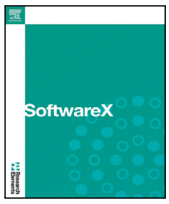
SoftwareX 29 (2025) 102048

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Code-Review-as-an-Educational-Service: A tool for Java code review in programming education

Matthew Beattie, Moira Watson, Desmond Greer, Bee-Yen Toh, Zheng Li *

School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, Belfast BT9 5AF, UK

ARTICLE INFO

Keywords:
Cloud-native architecture
Code quality
Code review
Programming education
Software as a service
Tooling support

ABSTRACT

High-quality source code is the foundation of successful and sustainable software development, while code review plays a crucial role in ensuring code quality. We place a special emphasis on the educational application of code review, aiming to assist novice students who are entry-level programmers establish industry-standard programming practices while reducing the likelihood of vulnerabilities and technical debt. Given that existing code review tools often require complex setups and are designed for large-scale, enterprise-level software projects, we advocate for the development of an easy-to-use, zero-configuration, and lightweight tool that is specifically tailored to the needs of educational environments. This paper reports our development of such a cloud-native code review tool as an educational service. Although still at the proof-of-concept stage, our internal and preliminary assessment has confirmed the promising usability and usefulness of this tool both for students (e.g., self-reviewing an individual exercise) and for educators (e.g., examining cohort exercises and prioritising teaching materials). By integrating this tool into our innovative project Automating Programming Education in Java, we believe that such an educational service would be able to make contributions to faster maturation of programming skills in students.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-24-00371
Permanent link to Reproducible Capsule	https://zenodo.org/doi/10.5281/zenodo.12515217
Legal Code License	BSD-3-Clause.
Code versioning system used	git
Software code languages, tools, and services used	Java, Spring Boot, React.js, MongoDB
Compilation requirements, operating environments & dependencies	https://github.com/MBeattie02/code_review_tool
If available Link to developer documentation/manual	https://github.com/MBeattie02/code_review_tool/tree/main/code_review_tool_documentation
Support email for questions	mbeattie13@qub.ac.uk, matthewbeattie22@gmail.com

1. Motivation and significance

In the fast-paced and detail-oriented field of software engineering, code quality is not just a technical concern but also a foundation for successful and sustainable software development. Poor quality code can be characterised by issues such as lack of readability, security vulnerabilities and non-adherence to coding standards, leading to a cascade of problems. These problems include inefficient code leading to increased security risks and regulatory non-compliance, significantly hampering software projects' immediate and long-term effectiveness as well as contributing to "technical debt" [1]. The term "technical debt" originates from Cunningham and describes a metaphorical representation of the extra development work that arises when easy-to-implement code is used in the short term instead of applying the best overall solution [2]. This debt accumulates interest over time, making future changes and enhancements more costly and time-consuming.

Code quality issues occur throughout the software development and maintenance life cycle, starting as early as the coding phase when developers inject smells and vulnerabilities into codebases. This can

* Corresponding author.
E-mail addresses: mbeattie13@qub.ac.uk (Matthew Beattie), m.watson@qub.ac.uk (Moira Watson), des.greer@qub.ac.uk (Desmond Greer), b.toh@qub.ac.uk (Bee-Yen Toh), zheng.li@qub.ac.uk (Zheng Li).

https://doi.org/10.1016/j.softx.2025.102048
Received 10 July 2024; Received in revised form 30 November 2024; Accepted 10 January 2025
Available online 22 January 2025

be caused by developers who may lack knowledge of correct software principles or are under pressure to deliver code quickly, and therefore sacrifice code quality for speed of development [3]. This can continue into code review, testing, and production, impacting QA testers, end-user testing, and project sign-off. The problem can persist and compound over time without proper tools and practices. If left unchecked, poor code quality can have a knock-on effect on the delivery of the software, stopping development and ultimately leading to more time refactoring and completely rewriting sections of code.

Interacting with poor-quality code can be particularly problematic for novice programmers, especially when initially learning how to program. Poorly developed codebases can in turn mislead beginners to adopting and applying substandard coding habits that they have experienced, believing them to be standard practice. Learning from poor-quality code examples can complicate understanding of new programming paradigms, as poor quality is often characterised by poor readability. Poor-quality code also hinders efficient debugging and problem-solving due to its unclear logic [4]. In contrast, high-quality code serves as an excellent learning resource and provides a solid foundation for continued development. Therefore, beginners should engage with well-structured, documented, and standard-adhering code from the start of their development career [5]. And we argue that code review plays a key role in such an engagement.

1.1. Related work

The absence of code review or manual code review can hinder timely issue detection, resulting in excessive work required to find and fix bugs or even making projects non-releasable due to poor quality, as evidenced by that around 60 percent of U.S. software engineering work time were wasted on finding and fixing avoidable errors [6]. Therefore, automated code review has become a major topic in software engineering and there has been an explosion in automated code review tools. Nevertheless, most of the tools are developed for large-scale software and enterprise-level usage, while they can be cumbersome and less effective to be applied to smaller exercise-level codebases or to projects using vanilla Java, which leaves a notable gap in the market for code review solutions suitable for the educational purpose.

For example, SonarQube, an open-source platform for continuously inspecting code quality, performs static code analysis to identify issues related to code quality, security, and coding standards.[1] Its key feature is the integration with projects that use Maven as their build system, via a `pom.xml` file within the project files. However, setting up build systems and configuring SonarQube integration would be too complex for novice students and actually out of the scope of programming education at the introductory level. Although its cloud version (*i.e.*, SonarCloud[2]) relieves the complexity by offering the same functionalities as services, its pricing model constrains its usage in education. Considering the increasing financial crisis in global higher education [7], it is barely possible to ask universities to purchase those cloud services for students year by year, not to mention that some costly and advanced functionalities are not needed in basic educational activities. On the other hand, SonarCloud's free-of-charge plan does not support teamwork and especially the communication between students and educators.

Another example is SpotBugs, a static analysis tool for identifying bugs in Java code. Similarly, it also requires complex and advanced project configurations. For instance, when using SpotBugs to perform optimal analyses, annotations like `@Nullable` need to be included in the codebase,[3] while such advanced concepts would cause cognitive overload for new learners working on basic Java programs and may negatively impact their learning of Java programming.

[1] https://docs.sonarsource.com/sonarqube/latest/
[2] https://www.sonarsource.com/products/sonarcloud/
[3] https://spotbugs.github.io/#using-spotbugs

1.2. This work

This work aims to provide a safety net for novice students by guiding them towards following industry-standard coding conventions and away from poor-quality code pitfalls. Functionally, the developed code review tool mainly leverages Abstract Syntax Tree (AST) analysis, so as to automatically conduct detailed examination of syntax and structure to identify code issues and suggest best practices that meet industry standards of coding. Non-functionally, compared to other existing solutions, this work is particularly focused on minimising configurations/operations and maximising the easiness to use the tool for beginners.

It is worth highlighting three features to help distinguish this work from the existing code review tools.

- This tool is designed in a cloud-native fashion and developed as a software-as-a-service solution. As such, there is little configuration work left for using this tool to perform code review tasks.
- This tool is seamlessly integrated with GitHub. In fact, GitHub is employed not only as the input for using this tool, but also as the venue to track students' learning progress (by automatically posting comments to the code repository).
- This tool is seamlessly integrated with Slack. This feature enables merging code review results automatically into the context of (optional) discussions between students and their educators.

1.3. Significance

The significance of this work is the tooling support against the shortage of teaching workforce on personal tutoring in software engineering and programming education.

It has been identified that the failure and dropout rates continue to be high in programming-relevant courses [8]. Despite the difficult nature of such subjects, a significant challenge would be the insufficient tutoring resources and a lack of personal instructions for novice students [9]. The current learning method faces significant challenges due to the increasing demands of large numbers of students from diverse backgrounds and at different levels. Firstly, there is a widely reported shortage of suitable teaching professionals, including professors, lecturers, and teaching assistants, which is exacerbated by the diverse needs of the student body. Furthermore, this shortage directly undermines student support, manifested in scenarios where students are required to wait for up to two hours during office hours to ask even a single question [10].

In contrast, by automatically reviewing source code and providing timely feedback, a suitable code review tool enables students' self-learning of programming and simultaneously maximises the efficiency of educators' guidance across an entire cohort. To validate the significance of this work, we have been internally assessing the usability and usefulness of this tool both for students and for educators. It is worth noting that usability and usefulness are two correlated properties of any user-oriented system for determining the system's satisfaction and usage [11]. Particularly, usability is a more fundamental property than usefulness, as a highly usable system can be functionally useless, while a useful system must have been effectively designed and usable [12].

Since this project is still under active development, we are currently targeting Technology Readiness Levels 3 and 4 (*i.e.*, TRL 3 and TRL 4) [13]. Our efforts are focused establishing a robust proof of concept and conducting "breadboard verification" in a laboratory setting. Correspondingly, we conducted lightweight and preliminary validation of this tool at this current stage, via internal trials and casual interviews with nine undergraduate students (including a four-student group) and three educators (including two lecturers and one learning tutor). Particularly, the student interviewees were contacted by the first author, and they were asked to follow the user manual document

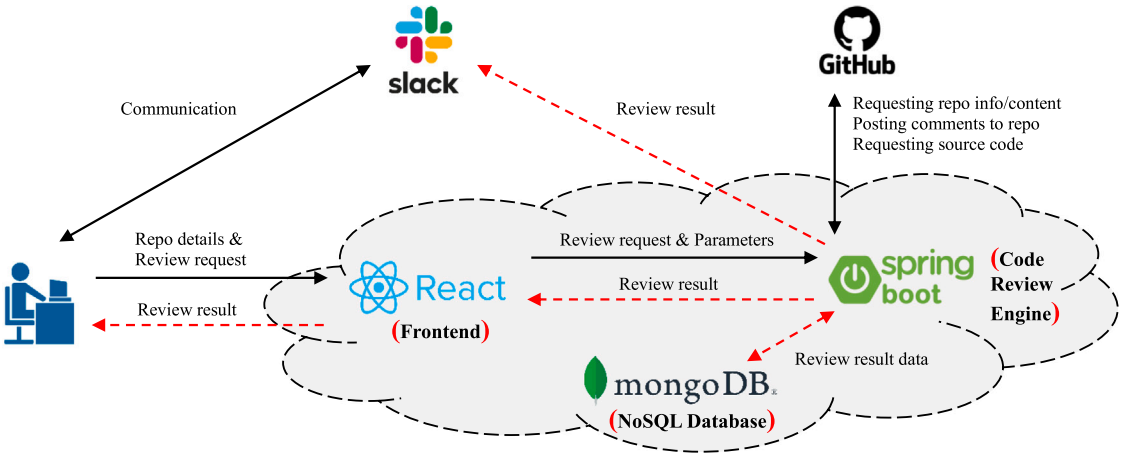


Fig. 1. The cloud-native architecture of this code review tool. (The red and dashed line arrows indicate the data flow of code review results, while the black line arrows indicate the non-result data flows, *e.g.*, review requests or user communication.).

to try this tool and give feedback during the lab sessions; the two lecturers listened to the first author's demo presentation and shared their thoughts about this tool; while the learning tutor directly joined the student group trial in a lab session and then attended the group interview together. The interview questions vary during our casual conversations, while we have generally covered three ones as follows.

- Do you think this tool is easy to use? (How easy do you think this tool is to use?)
- Do you think this tool is useful? (How useful do you think this tool is?)
- What do you think about this tool? (Do you have any suggestion about this tool?)

Interestingly, the responses from student interviewees and educator interviewees show two different patterns. Firstly, all the student interviewees confirmed that this tool was useful for their learning and self-learning of programming. However, some students were concerned about the usability of this tool, *e.g.*, the non-integration into IDE and the learning curve of using GitHub, especially for beginners. Although the integration into IDE is out of the scope of this tool, we have considered improving this tool's usability by allowing users to directly upload source code files. Secondly, all the educator interviewees appreciated the ease to use of this tool. However, it was suggested that the educational usefulness of this tool could be further improved by removing some non-critical features, *i.e.*, removing the identification of security risks. Since security knowledge is barely included in the fundamental/basic programming courses, such a feature may mislead novice students or cause cognitive overload when using this tool. We have taken into account this suggestion and planned particularly to (re-)evaluate this feature in our future and wider-scope validation studies.

2. Software description

2.1. Software architecture

To improve the cross-platform compatibility and maintainability of our tool, we have employed Docker containers to package different software components. Recall that "cloud-native computing relies on containers as a common denominator" [14]. The employed containerisation essentially leads to a cloud-native architecture for the whole system, as shown in Fig. 1. In other words, this tool can be conveniently deployed in both private cloud and public cloud, to be offered as an online educational service. Particularly, if deploying this tool in public cloud, the component NoSQL Database can be directly outsourced to a fully-managed cloud database service (*e.g.*, MongoDB Atlas on Azure[4]), to minimise the development and deployment efforts [15].

We explain the key components including brief implementation details in the following subsections.

2.1.1. Frontend

The frontend is responsible for the interactions between the code review engine and the user. The logic behind user interactions are all implemented as API calls. By taking into account the principles of user experience and user interface design, the frontend implementation has tried to avoid unnecessary elements and minimise user operations to improve the tooling efficiency.

The frontend implementation is mainly with `React.js`, a JavaScript library for building dynamic user interfaces following the component-based architecture. React is chosen as it simplifies the development process by allowing the creation of single pages with individual functionality that can be linked together. Each page updates and renders components in response to data changes during interaction.

2.1.2. Code review engine

The code review engine is responsible for performing AST analysis on the raw Java code retrieved from GitHub. The AST analysis is technically enabled via the Java Parser library. In brief, the Java Parser breaks down the Java Source Code into an AST tree representation to capture the hierarchical structure of the code's syntax. Then, additional algorithms within the code review engine can inspect this structure to identify coding issues and offer improvement suggestions.

The implementation of the code review engine uses Java 17 and employs Maven for dependency management and deployment. Java 17 is chosen as it is the most recent long-term support version of Java and is the required version for Spring Boot 3; while Spring Boot 3 is chosen as the Java framework, because its wide range of boilerplate code can reduce the time required to set up the environment. In addition, the use of annotations available in Spring Boot reduces the size of the codebase.

2.1.3. NoSQL database

Once the code review is completed, the results will be stored, ready for retrieval for further usage. We have chosen NoSQL database as the data storage solution, because it enables flexible data model to cater unstructured data and unforeseen data types, which conveniently addresses the requirement changes along the development of this tool. For example, after we decided to supplement the code review results with the repository information, we could quickly implement the relevant

[4] https://azure.microsoft.com/en-us/solutions/mongodb

new features with little code refactoring.

To ensure efficient and persistent NoSQL data storage, this database component employs MongoDB and the Spring Data MongoDB dependency that is structured around RESTful controllers. These controllers serve as the data access points for HTTP requests, facilitating the retrieval and writing operations from/to the database.

2.1.4. GitHub integration

By taking advantage of GitHub's RESTful APIs,[5] our tool seamlessly integrates with GitHub. This integration allows our tool to interact with various GitHub resources such as repositories, commits, issues, pull requests, user profiles, and raw code files. It is worth noting that we intentionally require codebase to be reviewed online (*i.e.*, via accessing GitHub repository) instead of being reviewed locally. This not only makes our whole solution purely cloud-native but also maximises the usability and usefulness of this tool, *e.g.*, in addition to student users, educators can collect multiple students' codebase venues and use the tool to conveniently examine students' code quality all together.

2.1.5. Slack integration

Similarly, we take advantage of Slack Web APIs[6] to seamlessly integrate Slack with our tool. This integration enables code review results to be optionally posted to predefined Slack channels, to facilitate group discussions among students or between students and their educator. To support Slack integration, technically, pre-processing is needed to compose JSON-format results of code review into Slack-compatible messages, and then the messages can be pushed to Slack channels via Slack API with a webhook POST request.

2.2. Software functionalities

As reflected by our architectural design (see Section 2.1), we have taken into account a wide range of functional features (*e.g.*, posting code review results as comments to GitHub and as messages to Slack) and non-functional features (*e.g.*, employing cloud-native tech stack to improve the tool's usability and portability). To save space, we only highlight the major functionalities related to code review, without exhaustively listing all the features of this tool.

- Calculating cyclomatic complexity of codebase.
- Identifying coding style issues.
  - ⋆ Checking class and interface naming conventions.
  - ⋆ Checking variable naming convention.
  - ⋆ Checking usage of Magic numbers.
  - ⋆ Checking indentation consistency.
  - ⋆ Checking brace style.
  - ⋆ Checking the import organisation.
- Identifying code smells.
  - ⋆ Checking existence of excessive parameters.
  - ⋆ Checking existence of methods with excessive length.
  - ⋆ Checking existence of God class.
  - ⋆ Checking existence of large classes.
  - ⋆ Checking correctness of using try block.
  - ⋆ Checking existence of data clumps.
  - ⋆ Checking existence of methods with primitive obsession.
  - ⋆ Checking correctness of using comments.
  - ⋆ Checking existence of dead methods.
  - ⋆ Checking existence of method chaining.
  - ⋆ Checking exception handling.
- Identifying security risks.
  - ⋆ Checking existence of deprecated APIs.
  - ⋆ Checking existence of SQL injection.
  - ⋆ Checking existence of cross-site scripting.
  - ⋆ Checking existence of insecure deserialization.
  - ⋆ Checking existence of hard-coded credentials.
  - ⋆ Checking existence of race condition.
  - ⋆ Checking existence of insecure crypto practices.
  - ⋆ Checking existence of high entropy strings.

It should be noted that the current version of this code review tool aims at typical and known coding issues selected from our students' exercises. We will keep enriching the functionalities to improve the generality of this tool.

3. Illustrative examples

We use two illustrative examples to demonstrate representative functions of this code review tool. The first one is about reviewing a single code file and delivering code review result, while the second one is about summarising historical code review results and offering learning materials for addressing the identified coding issues, as described in the following subsections.

3.1. Reviewing a single code file

This example represents the most common use case in which a user examines the code quality and style within a single `.java` file. By (optionally) using GitHub credentials to connect to a GitHub account, the user can locate a particular codebase repository, select a `.java` file within the repository, and choose a review type among `coding style`, `code smell`, and `security risks` to conduct code review for the selected `.java` file. The step-by-step operations have been documented in the shared user manual.

The coding style review result of an example code file is demonstrated in Fig. 2, including both metadata (*i.e.*, the repository information and the review report ID) and a list of identified coding issues (*i.e.*, the violations against predefined coding rules). To facilitate users quickly observing their coding issues within the source code context, the review result page enables a floating window to exhibit the content of the reviewed file (by clicking the "View Raw Code" button), as well as highlighting the lines where coding issues exist. The effect of such a floating window can be seen in Fig. 3.

3.2. Summary and statistics of historical code review results

This is another typical use case of this tool for the educational purpose, which summarises the existing code review results (either all or from a particular time point) and delivers frequency statistics of the identified coding issues. As a demonstration, Fig. 4 shows the summary and statistical results from all the code review trials/tests when developing this tool. According to the frequency ranking of different coding style issues, for example, incorrect indentation is clearly the most common one. Then, by clicking "incorrect indentation" on the ranking list, a pop-up window will appear showing a more detailed description about, and a suggested solution to, this issue. In addition, a minimal pseudocode/code example will be given, to help users intuitively understand what correct coding style is (indentation in this case).

It is worth mentioning that this functionality is applicable in multiple scenarios and beneficial for different types of users. For example, a student can use it to track study progress and learning effects by periodically observing his/her own code review summaries; while by statistically observing the overall code review results of a whole cohort, an educator can accordingly prioritise teaching materials and improve their teaching efficiency.

[5] https://docs.github.com/en/rest/quickstart

[6] https://api.slack.com/web

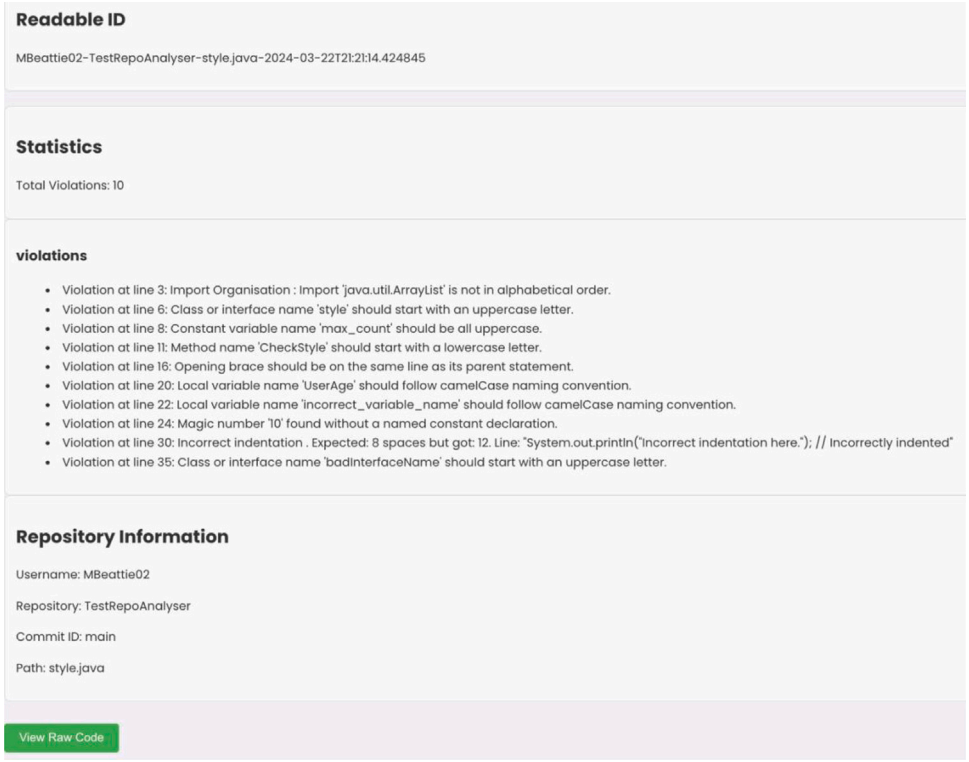


Fig. 2. The coding style review result of an example code file.

4. Expected and potential impact

This work is an integral part of "Automating Programming Education in Java", an ambitious initiative designed to revolutionise programming education through innovation. Our work focuses on developing a suite of automated educational tools, building on the successes of our previous work on a coding-oriented tool which facilitates just-in-time learning of programming [10].

Given the overall positive feedback from the initial trials of our early-version tools, we have gained more confidence in the whole "Automating Programming Education in Java" project's potential to make significant contributions to the education sector. Particularly, we believe that our code review tool would (potentially) play a crucial role in facilitating self-directed learning of programming and enhancing the assessment of learning outcomes, as well as improving teaching efficiency. For a comprehensive analysis of the benefits associated with this code review tool, please refer to Sections 1.3 and 3.2 of this document, where they are detailed extensively.

5. Conclusions

Automated code review is one of the hottest topics of software engineering in both academia and industry. To address production needs, the existing code review tools are normally required to be directly integrated into software projects and tightly coupled with codebase within the development environment. From the educational perspective, however, this can be daunting for novice programmers as they might not yet be familiar with advanced project configurations. Driven by this educational gap, we have developed an education-oriented and lightweight code review tool as part of our project Automating Programming Education in Java.

In conclusion, this work is distinguished from the existing code review tools in its education-friendly features. Particularly, it is worth highlighting that the cloud-native architecture makes this tool easy to access and easy to use without any configuration; the statistical summary of historical code review results (in different scenarios) enables students to track their learning effects and allows educators to prioritise teaching materials; while the informative explanations about the identified coding issues offer extra learning opportunities to students not only to correct mistakes but also to deepen understanding of best practices.

Our ongoing validation work employs empirical research methods (mainly casual interviews) to collect users' feedback to verify the usability and usefulness of this tool. In the meantime, we have been using the collected feedback to keep enriching this tool's functionalities. As such, the current validation of this tool is rather ad hoc than systematic and rigorous at this stage. Therefore, our future work will be unfolded towards two milestones: (1) We plan to deliver an updated and relatively stable version of this tool based on its trial usage in the upcoming academic year. (2) After that, we intend to conduct systematic and formal validation studies of this tool and to report the whole research including this tool to the educational community.

CRediT authorship contribution statement

Matthew Beattie: Writing – original draft, Visualization, Software, Investigation, Conceptualization. **Moira Watson:** Writing – review & editing, Supervision, Resources. **Desmond Greer:** Writing – review & editing, Validation, Conceptualization. **Bee-Yen Toh:** Writing – review & editing, Validation, Conceptualization. **Zheng Li:** Writing – original draft, Supervision, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

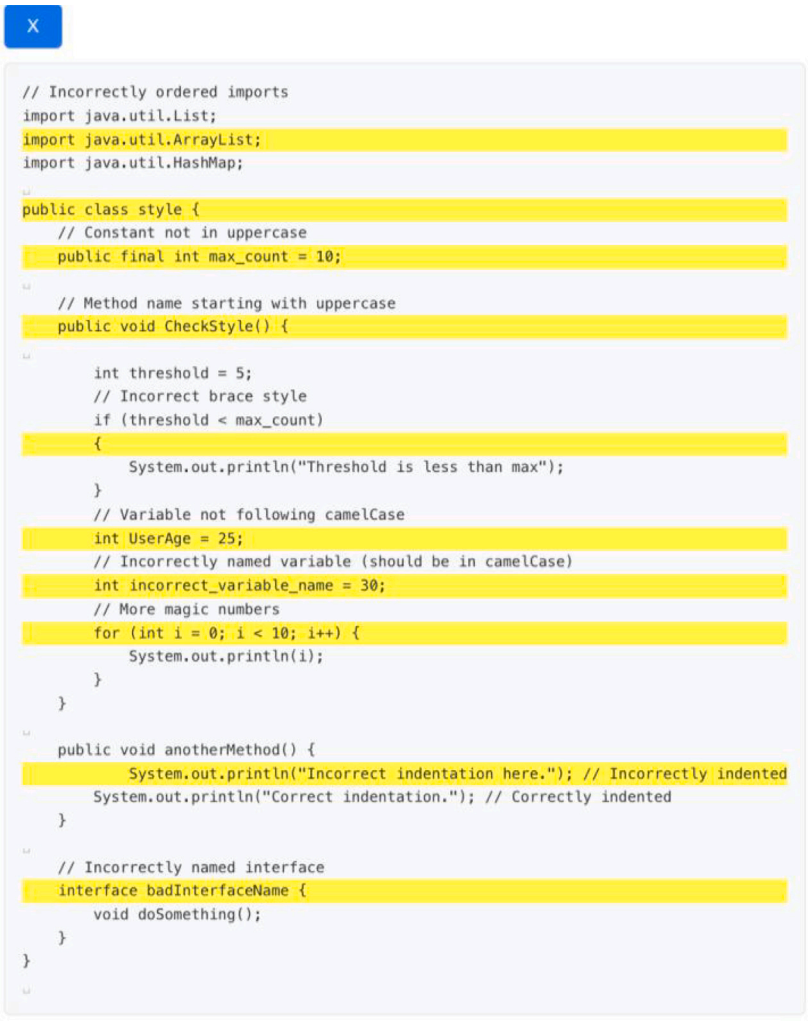


Fig. 3. The floating window of reviewed source code with highlighted lines where coding issues exist.

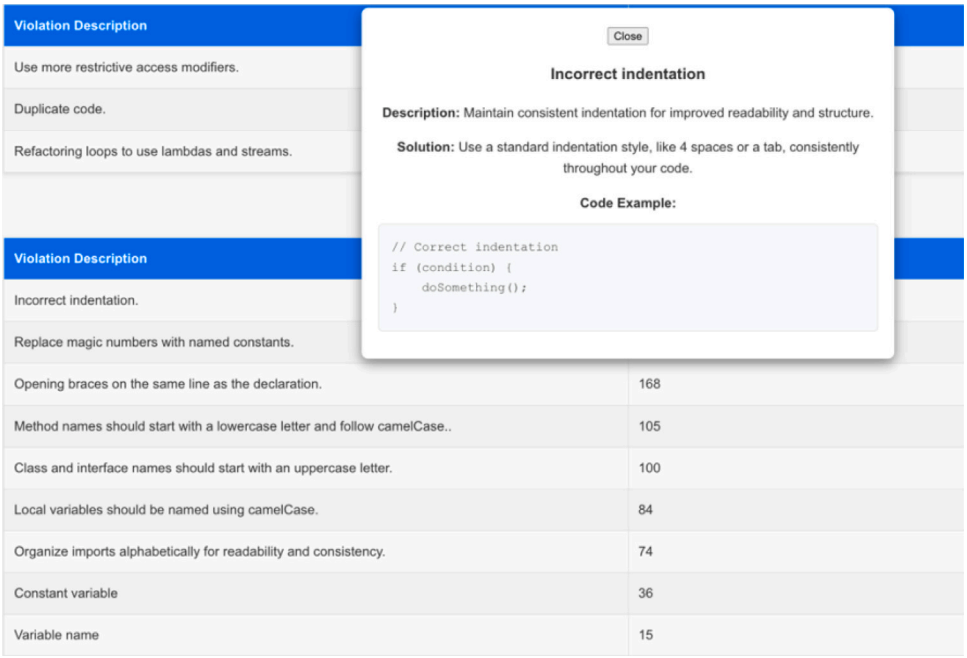


Fig. 4. Summary and statistics of historical code review results, with detailed explanations in pop-up windows.

References

[1] Blueoptima. How poor code quality can grind development to a halt: A deep dive. 2023, https://www.blueoptima.com/how-poor-code-quality-can-grind-development-to-a-halt-a-deep-dive/.

[2] Alves NS, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C. Identification and management of technical debt: A systematic mapping study. Inf Softw Technol 2016;70:100–21. http://dx.doi.org/10.1016/j.infsof.2015.10.008.

[3] Tsipenyuk K, Chess B, McGraw G. Seven pernicious kingdoms: A taxonomy of software security errors. IEEE Secur Priv 2005;3(6):81–4. http://dx.doi.org/10.1109/MSP.2005.159.

[4] Hermans F, Aivaloglou E. Do code smells hamper novice programming? A controlled experiment on scratch programs. In: Proceedings of the 24th IEEE international conference on program comprehension. Austin, Texas, USA: IEEE Computer Society; 2016, p. 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503706.

[5] Gutmann V, Starke E, Michaeli T. Investigating code smells in K-12 students' programming projects: Impact on comprehensibility and modifiability. In: Local proceedings of the 16th international conference on informatics in schools. Lausanne, Switzerland: Zenodo; 2023, p. 49–60. http://dx.doi.org/10.5281/zenodo.8431914.

[6] Jones C. Wastage: The impact of poor quality on software economics. Softw Qual Prof 2015;18(1):23–32, https://asq.org/quality-resources/articles/wastage-the-impact-of-poor-quality-on-software-economics?id=35a5bf7c776b407cb9d41d25b7f0650c.

[7] Jacob WJ, Gokbel V. Global higher education learning outcomes and financial trends: Comparative and innovative approaches. Int J Educ Dev 2018;58(1):5–17. http://dx.doi.org/10.1016/j.ijedudev.2017.03.001.

[8] Medeiros RP, Ramalho GL, Falcão TP. A systematic literature review on teaching and learning introductory programming in higher education. IEEE Trans Educ 2019;62(2):77–90. http://dx.doi.org/10.1109/TE.2018.2864133.

[9] Lahtinen E, Ala-Mutka K, Järvinen H-M. A study of the difficulties of novice programmers. ACM SIGCSE Bull 2005;37(3):14–8. http://dx.doi.org/10.1145/1151954.1067453.

[10] Li Z, Gorrepati SS, Greer D. On code example-aided just-in-time learning for programming education. In: Proceedings of the 30th Asia-Pacific software engineering conference. Seoul, Republic of Korea: IEEE Computer Society; 2023, p. 622–6. http://dx.doi.org/10.1109/APSEC60848.2023.00083.

[11] Tsakonas G, Papatheodorou C. Analysing and evaluating usefulness and usability in electronic information services. J Inf Sci 2006;32(5):400–19. http://dx.doi.org/10.1177/0165551506065934.

[12] Greenberg S, Buxton B. Usability evaluation considered harmful (some of the time). In: Proceedings of the SIGCHI conference on human factors in computing systems. Florence, Italy: ACM Press; 2008, p. 111–20. http://dx.doi.org/10.1145/1357054.1357074.

[13] Bruno I, Lobo G, Covino BV, Donarelli A, Marchetti V, Panni AS, et al. Technology readiness revisited: A proposal for extending the scope of impact assessment of European public services. In: Proceedings of the 13th international conference on theory and practice of electronic governance. Athens, Greece: ACM Press; 2020, p. 369–80. http://dx.doi.org/10.1145/3428502.3428552.

[14] The New Stack. An overview of containerization technologies. 2024, https://thenewstack.io/containers/.

[15] Li Z. Long live the image: Container-native data persistence in production. In: Proceedings of the IEEE 18th international conference on software architecture companion. Stuttgart, Germany: IEEE Computer Society; 2021, p. 82–5. http://dx.doi.org/10.1109/ICSA-C52384.2021.00020.