Adoption and Evolution of Code Style and Best Programming Practices in Open-Source Projects

Alvari Kupari
University of Auckland
Auckland, New Zealand
akup390@aucklanduni.ac.nz

Nasser Giacaman
University of Auckland
Auckland, New Zealand
n.giacaman@auckland.ac.nz

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

Abstract—Following code style conventions in software projects is essential for maintaining overall code quality. Adhering to these conventions improves maintainability, understandability, and extensibility. Additionally, following best practices during software development enhances performance and reduces the likelihood of errors. This paper analyzes 1,036 popular open-source JAVA projects on GITHUB to study how code style and programming practices are adopted and evolve over time, examining their prevalence and the most common violations. Additionally, we study a subset of active repositories on a monthly basis to track changes in adherence to coding standards over time. We found widespread violations across repositories, with Javadoc and Naming violations being the most common. We also found a significant number of violations of the GOOGLE Java Style Guide in categories often missed by modern static analysis tools. Furthermore, repositories claiming to follow code-style practices exhibited slightly higher overall adherence to code-style and best-practices. The results provide valuable insights into the adoption of code style and programming practices, highlighting key areas for improvement in the open-source development community. Furthermore, the paper identifies important lessons learned and suggests future directions for improving code quality in JAVA projects.

Index Terms—Software quality, Code style conventions and standards, Software maintainability, Open-source software, JAVA programming, Static analysis, Software evolution, Mining software repositories, Software development practices, Software metrics.

I. INTRODUCTION

Following code style conventions is essential for maintaining high-quality software [1]. Proper code style improves readability, maintainability, and consistency, making it easier for developers to collaborate and extend software projects [2]. Given the prominent role of open-source software [3], understanding the degree to which projects adhere to established style conventions is increasingly important.

Previous studies on code style and programming practice adoption [3]–[8] have been limited in scale. Many focused on small datasets and small number of code-style checkers. Additionally, it remains unclear how adherence to code-style evolves over time or whether explicitly claiming adherence in documentation leads to better code-style.

To address these gaps, this paper presents a large-scale empirical study on JAVA code style and best programming practice adherence. Specifically, we rely on GOOGLE's JAVA Style Guide [9], which is widely considered the most popular JAVA coding convention. To conduct this study, we extended

GRADESTYLE [10], an open-source tool we previously developed as a style checker for assessing the code style of student assignments. Our extension allows the tool to analyze open-source repositories. We also implemented eight new code style and programming practice checkers based on GOOGLE's JAVA Style Guide to improve the tool's ability to assess real-world projects. To the best of our knowledge, the extended version of GRADESTYLE provides the most comprehensive detection of GOOGLE JAVA code style violations available. Indeed, commercial tools like PMD and CHECKSTYLE miss important violations [10].

We analyzed 1,036 open-source JAVA repositories from GITHUB, selected based on their popularity (ranked by stars). By running GRADESTYLE on these repositories, we studied the adoption of nine code style and seven programming practices, identifying key areas for improvement. Additionally, we distinguished between repositories that explicitly claim to follow a style guide and those that do not, allowing us to examine whether stated intent correlates with actual adherence. Moreover, from this dataset, we selected 41 mature and actively maintained repositories for further analysis. Each repository was analyzed 12 times—once per calendar month over the past year. This allowed us to track how adherence to coding standards evolves over time.

The results of our study reveal several key findings: First, code-style and best-practice violations are widespread, with naming conventions and Javadoc documentation being among the most frequently violated categories. Second, repositories that explicitly claim to follow a coding standard exhibit slightly better adherence. Third, over time, adherence to programming best practices shows modest improvement, while certain code style violations, such as method and variable naming, become more prevalent.

In summary, this paper makes the following contributions:

- We present a large-scale empirical study of 1,036 popular open-source JAVA repositories, providing a comprehensive overview of code style and best programming practice adherence.
- We bring a series of important insights into the adoption and evolution of code style and best programming practice adherence. Our findings provide actionable insights for

open-source maintainers, tool developers, and researchers seeking to improve coding standards in JAVA projects.

- We propose a new version of GRADESTYLE for analyzing open-source repositories, with eight new violation checks based on GOOGLE's JAVA Style Guide. We released the new version of the tool [11] and all experimental data [12] to foster future work in this area.

II. EXPERIMENTAL DESIGN

Our study investigates the adoption and evolution of code style, programming practices in popular open-source JAVA projects in GITHUB. Specifically, our empirical study addresses the following four research questions (RQs):

RQ1 Violation Prevalence. How prevalent are different code style and programming practice violations in open-source JAVA projects?

RQ2 Claimed vs. Actual Code Style Adherence. Do repositories that explicitly claim to follow a code style guide (e.g., GOOGLE's JAVA Style Guide) exhibit better adherence to coding standards than those that do not?

RQ3 Evolution of Code Style and Best Programming Practices. How does code style and programming practice adherence evolve over time in actively maintained and mature JAVA repositories?

RQ4 Common Orderings of Class Elements. Is there a common ordering of JAVA elements in a class that is widespread in open-source repositories?

RQ1 aims to understanding the extent to which code style and programming practice violations occur, providing insights into common issues that developers face. Identifying frequent violations can highlight areas where better enforcement or improved tooling may be needed.

RQ2 investigates whether explicit mentions of code style in documentation or configuration files correlate with better compliance, providing insights into the effectiveness of declaring coding style adherence.

RQ3 analyzes the historical data (i.e., git commits) of well-maintained and actively developed JAVA repositories to assess how adherence to code style and programming practices evolves over time. This investigation helps determine whether long-term maintenance leads to improvements in best practices and overall code quality.

RQ4 aims to understand the prevalence of common orderings of JAVA elements within a class. While JAVA style guides, such as GOOGLE's, recommend consistency, they do not prescribe a specific order. As a result, there is no universally accepted standard. Identifying popular orderings can help developers identify and follow commonly used, but not strictly enforced standards for ordering Class elements.

A. Code Style Violations and Best Practices

We now explain the code style violations and programming best practices analyzed in this study. The two most widely known coding conventions for JAVA are ORACLE's (Sun's) JAVA Code Conventions [13] and GOOGLE's JAVA Style Guide [9].

These conventions are mostly similar, with minor differences. We selected **GOOGLE's JAVA Style Guide** because ORACLE's conventions have not been updated since April 20, 1999 (over 23 years ago). GOOGLE's guide, last updated in 2018, covers modern JAVA features introduced after 1999 and reflects how JAVA code style has changed over time [14].

We did not consider every possible violation or programming practice specified in the GOOGLE's JAVA Style Guide [9], as some of them vary depending on tools and developer preferences. For example, indentation, brace placement, and whitespace formatting usually follow GOOGLE's JAVA Style Guide, but formatting tools in IDEs may produce slightly different results [15]. These minor differences do not significantly impact code quality, so we excluded them from our study. Another example is the order of elements within a class. Although the order is crucial for code readability, as emphasized in GOOGLE's JAVA Style Guide [9], GOOGLE's guide does not enforce a specific order. Thus, we did not include ordering as a violation but still analyzed in RQ4 different ordering patterns to find common practices. Moreover, we incorporated additional checkers for JAVA best programming practices violations that we consider important, common, and interesting to analyze.

Table I summarizes the violations analyzed in this study, categorized into two groups according to the GOOGLE JAVA Style Guide.

- **Code style violations**: These violations focus on how code is written, particularly regarding naming conventions and formatting standards. Adhering to these guidelines enhances readability and maintainability, making future modifications easier.
- **JAVA best programming practices violations**: Unlike general code style, these violations relate specifically to JAVA coding practices. Although following these practices can also improve readability, their primary goal is to enhance maintainability, security, and performance.

Code style violations:

Class Names. Class and enum names must be in *UpperCamelCase* (each internal word starts with a capital letter). Class names should be nouns or noun phrases, such as `Customer` or `DataManager`. GOOGLE's JAVA Style Guide allows class names to start with adjectives followed by nouns, like `SortedMap` or `ImmutableSet`. Any class declaration not following these naming rules counts as a violation.

Package Names. Package names must be lowercase and match the directory structure. For example, a class `User.java` located in `src/org/project/User.java` must declare the package as `package org.project;`. Package names violating this rule count as violations.

Method Names. Method names should be verb phrases using *lowerCamelCase* (the first word starts lowercase, with each following word capitalized). Method declarations that break this rule count as violations.

Variable Names. Variable names, including instance fields, parameters, and local variables, should use *lowerCamelCase*

and not start with an underscore (_) or dollar sign ($). Static final variables must use uppercase letters with underscores separating words (e.g., `TOTAL_COUNT`, `COLOR_BLUE`). Variable names breaking these conventions count as violations.

Javadoc. Every public class, method, enum, and constructor must have a Javadoc comment, and these comments should contain a minimum number of words. While GOOGLE's JAVA Style Guide does not explicitly mention how long Javadocs should be, we required a minimum of 10 words to ensure clarity and depth in Javadoc comments. This promotes detailed documentation across projects. Method Javadocs must also include appropriate tags like `@param`, `@return`, and `@throws`, matching their method signatures. Specifically:

- **Classes and Enums**: Must have a Javadoc comment above the declaration, with at least 10 words.
- **Constructors**: Must have a Javadoc comment above the declaration, with at least 10 words.
- **Methods**: Must have a Javadoc comment above the declaration, with at least 10 words.

JAVA best programming practices violations:

Missing Override. Methods overriding or implementing superclass or interface methods must include the `@Override` annotation, unless the parent method is marked as `@Deprecated`. This annotation prevents errors, improves readability, and ensures refactoring safety.

Empty Catch Blocks. Caught exceptions should never be silently ignored. Typical actions include logging the error or rethrowing an exception. If intentionally left empty, the reason must be clearly explained with a comment:

```
1  try {
2    int value = Integer.parseInt(input);
3    processNumber(value);
4  } catch (NumberFormatException ok) {
5    // Non-numeric input is expected; continue normally
6  }
```

In test methods, exceptions with names starting with `expected` may be ignored without a comment, as shown below:

```
1  try {
2    emptyStack.pop();
3    fail();
4  } catch (NoSuchElementException expected) {}
```

Unqualified Static Access. Static class members must be accessed using the class name, not through an instance or method return value:

```
1  // doWork() is a static method
2  Utils.doWork(); // good
3  utilInstance.doWork(); // bad
4  getUtils().doWork(); // very bad
```

Finalize Override. Methods should never override `Object.finalize()`, since JAVA has scheduled finalization for removal due to reliability issues.

Private Instances. To support encapsulation, instance fields should be declared `private` or `protected`. External access to these fields should use public getter and setter methods.

String Concatenation. Frequent string concatenation, especially in loops, reduces performance significantly. The following pattern should be avoided:

```
1  String result = "";
2  for (int i = 0; i < 50000; i++) {
3    result += i + " ";
4  }
```

Instead, use a `StringBuilder`:

```
1  StringBuilder sb = new StringBuilder();
2  for (int i = 0; i < 50000; i++) {
3    sb.append(i).append(" ");
4  }
5  String result = sb.toString();
```

The first example can be over 500 times slower than the second[1]. Each instance of string concatenation in loops counts as a violation.

Useless Code. High-quality code should not contain unused imports, methods, variables, or commented-out code. Every line containing such unused or commented-out code counts as a violation.

B. Implementation

To detect code style violations, we evaluated several existing tools, initially focusing on well-known commercial tools such as CHECKSTYLE [16], PMD [17], SONARQUBE [18], ERRORPRONE [19], and SPOTBUGS [20]. However, these tools do not aim to detect many violations described in GOOGLE's JAVA Style Guide. For instance, although these tools can detect class and method naming conventions (*UpperCamelCase* and *lowerCamelCase*), they do not apply natural language processing (NLP) to ensure correct noun or verb usage, which is required by GOOGLE's JAVA Style Guide. Indeed, CHECKSTYLE marks GOOGLE's rules "5.2.2-Class-names" and "5.2.3-Method-names" as partially supported [21].

These existing tools also struggle detecting violations which require context specific information. For example, CHECKSTYLE cannot detect violations requiring context from multiple files simultaneously, such as *Missing Override* and *Unqualified Static Access*. This is because CHECKSTYLE analyses one file at a time, and is therefore unable to deduce the class hierarchy needed to detect such violations. Indeed, GOOGLE's JAVA style guide rules 6.1 and 6.3 are explicitly marked as "cannot be detected by Checkstyle due to its limitation: multiple file checking is not supported" [21].

We then turned our attention to research tools and selected **GRADESTYLE** [10], our own open-source tool, originally developed for educational purposes [10]. GRADESTYLE leverages existing static analysis libraries like PMD and CHECKSTYLE but proposes additional checks using static analysis and NLP techniques. This tool can detect 12 significant categories of JAVA code style and best-practice violations [10]. To the best of our knowledge, GRADESTYLE is the most comprehensive

---

[1]While the Java Just-In-Time (JIT) compiler may optimize simple string concatenations using `StringBuilder`, this optimization often does not apply to repeated concatenation in loops. Each iteration creates a new `String` object, leading to significant performance overhead due to the immutability of `String`. Using `StringBuilder` avoids this by reusing a mutable buffer.

Violation	Description	Impl.	GRADESTYLE	GOOGLE	Novel
Code Style Violations					
Class Names	Class names should have a noun, follow UpperCamelCase.	custom	✓	✓	✗
Method Names	Method names should have a verb, follow camelCase.	custom	✓	✓	✗
Variable Names	Variable names should follow camelCase.	custom	✓	✓	✗
Package Names	Package names should match the project structure and be all lowercase.	CS [16]	✓	✓	✗
Javadoc Formatting	Javadoc comments should be there, with comments for all tags.	CS [16]	✓	✓	✗
Class Javadocs	Classes should have a Javadoc comment above the class declaration.	custom	☆	✓	✗
Constructor Javadocs	Constructors should have a Javadoc comment above the constructor declaration (min 10 words).	custom	☆	✓	✗
Method Javadocs	Methods should have a Javadoc comment above the method declaration (min 10 words).	custom	☆	✓	✗
Field Javadocs	Fields should have a Javadoc comment.	custom	☆	✓	✗
Programming Practices Violations					
Private Instances	All fields within a class should be private to prevent outside access; getters/setters should be used.	custom	✓	✗	✗
Useless	For exmaple, fields and methods that project are not used are flagged as unnecessary.	custom	✓	✗	✗
String Concatenation	Strings should not be concatenated within loops to boost performance; `StringBuilder` should be used.	custom	✓	✗	✓
Missing Override	`@Override` annotations should be placed wherever a method is overridden for readability.	PMD [17]	☆	✓	✓
Empty Catch Block	Empty catch blocks can hide errors and make debugging difficult.	PMD [17]	☆	✓	✗
Unqualified Static Access	Static methods should be called using the class name to avoid confusion.	custom	☆	✓	✓
Finalize Override	`Object.finalize()` should not be overridden, as it is scheduled for removal.	custom	☆	✓	✓

tool currently available for detecting GOOGLE JAVA style violations[2].

While GRADESTYLE already handles several violations targeted in our study, it misses some key checks listed in GOOGLE's JAVA best practices [9], such as *Missing Override* and *Empty Catch Block*. GRADESTYLE is also built for an educational context where student repositories follow a predictable skeleton structure given by instructors, and is less suitable for open-source analysis where file structure varies considerably. Therefore, we extended GRADESTYLE to more comprehensively cover these missed checks, and to improve ease of use for research.

We also improved GRADESTYLE's granularity in detecting Javadoc violations. Originally, GRADESTYLE had a single configuration for Javadocs [10]. We modified GRADESTYLE to separately evaluate constructor, method, field, and class Javadocs. This improvement allows a more detailed and accurate analysis of Javadoc compliance. These additional violation checkers are crucial because adhering to them significantly enhances code readability and understandability, particularly valuable in open-source projects where many contributors collaborate without centralized oversight.

However, certain original GRADESTYLE violation checkers were intentionally excluded from our analysis due to limited relevance for open-source projects or because the results depend heavily on parameter choices, which are difficult to set universally for all projects (e.g., amount of comments [10]).

Table I summarizes each violation analyzed in this study (see Section II-A), indicating whether GRADESTYLE originally supported it or if we implemented it as a new extension in GRADESTYLE (marked with ☆). It also indicates which violations are specified in the GOOGLE JAVA Style Guide and which are not (Column "GOOGLE"). We also indicate which violation checkers are novel and, to the best of our knowledge,

has not been investigated in open-source repositories before (e.g., [3], [4]), by marking them in the "Novel" column.

The original GRADESTYLE integrates directly with CHECK-STYLE [16] (v. 10.3.2) for *PackageNames* and *Javadoc* violations. For detecting naming violations (*ClassNames* and *MethodNames*), GRADESTYLE uses regular expressions combined with NLP through EXTJWNL [22] (v. 2.0.5). Regular expressions check for correct camel-case notation, while EXTJWNL ensures class names contain appropriate nouns and method names contain appropriate verbs by performing Part-of-Speech (PoS) tagging. For other violations GRADESTYLE implements custom static analyses using JAVAPARSER [23] (v. 3.23.1), a library designed for static analysis of JAVA source code, which helps to precisely identify specific language constructs.

To add new violation checkers to GRADESTYLE, we relied on PMD [17] (v. 6.52.0) for detecting the programming practice violations `@Override` annotations and empty catch blocks. For the other new violation checkers, we implemented custom static analyzers using JAVAPARSER.

Furthermore, we adapted GRADESTYLE to better suit open-source projects. Originally, GRADESTYLE searched for JAVA files exclusively within the `src/main/java` directory at the project's root, a standard setup for student assignments. Open-source repositories, however, often follow different and non-standard directory structures (e.g., with MAVEN submodules). Therefore, we modified the tool to recursively search for JAVA files across multiple `src/main/java` directories located anywhere within the project's hierarchy.

C. Repository Mining

In this study, we aimed to collect a representative set of **high-quality JAVA repositories from GITHUB** by selecting projects in descending order based on their number of stars. Following previous research on mining repositories [24], the number of stars is widely regarded as the best proxy for popularity and, consequently, quality [24]. Popular repositories are expected to have higher code quality, a larger number of contributors, and

---

[2]Although CHECKSTYLE appears to support many rules from the GOOGLE JAVA Style Guide (see [21]), this is misleading. The site lists sub-rules individually, creating the impression of broader coverage. Conversely, GRADESTYLE groups related checks together.

better overall maintenance [25]. To balance generalizability with computational efficiency, we set a threshold of **1,036 repositories**. We retrieved the top-starred JAVA repositories using the GITHUB API [26] and stopped after obtaining 1,036 valid repositories.

To ensure meaningful analysis, a valid repository had to meet three selection criteria:

1) **Repository Size** Repositories larger than 2 GB were excluded to avoid excessive processing overhead.
2) **JAVA Codebase Size** We also excluded repositories with excessively large JAVA codebases. The GITHUB API provides a heuristic estimate of total bytes written in Java, which we used as an approximation of the codebase size. Repositories exceeding 2 million bytes of JAVA code were filtered out to ensure smooth execution of our analysis.
3) **Tool Compatibility** To verify compatibility, we initially conducted a preliminary check using CHECKSTYLE. Repositories containing JAVA files with syntax errors that caused tool failures were excluded from the dataset. We chose CHECKSTYLE for this initial check because it is computationally less expensive compared to GRADESTYLE. If a repository successfully passed the CHECKSTYLE test, we then ran our modified version of GRADESTYLE to ensure it operated without crashes.

The final dataset included 1,036 repositories with the following characteristics:

- **Stars** – Average: 4,816; Min: 1,814; Max: 49,470; Median: 3,098
- **Lines of Code (LOC)** – Average: 19,672; Min: 29; Max: 8,873,014; Median: 5,364

Due to the filtering criteria, our dataset does not consist solely of the top 1,036 GITHUB repositories ranked by stars. Instead, it includes the first 1,036 repositories that met all selection criteria. Despite this, all selected repositories have at least 1,814 stars, indicating significant popularity. Therefore, our dataset remains representative of popular open-source JAVA projects. We then ran our extended version of GRADESTYLE for all the 1,036 repositories.

D. Manual Validation of Violation Detection

To validate the accuracy of GRADESTYLE's violation detection and to quantify the amount of false positives, the first author performed a manual validation of the results using stratified random sampling. Indeed, manually verifying all detected violations would not be possible. Stratification ensures a more representative evaluation across different violation categories [27]. We determined the required sample size using a standard calculator [28] with a 15% margin of error and a 90% confidence level, resulting in a sample size of 31 violations per category.

For stratification, we divided the 1,036 repositories into 31 evenly sized groups. From each group, we randomly selected one violation per category. We selected the first violation for the category found within the group.

TABLE II
MANUAL VALIDATION RESULTS OF THE DETECTED VIOLATIONS

Violation Category	# Violations Checked	False Positives
Class Names	36	None
Method Names	42	None
Variable Names	39	None
Package Names	31	None
Javadocs Class	43	None
Javadoc Method	44	None
Javadoc Constructor	45	None
Javadoc Formatting	43	None
Private Instances	36	None
Useless	40	None
String Concatenation	31	None
Missing Override	31	None
Finalize Override	31	None
Unqualified Static Access	33	None
Empty Catch Block	35	None

GRADESTYLE provides direct links to flagged violation locations, facilitating the verification process. To verify a violation, the first author opened the file and line location specified by the violation report, and then performed a visual inspection to confirm both the presence of the violation, and whether the violation description accurately reflected the code.

Table II summarizes the number of violations manually checked for each category. The manual validation confirmed high accuracy of the GRADESTYLE: zero false positives were found, indicating that all detected violations matched the reported issues.

III. EXPERIMENTAL RESULTS

A. RQ1 - Violation Prevalence

RQ1: *How prevalent are different code style and programming practice violations in open-source JAVA projects?*

Absolute number of violations. Table III provides statistical insights into the absolute numbers of violations detected across the analyzed 1,036 JAVA repositories. The data highlights significant variability across different categories of code style and programming practice violations. All categories also had minimum values of 0, meaning every category had at least 1 repository with no violations.

Notably, Javadoc-related violations occur most frequently. For instance, *Javadoc Formatting* violations have the highest average (380.18) and the maximum number (6,213), indicating widespread inconsistencies in documenting code. Similarly, *Javadoc Method* violations also have a high average (303.26), reflecting substantial issues in method documentation across repositories.

Violations related to naming conventions, such as *Variable Names*, *Method Names*, and *Class Names*, also appear frequently. *Variable Names* show the highest average (88.34) among naming conventions, suggesting common issues with variable naming standards across repositories.

Programming practice violations generally show lower averages compared to code style violations. For example, *String Concatenation* (0.56), *Finalize Override* (0.09), and *Empty*
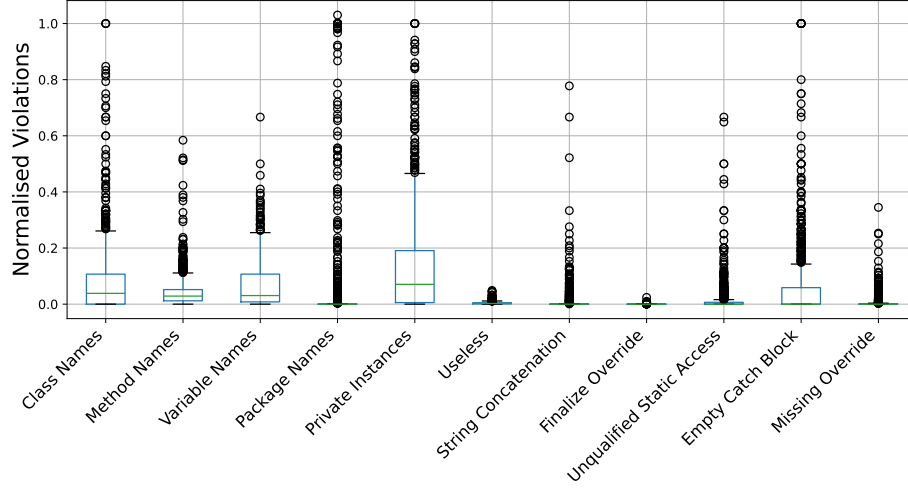
Fig. 1. Distribution of normalized violation counts across categories (RQ1)

Violation Category	Min	Max	Avg	Median
Class Names	0	231	9.45	3.0
Method Names	0	385	26.45	9.0
Variable Names	0	2,514	88.34	28.0
Package Names	0	308	4.30	0.0
Javadoc Class	0	768	56.53	25.0
Javadoc Method	0	5,159	303.26	144.0
Javadoc Constructor	0	516	55.91	29.0
Javadoc Formatting	0	6,213	380.18	124.0
Private Instances	0	1,025	30.02	8.0
Useless	0	920	36.09	13.5
String Concatenation	0	88	0.56	0.0
Finalize Override	0	15	0.09	0.0
Unqualified Static Access	0	135	2.76	0.0
Empty Catch Block	0	215	2.50	0.0
Missing Override	0	321	3.17	0.0

Catch Block (2.50) appear far less frequently. The particularly low frequency of *String Concatenation* violations suggests that developers of popular JAVA repositories on GITHUB are aware of the (very serious) performance issues of string concatenation in loops and actively avoid it, or that there tends to be few scenarios of string concatenation in loops.

Normalized number of violations. It is important to highlight that GRADESTYLE calculates violations both in absolute values and as normalized metrics relative to project size [10]. Specifically, violations are normalized based on the total number of relevant constructs (e.g., variables, classes, methods). For example, *VariableNames* violations are adjusted according to the total number of declared variables in a repository. This normalization ensures fairness, preventing larger projects from being disproportionately penalized simply due to their size. We argue that normalized values provide a more meaningful metric for comparing code quality across repositories of different sizes. Indeed, as previously reported the JAVA LOC of the projects ranges from 29 to ~9M.

Figure 1 presents the distribution of all non-Javadoc-related violations in their normalized form.

All categories exhibit a top-skewed distribution, where most repositories have relatively few normalized violations, while a smaller subset of repositories show significantly high violation counts. This indicates that while the majority of repositories adhere to coding standards, a fraction of them exhibit poor compliance with code style and programming best practices.

The results largely align with the absolute violation counts, with one key exception: the *Empty Catch Block* category shows a notably higher violation rate in the normalized distribution. While its median remains at 0 (indicating that at least half of the repositories have no violations of this type), the upper quartile reaches 0.0588, much higher than other programming practice violations. In contrast, other programming violations such as *Unqualified Static Access* (0.0066), *Missing Override* (0.0014), and *Finalize Override* (0.0) have considerably lower upper quartile values. This means that the worst 25% of repositories contain empty catch blocks in over 5.8% of their catch statements. Notably, one repository had a normalized violation rate of 1, indicating that every single catch block in the project was empty.

Despite this, programming practice violations remain less frequent than code style violations, suggesting better adherence to best practices in JAVA development. However, two notable exceptions stand out: *Private Instances* and *Empty Catch Blocks*. These two categories show violation rates comparable to the more frequently occurring code style violations.

Javadoc-related violations stand apart due to their significantly higher normalized values, as shown in Figure 2. The medians of Javadoc categories are 0.52, 0.47, 0.97, and 0.29, which are much higher than those of other categories. Notably, all Javadoc categories except *Javadoc Formatting* have a maximum normalized value of 1, meaning that some repositories lack Javadoc documentation entirely. *Javadoc*

TABLE IV
NORMALIZED VIOLATION THRESHOLDS BY CATEGORY AND TYPE (RQ2)

| Category	Type	25%	20%	15%	10%	5%	4%	3%	2%	1%	0%
Class Names	Code Style	93.34	89.77	83.69	73.17	55.41	50.77	44.98	39.09	34.27	30.98
Method Names	Code Style	98.94	98.36	96.24	92.37	73.07	62.64	52.12	35.04	22.10	13.03
Variable Names	Code Style	96.53	93.05	85.91	73.17	59.07	55.21	49.42	41.12	28.09	9.65
Package Names	Code Style	93.53	92.95	92.28	91.51	89.19	88.42	87.16	85.42	83.49	80.98
Finalize Override	Programming Practice	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.90	99.90	95.37
Unqualified Static Access	Programming Practice	99.03	98.84	97.97	96.72	93.15	91.70	89.09	86.10	79.15	67.18
Empty Catch Block	Programming Practice	94.21	91.89	88.71	83.20	73.07	70.75	66.80	64.67	61.97	60.33
Missing Override	Programming Practice	99.81	99.61	99.42	99.03	97.01	96.24	95.56	94.02	89.48	71.14
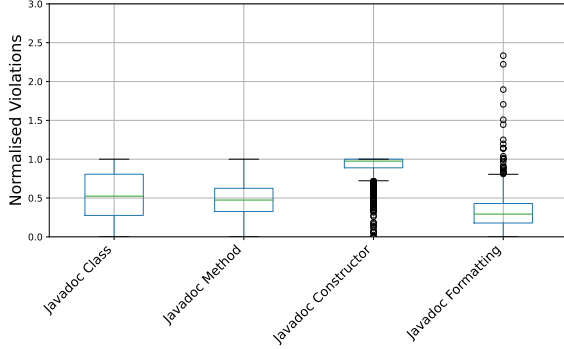
Fig. 2. Prevalence of Javadoc Violations (RQ1)

Formatting violations, however, exceed this threshold, with some repositories having normalized values greater than 1. This occurs because a single Javadoc comment can have multiple associated violations. One extreme outlier had a Javadoc Formatting violation rate of 6, meaning that, on average, each Javadoc comment contained six violations—the maximum possible number. These findings suggest that Javadoc compliance is a major issue in open-source JAVA repositories.

RQ1 in Summary: Our analysis of 1,036 open-source JAVA repositories reveals that Javadoc violations are the most frequent, while naming and programming practice violations occur less often. *Empty Catch Blocks* and *Private Instances* stand out as key areas for improvement.

B. RQ2 - Claimed vs. Actual Code Style Adherence

RQ2: *Do repositories that explicitly claim to follow a code style guide exhibit better adherence to coding standards than those that do not?*

RQ2 investigates whether repositories that claim adherence to a specific code style, particularly GOOGLE's JAVA Style Guide [9], exhibit higher compliance with coding standards than those that do not. Additionally, it examines whether repositories that strictly follow a code style explicitly document their adherence or if it remains implicit.

To identify repositories claiming adherence, we performed keyword searches on all markdown (.md) files located at the project's root using regular expressions to detect references to code style. We used two sets of regular expressions; one searches for general code-style references, such as "code style" and "coding standards". The other is more specific, and searches for GOOGLE JAVA Style mentions, such as "GOOGLE Java Style" or the official URL of the guide. We included all root markdown (.md) files to ensure we covered common files such as README.md and CONTRIBUTING.md where code-style and best-practice expectations are typically conveyed to contributors. We also checked for the presence of PMD or CHECKSTYLE configuration files, as these are the two most widely used static analysis tools for enforcing JAVA coding standards. While this automated approach cannot guarantee complete precision, it represents a best-effort solution. Indeed, manually analyzing all 1,036 repositories would be too costly.

The repositories were classified into three categories:

- **Mention of Code Style** (140 repositories): A root documentation file contains general references to code style, such as "code style" or "code-style".
- **Explicit Mention of GOOGLE's JAVA Style Guide** (16 repositories): The repository explicitly references GOOGLE's JAVA Style Guide, includes a link to its official documentation in a root markdown (.md) file, or contains configuration files for automated GOOGLE style enforcement (e.g., checkstyle.xml).
- **No Mention** (880 repositories): The repository does not explicitly reference any coding standard.

Next, we analyzed the prevalence of violations by defining a threshold to determine when a repository can be considered adherent. Table IV presents the percentage of repositories falling below specific normalized violation thresholds. For instance, 94.21% of repositories have normalized *Empty Catch Block* violations below 0.25. Note that Table IV only includes violations explicitly defined in the GOOGLE JAVA Style Guide (see the GOOGLE column in Table I). As a result, we excluded *Private Instances*, *Useless*, and *String Concatenation*, since these are not part of the official style guide. Additionally, we excluded Javadoc-related violations because RQ1 shows high frequency, making them less informative for characterizing code style adherence.

Based on this analysis, we set a threshold at 5% (0.05 normalized violations) to classify repositories as adherent, as the number of repositories passing this threshold stabilizes
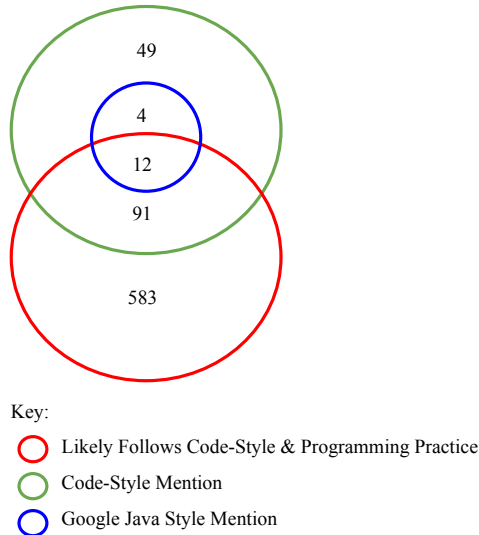
Fig. 3. Claimed Adherence vs. Actual Adherence (RQ2)

Fig. 4. Monthly Commit Activity over 1 Year (RQ3)

Fig. 5. Variation of Total Normalized Violations with Time (RQ3)

beyond this point. Setting the threshold to 0 would be unrealistic, as it assumes that repositories are entirely free of violations, which is rarely the case in practice. Instead, a repository is considered adherent to a code style category if its normalized violations fall below the chosen threshold.

To assess whether claimed adherence aligns with actual adherence, we analyzed the distribution of programming practice and code style violations across repositories. Figure 3 presents a Venn diagram illustrating how adherence rates compare to repositories' self-reported compliance.

The Venn diagram reveals that 12 out of 16 repositories (75%) that explicitly claim adherence to GOOGLE's JAVA Style Guide meet the 5% threshold for programming practice categories. This adherence rate is higher than the 65% observed among repositories that mention code style and the 66% among those with no explicit reference to coding standards.

> **RQ2 in Summary:** Repositories that explicitly reference GOOGLE's JAVA Style Guide demonstrate the highest adherence rates to code style and programming practices, with 75% meeting the 5% compliance threshold. However, adherence remains relatively high (65–66%) even among repositories that do not explicitly claim compliance, suggesting that many projects follow best practices implicitly.

C. RQ3 - Evolution of Code Style and Best Programming Practices

RQ3: *How does code style and programming practice adherence evolve over time in actively maintained and mature* JAVA *repositories?*

To answer RQ3, we examined the 1,036 repositories to identify those that are both mature and actively maintained. We analyzed commit histories over multiple time intervals, collecting timestamps of all commits made in the past year and counting how many intervals contained at least one commit. We
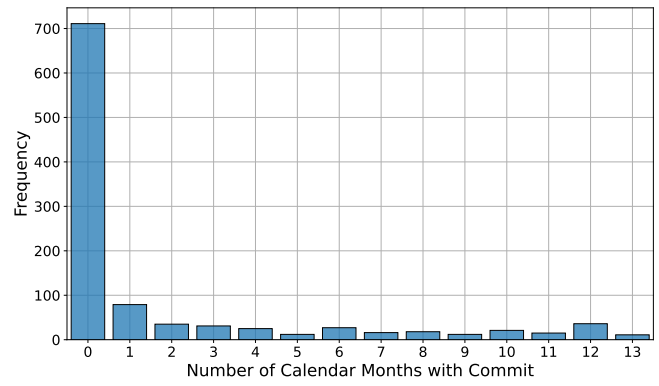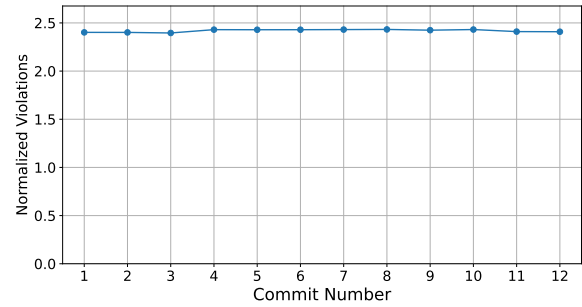
then examined the results to identify repositories with consistent activity. Figure 4 illustrates the distribution of monthly commits over a one-year period for the 1,036 repositories.

From this analysis, we found that 43 repositories have at least one commit every calendar month in the past year. Active maintenance was a crucial selection criterion, as regularly updated repositories with frequent commits provide a more reliable dataset for tracking code evolution over time. To further refine the selection, we filtered for mature repositories: those that are at least 36 months old. This criterion ensures that projects have had enough time to refine their coding standards. After applying this filter, 41 repositories were retained.

For each of these 41 repositories, we sampled 12 commits from the past year, selecting one commit per calendar month. The commit chosen for each month was the one closest to the middle of the month. For example, if a repository had commits on January 2nd, 10th, and 27th, the commit on January 10th was selected as it is closest to the median day of the month (15th). We considered only the last year to avoid analyzing repositories too early in their lifecycle, as early-stage projects may exhibit immature coding practices or function as proof-of-concept projects rather than mature projects.

To prevent selecting commits that were too close in time, we computed the smallest differences between commit dates and manually inspected them. The analysis confirmed that no two commits from the same repository were closer than 10 days apart, ensuring a well-distributed selection.

Finally, we ran the tool on each commit across all selected repositories. Each repository was analyzed 12 times, starting
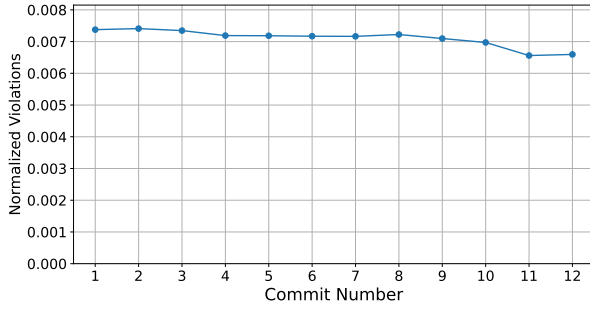
Fig. 6. Trend of Programming Practice Normalized Violations (RQ3)
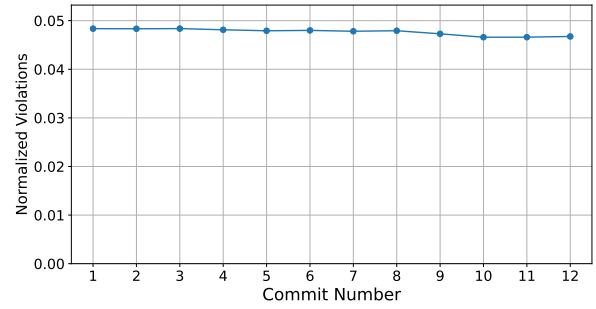

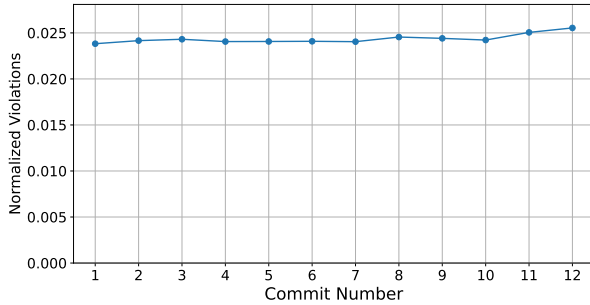Fig. 7. Trend of Class Names Normalized Violations (RQ3)

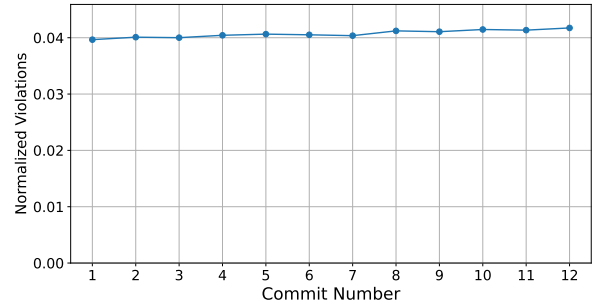Fig. 8. Trend of Variable Naming Normalized Violations (RQ3)

Fig. 9. Trend of Method Naming Normalized Violations (RQ3)

from the oldest commit (February 2024) to the most recent (January 2025). For each commit, the repository was checked out to the corresponding commit hash, restoring the codebase to its exact state at that point in time. Figure 5 shows the average normalized score for each commit across all 41 repositories. Overall, the total normalized score remains relatively stable over the 12-month period, with a slight increase of 0.0068 from February 2024 to January 2025. The variation is minimal, with a maximum fluctuation of 0.037, suggesting that code style and programming practice adherence remains largely unchanged.

Breaking down the results by category, we observe:

- **Programming practice violations** (Figure 6) show a slight downward trend, decreasing from 0.0074 to 0.0066 over the 12 months, suggesting gradual improvement in programming best practices.
- **Class name violations** (Figure 7) also decrease, with a small drop of 0.0014
- **Variable and method naming violations** (Figures 8 and 9) exhibit a slight upward trend, indicating a gradual decline in adherence to naming conventions.

Other categories showed no clear trends.

> **RQ3 in Summary:** Our analysis of actively maintained and mature repositories suggests that code style programming practice adherence remains relatively stable. Notably, class name and programming practices violations slightly decrease, whereas variable and method naming violations slightly increase.

D. RQ4 - Common Orderings of Class Elements

RQ4: *Is there a common ordering of* JAVA *elements in a class that is widespread in open-source repositories?*

The order of elements inside a class significantly impacts code *readability* [9]. While GOOGLE's JAVA Style Guide acknowledges its importance, it does not enforce a specific ordering [9]. Although we did not include ordering violations in our primary study, we want to know if a most common one exists. To achieve this, we analyzed the 1,036 repositories, evaluating different class element arrangements, including fields, constructors, instance methods, and static methods. This check is not implemented in CHECKSTYLE (see Section 3.4.2 in [21]), presumably due to the lack of general agreement on a standard ordering. In contrast, it is supported by GRADESTYLE. We extended GRADESTYLE by adding configurable ordering checks and evaluated four commonly used conventions.

Ordering 1: Inner Classes, Static Fields, Static Methods, Instance Fields, Constructors, Instance Methods (original order used in the GRADESTYLE paper [10]).

Ordering 2: Static Fields, Static Methods, Instance Fields, Constructors, Instance Methods, Inner Classes.

Ordering 3: Static Fields, Static Methods, Instance Fields, Instance Methods, Constructors, Inner Classes

Ordering 4: Instance Fields, Constructors, Instance Methods, Static Fields, Static Methods, Inner Classes.

Figure 10 presents the normalized scores for each ordering across the 1,036 repositories.
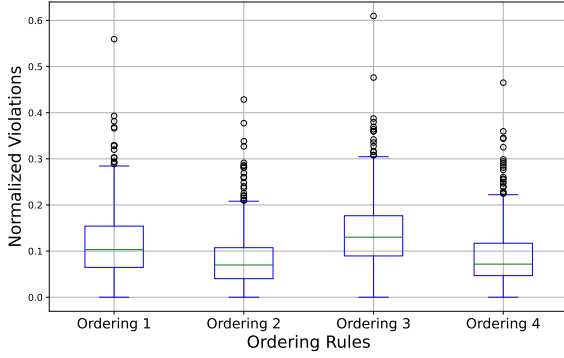
Fig. 10. Comparison of Ordering Rules (RQ4)

The results show that **Orderings 2 and 4** are the most commonly followed, with medians of 0.070 and 0.071, respectively—lower than Orderings 1 and 3 (0.10 and 0.13). This suggests that developers prefer structuring classes with static members grouped together or prioritizing instance-related elements first. A key actionable insight from our study is that open-source repositories should adopt either Ordering 2 or 4.

> **RQ4 in Summary:** Our analysis revealed two popular orderings of JAVA Class elements being used in the repositories. One places static fields and methods at the beginning of the class, while the other places them at the end. We also found that no one ordering was universal among the repositories.

IV. DISCUSSION

This section provides a discussion of the key findings, analyzing their implications for code style adherence, programming best practices, and the broader impact on open-source software development

A. Code Style Adoption

Our findings highlight a need for better adherence to code style conventions in open-source JAVA projects. Figure 1 shows high normalized scores for key categories such as class names, package names, variable names, and private instances, suggesting these areas require more attention from developers.

Conversely, normalized scores for programming practice and *useless code* categories indicate that these practices are largely followed. The primary exception is the *Empty Catch Block* category, which exhibits relatively high violation rates. This is concerning as empty catch blocks hide errors and make debugging harder, which is why it is part of the GOOGLE JAVA Style guide. Therefore more attention needs to be paid to properly make use of catch statements.

B. Explicit vs. Implicit Adherence

Interestingly, repositories that explicitly reference GOOGLE's JAVA Style Guide [9] exhibit the best adherence to programming practice violations. However, only 16 repositories made such claims, highlighting a lack of formal adoption. Encouraging more projects to reference and enforce established style guides could improve overall code quality.

Our results may also encourage owners of open-source JAVA projects to actively push good practices more than they do currently. Further standardizing of JAVA programming practices could also help to push this, for example by seeing common agreement on the ordering of elements within a JAVA class. Furthermore, enforcement of more novel violations should be overseen by managers, such as missing @*Override* annotations. These are especially important due to the annotation helping other developers to understand where subclasses are used. We hope that by releasing our tool with enhanced violation checkers, we can contribute to improving code style adherence within the JAVA community.

V. THREATS TO VALIDITY

Tool Effectiveness. The manual validation step focused only on false positives (i.e., incorrectly flagged violations) but did not systematically evaluate false negatives (i.e., undetected violations). If present, false negatives would indicate our results under-report the prevalence of violations, meaning the violations analysed in this study may be even more widespread. Despite these limitations, the existence of false negatives can only increase the number of true violations, not decrease. This means that our tool can still confidently serve as a lower limit for the number of violations, as we found zero false positives in our manual analysis. While we conducted limited testing for false negatives when extending the tool, a thorough false negative evaluation remains an important future work.

Time Frame for Evolution. Our study analyzed code evolution over the last one-year period, which may not be sufficient to capture long-term trends. Future research could extend this analysis to cover longer periods (e.g., 5–10 years) to better understand how code style adherence evolves in mature projects. Additionally, we sampled one commit per month, a granularity that may be too coarse for highly active repositories. This approach may have missed sudden bursts of code style activity or finer-grained changes in coding behavior. Future studies could employ more refined sampling strategies (e.g., sliding windows or multiple commits per month) to uncover subtler temporal dynamics in coding practices.

Selection Bias. The study focused on 1,036 of the most starred repositories on GITHUB. While this selection ensures popularity and relevance, it may introduce bias, as highly starred repositories may not be representative of the broader open-source JAVA ecosystem, including smaller or less-maintained projects. Exploring repositories with fewer stars remains an important direction for future work to achieve a more comprehensive understanding of the ecosystem.

Threshold Sensitivity. The results are influenced by the chosen 5% violation threshold, and a different threshold could lead to varying conclusions about adherence rates. We mitigate this threat by employing a data-driven approach to determine this threshold (See Table IV). We believe it effectively distinguishes adherence from non-adherence.

VI. Related Work

To the best of our knowledge, our study is the first of its kind in terms of scale, analyzing a large number of repositories and detecting coding violations that have not been examined in previous research. Moreover, we address novel research questions and provide new insights into code style adherence, its evolution over time, and the gap between claimed and actual adherence in open-source JAVA projects. We now discuss the most related work.

Coding style conventions and best programming practices are well-established principles that enhance software readability and maintainability [4]. Many programming languages offer official style guides, such as PYTHON's PEP8 and GOOGLE's coding standards. However, studies on open-source projects suggest that adherence is inconsistent [29]. Beller et al. [5] conducted a study of static analysis tools (including linters for style) across popular projects in JAVA, JAVASCRIPT, PYTHON, and RUBY [5]. They found that roughly half of the studied projects employ at least one automated static analysis tool to enforce best practices, but usually in an ad-hoc manner. Notably, coding style rules are not always strictly enforced—many teams use these tools to flag issues but do not mandate a zero-violation policy. Similarly, Boogerd et al. [6] observed that although style checkers raise a significant portion of build warnings, they rarely cause build failures, suggesting lenient enforcement. Our study builds on these insights by examining adherence trends specifically in JAVA and investigating whether explicit claims of compliance translate to actual adherence (RQ2).

The GRADESTYLE study [10] investigated coding style in student repositories, showing improvement in adherence after receiving automated feedback. Brown et al. [7] similarly studied novice programmers, observing slight increases in Javadoc use over time. While these works analyze controlled learning environments, our study extends this approach to real-world open-source projects.

Jonsson et al. [3] examined adherence to ORACLE's JAVA conventions in a limited set of open-source projects, using PMD and CHECKSTYLE for violation detection. Similarly, Elish and Offutt [4] analyzed 100 JAVA projects and found that only 4% of classes fully adhered to coding conventions. Both of these studies mainly focus on naming, commenting, and structure. Among naming related violations, Elish and Offutt found that 28 out of their 100 investigated JAVA classes had field naming violations, and 18 out of 100 classes had method naming violations. While it is hard to directly compare our results due to a different normalisation approach, these results line up roughly with those of our study, with most field and method names following convention. Our results also agree that field violations are generally more common than method violations. However, Elish and Offutt found no violations of class naming, a strong contrast to our findings, where class naming violations were the most frequent of the naming violations. This suggests that our NLP-based tool can detect naming violations missed by tools limited to camelCase checks.

While our study includes some of these aspects, we also consider violations that were not covered, or only partially addressed. For example, we check whether class names are nouns/noun phrases and method names are verbs/verb phrases, something not checked in either study. Our study also differs by including novel violation checks not covered by previous research, such as *Missing Override* and *Unqualified Static Access*. We also use a significantly larger dataset of 1,036 repositories and cover a broader range of violations, providing a more comprehensive view of modern JAVA coding practices.

VII. Conclusions and futue work

We provided a comprehensive and novel analysis of adherence of code style and best programming practices in 1,036 open-source JAVA projects on GITHUB. Our findings reveal that Javadoc-related violations are the most prevalent. We also observe that there is still room for improvement in other categories, such as naming conventions, private instances and other programming practices, such as empty catch blocks. Repositories that explicitly claim adherence to GOOGLE's JAVA Style Guide demonstrate noticeably better compliance. This highlights the importance of formalizing coding standards and expectations in open-source projects. By releasing the updated GRADESTYLE [11] and all experimental data [12], we aim to foster further research and encourage the open-source community to prioritize consistent coding standards. We submitted 14 pull requests fixing violations detected by our tool. One has been confirmed[3], and another merged[4].

While this study provides valuable insights into code style adherence and programming practice violations in open-source JAVA projects, several areas remain open for future research. We now highlight the three most promising ones.

First, our study examined code style evolution over a one-year period. Future research could extend this analysis to longer time frames to better understand long-term trends and the factors influencing adherence over time.

Second, while our study focuses on the most starred JAVA open-source repositories on GITHUB, future work could explore the adoption of code style and programming practices in repositories with a wider range of star counts. This is important because highly starred repositories may not be fully representative of the open-source ecosystem, where smaller or less popular projects may exhibit different adherence patterns and challenges in maintaining coding standards.

Third, although our dataset likely includes a variety of project types (such as libraries, frameworks, and applications) we did not explicitly classify them. Differentiating between these categories could uncover domain-specific adherence patterns and help identify which types of projects are more or less compliant with established coding standards. This represents an interesting direction for future work.

---

[3]https://github.com/x-ream/sqli/pull/61
[4]https://github.com/Aliucord/Aliucord/pull/521

REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 2014, pp. 281–293.

[2] A. Tornhill and M. Borg, "Code red: the business impact of code quality - a quantitative study of 39 proprietary production codebases," p. 11–20, 2022.

[3] S. Jonsson and N. Safavi, "Are open-source systems developed with good code quality? an empirical study," DIVA, Tech. Rep., 2023. [Online]. Available: https://lnu.diva-portal.org/smash/record.jsf?pid=diva2%3A1746234

[4] M. O. Elish and J. Offutt, "The adherence of open source java programmers to standard coding practices," KFUPM ePrints, 2002, accessed: 2025-02-17. [Online]. Available: https://eprints.kfupm.edu.sa/id/eprint/10664/1/Elish-SEA2002-Paper.pdf

[5] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016, pp. 470–481.

[6] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2008, pp. 277–286.

[7] N. C. C. Brown, P. Weill-Tessier, M. Sekula, A.-L. Costache, and M. Kölling, "Novice use of the java programming language," *ACM Transactions on Computing Education*, 2022, https://doi.org/10.1145/3551393.

[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Mining java class naming conventions," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 93–102.

[9] Google, "Google java style guide," https://google.github.io/styleguide/javaguide.html, Accessed: 2025.

[10] C. Iddon, N. Giacaman, and V. Terragni, "GradeStyle: GitHub-Integrated and Automated Assessment of Java Code Style," in *IEEE/ACM International Conference on Software Engineering, SEET track*, 2023, pp. 192–197.

[11] Alvari Kupari, Callum Iddon, Nasser Giacaman, and Valerio Terragni, "Gradestyle," https://github.com/Digital-Educational-Engineering/gradestyle.

[12] A. Kupari, N. Giacaman, and V. Terragni, "Source Code and experimental data," https://doi.org/10.5281/zenodo.16532529, 2025.

[13] Oracle, "Oracle's code conventions for the java programming language," https://www.oracle.com/java/technologies/javase/codeconventions-contents.html, Accessed: 2025.

[14] A. Trautsch, S. Herbold, and J. Grabowski, "A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects," *Empirical Software Engineering*, vol. 25, no. 6, pp. 5137–5192, 2020.

[15] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?" *Empirical Software Engineering*, vol. 25, pp. 5374–5412, 2020.

[16] "Checkstyle," https://checkstyle.org, Accessed: 2025.

[17] "PMD," https://pmd.github.io, Accessed: 2025.

[18] "SonarQube," https://www.sonarqube.org, Accessed: 2025.

[19] "Error Prone," http://errorprone.info, accessed: 2025.

[20] "SpotBugs," https://spotbugs.github.io, accessed: 2025.

[21] Checkstyle, "Checkstyle coverage of google java style guide," 2025, accessed: 2025-07-27. [Online]. Available: https://checkstyle.sourceforge.io/google_style.html

[22] YourKit, LLC, "Java API for creating, reading and updating dictionaries in WordNet format," https://extjwnl.sourceforge.net/, Accessed: 2025.

[23] Danny van Bruggen, "The most popular parser for the Java language," http://javaparser.org, Accessed: 2025.

[24] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[25] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[26] hub4j, "Github api for java," https://github.com/hub4j/github-api, 2025, accessed: 2025-01-25.

[27] R. Singh and N. S. Mangat, "Stratified sampling," in *Kluwer Texts in the Mathematical Sciences*. Kluwer Academic Publishers, 1996, vol. 15, pp. 102–144. [Online]. Available: https://doi.org/10.1007/978-94-017-1404-4_5

[28] Calculator.net, "Sample size calculator," Calculator.net, 2025, accessed: 2025-01-18. [Online]. Available: https://www.calculator.net/sample-size-calculator.html?type=1&cl=90&ci=15&pp=50&ps=100000&x=Calculate

[29] M. Motwani and Y. Brun, "Understanding why and predicting when developers adhere to code-quality standards," in *45th IEEE/ACM International Conference on Software Engineering: SEIP*. IEEE, 2023, pp. 432–444.