Towards a System for Measuring Source Code Quality

Jorge Hernandez*, Christian Unigarro and Hector Florez

Universidad Distrital Francisco Jose de Caldas, Bogota, Colombia

Abstract
Source code quality is an essential feature for software performance. Ideally, when the software is used, users should feel that it operates efficiently, ensuring that functional and non-functional requirements are met. This work discusses using metrics and attributes to evaluate software code quality, focusing primarily on object-oriented programming for academic projects. We describe the proposed software architecture designed to generate inspections for Python and Java projects and present the quality results.

Keywords
Software Quality, POO, Data Analysis, Software metrics, Technical Debt, Quality Attributes

1. Introduction

Object-oriented programming (OOP) has been a fundamental paradigm in software development for many years. Its origins trace back to the 1960s when the creation of Simula was widely regarded as the first object-oriented programming language. Developed in Norway by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center, Simula was initially designed as a simulation language to model complex systems. However, it introduced key concepts that would later become foundational to OOP, such as classes, objects, inheritance, and dynamic binding [1].

These innovations provided a new way of thinking about programming by organizing code around "objects" that represent both data and behavior, rather than merely focusing on procedures or functions. The introduction of classes allowed for the creation of reusable code structures, and inheritance enabled the extension of existing classes to create new functionality without duplicating code. Simula's impact extended beyond simulations, as its principles became the basis for more widespread adoption of OOP in other areas of software development.

The influence of Simula as the precursor of OOP can be seen in many modern programming languages, including Java, C++, and Python. These languages are commonly used to develop robust applications and continue to be a relevant topic of study due to the emergence of new devices with specific requirements. Modern applications must adapt to varying screen sizes, processing speeds, and new communication methods for new devices, such as smart TVs, wearables, and virtual reality glasses. Additionally, integrating artificial intelligence models in software and hardware for training, validation, and testing processes requires applying OOP concepts to improve code readability, which is crucial in deep learning. As we saw before OOP has helped to improve the quality of software, for example, attributes like maintainability and readability, becoming more critical as large development teams are tasked with building and evolving software in a rapidly changing environment [2, 3, 4].

Considering software quality, issues related to code have been widely discussed. Authors like Martin Fowler introduced the concept of "code smells," which help identify underlying problems that impact code execution and quality [5]. Although not considered direct metrics, these code smells can influence software quality attributes. In the 1990s, companies began focusing on measuring software quality and offering services to improve code quality. A notable example is McCabe [1] with one of the most

ICAIW 2024: Workshops at the 7th International Conference on Applied Informatics 2024, October 24–26, 2024, Viña del Mar, Chile
*Corresponding author.
jehernandezr@udistrital.edu.co (J. Hernandez); cjunigarrod@udistrital.edu.co (C. Unigarro); haflorezf@udistrital.edu.co (H. Florez)
0000-0002-7129-7919 (J. Hernandez); 0000-0002-5339-4459 (H. Florez)

[1]http://www.mccabe.com/

important, Cyclomatic Complexity metric, which measures the number of linearly independent paths through a program's source code, essentially quantifying its control flow complexity. Researchers also developed metrics such as MOOD (Metrics for Object-Oriented Design) [6] and MOOSE (Metrics for Object-Oriented Software Engineering) [7] and works by authors like Lorenz and Kidd [8] that continue to be referenced today.

New programming languages such as Golang, Scala, Rust, Kotlin, and Swift, have been born to support object-oriented programming. The diversity in syntax and compilation methods across these languages presents a challenge for existing OOP metrics, requiring adaptations to validate the quality of code written in these languages.

Innovations in measuring attributes such as performance have been achieved thanks to the use of machine learning. For example in this study [9], models are used to generate input data that simulates potential issues in software applications. This provides a more effective way to measure performance by identifying failures and bottlenecks related to resource usage, such as CPU and memory consumption.

In the academic context, there is a growing effort to enhance the learning of OOP by providing tools for both students and lecturers. Rubrics for student evaluation often rely on predefined metrics and tools that detect OOP issues are sometimes used. Feedback plays a critical role in enabling students to learn from their mistakes and improve over time. Understanding the relationship between code smells and quality attributes can improve the feedback provided to lecturers. Additionally, metrics help students avoid common mistakes and improve the overall quality of their applications [10].

In summary, this study aims to contribute to the evolving field of object-oriented programming by proposing a system to measure metrics that address the unique challenges posed by emerging technologies in the domain of the academy. Our approach leverages the metrics established in previous studies and established OOP metrics to provide a comprehensive framework for evaluating software quality.

This paper is structured as follows: Section 2 presents the concepts and terms for the metrics and attributes of software quality. Section 3 presents the other's works. Section 4 shows the new proposal system to generate inspections for Java and Python. Section 5 presents the next work. Section 6 concludes the paper.

2. Background

The definition of metrics in software quality derives from a variety of sources, including documentation from researchers and companies that have developed approaches to assess software quality [11]. In this section, we analyze several approaches to software quality metrics, from widely recognized standards and quality attributes cited by key authors to more recent research, which includes the examination of issues like code smells that can negatively affect software quality. Additionally, we cover basic concepts of object-oriented programming (OOP) and the programming languages considered in this study, as these elements are essential for understanding how metrics apply to real-world software development.

Software is generally written to solve specific problems, based on a list of requirements. The number of lines of code and its complexity vary depending on factors such as the programming paradigm, language, and operations needed to produce the desired results. Moreover, problems can often be approached in several ways, leading to different implementation variations, which complicates the evaluation of software quality.

To address this challenge, several evaluation methods are employed currently to assess code quality:

- **Testing**. This can include unit tests, integration tests, and acceptance tests.
- **Code Review**. A collaborative evaluation by a team to ensure code quality.
- **Static Code Analysis**. The use of tools to analyze and generate a score for the code.

Metrics for software quality can generally be classified into two main categories:

1. **Code execution.** Related to issues like code smells that can affect runtime performance and behavior.

Category	**Meaning**	**Examples of Code Smells**
Bloasters	Code that has grown too large and complex, making it difficult to maintain.	Long Method, Large Class, Long Parameter List
Object-Orientation Abusers	Code that violates object-oriented principles, complicating design modifications.	Switch Statements, Refused Bequest
Change Preventers	Code that is hard to modify without affecting other areas.	Divergent Change, Shotgun Surgery
Disposables	Unnecessary code that can be safely removed.	Duplicated Code, Dead Code
Couplers	Excessive coupling between components, complicating dependency management.	Feature Envy, Message Chains
Encapsulators	Issues with data encapsulation and information hiding.	Inadequate Encapsulation, Lack of Information Hiding

Table 1
Code Smell Categories, Meanings, and Examples

2. **Design-focused.** Concerned with the structure, flows, and requirement definitions of the software.

It´s important to point out that design-focused metrics are more difficult to measure as they cannot be assessed solely by evaluating code or class relationships. To measure these metrics, one must analyze how the code serves as a resource, how design patterns are applied, and how class interactions reveal the overall flow and architecture of the system.

2.1. Code Smells

A code smell is an indicator in source code that suggests potential problems in the software's structure or design, leading to issues such as poor maintainability, increased complexity, or future bugs. These smells often signal the need for refactoring to improve code quality and readability [12].

Martin Fowler, in Refactoring: Improving the Design of Existing Code, identifies 22 types of code smells, categorized to highlight common issues that impede code maintainability, comprehension, or extensibility. Notable examples include Long Method, Large Class, Divergent Change, and Feature Envy, which often require refactoring [5].

The relationship between code smells and software quality attributes is key to this study. Metrics assess code quality by reflecting attributes such as maintainability and scalability, and these attributes can be affected by specific code smells. Moreover, each smell is linked to a refactoring technique, creating a bridge between metrics, code smells, and quality attributes [13].

Table 1 categorizes code smells by the type of issue they pose, offering a structured approach for identifying and addressing these during code reviews and refactoring sessions.

By linking code smells with refactoring techniques and quality attributes, this study aims to provide comprehensive feedback to both students and professionals, helping them improve their software by addressing quality-related issues more effectively.

2.2. Software Quality Attributes

The metrics for evaluating source code are often based on quality attributes [14]. A quality attribute is a measurable characteristic that reflects how well the software meets non-functional requirements. These attributes are evaluated from an operational perspective to measure overall software performance.

Table 2 presents the ISO/IEC 25010 software quality model (SQuare). This framework defines key attributes that contribute to the enhancement of software products. The ISO/IEC 25010 model classifies quality into eight distinct characteristics. By systematically evaluating these attributes, the model

Attribute	Description
Functional Suitability	The ability of software to provide functions that meet the needs expressed by the business. This includes aspects such as correctness, completeness, and suitability.
Performance	The ability of the software to be transferred from one environment to another. This includes installability, replaceability, adaptability, and co-existence with other software.
Compatibility	It is the ability of the software to operate in different environments.
Usability	It is the ability of the software to be easily used. Mainly, it refers to the aesthetics of the user interfaces. The aim is for the user to be able to learn very quickly how to use the software.
Reliability	The ability of software to tolerate failures over time under given conditions.
Security	The software's ability to protect and secure data from being read or extracted by third parties.
Maintainability	The ability of the software to effectively implement a change in components.
Portability	The ability of the software to be transferred from one environment to another.

Table 2
Principal Attributes of Software Quality. Based on: ISO/IEC 25010 Software Quality Model.

ensures that the software can effectively meet user needs. The achievement of these quality attributes in our approach is crucial for user satisfaction.

2.3. Standards for software quality

Software architecture is based on non-functional requirements as a way to evaluate software quality. Most companies that produce software products implement QA processes. These processes can be guided by the standards that are already in place for the different entities.

Table 3 presents the main standards that exist today to evaluate and understand the quality of software. The ISO organization presents standards to create a software quality model. Additionally, it presents frameworks to define security and metrics to measure attributes (ISO/IEC 2502), focusing on the measurement and assessment of software attributes to ensure that they meet user needs and quality requirements. The Institute of Electrical and Electronics Engineers (IEEE) organization presents a quality assurance process and defines the software life cycle. The Object Management Group (OMG) organization presents metrics for software assurance (SwA). SwQ is a set of standards to measure the security, reliability, and resilience attributes of software systems throughout their life cycle. These metrics focus on identifying and mitigating risks related to software vulnerabilities to build reliable software products. The National Institute of Standards and Technology (NIST) provides a comprehensive set of security controls to ensure software quality through rigorous security measures. They also introduce quality metrics to assess software quality attributes such as performance, reliability, and maintainability to ensure that software products meet high standards of excellence. The American Productivity & Quality Center (APQC) provides a framework for organizations to improve their quality management processes through effective practices such as continuous improvement, process optimization, customer focus, and performance measurement, helping organizations to achieve quality.

Table 4 presents the tools for generating unit tests and measuring suitability and functional adequacy in projects built in Python and Java. In Python, PEP 8 is the style guide for Python code that provides conventions for readable and consistent code. It covers several aspects of coding style, including naming conventions, code layout, indentation, and documentation. By properly using this PEP 8 guide, you can produce more maintainable code, which facilitates better collaboration and reduces the likelihood of bugs in Python projects. Additionally, this table presents a comparison of various quality attributes and the tools used to assess these attributes in the Java and Python programming languages. Java tools

Table 3
Organizations, Standards, and Software Quality Attributes

Organization	Key Standards/Frameworks	Principal Attributes
ISO/IEC	ISO/IEC 25010 (Quality Model), ISO/IEC 9126 (Software Product Quality), ISO/IEC 27001 (Information Security)	Reliability, Security, Efficiency
IEEE	IEEE 730 (Software Quality Assurance), IEEE 829 (Testing Documentation), IEEE 12207 (Lifecycle Processes)	Testability, Maintainability, Portability
OMG	Software Assurance (SwA) Metrics	Security, Interoperability
NIST	NIST SP 800-53 (Security and Quality), NIST SP 500-299 (Quality Metrics)	Security, Reliability
SEI	Quality Attribute Workshop (QAW), SEI CERT (Security and Quality)	Security, Maintainability
ISO/IEC JTC 1/SC 7	ISO/IEC 25023 (Software Quality Measures), ISO/IEC 15939 (Software Measurement)	Reliability, Efficiency
APQC	Benchmarking Standards, Quality Management Best Practices	Efficiency, Effectiveness
ISO/IEC 27034	Guidelines for Application Security	Security, Usability

Table 4
Tools for Quality for Java and Python

Attribute	Java Tools	Python Tools
Functional Suitability	SonarQube, FindBugs, Checkstyle, PMD	Flake8, Pylint, SonarQube, Bandit
Performance Efficiency	JMeter, VisualVM, YourKit, SonarQube	SonarQube, Py-Spy, cProfile
Compatibility	SonarQube, Checkstyle, PMD	SonarQube, pylint, tox
Usability	SonarQube, Checkstyle	Flake8, Pylint, SonarQube, Black
Reliability	SonarQube, FindBugs, Checkstyle, PMD	SonarQube, Pylint, pytest, Coverage.py
Security	SonarQube, FindBugs, Checkstyle, PMD, OWASP Dependency-Check	SonarQube, Bandit, Safety, OWASP ZAP
Maintainability	SonarQube, FindBugs, Checkstyle, PMD	Flake8, Pylint, SonarQube, Radon
Portability	SonarQube	SonarQube, tox, pyenv

include SonarQube, VisualVM, YourKit, CheckStyle, FindBugs, PMD, OWASP Dependency-Check, and JMeter, while Python tools include SonarQube, Pytest, Radon, Tox, Pyenv, Py-Spy, CProfile, Flake8, Pylint, and Bandit. This comparison helps identify the tools available to maintain and improve software quality.

2.4. Adherence to OOP paradigm

To determine whether the software project complies with the object programming paradigm, the following can be reviewed:

Encapsulation

- **Python.** Class attributes are declared as private with a double underscore prefix to encapsulate them. Additionally, provide getter and setter methods to access and modify private attributes.

- **Java.** Use access modifiers (private, protected, public) to control access to class members. Encapsulate data fields by declaring them private and providing public getter and setter methods.

Inheritance

- **Python.** Use subclasses to create new classes that inherit attributes and methods from parent classes. Also, ensure that subclass methods override superclass methods when necessary.
- **Java.** Use the extends keyword to create subclasses that inherit from parent classes. Apply the @Override annotation when overriding superclass methods to ensure correct behavior.

Polymorphism

- **Python.** Use method overriding and overloading to achieve polymorphic behavior. Additionally, implement polymorphism through duck typing, where objects are judged based on their behavior rather than their type.
- **Java.** Achieve polymorphism through method overriding and overloading. Use inheritance and interfaces to define common behaviors and allow objects of different classes to be treated uniformly.

Abstraction

- **Python.** Use abstract base classes (ABC) from the abc module to define abstract methods and implement interfaces. Implement abstract classes that provide a model for subclasses to follow.
- **Java.** Define abstract classes and methods using the abstract keyword. Use interfaces to define contracts for classes and provide a form of multiple inheritance.

3. Related Work

In software engineering, metrics are crucial to assess various aspects of software quality. These metrics can be categorized according to their focus: some target runtime issues (code) [9], while others address design-related concerns or the implementation of object-oriented principles such as design patterns, SOLID principles, and core concepts such as encapsulation and inheritance [15].

The concept of code smells is popularized by influential authors such as Martin Fowler and Robert C. Martin, who refer to indicators of potential internal problems within software [13]. Code smells highlight code areas that may require refactoring to improve quality and maintainability. They point out specific issues and guide developers toward appropriate refactoring techniques.

Recent research indicates that current tools can detect many of the code smells defined by Fowler, but there is still a need for tools that support a wider range of programming languages. Tools such as DECOR/DETEX can identify various code smells using Domain-Specific Languages (DSL), but the number of detectable smells has been increasing to meet new software requirements.

Each code smell is associated with specific quality attributes, both internal and external. Understanding these relationships is essential because refactoring to address one code smell can sometimes lead to new smells, creating a cycle of software deterioration. This underscores the importance of exploring how refactoring affects code smells and overall quality [13].

Machine learning techniques are increasingly used to measure performance, usability, and security attributes, often focusing on runtime rather than design-time issues [16]. These techniques can learn from input data to identify problems related to software attributes, providing models that can produce specific problems with specific data.

The metrics related to maintainability are particularly significant due to their impact on development costs. Prioritizing metrics according to their relevance in industry and academia can help to organize these attributes by their importance and impact [17].

Several established metrics from companies such as McCabe, renowned for their pioneering contributions to software engineering in the 1990s, focus on various aspects of software analysis including security, quality, release management, and overall software process management. McCabe's metrics are designed to provide deep insights into the structure and behavior of software systems, helping in the identification of potential risks and areas for improvement. Metrics include:

- Cyclomatic Complexity (CC)
- Essential Complexity (EC)
- Module Design Complexity (MDC)
- Module Integration Complexity (MIC)

Chidamber and Kemerer [6] introduced MOOSE (Metrics for Object-Oriented Software Engineering), a comprehensive set of metrics designed to evaluate object-oriented software through various key traits such as inheritance, coupling, and cohesion. MOOSE provides a framework for assessing different aspects of object-oriented design, which are critical for understanding software quality and maintainability in the context of object-oriented programming. Metrics include:

- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling Between Object Classes (CBO)
- Response for a Class (RFC
- Lack of Cohesion of Methods (LCOM)

Lorenz and Kidd [8] proposed a set of metrics to evaluate different aspects of software design, with a focus on complexity, modularity, and maintainability. Their metrics are designed to provide insights into how well the software is structured and organized, helping to identify potential issues that could impact the software's effectiveness and ease of maintenance. Metrics include:

- Number of Scenario Scripts (NSS)
- Number of Key Classes (NKC)
- Average Number of Support Classes per Key Class (ANSCK)
- Number of Subsystems (NSUB)
- Class Size (CS)
- Class Inheritance (CI)
- Number of Methods (NOM)

Fernando Brito e Abreu [7] introduced MOOD (Metrics for Object-Oriented Design) to capture essential elements of object-oriented programming, focusing on key aspects such as inheritance, encapsulation, and coupling. These metrics aim to assess the quality of object-oriented design by evaluating how well these core principles are applied in the software. Metrics include:

- Method Hiding Factor (MHF)
- Attribute Hiding Factor (AHF)
- Method Inheritance Factor (MIF)
- Attribute Inheritance Factor (AIF)
- Polymorphism Factor (PF)
- Coupling Factor (CF)

According to related work, code smells and metrics are strongly related to quality attributes. The metrics mentioned above are primarily concerned with how each one can be measured. For example, McCabe suggests Cyclomatic Complexity, one of the easiest metrics to measure because it can be analyzed using regular expressions without requiring relationships, indicating lower coherence. In this study, we aim to explore these relationships and evaluate the feasibility of each metric, to better connect these concepts and provide more valuable feedback for practitioners and students.

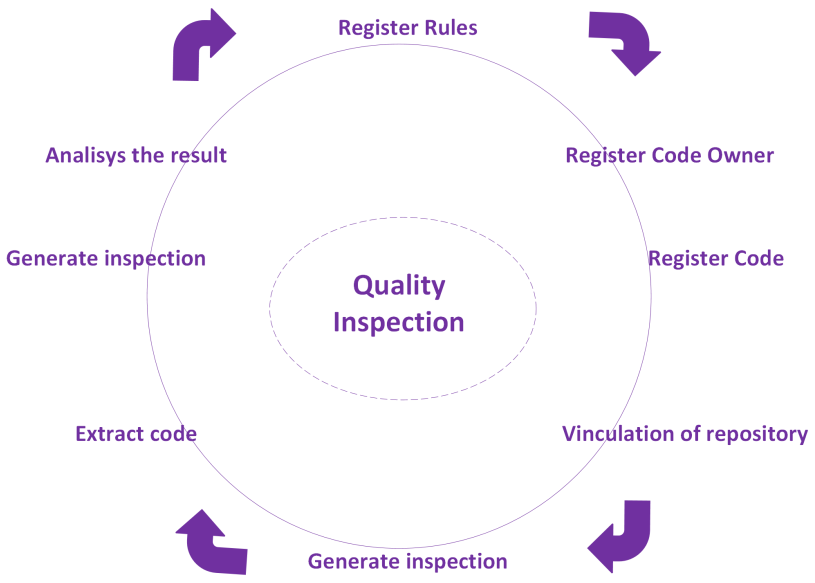

Figure 1: Flow for generate inspection

4. Proposed Analysis Model

4.1. Web System for generating inspections

Quality measurement in software development is the process of evaluating various attributes and user satisfaction against business needs. By systematically measuring these attributes, developers can identify defects, optimize performance, and ensure that software meets user expectations and business requirements. Effective quality measurement is crucial to delivering high-quality software, reducing development costs, and maintaining customer trust.

Quality Code System (QuUD) is the name of the system. QuUD will measure code quality following a circular flow to ensure code inspection. The system has two main roles: the admin generates and handles the rules, code owner is the users who generate inspections. Figure 1 presents the general process. The process starts with the administrators' configuration of quality rules. The administrators must establish specific standards and criteria to be applied to the Java and Python languages for each quality attribute. A weight will be defined for each attribute. Additionally, rules are established to determine if the project complies with the object-oriented programming paradigm.

Once the rules are defined, the project owners must register in the system and link their source code repositories. This step is done by connecting the code that must be on GitHub to automatically analyze the code. Initially, the repository must be public.

By linking the project, the project owners will be able to initiate quality inspections. By starting an inspection, the system must extract the code from the repositories and execute the analysis according to the predefined quality rules. The inspection process involves evaluating the code in terms of the attributes defined in Table 2 and checking if OOP paradigm is achieved based on 2.4. Once the inspection is complete, the system will send a push notification to the project owner with the results. The idea of the report is to provide a score for each attribute. Additionally, an overall score of the result will be displayed. Code owners can review the results to make corrections that improve the quality of the software. The steps ensures that the code is evaluated multiple times according to the established rules.

4.2. Quality measurement according to quality attributes and POO

To calculate the source code inspection score, a static scan of the code is performed to assess whether it complies with the OOP paradigm. Additionally, an evaluation of the attributes is performed. Rules are defined for each attribute and for the OOP paradigm. Thus, a weight is defined for each attribute and a weight to determine whether it complies with the OOP paradigm. Then, a score is generated for each attribute and for OOP. Then, a final score is calculated. To calculate the source code inspection score, follow this procedure:

1. Calculate the score of the quality attribute:

$$\text{Total Attribute Score} = \frac{\sum_{i=1}^{n}(R_i \times W_i)}{\sum_{i=1}^{n} W_i}$$

3. Total Paradigm Score Calculation:

$$\text{Total Paradigm Score} = \frac{\sum_{j=1}^{m}(P_j \times W_j)}{\sum_{j=1}^{m} W_j}$$

4. Calculating the Final Inspection Score:

$$\text{Final Score} = \alpha \times \text{Total Attribute Score} + (1 - \alpha) \times \text{Total Paradigm Score}$$

Where:

- α: It is a weighting factor that determines the relative importance between attributes and paradigm compliance (with $0 \leq \alpha \leq 1$).
- R_i: Score obtained for the attribute i.
- W_i: Weight assigned to the attribute i.
- P_j: Score obtained for compliance with the paradigm j.
- W_j: Weight assigned to the paradigm j.
- n: Total number of rules of quality attributes.
- m: Total number of rules to determine if it complies with the OOP programming paradigm.

4.3. Approach Architecture

QuUD is composed of backend and frontend components. Figure 2 shows the architecture of the system. For authentication, the Firebase Auth service will be used. This will allow for different authentication mechanisms such as Gmail, User-password, text message, etc. In the front, there are two main sub-components, the code-scanner will be in charge of allowing project owners to register, link their projects, and consult the results of the inspections. In the front, the other component is the Admin, which will mainly be in charge of managing the inspection rules. In the Backend, there are two databases. The first is the real-time database, which will be in charge of saving events about whether the inspections have finished or if an inspection should be started. The PostgreSQL database will save the results of the inspections. In the backend, there are three sub-components:

- **Project extractor:** It is responsible for connecting to the repositories and starting the scan. Also, this sub-component calculates if is a Java or Python project.
- **Event handler:** It mainly receives two events, scan start and scan end. When the scan starts the *Engine Scanner* sub-component receives the event. When the scan ends the *Web - Code Scanner* sub-component receives the event and queries the *Project Extractor* sub-component to display the final result.
- **Scanner engine:** This is the main component of the project, responsible for receiving the rules and the source code to start calculating the inspection result. The static code scanning will be focused on evaluating whether the OPP paradigm is well implemented. Additionally, each quality attribute will be evaluated.

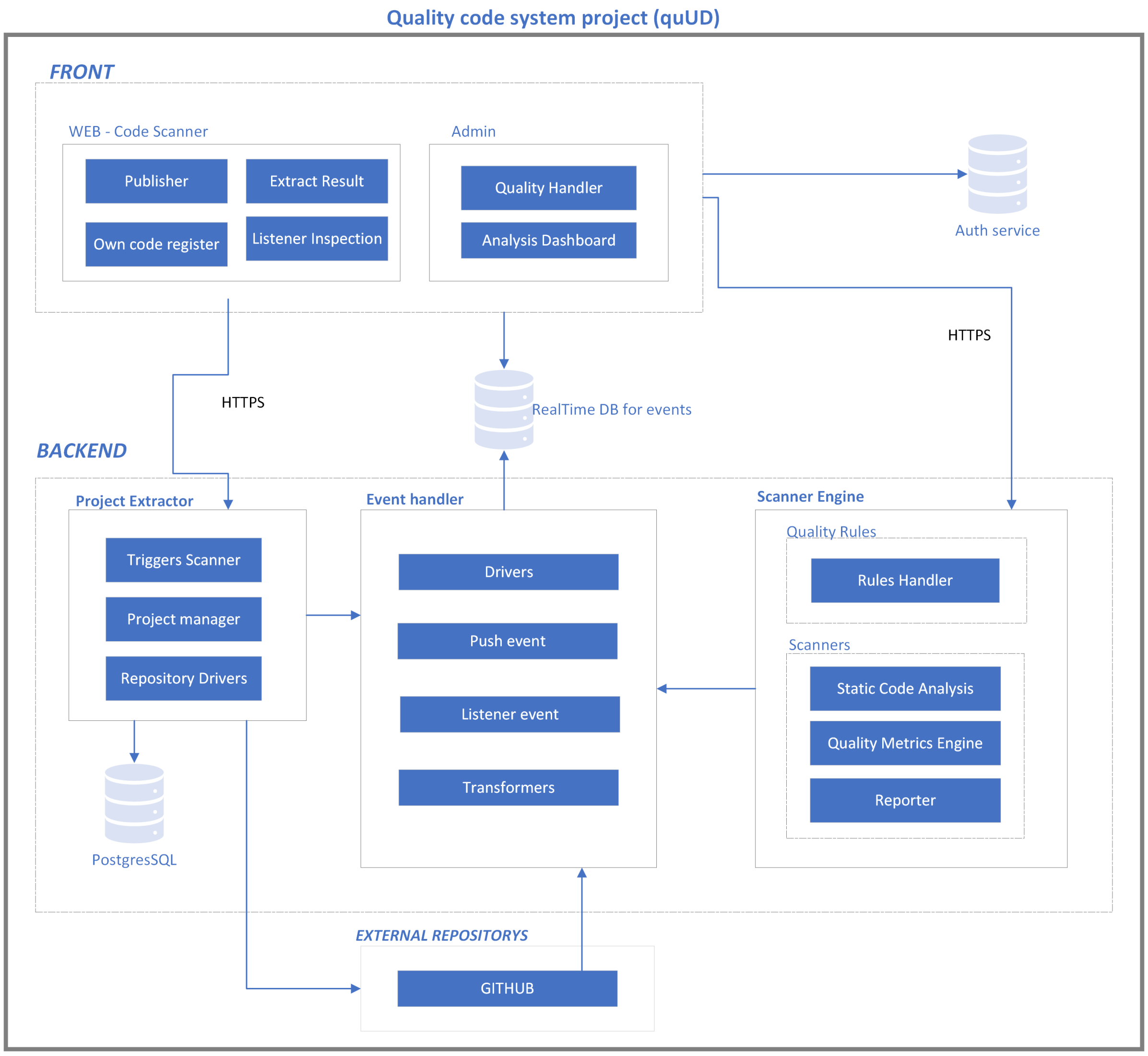

Figure 2: Architecture of the approach. The system needs several components to generate inspections in Python and Java projects.

5. Future Work

5.1. Focus on Performance and Scalability Metrics

Metrics focused on performance and scalability require different approaches. Some of them use machine learning techniques to learn what input causes stress in applications. Our model is focused on code, specifically for detecting issues through mapping strings, that can affect performance but not in the execution or from an experimental perspective. The detection of what can stress applications could help to detect what problems in code cause weakness in performance and scalability.

5.2. Application Development

The development of the model proposed lets us go to the next steps, more related to experiments in the focused area, of academics. We recommend the use of SCRUM and Agile methodology after defining product requirements. Also, it is necessary to take design and UI into the development process because, due to students and teachers it's important to make it easy to use. Another part is the infrastructure design, which requires services and tools to support events and preserve data. This definition includes the next steps after application design and it has to take requirements according to the amount of users in the academic area who are going to use the application for the first time.

5.3. Measure Other Programming Languages

Other programming languages, such as C++, Javascript, Golang, Rust, and Kotlin are also considered in this study as a future extension of our approach. The rise of event-driven architectures and the development of mobile and single-page applications have led to the creation of new languages that address emerging requirements, such as enhanced performance and management of advanced technologies like event-driven databases.

5.4. Adaptation to Other Domains

Other domains require a different approach, firstly we aim to support students and lecturers in the academic area, but it could be useful to support practitioners. The enterprise domain requires different communication, because the objective is different. A company focuses on revenue because the development has to be quick and sometimes is necessary to give more importance to some quality attributes at the beginning. For example, for the development of a new product, companies can use monoliths keeping coupling in mind for some features and they can go fast for the launch of applications, and with results they can validate if the product evolves or not.

5.5. Communication of Metrics

Effective communication of metrics is crucial to the evolution of our approach. However, this can be challenging, especially as the number of variables increases. It becomes more difficult to convey not only the metrics themselves but also the related quality attributes, both general and specific. It's important to enhance the evaluated application while simultaneously guiding users on how to address any identified issues.

5.6. Evaluation of the model

The idea is to generate a mechanism to evaluate whether the model is correct. In a subsequent article, different projects will be made as case studies. The idea is to check whether the scores are being calculated correctly according to the principle of the OOP paradigm (encapsulation, abstraction, inheritance, polymorphism, etc.) and whether the quality attribute is improved.

5.7. Relationship Between Code Smells and Metrics

Although there is not a clear relation between code smells and metrics, in the specific area of programming, code smells deteriorate metrics, and studying this relation could give us the capacity to provide better feedback, with accuracy because once we have this relation we can add refactoring techniques. According to this study [12] metrics and quality attributes have a relation at the same time metrics have with quality attributes as we can see, having this relation in mind we can translate into the relation of metrics and code smells.

6. Conclusions

This paper explains the different quality attributes that a software code can have. A system was proposed that will be able to control the quality of source code for Python and Java using a rules engine. The system must evaluate the quality attributes and whether it complies with the object-oriented paradigm.

We proposed a novel way of calculating the quality of the code by scoring it based on the quality attributes and whether it complies with the object-oriented paradigm. If it complies with the object-oriented paradigm, the software project can guarantee the sole responsibility of the SOLID principle.

Additionally, in an academic context, students will be able to know if they are using the principles of the paradigm correctly.

Software quality should not be neglected and, on the contrary, mechanisms or models must be in place to ensure quality. It is important to measure quality in terms of quality attributes because if you are in a software development phase, you can establish in the design phase which attributes the system should prioritize.

References

[1] B. Gr, R. Maksimchuk, M. Engel, B. Young, J. Conallen, K. Houston, Object-oriented analysis and design with applications, 2007.

[2] H. Florez, M. Sánchez, J. Villalobos, G. Vega, Coevolution assistance for enterprise architecture models, in: Proceedings of the 6th International Workshop on Models and Evolution, 2012, pp. 27–32.

[3] U. Erdemir, U. Tekin, F. Buzluca, E-quality: A graph based object oriented software quality visualization tool, in: 2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), IEEE, 2011, pp. 1–8.

[4] P. Gómez, M. Sánchez, H. Florez, J. Villalobos, Co-creation of models and metamodels for enterprise architecture projects, in: Proceedings of the 2012 Extreme Modeling Workshop, 2012, pp. 21–26.

[5] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[6] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (1994) 476–493.

[7] F. B. Abreu, M. Goulão, R. Esteves, Toward the design quality evaluation of object-oriented software systems, in: Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, 1995, pp. 44–57.

[8] M. Lorenz, J. Kidd, Object-oriented software metrics: a practical guide, Prentice-Hall, Inc., 1994.

[9] M. Grechanik, C. Fu, Q. Xie, Automatically finding performance problems with feedback-directed learning software testing, in: 2012 34th international conference on software engineering (ICSE), IEEE, 2012, pp. 156–166.

[10] M. Stegeman, E. Barendsen, S. Smetsers, Designing a rubric for feedback on code quality in programming courses, in: Proceedings of the 16th Koli Calling International Conference on Computing Education Research, 2016, pp. 160–164.

[11] H. Florez, M. Sanchez, J. Villalobos, Extensible model-based approach for supporting automatic enterprise analysis, in: 2014 IEEE 18th international enterprise distributed object computing conference, IEEE, 2014, pp. 32–41.

[12] A. Kaur, A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes, Archives of Computational Methods in Engineering 27 (2020) 1267–1296.

[13] G. Lacerda, F. Petrillo, M. Pimenta, Y. G. Guéhéneuc, Code smells and refactoring: A tertiary systematic review of challenges and observations, Journal of Systems and Software 167 (2020) 110610.

[14] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transactions on software engineering 28 (2002) 4–17.

[15] G. Travassos, F. Shull, M. Fredericks, V. R. Basili, Detecting defects in object-oriented designs: using reading techniques to increase software quality, ACM sigplan notices 34 (1999) 47–56.

[16] D. Himali, S. Kodithuwakku, Object-Oriented Software Quality Metrics, Technical Report, University of Peradeniya, Sri Lanka, 2005.

[17] L. Ardito, R. Coppola, L. Barbato, D. Verga, A tool-based perspective on software code maintainability metrics: A systematic literature review, Scientific Programming 2020 (2020) 8840389.