Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments

Anastasiia Birillo
JetBrains Research
anastasia.birillo@jetbrains.com

Ilya Vlasov
Saint Petersburg State University
ilyavlasov2011@gmail.com

Artyom Burylov
Stepik
Miro
avburylov@gmail.com

Vitalii Selishchev
Computer Science Center
vvselishchev@gmail.com

Artyom Goncharov
Computer Science Center
artyom.goncharov1@gmail.com

Elena Tikhomirova
JetBrains Research
elena.tikhomirova@jetbrains.com

Nikolay Vyahhi
Stepik
vyahhi@stepik.org

Timofey Bryksin
JetBrains Research
Saint Petersburg State University
timofey.bryksin@jetbrains.com

ABSTRACT

In software engineering, it is not enough to simply write code that only works as intended, even if it is free from vulnerabilities and bugs. Every programming language has a style guide and a set of best practices defined by its community, which help practitioners to build solutions that have a clear structure and therefore are easy to read and maintain. To introduce assessment of code quality into the educational process, we developed a tool called *Hyperstyle*. To make it reflect the needs of the programming community and at the same time be easily extendable, we built it upon several existing professional linters and code checkers. *Hyperstyle* supports four programming languages (Python, Java, Kotlin, and Javascript) and can be used as a standalone tool or integrated into a MOOC platform. We have integrated the tool into two educational platforms, Stepik and JetBrains Academy, and it has been used to process about one million submissions every week since May 2021.

CCS CONCEPTS

• **Social and professional topics** → **Student assessment**; • **Applied computing** → *Education*; • **Human-centered computing** → Interactive systems and tools.

KEYWORDS

programming education, code quality assessment, learning programming, refactoring, code formatting

ACM Reference Format:
Anastasiia Birillo, Ilya Vlasov, Artyom Burylov, Vitalii Selishchev, Artyom Goncharov, Elena Tikhomirova, Nikolay Vyahhi, and Timofey Bryksin. 2022. *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 V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499294

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499294

1 INTRODUCTION

Code quality is an important aspect in software development [32]. Poor coding style may result in writing incomprehensible code that is difficult to maintain and test [25]. Creating code quality awareness is an important step in preparing programming students to work as professional developers [21, 28, 29]. However, many students do not pay enough attention to code quality, since the main goal for them is to submit a solution that passes all the tests [28]. They rarely research more complex solutions or best practices [28, 29] and may make the same mistakes over again. Therefore, learners may need an incentive to improve their coding style [28].

Code quality can be checked manually or with special tools. During manual evaluation, the teacher takes into account the task context and difficulty, and can provide personalized feedback [29]. However, it scales rather poorly. Programming tasks in massive open online courses (MOOC) cannot be assessed manually because of the sheer amount of resources needed to adequately evaluate each student's performance [37]. Although there exist a lot of tools for automatic code quality assessment (linters), most of them have not been adapted to the learning process: they aim to detect bugs [26], code smells [23], or vulnerabilities [27] in small to large codebases rather than individual projects or single-file solutions. Such tools usually have high threshold values in their default settings, so minor issues such as small duplicated blocks of code are usually not reported [29]. Moreover, professional code analysis tools naturally do not aim to track students' progress or provide detailed educational feedback which is crucial for the educational process [29].

On the other hand, existing research tools [20, 22, 30, 38] designed for assessment of solutions to programming assignments do include only relevant checks and provide feedback which is detailed enough for learners. However, most such tools support just one programming language and a limited number of embedded tasks, mainly for beginners. Finally, several researchers have suggested that the better way of creating code quality assessment tools is not to write one's own validators but to reuse inspections of several real-world analyzers [35] and adapt them to the educational process [29].

In this work, we propose *Hyperstyle* [7], a tool for automated assessment of the code quality of programming solutions that are written in Python, Java, JavaScript, and Kotlin. With this tool, we aim to find areas of improvement in programs that pass all the necessary tests but still have weaknesses in terms of readability, maintenance, and complexity.

Hyperstyle adapts existing professional code analyzers to the content and goals of programming assignments. In addition, it takes into account students' history of solutions to indicate repeated mistakes. All code quality issues have a difficulty level assigned. Thus, a teacher or a student can get only code quality issues that satisfy the desired level of programming experience. *Hyperstyle* is flexible and can be extended by other researchers and practitioners according to their needs.

We integrated *Hyperstyle* into two online educational platforms: Stepik [16] and JetBrains Academy [8]. Within these platforms, more than one million code fragments are submitted by students each week. Previously, these platforms only used an automated task validation system that indicated how many tests were passed for a particular submission, but with *Hyperstyle* the feedback was enriched also with code quality reports.

To sum up, with this work we make the following contributions:

- We implement *Hyperstyle*, a tool for automated evaluation of the code quality of solutions to programming assignments that could be used in MOOCs to provide detailed adapted feedback to programming students. The tool mainly targets Python and Java, but can also work with Kotlin and JavaScript. Its source code is open and available on GitHub [7]. To simplify the deployment process, we also provide a Docker image [6] with the tool. All research artifacts and supplementary materials are available online [1].
- We provide a dataset of 107 solutions to six programming tasks in Java created by 17 students with different programming experience. We use this dataset to compare the performance of *Hyperstyle* with *Tutor* [30], the only similar tool with an open implementation.
- We evaluate the tool's impact on real students from the Stepik and JetBrains Academy education platforms by comparing the median number of code quality issues of 300 Python and 294 Java learners before and after the tool's integration. In total, we collected 24,250 submissions for Python and 5,192 for Java. This dataset is also open and available [1].

2 BACKGROUND

In this section, we provide an overview of professional code quality assessment tools and discuss key points related to adapting such tools to education.

2.1 Professional Tools

The first group of code quality assessment tools is used in the development of software projects and provides warnings about typical problems. In some cases, the tools propose relevant code fixes as well. They can be divided into several categories:

(1) Code quality checks and automated fixes provided by integrated development environments (IDEs), such as *IntelliJ IDEA*, *Visual Studio*, or *Eclipse*.
(2) Analyzers (linters), *e.g.*, *flake8* [5] or *Pylint* [13] for Python, *PMD* [12] or *Checkstyle* [2] for Java. They work fast, but are usually language-specific and do not perform deep checks, for instance, involving control flow or data flow analysis. However, such tools are used pervasively across the industry because they are easy to integrate, trustworthy [36], and provide actionable output.
(3) Code security and quality platforms, such as *Codacy* [4], *SonarQube* [15], or *Qodana* [14]. These tools allow software developers to integrate project-level checks locally, on build servers, or other remote resources.

The main goal of professional tools is to prevent the appearance of inefficient, complex, or vulnerable code [35]. However, most checks performed by such static analysis tools cannot be applied directly to the assessment of solutions to programming assignments. The main reason is that students' solutions are usually quite small and therefore do not require most checks designed to ensure the validity and security of industry-level projects [18, 28, 37]. For example, such errors can be related to performance in high-loaded systems and are very difficult to fix in the education process. So it seems that checks such as `AvoidFileStream`[1] for PMD [12] that allows managing garbage collection should be disabled for student code. In addition, getting a large number of complex code quality issues can overwhelm and discourage students [29].

Another important requirement of the educational setting that is not met by professional code quality assessment tools is tracking progress: students may make the same mistakes over again and disregard coding best practices or programming language features [28]. Professional tools do not consider such educational aspects because real-world projects usually have other goals.

2.2 Education-Oriented Tools

The issue of assessing code quality in education has been actively studied before [21, 28, 29]. Several tools have been created that partially reimplement code quality checks from professional solutions that are relevant to learning needs [20, 22, 30, 38].

FrenchPress [20] is an Eclipse plugin that displays adapted messages for a small set of common code style inconsistencies. *WebTA* [38] is a web-based tool that reports failed tests, errors common to novices, and stylistic issues. The tool provides students with adapted feedback that is more appropriate for their level of understanding.

Choudhury et al. [22] and, several years later, Keuning et al. [30] proposed two very similar tools: *AutoStyle* and *Tutor*, respectively. These tutoring systems let students practice with improving small programs that are already functionally correct. The systems are implemented as special environments that display code quality hints step by step and let students improve the code.

[1] `AvoidFileStream` inspection in PMD: https://pmd.github.io/pmd-6.36.0/pmd_rules_java_performance.html#avoidfilestream

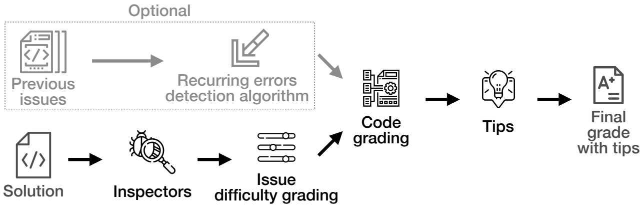

Figure 1: Workflow of *Hyperstyle*.

To sum up the benefits, assessment tools for education apply checks that are relevant to the students' skill levels and adapt the requirements accordingly. In addition, all of them adapt the reports to the needs of the learners by providing tips and explanations. Some of them allow tracking students' progress.

However, existing education-oriented tools also have common limitations: they are focused on only one programming language each (mostly Java) and often handle only basic errors typical of novice programmers. Another important drawback is that these tools are difficult to extend for handling more complex errors, since they implement their own validators and do not reuse selected checks from professional analyzers.

3 WORKFLOW OF *HYPERSTYLE*

Hyperstyle is implemented as a Python tool. It currently targets solutions in Python and Java, but can also work with Kotlin and JavaScript.

Figure 1 presents the workflow of *Hyperstyle*. The first step of handling a student solution is finding all code quality problems. Each supported language can have several inspectors. Then, depending on the selected difficulty level, output only by relevant inspections is taken into account. After that, all found errors are aggregated and the final grade is calculated. If a history of the student's previous errors is provided, the current solution can be penalized if some errors are repeated over a certain period of time. This step is optional and is disabled if no solution history is provided. In the end, the least understandable explanations from the analyzers are replaced with more detailed ones.

As input, the tool takes code fragments and optional additional information, *e.g.*, programming language, history of previous inspections, and so on. *Hyperstyle* returns the results in the JSON format, which could then be displayed by MOOC platforms in any way suitable: integrated with other user data, highlighted, or omitted if necessary. The report contains the following information by default: an overall score, a code quality summary, a list of errors by categories with feedback messages, inspections that contributed to penalizing the score (if this happened). For each error, the tool reports the exact position (number of code line) in the solution.

The rest of this section describes all key concepts and steps of the workflow in more detail.

3.1 Inspectors and Error Categories

Code quality issues can be divided into several categories. In our work, we have identified five main semantic categories of errors. The categorization is based on the error types used in most of the code inspectors we reviewed and on our own experience:

- *Code style*: The code fragment violates one of the rules of the commonly accepted style guide for the chosen language. By fixing such issues, students learn to follow popular coding conventions, like PEP-8 [11] for Python or Oracle Java code conventions [10] for Java.
- *Code complexity*: The solution is poorly designed or overly complicated. By fixing such issues, students learn to make code easier for understanding, editing, and debugging.
- *Error-proneness*: The code contains a potential bug. Even if the code passes all automatic tests, it may behave incorrectly in some cases, which would be a problem in a real-world environment. By fixing such issues, students learn to write reliable code.
- *Best practices*: The code does not follow the widely accepted recommendations and idioms of the chosen language. Some features of the language can be used in an inefficient or obsolete way. By fixing such issues, students learn to use language features correctly.
- *Minor issues*: The code contains problems usually related to incorrect spelling. These problems are worth solving because they hinder the readability of the code. For such errors, the final grade is not reduced.

We provide examples of all such error types in the supplementary materials [1].

For each language, we selected analyzers that find problems from the categories described above. From each analyzer, we manually selected checks that are relevant to typical programming tasks in MOOCs and categorized them. The list of supported languages, used linters, and the number of unique checks in each error category are provided in Table 1.

Category Language	CS	CC	EP	BP	MI	Total
Python (flake8, pylint, radon)	146	35	162	254	3	**600**
Java (Checkstyle, PMD)	50	8	51	110	3	**222**
Kotlin (Detekt)	70	12	21	75	0	**178**
JavaScript (ESlint)	17	1	15	34	0	**67**

Table 1: Number of unique error checks per language and category: CS — Code style, CC — Code complexity, EP — Error-proneness, BP — Best practices, MI — Minor issues.

Hyperstyle is easily extendable. Adding support for a new language requires just implementing an inspector module for this language. The implementation of each inspector includes code running the linter, parsing its result, and categorizing all errors into the five categories mentioned above.

3.2 Code Grading

Four grades are awarded by the tool:

- *EXCELLENT* means that the code strictly follows the style guide, does not have complexity or error-proneness issues, and is easy to read and modify.

- *GOOD* means that the code is readable and relatively easy to edit, maintain and extend, but still contains some minor code quality issues.
- *MODERATE* means that the code follows the style guide only partially, most of language features are not used correctly. Sometimes the code may be challenging to understand.
- *BAD* means that the code is hard to read and modify and probably does not follow the style guide. Also, the solution might have high complexity and might be error-prone.

Note that the *EXCELLENT* grade does not guarantee the complete absence of errors. However, it does certify that the code does not include any *common* code quality errors in this language.

All errors detected by the tool can be divided into two types by how they contribute to the intermediate score within their semantic category:

- *Countable*: All instances of such error type are counted. If the number of errors reaches certain threshold values, the respective score is assigned. An example of this kind of errors is the number of places in the code where the `for` loop can be replaced with `forEach`.
- *Measurable*: The severity of such errors is evaluated within a scale or an interval. Reaching certain threshold values within the scale impacts the score. An example of this kind of errors is the length of lines in the code.

The thresholds of values were selected iteratively. First, we divided each error category into several subcategories and came up with thresholds for them. Each subcategory contains only one kind of issues: either only countable issues or only a measurable issue. Each measurable issue makes its own subcategory and measures the same metric value, *e.g.,* metrics *Length of a Code Line* and *Number of Code Lines in a Function* are semantically different. At the same time, several countable issues can be placed into one subcategory, *e.g.,* `for` *Loop can be Replaced with* `forEach` and *Unnecessary Local Variable Before* `return` are both from the one category (*Best practices*). All subcategories with their description can be found in the supplementary materials [1]. The final grade is calculated as the minimum of the grades for each subcategory.

The initial thresholds were proposed for each subcategory and each language by three experts with more than four years of programming experience each. Then the experts ran *Hyperstyle* on 50 thousand student solutions with the proposed thresholds and randomly checked 100 code fragments to verify that the assessment worked as expected. After that, several thresholds, mostly from the *Code complexity* and *Error-proneness* categories, were modified. These thresholds were lowered since students' solutions are smaller and easier than real-world projects. After that, we integrated *Hyperstyle* into the Stepik and JetBrains Academy platforms. At this stage, a test group of students reported unreasonably low grades, and the initial group of experts checked each report manually and decided to decrease (or keep) thresholds values.

The final thresholds were evaluated by an empirical analysis of 250 thousand solutions for each language. In total, one million solutions were analyzed for all four languages. For each solution, we calculated the metrics for measurable errors and the frequency of countable errors within each subcategory. We plotted distributions of these errors by subcategories, indicated the thresholds and checked how many solutions received each of the grades. If a lot of submissions had a number of issues that was lower or higher than the selected thresholds, the initial group of experts manually looked at several fragments to check whether the boundaries were set too high or too low. Finally, the threshold values were modified according to this empirical analysis. Examples of error distribution charts are provided in the supplementary materials [1].

To illustrate this process, let us consider several insights about Python solutions. In addition to constructing plots, we calculated the percentage of the number of students solutions that correspond to the thresholds and the grades. For example, a software metric called *Maintainability Index* [9] did not influence the grade in most cases (99.29%) because it was not applicable to students' solutions, which were mostly small in size. We also discovered that the distributions of errors from the *Best practices* and *Error-proneness* categories were very similar and probably should have the same thresholds, *e.g.,* for the *Best practices* category these values are: EXCELLENT — 88.3%, GOOD – 8.2%, MODERATE — 3.4%, and zero for the BAD grade. Also, grades for the *Length of Bool Expressions* subcategory had very low thresholds and 97.93% of solutions had the highest grade in this subcategory.

3.3 Detection of Recurring Errors

When solving programming problems, students often make the same mistakes over and over again [28]. We also noticed this while analyzing students' solutions gathered from the Stepik and JetBrains Academy platforms, so we developed an algorithm to detect such recurring errors and show them to the students. The algorithm analyzes recent solutions in a programming language by a particular student and finds errors that are identical to the ones reported for the current solution.

However, if a student's score is repeatedly decreased, they may become demotivated and upset, since the real reason for repeating an error may be the lack of understanding of this error [31]. To avoid this problem, based on existing analysis and research [19, 21, 24, 28] we designed different penalty rates for different subcategories of errors. The final coefficient for each error subcategory is calculated by three criteria:

- *Prevalence*: How common the mistake is among students. The most frequent mistakes [19, 24] are about code formatting (*e.g.,* incorrect brackets or indentation) and best practices of more experienced programmers (*e.g.,* using `enumerate` instead of `range` in Python).
- *Difficulty*: How hard it is to fix the error. Complexity issues turn out to be the most difficult for students [28]. Also, fixing code according to industrial software metrics is often too hard for students [28] since these metrics usually are too abstract or too complex (*e.g., Lack of Cohesion of Methods* [3] in a class).
- *Importance*: How important it is to fix the error. Not all issues should be fixed immediately [21, 34]. For example, such software metric as *Number of Children of a Class* can be mostly ignored in student solutions.

The coefficient for each subcategory is calculated as the sum of all three criteria divided by the maximum of them. The exact values of these coefficients can be found in our supplementary materials [1].

To calculate the final penalty coefficient, we count the number of issues for each subcategory, multiply it by the coefficient for this subcategory, sum these products for all subcategories and normalize the sum. As a result, the final penalty coefficient ranges from 0 to 1.

The final code quality grade for each subcategory is reduced by one, two, or three levels. If this coefficient is in the range of $[0; 0.5)$ then the final grade is not reduced. Next, for every 0.2 points, the grade is decreased by one level. The grade reduction factors are presented in Table 2. These thresholds were selected by three experts with programming experience of more than four years. Two of them also have teaching experience of more than three years. The main goal of the thresholds is not to decrease the grade if the number of recurring errors is low or these errors are too difficult for students.

Let us consider an example. Let the initial grade (without a penalty) be *GOOD* and let the penalty coefficient be 0.6. According to Table 2, the final grade for the student will be decreased by one level and become *MODERATE*, since $0.6 \in [0.5; 0.7)$.

Final grade Initial grade	EXCELLENT	GOOD	MODERATE	BAD
EXCELLENT	$[0, 0.5)$	$[0.5, 0.7)$	$[0.7, 0.9)$	$[0.9, 1]$
GOOD	—	$[0, 0.5)$	$[0.5, 0.7)$	$[0.7, 1]$
MODERATE	—	—	$[0, 0.5)$	$[0.5, 1]$
BAD	—	—	—	$[0, 1]$

Table 2: Influence of the penalty coefficient on the final grade.

3.4 Difficulty Levels of Errors

We introduced several difficulty levels of code quality issues. This way, learners can choose the level appropriate for their current skill set and thus will not be demotivated by requirements that do not match their level.

We use three levels that correlate to how hard it is to fix a particular code quality issue based on the criteria of prevalence, difficulty, and importance described in Section 3.3:

- *Easy*. This group mainly consists of formatting issues from the *Code style* category. These problems are prevalent and can be fixed easily [19, 24].
- *Medium*. Issues from this group are related to *Best practices* applied by more experienced programmers, *e.g.,* using more efficient built-in functions of the programming language.
- *Hard*. This level contains *Error-proneness* and *Code complexity* issues. Fixing them requires a certain level of experience and knowledge.

As future work, we plan to come up with an algorithm for determining the difficulty level of a task automatically.

3.5 Providing Feedback

Hyperstyle provides descriptions for all found errors. Default output messages of professional code analyzers are often too brief, whereas students may need more thorough explanations [29]. We came up with custom messages for some of the issues, mainly for the most difficult categories, *e.g., Code complexity* or *Error-proneness*. On the other hand, we preserve standard messages for easy-to-fix issues, which mostly belong to *Code style* and *Best practices* categories.

Code quality issue	***Tutor***	***Hyperstyle***
`For` loop can be `forEach`	8	9
Simplify boolean expression	1	1
Simplify boolean return	1	1
Short algebraic operation	4	-
Switch `If Else` branches	1	-
Replace `For` to `While`	1	-
Parameter assignment check	-	2
Total	**16**	**13**

Table 3: Number of code quality issues detected by *Tutor* and *Hyperstyle*. Formatting issues detected by *Hyperstyle* are omitted since *Tutor* is not able to find them.

4 EVALUATION

We evaluated the tool in two different scenarios. Firstly, we compared our tool with previous work on a dataset of selected student solutions. Even though there are several papers introducing similar tools, only one, *Tutor* [30], was openly available for comparison. Next, we evaluated the usefulness of our tool in a setting of a programming course within several MOOC platforms: Stepik [16] and JetBrains Academy [8]. For that, we compared the median number of code quality issues per student before the tool was integrated into the platforms and after that.

4.1 Comparison with Similar Tools

Several similar tools have been introduced in prior work [20, 22, 30, 38]. However, the only tool that is publicly available to run is a demo version [17] for *Tutor* [30], which we used for the comparison with *Hyperstyle*. No other versions of *Tutor*, open or proprietary, are available except for this demo. In it, *Tutor* can assess code quality only for six built-in tasks written in Java. These tasks mostly target beginners and require to implement the body of a given function. Since we cannot extend *Tutor*, we had to perform the comparison on these six tasks. We asked 17 people to solve all of them. The programming background of the participants ranged from no experience to several years of using Java in industry. We included experienced programmers in this study, because sometimes they also attend computer science MOOCs [33] and code quality assessment tools should suit all types of students. In total, for the six tasks, we collected 107 snippets of code. We make this dataset publicly available in our supplementing materials [1].

We ran *Hyperstyle* and *Tutor* on each code fragment and compared their output. From our dataset, *Tutor* failed to grade 56 fragments (about 52% of the solutions), raising 20 unique types of errors. Almost all of the errors were about unsupported language elements, such as conditional operators or initialization of arrays. *Hyperstyle* managed to grade all the provided code fragments.

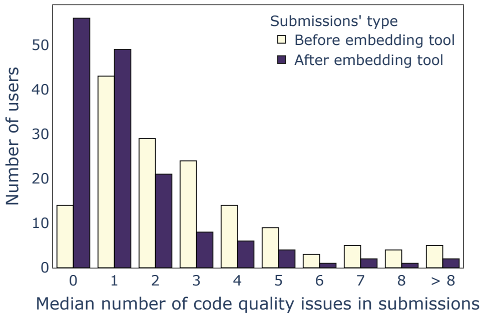

Figure 2: Influence of *Hyperstyle* on students' code style for Python solutions

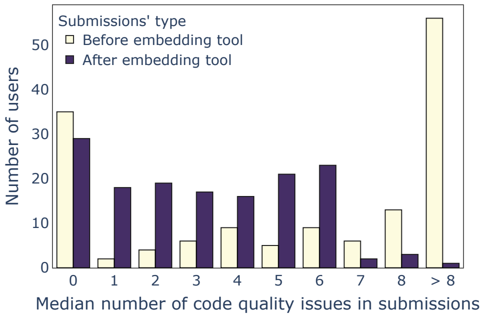

Figure 3: Influence of *Hyperstyle* on students' code style for Java solutions

Table 3 provides the comparison of the code quality issues found by both tools in the remaining part of the dataset: the 51 fragments (48%) that *Tutor* processed successfully. Both tools found the same three unique issues (ten cases in total by *Tutor* and 11 in total by *Hyperstyle*). In addition, *Tutor* found three unique issues (six cases in total) that were not found by *Hyperstyle*. *Hyperstyle* managed to find one additional unique issue (two cases in total) that was not found by *Tutor*. In total, *Tutor* found 16 code quality issues and *Hyperstyle* found 13. On the remaining 56 solutions that *Tutor* was unable to process, *Hyperstyle* found five more code quality issues (five cases in total). On top of that, on the whole dataset *Hyperstyle* found 54 fragments (about 50% of the dataset) with 201 formatting issues in total, which appears to be by far the most prevalent type of mistakes found in this comparison.

4.2 Influence on Students' Code Quality

To measure the usefulness of *Hyperstyle* for students, we compared the dynamics of code quality issues before integrating this tool in the education platforms and after that. Checks by *Hyperstyle* have been run on every submission, so students saw the reports and became aware of their errors even if they did not take action.

We gathered datasets with Python and Java solutions from the Stepik and JetBrains Academy platforms, selecting submissions by students who had at least ten solutions either in the period before *Hyperstyle* was introduced, or after. We selected only such submissions that the students agreed to make publicly available and anonymized all the data. For Python, we gathered submissions of 300 students. The first part of the dataset collected before introducing *Hyperstyle* contains 9,843 submissions from 150 students, the second one that was collected after introducing the tool has 14,407 submissions from 150 students. For Java, we gathered submissions from 294 students. There are significantly fewer publicly available solutions for Java than Python on the Stepik and JetBrains Academy platforms, so we gathered 2,000 submissions from 145 students before *Hyperstyle* was integrated and 3,192 submissions from 149 students after that. The dataset is available in our supplementary materials [1].

We ran the tool on all fragments for each student and counted the number of the reported code quality issues. For each student, we calculated the median number of the issues in their submissions.

Figure 2 shows how many Python students had a median of 0, 1, etc. code quality issues before and after *Hyperstyle* was introduced. These results can be interpreted as circumstantial evidence that the tool contributed to improving the students' code quality on these platforms: the number of students who had no code quality issues increased four times, and the number of students who made two or more errors decreased.

Figure 3 presents results for Java solutions. The number of students who made fewer mistakes (one to six) increased, and the number of those who made more than six mistakes, decreased. The results are especially notable for those who made more than eight errors: their number dropped dramatically from 56 to just one.

5 CONCLUSION AND FUTURE WORK

In this paper, we introduce Hyperstyle—a tool that provides detailed feedback on the code quality of programming solutions, which output can be easily integrated into MOOC platforms. The tool currently targets Python and Java, but can also work with Kotlin and JavaScript. The tool detects 600 code quality issues for Python and 222 issues for Java divided into five categories: *Code style*, *Code complexity*, *Error-proneness*, *Best practices*, and *Minor issues*. The tool can also detect recurring issues and thus help students to find issues that repeat over time. The first version of the tool was launched on the Stepik and JetBrains Academy platforms in January 2020. The current version of *Hyperstyle* has been running there since May 2021, handling roughly one million submissions every week.

We got very positive feedback from the creators of the Stepik and JetBrains Academy platforms about the quality of the hints, the tool's performance and flexibility. Since the tool adapted professional linters, students can switch from the online code editors to IDEs more easily since the sets of inspections are very similar. The most difficult tips were changed into more detailed ones that help students to work with hard code quality issues.

Future work on *Hyperstyle* involves categorizing code quality issues not only by their semantics and difficulty levels but also priority. Apart from that, we are planning to focus on Kotlin and JavaScript languages, since the tool currently supports mostly Python and Java. Finally, we are going to develop an algorithm to classify tasks by difficulty so that inspections irrelevant to a task can be disabled automatically.

REFERENCES

[1] 2021. *Artifacts and supplementary material.* https://doi.org/10.5281/zenodo.5749825

[2] 2021. *Checkstyle.* https://checkstyle.sourceforge.io/

[3] 2021. *Class cohesion measuring tool for Python.* https://github.com/mschwager/cohesion

[4] 2021. *Codacy.* https://www.codacy.com/

[5] 2021. *flake8.* https://flake8.pycqa.org/en/latest/

[6] 2021. *Hyperstyle docker image.* https://hub.docker.com/r/stepik/hyperstyle

[7] 2021. *Hyperstyle tool.* https://github.com/hyperskill/hyperstyle

[8] 2021. *JetBrains Academy.* https://www.jetbrains.com/academy/

[9] 2021. *Maintainability Index.* https://radon.readthedocs.io/en/latest/intro.html#maintainability-index

[10] 2021. *Oracle Java code conventions.* https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

[11] 2021. *PEP 8 – Style Guide for Python Code.* https://www.python.org/dev/peps/pep-0008/

[12] 2021. *PMD.* https://pmd.github.io/

[13] 2021. *Pylint.* https://www.pylint.org/

[14] 2021. *Qodana.* https://www.jetbrains.com/help/qodana/getting-started.html

[15] 2021. *SonarQube.* https://www.sonarqube.org/

[16] 2021. *Stepik.* https://stepik.org/

[17] 2021. *Tutor demo version.* https://www.hkeuning.nl/rpt/

[18] Korhan Akcura, Reza Shalchian, Abhijit Patil, Rattandeep Singh, and Jay Tanna. [n.d.]. Static Versus Dynamic Source Code Analysis. ([n. d.]).

[19] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education.* 522–527.

[20] Hannah Blau and J Eliot B Moss. 2015. FrenchPress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education.* 15–20.

[21] Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle Van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, and Bonnie MacKellar. 2018. " I know it when I see it" Perceptions of Code Quality: ITiCSE'17 Working Group Report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports.* 70–85.

[22] Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. 2016. Scale-driven automatic hint generation for coding style. In *International Conference on Intelligent Tutoring Systems.* Springer, 122–132.

[23] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[24] Ruvo Giuseppe, Tempero Ewan, Luxton-Reilly Andrew, Rowe Gerard, and Giacaman Nasser. 2018. Understanding semantic style by analysing student code. 73–82. https://doi.org/10.1145/3160489.3160500

[25] Robert L Glass. 2002. *Facts and Fallacies of Software Engineering.* Addison-Wesley Professional.

[26] Sudheendra Hangal and Monica S Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* IEEE, 291–301.

[27] Willy Jimenez, Amel Mammar, and Ana Cavalli. 2009. Software vulnerabilities, prevention and detection methods: A review1. *Security in model-driven architecture* 215995 (2009), 215995.

[28] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education.* 110–115.

[29] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education.* 119–125.

[30] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education.* 562–568.

[31] Kris MY Law, Victor CS Lee, and Yuen-Tak Yu. 2010. Learning motivation in e-learning facilitated computer programming courses. *Computers & Education* 55, 1 (2010), 218–228.

[32] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship.* Pearson Education.

[33] Heather Miller, Philipp Haller, Lukas Rytz, and Martin Odersky. 2014. Functional programming for all! Scaling a MOOC for students and professionals alike. In *Companion Proceedings of the 36th International Conference on Software Engineering.* 256–263.

[34] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fmri study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 524–536.

[35] Nick Rutar, Christian B Almazan, and Jeffrey S Foster. 2004. A comparison of bug finding tools for java. In *15th International symposium on software reliability engineering.* IEEE, 245–256.

[36] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Commun. ACM* 61, 4 (2018), 58–66.

[37] Liisa Sakerman. 2021. Overview of the advantages and disadvantages of static code analysis tools. (2021).

[38] Leo C Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education.* 738–744.