DEVELOPMENT OF AN INTELLIGENT CODE REVIEWER RECOMMENDATION SYSTEM FOR PROGRAMMING CONVENTIONS

BY

EKEKWE TANYALOUISE KELECHI CHISOM
(20CG028072)

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.

MAY 2024

CERTIFICATION

I hereby certify that this project was carried out by Ekekwe Tanyalouise Kelechi Chisom (20CG028072) in the Department of Computer and Information Sciences, College of Science and Technology, Covenant University, Ogun State, Nigeria, under my supervision.

Dr. Itunuoluwa Isewon _____

Supervisor **Signature and Date**

Professor Olufunke O. Oladipupo _____

Head of Department **Signature and Date**

DEDICATION

I dedicate this work to God, who has been my very present help in times of need during my 4-year journey in this institution. I also dedicate this work to my friends and family, who support me.

ACKNOWLEDGEMENTS

My most profound gratitude goes to Almighty God, who has kept me and sustained me right from the very start of my degree up until this point. To Him be all the glory.

I want to express my sincere gratitude to my younger sister, Daniella Ekekwe, who continues to encourage me to follow my passion for computer science, especially during this project.

I am extremely grateful to my parents, Mr and Mrs Ekekwe for their sacrifices, contributions, and prayers towards the completion of this degree. I am extremely privileged to have them in my life.

My sincere gratitude also goes to my hardworking supervisor, Dr Itunuoluwa Isewon, who provided her support even when she was on a work break. Thank you for your excellent supervision and guidance ma.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This page has been intentionally left blank.

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 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 significant 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, Heeren, & Jeuring, 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 real-time 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 frequently 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 the 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 as 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). Poor code quality leads to problems such as 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 codes that create future maintenance and collaboration challenges. This highlights the need to integrate coding conventions into the learning process better. This study addresses 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 conventions using Natural Language Processing techniques and static code analysis. The objectives of this study are:

i. To identify the specific challenges beginner programmers face 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 (UI) and user experience (UX) of the intelligent code review tool.
v. To implement a working prototype of the intelligent code review tool.

1.4. METHODOLOGY

i. Review existing literature on programming conventions and analyze and observe 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 intelligent code review tool components using UML diagrams such as an activity diagram, sequence diagram, use case diagram, etc., and system architecture diagrams.
iv. Designing the user interface (UI) and user experience (UX) of the tool using the design tool Figma.
v. Implementation of a working prototype of the tool using the React framework for the frontend, Rust for the static analyzer component, and for the backend and transfer learning with the 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** i. Review existing literature on programming conventions ii. Analyze and observe 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** i. Acquiring sufficient datasets from coding standards documents or open-source code repositories. ii. Processing 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 intelligent code review tool components using UML diagrams such as an activity diagram, sequence diagram, use case diagram, etc., and system architecture diagrams.
4	To design the user interface (UI) and user experience (UX) of the intelligent code review tool.	**UI/UX Design of Components** Designing the user interface (UI) and user experience (UX) of the tool using the design tool Figma.

S/N	OBJECTIVES	METHODOLOGY
5	To implement a working prototype of the intelligent code review tool.	**Implementation of a working Prototype** Implementation of a working prototype of the tool using the React framework for the frontend, Rust for the static analyzer component, and for the backend and transfer learning with the Dolphin Model based on Mixtral (Hartford, 2023) for the explanation module.

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:

i. **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.
ii. **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.
iii. **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 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 encompass the review of programming conventions or coding standards, coding smells and code quality, code quality management, lack of programming conventions and it's effects.

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).

```python
# Variable and function names in lowercase_with_underscores
total_count = 0
calculate_total()

# Class names in CapitalizedCamelCase
class UserAccount:
    pass

# Constant names in ALL_CAPS
PI = 3.14159
```

Figure 2.1: Some naming conventions in python

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 design 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). During the process, one or more reviewers assess the code for errors, adherence to coding standards, test coverage, and more. These MCR tools automate many aspects of the review process, making it faster and more efficient compared to traditional peer review methods that rely solely on manual processes (Qiao et al., 2024). For example, tools like GitHub, GitLab, and provide features such as inline commenting, automated code formatting checks, and integration with Continuous Integration/Continuous Deployment (CI/CD) pipelines, streamlining the review process.

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:

i. 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.
ii. In *Step 2*, the project or code owner selects one or more reviewers, usually using heuristics, to review the new pull requests made by the code author(s).
iii. In *Step 3*, the reviewer(s) are notified of their assignement.
iv. In *Step 4*, the reviewer(s) check the code for defects or suggest improvement.
v. In *Step 5*, the reviewer(s) and author(s) discuss the feedback from the review.
vi. 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.

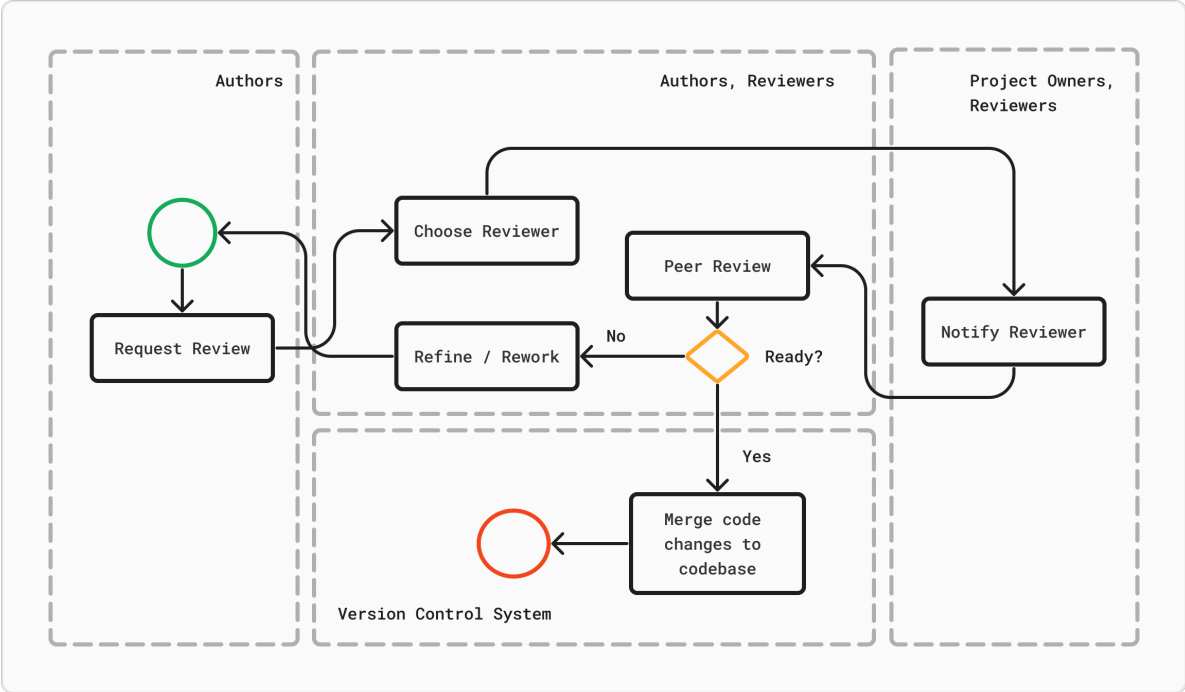

Figure 2.2: Overview of steps in modern code reviews, adapted from (Badampudi et al., 2023)

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 (Fan et al., 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

Static code analysis tools play a crucial role in software development by assisting developers in enhancing the quality of their code. These tools statically evaluate source code to identify bugs, security vulnerabilities, duplications, and code smells. As noted by Møller & Schwartzbach (2020), the practice of static analysis has also been instrumental in optimizing compilers since the 1960s and has since expanded to aid in software debugging and quality improvement (Danilo et al., 2021). The adoption of static analysis tools has become increasingly prevalent in the industry, offering developers insights into code quality standards and potential defects (Ilyas & Elkhalifa, 2016).

There are various types of static analysis tools available, each specializing in different programming languages and types of defects. Some tools focus on general-purpose code analysis, while others are tailored for specific languages like C/C++, Java, or Python. These tools such as Cppcheck, FindBugs, and SonarQube, use predefined rules or patterns to identify issues and provide actionable insights to developers for improving code quality. Static analysis tools are continuously evolving to meet the changing needs of software development and to provide more comprehensive coverage in detecting defects and vulnerabilities within the codebase (Nachtigall et al., 2019).

2.5. REVIEW ON THE EFFECTS OF UNCLEAN CODE

"Unclean code" is a term used to describe source code that is difficult to maintain, evolve, and change, often due to poor software engineering practices (Silva Carvalho et al., 2017). This type of code is characterized by poor readability, lack of maintainability, and the presence of "code smells" - indicators of software design problems (Gupta, 2018). These code smells, like long and complex functions scattered throughout the codebase (making them harder to identify and manage) can significantly increase the likelihood of errors being introduced during modifications, change-proneness, and faults remaining undetected, fault-proneness (Palomba et al., 2017). As a result, unclean code can lead to a vicious cycle of bugs, rework, and project delays, ultimately impacting software quality and project success.

While the importance of clean code in preventing resource loss has been well-established (Digkas et. al., 2022), unclean code remains a persistent challenge in real-world software development projects (Chirvase et al., 2021). Several factors contribute to this issue, including a lack of developer motivation, limited knowledge of clean code principles, and a general lack of awareness regarding the importance of code quality. Inappropriate code reuse practices, often fueled by time pressures and influenced by the ethical climate within a development team (Sojer et al., 2014), further exacerbate the problem. To address this challenge, it's crucial to encourage developers to prioritize clean code practices and reassess their professional values in relation to their craft.

2.5.1. TECHNICAL DEBT

Technical debt (TD) is a metaphor that represents the compromise between maintaining clean, well-structured code and rapid development (Ampatzoglou et al., 2016). It encompasses the design choices, development decisions, and coding practices that prioritize immediate convenience over long-term sustainability (Albuquerque et al., 2022). While technical debt (TD) can offer short-term benefits, such as reducing time-to-market, it can ultimately harm the overall quality of software systems by introducing bugs, increasing complexity, and hindering future maintenance efforts (Rios et al., 2019).

In managing technical debt, it is essential to consider various strategies for addressing prioritization, refactoring, and automation. Prioritizing between new

features and debt repayment involves assessing the impact on software quality, business value, and risks (Lenarduzzi et al., 2021). Refactoring practices focus on restructuring code to improve readability, maintainability, and quality without altering external behavior. Continuous refactoring helps reduce technical debt and enhance long-term maintainability. Automated tools for code analysis, such as SonarQube and Findbugs, can help identify technical debt indicators like code smells, duplication, and complexity, aiding efficient debt detection and resolution.

2.6. REVIEW OF RELEVANT CONCEPTS

This section provides an in-depth explanation of the various concepts that hold significance in the context of this project. These concepts include:

2.6.1. ARTIFICIAL INTELLIGENCE

Artificial intelligence (AI) is the ability of a machine or computer system to perform tasks that typically require human intelligence, such as logical reasoning, learning, and problem-solving (Morandín-Ahuerma 2022). It is based on machine learning algorithms and technologies, allowing machines to apply cognitive abilities and perform tasks autonomously or semi-autonomously (Morandín-Ahuerma 2022).

AI can be categorized by its cognitive capacity and autonomy, with various degrees of each (Morandín-Ahuerma 2022). The field of AI encompasses a range of modules, including knowledge representation, problem-solving, and natural language processing (Tecuci 2012). It is designed to imitate human cognitive abilities and can handle complex problems in an intelligent and adaptive manner (Wu 1986).

2.6.2. TRANSFER LEARNING

Transfer learning is a machine learning strategy that includes exploiting knowledge learnt from a task to improve performance on a separate but connected task. In other words, transfer learning entails applying pre-trained models or knowledge from one activity to improve learning in another.

Transfer learning is often used in deep learning to improve a model's performance on a new task by applying knowledge learnt from a previously trained model. The pre-trained model can be trained on a huge dataset and utilized as a place to begin for a new, related task. The learnt parameters of the pre-trained model can be fine-tuned or updated with less data tailored to the current job, resulting in a model that can learn faster and with a greater degree of precision (Zhuang et al., 2021)

Transfer learning can be described as a technique that is extremely useful in NLP tasks when there is a shortage of training data for a certain task. A pre-trained language model trained on a big corpus, for example, can be changed or fine-tuned to suit to a specific language-generation task using a comparatively smaller dataset. Researchers have also utilized transfer learning to improve performance on similar tasks, such as using a pre-trained sentiment analysis model to improve efficiency on a text classification task (Ruder, Peters, Swayadimpta, & Wolf, 2019).

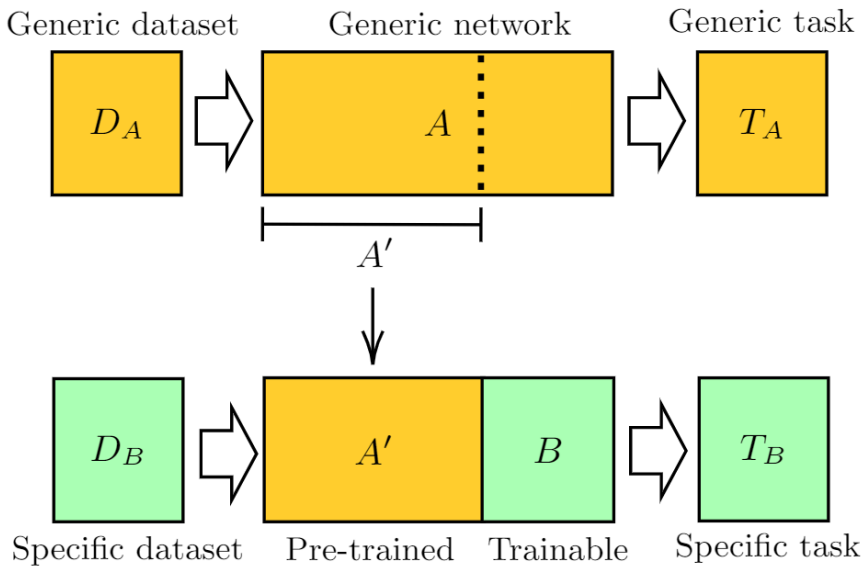

Figure 2.3: Overview of transfer learning

Nowadays, transfer learning is an effective strategy for increasing the efficiency and accuracy of machine learning models, especially when significant amounts of training data are unavailable for use.

2.6.3. LANGUAGE LEARNING MODELS

Large Language Models (LLMs) have emerged as a revolutionary force in various domains, and software engineering is no exception. These powerful AI models, trained on massive text datasets, exhibit remarkable capabilities in processing and generating human-like language.

LLMs are demonstrating potential in automating code generation. By analyzing large codebases and identifying patterns, they can assist developers by generating code snippets, completing repetitive tasks, and even suggesting potential implementations for functionalities. This can significantly boost developer productivity and reduce boilerplate code writing. Research by (Guu et al., 2021) showcases how LLMs like Codex can generate functional code with impressive accuracy, particularly for well-defined tasks.

In recent developments, LLMs have proved valuable in enhancing code understanding. Their ability to analyze code structure and semantics allows them to extract insights and answer developer queries. This can be particularly beneficial for complex codebases or legacy systems where understanding existing code functionality can be challenging. For instance, (Xu et al., 2021) demonstrate how LLMs can be trained to summarize code functionalities and identify potential code smells, aiding developers in code comprehension and refactoring efforts.

Finally, LLMs hold promise for revolutionizing code review practices. By learning from human-written code reviews, they can assist developers by identifying potential issues like syntax errors, logical flaws, and adherence to coding standards. This can streamline the review process by highlighting areas requiring attention and suggesting potential fixes. Studies by (Liu et al., 2023) explore the potential of LLMs to generate code review comments, highlighting areas for improvement and providing relevant code documentation or examples. However, it's crucial to remember that LLMs are still under development, and human expertise remains essential for complex code analysis and decision-making.

2.6.4. PARSING

Parsing is a fundamental concept in computer science, particularly in programming languages and natural language processing. It refers to the process of analyzing a string of symbols, according to a set of rules, to determine its

structure and meaning. Imagine it as taking a jumbled sentence and rearranging the words to form a grammatically correct and understandable sentence.

Parsing, or syntax analysis, is a crucial aspect of computer science and linguistics, with applications in advanced compilers, computational linguistics, web browsers, and data compression programs (Grune, 2008). It involves the analysis of code to identify its structure and meaning and is often used in programming exercises to reinforce concepts of parsing, regular and context-free grammar, and abstract syntax trees (Werner, 2003). In the context of on-the-fly syntax highlighting, Palma (2022) proposes a deep learning-based approach that can achieve near-perfect accuracy in predicting correct and incorrect language derivations, outperforming regular expression-based strategies. This approach is particularly useful in online collaborative tools for software developers, where code highlighting quality is often sacrificed for system responsiveness.

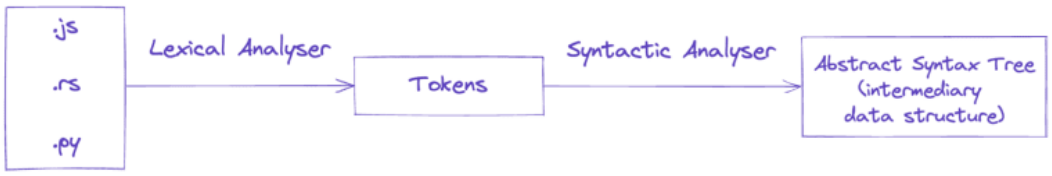

Figure 2.4: A schema of a common parsing processes

2.6.5. TOKENIZATION: LEXICAL ANALYSIS

Tokenization involves breaking down a stream of code into smaller, meaningful units called tokens. These tokens can be keywords (like `if`, `for`, or `while`), identifiers (user-defined names like variable or function names), operators (such as `+`, `-`, or `*`), or special symbols (like `;`, `(`, or `)`). The process is analogous to how we segment sentences into individual words. Just as spaces separate words in a sentence, specific delimiters in the code (like whitespace, punctuation, or operators) signal the boundaries between tokens in the code stream. v
The importance of tokenization in static analysis stems from its role in transforming raw code into a structured representation. This structured representation, often in the form of an abstract syntax tree (AST), allows for efficient analysis of the code's syntax, semantics, and potential issues. By identifying the individual tokens and their types, static analysis tools can understand the basic building blocks of the code and perform checks for errors, inefficiencies, or security vulnerabilities. Tokenization paves the way for

further analysis phases, enabling static analysis tools to reason about the code's structure, data flow, and potential control flow paths.

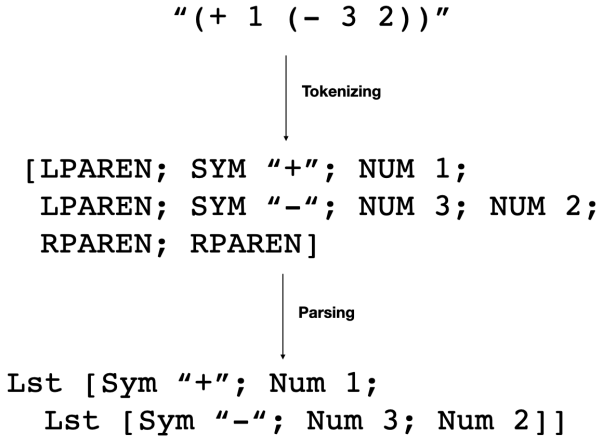

Figure 2.5: Tokenization and parsing overview.

In conclusion, tokenization serves as the essential first step in the static analysis process. By segmenting code into meaningful units, it transforms raw code into a structured format suitable for further analysis. This structured representation empowers static analysis tools to identify and flag potential issues within the code, ultimately contributing to improved code quality, security, and maintainability.

2.6.6. SYNTACTIC ANALYSIS

Following the initial breakdown of code into tokens during tokenization, static analysis enters the stage of syntactic analysis. Here, the focus shifts from individual elements to understanding the overall structure and organization of the code. This analysis aims to verify if the code adheres to the grammatical rules, or syntax, of the programming language.

Syntactic analysis often employs a parser, a program specifically designed to analyze the sequence of tokens. The parser leverages a set of formal rules, typically defined by a context-free grammar, to determine if the arrangement of tokens is valid according to the language's syntax. Imagine it as checking if a sentence follows the grammatical rules of a language, ensuring proper word order and clause structure. If the parser successfully processes the entire sequence of

tokens without encountering any violations, the code is considered syntactically correct.

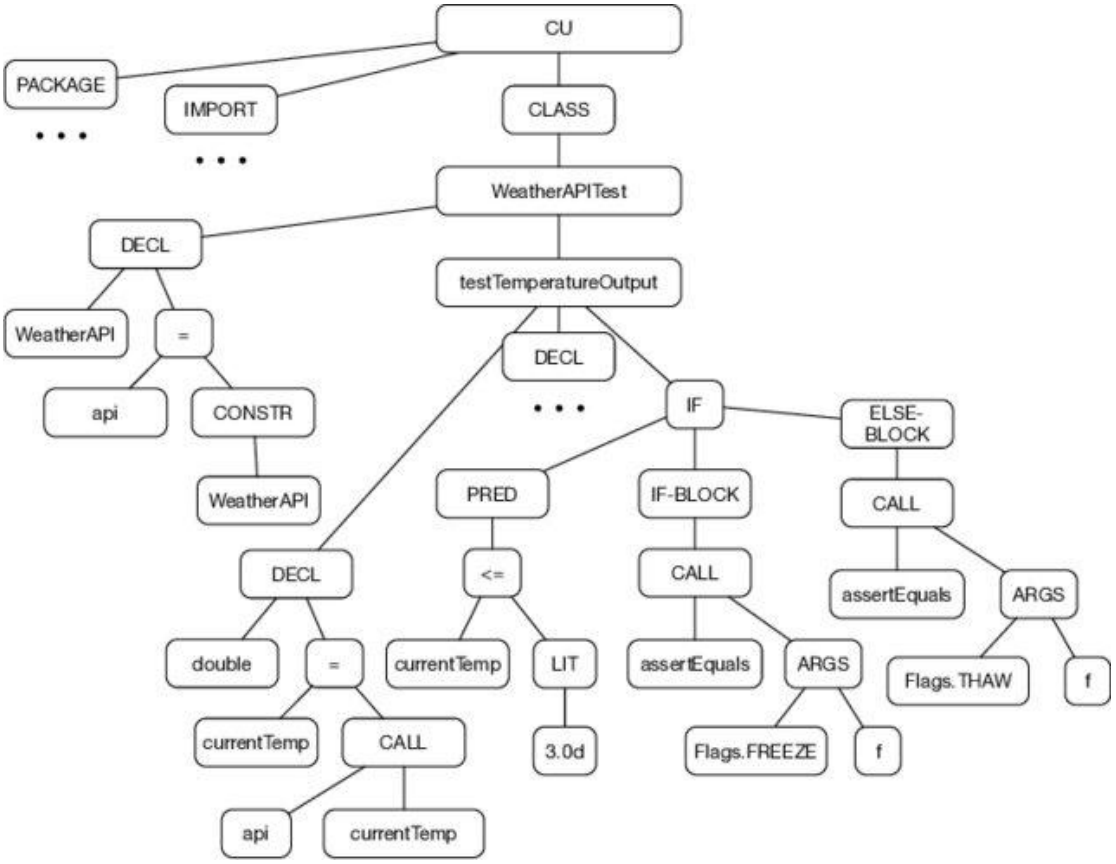

Figure 2.6: A simplified abstract syntax tree (AST) (Samoaa, Longa, Mohamad, Haghir Chehreghani, & Leitner, 2022).

A crucial outcome of successful syntactic analysis is the generation of an Abstract Syntax Tree (AST). This AST serves as a tree-like representation of the code's structure, capturing the hierarchical relationships between different code elements. The AST essentially becomes a blueprint of the code, reflecting its organization and flow without including concrete details like variable names or specific values. This high-level representation proves invaluable for subsequent static analysis phases, enabling further checks for semantic errors, potential inefficiencies, or security vulnerabilities within the code's overall structure.

2.7. REVIEW OF RELEVANT SYSTEMS

The review of relevant systems in this paper aims to explore various tools and approaches that have been developed to enhance code quality and improve the code review process. These systems include:

2.7.1. A TUTORING SYSTEM TO LEARN CODE REFACTORING (KEUNING, HEEREN, & JEURING, 2021)

In the field of programming education, a significant focus has been placed on developing tutoring systems and assessment tools. These tools help students by identifying errors in their code and assigning grades. However, there is a gap in these existing systems – they do not adequately address aspects of code quality like style and maintainability. These qualities are crucial for professional software development, as poorly written code can be difficult to understand, maintain, and test. Novice programmers, in particular, often struggle with writing clean and maintainable code.

This paper describes a tutoring system designed to address this gap. The system provides students with exercises that begin with functionally correct code, but the code is poorly written. Students are tasked with improving this code by applying refactorings in a step-by-step manner. The system offers hints and feedback to guide students throughout this process. These hints are presented in a hierarchical structure, allowing students to explore different levels of detail for each issue they encounter.

One of the key strengths of this system is that it is specifically designed for novice programmers. The system utilizes terminology and phrasing that is easy for beginners to understand. Additionally, the guidance provided by the system is gradual. Students can request more specific hints or explore alternative suggestions based on their needs. The system also focuses on issues that are commonly faced by novice programmers. These issues are identified through research and through the input of experienced teachers. The feedback and hints provided by the system are based on the suggestions from these teachers.

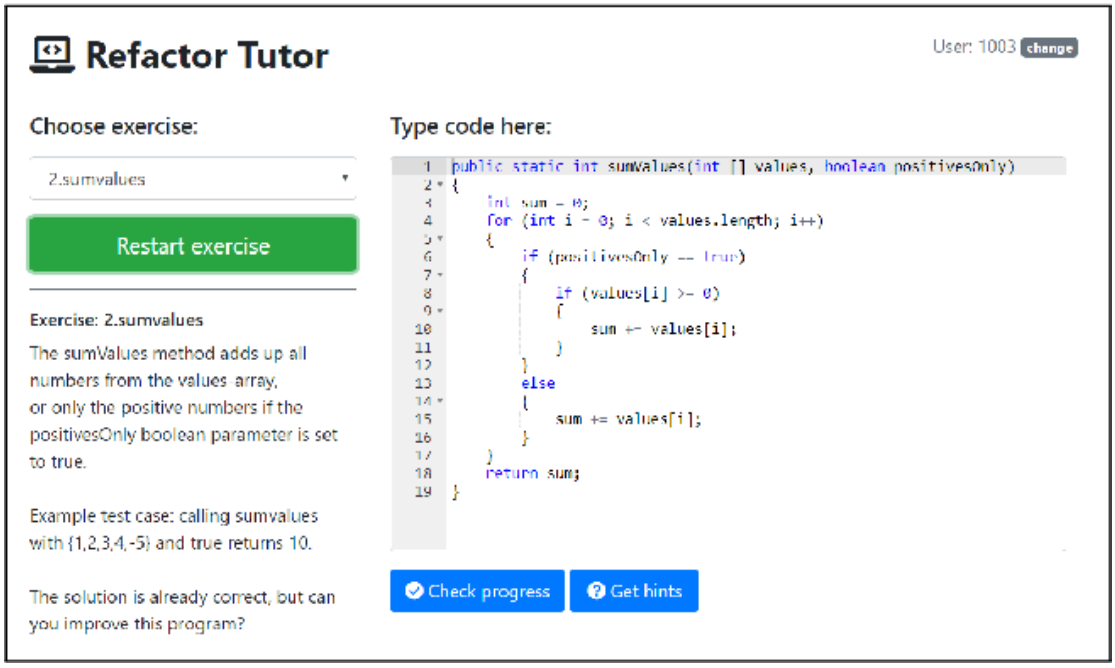

Figure 2.7: Web application for the tutoring system (Keuning et al., 2021)

An evaluation of the system was conducted by comparing the system's feedback with suggestions from teachers on how to improve student code. The results showed a strong correlation between the suggestions offered by the system and those provided by the teachers. Additionally, a preliminary study with students indicated that the system's hints help solve refactoring exercises.

While this tutoring system offers a valuable approach, it has limitations compared to intelligent code review systems. The tutoring system's effectiveness relies on a predefined set of exercises and rules, which may not cover the vast potential scenarios encountered in real-world coding. Additionally, the system requires manual effort from teachers to create model solutions for generating new rules.

Leveraging static analysis and LLMs offers several advantages over the tutoring system. First, LLM-based systems can continuously learn and adapt from vast code repositories, identifying a broader range of issues and keeping pace with evolving coding practices. This scalability eliminates the need for a predefined set of exercises and manual rule creation by teachers in the tutoring system. Additionally, LLMs can analyze code for a wider range of quality concerns, including security vulnerabilities and performance bottlenecks, while also considering the code's context for more nuanced feedback. Overall, LLM-based code review offers a more scalable, adaptable, and comprehensive

approach to improving code quality, potentially better preparing students for the complexities of real-world coding.

Table 2.2: Code Quality Improvement Methods: Tutoring System vs. Intelligent Code Review

Feature	**Tutoring System**	**Intelligent Code Review**
Methodology	Predefined set of exercises and pre-defined rules	Machine learning on vast code repositories
Adaptability	Limited, requires manual effort for expansion	Scalable, learns and adapts automatically
Feedback Style	Step-by-step guidance for prescribed refactorings	Nuanced feedback based on code context
Knowledge Base	Static set of teacher-defined rules	Continuously learns from code repositories
Efficiency	Requires teacher effort to create model solutions	Potentially learns from existing code examples

2.7.2. CODACY

While numerous SCA tools exist, Codacy offers a user-friendly and comprehensive solution specifically designed to streamline the code review process.

One of Codacy's key strengths lies in its ease of use. The platform boasts a user-friendly interface that simplifies integration with popular development workflows and version control systems (Codacy, 2013). This allows developers to seamlessly incorporate SCA into their existing development routine, minimizing disruption and promoting continuous code improvement. Furthermore, Codacy

offers analysis for a broad range of programming languages, making it a versatile tool for teams working on diverse projects (Codacy, 2013).

While user-friendliness and versatility are valuable features, Codacy goes beyond basic functionalities. The platform leverages a powerful suite of static analysis engines, enabling it to detect a wide spectrum of code quality issues (Codacy, 2013). This includes identifying potential security vulnerabilities, adherence to coding best practices, and opportunities for code refactoring. Additionally, Codacy provides developers with clear and actionable feedback, guiding them towards resolving identified issues and enhancing overall code quality (Codacy, 2013).

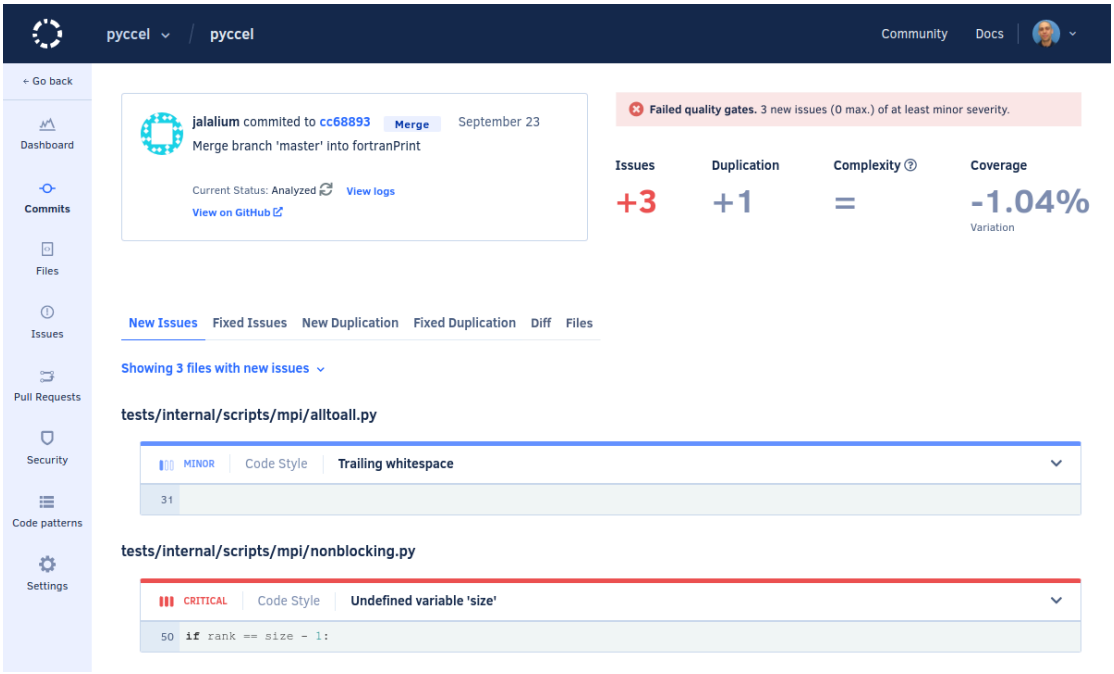

Figure 2.8: Overview of issues in recent commits on Codacy Dashboard

While Codacy offers a user-friendly and comprehensive SCA solution, it has limitations compared to LLM-based intelligent code review tools. Codacy relies on a predefined set of rules and analysis engines, potentially missing emerging coding issues or remaining blind to certain coding styles. LLM-based tools, on the other hand, can continuously learn and adapt from vast code repositories, potentially identifying a broader range of code quality concerns and best practices that may not be explicitly encoded within Codacy's rule set.

Table 2.3: Code Quality Improvement Methods: Codacy vs. Intelligent Code Review

Feature	**Tutoring System**	**Intelligent Code Review**
Methodology	Predefined set of exercises and pre-defined rules	Machine learning on vast code repositories
Adaptability	Limited, based on rule updates	Scalable, Continuous learning and adaptation
Scope of Issues Detected	Defined set of coding issues	Broader range of issues and best practices
Knowledge Base	Static set of analysis rules	Continuously learns from code examples

2.8. SUMMARY OF LITERATURE REVIEW

This review of literature explored various approaches to improving code quality. A code review system utilizing static analysis and LLMs (Large Language Models) emerged as a more promising solution. LLM-based systems benefit from continuous learning capabilities, enabling them to identify a wider range of code quality issues and adapt to evolving coding practices. Additionally, they can analyze code for context-specific nuances, leading to more comprehensive feedback compared to a static set of rules.

Overall, LLM-based code review offers a more scalable, adaptable, and effective approach to improving code quality, particularly for preparing students for the demands of professional software development. The proposed approach will utilize transfer learning approach by fine-tuning the Dolphin Model (Hartford, 2023) with a smaller conventions dataset.

CHAPTER THREE
SYSTEM ANALYSIS AND DESIGN

3.1. PREAMBLE

This section details the analysis and design of the proposed system. It presents the requirements for the system, the techniques taken to achieve the system as well as diagrams to model the proposed system.

3.2. THE PROPOSED SYSTEM

This proposed project will be a web-based application. It will allow users to connect their codebases through code hosting repositories like GitHub or allow users to import their code into the application. The application then provides an extensive review of the code, highlighting areas that need improvement, and giving recommendations on how to improve the code. The system will also provide explanations and justifications for the suggestions made using intelligent responses from a language model.

3.3. REQUIREMENTS ANALYSIS

In software development, requirement analysis includes the descriptions of the services that would be provided by the system, as well as its operational limits. It lays down the features that are required to suit the users' requirements, these requirements are then further separated into functional requirements, non-functional requirements and user experience requirements. The various requirements for the proposed system are detailed in this section.

3.3.1. FUNCTIONAL REQUIREMENTS

Functional requirements of the system are specific functionalities or services that the system is expected to perform. They describe what the application should do and include operations, activities, computational tasks, data manipulation, user interface behavior, and more. These requirements are as follows:

i. Users should be able to input their code or to link their respective codebases from code hosting sites.

ii. Users should be allowed to select the coding conventions they want to adhere to.
iii. The system should analyze the code and provide feedback on areas that need improvement.
iv. The system should generate appropriate and contextually relevant explanations and justifications based on the content of the code and the suggestions made.
v. The system should allow users to view the code and the suggestions side by side.
vi. The system should allow users to accept or reject the suggestions made.
vii. The system should provide a summary of the code review.
viii. The system should allow users to download the reviewed code.

3.3.2. NON-FUNCTIONAL REQUIREMENTS

Non-functional requirements are not directly related to the system functionality but rather are defined in terms of system performance. They explain the system behavior, features and overall attributes that can affect the user's experience. The non-functional requirements for the system in this study are as follows:

i. The system should be user-friendly and easy to navigate.
ii. The system should seamlessly engage with code hosting repositories like GitHub for easy retrieval and pushing of code.
iii. The system should provide feedback within a reasonable time frame.
iv. The system should be able to provide explanations and justifications that are contextually relevant to the submitted code.
v. The system should be able to provide a summary of the code review.

3.3.3. USER EXPERIENCE REQUIREMENTS

User experience requirements are the factors that specify the expected things the user should encounter while making use of the system and the outcome of their experience. In the case of this system, the user experience requirements include:

i. The system should be easy to use and navigate.
ii. The system should provide clear and concise feedback.
iii. The system should provide explanations and justifications that are easy to understand.
iv. The system should provide a summary of the code review.
v. The system should allow users to download the reviewed code.

3.4. DATA COLLECTION

Two large datasets were used to re-train/fine-tune the Dolphin 2.5 Mixtral 8x7b model. The first dataset, MSR 2019, is a large dataset containing javascript code snippets extracted from Stack Overflow posts and the results of running ESLint to identify potential errors and stylistic issues (Ferreira Campos et al., 2019). The second dataset comprises parsed ASTs in JSON format used in the study by Raychev et al. (2016). These datasets are extensive, containing more than 480,000 JavaScript code files.

3.4.1. DESCRIPTION OF THE MSR 2019 DATASET

Ferreira Campos, Smethurst, Moraes, Bonifácio, and Pinto (2019) conducted a study to characterize the common coding violations in contemporary JavaScript code, resulting in the MSR 2019 dataset. It contains 336,000 JavaScript code snippets extracted from Stack Overflow posts. The code snippets were retrieved from SOTorrent (Baltes et al., 2019), a dataset that curates the extraction and evolution of Stack Overflow code snippets. The data was pre-processed and stored in a CSV file for further processing. The CSV file is in the following format:

i. ID of the question (PostId)
ii. Content (in this case the code block)
iii. Length of the code block
iv. Line count of the code block
v. Score of the post
vi. Title

PostId	Content	Length	LineCount	score	title
111111	function newClosure(someNum,	766	21	7654	How do JavaScript closures work?
111111	function buildList(list) {
&#:	530	17	7654	How do JavaScript closures work?
111111	var gLogNumber, gIncreaseNum	574	18	7654	How do JavaScript closures work?
111111	function say667() {
239	9	7654	How do JavaScript closures work?
5767357	Array.prototype.indexOf || (Array	437	15	6245	How do I remove a particular element from an array in JavaScript
359509	var a = [1,2,3];
var	336	14	5674	Which equals operator (== vs ===) should be used in JavaScript c
15012542	.directive('whenActive', functior	522	15	4523	"Thinking in AngularJS" if I have a jQuery background?
15012542	<ul class="main-menu">
&	451	16	4523	"Thinking in AngularJS" if I have a jQuery background?
15012542	.directive('myDirective', functior	392	12	4523	"Thinking in AngularJS" if I have a jQuery background?
15012542	.directive('myDirective', functior	413	12	4523	"Thinking in AngularJS" if I have a jQuery background?
14220323	function findItem() {
214	10	4379	How do I return the response from an asynchronous call?
14220323	// Using 'superagent' which will r	1175	32	4379	How do I return the response from an asynchronous call?
14220323	function foo(callback) {
&#:	235	9	4379	How do I return the response from an asynchronous call?
14220323	function delay() {
628	18	4379	How do I return the response from an asynchronous call?
14220323	function ajax(url) {
465	19	4379	How do I return the response from an asynchronous call?
14220323	function checkPassword() {&#xD	368	15	4379	How do I return the response from an asynchronous call?
14220323	checkPassword()
.(	273	11	4379	How do I return the response from an asynchronous call?
950146	function loadScript(url, callback)	566	14	4206	How do I include a JavaScript file in another JavaScript file?
9329476	// `a` is a sparse array
&#xA	387	14	3828	For-each over an array in JavaScript?

Figure 3.9: A sample of the rows and columns of the pre-processed javascript code snippet dataset.

A simple Python script then processed the CSV file, extracting the code and merging all code blocks into their respective JavaScript files named after the PostID. This resulted in 336,000 Javascript files. Next, a script split the processed dataset into 20 evenly distributed parts and ran 20 instances of ESLint to generate reports, producing 20 JSON files.

3.4.2. DESCRIPTION OF THE 150K JAVASCRIPT DATASET

The second dataset, which was used to train and evaluate the DeepSyn tool Raychev, Bielik, Vechev, & Krause (2016), comprises JavaScript programs collected from GitHub repositories. This dataset was meticulously preprocessed to remove duplicate files, project forks, and obfuscated files. The programs were then parsed using the error-tolerant Acorn parser and subsequently tokenized and converted into Abstract Syntax Trees (ASTs) in JSON format. For training and evaluation purposes, the dataset was divided into training, validation, and test sets. The resulting dataset is a comprehensive collection of parsed AST serialized in JSON format, ready for analysis and application.

As an example, given a simple program to print “Hello World!” in javascript:

```
console.log("Hello World!");
```

Figure 3.10: Example javascript code

The serialized AST in JSON format is as follows:

```
[
    {"id":0, "type": "Program", "children":[1]},
    {"id":1, "type": "ExpressionStatement", "children":[2]},
    {"id":2, "type": "CallExpression", "children":[3,6]},
    {"id":3, "type": "MemberExpression", "children":[4,5]},
    {"id":4, "type": "Identifier", "value": "console"},
    {"id":5, "type": "Property", "value": "log"},
    {"id":6, "type": "LiteralString", "value": "Hello World!"}
]
```

Figure 3.11: Example of a serialized AST in JSON format of the chosen dataset

To evaluate the performance after tuning the Dolphin Mixtral model, we will utilize 50,000 files from this dataset.

3.5. PHYSICAL DESIGN

A physical design represents the physical elements of a software system and relates to the actual input/output operations of the system. The focus is on how data is entered into the system, validated, processed, and output. It has two major categories, the input design and the output design. For this system, the physical design is concerned with how the code is inputted into the system, how the system processes the code, and how the system outputs the results of the code review. These design considerations are essential to generate a working system that specifies all the features of a candidate system.

3.5.1. THE PROPOSED SYSTEM ARCHITECTURE

The system architecture describes the structure of the system and the components that make up the system. It provides a high-level overview of how the system will be designed and how the components will interact with each other. The system architecture for the proposed system is detailed in this section.

The proposed intelligent code review application is composed of several components designed to streamline and enhance the code review process. These components include the Code Importer or Fetcher mechanism (Github integration or manually importing code), a simple static analysis engine, a user-friendly interface for users, a recommendations module or generator (Dolphin Mixtral model), and a comprehensive reporting system. Additionally, the application integrates with version control systems such as Git. The functions of the components of the architecture are as follows:

Code Importer/Fetcher:
This component allows users to import their code into the system either by linking their codebases from code hosting repositories like GitHub or by manually uploading their code. This component serves as the entry point for the code review process.

Static Analysis Architecture
The system's static analyzer plays a crucial role in examining user-submitted code. It begins by tokenizing the code, parsing it, and constructing an abstract syntax tree (AST) to comprehend the code's structure and semantics. Through this analysis, it identifies code quality concerns, adherence to coding conventions, and areas for enhancement. The static analyzer encompasses several key components:

i. Lexer: The lexer tokenizes raw source code into tokens, facilitating further analysis.

ii. Parser: The parser constructs an Abstract Syntax Tree (AST) using a recursive descent parsing method, capturing code hierarchy systematically.

iii. Semantic Analyzer: The semantic analyzer verifies the code's semantic correctness, ensuring adherence to programming language rules and identifying logical errors.

iv. Code Rule Enforcement: This component checks code against coding standards, detecting stylistic issues and ensuring consistency.

v. Report Generation: Detailed reports are created, highlighting the identified issues, the AST, and code snippets. This report serves as a query that would be fed to the LLM for contextual recommendations or suggestions based on the violations found in the analysis.

User Interface

The user interface component provides an intuitive and user-friendly platform for users to interact with the system. It offers functionalities for code submission, viewing analysis results, accepting or rejecting suggestions, and downloading the reviewed code.

Recommendations Module

The recommendations module, powered by the Dolphin Mixtral model, generates contextually relevant suggestions based on the code analysis results. Leveraging the model's deep learning capabilities, the module provides nuanced feedback, explanations, and justifications for the identified issues. It offers actionable insights to users, guiding them toward code quality improvement and best practices.

Reporting System

The reporting system compiles the analysis results, recommendations, and justifications into a comprehensive report. This report summarizes the code review process, highlighting the identified issues, suggested improvements, and contextual explanations. Users are allowed to accept the suggested changes from this report. It serves as a valuable resource for users to understand the code quality concerns and the rationale behind the recommendations.

The proposed system architecture is designed to streamline the code review process, enhance code quality, and provide users with actionable insights for code

improvement. By integrating static analysis, deep learning, and user-friendly interfaces, the system aims to offer a comprehensive solution for code review and quality assurance.

3.6. LOGICAL DESIGN

The logical design of a system describes the functional requirements of the system and how these requirements are implemented. It refers to the conceptual and abstract representation of a system's architecture, functionality, and components. For the proposed system, the goal of logical design is to define the system's structure, data flow, interfaces, and interactions between various components in a way that is independent of any particular technology or platform. The logical design of the proposed system is detailed in this section.

3.6.1. USE CASE DIAGRAM

A use case diagram is a visual representation of the interactions between users and a system. It illustrates the various ways users can interact with the system and the system's responses to these interactions.

The use-case diagram for the proposed system shows that a user can interact with the system by importing their code, analyzing the code, viewing the analysis results, accepting or rejecting suggestions, and downloading the reviewed code. The system, in turn, processes the code, generates recommendations, provides explanations, and compiles a summary report for the user.

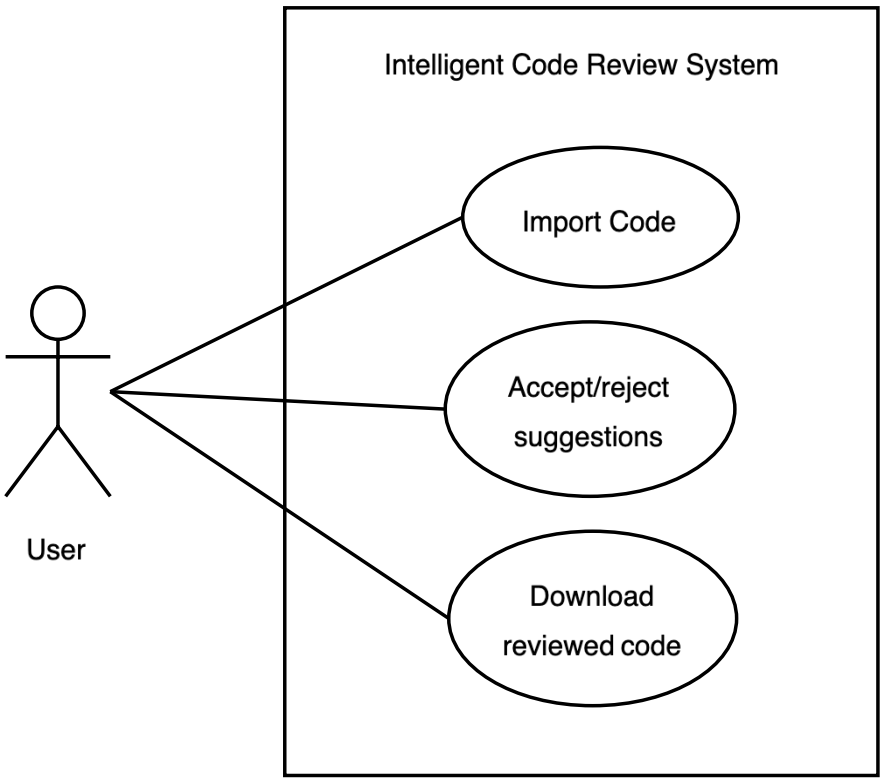

Figure 3.12: Use Case Diagram for the Proposed System

The use-case diagram provides a high-level overview of the system's functionality and user interactions, outlining the key features and capabilities of the proposed system.

3.7. ACTIVITY DIAGRAM

An activity diagram illustrates the flow of activities within a system, showing the sequence of actions and decisions that occur during a process. It provides a detailed view of how the system processes information and performs tasks, describing the steps in a use case diagram.

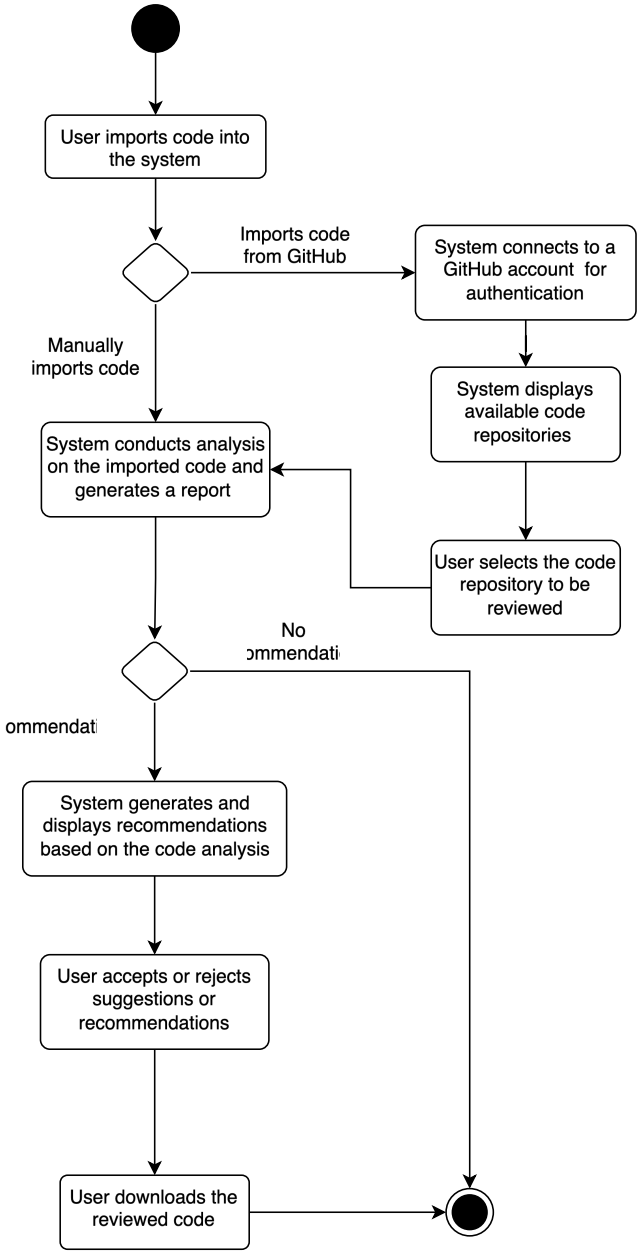

Figure 3.13: Use Case Diagram for the Proposed System

3.8. SEQUENCE DIAGRAM

A sequence diagram shows how objects interact in a particular scenario of a use case. It illustrates the sequence of messages exchanged between objects, highlighting the order of interactions and the flow of control in a system.

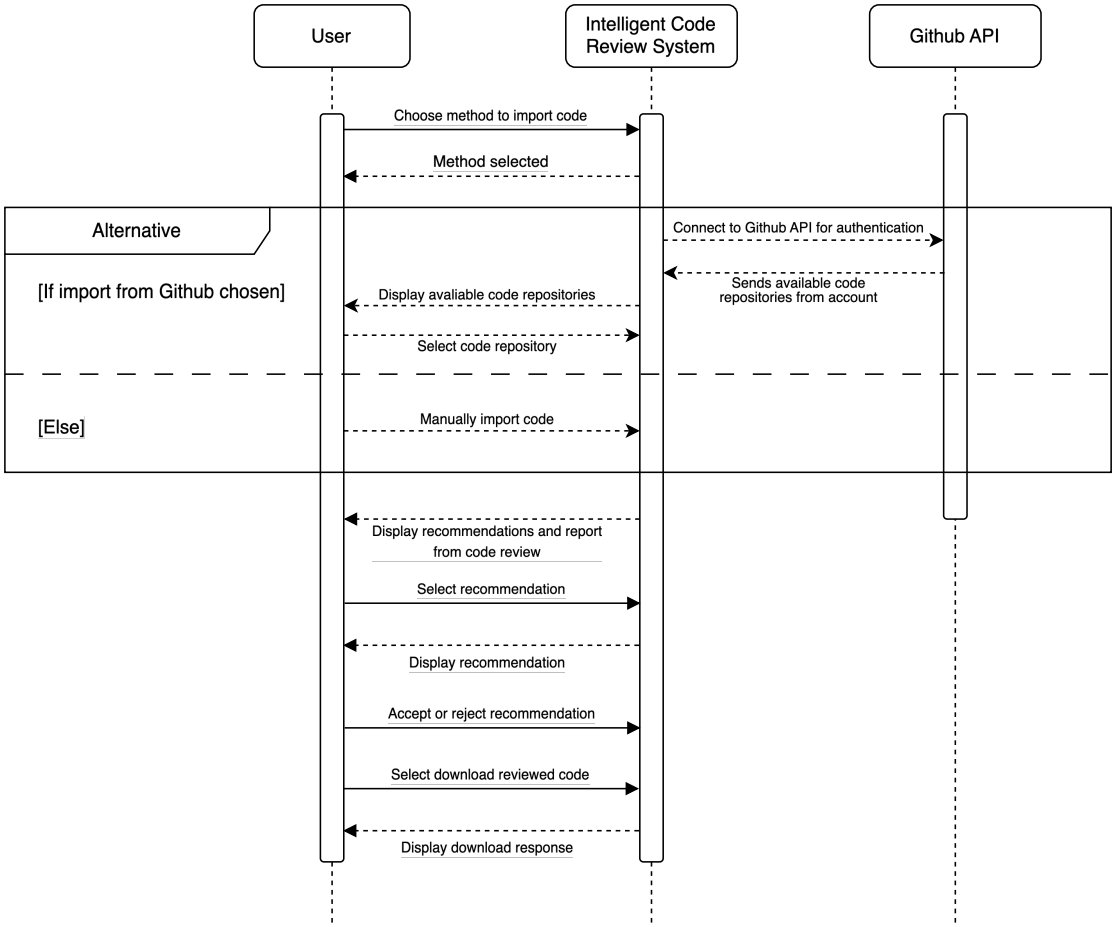

Figure 3.14: Sequence Diagram for the Proposed System

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., Guimaraes, E. T., Tonin, G. S., Perkusich, M. B., Almeida, H., & Perkusich, A. (2022). Perceptions of Technical Debt and its Management Activities - A Survey of Software Practitioners. *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, 220–229. https://doi.org/10.1145/3555228.3555237

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

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., & Systa, K. (2016). The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, *0*, 9–16. https://doi.org/10.1109/MTD.2016.8

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

Baltes, S., Treude, C., & Diehl, S. (2019). SOTorrent: studying the origin, evolution, and usage of stack overflow code snippets. *Proceedings of the 16th International Conference on Mining Software Repositories*, 191–194. https://doi.org/10.1109/MSR.2019.00038

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

Chirvase, A., Ruse, L., Muraru, M., Mocanu, M., & Ciobanu, V. (2021). Clean Code - Delivering A Lightweight Course. *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, *0*, 420–423. https://doi.org/10.1109/CSCS52396.2021.00075

Codacy. (2013). *An easy-to use code quality review solution*. https://blog.codacy.com/loosely-coupled-architecture

Danilo, N., Stefanović, D., Dakic, D., Sladojevic, S., & Ristic, S. (2021). *Analysis of the Tools for Static Code Analysis*. 1–6. https://doi.org/10.1109/INFOTEH51037.2021.9400688

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.

Fan, L., Su, T., Chen, S., Meng, G., Liu, Y., Xu, L., & Pu, G. (2018). *Efficiently manifesting asynchronous programming errors in Android apps*. 486–497. https://doi.org/10.1145/3238147.3238170

Ferreira Campos, U., Smethurst, G., Moraes, J. P., Bonifácio, R., & Pinto, G. (2019). Mining Rule Violations in JavaScript Code Snippets. *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, *0*, 195–199. https://doi.org/10.1109/MSR.2019.00039

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/

Ilyas, B., & Elkhalifa, I. (2016). *Static Code Analysis: A Systematic Literature Review and an Industrial Survey*. https://api.semanticscholar.org/CorpusID:65258085

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.

Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., & Arcelli Fontana, F. (2021). A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*, *171*, 110827–110828. https://doi.org/https://doi.org/10.1016/j.jss.2020.110827

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.

Møller, A., & Schwartzbach, M. I. (2020). *Static Program Analysis*. https://cs.au.dk/~amoeller/spa/

Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). Explaining Static Analysis - A Perspective. *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, *0*, 29–32. https://doi.org/10.1109/ASEW.2019.00023

Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R., & Lucia, A. D. (2017). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, *23*, 1188–1221. https://api.semanticscholar.org/CorpusID:215767620

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

Qiao, Y., Wang, J., Cheng, C., Tang, W., Liang, P., Zhao, Y., & Li, B. (2024). Code Reviewer Recommendation Based on a Hypergraph with Multiplex Relationships. *Arxiv*. https://api.semanticscholar.org/CorpusID:267061002

Raychev, V., Bielik, P., Vechev, M., & Krause, A. (2016). Learning programs from noisy data. *SIGPLAN Not.*, *51*(1), 761–774. https://doi.org/10.1145/2914770.2837671

Rios, N., Mendonça, M., Seaman, C., & Spínola, R. (2019). *Causes and Effects of the Presence of Technical Debt in Agile Software Projects*.

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

Samoaa, P., Longa, A., Mohamad, M., Haghir Chehreghani, M., & Leitner, P. (2022). *TEP-GNN: Accurate Execution Time Prediction of Functional Tests using Graph Neural Networks*.

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

Silva Carvalho, L. P. da, Novais, R. L., Nascimento Salvador, L. do, & Mendonça, M. G. (2017). An Ontology-based Approach to Analyzing the Occurrence of Code Smells in Software. *International Conference on Enterprise Information Systems*. https://api.semanticscholar.org/CorpusID:21143193

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

Sojer, M., Alexy, O., Kleinknecht, S., & Henkel, J. (2014). Understanding the Drivers of Unethical Programming Behavior: The Inappropriate Reuse of Internet-Accessible Code. *Journal of Management Information Systems*, . https://doi.org/10.1080/07421222.2014.995563

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