factors into two class: those pertaining to the typographic arrangement and those measuring the structural content of the code. They further extended their taxonomy into four major categories: general practices, typographic style, control structure style and information structure style [18]. Essentially, style analyzers are grading tools that compute a single style score for each program to measure the program quality. Such grading method is not adopted in our study. However, the selected stylistic metrics and the corresponding taxonomy proposed by the aforementioned work are indeed important references for our research.

Style Formatter. Reading source code is the most time-consuming task during the entire software maintenance process [9, 19]. A better programming style could significantly release the comprehension burden. Corbo et al. [8] provided Smart Formatter that allows programmers to learn coding style rules from existing source code and to apply these rules to the code under development. Allamanis et al. [2] developed NATURALIZE that learns the style of a codebase and suggests revisions to improve stylistic consistency. Arai et al. [4] proposed an Eclipse plugin tool, which was capable of learning the proper programming style of Java and making recommendations in terms of naming format, correct scope level, etc. Makela et al. [13] presented an interactive style checking tool that visually shows the places in Java programs where a chosen style is not followed. Although quantities of well-developed programming standards and the corresponding formatting tools were provided by previous work, the stylistic inconsistency within project teams is still inevitable due to many reasons. Therefore, this study concentrates solely on the analysis of such issue that even style formatters failing to address.

Plagiarism Detection and Authorship Identification. In software programming, the reuse of whole or part of the source code without any reference or permission of the original author can be classified as software plagiarism. Arabyarmohamady et al. [3] proposed a plagiarism detection framework based on coding style, which was divided into four modules: feature extraction module, similarity detection module, developer profile module and style level checking module. Another similar trend is to trace the origins of programs (i.e., identify the author of a piece of software). Oman et al. [17] explored the issue of authorship identification in an attempt to assist software plagiarism. They built a Pascal source code analyzer that generated an array of Boolean measurements in order to capture the typographic or layout style characteristics of programs. Krsul et al. [12] found a set of programming style characteristics that remain constant for a significant portion of the programs. A total of 49 metrics were proposed to capture such characteristics of C source code. Furthermore, their experiments demonstrated that Gaussian Likelihood Classifiers and MLP Neural Networks were able to classify the programs with very low error rates. Based on Krsul's study [12], Ding et al. [10] extracted 56 stylistic metrics from Java source code for the purpose of authorship identification. Their study showed that the authorship of 62.6%-67.2% of the Java programs considered could be correctly identified with the extracted metrics. One thing our study and [3, 10, 12] have in common is that we all follow the attribute counting method to characterize programming style, yet our choice of stylistic metrics is quite different. Moreover, the design of our empirical experiments is also influenced by the accumulated research experiences in this field.

Research in the area of programming style has a long history, which has attracted the attention of many researchers and practitioners. However, there is only a little focus on the problem of stylistic inconsistency. Therefore, the target of our study is not to propose more programming guidelines or to develop better formatting tools to guarantee one particular standard, but to shed light on the research of unavoidable stylistic inconsistency within software projects.

6. CONCLUSIONS

Although a great number of software methodologies and guidelines are provided, the problem of stylistic inconsistency during the entire development and maintenance process is still unavoidable. Instead of proposing more coding standards to prevent such problem, this study focused on the representation of different programming styles and the measurement of inconsistency degrees, which are the fundamental challenges need to be addressed first. More specifically, this paper made the following contributions:

- A collection of stylistic metrics and the corresponding taxonomy are proposed to characterize the essential features of the programming style for an individual or a single document, which serves as the basis for our study. A useful and usable tool (available on-line[8]) is developed for extracting those metrics from C/C++ projects and generating CSV-formatted dataset automatically to support further analysis. An empirical experiment confirms the feasibility and validity of such

fingerprinting methodology (RQ1).
- Making use of hierarchical agglomerative clustering, dendrograms could be produced in order to reveal the stylistic (dis)similarities among source files, which is an explainable and understandable outcome for practitioners. The effectiveness of such methodology has been supported by empirical experiment based on several open source projects (RQ2).

In future, we plan to expand this research to other popular languages. Although slanted toward C/C++ projects, the measures are adaptable for other high-level programming languages with a slight adjustment.

In this study, we have provided some tools and techniques as the theoretical and experimental basis for quantitatively measuring the degree of stylistic inconsistency within software projects, this is to improve program comprehension and reduce the maintenance cost as the ultimate goal, which is a pioneering work in this area. We hope that our study could draw more attention of both researchers and practitioners to this direction, which we believe is worthy of future research studies.

7. ACKNOWLEDGMENTS

This research is supported by the City University of Hong Kong research funds (Project No. 7200354, 7004222, 7004474).

8. REFERENCES

[1] Akhlaq, U. *Impact of Software Comprehension in Software Maintenance and Evolution*. PhD thesis, Blekinge Institute of Technology, 2010.

[2] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014* (New York, New York, USA, 2014), ACM Press, pp. 281–293.

[3] Arabyarmohamady, S., Moradi, H., and Asadpour, M. A coding style-based plagiarism detection. In *Proceedings of 2012 International Conference on Interactive Mobile and Computer Aided Learning (IMCL)* (nov 2012), no. Imcl, IEEE, pp. 180–186.

[4] Arai, M. Development and evaluation of Eclipse plugin tool for learning programming style of Java. In *2014 9th International Conference on Computer Science & Education* (aug 2014), vol. 30, IEEE, pp. 495–499.

[5] Avgustinov, P., Baars, A. I., Henriksen, A. S., Lavender, G., Menzel, G., Moor, O. D., Schafer, M., and Tibble, J. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (may 2015), IEEE, pp. 437–447.

[6] Berry, R. E., and a.E. Meekings, B. A style analysis of C programs. *Communications of the ACM 28*, 1 (jan 1985), 80–88.

[7] Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., and Sharif, B. The impact of identifier style on effort and comprehension. *Empirical Software Engineering 18*, 2 (apr 2013), 219–276.

[8] Corbo, F., del Grosso, C., and di Penta, M. Smart Formatter: Learning Coding Style from Existing Source Code. In *2007 IEEE International Conference on Software Maintenance* (oct 2007), no. i, IEEE, pp. 525–526.

[9] Deimel, L. E. The uses of program reading. *ACM SIGCSE Bulletin 17*, 2 (jun 1985), 5–14.

[10] Ding, H., and Samadzadeh, M. H. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software 72*, 1 (jun 2004), 49–57.

[11] Hopkins, B., and Skellam, J. A new method for determining the type of distribution of plant individuals. *Annals of Botany 18*, 2 (1954), 213–227.

[12] Krsul, I., and Spafford, E. H. Authorship analysis: identifying the author of a program. *Computers & Security 16*, 3 (jan 1997), 233–257.

[13] Mäkelä, S., and Leppänen, V. Japroch: A tool for checking programming style. *Kolin Kolistelut-Koli Calling 2004* (2004), 151.

[14] Miara, R. J., Musselman, J. a., Navarro, J. a., and Shneiderman, B. Program indentation and comprehensibility. *Communications of the ACM 26*, 11 (nov 1983), 861–867.

[15] Nguyen, V., Deeds-Rubin, S., Tan, T., and Boehm, B. A sloc counting standard. In *COCOMO II Forum* (2007), vol. 2007.

[16] Oman, P. W., and Cook, C. R. A paradigm for programming style research. *ACM SIGPLAN Notices 23*, 12 (dec 1988), 69–78.

[17] Oman, P. W., and Cook, C. R. Programming style authorship analysis. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990's Computing trends in the 1990's - CSC '89* (New York, New York, USA, 1989), ACM Press, pp. 320–326.

[18] Oman, P. W., and Cook, C. R. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation - CSC '90* (New York, New York, USA, 1990), ACM Press, pp. 244–250.

[19] Raymond, D. R. Reading source code. *Proc. CASCON* (1991), 3–16.

[20] Rees, M. J. Automatic assessment aids for pascal programs. *SIGPLAN Not. 17*, 10 (Oct. 1982), 33–42.

[21] Sommerville, I. *Software Engineering*. International Computer Science Series. Pearson, 2011.

[22] Tibshirani, R., Walther, G., and Hastie, T. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 63*, 2 (may 2001), 411–423.

[23] Ward, J. H. Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association 58*, 301 (mar 1963), 236.

[24] Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 215–223.