Tuijin Jishu/Journal of Propulsion Technology
ISSN:1001-4055
Vol. 44No. 4 (2023)

A Comprehensive Study on Software Maintainability:Evolution,Practices, and Contemporary Implications

[1]Md Touhidul Islam, [2]Anupam Das, [3]Mohammad Shahriar Syeed, [4]MihoriMoasir Riza, [5]Hasan Sarwar

[12345]Department of Computer Science & Engineering, United International University, Dhaka, Bangladesh

Abstract—Software quality management is a critical concern in the field of software engineering, aimed at ensuring the delivery of reliable and maintainable software systems. Among the various software quality attributes, "maintainability" holds a pivotal role as it determines the ease with which a software system can be modified, extended, and fixed throughout its lifecycle. This research paper delves into the multifaceted dimensions of software quality attribute "maintainability," exploring its history, relation in SQM five model, metrics used for evaluating quality, market standard of maintainability, cost of quality of maintainability, how it is addressed in the software life cycle (waterfall/agile), how it is addressed in project management tools and implications on cloud computing. Through extensive analysis, and studying the comprehensive reviews of existing literature, this paper sheds light on the best practices available in the market associated with maintaining software systems over time. By investigating real-world scenarios and drawing insights from industry standards, the paper provides valuable guidance for software practitioners and researchers alike to enhance software maintainability and contribute to the overall quality of software products.

Index Terms— *software quality management, software quality attributes, SQM five model, metrics for maintainability*

Introduction

In today's rapidly evolving technological landscape, software systems play an indispensable role in enabling businesses, industries, and societies to thrive. As software continues to pervade every facet of modern life, ensuring the quality of software products has become paramount. Software quality encompasses a diverse array of attributes, each influencing the overall effectiveness and reliability of the software. Among these attributes, "maintainability" stands as a cornerstone, addressing the long-term sustainability and adaptability of software systems.

Maintainability goes beyond the initial development phase; it encompasses the entire software lifecycle, from conception to retirement. A software system's maintainability determines how well it can accommodate changes, updates, and enhancements as requirements evolve or unforeseen challenges arise. Effective software maintenance mitigates the risk of stagnation, technical debt, and obsolescence, enabling organizations to respond swiftly to changing market demands and emerging technologies.

The significance of software maintainability is underscored by its direct impact on cost, time, and resources. Software systems that are difficult to maintain can lead to prolonged development cycles, increased error rates, and diminished user satisfaction[1]. Conversely, highly maintainable software systems empower development teams to implement changes efficiently, minimize downtime, and optimize the allocation of resources. This not only bolsters a company's competitive edge but also enhances the user experience and fosters innovation.

This research paper points out how impactful maintainability can be on real time software development and quality approach. Starting from the history up to modern cloud computing technology, this paper explains the best practices of Software quality attribute "Maintainability".

I. History of Software Maintainability

Software maintainability is one of the essential software quality attributes that refers to the ease with which a software system can be modified or enhanced over its lifecycle. It involves making the codebase understandable, well-organized, and adaptable to changes without introducing errors or negatively impacting the system's overall functionality. The history of software maintainability can be traced back to the early days of software development. Let’s list it down its evolution.

Early Software Development (1950s-1960s):

During the initial stages of software development, maintainability was not a significant concern, as programming was primarily done by a single individual or a small team. The focus was on making the code work, and there was little consideration for making it maintainable[2].

Emergence of Software Engineering (1960s-1970s):

As software systems became more complex and organizations started developing large-scale applications, the need for maintainable code became evident. Researchers and practitioners began to recognize the importance of structuring code in a way that facilitated easier maintenance[3].

Structured Programming (late 1960s-1970s):

Structured programming emerged as a programming paradigm that emphasized dividing code into smaller, manageable functions or procedures. This approach improved code readability and made it easier to maintain and modify programs. The concept of modular programming helped in isolating changes, reducing the risk of introducing errors in the system.

Software Crisis (1970s):

During the software crisis of the 1970s, when many software projects faced significant challenges in terms of delays, cost overruns, and poor quality, the importance of maintainability became more evident. The lack of maintainable code made it difficult to adapt to changing requirements and resulted in increased maintenance costs.

Software Engineering Principles (1980s-1990s):

The field of software engineering began to formalize principles and best practices for writing maintainable code. Encapsulation, abstraction, information hiding, and modularity became key concepts to enhance code maintainability. The adoption of design patterns and object-oriented programming further improved software maintainability[4].

Quality Models and Standards (1980s-1990s):

Quality models and standards, such as ISO 9126, emerged during this period. These models identified maintainability as one of the essential software quality attributes and provided guidelines for evaluating and improving it[5].

Agile Software Development (2000s):

The rise of agile methodologies in the 2000s further emphasized the importance of maintainability. Agile practices, such as continuous integration, test-driven development, and refactoring, promote maintainable code by encouraging frequent code reviews, automated testing, and iterative development[6].

DevOps and Continuous Delivery (2010s): The DevOps movement brought together development and operations teams to work collaboratively on software projects. The emphasis on continuous delivery and automation led to a focus on maintainable code, as frequent updates and deployments require code that can be easily modified and tested[7].

SoftwareasaService(SaaS)andCloud Computing (2010s):

With the advent of cloud computing and SaaS models, software providers needed to maintain and update their applications more frequently. Maintainability became crucial to ensure seamless updates and improvements without disrupting service for users.

Maintainability In Five SQM Models

Maintainability is a crucial aspect of software quality, and it is addressed differently in various software quality models and frameworks. The relation of maintainability in five well-known software quality models[8] is given below:

McCall's Quality Model:

McCall's Software Quality Model, proposed by John McCall and his colleagues in 1977, indeed includes maintainability as one of the main characteristics. The model identifies 11 specific software quality factors, which are grouped under three main categories: product operation, product revision, and product transition. In McCall's model, "maintainability" is considered a part of the "Product Revision" category, along with other factors such as flexibility, testability, and adaptability.

Dromey's Quality Model:

Richard Dromey's software quality model emphasizes five primary software quality attributes, including "maintainability." In Dromey's model, maintainability is defined by three key sub-attributes:

Comprehensibility: This sub-attribute refers to the ease with which the code and documentation can be understood by developers and maintainers. Code that is well-organized, well-documented, and follows established coding conventions is more comprehensible and therefore more maintainable.

Adaptability: Adaptability focuses on the ease with which the software can be modified or adapted to accommodate changes in requirements, technology, or the environment. Maintainable software is designed to facilitate modifications without introducing unintended side effects or breaking the existing functionality.

Accommodability: Accommodability relates to the software's ability to accommodate changes without adversely affecting its overall integrity and functionality. A maintainable system is structured in a way that allows for changes to be made to specific components without causing widespread disruptions.

Dromey's model recognizes the importance of making software easy to understand, modify, and adapt over time, which aligns well with the concept of software maintainability. By addressing comprehensibility, adaptability, and accommodability, Dromey's model provides a comprehensive framework for assessing and enhancing the maintainability of software systems.

Boehm's Quality Model:

Barry Boehm's Quality Model, also known as the Boehm's Quality Model Spiral, is a comprehensive software quality framework. It includes a set of key quality attributes, but it does not explicitly mention maintainability as a standalone attribute. However, maintainability is related to aspects like changeability and flexibility, which are considered essential for accommodating modifications and changes in the software.

FURPS+:

FURPS+ (Functionality, Usability, Reliability, Performance, Supportability) is a software requirements classification model that includes a set of critical software qualities. Within the "Supportability" aspect of FURPS+, maintainability is addressed. Supportability emphasizes the ease of maintenance and enhancement over the software's lifecycle.

ISO 9126 (and its successor, ISO/IEC 25010):

ISO 9126 is an international standard for software quality evaluation. In ISO 9126, maintainability is explicitly recognized as one of the main characteristics, along with functionality, reliability, usability, efficiency, and portability. ISO 9126 defines maintainability in terms of sub-characteristics like modularity, reusability, analyzability, and modifiability, which align with the principles of easy maintenance, code understandability, and adaptability.

ISO/IEC 25010 is the successor to ISO 9126 and provides a more detailed and updated definition of software quality characteristics. In ISO/IEC 25010, maintainability is retained as one of the main characteristics, and its sub-characteristics are expanded to include the concepts of scalability and testability.

Metrics Uses For Maintainability

Maintainability is a crucial software quality attribute that focuses on the ease of modifying, extending, and fixing a software system. Evaluating software maintainability is challenging since there is no singular metric or tool that can definitively compare the maintainability of different applications accurately. Although human reviewers are essential, they cannot thoroughly analyze entire code repositories to provide a conclusive answer. Therefore, some level of automation is necessary. Several metrics are used to assess and measure the maintainability of software.

Cyclomatic Complexity (CC): Cyclomatic Complexity is a metric that quantifies the complexity of a program by counting the number of linearly independent paths through the code. Each decision point (such as an if statement or loop) contributes to the complexity. High CC can indicate increased code complexity, making it harder to understand, test, and maintain. Maintaining low cyclomatic complexity is desirable to ensure that code remains comprehensible and adaptable[9].

Maintainability Index (MI): The Maintainability Index is a composite metric that takes into account various factors, including cyclomatic complexity, lines of code, and code documentation. It provides a single score that represents the maintainability of the codebase. A higher MI score indicates better maintainability. This metric is valuable for comparing different codebases and identifying areas that may require improvement to enhance maintainability[10].

Code Duplication: Code duplication occurs when similar or identical code segments appear in multiple places within the codebase. High levels of code duplication can lead to inconsistencies, increase the effort required for maintenance, and introduce a higher risk of introducing bugs during changes. Identifying and reducing code duplication contributes to improved maintainability[11].

Dependency Metrics (Fan-in and Fan-out): Fan-in measures how many modules depend on a particular module, while fan-out measures how many modules a particular module depends on. High fan-in can indicate that a module is widely used and potentially critical. High fan-out can indicate a module with many dependencies, which might result in increased complexity. Monitoring and managing module dependencies helps maintain a clear and modular structure[12].

Depth of Inheritance Tree (DIT): DIT measures the number of levels in the inheritance hierarchy for a class. A deep inheritance tree can increase complexity and make it more challenging to understand and modify the code. Limiting the depth of the inheritance tree can contribute to maintainability by reducing the impact of changes across a complex hierarchy[13].

Class Coupling: Class coupling measures how many other classes a class is directly dependent on. High class coupling can result in tight interconnections between classes, making changes to one class affect others. Low coupling, on the other hand, promotes modularity and separation of concerns, which can lead to improved maintainability[14].

Response for a Class (RFC): RFC counts the number of methods in a class plus the number of methods that can be called from outside the class. High RFC indicates that the class has many responsibilities and interactions, which can lead to increased complexity and maintenance challenges. Limiting the RFC of a class can help improve maintainability by promoting focused and manageable classes[15].

Documentation Coverage: Adequate documentation, including comments, API references, and user guides, improves maintainability by providing insights into the code's purpose and behavior. Monitoring the documentation coverage helps ensure that future maintainers can understand and work with the codebase effectively[16].

Bug Fix Rate: The rate at which bugs are discovered and fixed reflects the software's stability and the ease of maintaining and improving it over time. A high bug fix rate may indicate ongoing maintenance challenges that need attention to enhance the software's maintainability[17].

Market Standard Of Maintainability

Specific market standards for software maintainability may vary based on the nature of the software, the industry, and the organization's goals. While there is no one-size-fits-all market standard for software quality attribute maintainability, industry best practices and guidelines have emerged over time. These standards help organizations ensure that their software systems are maintainable, adaptable, and of high quality.

Industry Standards and Frameworks: Various industries and domains have established standards and frameworks that include guidelines for software quality attributes, including maintainability. For example, ISO/IEC 25010 [18] (formerly ISO/IEC 9126) provides a comprehensive framework for software quality characteristics, including maintainability. In the context of specific industries like medical devices or automotive, there may be additional regulatory standards that influence maintainability requirements.

Software Development Methodologies: Agile methodologies, DevOps practices, and Continuous Integration/Continuous Deployment (CI/CD) pipelines have become industry standards for software development. These methodologies emphasize iterative development, automated testing, and continuous improvement, all of which contribute to better maintainability.

Code Review and Static Analysis: Industry standards often advocate for code reviews and static code analysis tools to identify and address maintainability issues early in the development process. Code reviews promote best practices, code clarity, and adherence to coding standards.

Automated Testing: High test coverage, including unit tests, integration tests, and regression tests, is a standard practice for ensuring software quality and maintainability. Automated testing helps catch defects early and provides a safety net for making changes without introducing regressions.

Documentation: Clear and comprehensive documentation, including code comments, API documentation, and user guides, is a standard practice for enhancing maintainability by aiding developers' understanding of the codebase.

Refactoring: Regular refactoring, supported by automated tests, is a standard technique for improving code quality and maintainability over time. Addressing technical debt through refactoring helps prevent the accumulation of legacy code.

Change Management and Version Control: Version control systems and change management practices are industry standards for tracking changes, enabling collaboration, and ensuring that software versions are properly managed.

Continuous Monitoring and Improvement: Ongoing monitoring of software metrics, bug tracking, and performance monitoring are standard practices to proactively identify and address maintainability issues as a software system evolves.

Cost Of Quality Of Maintainability

The expenses associated with software maintenance arise from making changes to an application, either to support new use cases or update existing ones, as well as ongoing bug fixing after deployment. Maintenance costs alone can account for a substantial portion of the Total Ownership Cost (TCO) of the software, reaching as high as 70-80%!

Software quality cost for maintainability can be classified with these four categories [19].

Corrective maintenance-these costs arise from modifying the software to address issues discovered after its initial deployment, typically constituting around 20% of the software maintenance expenses.

Adaptive maintenance- these costs result from modifying the software solution to ensure its effectiveness in a dynamically changing business environment, accounting for approximately 60% of the software maintenance costs.

Perfective maintenance- these costs arise from improving or enhancing the software solution to enhance its overall performance, generally making up around 3% of the software maintenance costs.

User Support-these costs are associated with response to user demands other than adaptive,corrective, perfective, typically comprising 17% or more of the software maintenance expenses.

How It Is Addressed In Software Life Cycle
Maintainability in Waterfall Approach

In the traditional Waterfall software development model, maintainability remains an important software quality attribute, even though the emphasis on addressing maintainability-related concerns may differ compared to more iterative and agile approaches. During the requirements phase, it's important to ensure that the software requirements are well-defined, complete, and clear. Ambiguous or incomplete requirements can lead to challenges in future maintenance efforts. Thoroughly reviewing and refining requirements with stakeholders can contribute to a more maintainable software system. The Waterfall model places a strong emphasis on design before development begins. During this phase, attention is given to creating a solid architectural design that is modular, well-structured, and follows established design principles. Design decisions should aim to reduce dependencies, minimize complexity, and promote code reusability, which are all factors that influence maintainability.

Then in the coding phase it typically follows the design phase. Maintaining code clarity, following coding standards, and applying appropriate comments and documentation are crucial aspects to ensure future maintainability. While Testing is essential in any software development process, Comprehensive testing, including unit testing, integration testing, and system testing, helps ensure that the software is functioning correctly and minimizes the risk of defects that could impact maintainability. Properly tested code reduces the likelihood of introducing regressions during future maintenance. Documentation is a key element throughout the entire process. Comprehensive documentation, including design documents, user manuals, and technical specifications, aids future maintainers in understanding the software's functionality and design decisions.

In the Waterfall model, changes to the software are typically handled through formal change management processes. Requests for changes are evaluated and incorporated into future releases. Maintaining detailed records of changes and their impact helps ensure that future maintenance efforts are well-informed. And finally, Version control systems play a role in managing the codebase's history and ensuring that changes are tracked and recorded. Version control helps in maintaining different versions of the software, making it easier to revert changes or apply patches when needed.

Maintainability in Agile Approach

In the Agile software development model, maintainability remains a fundamental and important software quality attribute. Agile methodologies, with their iterative and incremental approach, place a strong emphasis on delivering working software frequently while continuously addressing maintainability concerns. Agile teams create user stories and add them to the product backlog. During backlog refinement sessions, the team discusses, clarifies, and breaks down user stories into smaller tasks. Maintaining a well-organized and groomed backlog ensures that future development efforts can be planned effectively, minimizing the introduction of technical debt. Normally Agile projects are organized into iterations (sprints), each lasting a fixed duration (e.g., 1-4 weeks). During each iteration, the team focuses on delivering a potentially shippable increment of the software. Maintaining a regular cadence of iterative development allows for continuous improvement, adjustments, and the opportunity to address maintainability issues in smaller, manageable increments.

Teams often employ CI/CD practices to ensure that code changes are frequently integrated and tested. Automated testing, code reviews, and rapid integration help identify and address defects and maintainability concerns early in the development process and promotes ongoing refactoring, which involves improving the internal structure of the code without changing its external behavior. This practice helps keep the codebase clean, modular, and maintainable over time.

Test-Driven Developmentis a practice where tests are written before the code is implemented. This approach helps ensure that the software is designed with testability in mind, and it contributes to maintainability by fostering a culture of automated testing and preventing the introduction of defects. Regular code reviews are a standard practice in Agile. Peer reviews help catch coding errors, ensure adherence to coding standards, and promote discussions about maintainability and design decisions. After each iteration, Agile teams hold sprint retrospectives to reflect on what went well and what could be improved. Addressing maintainability concerns and technical debt is a common topic during retrospectives to continuously enhance the software's maintainability.

Agile values working software over comprehensive documentation, but essential documentation, such as user stories, design decisions, and architectural diagrams, should be maintained to aid understanding and future maintenance efforts. Agile methodologies emphasize adapting to changing requirements. This approach helps maintain software relevance and enables the team to adjust based on evolving user needs and technological advancements.

How it is addressed in project management tools

There are various software quality management tools and practices that can help ensure and improve software maintainability throughout a project. These tools aid in monitoring, analyzing, and addressing factors that contribute to maintainability. Below some specific and must have tools names are mentioned.

Static Code Analysis Tools: These tools analyze the source code without executing it and identify potential issues related to code quality, including maintainability. Examples include SonarQube, Checkstyle, and ESLint. These tools can detect code smells, complex code, and other maintainability-related concerns.

Code Review Tools: Collaborative code review platforms like GitHub, GitLab, and Bitbucket provide features for code reviews, which help identify maintainability issues through peer feedback and discussions. Code reviews can highlight areas of the codebase that might require refactoring or improvements.

Unit Testing Frameworks: Writing comprehensive unit tests using frameworks like JUnit (Java), NUnit (.NET), or pytest (Python) ensures that changes or updates do not negatively impact the existing functionality. Strong test coverage contributes to the maintainability of the codebase.

Continuous Integration (CI) Tools: CI tools like Jenkins, Travis CI, and CircleCI automate the building and testing of the software whenever changes are made to the codebase. Automated testing helps catch regressions early and maintains the overall quality of the codebase.

Code Metrics Tools: Tools like CodeClimate and Understand analyze code metrics, including cyclomatic complexity, code duplication, and code churn. These metrics provide insights into areas of the codebase that may need attention for improved maintainability.

Dependency Management Tools: Dependency management tools like Maven (Java), npm (JavaScript), and pip (Python) help track and manage software dependencies. Keeping dependencies up-to-date and well-maintained is essential for long-term maintainability.

Documentation Tools: Tools like Doxygen, Javadoc, and Sphinx generate documentation from inline comments in the code. Well-maintained documentation aids in understanding the codebase and contributes to maintainability.

Refactoring Tools: Integrated development environments (IDEs) often include refactoring capabilities that help automate code restructuring. IDEs like IntelliJ IDEA, Eclipse, and Visual Studio provide features to assist in safely improving code maintainability.

Version Control Systems: Version control systems like Git help manage changes to the codebase over time. Proper branching and versioning strategies facilitate maintenance efforts and allow for safe experimentation.

Issue Tracking and Bug Reporting Tools: Tools like JIRA, Trello, and Bugzilla help track and manage issues, including maintainability-related tasks and improvements. Issues related to code quality and maintainability can be prioritized and addressed.

Code Linters: Linters such as ESLint, Pylint, and RuboCop provide automated checks for coding standards and style guidelines, which contribute to consistent and maintainable code.

Automated Documentation Generators: Tools like Swagger (for APIs) and JSDoc (for JavaScript) help generate API documentation from code annotations, promoting clear and up-to-date documentation for maintainability.

It's important to integrate these tools and practices into your software development process and tailor them to your specific project's needs. Combining multiple tools and approaches helps create a comprehensive strategy for ensuring software maintainability throughout the project's lifecycle.

Implication On Cloud Computing

Cloud computing has significant implications on the software quality attribute of "maintainability." Incorporating cloud computing into software development requires a careful balance between the benefits and challenges it brings. Proper architectural design, adherence to best practices, and ongoing monitoring are essential to ensuring that cloud-based software systems remain maintainable, adaptable, and of high quality over time.

Cloud platforms allow software systems to scale up or down based on demand. This can enhance maintainability by providing the ability to handle increased workloads without significant code changes. However, proper design and architecture are crucial to fully realize this benefit. Cloud environments facilitate automated deployment and continuous integration/continuous deployment (CI/CD) pipelines. These practices improve maintainability by enabling rapid and controlled updates, reducing the complexity of managing software versions. Cloud services often handle resource provisioning, allocation, and management. This can simplify maintenance tasks related to hardware and infrastructure management and outsources hardware maintenance and data center operations, allowing developers to focus more on the software itself rather than the underlying infrastructure. This kind of platform enables remote collaboration, which can enhance maintainability by fostering efficient communication and knowledge sharing among distributed development teams[20].

While cloud computing offers various benefits, it also introduces specific challengesand considerations related to maintaining software systems. Depending on specific cloud services and APIs can lead to vendor lock-in, potentially making future maintenance and porting efforts more challenging. Proper data management strategies are essential in the cloud, as data is often distributed and accessed remotely. Poor data management can negatively impact maintainability, data integrity, and performance. Cloud services might introduce latency and performance challenges. Designing for optimal performance and monitoring can impact the software's maintainability. Cloud-based systems rely heavily on network connectivity. Anticipating and addressing network issues is important for maintaining system availability and reliability.

Cloud systems also often involve distributed architectures, microservices, and containerization. Proper design and orchestration are required to manage this complexity effectively. Costs can escalate if not managed carefully. Uncontrolled costs can impact long-term maintainability. Robust monitoring and logging practices are necessary to quickly identify and address issues in cloud environments, contributing to maintainability.

Conclusion

As software development continues to evolve, maintainability remains a critical concern. Advances in artificial intelligence, machine learning, and automated code generation may impact how maintainability is addressed in the future. However, the fundamental goal of creating software that can be easily modified and extended will remain constant. The concept of software maintainability has evolved over time in response to the growing complexity of software systems and the need for cost-effective and efficient maintenance. As software becomes an increasingly integral part of modern life, maintaining and improving software systems will continue to be a vital aspect of the software development process.

References

[1] https://www.researchgate.net/publication/308362596_User_Satisfaction_and_System_Success_An_Empirical_Exploration_of_User_Involvement_in_Software_Development

[2] T. Haigh, "Software in the 1960s as concept, service, and product," in IEEE Annals of the History of Computing, vol. 24, no. 1, pp. 5-13, Jan.-March 2002, doi: 10.1109/85.988574.

[3] Aoyama, Mikio. "Beyond software factories: concurrent-development process and an evolution of software process technology in Japan." Information and Software Technology 38.3 (1996): 133-143.

[4] Booch, Grady. Object oriented design with applications. Benjamin-Cummings Publishing Co., Inc., 1990.

[5] Fitzpatrick, Ronan. "Software quality: definitions and strategic issues." (1996).

[6] Hoda, Rashina, Norsaremah Salleh, and John Grundy. "The rise and evolution of agile software development." IEEE software 35.5 (2018): 58-63.

[7] Beattie, Tim, et al. DevOps Culture and Practice with OpenShift: Deliver continuous business value through people, processes, and technology. Packt Publishing Ltd, 2021.

[8] https://www.researchgate.net/publication/228991952_Quality_Models_in_Software_Engineering_Literature_An_Analytical_and_Comparative_Study

[9] Ebert, Christof, et al. "Cyclomatic complexity." IEEE software 33.6 (2016): 27-29.

[10] Ganpati, Anita, Arvind Kalia, and Hardeep Singh. "A comparative study of maintainability index of open source software." Int. J. Emerg. Technol. Adv. Eng 2.10 (2012): 228-230.

[11] Chen, Chang-Feng, AzlanMohd Zain, and Kai-Qing Zhou. "Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review." Neural Computing and Applications 34.23 (2022): 20507-20537.

[12] Murgia, Alessandro, et al. "Refactoring and its relationship with fan-in and fan-out: An empirical study." 2012 16th European Conference on Software Maintenance and Reengineering. IEEE, 2012.

[13] Prykhodko, Sergiy, Natalia Prykhodko, and TetyanaSmykodub. "A statistical evaluation of the depth of inheritance tree metric for open-source applications developed in Java." Foundations of Computing and Decision Sciences 46.2 (2021): 159-172.

[14] Souley, Boukari, and Baba Bata. "A class coupling analyzer for Java programs." West African Journal of Industrial and Academic Research 7.1 (2013): 3-13.

[15] Prykhodko, Sergiy, and Natalia Prykhodko. "A Technique for Detecting Software Quality Based on the Confidence and Prediction Intervals of Nonlinear Regression for RFC Metric." 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT). IEEE, 2022.

[16] Van der Meij, Hans. "The role and design of screen images in software documentation." Journal of computer assisted learning 16.4 (2000): 294-306.

[17] Zou, Weiqin, et al. "An empirical study of bug fixing rate." 2015 IEEE 39th Annual Computer Software and Applications Conference. Vol. 2. IEEE, 2015.

[18] Estdale, John, and Elli Georgiadou. "Applying the ISO/IEC 25010 quality models to software product." Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25. Springer International Publishing, 2018.

[19] https://www.researchgate.net/publication/228535109_Software_maintenance_productivity_and_maturity

[20] Viegas, Eduardo, et al. "Enhancing service maintainability by monitoring and auditing SLA in cloud computing." Cluster Computing 24 (2021): 1659-1674.