DEVELOPMENT OF AN INTELLIGENT CODE REVIEWER RECOMMENDATIONS SYSTEM FOR PROGRAMMING CONVENTIONS

BY

EKEKWE TANYALOUISE
(30XX123456)

A PROJECT SUBMITTED TO THE DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES, COLLEGE OF SCIENCE AND TECHNOLOGY, COVENANT UNIVERSITY OTA, OGUN STATE.

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF THE BACHELOR OF SCIENCE (HONOURS) DEGREE IN COMPUTER SCIENCE.

APRIL 2024

Certification

I hereby certify that this project was carried out by Ekekwe Tanyalouise (30XX123456) in the Department of Computer and Information Sciences, College of Science and Technology, Covenant University, Ogun State, Nigeria, under my supervision.

Dr. Supervisor _____

Supervisor **Signature and Date**

Professor Olufunke O. Oladipupo _____

Head of Department **Signature and Date**

Dedication

This page has been intentionally left blank.

Acknowledgements

This page has been intentionally left blank.

Table of Contents

Certification ..... i.
Dedication ..... ii.
Acknowledgements ..... iii.
Abstract ..... vii.
1. CHAPTER ONE: INTRODUCTION 1
1.1. BACKGROUND INFORMATION 1
1.2. STATEMENT OF THE PROBLEM 2
1.3. AIM AND OBJECTIVES 3
1.4. METHODOLOGY 3
1.5. SIGNIFICANCE OF THE STUDY 6
1.6. SCOPE OF THE STUDY 6
1.7. LIMITATIONS OF THE STUDY 7
1.8. STRUCTURE OF THE RESEARCH 7
2. CHAPTER TWO: LITERATURE REVIEW 8
2.1. PREAMBLE 8
2.2. REVIEW OF PROGRAMMING CONVENTIONS 8
2.2.1. BRIEF HISTORY OF PROGRAMMING CONVENTIONS 8
2.2.2. MAIN TYPES OF PROGRAMMING CONVENTIONS 9
2.3. REVIEW ON CODING SMELLS AND CODE QUALITY 10
2.4. REVIEW ON CODE QUALITY MANAGEMENT 11
2.4.1. MODERN CODE REVIEW: A STREAMLINED APPROACH 11
2.4.2. INTELLIGENT CODE REVIEW TOOLS 13
2.4.3. STATIC ANALYSIS 13
2.5. REVIEW ON THE EFFECTS OF UNCLEAN CODE 14
2.5.1. TECHINICAL DEBT 14
2.6. REVIEW OF RELEVANT CONCEPTS 14
2.7. REVIEW OF RELATED METHODS 14
2.8. REVIEW OF EXISTING SYSTEMS 14
References 15

List of Tables

Table 1.1: Objectives-Methodology Mapping Table 5

List of Figures

Figure 2.1: Overview of steps in modern code reviews, adapted from (Badampudi et al., 2023) . . 12

Abstract

This page has been intentionally left blank.

Chapter 1.
CHAPTER ONE: INTRODUCTION

1.1. BACKGROUND INFORMATION

The evolution of coding standards and best practices has deep roots, dating back to the early days of programming (Dijkstra et al., 1972). As software development methodologies and technologies have advanced, so too have coding standards. This co-evolution is particularly evident when examining the shift from structured programming in the 1960s to object-oriented programming in the 1980s (Pratap, 2023). Early coding standards emerged alongside structured programming to promote modular, readable code and followed a clear control flow; these characteristics became essential as software projects grew in complexity (Medoff, 2015). Structured programming approaches like topdown design and control flow statements (if statements, loops) rely on well formatted code to be easily understandable and maintainable. Object-oriented design principles like encapsulation and code reusability are best achieved through consistent naming conventions and code organization.

Among software engineering techniques, programming conventions play a critical role in promoting the readability of source code (InstituteData, 2023). These conventions, which can encompass aspects like indentation, naming conventions, and code formatting, act as a shared language for developers, allowing them to quickly grasp the purpose and structure of the code, even if written by someone else (Chen et al., 2018). According to Kourie & Pieterse (2008), there is a need to formally educate students to become professional, responsible, and reliable developers capable of producing quality code. However, Wiese et al. (2017) identifies a gap in the existing literature directed at teaching programming conventions during the initial stages of programming education. A concerning find, as poorly formatted and inconsistently named code becomes a major hurdle as projects grow in size and complexity, hindering both understanding and maintenance. If early adoption of programming conventions is prioritized, beginner programmers can lay a solid foundation of clean, maintainable, and collaborative code throughout their software development journey (Keuning et al., 2021).

One potential solution to bridge this gap and encourage early adoption of coding conventions is through intelligent code review tools (Kim et al., 2022). These tools are already widely used in professional software development to automate tasks like identifying syntax errors and potential bugs (Sadowski et al., 2018). In the context of learning conventions, these tools can be leveraged as a valuable learning resource for beginners. According to Bancroft & Roe (2006), feedback or recommendations are essential in the learning process, especially when they are available on request. Imagine a code reviewing tool that not only highlights potential errors but also flags sections that deviate from established coding standards. This real time feedback can raise beginners' awareness of the importance of proper formatting, naming conventions, and code organization. By integrating educational resources and suggestions directly into the development workflow, intelligent code reviewing tools

can become powerful allies for beginner programmers, helping them write clean, maintainable, and well structured code from the very beginning.

While established code reviewing platforms like those offered by GitHub and Codacy are valuable tools in the programmer's arsenal, they have a blind spot, coding conventions. These platforms primarily focus on identifying functional errors and bugs, ensuring the code works as intended (Badampudi, Unterkalmsteiner, & Britto, 2023). However, they often fall short in addressing whether the code adheres to established coding conventions. Some might offer essential error messages related to "code smells" or stylistic inconsistencies, but lack the depth and interactivity needed to be a true learning resource for beginners who need to understand the "why" behind coding conventions (Wessel et al., 2020). This gap in feedback leaves aspiring programmers vulnerable to developing bad habits and writing code that becomes a tangled mess down the line.

Intelligent code reviewing tools, designed specifically for beginners, can address this limitation by transforming code review into a powerful learning resource. These tools leverage AI and established coding standards to go beyond simply highlighting missed conventions (Kim et. al., 2022) This study proposes a realtime mentor who does not just point out formatting issues (a language linter) but also suggests best practices and alternative approaches based on the specific code and its intended function. The shift towards a more interactive and personalized learning experience could significantly improve the quality, maintainability, and understanding of code written by beginners. Ultimately, this will lead to a smoother transition into professional software development by equipping them with the necessary skills to write clean, maintainable, and collaborative code from the outset.

1.2. STATEMENT OF THE PROBLEM

The initial stages of learning to program are often focused on core functionalities and building working programs, coding standards often get sidelined. Understandably, new programmers are eager to understand the basic functions of languages and to develop programs that work, often neglecting to apply coding conventions during their development workflow. According to research conducted by Joni & Soloway (1986), 90% of students used the wrong coding standards in programming exercises and Ruvo et al. (2018) finds that these bad habits persist up till into students' 4th year of a BS degree. The tendency to skip learning these established practices can create significant problems for beginners down the line. By neglecting to adhere to coding standards and conventions, beginners risk developing poor coding habits that can impact the readability, maintainability, and scalability of their code (Popić et al., 2018).

The effects of poor code quality don't stop at developers themselves. It adversely affects the software industry as a whole by introducing a concept known as technical debt (Han et al., 2020). Technical debt is a metaphor that refers to the hidden costs of neglecting good coding practices. Just like financial debt accrues interest over time, poorly written code accumulates complexity and becomes harder

to understand and modify as the project grows (Digkas et al., 2022). This leads to several problems such as e.g, higher costs as bug fixes and new features become cumbersome, and an increased risk of software failure due to vulnerabilities.

Although existing code review platforms offer insights into basic coding style issues, there's an opportunity to enhance their capabilities for beginners, particularly by addressing the gap in feedback on the "why" behind conventions (Wessel et. al., 2020). Understanding the rationale is crucial for preventing bad habits and tangled code that create future maintenance and collaboration challenges. This highlights the need for better integration of coding conventions into the learning process. This study aims to address this need by developing an intelligent code-reviewing tool with a tutoring focus. A novel approach that has the potential to significantly improve the overall quality of code produced in our industry.

1.3. AIM AND OBJECTIVES

The aim of this study is to develop an intelligent code reviewing tool designed with a tutoring focus to empower beginner programmers in understanding and applying coding coventions using Natural Language Processing techniques and static code analysis. The objectives of this study are:

i. To identify the specific challenges faced by beginner programmers in learning programming conventions.
ii. To gather and process the coding styles and general standards data for the static code analyzer to identify potential style violations based on those datasets.
iii. To design and develop the intelligent explanation module and the interactive guidance features of the code reviewing tool.
iv. To design the user interface and user experience of the intelligent code review tool.
v. To implement a working prototype of the intelligent code review tool.

1.4. METHODOLOGY

i. Review of existing literature on programming education and analyzing and observation of existing systems.
ii. Acquiring sufficient datasets from coding standards documents or open-source code repositories and processing them using data processing tools such as Python libraries (Pandas, Natural Language Toolkit (NLTK) or Spacy).
iii. Modelling and designing the components of the intelligent code review tool using UML diagrams such as an activitiy diagram, sequence diagram, use case diagram etc. as well as system architecture diagrams.
iv. Designing the user interface and user experince of the tool using the design tool Figma.

v. Implementation of a working protoype of the tool using the React framework for the frontend, Rust for the static analyzer component and for the backend and transfer learning with Dolphin Model based on Mixtral (Hartford, 2023) for the explanation module.

Table 1.1: Objectives-Methodology Mapping Table

S/N	OBJECTIVES	METHODOLOGY
1	To identify the specific challenges faced by beginner programmers in learning programming conventions.	**Extensive Literature Review and Existing Systems** 1. Review of existing literature on programming education 2. Direct analysis and observation of existing systems.
2	To gather and process the coding styles and general standards data for the static code analyzer to identify potential style violations based on those datasets.	**Data Gathering and Processing** 1. Acquiring sufficient datasets from coding standards documents or open-source code repositories 2. Processing the data using data processing tools such as Python libraries (Pandas, Natural Language Toolkit (NLTK) or Spacy).
3	To design and develop the intelligent explanation module and the interactive guidance features of the code reviewing tool.	**Design and Development of Components** Modelling and designing the components of the intelligent code review tool using UML diagrams such as an activitiy diagram, sequence diagram, use case diagram etc. as well as system architecture diagrams.
4	To design and develop the intelligent explanation module and the interactive guidance features of the code reviewing tool.	**Design and Development of Components** Modelling and designing the components of the intelligent code review tool using UML diagrams such as an activitiy diagram, sequence diagram, use case diagram etc. as well as system architecture diagrams.
5	To implement a working prototype of the intelligent code review tool.	**Implementation of a working Prototype** ~~Implementation of a working protoype of~~ the tool using the React framework for the frontend, Rust for the static analyzer

1.5. SIGNIFICANCE OF THE STUDY

This project proposes a novel intelligent code review tool that addresses the lack of beginner-friendly learning resources on coding conventions. By offering real-time feedback, explanations, and interactive guidance, this tool empowers beginners to write cleaner code from the outset, while also introducing a potentially revolutionary approach to developer education through open-source LLMs and interactive features.

The benefits of this project extend far beyond individual programmers. Widespread adoption of this tool by educational institutions can lead to a new generation of developers writing cleaner, more maintainable code. This translates to significant cost savings for businesses due to improved code quality and maintainability. Additionally, fostering a common understanding of coding conventions can enhance collaboration and communication among developers, leading to more efficient code reviews and faster development cycles. Finally, equipping beginner programmers with a solid foundation in coding conventions can significantly reduce the time and resources required to onboard them into new projects. Overall, this project has the potential to significantly impact the software development industry by empowering programmers, fostering innovation in education, and ultimately contributing to a more efficient and productive development landscape.

1.6. SCOPE OF THE STUDY

This project maintains a strict development focus to address the gap in learning resources concerning coding conventions for beginner programmers. The intelligent code reviewing tool being developed will leverage existing open-source large language models trained on code to provide explanations and justifications. The user interface will prioritize clarity and ease-of-use for beginners, incorporating interactive features like visual aids, tutorials, and code refactoring suggestions to solidify understanding.

While the project excludes user research or the development of entirely new LLM models, it will concentrate on fine-tuning existing open-source options. Development will be completed within a 16-week timeframe, targeting beginner programmers with a foundational grasp of a chosen programming language. This clearly defined scope ensures the project remains focused on development while delivering a valuable tool to empower beginner programmers and enhance their coding skills.

1.7. LIMITATIONS OF THE STUDY

The major limitations of this study include the following:

1. **Limited Language Support:** Initially, the tool will only be able to review code written in one programming language, limiting its reach to programmers using that specific language.
2. **Training Data Dependency:** The quality of explanations and justifications provided by the LLM component heavily relies on the training data. Limited or biased data could lead to inaccurate or misleading responses.
3. **Limited Scope of Conventions:** The first iteration of the tool will focus on a core set of coding conventions. We plan to expand the scope to address a wider range of conventions in future versions.

1.8. STRUCTURE OF THE RESEARCH

Chapter One of the project contains an explanation of the project, problems with existing solutions, the need for an improved solution, the method of implementation, the significance of the study, and the limitations. Chapter Two describes the existing systems related to the project topic, the methodology, algorithm, and techniques used in related systems. Chapter Three describes the analysis and system design. Chapter Four shows the implementation of the system in detail and the results obtained. Chapter Five summarises the project and gives recommendations, suggestions, conclusions, and references.

Chapter 2.

CHAPTER TWO: LITERATURE REVIEW

2.1. PREAMBLE

The importance of coding conventions remains a hurdle for beginners in today's software development landscape, particularly when it comes to ensuring clean and well-formatted code. This literature review lays the groundwork for a solution: an intelligent code review system utilizing open-source, code-focused LLMs and static analysis. This literature begins with a review of programming conventions or coding standards, coding smells and code quality, code quality management, lack of programming conventions and it's effects. *not finished*

2.2. REVIEW OF PROGRAMMING CONVENTIONS

Programming conventions, also known as coding conventions or coding standards, are repositories of rules and guidelines that encompass all aspects of improving code quality (Smit et al., 2011). These act as software development guidelines that set limitations, promote specific practices, or enforce constraints (Rodrigues & Montecchi, 2019).

However, programming conventions are not static. As programming languages evolve and new paradigms emerge, these conventions co-evolve to influence how code is structured, written, and maintained. For example, the transition from procedural languages, C, to object-oriented languages, C++ and Java, introduced entirely new conventions related to class structures, inheritance, and polymorphism (Pressman & Maxim, 2014).

2.2.1. BRIEF HISTORY OF PROGRAMMING CONVENTIONS

In the early days, 1940s-1960s, of languages like FORTRAN and COBOL, conventions focused primarily on basic formatting and readability due to limited processing power and the linear nature of

these languages (Medoff, 2015). Clear documentation was crucial for maintaining code written on punch cards, so conventions often encouraged detailed comments and code structure mirroring the documentation layout.

With the rise of procedural programming languages like C in the 1960s-1980s, conventions evolved to address modularity and control flow. Indentation became a cornerstone for defining code blocks and functions, promoting better organization and readability (Kernighan & Ritchie, 1978). As code sharing and collaboration increased, platform-specific conventions emerged to ensure portability and maintainability across different systems (Pressman & Maxim, 2014).

The shift to object-oriented languages like C++ and Java in the 1980s-2000s necessitated entirely new conventions for class structures, inheritance, and polymorphism (Meyer, 1997). Community-driven coding standards like PEP 8 for Python, (Rossum et al., 2001), or Google's JavaScript Style Guide further emphasized code consistency within specific programming communities (Google, n.d.). These trends continue in the modern age, with conventions focusing on maintainability and scalability in complex, distributed software projects (Beck, 1999) Integration with code analysis tools further enforces adherence to conventions, ensuring clean, maintainable code remains a priority regardless of the programming language or paradigm used.

2.2.2. MAIN TYPES OF PROGRAMMING CONVENTIONS

There three main types of coding conventions which include: Naming conventions, Formatting conventions and commenting conventions.

In programming, naming conventions establish guidelines for naming variables, functions, classes, and other code elements. These conventions promote consistency, clarity, and readability throughout the codebase (Herka, 2022). Following them allows developers to readily grasp the purpose and role of different code elements. For instance, using descriptive names like `calculateTotalPrice` instead of cryptic abbreviations enhances code understandability and maintainability (Pugh, 2018).

Formatting conventions encompass practices related to the visual structure of code. These conventions dictate aspects such as indentation, spacing, line breaks, and the use of code blocks (Piater, 2005). Adhering to formatting standards enhances code maintainability and comprehension. Consistent formatting makes code easier to read, debug, and modify, especially when multiple developers collaborate on a project (Broad Research Communication Lab, n.d.).

Commenting conventions provide guidelines for writing clear, concise, and informative comments within the codebase (Piater, 2005). Comments are essential for explaining the functionality and intent of the code, especially in complex or critical sections. Effective comments help developers understand the reasoning behind specific code decisions, making it easier to maintain and update the code in the future (Broad Research Communication Lab, n.d.).

2.3. REVIEW ON CODING SMELLS AND CODE QUALITY

In software development, code quality plays a pivotal role in determining the reliability, maintainability, and efficiency of a software system. While robust code ensures a program's reliability and efficiency, clear and readable code simplifies future modifications and updates. High-quality code not only enhances the functionality and performance of an application but also simplifies the process of future modifications and updates.

Code quality is a concept that lacks a precise definition in the literature, often characterized by vague and varied interpretations (Keuning et al., 2023). To comprehensively assess or categorize code quality, it is essential to consider the static properties of the code as opposed to the dynamic properties of code such as correctness, test coverage, and runtime performance. The evaluation of these static properties, as outlined by Stegeman et al. (2016), encompasses criteria such as documentation, layout, naming, flow, expressions, idiom, decomposition, and modularization. Adherence to coding standards is essential across all these categories to ensure a high level of code quality.

However, problems found in these categories (Stegeman et. al., 2016), are called "code smells". This term was introduced by Fowler (1999). Code smells might hint at deeper problems in the de-

sign of functionally correct code, affecting the quality of code (Tufano et al., 2015), (Albuquerque et al., 2023). These smells can be a consequence of prioritizing short-term goals like reusability over long-term maintainability. While reusability is desirable for well-crafted software (Pandey et al., 2020), pressures like frequent requirement changes, increasing project size, and time constraints can lead to code deterioration (Verma et al., 2023).

One crucial aspect of code quality lies in the inherent trade-offs developers face when optimizing different code characteristics. For instance, prioritizing code readability and understandability might come at the expense of achieving peak performance (Sas & Avgeriou, 2020). Conversely, highly optimized code for speed might be less maintainable in the long run, requiring more effort for future modifications.

2.4. REVIEW ON CODE QUALITY MANAGEMENT

Code quality management refers to the systematic process of overseeing and ensuring the quality of code within a software development project. It involves implementing strategies, practices, and tools to maintain high standards of reliability, maintainability, and efficiency in the codebase (Sonar Source, n.d.).

For a long time, peer code review, a manual inspection of code by other developers on a software development team, has been recognised at a tool for improving the quality of code (Sadowski et. al., 2018). Through peer code review, developers can identify potential issues like bugs, code smells, and suboptimal coding practices. This collaborative approach fosters knowledge sharing, improves coding consistency, and ultimately elevates the overall quality of the codebase.

2.4.1. MODERN CODE REVIEW: A STREAMLINED APPROACH

However, the limitations of manual code review or peer review in today's fast-paced development environments are becoming increasingly apparent. The sheer volume of code produced, coupled with time constraints, can make thorough code review a challenge (Sadowski et. al., 2018). The immediate

solution adopted today to this limitation is called Mordern Code Review (MCR). MCR is a lightweight approach to traditional code inspections or peer review (Badampudi et al., 2023).

MCR's process as explained in the works by Badampudi et al., (2023) follows a series of six main steps that integrate with version contol systems (e.g., Gerrit, Github and GitLab). The steps involved in this process include:

1. In *Step 1*, the code author(s) or developers submit code or changes. Usually, version control systems like Gerrit and Github are used, the developer creates a pull request.
2. In *Step 2*, the project or code owner selects one or more reviewers, using heuristics, to review the new pull requests made by the code author(s).
3. In *Step 3*, the reviewer(s) are notified of their assignement. In
4. In *Step 4*, the reviewer(s) check the code for defects or suggest improvement.
5. In *Step 5*, the reviewer(s) and author(s) discuss the feedback from the review.
6. In *Step 6*, the pull request or code change is rejected or sent back to the author(s) for refinement. If no further rework is required, the code change is merged into the codebase.

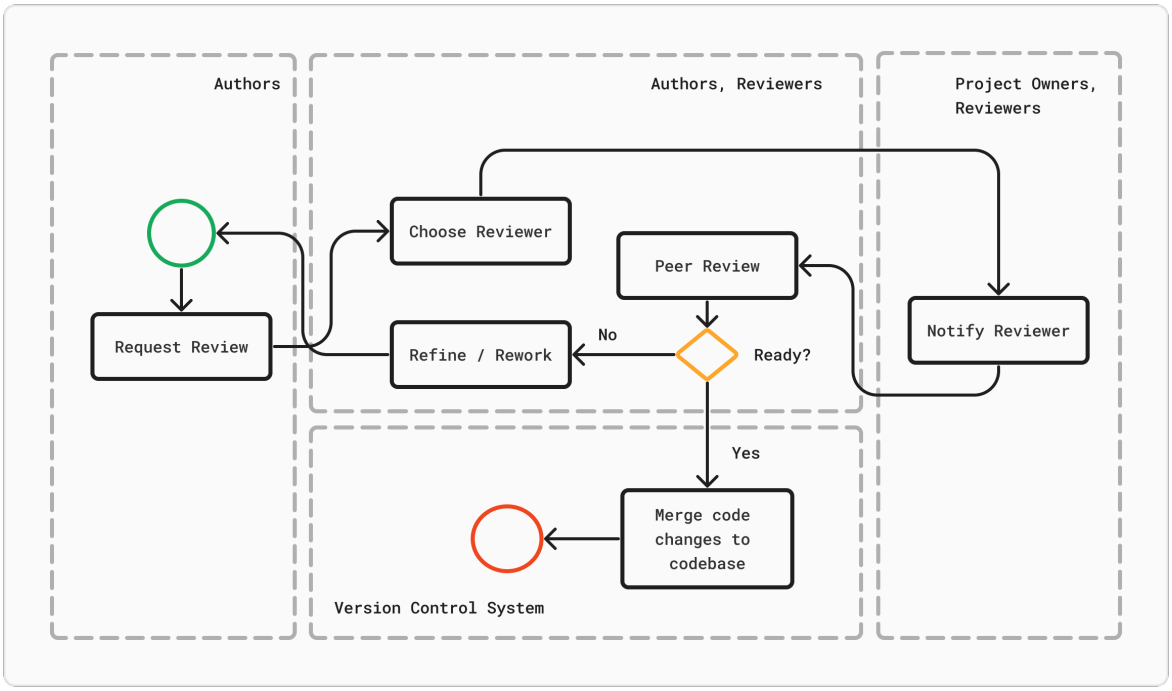

Figure 2.1: Overview of steps in modern code reviews, adapted from (Badampudi et al., 2023)

2.4.2. INTELLIGENT CODE REVIEW TOOLS

Software development teams using the MCR process struggle to keep pace with the ever-increasing scale of software projects (J. Siow, 2019), as code reviewers often have to allocate a huge amount of time in performing code review tasks due to the amount of code changes in large-scale codebases. While some part of the code review processes have been automated with the use of linters or static analysis tools that contain rules related to coding best practices (Bielik et al., 2017), there's still a huge amount of effort put into the review process, as these tools mainly detect common issues and leave important aspects such as defects, architectural issues, and testing for code (Gupta, 2018).

Although defect detection remains important in these code review processes (Lingling Fan, 2018), industry developers increasingly value code quality aspects that promote long-term maintainability, understanding of source code and code improvement (Alberto Bacchelli, 2013). Many research studies aim to reduce the amount of time spent on code review processes by providing solutions to close the learning gap of these systems. Research work such as DeepCodeReviewer (DCR) by (Gupta, 2018), aims to recommend reviews by learning the relevancy between the source code and review and Code Review Bot, (Kim et. al., 2022), is designed to process review requests holistically regardless of such environments, and checks various quality-assurance items such as potential defects in the code, coding style, test coverage, and open source license violations.

Essentially, intelligent code review tools, such as DeepCodeReviewer (DCR) and Code Review Bot, use deep learning and automation to enhance the code review process. DCR, for example, recommends relevant code reviews based on historical data, while Code Review Bot checks for potential defects, coding style, test coverage, and open source license violations. These tools have been found to be effective in improving the efficiency of change-based code review. They have also been well-received by developers, with positive feedback and active response to reviews.

2.4.3. STATIC ANALYSIS

2.5. REVIEW ON THE EFFECTS OF UNCLEAN CODE

2.5.1. TECHINICAL DEBT

2.6. REVIEW OF RELEVANT CONCEPTS

1. artificial intelligence
2. transfer learning
3. LLMs
4. static analysis:
    1. parsing
    2. tokenization: lexical analysis
    3. syntatic analysis
    4. AST

2.7. REVIEW OF RELATED METHODS

2.8. REVIEW OF EXISTING SYSTEMS

References

Alberto Bacchelli, C. B. (2013). Expectations, outcomes, and challenges of modern code review. *2013 35th International Conference on Software Engineering (ICSE)*, 712–721. https://api.semanticscholar.org/CorpusID:220663293

Albuquerque, D., Guimarães, E., Perkusich, M., Almeida, H., & Perkusich, A. (2023). Integrating Interactive Detection of Code Smells into Scrum: Feasibility, Benefits, and Challenges. *Applied Sciences*, *13*(15). https://doi.org/10.3390/app13158770

Badampudi, D., Unterkalmsteiner, M., & Britto, R. (2023). Modern Code Reviews—Survey of Literature and Practice. *ACM Trans. Softw. Eng. Methodol.*, *32*(4). https://doi.org/10.1145/3585004

Bancroft, P., & Roe, P. (2006). *Program annotations: Feedback for students learning to program*. 19–23.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company.

Bielik, P., Raychev, V., & Vechev, M. (2017). *Learning a Static Analyzer from Data*. 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

Broad Research Communication Lab. *Coding and comment style : Broad institute of MIT and harvard*. https://mitcommlab.mit.edu/broad/commkit/coding-and-comment-style/

Chen, H.-M., Chen, W.-H., & Lee, C.-C. (2018). An automated assessment system for analysis of coding convention violations in Java programming assignments*. *Journal of Information Science and Engineering*, *34*, 1203–1221. https://doi.org/10.6688/JISE.201809_34(5).0006

Digkas, G., Chatzigeorgiou, A., Ampatzoglou, A., & Avgeriou, P. (2022). Can Clean New Code Reduce Technical Debt Density?. *IEEE Transactions on Software Engineering*, *48*(5), 1705–1721. https://doi.org/10.1109/TSE.2020.3032557

Dijkstra, E. W., Dahl, O. J., & Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Google. *Google javascript style guide*. https://google.github.io/styleguide/jsguide.html

Gupta, A. (2018). *Intelligent code reviews using deep learning*. https://api.semanticscholar.org/CorpusID:52219239

Han, D., Ragkhitwetsagul, C., Krinke, J., Paixao, M., & Rosa, G. (2020). Does code review really remove coding convention violations?. *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, *0*, 43–53. https://doi.org/10.1109/SCAM51674.2020.00010

Hartford, E. (2023). *dolphin-mixtral-8x7b*. Cognitive Computations. https://erichartford.com/dolphin-25-mixtral-8x7b

Herka, I. (2022). *Naming conventions in programming – a review of scientific literature — Makimo – Consultancy & Software Development Services*. https://makimo.com/blog/scientific-perspective-on-naming-in-programming/

InstituteData. (2023). *Understanding coding conventions in software engineering | institute of data*. https://www.institutedata.com/blog/software-engineering-coding-conventions/

J. Siow, L. F. S. C. Y. L., Cuiyun Gao. (2019). CORE: Automating Review Recommendation for Code Changes. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 284–295. https://api.semanticscholar.org/CorpusID:209439473

Joni, S.-N., & Soloway, E. (1986). But My Program Runs! Discourse Rules for Novice Programmers. *Journal of Educational Computing Research*, *2*, . https://doi.org/10.2190/6E5W-AR7C-NX76-HUT2

Kernighan, B. W., & Ritchie, D. M. (1978). *The C programming language*. Prentice-Hall, Inc.

Keuning, H., Heeren, B., & Jeuring, J. (2021). A Tutoring System to Learn Code Refactoring. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, 562–568. https://doi.org/10.1145/3408877.3432526

Keuning, H., Jeuring, J., & Heeren, B. (2023). A Systematic Mapping Study of Code Quality in Education. *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 5–11. https://doi.org/10.1145/3587102.3588777

Kim, H., Kwon, Y., Joh, S., Kwon, H., Ryou, Y., & Kim, T. (2022). Understanding automated code review process and developer experience in industry. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1398–1407. https://doi.org/10.1145/3540250.3558950

Kourie, D. G., & Pieterse, V. (2008). Reflections on coding standards in tertiary computer science education. *South African Computer Journal*, *41*, 29–37.

Lingling Fan, S. C. G. M. Y. L. L. X. G. P., Ting Su. (2018). Efficiently Manifesting Asynchronous Programming Errors in Android Apps. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 486–497. https://api.semanticscholar.org/CorpusID:51954364

Medoff, M. (2015). *The evolution of coding standards*. https://www.exida.com/Blog/the-evolution-of-coding-standards

Meyer, B. (1997). *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc.

Pandey, A. K., Tripathi, A., Alenezi, M., Agrawal, A., Kumar, R., & Khan, R. A. (2020). A Framework for Producing Effective and Efficient Secure Code through Malware Analysis. *International Journal of Advanced Computer Science and Applications*, *11*(2). https://doi.org/10.14569/IJACSA.2020.0110263

Piater, J. (2005). *Formatting*. https://iis.uibk.ac.at/public/piater/courses/Coding-Style/ar01s03.html

Popić, S., Velikic, G., Jaroslav, H., Spasic, Z., & Vulić, M. (2018). *The Benefits of the Coding Standards Enforcement and its Impact on the Developers Coding Behaviour-A Case Study on Two Small Projects*. https://doi.org/10.1109/TELFOR.2018.8612149

Pratap, P. (2023). The evolution of computer programming languages. *International Journal of Advanced Research in Science, Communication and Technology*, *3*, 69–76. https://doi.org/10.48175/ijarsct-13110

Pressman, R. S., & Maxim, B. R. (2014). *Software engineering : a practitioner's approach*. Mcgraw-Hill Education.

Pugh, D. (2018). *A brief list of programming naming conventions*. https://www.deanpugh.com/a-brief-list-of-programming-naming-conventions

Rodrigues, E., & Montecchi, L. (2019). *Towards a structured specification of coding conventions*. https://doi.org/10.1109/prdc47002.2019.00047

Rossum, G. van, Warsaw, B., & Coghlan, N. (2001). *PEP 8 – Style Guide for Python Code | peps.python.org*. https://peps.python.org/pep-0008/

Ruvo, G., Tempero, E., Luxton-Reilly, A., Rowe, G., & Giacaman, N. (2018). *Understanding semantic style by analysing student code*. 73–82. https://doi.org/10.1145/3160489.3160500

Sadowski, C., Söderberg, E., Church, L., Sipko, M., & Bacchelli, A. (2018). Modern code review: a case study at google. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 181–190. https://doi.org/10.1145/3183519.3183525

Sas, D., & Avgeriou, P. (2020). Quality attribute trade-offs in the embedded systems industry: an exploratory case study. *Software Quality Journal*, *28*(2), 505–534. https://doi.org/10.1007/s11219-019-09478-x

Smit, M., Gergel, B., Hoover, H. J., & Stroulia, E. (2011). *Code convention adherence in evolving software*. https://doi.org/10.1109/ICSM.2011.6080819

Sonar Source. *What is code quality? Definition guide*. https://www.sonarsource.com/learn/code-quality/

Stegeman, M., Barendsen, E., & Smetsers, S. (2016). Designing a rubric for feedback on code quality in programming courses. *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 160–164. https://doi.org/10.1145/2999541.2999555

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and Why Your Code Starts to Smell Bad. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, *1*, 403–414. https://doi.org/10.1109/ICSE.2015.59

Verma, R., Kumar, K., & Verma, H. K. (2023). Code smell prioritization in object-oriented software systems: A systematic literature review. *Journal of Software: Evolution and Process*, *35*(12), e2536. https://doi.org/https://doi.org/10.1002/smr.2536

Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., & Gerosa, M. A. (2020). What to Expect from Code Review Bots on GitHub? A Survey with OSS Maintainers. *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 457–462. https://doi.org/10.1145/3422392.3422459

Wiese, E. S., Yen, M., Chen, A., Santos, L. A., & Fox, A. (2017). Teaching students to recognize and implement good coding style. *Proceedings of the Fourth (2017) ACM Conference on Learning*. https://doi.org/10.1145/3051457.3051469