



# How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects

Wei Qin Zou<sup>1</sup>  · Jifeng Xuan<sup>2</sup> · Xiaoyuan Xie<sup>2</sup> · Zhenyu Chen<sup>1</sup> · Baowen Xu<sup>1</sup>

Published online: 03 June 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

GitHub is a popular code platform that provides infrastructures to facilitate collaborative development. A Pull Request (PR) is one of the key ideas to support collaboration. Developers are encouraged to submit PRs to ask for the integration of their contributions. In practice, not all submitted PRs can be integrated into the codebase by project maintainers. Existing studies have investigated factors affecting PR integration. Nevertheless, the code style of PRs, which is largely considered by project maintainers, has not been deeply studied yet. In this paper, we performed an exploratory analysis on the effect of code style on PR integration in GitHub. We modeled the code style via the inconsistency between a submitted PR and the existing code in its target codebase. Such modeling makes our study not limited by a specific definition of code style. We conducted our experiments on 50,092 closed PRs in 117 Java projects. Our findings show that: (1) There indeed exists code style inconsistency between PRs and the codebase. (2) Several code style criteria on how to use spaces or indents, make comments, and write code lines with a suitable length, tend to show more inconsistency among PRs. (3) A PR that is consistent with the current code style tends to be merged into the codebase more easily. (4) A PR that violates the current code style is likely to take more time to get closed. Our study shows evidence to developers about how to deliver better contributions to facilitate efficient collaboration.

**Keywords** Pull request · Code style inconsistency · Exploratory study

---

Communicated by: Ahmed E. Hassan

---

✉ Zhenyu Chen  
zychen@nju.edu.cn

Extended author information available on the last page of the article.

# 1 Introduction

GitHub<sup>1</sup> is a widely-used platform for collaborative software development upon the Git version control system (Kalliamvakou et al. 2014). According to the official website<sup>2</sup>, GitHub holds more than 52 million repositories up to October 31st, 2017<sup>3</sup>. Besides the majority of open source organizations, commercial companies, such as Microsoft and VMware, are putting their projects into GitHub to attract talented developers (Kalliamvakou et al. 2015).

As a social development platform, GitHub fully supports collaboration via managing Pull Requests (PRs). A PR consists of one or more commits submitted by the contributor, which are requested to be integrated into the repository. A popular repository may receive a great amount of PRs and the project maintainers need to perform careful code reviews to decide which PRs can be merged into the codebase. For example, up to October 31st, 2017, the repository “rails/rails”<sup>4</sup> has closed 19,459 PRs, including 13,128 explicitly marked as merged ones. As a consequence, for the open source community, it is important to understand what makes PRs be accepted or rejected.

The code style of PRs relates to the determination of their integration. (Gousios et al. 2015) conducted a qualitative survey with experienced project owners on the challenges of integrating PRs in GitHub. Their survey found that the code style greatly affected the acceptance of PRs. (Rigby and Storey 2011) and (Bacchelli and Bird 2013) also found that code reviewers, in other platforms rather than GitHub, cared about the code style of patches during the patch reviewing process. In addition, we observed that many projects in GitHub explicitly required their developers to keep the code style consistent with the existing code. For example, in the project “AndlyticsProject/andlytics”,<sup>5</sup> contributors were asked to keep their code consistent with the around code: “Try to be consistent with the code around you. As a general rule: 1) use tabs not spaces; and 2) aim for a maximum line length of 100, unless it looks better being on a single line”<sup>6</sup>.

*How does Code Style Inconsistency affect Pull Request Integration?* Though code style is emphasized during collaboration in the pull-based platform, there exists no prior deep study on the correlation between code style inconsistency and the integration of PRs.

In this paper, we conducted an exploratory study on the integration of PRs from 117 projects in GitHub. This study investigated the effect of code style on PR integration based on 50,092 collected PRs since 2011. We quantitatively identified the code style based on 37 pre-defined criteria for each PR. *Our model of the code style focused on the code style inconsistency between each submitted PR and the existing code in its codebase.* For the sake of simple calculation, the term “existing code in the codebase” denotes the original source code files that would be updated by the submitted PR. Hence, our study is not limited by a specific definition of code style. We tried to show the effect of code style inconsistency on PR integration via answering the following four Research Questions (RQs).

---

<sup>1</sup><http://github.com>

<sup>2</sup><http://github.com/features>

<sup>3</sup>In GitHub, a “repository” denotes a project in general. In this paper, we use “repository” and “project” interchangeably.

<sup>4</sup><http://github.com/rails/rails>

<sup>5</sup><https://github.com/AndlyticsProject/andlytics>

<sup>6</sup><http://github.com/AndlyticsProject/andlytics#contributing>

**RQ1. Is there any difference between the code style of submitted PRs and that of the existing source code?** All experimental projects had a certain number of PRs that showed inconsistency in code style. PRs in all projects showed inconsistency in at most 8 to 28 code style criteria; 50% PRs on all projects showed inconsistency in no more than 3 criteria. Besides, the policy of a project on code style and the experience of PR authors to some extent would affect the ratio of PRs with no inconsistency in code style in the project.

**RQ2. Which code style criteria tend to show much inconsistency among PRs?** Several code style criteria on how to use spaces or indents, make comments, and write code lines of suitable length, tended to show more inconsistency among PRs. Besides, there also existed several criteria (such as writing import and package statements in one line) that rarely showed inconsistency in most projects.

**RQ3. How does code style affect the merging of PRs?** The code style inconsistency had a small negative effect on the merging of PRs. Besides, two code style criteria on using tabs and writing code lines of suitable length had a relatively larger effect on the merging of PRs, compared to the other criteria.

**RQ4. How does code style affect the closing time of PRs?** The code style inconsistency had a small positive effect on the time cost of closing PRs. We further found that two code style criteria on writing suitable-length code lines and providing Javadoc for a public class and non-private members had a relatively larger effect on the closing time of PRs than the other criteria.

Our major contributions are listed as follows:

1. We conduct an exploratory study on the code style inconsistency of 50,092 PRs from 117 GitHub projects. We propose to measure the code style inconsistency between a PR and the existing code in the codebase instead of directly modeling the code style. This avoids the limitation by a specific definition of the code style.
2. We empirically identify some code style criteria, which reveal most difference between submitted PRs and the existing code in the codebase.
3. We analyze the effect of the code style on integrating PRs, including the decision of merging new PRs and the time cost of closing PRs.

The rest of this paper is structured as follows. We discuss the background and motivation in Section 2. Then we introduce our experimental methodologies in Section 3. Experimental results are presented in Section 4. Section 5 discusses the implications of our findings and the threats to the validity of our work. We list the related work and conclude our work in Sections 6 and 7, respectively.

## 2 Background and Motivation

In this section, we first briefly introduce GitHub, mainly focusing on pull-based development and the mechanism of PRs; then we present the motivation of our study.

## 2.1 Background

GitHub is a mainstream development platform, which provides code hosting and collaboration (Kalliamvakou et al. 2014). This platform is built based on a version control system Git.

**Pull-based development** Pull-based development provides a platform for developers to submit their code changes via pull requests. In GitHub, developers generally cannot directly submit their commits into a target project that is initialized by other developers. Instead, a developer usually needs to *fork* (i.e., clone) the project into their own account as a copy; then he/she makes code changes to the forked project. A PR is submitted if he/she expects to merge code changes into the target project. Then the project maintainer will review the PR and decide whether to merge it into the codebase.

**Pull Request (PR)** PRs are the key of collaborating with each other in GitHub (Gousios et al. 2016). If a developer wants to merge his/her own contributions into the target project, he/she could submit a PR to the queue of PRs of the target project. A PR consists of one or more *commits*, each of which contains code changes to the codebase at a time. Once a PR is submitted, the target project will timely present the detailed difference, i.e., deleting or adding source code. In GitHub, a modified line in *diffs* is always presented as a deleted line plus an added line. A maintainer of the target project then conducts code review on the submitted PR. The PR will be merged into the project if its commits are accepted by the maintainer; otherwise, the maintainer will close the PR or ask the submitter to modify for next-round code review.

## 2.2 Motivation

It has always been emphasized by many Open Source Software (OSS) communities that following the current code style is important. For example, the GNU community requires all contributions to follow the *GNU Coding Standards*<sup>7</sup>, which contains various standards, including the code style. These standards aims to build clean and consistent systems during collaborative development. Similarly, all contributors participating in Google projects are asked to obey the Google Style Guides<sup>8</sup>.

As a popular coding platform for distributed development, GitHub encourages repository maintainers to explicitly and clearly describe their coding style in the *readme* or *contribution* documents<sup>9</sup>. Meanwhile, contributors are reminded to comply with the code styles requested by their target repositories: “*Contribute in the style of the project to the best of your abilities. This may mean using indents, semi colons or comments differently than you would in your own repository, but makes it easier for the maintainer to merge, others to understand and maintain in the future*”<sup>10</sup>.

---

<sup>7</sup>[http://www.gnu.org/prep/standards\\_toc.html](http://www.gnu.org/prep/standards_toc.html)

<sup>8</sup><http://github.com/google/styleguide>

<sup>9</sup>It is common for a project to have a readme file or a contribution file. The readme file broadly describes the project; while the contribution file mainly introduces the tips to contribute to this project.

<sup>10</sup><http://guides.github.com/activities/contributing-to-open-source/#contributing>

In fact, there do exist many projects following this practice. For example, the project “mongodb/mongo”<sup>11</sup> requests “*all commits to the MongoDB repository must follow the kernel development rules*”. Gousios et al. (2015) have conducted a survey with GitHub repository maintainers on the problem of merging contributions. They found that the code style was greatly concerned by maintainers when deciding whether to accept a contribution or not.

Unfortunately, due to the openness of GitHub repositories, there always exist contributors who do not follow the requested code styles. Project maintainers have to expend more efforts to communicate with contributors to cope with the issues of the code style. For example, a PR with ID 3512 in “rubinius/rubinius” was asked not to use single-letter variable names by the reviewer<sup>12</sup>: “*One general request in advance: please don’t use single letter variables. Looking at code I have no clue what n means*”.

For another example, in project “querydsl/querydsl”<sup>13</sup>, from the commit logs, we found that many commits aimed at fixing code style problems: “*Remove some extra whitespace; Normalize tabs to spaces; Code style: remove final in local variables*”.

Motivated by the above evidence<sup>14</sup>, we are interested to know how prevalent these code style problems are, and how these problems affect the PR processing. Thus we decided to conduct a deep study on the effect of the code style on PR integration.

### 3 Methodologies

Our experimental process included four parts. First, we crawled relevant projects from GitHub as our experimental subjects. Secondly, we defined 37 code style criteria based on literature to characterize the code style of PRs. Thirdly, we calculated the code style inconsistency based on the above criteria. Last, we explored the effect of code style on integrating PRs via answering four Research Questions (RQs) as in Section 4.

#### 3.1 Target Projects

Java is one of top active languages in GitHub<sup>15</sup>. Thus, we chose non-forked Java projects (i.e., these projects were not copies of other projects) as our experimental subjects. In our work, we investigated the effect of code style inconsistency on PR integration via controlling confounding factors (such as the commit size and the number of forked projects) that may also affect the PR integration. To exactly measure confounding factors, it was essential to completely record historical events, such as events of adding members and forking projects. Considering GitHub only preserves the latest three months of records for some historical events (e.g., adding members to a repository), we decided to use another data source of GitHub historical events, namely GitHub Archive<sup>16</sup>.

<sup>11</sup><http://github.com/mongodb/mongo>

<sup>12</sup><http://github.com/rubinius/rubinius/pull/3512>

<sup>13</sup><http://github.com/querydsl/querydsl>

<sup>14</sup>In this study, some motivation examples (i.e., GNU, Google, and GitHub) mainly came from manual search of code style related documentation in well-known open source communities or company originated open source projects; while other examples (i.e., mongodb/mongo, rubinius/rubinius, and querydsl/querydsl) were collected by manually checking the documents and commit logs of some randomly selected popular projects on GitHub.

<sup>15</sup><http://github.info>

<sup>16</sup><https://www.githubarchive.org/>

**Table 1** Summary of attributes on 117 GitHub java projects

| Attribute     | Min  | Median | Max     | Mean    | St. Dev. |
|---------------|------|--------|---------|---------|----------|
| size (MB)     | 2.99 | 41.89  | 1926.74 | 137.71  | 296.75   |
| forks         | 19   | 325    | 4338    | 616.4   | 842.76   |
| total PRs     | 207  | 417    | 4,670   | 687.6   | 826.93   |
| closed PRs    | 203  | 394    | 4,631   | 667.6   | 810.99   |
| total issues  | 240  | 988    | 7,228   | 1,451.1 | 1,328.60 |
| closed issues | 235  | 912    | 6,677   | 1,334.3 | 1,216.38 |
| developers    | 10   | 63     | 639     | 81.48   | 75.46    |

GitHub Archive provides more than 20 event types, which makes it possible for us to compute confounding factors. Since GitHub Archive only stores historical events that appeared after Feb. 2011, we only considered projects created after Feb. 2011 in the experiments. This guaranteed that we would have complete historical events for each project. There were 1,846,386 non-forked Java projects created after Feb. 2011 at the moment (i.e., Jan. 2016) of data crawling.

Following the work by Vasilescu et al. (2015), among 1,846,386 projects, we only chose projects with over 200 closed PRs. This made us focus on projects which truly used the PR mechanism to perform collaborative development. Meanwhile, since we took the number of issues reported by a developer as a proxy of his/her activity (the developers' activity may affect the integration of their submitted PRs), we also filtered out projects that had less than 20 closed issues in GitHub. Besides, in GitHub, developers are divided into insiders and outsiders: an *insider* is a developer who can directly commit to a target project while an *outsider* is a developer who cannot. In this study, we chose projects that were developed by a collaborative team of both insiders and outsiders. Specifically, we required that each project contained more than 10 developers and the number of outsiders was not smaller than that of insiders. This helped us avoid projects that were mainly built by a small team of insiders.

For data collection, we implemented a web crawler to retrieve the statistics of projects and directly compared them with the above thresholds. In details, the numbers of closed PRs, issues, contributors (i.e., developers) were listed in the site of each project in GitHub; the number of insiders could be retrieved by counting the assignee list of issues<sup>17</sup>.

After filtering, 118 projects were left. For those projects, we manually checked their readme documents, so as to ensure that each project was a typical software development project. During checking process, we found that one project, i.e., “filipg/amu\_automata\_2011”<sup>18</sup>, was created for a programming course in a university. Thus, we removed it from our data set. Finally, we got 117 projects to conduct experiments. Table 1 presents more details about those projects.

### 3.2 Code Style Criteria

To conduct an exploratory analysis, we characterized the code style of PRs with 37 code style criteria (in Table 2) that involved almost every aspect of a code style. These criteria

<sup>17</sup><https://help.github.com/articles/filtering-issues-and-pull-requests-by-assignees/>

<sup>18</sup>[http://github.com/filipg/amu\\_automata\\_2011](http://github.com/filipg/amu_automata_2011)

**Table 2** The 37 criteria for characterizing the code style

|               | criteria          | Description   |
|---------------|-------------------|---|
| Structure     | noLineWrap        | Never break import and package lines.   |
|               | noStarImport      | No .* imports.  |
|               | oneTopClass       | Put a top class in its own file.  |
|               | emptyLineSep      | Use a blank line after header, package, import lines, class, methods, fields, static, and instance initializers.                              |
| Formatting    | whitespaceAround  | Use a space between a reserved word and its follow-up bracket, e.g., if (.  |
|               | genericWhitespace | Use a space before the definition of generic type, e.g., List <.  |
|               | opWrap            | Break after '=' but before other binary operators.  |
|               | sepWrap           | Break after ',' but before '.'.   |
|               | maxLineLen        | 100 characters in maximum.  |
|               | leftCurly         | Put '{' on the same line of code.   |
|               | rightCurly        | Put '}' on its own line.  |
|               | emptyBlock        | No empty block for control statements.  |
|               | needBraces        | Use '{}' for single control statements.   |
|               | noTab             | No '\t' in source code.   |
|               | indentation       | Set a basic offset as four spaces.  |
|               | noMultiVar        | Put a variable declaration on its own line.   |
|               | oneStmntPerLine   | Each line holds one statement.  |
|               | annotationLoc     | Put each annotation one line before a class or a method.  |
|               | upperEll          | Use 'L' for Long integer literals.  |
|               | modifierOrder     | Follow the order: public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp.                   |
|               | Naming            | fallThrough   |
| needDefault   |                   | Use "default" in switch.  |
| typeName      |                   | Be in UpperCamelCase, e.g., BinarySearchTree.   |
| packageName   |                   | Be in all lowercase, with consecutive words concatenated together by '.', e.g., com.edu.nameusage.  |
| methodName    |                   | Be in lowerCamelCase, e.g., getName.  |
| memberName    |                   | Be in lowerCamelCase, e.g., localAddress.   |
| parameterName |                   | Be in lowerCamelCase, e.g., customerId.   |
| Comments      | catchParaName     | Be in lowerCamelCase or in one-character lowercase, e.g., divideZeroException or e.   |
|               | localVarName      | Be in lowerCamelCase, e.g., clientAccount.  |
|               | noEmptyAtDes      | Use a description after @ tag.  |
|               | javadocMethod     | Javadoc is mandatory for a public class and its non-private members.  |
|               | javadocParagraph  | In Javadoc, each blank line is leaded with *. Each paragraph except the first one, needs a <p> before its first word.                         |
|               | tagIndent         | In Javadoc, use four spaces for new indentation level in @ clauses.   |
|               | summaryDoc        | No phrases like "This method returns" in Javadoc summary.   |
|               | cmtIndent         | Comments indent at the same level with surrounding code.  |
|               | singleLineJavadoc | A Javadoc can be placed in a single line when it can fit in a line and contains no tag @.   |
|               | atOrder           | Follow the order: @author, @version, @param, @return, @throws, @exception, @see, @since, @serial, @serialField, @serialData, and @deprecated. |

were extracted via manually checking widely-used code style rules for the Java language. For Java projects, there mainly exist two broadly-accepted documents of code styles, i.e., the Java code styles by Google (2013) and by Oracle (Sun Inc.) (Oracle 1999). In this paper, we focused on code style criteria which were mentioned in both two documents. The obtained 37 criteria were also basically the metrics that *Checkstyle*<sup>19</sup> (a very popular Java code style checking tool) reported.

*Note that we did not conduct the study based on a specific code style.* Instead, we defined a list of code style criteria and measured the source code against them. In other words, we compared a PR and its corresponding existing code in the codebase against a code style criterion. If they both violated (or neither violated) a code style criterion, then we considered they kept consistent under the criterion. This setting avoided the bias by different choices of code style. Section 3.3 describes how to define the code style inconsistency in details.

Code style criteria could not be directly converted into numeric measurements (Google 2013). A submitted PR indicated a potential change of source code. In our work, we aimed to explore the relationship between the code style of a PR and its integration to the codebase. That is, we measured the code style of a PR by weighting the inconsistency before submitting the PR and after the potential merging of the PR.

Specifically, for each PR, we first extracted the files that were modified by the PR. For each file, we maintained two code versions, i.e., the original version (denoted as **original**) and the modified version (denoted as **modified**) before and after merging the PR to the codebase. Then we measured the code styles of the **original** and the **modified** files against on the criteria in Table 2, respectively. Then, we computed the inconsistency between their code styles (see Section 3.3). Notice that we only compared **modified** files with their corresponding **original** files within PRs rather than with the files of the whole codebase. We made this choice mainly because the codebase is always much larger and may contain various code styles, which makes our calculation complex and more importantly may involve other factors that affect our analysis.

Given one file, for both the **original** version and the **modified** version, we represented its code style with a 37-dimension Boolean vector; each dimension denoted whether the code within the file violated one corresponding criterion in Table 2. In other words, we used the violation results of 37 code style criteria as the code style. Then the code style of **original** and **modified** version for one file could be defined as follows:

$$s_{original} = \langle o_1, o_2, \dots, o_{37} \rangle \text{ and } s_{modified} = \langle m_1, m_2, \dots, m_{37} \rangle$$

where  $o_i \in \{1, 0\}$  and  $m_i \in \{1, 0\}$  denote whether the  $i$ th criterion is violated and  $1 \leq i \leq 37$ . For instance,  $o_1 = 1$  means the first criterion is violated in  $s_{original}$ .

Note that in our definition of the code style, we only distinguished whether a code style criterion was violated or not. It was possible to further characterize how much a code style criterion was violated. As further distinguishing different scenarios of violation would lead to high complexity, we instead chose to use the Boolean values (violating a code style criterion or not) to make our idea clear in this study.

<sup>19</sup><http://checkstyle.sourceforge.net/>

### 3.3 Code Style Inconsistency in PRs

We adopted a two-step approach to calculate the code style inconsistency for each PR. First, we calculated the inconsistency of code styles between each pair of the **original** version and the **modified** version of one file. The inconsistency for each pair is defined as follows:

$$\text{each\_inconsistency}(s_{\text{original}}, s_{\text{modified}}) = s_{\text{original}} \text{ XOR } s_{\text{modified}}$$

where XOR denotes the exclusive or value between two Boolean vectors. each\_inconsistency is a 37-dimension vector and its each element  $e_i \in \{1, 0\}$ .

Then, for a PR that changed a set F of files, the code style inconsistency for a PR is defined as follows:

$$s_{pr} = \langle p_1, p_2, \dots, p_{37} \rangle$$

where

$$p_i = \begin{cases} 0 & \sum_{file \in F} e_i = 0 \\ 1 & \text{otherwise} \end{cases}$$

Here,  $p_i = 1$  means at least one file that is changed by the PR is inconsistent with the  $i$ th criterion; while  $p_i = 0$  means all files within the PR keep consistent with the  $i$ th criterion.

After we got the inconsistency over the 37 code style criteria, we defined the total inconsistency of a PR as follows:

$$\text{total\_inconsistency} = \sum_{1 \leq i \leq 37} s_{pr}[i]$$

where  $\text{total\_inconsistency} \in \{0, 1, \dots, 37\}$ . A high value indicates that the code style between the PR and the existing code is highly inconsistent.

Notice that when we calculated the code style inconsistency for a PR, we only considered the files that were modified by the PR. Newly added files that did not have corresponding original files were ignored during calculation.

## 4 Experimental Results

In this section, we first describe our experiment setup, including data preparation, RQ designs, and confounding factors in Section 4.1; then we present detailed analysis via answering four Research Questions (RQs) in Sections 4.2-4.5.

### 4.1 Experiment Setup

#### 4.1.1 Data Preparation

We conducted experiments on 117 projects mentioned in Section 3.1. Since we cannot determine the final status of open PRs, we only chose PRs that were closed before Jan. 2016 (i.e., the timestamp we crawled data) to perform our experiments. In total, there were 78,112 closed PRs for 117 projects. Then, we removed PRs which mainly targeted modifying non-Java files. To be specific, a PR was removed if the ratio of modified lines of Java code over all modified lines in a PR was less than 0.5. Then 51,939 PRs were remained. Some PRs (799 PRs) had only deleted code for which we had no corresponding added code to calculate the code style inconsistency for them, thus we removed them from our data set.

Meanwhile, we also removed a small number of PRs (805 PRs) that only added code files since our study only considered the inconsistency of code style between new changes and the original files. These 805 PRs that only added files did not rely on original files. Thus we removed these 805 PRs. Then 50,335 PRs were remained.

During data crawling, we found that there were very few (243) PRs whose modified files were incomplete or unavailable. Thus, they were also filtered out from the dataset. Finally, we got 50,092 PRs. Table 3 shows the basic summary of our experimental data. Our following experiments are conducted based on all these 50,092 PRs.

#### 4.1.2 Design of RQs

We explored the effect of code style inconsistency on the integration of PRs via the following four RQs. Such exploration revealed nine findings about the code style and PR integration.

**RQ1. Is there any difference between the code style of submitted PRs and that of the existing source code?**

**RQ2. Which code style criteria tend to show much inconsistency among PRs?**

**RQ3. How does code style affect the merging of PRs?**

**RQ4. How does code style affect the closing time of PRs?**

In RQ1, we gave a general statistics on the code style inconsistency between submitted PRs and existing code. In RQ2, we further analyzed the code style criteria and answered which criteria contributed most to the above inconsistency. In the follow-up RQ3 and RQ4, we leveraged regression models to measure the effect of code style inconsistency on PR integration, including the decision of merging PRs and the time cost of closing PRs.

In GitHub, PRs and projects are hierarchically structured: the level of PRs are viewed as the micro level and the level of projects are the macro level. Considering our data contained a hierarchical structure, we chose to use mixed-effects models for our experiments in RQ3 and RQ4. Mixed-effects models could provide accurate estimates of relations between individual-level explanatory variables and a response variable with considering group structure within data (Cohen et al. 2013). Using mixed-effects models could avoid the potential problem of overestimating the significance of individual regression coefficients that generally happens when applying linear or logistic regression models to hierarchical datasets (Cohen et al. 2013).

We built two types of mixed-effects models during our experiments, i.e., a linear mixed-effects model (LMM) and a generalized linear mixed-effects model (GLMM). LMM applies

**Table 3** Statistical summary of experimental data after data preparation

| Level   | Attribute              | Min     | Median | Max      | Mean  | St. Dev. |
|---------|------------------------|---------|--------|----------|-------|----------|
| Project | closed PRs             | 67      | 251    | 3,793    | 428.1 | 529.26   |
|         | merged PRs             | 35      | 214    | 3,419    | 351.2 | 453.95   |
| PR      | closed time (day)      | 0.00001 | 0.65   | 1,144.93 | 8.41  | 34.97    |
|         | ratio of Java code     | 0.50    | 1.00   | 1.00     | 0.96  | 0.10     |
|         | modified lines of code | 1       | 81     | 349,967  | 1,593 | 9,161.72 |

to continuous response variables while GLMM is an extension of linear mixed-effects models that allows non-normal response variables (e.g., binary variables). Thus we accordingly built GLMMs for the merging of PRs (a binary response variable), and LMMs for the time cost of closing PRs (a numeric response variable). A mixed-effects model is a statistical model containing both fixed effects and random effects. The fixed effects estimate the population level coefficients which are constant across the population, while the random effects account for individual differences in response to an effect (Cohen et al. 2013). Correspondingly, to build a mixed-effects model, we need to specify fixed factors (with fixed effects) and random factors (with random effects).

Considering different projects may respond differently to the integration of PRs (e.g., some projects may be more conservative in merging PRs or tend to take a longer time to close PRs), to capture such a project-level variability in the response variable, we introduced *project ID* (from 1 to 117 for 117 projects) as a random factor when building mixed-effects models. Setting project ID as a random factor allowed the intercept to vary by projects rather than to be fixed for all projects. Code style inconsistency and 8 confounding factors (in Section 4.1.3) were taken as fixed factors.

Two widely used summary statistics, i.e., a marginal  $R^2$  ( $R_m^2$ ) and a conditional  $R^2$  ( $R_c^2$ ) (Nakagawa and Schielzeth 2013; Johnson 2014) are used to quantify the goodness-of-fit of a mixed-effects model (including LMM and GLMM).  $R_m^2$  describes the proportion of variance explained by the fixed factors alone while  $R_c^2$  describes the proportion of variance explained by the fixed and random factors together (more technical details of calculating  $R_m^2$  and  $R_c^2$  for both LMM and GLMM models can be found in Nakagawa and Schielzeth 2013). During our experiments, we used functions `lmer` and `glmer` in R package `lme4` (Bates 2010) to build the corresponding models. The  $R_m^2$  and  $R_c^2$  were obtained by function `r.squaredGLMM` in R package `MuMIn` (Bartoń 2013).

Based on the results of mixed-effects regression models, we further used Cohen's  $f^2$  to gauge the effect of code style inconsistency on PR integration. Cohen's  $f^2$  is a commonly-used effect size measure that allows us to evaluate the effect size of one variable within the context of a regression model (Selya et al. 2012). The calculation of *Cohen's  $f^2$*  is defined as

$$\frac{R_{AB}^2 - R_B^2}{1 - R_{AB}^2}$$

Here  $R_B^2$  is the variance accounted by variable set B; and  $R_{AB}^2$  is the variance accounted by variable sets B and A together. In this study, B is a set of fixed factors (i.e., eight confounding factors in Table 4) and the project ID random factor; while A is the code style inconsistency. Both  $R_B^2$  and  $R_{AB}^2$  refer to marginal  $R^2$  of a mixed-effects model. Values 0.02, 0.15, and 0.35 for  $f^2$  are suggested as a minimum value to determine that the effect is small, median, and large, respectively (Kabacoff 2015).

### 4.1.3 Confounding Factors

Existing works (Hellendoorn et al. 2015; Gousios et al. 2015; Tsay et al. 2014a) have revealed that three types of attributes (including the complexity of PRs, the maturity of projects, and the experience of developers) may affect PR integration. In our work, we used eight following metrics to represent the above three types of attributes, as the confounding factors in RQ3 and RQ4. Table 4 shows the summary of these eight confounding factors.

**Table 4** Summary of 8 confounding factors of 50,092 PRs on 117 projects

| Confounding factor | Min    | Median | Max   | Mean   | St. Dev. |
|--------------------|--------|--------|-------|--------|----------|
| pr_chgFileNum      | 1      | 4      | 8408  | 39.71  | 190.50   |
| pr_avgCmitSize     | 0.50   | 47.67  | 80473 | 283.55 | 1221.50  |
| proj_fork          | 0      | 120    | 4239  | 238    | 296.25   |
| proj_age (year)    | 0.0002 | 1.86   | 4.86  | 1.90   | 1.09     |
| proj_insider       | 0      | 0      | 11    | 0.52   | 1.67     |
| proj_openPR        | 0      | 6      | 106   | 12.14  | 15.64    |
| dev_issue          | 0      | 2      | 2677  | 26.38  | 97.46    |
| dev_mergedPR       | 0      | 20     | 797   | 69     | 115.55   |

**pr\_chgFileNum** The total number of modifying files within a PR. In general, one PR that changed several files for many times are complexity prone.

**pr\_avgCmitSize** The average number of changed lines over commits within a PR. This metric measures the complexity of commits, to some extent. Large commits may hinder the processing of PRs.

**proj\_fork** The number of forks from the project. This metric may indicate the popularity of a project and the popularity may affect the processing of PRs.

**proj\_age** The project age since its creation. An older project may indicate a more mature project. A mature project may tend to be cautious at merging new PRs and may have a pre-defined process to review PRs. This may leads to a long time cost of closing PRs.

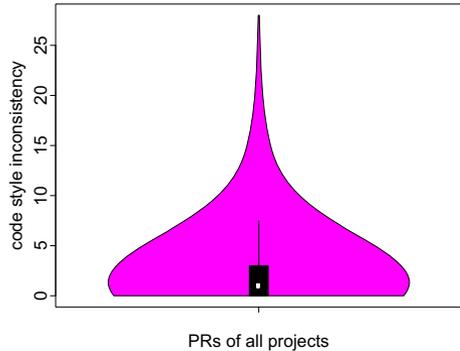
**proj\_insider** The number of insiders within the project before the submitted PR. PRs are often needed to be reviewed by insiders. If there are many insiders, maybe more PRs will be handled in a more effective and efficient way.

**proj\_openPR** The number of remained open PRs within the project before submitting the PR. All open PRs need to be handled by the project maintainers. More open PRs may delay the process of the newly submitted PR.

**dev\_issue** The number of issues reported by the developer before he/she submits the PR. A developer who reported more issues may indicate he/she is more active and concerned about the project. Maintainers might better respond to the PRs submitted by those developers.

**dev\_mergedPR** The number of merged PRs submitted by the developer before he/she submits the new PR. In general, the more merged PRs a developer has, the more experienced he/she is. PRs submitted by a more experienced developer might be more easily to be merged in a quicker way.

**Fig. 1** General statistics on the code style inconsistency for all 50,092 PRs of 117 projects. The first quantile, median, and third quantile values are 0, 1, and 3, respectively. The minimum, mean, and maximum values are 0, 2.12, and 28, respectively



#### 4.2 RQ1. Is there any Difference Between the Code Style of Submitted PRs and that of the Existing Source Code?

**Goal** This RQ investigated the general statistics of the code style inconsistency of PRs. We showed that there indeed existed differences of code style before and after the potential merging of PRs.

We used all 50,092 closed PRs on 117 projects to conduct our experiments. For each PR, we retrieved the **original** and **modified** files. Then we configured the tool *Checkstyle*<sup>20</sup> to get the code style of the **original** and **modified** files (i.e., the corresponding violations to 37 code criteria mentioned in Section 3.2). After that, we used the method mentioned in Section 3.3 to calculate the code style inconsistency between the PR and the existing code. Fig. 1 visualizes the code style inconsistency with a violin plot on all PRs of 117 projects.

As shown in Fig. 1, the minimal value of inconsistency was zero. This meant that, on the whole, a number of PRs kept a consistent code style with the existing code in the codebase. For each project, we further checked the minimal, median, and maximum values of code style inconsistency of all PRs, respectively. Table 5 shows the statistic details. From the table, we found that the minimum values of all projects were 0.0. This meant that all projects had some PRs that kept a consistent code style with existing code; 117 projects kept the median value no more than 3.0. That is, all projects showed that 50% PRs had inconsistent code style in at most three criteria; The maximum values of inconsistency in all projects ranged from 8 to 28, with a standard deviation of 4.63. This meant that all PRs kept consistent with the original code from 9 (= 37 - 28) to 29 (= 37 - 8) code criteria.

**Finding 1.** The code style inconsistency indeed existed in all projects under evaluation: All projects had a certain number of PRs that had inconsistent code style with existing code in the codebase; PRs in all projects showed inconsistency in at most 8 to 28 code style criteria. Meanwhile, all projects showed that 50% PRs had inconsistent code style in no more than three code style criteria.

To further understand the code style inconsistency in PRs, we calculated the ratio of PRs, which had no inconsistency with their existing code in the codebase. We found that the ratio of PRs with no inconsistency in code style varied in different projects. The least ratio

<sup>20</sup>Checkstyle is a highly configurable tool of checking code style. The code style by Google and Oracle are supported by the tool. In our experiment, we configured Checkstyle to check whether a piece of code violates 37 code style criteria.

**Table 5** Summary of code style inconsistency over 117 projects

| Inconsistency         | Min | Median | Max | Mean | St. Dev. |
|-----------------------|-----|--------|-----|------|----------|
| Minimum_inconsistency | 0   | 0      | 0   | 0    | 0        |
| Median_inconsistency  | 0   | 1      | 3   | 0.89 | 0.75     |
| Maximum_inconsistency | 8   | 19     | 28  | 18.7 | 4.63     |

The `minimum_inconsistency`, `median_inconsistency`, and `maximum_inconsistency` represent the minimum, median, and maximum code style inconsistency for all PRs of a project respectively

was 15.6% for the project “magefree/mage”<sup>21</sup> and the largest one was 79.3% in the project “CUTR-at-USF/OpenTripPlanner-for-Android”<sup>22</sup>. 50 projects had no more than 40% PRs that showed no inconsistency; 81 projects had no more than 50% PRs that kept consistent code style with existing code.

**Finding 2.** Different projects had different ratios of PRs with no inconsistency in code style. Many projects tended to have at most 50% PRs keeping a consistent style after changing code.

To get deeper insights into why the ratio of PRs with no inconsistency in code style varied in different projects, we further checked whether the policy of a project on code style or the experience of the PR authors would affect the ratio of PRs with no inconsistency in code style in the project.

Specifically, for the policy of a project on code style, we mainly concerned whether a project ever declared any code style requirements in its documentation or configured any code style checking tools within its codebase. To figure this out, for each project, we manually checked its *readme/contribution* files, wiki, official website (if exists) and configuration files. We found that 90 out of 117 projects had various requirements on code style. Among the top 20 projects that had the largest ratios of PRs with no inconsistency in code style, 17 projects had their own policy on code style. We did not, however, observe any obvious trends in the remaining 97 projects.

Then, we conducted Spearman correlation analysis to explore the correlation between PR authors’ experience and the ratios of PRs with no inconsistency in code style in different projects. Spearman correlation is widely used to check how well the relationship between two variables can be described by an arbitrary monotonic function. It also does not make any assumptions about the distributions of the data (Hauke and Kossowski 2011).

To conduct Spearman correlation analysis, for each PR, we first calculated its author’s experience at the moment this PR was submitted. Four proxy metrics were used to measure each PR author’s experience, i.e., the number of changed lines of all commits he/she contributed to the codebase (`cmitChgLineCnt`), the number of closed PRs (`cldPRCnt`), merged PRs (`mergedPRCnt`), and issues (`issueCnt`) he/she submitted to a project. Then, we took the average values of `cmitChgLineCnt`, `cldPRCnt`, `mergedPRCnt`, and `issueCnt` as the overall developer experience of the project. Last, we conducted Spearman correlation analysis between the ranked overall developer experience and the ratios of PRs with no inconsistency in code style.

<sup>21</sup> <http://github.com/magefree/mage>

<sup>22</sup> <http://github.com/CUTR-at-USF/OpenTripPlanner-for-Android>

We found that the Spearman correlation coefficient was  $-0.237$  for the average `cmitChgLineCnt` (with  $p$ -value= $0.0101$ ). This meant that there was a small negative correlation between the average `cmitChgLineCnt` and the ratio of PRs with no inconsistency in code style in a project at the significance level of  $p$ -value $<0.05$ . The Spearman correlation coefficients for the average `cldPRCnt`, `mergedPRCnt`, and `issueCnt` were  $0.035$ ,  $-0.021$ , and  $-0.008$  respectively, with corresponding  $p$ -values being  $0.7075$ ,  $0.8227$ , and  $0.933$ . Those  $p$ -values indicated that we were not able to reach any conclusion about whether there existed a monotonic relationship between these three metrics and the ratio of PRs with no inconsistency in code style in a project.

**Finding 3.** Top 20 projects with largest ratios of PRs with no inconsistency in code style generally had explicit requirements on code style. Meanwhile, the average changed lines of commits submitted by individual developers had a small negative correlation with the ratio of PRs with no inconsistency in code style in a project.

### 4.3 RQ2. Which Code Style Criteria Tend to Show Much Inconsistency Among PRs?

**Goal** We have observed that PRs were not always different from the original source code in *each* code style criterion. This inspired us to further investigate which code style criterion revealed higher differences. RQ2 could help developers understand the divergence of code style in practice.

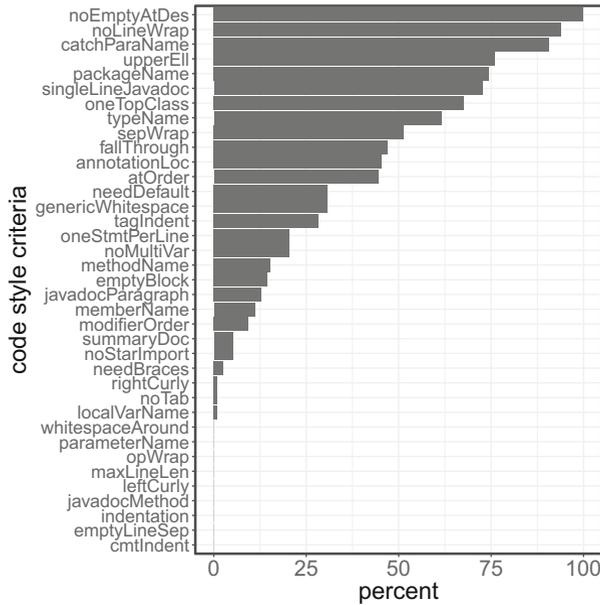
First, we explored which code style criterion never showed inconsistency. We calculated the inconsistency of each code criterion according to Section 3.3. Then we collected all code criteria, which never showed inconsistency in all PRs on one project.

Similarly, we used *Checkstyle* to get the violation results in each code criterion for all 50,092 PRs in 117 projects. Figure 2 shows the code style criteria, which never show inconsistency on one project. The vertical axis represents code criteria, which have no inconsistency in all PRs in each project; the horizontal axis is the ratio of projects over all 117 projects.

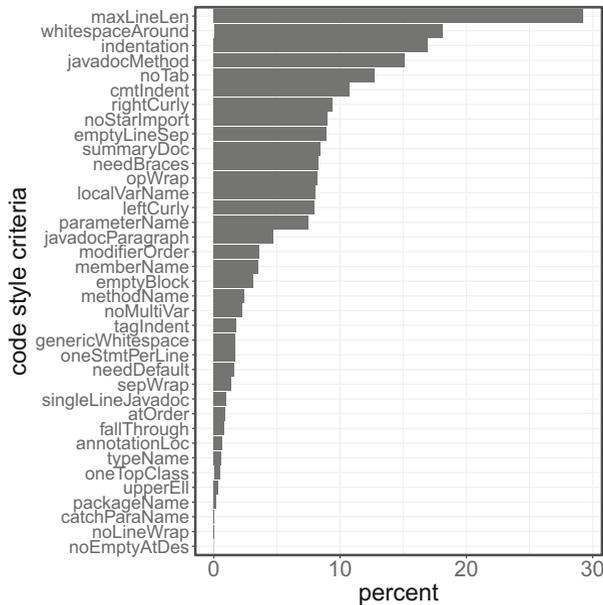
As shown in Fig. 2, nine code style criteria, i.e., from `noEmptyAtDes` to `sepWrap`, showed no inconsistency in over half of the projects. This fact indicated that many developers tended to agree with each other on these nine criteria. In particular, the code criterion `noEmptyAtDes` (i.e., a description is needed after `@` tag) showed no inconsistency in all projects. The other two criteria that showed no inconsistency over most projects were `noLineWrap` (i.e., import and package statements cannot be line-wrapped; that is, such statements should be in one line) and `catchParaName` (i.e., the parameter name of the catch code should be written in lowerCamelCase or in one-character lowercase). From Fig. 2, we also found that nine criteria, i.e., from `whitespaceAround` to `cmtIndent`, never showed consistency over all PRs of specific project. In other words, each project had at least one PR that violated these criteria.

**Finding 4.** Developers kept no inconsistency on the criteria “`noEmptyAtDes`”, “`noLineWrap`”, and “`catchParaName`” in most projects.

We continued to investigate how these code style criteria revealed differences in PRs. Specifically, we counted the number of PRs with code style inconsistency on each code style criterion, and then ranked these criteria based on their ratios of inconsistent PRs over all the 50,092 PRs. Figure 3 visualizes the results, where the vertical axis represents code



**Fig. 2** Percentage of projects where code style criteria show no inconsistency in any PR. For instance, the bar of the code style metric “noLineWrap” indicates that 94.01% (110 out of 117) projects have no inconsistent PRs, each of which tends to not violate this code style criterion



**Fig. 3** Code style criteria with the number of inconsistent PRs

style criteria and the horizontal axis displays the ratio of inconsistent PRs over all the 50,092 PRs. As shown in Fig. 3, six criteria (i.e., `maxLineLen` to `cmtIndent`) showed inconsistency over 10% PRs, among which `maxLineLen` showed the largest inconsistency (with almost 30% inconsistent PRs). This indicated that developers tended to behave differently in the ways of using spaces or indents, making comments, and writing code lines with a suitable length. Besides, we could observe that some criteria (such as `noLineWrap`, `catchParaName`, etc.) which always showed no inconsistency in specific projects (Finding 4), on the whole, also had the least inconsistency over all the 50,092 PRs.

**Finding 5.** Six code style criteria, i.e., “`maxLineLen`”, “`whitespaceAround`”, “`indentation`”, “`javadocMethod`”, “`noTab`”, and “`cmtIndent`”, showed most inconsistency among all PRs. This means developers tended to differ with others in using spaces or indents, making comments, and writing code lines with a suitable length.

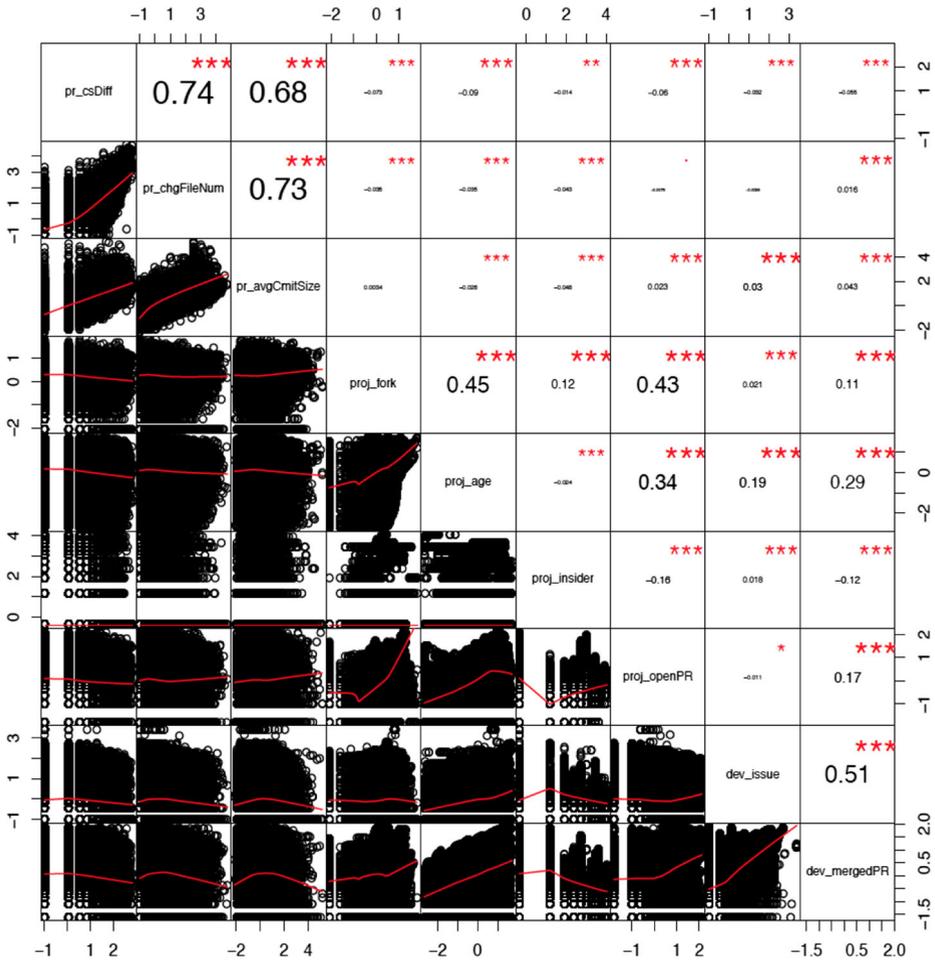
#### 4.4 RQ3. How Does Code Style Affect the Merging of PRs?

**Goal** Developers want to see their contributions accepted. However, many PRs are rejected without being merged into a project. As mentioned in Section 2.2, the code style is considered as a key factor that affects the acceptance of a PR. In this section, we quantitatively investigated the correlation between the code style of PRs and the decision of merging PRs.

Among 50,092 closed PRs, 38,169 PRs were explicitly displayed as merged in GitHub, with 11,923 PRs as unmerged. However, as mentioned by Gousios et al. (2014), some PRs are displayed as unmerged in GitHub even when they are actually merged. They proposed four heuristic rules to find merged PRs that are shown as unmerged in GitHub. In this study, we also applied these heuristic rules to 11,923 unmerged PRs. Specifically, besides commits within PRs, we also downloaded the codebase of each project and crawled all comments made on PRs (these data were necessary for applying the heuristic rules). After that, we programmed to obtain PRs that met these heuristic rules. As a result, we got a list of 3,213 merged PR candidates. Then we manually checked each candidate, and found that 2,921 unmerged PRs were actually merged. This meant that our dataset included 41,090 (38169+2921) merged PRs and 9,002 unmerged PRs.

As explained in Section 4.1.2, we decided to build generalized mixed-effects models (GLMM) to analyze the effect of code style inconsistency on the merging of PRs. In GLMM, the response variable was the merging status of a closed PR. The code style inconsistency and eight confounding factors introduced in Section 4.1 were taken as fixed factors; while project ID was set as a random factor. We used the method introduced in Section 3.3 to calculate the code style inconsistency for each PR. The fixed factors were log transformed (i.e., using  $\log(x)$ ) and then centered by using the scale function in R. This could not only decrease multicollinearity among explanatory variables and between random intercepts and slopes, but also make all explanatory variables relatively comparable (Cohen et al. 2013; Jaeger 2011).

Furthermore, considering the inherent collinearity of explanatory variables (i.e., two variables are correlated) would potentially threaten their statistical and inferential interpretation (Jiarpakdee et al. 2018), we further checked the correlation between explanatory variables before model building. Fig. 4 presents the detailed Pearson correlation results. From the figure, we found that, some variables (e.g., the code style inconsistency (`pr_csDiff`) and `pr_chgFileNum`) were indeed (highly) correlated. This indicated a necessity to mitigate the



**Fig. 4** Pearson correlation between explanatory variables. The bottom part of the diagonal displays the bivariate scatter plots and a fitted line. The top part of the diagonal shows the value of the correlation plus the significance level as stars ('\*\*\*'  $p < 0.001$ , '\*\*'  $p < 0.01$ , '\*'  $p < 0.05$ ). The horizontal and vertical axes represent a pair of two explanatory variables whose values were log transformed and centered before correlation calculation

collinearity among variables during model building so as to properly evaluate individual effects within the model. One easy way to avoid collinearity is to directly drop correlated variables (Jiarpakdee et al. 2018). However, dropping variables would not only reduce model explanatory power (the unique contributions of the dropped variables are ignored) but also bring the problem of choosing which variables to drop. We instead adopted another widely-used strategy, i.e., residualization, to mitigate the collinearity problem (Graham 2003; Cohen-Goldberg 2012; Lemhöfer et al. 2008).

The idea of residualization is to regress a variable against another correlated variable, and replace the variable with the residuals from the regression (the residuals for the variable represents the part that could not be explained by its correlated variable) (Graham 2003). Since *pr\_csDiff* is the factor of interest in this study, we chose to residualize confounding

factors against `pr_csDiff` if they were found to be correlated with `pr_csDiff`. Specifically, we did residualization on any confounding factor whose Pearson correlation coefficient with `pr_csDiff` is larger than 0.1. We chose the threshold of 0.1 mainly because (1) low level of collinearity would also potentially bias the analysis (Graham 2003), and (2) 0.1 is thought to represent a small association according to Cohen's guide (Cohen 1977). Based on this threshold, we residualized two confounding factors against `pr_csDiff`, i.e., `pr_chgFileNum` and `pr_avgCmitSize` whose Pearson correlation coefficients with `pr_csDiff` were 0.74 and 0.68 respectively.

Note that we only considered the correlation between confounding factors and `pr_csDiff` during residualizations. If two confounding factors were found to be correlated (with coefficient > 0.1), we did not perform corresponding residualizations. We made this decision mainly because ignoring the correlation between confounding factors (1) would not affect our analysis of the effect of code style inconsistency on PR integration (Jaeger 2011), (2) would not affect the fitness of the regression model (Cohen et al. 2013), and (3) would simplify our model and make our model relatively more understandable (Graham 2003).

To calculate the Cohen's  $f^2$  that measured the effect size of `pr_csDiff` on PR integration, we first built a full model with both `pr_csDiff` and 8 confounding factors (among which `pr_chgFileNum` and `pr_avgCmitSize` were residualized against `pr_csDiff`). Then we built another model that excluded `pr_csDiff`. Tables 6 and 7 show the results of the regression models on the merging of PRs with and without code style inconsistency.

From Table 6, we found that the fixed factors (i.e., the code style inconsistency and eight confounding factors) could explain 17% of the variance ( $R_m^2=17\%$ ) while the value for combined fixed-and-random factors (i.e., fixed factors and the random project factor) was 31% ( $R_c^2 = 31\%$ ), indicating that project-to-project variability contributed to some variability that were explained.

In Table 6, we could observe that the factor code style inconsistency (i.e., `pr_csDiff`) had a significant (p-value < 0.001), negative (the coefficient < 0) effect on the merging of PRs. Besides, combined with the results of Table 7, we could further find that the model (in Table 6) with `pr_csDiff` accounted for 2.1% (i.e., 0.312 - 0.291) more variance than that of the model without `pr_csDiff` (in Table 7). Based on the  $R_c^2$  results, we could compute the effect size of code style inconsistency on the merging of PRs as  $(0.312 - 0.291)/(1 - 0.312) = 3.11\%$ . This effect size was small according to Cohen's criteria (with a minimal value 2% for a small effect; 15% or greater for a medium effect; and 35% or greater for a large effect) (Cohen 1977).

**Finding 6.** The code style inconsistency of PRs had a small negative effect on the decision of merging PRs. This meant that a PR with larger code style inconsistency was more likely to be rejected.

We have found that the overall code style inconsistency of PRs would negatively affect the decision of merging PRs (Finding 6). Our next step was to further investigate the effect of individual code style criterion on the merging of PRs. This could help practitioners be more careful about certain code style criteria (that would affect PRs acceptance) to more easily get a PR acceptance.

In our study, each code style criterion was a binary variable that had only two values, i.e., 0 and 1. 0 means that a PR kept consistent with the codebase in a certain code style criterion while 1 means that a PR was inconsistent with the codebase in the criterion (the calculation of the values of each criterion could be found in Section 3.3). To properly evaluate the effect of individual code style criterion, like in Table 6, for each criterion, we also built a

**Table 6** GLMM regression model with code style inconsistency and 8 confounding factors on the decision of merging PRs

| Variable       | Coeffs (Errors) | z value | Pr (>  z )   |
|----------------|-----------------|---------|--------------|
| (Intercept)    | 1.976 (0.081)   | 24.53   | < 2e-16 ***  |
| pr_csDiff      | -0.276 (0.013)  | -20.58  | < 2e-16 ***  |
| pr_chgFileNum  | -0.061 (0.022)  | -2.79   | 0.005 **     |
| pr_avgCmitSize | -0.139 (0.021)  | -6.77   | 1.32e-11 *** |
| proj_fork      | -0.075 (0.025)  | -3.04   | 0.002 **     |
| proj_age       | -0.059 (0.024)  | -2.42   | 0.016 *      |
| proj_insider   | 0.038 (0.034)   | 1.10    | 0.272        |
| proj_openPR    | -0.321 (0.025)  | -12.70  | < 2e-16 ***  |
| dev_issue      | -0.068 (0.021)  | -3.19   | 0.001 **     |
| dev_mergedPR   | 0.874 (0.020)   | 44.46   | < 2e-16 ***  |

\*\*\*\*' p<0.001    \*\*\*' p<0.01    \*\*' p<0.05  
Akaike's Information Criterion (AIC): 37951.9    Log Likelihood: -18964.9

$$R_m^2 = 0.168. R_c^2 = 0.312$$

full GLMM model which took both eight confounding factors and the criterion as fixed factors and project id as the random factor. During model building, the continuous confounding factors were first log transformed and centered and then were residualized against the criterion if their Pearson coefficients with the criterion were larger than 0.1; the binary criterion was directly entered into the model. After we built the full model, we built another GLMM model that excluded the criterion from the full model. Both models took the merging status of a PR as their response variable. At last, for each criterion, we computed its effect size

**Table 7** GLMM regression model with *only* 8 confounding factors on the decision of merging PRs

| Variable       | Coeffs (Errors) | z value | Pr (>  z )   |
|----------------|-----------------|---------|--------------|
| (Intercept)    | 1.954 (0.077)   | 25.32   | < 2e-16 ***  |
| pr_chgFileCnt  | -0.078 (0.022)  | -3.49   | 0.000 ***    |
| pr_avgCmitSize | -0.140 (0.020)  | -6.78   | 1.16e-11 *** |
| proj_fork      | -0.063 (0.024)  | -2.57   | 0.010 *      |
| proj_age       | -0.045 (0.024)  | -1.85   | 0.065 .      |
| proj_insider   | 0.043 (0.033)   | 1.26    | 0.209        |
| proj_openPR    | -0.305 (0.025)  | -12.18  | < 2e-16 ***  |
| dev_issue      | -0.071 (0.021)  | -3.38   | 0.0007 ***   |
| dev_mergedPR   | 0.883 (0.019)   | 45.20   | < 2e-16 ***  |

\*\*\*\*' p<0.001    \*\*\*' p<0.01    \*\*' p<0.05  
Akaike's Information Criterion (AIC): 38372.9    Log Likelihood: -19176.5

$$R_m^2 = 0.155. R_c^2 = 0.291$$

(i.e., Cohen’s  $f^2$ ) based on the results of two GLMM models built above for the criterion. Table 8 shows the effect size of the inconsistency of each code style criterion on the merging of PRs.

**Table 8** The effect size (i.e., Cohen’s  $f^2$ ) of individual code style criterion on the decision of merging PRs

| Variable          | Effect direction | Effect size (Level) | Significance |
|-------------------|------------------|---------------------|--------------|
| annotationLoc     | Negative         | 0.00 (negligible)   |              |
| atOrder           | Negative         | 0.11 (negligible)   | ***          |
| catchParaName     | Negative         | 0.02 (negligible)   |              |
| cmtIndent         | Negative         | 1.67 (negligible)   | ***          |
| emptyBlock        | Negative         | 0.25 (negligible)   | ***          |
| emptyLineSep      | Negative         | 0.75 (negligible)   | ***          |
| fallThrough       | Negative         | 0.09 (negligible)   | ***          |
| genericWhitespace | Negative         | 0.24 (negligible)   | ***          |
| indentation       | Negative         | 1.21 (negligible)   | ***          |
| javadocMethod     | Negative         | 1.03 (negligible)   | ***          |
| javadocParagraph  | Negative         | 0.44 (negligible)   | ***          |
| leftCurly         | Negative         | 0.35 (negligible)   | ***          |
| localVarName      | Negative         | 0.62 (negligible)   | ***          |
| maxLineLen        | Negative         | 2.01 (small)        | ***          |
| memberName        | Negative         | 0.20 (negligible)   | ***          |
| methodName        | Negative         | 0.04 (negligible)   | ***          |
| modifierOrder     | Negative         | 0.31 (negligible)   | ***          |
| needBraces        | Negative         | 0.64 (negligible)   | ***          |
| needDefault       | Negative         | 0.07 (negligible)   | ***          |
| noLineWrap        | Positive         | 0.00 (negligible)   |              |
| noMultiVar        | Negative         | 0.08 (negligible)   | ***          |
| noStarImport      | Negative         | 0.56 (negligible)   | ***          |
| noTab             | Negative         | 2.53 (small)        | ***          |
| oneStmtPerLine    | Negative         | 0.42 (negligible)   | ***          |
| oneTopClass       | Negative         | 0.00 (negligible)   |              |
| opWrap            | Negative         | 0.64 (negligible)   | ***          |
| packageName       | Negative         | 0.00 (negligible)   |              |
| parameterName     | Negative         | 0.59 (negligible)   | ***          |
| rightCurly        | Negative         | 0.68 (negligible)   | ***          |
| sepWrap           | Negative         | 0.27 (negligible)   | ***          |
| singleLineJavadoc | Negative         | 0.06 (negligible)   | ***          |
| summaryDoc        | Negative         | 0.76 (negligible)   | ***          |
| tagIndent         | Negative         | 0.32 (negligible)   | ***          |
| typeName          | Negative         | 0.03 (negligible)   | *            |
| upperEll          | Negative         | 0.01 (negligible)   |              |
| whitespaceAround  | Negative         | 1.47 (negligible)   | ***          |

‘\*\*\*’  $p < 0.001$     ‘\*\*’  $p < 0.01$     ‘\*’  $p < 0.05$

Minimum Value for Effect Size: Small: 2%, Medium: 15%, Large: 35%

Effect Direction represents whether an effect is positive or negative. The values of Effect Size are displayed in a percent format (e.g., 2.01 in the table means 2.01%)

From the table, we found that 31 out of 37 code style criteria had a significant (marked with one or multiple asterisks) negative correlation with the merging of PRs. Among these 31 criteria, two criteria, i.e., `noTab` and `maxLineLen`, were found to have a larger negative correlation with the decision of merging PRs than the other criteria: the effect sizes of `noTab` and `maxLineLen` were small while the effect sizes of the remaining criteria were negligible according to Cohen's criteria (Cohen 1977). This indicates that developers may need to pay attention to the usage of tabs and the length limit on code lines of a target project if they want to have their contributions more easily got accepted. For the criteria that were not statistically significant (6 out of 37), unfortunately, not much conclusion can be drawn.

**Finding 7.** Among 31 out of 37 code style criteria that significantly negatively affected the merging of PRs, two criteria namely “noTab” and “maxLineLen” had a relatively larger effect on the decision of merging PRs compared to the other criteria. The results were inconclusive for the remaining 6 code style criteria.

#### 4.5 RQ4. How Does Code Style Affect the Closing Time of PRs?

**Goal** Developers also care about the processing time of the PRs. In this section, we leverage the time cost of closing PRs to evaluate the progress of handling submitted PRs. A short time cost of closing PR implies that the maintainer quickly responds to the contribution by the developer. Different from the binary decision of the merging status in Section 4.4, the time cost of closing PR is a numeric value, which can further characterize the correlation between the code style and the PR integration.

We collected the time cost of closing PRs for all 50,092 closed PRs in 117 projects. The time cost of closing a PR was calculated as the time difference between the timestamp of closing the PR and the timestamp of submitting the PR in days. Then we used the R package `lme4` to build a mixed-effects model, to measure how code style inconsistency would affect the time cost to close PRs. Within the model, the time cost of closing PRs was taken as the response variable. The code style inconsistency (i.e., `pr_csDiff`) together with eight confounding factors were taken as fixed factors. The project id was set as a random factor. Like in RQ3, we log transformed and centered the fixed factors and the response variable during model building. We further residualized two confounding factors (i.e., `pr_chgFileNum` and `pr_avgCmitSize`) whose Pearson correlation coefficients with `pr_csDiff` is larger than 0.1. After we built the model described above, we built another model that excluded `pr_csDiff` from the previously built model. Inspection of residual plots showed that our dataset for LMM did not reveal obvious deviations from normality. The effect size of code style inconsistency was then computed based the results of the two models.

Tables 9 and 10 show the final results. From Table 9, we found that the fixed factors alone explained 13% ( $R_m^2=13\%$ ) variance of the closing time of PRs; while the combined fixed-and-random factors could explain 23% ( $R_c^2=23\%$ ) variance of the closing time of PRs. This indicates that the time in processing PRs indeed varied from project to project. We also found that the code style inconsistency (i.e., `pr_csDiff`) had a significant ( $p\text{-value}<0.001$ ), positive effect (the coefficient $>0$ ) on the time cost of closing PRs. Meanwhile, we observed that the inclusion of `pr_csDiff` could make the model explain 3.8% more variance (the percentage of explained variance rose from 19% in Table 10 to 22.8% in Table 9). Correspondingly, the effect size of `pr_csDiff` was  $(0.228 - 0.190)/(1 - 0.228) = 4.82\%$ , which indicated a small effect according to Cohen's criteria (the range of 2% - 15% corresponds

**Table 9** Linear mixed-effects model with code style inconsistency and 8 confounding factors on the closing time of PRs

| Variable       | Coeffs (Errors) | z value | Pr (>  z  ) |
|----------------|-----------------|---------|-------------|
| (Intercept)    | 0.040 (0.029)   | 1.407   | 0.162       |
| pr_csDiff      | 0.188 (0.004)   | 46.469  | < 2e-16 *** |
| pr_chgFileCnt  | 0.064 (0.007)   | 9.461   | < 2e-16 *** |
| pr_avgCmitSize | 0.072 (0.006)   | 11.446  | < 2e-16 *** |
| proj_fork      | 0.024 (0.008)   | 3.055   | 0.002 **    |
| proj_age       | 0.094 (0.008)   | 12.198  | < 2e-16 *** |
| proj_insider   | -0.024 (0.012)  | -2.039  | 0.041 *     |
| proj_openPR    | 0.198 (0.008)   | 26.246  | < 2e-16 *** |
| dev_issue      | -0.018 (0.006)  | -2.820  | 0.005 **    |
| dev_mergedPR   | -0.219 (0.006)  | -35.733 | < 2e-16 *** |

\*\*\*' p<0.001    '\*\*' p<0.01    '\*' p<0.05

$R_m^2 = 0.134$ .  $R_c^2 = 0.228$

to a small effect) (Cohen 1977). This means that the larger the code style inconsistency of a PR is, the more time it is likely to take to close the PR. It would be necessary for developers to keep a consistent code style with the codebase if they want to see their PRs got processed in a quicker way.

**Finding 8.** The code style inconsistency of PRs had a small positive effect on the time cost of closing PRs. This means a PR with larger code style inconsistency tended to need more time to be closed.

Similar to RQ3, we also built linear mixed-effects models to explore how the inconsistency of each code style criterion would affect the time cost of closing PRs. Similarly, for each code style criterion, we built two linear mixed-effects models with including and

**Table 10** Linear mixed-effects model with *only* 8 confounding factors on the closing time of PRs

| Variable       | Coeffs (Errors) | z value | Pr (>  z  ) |
|----------------|-----------------|---------|-------------|
| (Intercept)    | 0.038 (0.028)   | 1.337   | 0.184       |
| pr_chgFileCnt  | 0.067 (0.007)   | 9.692   | <2e-16 ***  |
| pr_avgCmitSize | 0.071 (0.006)   | 11.125  | <2e-16 ***  |
| proj_fork      | 0.017 (0.008)   | 2.081   | 0.037 *     |
| proj_age       | 0.087 (0.008)   | 11.051  | <2e-16 ***  |
| proj_insider   | -0.028 (0.012)  | -2.341  | 0.019 *     |
| proj_openPR    | 0.190 (0.008)   | 24.686  | <2e-16 ***  |
| dev_issue      | -0.013 (0.006)  | -2.083  | 0.037 *     |
| dev_mergedPR   | -0.233 (0.006)  | -37.292 | <2e-16 ***  |

\*\*\*' p<0.001    '\*\*' p<0.01    '\*' p<0.05

$R_m^2 = 0.099$ .  $R_c^2 = 0.190$

excluding the criterion respectively. During model building, 8 confounding factors and the to-be-evaluated code style criterion (if included) were taken as fixed factors, and project id was set as the random factor. Meanwhile, 8 confounding factors and the response variable (i.e., the time cost of closing PRs) were log transformed and centered during model building. The to-be-evaluated criterion which was already a binary variable was directly entered into the model. Like in RQ3, we further residualized any confounding factor against the to-be-evaluated criterion if there existed a correlation between them (i.e., the Pearson correlation coefficient  $> 0.1$ ). The residualized confounding factors then worked as explanatory variables in both two models. After building two models for each criterion, we then computed the effect-size (i.e., the Cohen's  $f^2$ ) of each criterion based on the variances explained by these two models. Table 11 shows the effect size of each code style criterion.

From the table, we found that the coefficients of 34 (out of 37) code style criteria were statistically significant (marked with one or multiple asterisks). Further according to Cohen's criteria (Cohen 1977), javadocMethod and maxLineLen had a small positive effect on the closing time of PRs, while the effects of the remaining 32 criteria were negligible. For the criteria that were not statistically significant, unfortunately, we were not able to reach any conclusions from them.

**Finding 9.** Two code style criteria namely “javadocMethod” and “maxLineLen” had a relatively larger positive effect on the closing time of PRs than the other criteria with significant effects.

## 5 Discussion

In this section, we first discuss the implications of our findings in Section 5.1. Then we list the main threats to the validity of our work in Section 5.2.

### 5.1 Implications

We briefly present some implications of our findings on developers' activities and future research.

**Developers should always keep consistent with the current code style** Through RQ3 and RQ4, we have found that code style inconsistency of PRs would affect PR integration, including both the merging of PRs and the time cost of closing PRs. Prior to our work, Gousios et al. (2015) also reported that code style was greatly considered by PR integrators; with the help of natural language models, Hellendoorn et al. (2015) found that accepted PRs were indeed more similar to the project than rejected ones (more details in Section 6.2). In this regard, our result reinforced and provided more quantitative support for their work. These findings indicate that developers should try their best to keep the same code style with the existing code while contributing. In this way, they are more likely to get their contributions accepted in a quicker way. For example, they may had better familiarize themselves with the policy of a project on code style before contributing and if possible run some Checkstyle-like tools or specific tools configured by the project to eliminate inconsistent code. They also need to pay special attention to the way they use spaces/indents, make comments, and write suitable-length code lines since these style criteria are more likely to affect PR integration.

**Table 11** The effect size (i.e., Cohen's  $f^2$ ) of individual code style criterion on the closing time of PRs

| Variable          | Effect direction | Effect size (Level) | Significance |
|-------------------|------------------|---------------------|--------------|
| annotationLoc     | Positive         | 0.00 (negligible)   | *            |
| atOrder           | Positive         | 0.20 (negligible)   | ***          |
| catchParaName     | Positive         | 0.00 (negligible)   |              |
| cmtIndent         | Positive         | 1.19 (negligible)   | ***          |
| emptyBlock        | Positive         | 0.32 (negligible)   | ***          |
| emptyLineSep      | Positive         | 1.08 (negligible)   | ***          |
| fallThrough       | Positive         | 0.05 (negligible)   | ***          |
| genericWhitespace | Positive         | 0.25 (negligible)   | ***          |
| indentation       | Positive         | 1.78 (negligible)   | ***          |
| javadocMethod     | Positive         | 2.35 (small)        | ***          |
| javadocParagraph  | Positive         | 1.00 (negligible)   | ***          |
| leftCurly         | Positive         | 0.81 (negligible)   | ***          |
| localVarName      | Positive         | 1.13 (negligible)   | ***          |
| maxLineLen        | Positive         | 3.49 (small)        | ***          |
| memberName        | Positive         | 0.51 (negligible)   | ***          |
| methodName        | Positive         | 0.35 (negligible)   | ***          |
| modifierOrder     | Positive         | 0.39 (negligible)   | ***          |
| needBraces        | Positive         | 0.73 (negligible)   | ***          |
| needDefault       | Positive         | 0.26 (negligible)   | ***          |
| noLineWrap        | Negative         | 0.00 (negligible)   |              |
| noMultiVar        | Positive         | 0.22 (negligible)   | ***          |
| noStarImport      | Positive         | 0.68 (negligible)   | ***          |
| noTab             | Positive         | 1.22 (negligible)   | ***          |
| oneStmntPerLine   | Positive         | 0.24 (negligible)   | ***          |
| oneTopClass       | Positive         | 0.07 (negligible)   | **           |
| opWrap            | Positive         | 1.11 (negligible)   | ***          |
| packageName       | Positive         | 0.02 (negligible)   | **           |
| parameterName     | Positive         | 1.11 (negligible)   | ***          |
| rightCurly        | Positive         | 1.29 (negligible)   | ***          |
| sepWrap           | Positive         | 0.22 (negligible)   | ***          |
| singleLineJavadoc | Positive         | 0.11 (negligible)   | ***          |
| summaryDoc        | Positive         | 1.61 (negligible)   | ***          |
| tagIndent         | Positive         | 0.30 (negligible)   | ***          |
| typeName          | Positive         | 0.10 (negligible)   | ***          |
| upperEll          | Positive         | 0.00 (negligible)   |              |
| whitespaceAround  | Positive         | 1.49 (negligible)   | ***          |

‘\*\*\*’  $p < 0.001$     ‘\*\*’  $p < 0.01$     ‘\*’  $p < 0.05$

Minimum Value for Effect Size: Small: 2%, Medium: 15%, Large: 35%

Effect Direction represents whether an effect is positive or negative. The values of Effect Size are displayed in a percent format (e.g., 2.35 in the table means 2.35%)

**Maintainers may need to take actionable strategies to help contributors better identify the code style of projects and format their inconsistent code** Despite the code style is concerned a lot by project maintainers (Gousios et al. 2015), the RQ1 results, however, revealed that there still existed a number of PRs that violated the current code style of projects. As project maintainers, they may also need to make more efforts to help their (potential) contributors keep a consistent code style. For example, they can explicitly and detailedly declare their code style requirements in some noticeable places (e.g., readme/contributing files) in their GitHub websites. Besides, it would be valuable for them to integrate some code style checking tools (such as Checkstyle or Eclipse built-in Formatter) into their daily-routine development to help automatically detect and further format inconsistent code whenever contributors submit their PRs or add new commits to PRs.

**More efforts are needed to identify the usage patterns for some code style criteria that are cared much by developers** In RQ2, we observed that some code style criteria tended to show more inconsistency. Specifically, we found that developers differed most with others on the way of using whitespace, indenting code, writing proper Javadoc comments, and writing less-lengthy code. This, to some extent, coincides with an existing study that investigated the problems found during code review: 75% problems figured out by reviewers were non-functional problems, among which, many problems were related to comments, indentation, space usage, and long line (Mäntylä and Lassenius 2009). For those criteria, it would be very helpful if some approaches can be proposed to automatically obtain their usage patterns within the codebase. Ideas from existing studies (Allamanis et al. 2014; Hellendoorn et al. 2015) that built N-gram language models to infer the stylistic aspects/conventions of code might help in this direction. Additionally, the pervasiveness of problems related to these criteria in code review and the ability of Checkstyle to check the violations of these criteria indicates that it would be very helpful for practitioners to embed Checkstyle-like tools into their code review process (as reported in Sadowski et al. (2018), Checkstyle has been deployed as part of the general developer workflow in Google).

**Further investigation into the effect of code style inconsistency on software maintenance is needed** In this paper, we highlighted the effect of code style inconsistency on PR integration. However, we have not looked into the potential effect of code style inconsistency on software maintenance. For example, is there any correlation between code style inconsistency and code quality? For another instance, how much effort did developers put into resolving problems related to inconsistent code style? It would be interesting to conduct further investigations into these problems. Such investigations can not only help us obtain a better understanding about the code style, but also can help us find more potential concerns by developers in maintaining a consistent code style.

**Some replication studies on projects developed in other languages are encouraged** In this study, we mainly focus on exploring the code style inconsistency problem among Java projects. We still have little knowledge about whether our conclusions are applicable to projects developed in other languages (such as python, javascript, etc.). We believe that it would be valuable to replicate our study in other languages. This would not only help us gain more insights into the pervasiveness of code style related problems in a big picture, but also may provide some hints on how we should focus our efforts to solve these problems in the future.

## 5.2 Threats to Validity

**Internal Validity** In this study, we only used **original** and **modified** files involved within a PR to show the code style of PRs. However, there exist many other ways to extract code to calculate the code style. One potential way is to compare the code style between a new PR and the whole codebase. However, the codebase is always much larger and may contain different kinds of code styles. This leads to a complex result in calculation, which may involve other factors that affect the analysis. Another potential way is to compare the code style between a new PR and a maintainer. However, determining the code style of a maintainer is difficult since we cannot identify the code style of a maintainer, even knowing his merged PRs.

Besides, as mentioned in Section 3.3, we calculated the code style inconsistency based on binary values, which represented whether **original** or **modified** files violated a certain code style criterion. We note that a better way of characterizing the code style is to use numeric values, e.g., from 0 to 1. In our work, we focused on the difference between the code style of **original** and **modified** files involved within a PR. Thus, we used a simple way, i.e., the binary values for the code style inconsistency.

The binary values we measured code style inconsistency may cause some inconsistency in certain code style criteria to be counted as no inconsistency. For example, if a project uses 140 character line length, and a new PR uses 200 character line length, in this case, since both of them would differ from the criterion in Table 2 (100 character line length), our approach would take them as consistent in the criterion despite they are actually inconsistent. Such cases may pose a potential threat to our analysis. However, since many projects did not mention their concrete requirements on specific code style criteria, we decided to use a global code style (that was developed by retrieving the common subset (in Table 2) of Google and Oracle Java code styles) for all experimental projects. The popularity of Google and Oracle Java code styles makes us believe that such a global code style could still shed some light on the issue of how code style inconsistency would affect PR integration. We would try to measure and eliminate this threat in our follow-up studies.

**External Validity** In this paper, we only considered Java projects on GitHub. We cannot guarantee that our findings can be generalized to projects in another language on GitHub. Besides, we are not sure whether the results can also be applicable to other OSS platforms, such as SourceForge<sup>23</sup> and BitBucket<sup>24</sup>. In addition, as we only conducted experiments on OSS projects, the conclusions may not hold on industrial projects. Whereas, considering the great popularity of GitHub platform, we believe that our quantitative study can, to some extent, help developers understand the effect of code style on PR integration. In future, we plan to conduct experiments on more projects to explore the generalization problem.

## 6 Related Work

### 6.1 Code Style Definition

Defining an appropriate code style is a challenging topic in research community and industry. Early research work mainly focused on defining a “good” code style for the language

---

<sup>23</sup><http://sourceforge.net>

<sup>24</sup><http://bitbucket.org/>

of Pascal, C, and C++ (Marca 1981; Rees 1982; Berry and Meekings 1985; Bridger and Pisano 2001). Oman and Cook (1990) proposed a taxonomy for programming style to help people hold a coherent view on the basis and application of programming styles. McConnell (1993) presented many useful tips and explanations about the style of programming. For the Java language, which we mainly focused on in this paper, researchers also defined some basic elements of the code style. Vermeulen (2000) provided many code style guidelines. Two kinds of Java code styles are widely used in software development in practice: one is the code style by Oracle, proposed by the original inventors of Java (Oracle 1999); the other is by (Google 2013). These two kinds of Java code style have many terms in common, but contain some differences. For example, the Google code style prefers to use two spaces to align code while the Oracle code style suggests four spaces instead.

In this paper, we manually collected existing code styles to model the differences of code style between the change in a PR and the previous code before merging the PR. We focused on the inconsistency between the code style of PR and its existing code, not the definition of a “good” code style. That is, defining a good code style with using two spaces (such as in Google) or four spaces (such as in Oracle) would not affect the findings in our experiments.

## 6.2 Code Style Inconsistency

In a qualitative study, Gousios et al. (2015) reported that code style would greatly affect the acceptance of contributions. To quantitatively evaluate the effect, Hellendoorn et al. (2015) built natural language models to capture the stylistic properties of code and used mean entropy to measure the style similarity between PRs and the project code. They found that accepted PRs were more similar with the project than rejected PRs. This study is closest to our work.

Different from the work of Hellendoorn et al., we measured code style inconsistency based on 37 concrete code style criteria rather than on the abstract entropy. Despite less generic than their model, our model is, however, more explainable in that developers can understand in what aspects they did not conform to the project style and can easily take corresponding actions to format the code with inconsistent code style based on our model results. Besides the merging of PRs, we also studied how code style inconsistency would affect the closing time of PRs. This extends our knowledge on how code style inconsistency would affect the processing of PRs. What is more, we further explored the inconsistency of individual code style criterion and their potential effect on PR integration. This further helps to inform practitioners which code style criteria they should be more careful in order to get a PR acceptance in a quicker way. At last, after we got an overview of to what extent PRs kept a consistent code style with targeting projects, we further explored how the code style policy of a project and PR authors’ experience (measured in four kinds of contributions) was correlated with the ratio of PRs with no inconsistency in code style in the project. Our work not only helped us obtain a more complete view of the effect of code style on PRs processing but also provided some actionable insights that developers could refer to better contribute to a project.

Some research studies on code review found that reviewers often checked whether the reviewed code violated the existing project style, code guidelines, or team standards (Rigby and Storey 2011; Bacchelli and Bird 2013). Butler et al. (2009) found out that there existed a significant association between identifier names and code quality. Miara et al. (1983) figured that program indentation would affect its comprehensibility. (Boogerd and Moonen 2009) declared that code standard violations may introduce faults. Smit et al. (2011) identified several coding conventions which affected the code maintainability most. Allamanis

et al. (2014) proposed a framework, i.e., NATURALIZE, to infer naming and formatting conventions from code. Their tools could help developers better maintain a consistent code style within projects. Balachandran (2013) developed a review bot, which integrated *Checkstyle* and *FindBugs*<sup>25</sup> to automatically check the code style violations and potential bugs. In this paper, we identified the code style inconsistency to show its effect on PR integration. Our work is a kind of exploratory analysis of PRs based on the code style.

### 6.3 Factors of PR Integration

Many researchers have investigated which factors influenced the integration of PR (Gousios et al. 2015; Yu et al. 2015; Soares et al. 2015). Some focused on exploring factors that would affect the acceptance of PRs (Gousios et al. 2015; Tsay et al. 2014a); while some others focused on what would influence the time cost of closing PRs (Yu et al. 2015; Gousios et al. 2014). Gousios et al. (2015) conducted a survey among project integrators and concluded both technical and social factors, such as code quality and developer reputation, which would affect the acceptance of PRs. Tsay et al. (2014a) quantitatively measured the effect of different social and technical factors on the acceptance of contributions in GitHub. They also investigated how people evaluated a PR through discussions (Tsay et al. 2014b). (Vasilescu et al. 2015) found that *continuous integration* could make more PRs merged. Rahman and Roy (2014) found that factors, such as the maturity of a project and the number of involved developers, can affect the merging of PRs. Padhye et al. (2014) showed that PRs of bug fixes tended to be more likely to be merged than that of feature enhancements. Zhang et al. (2018) explored the impact of competing PRs on PR integration. Different from above studies, we mainly focused on quantitatively evaluating the effect of code style inconsistency on the decision of merging PRs. Our work complements existing studies on the merging of PRs.

(Yu et al. 2015) found that factors, such as the commit size and the number of comments, can effectively indicate the latency of PRs. Zhang et al. (2014) found that PRs using *@-mention* tended to need more time to be handled. Gousios et al. (2014) found that the track record of a developer had a strong correlation with the time cost to close his/her PRs. This was also confirmed by our study. To help people better cope with large PRs, some researchers proposed methods to rank PRs and recommend competent developers to process PRs (van der Veen et al. 2015; Yu et al. 2014; de Lima Júnior et al. 2015). In our work, we considered an orthogonal research problem, namely, the effect of code style inconsistency on the closing time of PRs. We found that the code style inconsistency would also delay the processing of PRs.

## 7 Conclusions

In this paper, we conducted an exploratory study on the effect of code style on PR integration in GitHub with 117 projects. We found that there indeed existed PRs, which held different code styles with the existing code in the codebase; several code style criteria generally revealed high divergence while several other criteria always indicated no inconsistency. The code style inconsistency between PRs and the existing code would affect the process of merging PRs into the codebases; this inconsistency would also affect the time cost to close a PR.

<sup>25</sup><http://findbugs.sourceforge.net>

In the future, we plan to investigate the correlation between the code style and the code quality. We also plan to conduct experiments on more projects to improve the generalizability of our findings.

**Acknowledgments** The authors would like to greatly thank our lab members, Yufeng Zhao, Yiming Chen, and Mengting Zhou, for crawling GitHub project data for experiments. This work is partly supported by the National Natural Science Foundation of China (Grant No.61690201, 61772014, 61802171, 61872273, 61572375), and the China Scholarship Council Scholarship. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

## References

- Allamanis M, Barr ET, Bird C, Sutton CA (2014) Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 281–293
- Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proceedings of the 35th International Conference on Software Engineering, pp 712–721
- Balachandran V (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: Proceedings of the 35th International Conference on Software Engineering, pp 931–940
- Bartoń K (2013) Mumin: Multi-model inference. r package version 1.9. 13 The Comprehensive R Archive Network (CRAN), Vienna, Austria
- Bates DM (2010) lme4: Mixed-effects modeling with r
- Berry RE, Meekings BAE (1985) A style analysis of C programs. *Commun ACM* 28(1):80–88
- Booger C, Moonen L (2009) Evaluating the relation between coding standard violations and faultswithin and across software versions. In: Proceedings of the 6th International Working Conference on Mining Software Repositories, pp 41–50
- Bridger A, Pisano J (2001) C++ coding standards
- Butler S, Wermelinger M, Yu Y, Sharp H (2009) Relating identifier naming flaws and code quality: An empirical study. In: Proceedings of the 16th Working Conference on Reverse Engineering, pp 31–35
- Cohen J (1977) *Statistical power analysis for the behavioral sciences* (revised ed.)
- Cohen J, Cohen P, West SG, Aiken LS (2013) *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, Evanston
- Cohen-Goldberg AM (2012) Phonological competition within the word: Evidence from the phoneme similarity effect in spoken production. *J Mem Lang* 67(1):184–198
- de Lima Júnior ML, Soares DM, Plastino A, Murta L (2015) Developers assignment for analyzing pull requests. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp 1567–1572
- Google (2013) Google Java code style. <http://google.github.io/styleguide/javaguide.html>
- Gousios G, Pinzger M, van Deursen A (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, pp 345–355
- Gousios G, Zaidman A, Storey MD, van Deursen A (2015) Work practices and challenges in pull-based development: The integrator’s perspective. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, pp 358–368
- Gousios G, Storey MD, Bacchelli A (2016) Work practices and challenges in pull-based development: the contributor’s perspective. In: Proceedings of the 38th International Conference on Software Engineering, pp 285–296
- Graham MH (2003) Confronting multicollinearity in ecological multiple regression. *Ecol* 84(11):2809–2815
- Hauke J, Kossowski T (2011) Comparison of values of pearson’s and spearman’s correlation coefficients on the same sets of data. *Quaest Geogr* 30(2):87–93
- Hellendoorn V, Devanbu PT, Bacchelli A (2015) Will they like this? evaluating code contributions with language models. In: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories, pp 157–167
- Jaeger FT (2011) Fitting, Evaluating, and Reporting Mixed Models
- Jiarpakdee J, Tantithamthavorn C, Treude C (2018) Autospearman: Automatically mitigating correlated software metrics for interpreting defect models. In: Proceedings of the 34th International Conference on Software Maintenance and Evolution, pp 92–103
- Johnson PC (2014) Extension of nakagawa & schielzeth’s r2glmm to random slopes models. *Methods Ecol Evol* 5(9):944–946

- Kabacoff R (2015) R in action: data analysis and graphics with R. Manning Publications Co.
- Kalliamvakou E, Gousios G, Blincoe K, Singer L, Germán DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp 92–101
- Kalliamvakou E, Damian DE, Blincoe K, Singer L, Germán DM (2015) Open source-style collaborative development practices in commercial projects using github. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, pp 574–585
- Lemhöfer K, Dijkstra T, Schriefers H, Baayen RH, Grainger J, Zwitserlood P (2008) Native language influences on word recognition in a second language: A megastudy. *J Exper Psychol Learn Memory Cogn* 34(1):12
- Mäntylä MV, Lassenius C (2009) What types of defects are really discovered in code reviews? *IEEE Trans Softw Eng* 35(3):430–448
- Marca D (1981) Some pascal style guidelines. *ACM Sigplan Not* 16(4):70–80
- McConnell S (1993) Code complete: a practical handbook of software construction. Microsoft Press
- Miara RJ, Musselman JA, Navarro JA, Shneiderman B (1983) Program indentation and comprehensibility. *Commun ACM* 26(11):861–867
- Nakagawa S, Schielzeth H (2013) A general and simple method for obtaining  $r^2$  from generalized linear mixed-effects models. *Methods Ecol Evol* 4(2):133–142
- Oman PW, Cook CR (1990) A taxonomy for programming style. In: Proceedings of the ACM 18th Annual Computer Science Conference on Cooperation, pp 244–250
- Oracle (1999) Oracle java code style. <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>
- Padhye R, Mani S, Sinha VS (2014) A study of external community contribution to open-source projects on github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp 332–335
- Rahman MM, Roy CK (2014) An insight into the pull requests of github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp 364–367
- Rees MJ (1982) Automatic assessment aids for pascal programs. *SIGPLAN Not* 17(10):33–42
- Rigby PC, Storey MD (2011) Understanding broadcast based peer review on open source software projects. In: Proceedings of the 33rd International Conference on Software Engineering, pp 541–550
- Sadowski C, Aftandilian E, Eagle A, Miller-Cushon L, Jaspán C (2018) Lessons from building static analysis tools at google. *Commun ACM* 61(4):58–66
- Selya AS, Rose JS, Dierker LC, Hedeker D, Mermelstein RJ (2012) A practical guide to calculating cohen's  $f^2$ , a measure of local effect size, from proc mixed. *Front Psychol* 3:111
- Smit M, Gergel B, Hoover HJ, Stroulia E (2011) Code convention adherence in evolving software. In: Proceedings of the IEEE 27th International Conference on Software Maintenance, pp 504–507
- Soares DM, de Lima Júnior ML, Murta L, Plastino A (2015) Acceptance factors of pull requests in open-source projects. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp 1541–1546
- Tsay J, Dabbish L, Herbsleb JD (2014a) Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering, pp 356–366
- Tsay J, Dabbish L, Herbsleb JD (2014b) Let's talk about it: evaluating contributions through discussion in github. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp 144–154
- Vasilescu B, Yu Y, Wang H, Devanbu PT, Filkov V (2015) Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp 805–816
- van der Veen E, Gousios G, Zaidman A (2015) Automatically prioritizing pull requests. In: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories, pp 357–361
- Vermeulen A (2000) The Elements of Java (TM) Style. Cambridge University Press, Cambridge
- Yu Y, Wang H, Yin G, Ling CX (2014) Reviewer recommender of pull-requests in github. In: Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution, pp 609–612
- Yu Y, Wang H, Filkov V, Devanbu PT, Vasilescu B (2015) Wait for it: Determinants of pull request evaluation latency on github. In: Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories, pp 367–371
- Zhang Y, Yin G, Yu Y, Wang H (2014) Investigating social media in github's pull-requests: a case study on ruby on rails. In: Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies, pp 37–41
- Zhang X, Chen Y, Gu Y, Zou W, Xie X, Jia X, Xuan J (2018) How do multiple pull requests change the same code: A study of competing pull requests in github. In: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, pp 228–239



**Wei qin Zou** received the bachelor degree in software engineering and the master degree in computer science from Dalian University of Technology, China, in 2010 and 2013, respectively. She is working toward a PhD degree at the Software Institute, Nanjing University, China, advised by Prof. Baowen Xu and Prof. Zhenyu Chen. Her research interests include empirical study and mining software repositories.



**Jifeng Xuan** is a professor at the School of Computer Science, Wuhan University, China. He received the BSc degree and the PhD degree from Dalian University of Technology, China. He was previously a postdoctoral researcher at the INRIA Lille Nord Europe, France. His research interests include software testing and debugging, software data analysis, and search based software engineering. He is a member of the ACM, IEEE, and CCF.



**Xiaoyuan Xie** received B.Sc. and M.Phil. degrees in Computer Science from Southeast University, China in 2005 and 2007, respectively, and received PhD degree in Computer Science from Swinburne University of Technology, Australia in 2012. She is currently a professor in School of Computer Science, Wuhan University, China. Her research interests include software analysis, testing, debugging, and search based software engineering.



**Zhenyu Chen** is a Professor of Software Institute in Nanjing University. His research interests are mainly the area of intelligent software engineering. He is the Founder of [moocetest.net](http://moocetest.net). He served as the editor board of the IEEE Transactions on Reliability Journal Associate Editor, the Guest Editor of JSS and SP&E Journal, the PC co-chair of QRS 2016, TSA 2016, QSIC 2013, AST 2013. He has published more than 100 papers in the leading academic conferences and journals such as TOSEM, TSE, ICSE, FSE, ISSTA, ICST, etc. He owns more than 40 patents (22 granted), and some of his patents have been transferred to well-known software companies such as Baidu, Alibaba and Huawei.



**Baowen Xu** received the bachelor, master, and PhD degrees in computer science from Wuhan University, Huazhong University of Science and Technology, and Beihang University in 1982, 1985 and 2002, respectively. He is currently a Professor in the Department of Computer Science and Technology at Nanjing University. His major research interests are programming languages, software testing, software maintenance, and software metrics. He has published extensively in premiere software engineering journals and conferences such as TOSEM, TSE, JSS, TR, ICSE, FSE, ICSME, ICST, etc. He is a member of the IEEE.

## Affiliations

Wei Qin Zou<sup>1</sup>  · Jifeng Xuan<sup>2</sup> · Xiaoyuan Xie<sup>2</sup> · Zhenyu Chen<sup>1</sup> · Baowen Xu<sup>1</sup>

Wei Qin Zou  
wqzou@smail.nju.edu.cn

Jifeng Xuan  
jxuan@whu.edu.cn

Xiaoyuan Xie  
xxie@whu.edu.cn

Baowen Xu  
bw Xu@nju.edu.cn

<sup>1</sup> State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

<sup>2</sup> School of Computer Science, Wuhan University, Wuhan, China