2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)

2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET) | 979-8-3503-2259-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEET58685.2023.00024

GRADESTYLE: GitHub-Integrated and Automated Assessment of Java Code Style

Callum Iddon
University of Auckland
Auckland, New Zealand
cidd131@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*—Every programming language has its own style conventions and best practices, which help developers to write readable and maintainable code. Learning code style is an essential skill that every professional software engineer should master. As such, students should develop good habits for code style early on, when they start learning how to program. Unfortunately, manually assessing students' code with timely and detailed feedback is often infeasible, and professional static analysis tools are unsuitable for educational contexts. This paper presents GRADESTYLE, a tool for automatically assessing the code style of Java assignments. GRADESTYLE automatically checks for violations of some of the most important Google Java Style conventions, and Java best practices. Students receive a report with a code style mark, a list of violations, and their source code locations. GRADESTYLE nicely integrates with GitHub and GitHub Classroom, and can be configured to provide continuous feedback every time a student pushes new code. We adopted our tool in a second-year software engineering programming course with 327 students and observed consistent improvements in the code style of their assignments.**

***Index Terms*—computing education, code style, programming courses, automated marking, GitHub, Java programming language**

I. INTRODUCTION

Enforcing code style conventions and best practices is paramount to have maintainable code that is less prone to errors. As such, the importance of good code style is well acknowledged by both academia and industry [1], [2].

Developing good habits for code style early on when learning to program is essential [3]. One difficulty is that novice students will not necessarily agree that "expert code" (which possesses better code style) is more readable than non-expert code—despite them recognising that it is expert code [4]. This presents a major challenge for instructors to get novice programmers on board to practice good code style [5], especially as student learning demands frequent and timely feedback that is too difficult to maintain when manually marking [6].

Code style should be marked to incentivise students to learn how to write high quality code. Otherwise, they would likely neglect code style in favour of the functionality aspect of an assignment. Indeed, numerous studies report that a wide range of code quality issues are prevalent in student code [7]–[9].

Automatic static analysis tools (e.g., CHECKSTYLE [10], PMD [11], and SONARQUBE [12]) have made valuable contributions in improving code quality for industry and open-source software projects [13]. However, these tools tend to be a challenge even for professionals to incorporate into their development workflow [14]. These tools also tend to be unsuitable for educational contexts [6], especially if the desire is to have them student-facing since they can be overwhelming to students [6]. In addition, static analysis tools were not intended to be used as educational tools, and thus they do not translate violations into a code quality mark.

In an educational context, automated feedback and assessment tools have tended to focus on *functionality* rather than code quality [2], [15]–[19]. The few tools that provide code style feedback to students have helped students improve code quality [5], [6], enabling scalability to support large numbers of students and submissions [20].

This paper presents **GRADESTYLE** [21], a code style marker tool for assignments in Java, which is one of the most popular programming language of choice for introductory programming courses [22]. GRADESTYLE detects 12 important categories of Java code style violations and best practice violations. GRADESTYLE relies on existing professional code style checkers (i.e., CHECKSTYLE and PMD) to detect certain categories of violations (e.g., formatting issues and code clones). However, it also implements novel detectors of other important categories of code style and best practices that are not supported by neither current professional code style checkers nor educational code style markers.

With the popularity of Git and educational solutions, such as GitHub Classroom, we also wanted to support programming courses as they increasingly use Git platforms to manage programming assignments [23]. As such, GRADESTYLE integrates with GitHub.

GRADESTYLE is publicly available under the AGPLv3 license, which allows instructors to use it in their courses and to extend it with new detectors of code style violations.

`https://github.com/Digital-Educational-Engineering/gradestyle`

II. GRADESTYLE

Figure 1 shows the logical workflow of GRADESTYLE. There are two alternative ways to use the tool: In the first way (Figure 1 left), the instructor decides when to run the automated assessment and send the reports to students. GRADESTYLE generates markdown reports, shared with students either by

979-8-3503-2259-0/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEET58685.2023.00024

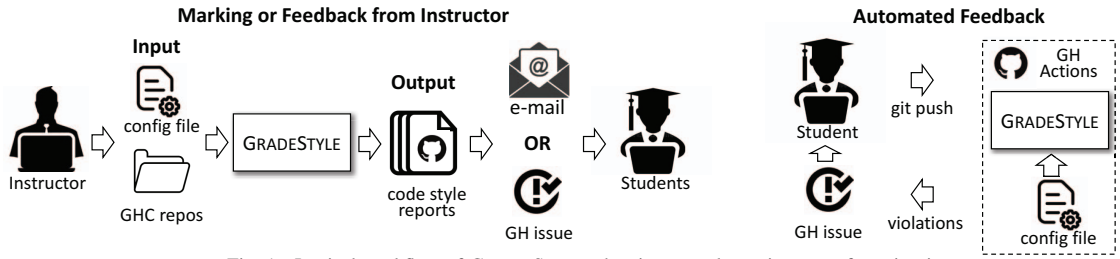


Fig. 1. Logical workflow of GRADESTYLE showing two alternative ways for using it.

email or by opening a GitHub issue in each student's repo. In this case, the automated assessment can either be a preliminary code style feedback, or the final code style mark (if the assignment is due). In the second way (Figure 1 right), GRADESTYLE is integrated in GitHub and provides continuous automated feedback. Every time a student pushes commits to the `main` branch, a GitHub action runs GRADESTYLE and automatically opens a GitHub issue with the violations, if any.

A. Input

GRADESTYLE runs with the command: `java -jar GradeStyle.jar config.properties`. The input configuration file specifies (i) the text messages to attach to each individual report, and (ii) the categories of violations to mark with their settings. If the instructor runs GRADESTYLE (Figure 1 left), the config file must also contain the path to a folder containing the students' repos. Such a folder can be automatically downloaded from GitHub Classroom using the provided scripts. By default, GRADESTYLE analyses the latest commit of the `main` branch. Nonetheless, it can also analyse student assignments that are not Git repos.

Every violation category shares three setting fields:

mode $= \{$`ABSOLUTE, RELATIVE`$\}$. `ABSOLUTE` means that the tool reports the raw number of violations for this category regardless of the code size. `RELATIVE` means that the tool uses the number of violations divided by the "size" of the assignments. This is to avoid penalising students with larger codebases. For example, if the violations of *VariableNames* is set to `RELATIVE`, the number of violations is adjusted considering the total number of declared variables. This avoids treating a student that has a few violations over a large number of variables the same as another student with the same number of violations over fewer variables.

scores $= \langle x_1, x_2, \cdots x_k \rangle \in \mathbb{N}$, where $x_i < x_{i+1} \forall i = 1 \cdots k-1$. k represents the maximum marks for the category. For example, let us assume that we set $k = 3 : \langle x_1, x_2, x_3 \rangle$. For each category, a student can get four possible marks: 3, 2, 1, or 0 (the higher, the better). The sequence of numbers $\langle x_1, x_2, x_3 \rangle$ specifies how GRADESTYLE calculates the marks based on the number of violations. Let `num-viol` denote the number of violations for that category. Students will get 3 marks (full marks) if `num-viol` $\leq x_1$, 2 marks if `num-viol` $> x_1$ and $\leq x_2$, 1 mark if `num-viol` $> x_2$ and $\leq x_3$, and 0 marks if `num-viol` $> x_3$. For example, given the scores $\langle x_1 = 4, x_2 = 6, x_3 = 8 \rangle$; if students make two violations, they would still get full marks (= 3) because $2 < x_1$, while if they make seven violations then they will get one mark.

This representation gives a high degree of flexibility. Instructors could set these thresholds higher for the first assignment to tolerate mistakes and boost students' confidence. Also, instructors could decide to give more weight (i.e., a higher k) to certain categories of violations, or give higher thresholds to those violations that are more difficult to avoid.

examples $= n$. The maximum number of examples of violations to show students in their feedback report. Each example is a GitHub hyperlink pointing to the exact line number containing the violation. Setting n relatively small (=10 or =20) avoids polluting the report with too many examples.

Besides these three settings, some categories have specific ones, which we explain below.

B. Code Style Violations

This subsection describes the code style violations currently implemented in GRADESTYLE. The two most popular and well-known coding conventions for Java are Oracle's (Sun's) Java Code Conventions [24] and Google's Java Style Guide [25]. They are mostly identical, except for a few differences. Between the two, we chose Google's because the last time Oracle updated the conventions was over 23 years ago (on April 20, 1999). Google's Java Style Guide (last updated in 2018) provides guides for modern Java features that were introduced after 1999, and recognises that Java's code style has evolved and matured over time [26].

We decided not to blindly check for all of Google's Java Style violations because that would be daunting for students, negatively affecting their learning and motivation [27]. Not every violation has the same severity level. As such, we selected a subset of them that we deem to be the most important ones contributing to code maintainability. We also included additional violations, not included in the Google's Java Style, representing best practices for Java and OOP programming.

Google's Java Style violations

Formatting Indentation, brace placement, and white spaces must follow Google's Java Style Guide [25]. A violation is generated for every line of code that disobeys a formatting rule. If a single line introduces multiple violations, GRADESTYLE reports multiple violations that reference the same line.

PackageNames Package names should contain lowercase letters only, and must follow the directory struc-

ture. For example, the Java class `A.java` located in `src/com/domain/A.java` must have as package declaration `package com.domain;`. A violation is every package name that does not satisfy this convention.

ClassNames Class and enum names should be in UpperCamelCase (the first letter of each internal word capitalised). Class names should be nouns or noun phrases. For example, `Car` or `ArrayList`. Following Google's Java code style, class names can start with an adjective followed by a noun (e.g., `LinkedList` or `ImmutableList`). A violation is every class declaration whose name does not satisfy this convention.

MethodNames Method names should be verb phrases in lowerCamelCase (the first letter lowercase, with the first letter of each internal word capitalised). A violation is every method declaration whose name that does not satisfy this convention.

VariableNames Variable names (including instance fields, parameters, and local variables) should be in lowerCamelCase, and should not start with underscore _ or dollar sign $ characters, even though both are allowed. Names representing variables that are both `static` and `final` must be all uppercase using underscores to separate words (e.g., `MAX_ITERATIONS, COLOR_RED`). A violation is every variable name that does not satisfy the above conventions.

JavaDoc GRADESTYLE verifies that every public method, class, and enum has its own JavaDoc, and every JavaDoc should contain at least `minWords`. We do not require JavaDoc for instance fields as they should always be private (see the *PrivateMembers* category). The JavaDoc of methods should have entries for the parameter list (with `@param` tags), the return value (with `@return` tags), and any throw exceptions (with `@throws` tags). These entries should be consistent with the methods' signatures. A violation is every method, class, and enum without a JavaDoc or each line in a method's JavaDoc that contains inconsistency among the `@param`, `@return`, and `@throws` tags and the corresponding method's signature.

Java's Best Practices

Commenting This category has two levels of granularity:

Line level Good code style demands that code comments provide additional information that is not readily available in the code itself. GRADESTYLE checks this by computing the Levenshtein distance [28] between every line of comment and the associated line of code (i.e., the line of code immediately under the comment, or on the same line). Such a distance measures the number of edits needed to transform one string into another [28]. This category requires an additional setting (`levenshteinDistance`) that specifies the minimum allowed Levenshtein distance between a comment and its associated line of code. For example, consider the following:

```
isOpen = false; // set isOpen to false
```

This raises a violation as the distance between `isOpen = false;` and "set isOpen to false" is 8, which is lower than the minimum allowed (e.g., 10). A violation is every line of comment that is too similar to its associated line of code.

Method level It is unlikely that every line in a method needs to be commented or that long methods do not need any comments. As such, this category requires additional settings that define a lower and upper bound on the percentage of commented lines of code in a method. A violation is every method that contains comments for less than `minFrequency` or more than `maxFrequency` of the methods' total lines of code. For marking our students, we found that `minFrequency` $= 10\%$ and `maxFrequency` $= 70\%$ work reasonably well. However, short methods (e.g., getters and setters) often do not require comments. Therefore, this category needs an additional setting (`minLines`) that specifies the minimum lines of code that a method must have to be checked against the criteria above.

PrivateMembers An important OO design principle is information hiding. Instance fields of a class should be `private` or `protected`. Other classes should use `public` getters/setters to access them. This protects data from unwanted access.

Ordering The order of elements inside a class declaration has a great effect on code *readability* [25]. However, there is no single correct way for how to do it. In fact, Google's Java Style acknowledges the importance of ordering class elements, but does not impose any order [25]. We chose a popular order of elements, which instructors can change according to their preference: (i) inner classes, (ii) static fields, (iii) static methods, (iv) instance fields, (v) constructors, and (vi) instance methods.

StringConcatenation Frequent string concatenation decreases runtime performance significantly. Cumulative string concatenation in a loop, like the following, should be avoided:

```
String nums = "";
for (int i = 0; i < 100000; i++) {
  nums += " " + i;
}
```

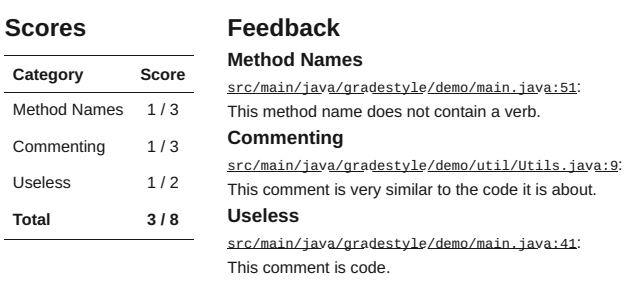
The reason is that Java instantiates a new `String` object at every loop cycle. One can achieve the same result with a `StringBuilder`, avoiding needless object creations:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100000; i++) {
  sb.append(" ").append(i);
}
String nums = sb.toString();
```

On a 2.9 GHz Intel i7, the first code snippet takes 4,010 ms to run, while the second only 7 ms. A violation is every String concatenation inside a loop (i.e., `for`, `do`, or `while`).

Useless High quality source code should be free of *useless* code that does not play any role in the runtime behaviour of the program. GRADESTYLE checks for the presence of commented code and unused imports, methods, and variable declarations. A violation is every line of code that contains useless code.

Clones A code clone is a code fragment that is identical or similar to another [29]. Code clones are considered harmful for two main reasons. First, multiple duplicates of code increase maintenance costs. Second, inconsistent changes to cloned code introduce faults. There are four types of clones [29]: *Type-1* Exact copy, the only differences are in white space and



Scores

Category	Score
Method Names	1 / 3
Commenting	1 / 3
Useless	1 / 2
Total	**3 / 8**

Feedback

Method Names

src/main/java/gradestyle/demo/main.java:51:
This method name does not contain a verb.

Commenting

src/main/java/gradestyle/demo/util/Utils.java:9:
This comment is very similar to the code it is about.

Useless

src/main/java/gradestyle/demo/main.java:41:
This comment is code.

Fig. 2. Example of GRADESTYLE's student report.

comments. *Type-2* Same as type 1, but also variable renaming. *Type-3* (near-miss clones) Same as type 2, but also changing or adding a few statements. *Type-4* (semantic clones) Semantically identical, but not necessarily the same syntax.

These types are increasingly challenging to detect, with semantic clone (type-4) being the most complex one. GRADESTYLE detects near-miss clones (type-3) because semantic clone detection leads to many false alarms [30]–[32]. This category requires an additional setting (`tokens`) that specifies the minimum number of identical tokens that two code fragments must have to be considered as near-miss clones. A violation is every code fragment that is a near-miss clone.

C. Output

The output of GRADESTYLE is twofold: (i) a folder containing the personalised code style reports for each of the students, and (ii) a CSV file where each row breaks down, for each student, the number of violations per category. The last column of each row gives the code style mark, which is the sum of the scores of each category specified in the config file. Such a CSV file facilities uploading the marks to Learning Management Systems (e.g., Canvas, Moodle).

Figure 2 shows an example report that GRADESTYLE automatically generates. The beginning of the report gives a table summarising the scores for each category. It follows the instances of the detected violations (including the exact location of the violation in the code via the GitHub link.

nurturing their ability to self-regulate their learning [33].

To showcase the violations that GRADESTYLE detects, we created this GitHub repository `https://github.com/Digital-Educational-Engineering/gradestyle-demo-java`. The repository is based on one of our student assignments and contains instances of all the detectable violations. The automatically generated code style report for this repository is: `https://github.com/Digital-Educational-Engineering/gradestyle-demo-java/issues/1`.

D. Implementation

We implemented the majority of GRADESTYLE's novel violation detectors using JAVAPARSER [34] (v. 3.23.1), a library to analyse Java source code and identify the language constructs of interest (e.g., variable names for the *VariableNames* category).

To detect violations of the *ClassNames* and *MethodNames* categories, we use EXTJWNL [35] (v. 2.0.5) in combination with regex expressions. The regex expressions check for camelcase notation in the extracted names and EXTJWNL checks for the proper use of nouns and verbs in class and method names, respectively. In particular, GRADESTYLE relies on the Part of Speech tagger (PoS) of EXTJWNL to identify a word's function (e.g., subject, article, verb) in a class or method name.

GRADESTYLE also relies on the APIs of CHECKSTYLE [10] (v. 10.3.2) to implement three detectors (i.e., *Formatting*, *PackageNames*, and *JavaDoc*). For the code clone detection we used PMD [11] (v. 6.45.0) COPY/PASTE DETECTOR (CPD), configuring it for Type-3 code clones. To read the GitHub repos and to open GitHub issues, GRADESTYLE relies on the libraries JGIT [36] (v. 6.0.0) and GITHUB API [37] (v. 1.3).

III. IMPACT DISCUSSION

This section discusses our application of GRADESTYLE in a second-year object-orientated programming course from 1,488 submissions made by 327 students.

A. Course Context

The course has four programming based assignments for students to apply theoretical components in a practical environment. In previous years, teaching assistants manually marked the code style of each submission and gave individual feedback. Feedback would not be given to students for up to six weeks after submission due to the high volume of marking. Students would also only receive a brief list of issues that were found in their code, if any. Feedback would often be ignored due to the late delivery and vague descriptions of the issues [38].

In 2022, we used GRADESTYLE for formative and summative feedback of code style for all four assignments. Code style contributed a total of 5.5% towards the final grade.

The first assignment (A1) was a first introduction to OO programming concepts and the Java language. The second assignment (A2) required students to model a problem using OO principles. Assignment three (A3) required students to apply design patterns to their model of an OO problem. The fourth assignment (A4) required students to design data structures to which they apply a search algorithm. We evaluated students using the *Formatting*, *ClassNames*, *MethodNames*, *VariableNames*, and *Commenting* categories for all assignments. A2, A3, and A4 also included the *PrivateMembers* and *Ordering* categories. The *Useless* category was included for A3 and A4. We added additional categories throughout the course to allow students to focus on a few aspects of their code and gradually introduce new code quality attributes.

We used GitHub Classroom (GHC) to manage the assignments. For each assignment, GHC created the students' repositories and initialised them with starter code that we provided to them. Such starter code gives skeleton classes with missing functionality. To finish the assignment, students had to complete the skeleton classes and create new classes and methods. We gave around four weeks to submit each assignment. A few days before the deadline, we ran GRADESTYLE on the latest commit and provided the feedback report to students.

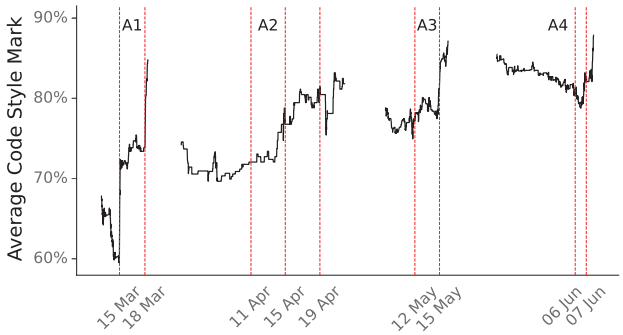


Fig. 3. Average code style mark over time for the four assignments. Red vertical lines indicate when we released the code style feedback.

B. Evaluation Setup

To evaluate the impact of GRADESTYLE on students' learning, we analysed every commit of all assignments to track the average code style mark throughout the semester, totaling 22,364 Git commits. We included in the results only students who had made a *substantial contribution* to their assignment before receiving the first preliminary feedback. The contribution requirement ignores the students that started after the first feedback was released. Also, it ignores the code style marks of the initial commits, which are disproportionately high as the starter code had no violations and students had not made many changes. We measured a student's contribution by the cumulative total of insertions and deletions to their GitHub repository. We define a substantial contribution one with at least 100 changes for A1, and 600 for the remaining assignments. We reduced the contribution requirement for A1 as it was a significantly smaller assignment. To report average marks of a sufficiently large number of students, we report the average after 75% of students had made a significant contribution.

C. Results and Discussion

Figure 3 shows the results. Each black line represents the average code style marks over time for each assignment. Each line starts when at least 75% of students made a substantial contribution, and ends when the assignment was due. Vertical red lines indicate when we released the formative feedback.

The average code style mark increases after each feedback report. The impact of the feedback for A1 is more pronounced as it was the first feedback students received for code style. The minimum average mark increases from 59.11% (A1) to 69.66% (A2), 74.95% (A3), and 79.02% (A4) throughout the course despite the introduction of additional categories. The final average marks of 84.59% (A1), 81.82% (A2), 86.52% (A3), and 87.92% (A4) show an upward trend. The exception of the final average mark for A1 being higher than A2 was due to the lower number of categories considered for A1. The average mark after 75% of students had made a significant contribution (the beginning of each black line) increased from 68.06% (A1) to 74.22% (A2), 78.36% (A3), and 85.56% (A4). This demonstrates the lasting impact of GRADESTYLE on students as they maintained better code quality before receiving any feedback in later assignments.

To analyse the largest assignment, A4, GRADESTYLE takes on average 741 ms per commit. Such a low computational cost allows the deployment of GRADESTYLE in Massive Open Online Courses (MOOCs) with thousands of students. Also, when used with GitHub Actions, GRADESTYLE will not easily consume all of the GitHub Actions budget.

IV. RELATED WORK

Professional software developers often rely on static analysis tools (informally called "linters") to verify that their code follows an expected set of code conventions. Some of the popular examples include CHECKSTYLE [10], PMD [11], SONARQUBE [12], ERRORPRONE [39], and SPOTBUGS [40]. While such tools are popular and contribute to improving code style, they are unsuitable for educational contexts [5], [27], [41]. Indeed, even when provided with linters that identify code quality issues, students will often neglect the recommendations [42]. The reason is that such tools report a manifold of violations that can overwhelm students [6], [41]. Moreover, linters do not translate violations into a code style score. Such a score is essential for students to track their progress and for instructors to aid automated assessment.

To address such inadequacies, DROP PROJECT [41], PROGEDU [5], CSS [43], and HYPERSTYLE [27] report code style violations in a way that is more suitable for students who are learning programming. These tools are built on top of professional linters (such as CHECKSTYLE or PMD) but they change how they report violations to students, often including a code style mark. GRADESTYLE belongs to such a category of tools. Similarly, it leverages existing linters for detecting some categories of violations. However, GRADESTYLE differs from these tools by detecting violation categories beyond those detected by existing professional linters. For example, linters detect class and method names that are not UpperCamelCase and lowerCamelCase, respectively. However, differently from GRADESTYLE, they do not employ natural language processing to verify the correct usage of noun or verb phrases. Moreover, GRADESTYLE implements novel detectors guided by Java best practices. For example, it detects comments that are too syntactical similar to the associated line of code, and string concatenations inside loops.

Another key difference of GRADESTYLE compared to most of the above tools is in its workflow—it aims to seamlessly integrate in courses using Git, such as GitHub Classroom.

V. CONCLUSION

Promoting code style awareness is crucial for turning students into professional developers. This paper presented GRADESTYLE, a tool for the automated assessment and feedback of Java code style. Our experience of using GRADESTYLE on a class of 327 undergraduate students shows that the number of violations decreased over time. The feedback from the tool contributed (at least in part) to this downtrend.

In the future, we will support other popular programming languages, and show the evolution of code style over time.

REFERENCES

[1] A. Tornhill and M. Borg, "Code red: the business impact of code quality - a quantitative study of 39 proprietary production codebases," in *TechDebt '22: International Conference on Technical Debt, Pittsburgh Pennsylvania, May 17-18, 2022* (N. A. Ernst, V. Lenarduzzi, and T. Sharma, eds.), pp. 11–20, ACM, 2022.

[2] H.-M. Chen, B.-A. Nguyen, Y.-X. Yan, and C.-R. Dow, "Analysis of learning behavior in an automated programming assessment environment: A code quality perspective," *IEEE Access*, vol. 8, pp. 167341–167354, 2020.

[3] K. Ala-Mutka, T. Uimonen, and H.-M. Jarvinen, "Supporting students in C++ programming courses with automatic program style assessment," *Journal of Information Technology Education: Research*, vol. 3, pp. 245–262, January 2004.

[4] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking code readability, structure, and comprehension among novices: It's complicated," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 84–94, 2019.

[5] H.-M. Chen, W.-H. Chen, and C.-C. Lee, "An automated assessment system for analysis of coding convention violations in Java programming assignments.," *J. Inf. Sci. Eng.*, vol. 34, no. 5, pp. 1203–1221, 2018.

[6] J. C. Paiva, J. P. Leal, and A. Figueira, "Automated assessment in computer science education: A state-of-the-art review," *ACM Trans. Comput. Educ.*, vol. 22, jun 2022.

[7] S. H. Edwards, N. Kandru, and M. B. Rajagopal, "Investigating static analysis errors in student Java programs," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, (New York, NY, USA), p. 65–73, ACM, 2017.

[8] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code," in *Proceedings of the 20th Australasian Computing Education Conference*, ACE '18, (New York, NY, USA), p. 73–82, ACM, 2018.

[9] P. Ardimento, M. L. Bernardi, and M. Cimitile, "Software analytics to support students in object-oriented programming tasks: An empirical study," *IEEE Access*, vol. 8, pp. 132171–132187, 2020.

[10] "Checkstyle." https://checkstyle.org, Accessed: Oct 2022.

[11] "PMD." https://pmd.github.io, Accessed: Oct 2022.

[12] "SonarQube." https://www.sonarqube.org, Accessed: Oct 2022.

[13] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 38–49, 2018.

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *35th International Conference on Software Engineering*, pp. 672–681, 2013.

[15] H. Keuning, B. Heeren, and J. Jeuring, "A tutoring system to learn code refactoring," in *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, (New York, NY, USA), p. 562–568, ACM, 2021.

[16] R. Luukkainen, J. Kasurinen, U. Nikula, and V. Lenarduzzi, "ASPA: A static analyser to support learning and continuous feedback on programming courses. an empirical validation," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 29–39, 2022.

[17] L. C. Ureel II and C. Wallace, "Automated critique of early programming antipatterns," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, (New York, NY, USA), p. 738–744, ACM, 2019.

[18] N. Körber, K. Geldreich, A. Stahlbauer, and G. Fraser, "Finding anomalies in scratch assignments," in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2021, Madrid, Spain, May 25-28, 2021*, pp. 171–182, IEEE, 2021.

[19] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: an approach based on formal semantics," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019, Montreal, QC, Canada, May 25-31, 2019* (S. Beecham and D. E. Damian, eds.), pp. 126–137, IEEE / ACM, 2019.

[20] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, 2018.

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

[22] J. Hong, "The use of java as an introductory programming language," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 4, no. 4, pp. 8–13, 1998.

[23] Y.-C. Tu, V. Terragni, E. Tempero, A. Shakil, A. Meads, N. Giacaman, A. Fowler, and K. Blincoe, "Github in the classroom: Lessons learnt," in *Australasian Computing Education Conference*, ACE '22, (New York, NY, USA), p. 163–172, ACM, 2022.

[24] Oracle, "Oracle's code conventions for the java programming language." https://www.oracle.com/java/technologies/javase/codeconventions-contents.html, Accessed: Oct 2022.

[25] Google, "Google java style guide." https://google.github.io/styleguide/javaguide.html, Accessed: Oct 2022.

[26] 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.

[27] A. Birillo, I. Vlasov, A. Burylov, V. Selishchev, A. Goncharov, E. Tikhomirova, N. Vyahhi, and T. Bryksin, "Hyperstyle: A tool for assessing the code quality of solutions to programming assignments," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*, (New York, NY, USA), p. 307–313, ACM, 2022.

[28] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[29] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 485–495, IEEE, 2009.

[30] V. Saini, F. Farmahinifarahani, H. Sajnani, and C. Lopes, "Oreo: Scaling clone detection beyond near-miss clones," in *Code Clone Analysis*, pp. 63–74, Springer, 2021.

[31] R. Tiarks, R. Koschke, and R. Falke, "An assessment of type-3 clones as detected by state-of-the-art tools," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 67–76, IEEE, 2009.

[32] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *2011 18th Working Conference on Reverse Engineering*, pp. 13–22, IEEE, 2011.

[33] C. Ott, A. Robins, and K. Shephard, "Translating principles of effective feedback for students into the cs1 context," *ACM Trans. Comput. Educ.*, vol. 16, jan 2016.

[34] Danny van Bruggen, "The most popular parser for the Java language." http://javaparser.org, Accessed: Oct 2022.

[35] YourKit, LLC, "Java API for creating, reading and updating dictionaries in WordNet format." https://extjwnl.sourceforge.net/, Accessed: Oct 2022.

[36] PMD Team, "JGit: Java library implementing the Git version control system:." https://www.eclipse.org/jgit, Accessed: Oct 2022.

[37] Kohsuke Kawaguchi, "GitHub-API for Java." https://github-api.kohsuke.org, Accessed: Oct 2022.

[38] D. Salter, "The challenge of feedback: Too little too late," in *Proceedings of EdMedia + Innovate Learning 2008* (J. Luca and E. R. Weippl, eds.), (Vienna, Austria), pp. 3925–3926, Association for the Advancement of Computing in Education (AACE), June 2008.

[39] "Error Prone." http://errorprone.info. Accessed: October 2022.

[40] "SpotBugs." https://spotbugs.github.io. Accessed: October 2022.

[41] B. P. Cipriano, N. Fachada, and P. Alves, "Drop Project: An automatic assessment tool for programming assignments," *SoftwareX*, vol. 18, pp. 1–7, 2022.

[42] H. Keuning, B. Heeren, and J. Jeuring, "Code quality issues in student programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2017, Bologna, Italy, July 3-5, 2017*, pp. 110–115, ACM, 2017.

[43] O. Karnalim and Simon, "Promoting code quality via automated feedback on student submissions," in *IEEE Frontiers in Education Conference, FIE 2021, Lincoln, NE, USA, October 13-16, 2021*, pp. 1–5, IEEE, 2021.