Open Access ISSN: 3007-1437 (Online), ISSN: 3007-1429 (Print)

Alkadhim Journal for Computer Science (KJCS)

Journal Homepage: https://alkadhum-col.edu.iq/JKCEAS

Practical Implementation of Software Metrics to improve Maintainability in Open-Source Systems

[1]Nadia Mahmood Hussien *

[1]Department of Computer Science, College of Science, Mustansiriyah University, Baghdad – Iraq

Article information

Article history:
Received: November, 14, 2025
Accepted: December, 16, 2025
Available online: December,25, 2025

Keywords:
Lines of Code (LOC), Cyclomatic Complexity, Open-Source Software.

Corresponding Author:
Nadia Mahmood Hussien
nadia.cs89@uomustansiriyah.edu.iq

DOI:
https://doi.org/10.61710/kjcs.v3i4.135

Abstract

Common software metrics and maintainability measures in open-source Java are examined in this research. The authors test the major object-oriented metrics—Coupling Between Objects (CBO), Lines of Code (LOC), Weighted Methods per Class (WMC), Lack of Cohesion of Methods (LCOM), Depth of Inheritance Tree (DIT), and Cyclomatic Complexity—against real-world maintainability indicators like bug counts, code modifications, and developer turnover. There is currently no workable, repeatable, and experimentally verified process that combines measurement analysis with actual maintainability indications in freely available systems, although a wealth of research on software metrics. The authors use Python data analysis and visualization to find statistically significant patterns in Spearman correlation analysis.

The findings indicate that the strongest predictors of maintainability concerns are Connectivity Among Entities ($\rho = 0.82$) and cyclic Structure ($\rho = 0.77$), subsequent to WMC ($\rho = 0.70$) and LOC ($\rho = 0.65$). The smallest and insufficient predictor of flaw frequency is DIT ($\rho = 0.30$). Finally, this study ties the gap of theoretical academic measures and the real-world aspects of software engineering.

1) Introduction

Maintainability of the software has become one of the most important quality of current software engineering and has a direct influence on the life span, affordability, and alteration resiliency of the system [1]. Software systems have grown in scale remarkably, and with that growth, it is becoming increasingly difficult to guarantee that they are maintainable by having clean, maintainable code, requiring quantitative tools, which determine the degree of software ssustainability at an early stage of the development lifecycle [2]. Object-oriented design measures have entered this field, providing quantitative measures of the quality of code by measuring structural and behavioral properties of code [3].

The Chidamber and Kemerer (CK) metric suite, i.e., Coupling Between Objects (CBO), Weighted Methods per Class (WMC), Lack of Cohesion of Methods (LCOM), and Depth of Inheritance Tree (DIT), are just a few of

them that have been confirmed time and again on their ability to predict defect-proneness and maintenance [4, 5]. Recent efforts incorporate cyclomatic complexity to quantify the complexity of the control flow with modifiability modeling being extended even further [6].
although research has driven the development of metric theory and prediction models, in practice, industrial projects in the real world omit the systematic use of metrics because of the hard work of integrating the tools and the unavailability of data and the inability to interpret findings [7]. Although open-source software (OSS) ecosystems offer abundant empirical evidence on which to base validation, there is a paucity of workflows that practitioners may use on a daily basis, which restricts their influence on the practice of software [8,9].
This research is focused on creating a clear, reproducible and practical pipeline, which relates object-oriented metrics to usable supportability knowledge about open-source Java projects. Our study provides an automated hands-on process of metric extraction, maintainability indicator gathering, and Spearman statistical analysis. Combining the quality of the academic and the value of the practical are interpretable visuals and actionable insights instead of numerical correlations. The proposed technique is verified with the help of the real open-source software project to create a customizable structure of academic and industry research. This work is applicable to the software quality assurance, CI/CD pipeline integration, technical debt management, software engineering education, and design of the tool of code analysis. Outline of the paper Section 2 refers to the relevant work and the gap in the research; Section 3 elaborates on the methodology, including project selection, metric extraction, and statistical analysis; Section 4 comments on the results and discusses them; and Section 6 concludes with major conclusions and perspectives.

2) Related Work and Research Gap

Over the last several years, the software engineering community has become very interested in seeking answers as to the practical implications of object-oriented metrics to software maintainability, both due to the growth of open-source repositories and due to new possibilities presented by data mining. Lee et al. (2023) [11] have also given another example, where an automated combination of a static code analysis and issue tracker was established to measure maintainability of a Java project and reported that the results of the combination are highly correlated with the traditional measurements and maintenance effort. Similarly, Park and Kim (2022) [12] examined the characteristics of software change in metrics over time to show that the dynamics of change with time result in maintainability and defect inclination issues in agile environments.
Moreover, Zhang et al. (2024) [13] employed hybrid machine learning models which were a combination of object-oriented measures with developer activity information to enhance the accuracy of forecasting maintainability of industrial systems. Even though these studies enhance the profession, they also reveal that there are many still existing challenges. They tend to focus on predictive modeling but without giving reproducible and end-to-end processes. The practical interpretability of their conclusions is disregarded by most of them leaving developers without actionable insights. Regardless of the number of publicly available data, however, limited studies have employed such methodologies to open-source Java projects.
Having a focus on maintainability, code complexity, and the quality of documentation, the current study by (Fawareh, H. J in 2025) explores the influence of AI-generated code on the quality of software. Important metrics, including upkeep index (MI), line of code (LOC), cyclical complexity (CC), Turing volume (V), and comment ratio, were evaluated by comparing AI-generated code to open-source code available on GitHub to three jobs of different difficulty (easy, medium, and hard). These findings indicate that AI-generated code is usually more verbose, but its cyclical complexities is less likely to be on simpler jobs, which reduces the rate of mistakes. Maintainability is a significant help to machine generated program workers in complicated jobs and offers more documentation dependent on recommendations[14].
Consequently, there still remains a lack of succinct, integrating, evidence-based approach in the scientific community that systematically associates actual indicators of real sustainability in freely accessible Java software with object-oriented measures[10].
The reason why this hole exists is due to a lack of analyses that integrate both measurement collection, maintenance indicators and statistical inference into a single unambiguous and repeatable procedure that can be conducted on OO systems and not the general absence of metric investigation. In this study, this gap is filled in three aspects. It is a hands-on style of methodology that combines metric extraction, data cleaning, statistical correlation and visual interpretation in an open-source repeatable tool chain. The paper focuses on

interpretability and usability to enable scholars and practitioners to use the findings to apply in the actual codebase. The proposed method is tested experimentally based on the selected open-source Java projects, and its relevance and applicability in both academic, industrial, and open-source software environments are demonstrated. Some past work is compared in Table 1.

Table (1) : Comparison of Related Work

Study	Approach	Tools Used	Scope	Limitation Addressed in This Study
Lee et al. (2023)	Static analysis + issue tracking	Custom framework	Java projects	Adds reproducible pipeline
Park & Kim (2022)	Temporal metric analysis	Statistical methods	Agile Java projects	Adds end-to-end implementation
Zhang et al. (2024)	ML + developer activity	ML models, activity logs	Industrial systems	Improves interpretability, usability
Fawareh, H. J (2025)	Metric extraction + stats + visualization	CK Tool, JavaParser, Python	Open-source Java projects	Novel reproducible workflow; enhances maintainability, complexity, and documentation quality.
This study (2025)	Metric extraction + stats + viz	CK Tool, JavaParser, Python	Open-source Java projects	Novel reproducible, interpretable workflow

3) Methodology

The section outlines a stepwise plan of a strategy of measuring object-oriented software measurements and maintainability indicators in open-source Java projects. Scientific rigor and feasibility are afforded by automated metric extraction, narrow data gathering, stringent statistical analysis and pertinent visualization.
The technique is appropriate in academic and industry research because of its transparency, repeatability and scalability. The authors provide UML diagrams of the system components, processes, and data flows to help the reader and practitioners learn about the structure of the framework.

3.1 Project Selection

To ensure that the dataset is relevant, diverse, and up-to-date, three medium-sized Java-based projects and in constant use that are open-source on GitHub have been selected. The set of projects chosen includes a variety of Java ecosystem services and offers an adequate empirical research background. Approved projects as will:

i. **JUnit:** A common unit testing approach of Java code and continuous integration.
ii. **Apache Commons Lang:** A utility library that is being used in most applications to enhance standard Java classes.
iii. **Google Gson:** This is a powerful package, which converts the Java objects to the JSON and vice versa in order to serialize and deserialize simple information.

The selection criteria are carefully designed in a way that will give substantial and generalizable results. The poll projects all contained more than 1,000 GitHub stars indicating that they are acceptable and useful in the community. As the chosen projects have been continuously committing and maintaining over the last two years, the information reflects the up-to-date development processes. The project codebase is 10,00050,000, which is the depth of analysis. These criteria concentrate research on technically and practically useful projects, which gives a robust basis on software measurements and maintainability indicators.

3.2 Software Metric Extraction

The authors extracted important software metrics from chosen Java applications using two major tools:
i. CK Tool: A tool for gathering Chidamber and Kemerer (CK) object-oriented metrics.
ii. **JavaParser:** JavaParser is a strong code analysis package that extracts structural information from Java

Abstract Syntax Trees (ASTs), the procedure is:

a) Every Java component was handled one after the other.

b) Measurements from the CK Tool were obtained using the default parameters.

c) Java Parser was set up to capture AST-driven compositional metrics by traversing every class and method.

d) Utilizing distinct class identities (such as the class label and package), obtained measurements from the two programs were combined.

e) During analysis of statistics, each result were kept in organized CSV files.

In table 2, extracted metrics include:

Table (2): Description of Software Metrics Used

Metric	Description
Lines of Code (LOC)	Counts the number of lines in a class or method.
Coupling Between Objects (CBO)	Measures interdependence between classes, affecting modularity and reuse.
Lack of Cohesion of Methods (LCOM)	Assesses how closely related a class's methods are.
Weighted Methods per Class (WMC)	Captures overall class complexity by summing method complexities.
Depth of Inheritance Tree (DIT)	Indicates a class's position in the inheritance hierarchy.
Cyclomatic Complexity	Quantifies control flow complexity using McCabe's metric.

3.3 Collection of Maintainability Indicators

Three primary variables measured maintainability. Git log-extracted file change frequency predicts monthly file commits. Second, the amount of GitHub issue tracker requests for files or classes. Developer churn—the number of developers who worked on a file or class throughout the study—is the third indicator. Indicators reveal system stability, evolution, and maintenance overhead.

3.4 Data Integration and Cleaning

The extracted metrics on software and serviceability parameters were then integrated and preprocessed in the following way:

I. Association of Data: To ensure adequate data records arrangement, software performance and maintainability indications were organized with the help of unique identifiers such as class name and package path.

II. Cleaning Data: It was cleaned by removing entries that either had incorrect or no information. In cases where it was applicable, anomalies were identified and removed with the help of the median range (IQR) process. The numeric data were normalized using the Min-Max adjustment so that they could be compared with measurements of different measures.

III. Implementation: Each stage was implemented with Python 3.9, and the Pandas package was applied to transform, filter, and merge data. In the process of further research, the cleaned data got exported in structured CSV files.

Table 3 provides a summary of the workflow.

Table (3) : Data Processing Workflow

Step	Description	Tool Used
Integration	Merge software metrics with maintainability indicators using unique identifiers	Python Pandas
Cleaning	Remove missing/inconsistent entries, handle outliers, normalize values	Python Pandas

3.5 Overview of the Workflow

The study was performed with Python (version 3.9), and an appropriate methodology implied consistency in the extraction of metrics, data preparation, statistical analysis, and visualization. Data loading, cleaning, merging, and converting were done using the Pandas library. SciPy was used to conduct statistical analysis, in the form of Spearman rank correlation and calculation of p-value. Seaborn and Matplotlib were used to perform its data

visualization, and included heat maps, scatter diagrams, and histograms. The most important workflow tasks were:

i. Create Pandas, import data frames and software metrics of maintainability.
ii. Cleanse data of imperfect or conflicting data.
iii. Group files by similar identifiers (e.g. file path, class names).
iv. Observe the correlations of the metrics and indicators of Spearman correlation coefficients.
v. Statistically significant (alpha = 0.05).
vi. Image correlations to prefer meaning and perception.

This workflow is understandable, can be repeated, and scaled and thus other people can easily repeat or enhance anything in this research in their future studies.

3.6 Functional and Non-functional Requirements

A. **Functional requirements**: explain the specific behavior and actions of the system, and processing of data, derivation of measures, and creation of visualisations. These functions are essential in the case of empirical software analysis tools in the sense that, it is possible to perform the calculation of metrics and the complexity of software expressed in the form of maintainability automatically [15-16].
B. **Non-functional requirements**: specify the quality requirements of the system quality example, it may consist of performance, scaling and reproducibility. Modularity is the most important aspect of an analytical tool to be used in a research, modularity in that the tool is supposed to be used again and again and is also supposed to adapt to the various environments and databases; speed in the calculation and traceability of the constantly evolving datasets would also be of importance [17].

The functional and non-functional requirements have been summarized in tabulated form in tables 4 and 5, respectively.

Table (4): Functional Requirements

ID	Requirement
FR1	The system shall import and parse metric data from Java source code.
FR2	The system shall extract object-oriented metrics (e.g., CBO, LCOM, LOC, etc.).
FR3	The system shall collect maintainability indicators from version control logs.
FR4	The system shall merge software metrics with maintainability data.
FR5	The system shall compute statistical correlations between metrics and indicators.
FR6	The system shall visualize relationships using heat maps and scatter plots.
FR7	The system shall generate reports summarizing analytical results.

Table (5): Non-Functional Requirements

ID	Requirement
NFR1	The system should be developed using Python 3.9 and compatible libraries.
NFR2	The analysis tools should process data for projects with at least 10,000 LOC.
NFR3	The visualizations must be clear, accurate, and exportable.
NFR4	The correlation analysis should complete within 2 minutes for medium datasets.
NFR5	The system should ensure reproducibility of results by saving configurations.
NFR6	The tool must be modular and extensible for future metric integrations.

3.7 UML Diagrams

1- Use case : This use case diagram describes the primary actors in the system—Developer and Metrics Analyzer—and shows how they relate to each other. The developer pushes the projects and executes the metric extraction toolsets, where the metrics analyzer processes it and provides graphical reports. Finally, the developer gets feasible suggestions on how best to make the code maintainable.

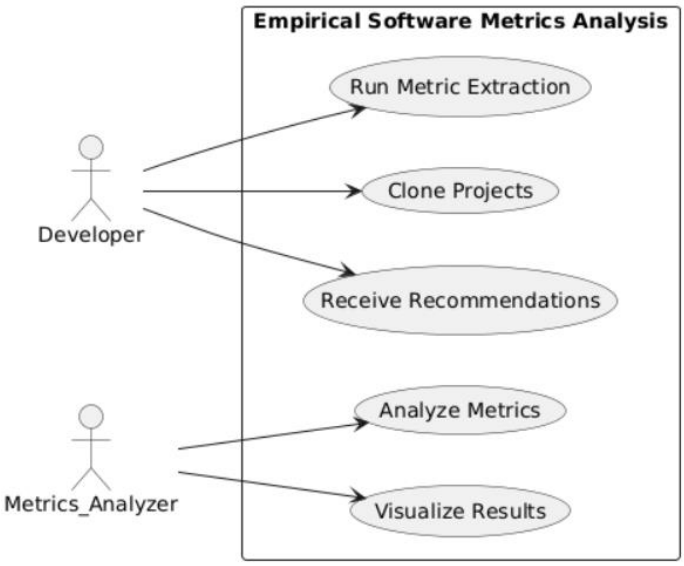


Figure (1): Use Case Diagram for Software Metrics Analysis System

2- **Activity Diagram:** The given activity diagram can demonstrate the chronological working process of the empirical analysis starting with cloning the repositories and continuing with the work on extracting metrics, data processing, performing analysis and its interpretation, and finally working out the conclusions. It explains a process-by-process pipeline of data and shows what needs to be done in order to attain reproducibility and transparency.

Figure (2): Activity Diagram of the Workflow

3- **Sequence Diagram:** The given sequence diagram explains the flow of interaction between the elements that are the Developer and different tools and services in the process of gathering and analyzing data. It focuses the exchanging of data and control messages and characterizes the dynamic behavior and interdependencies involved in the system

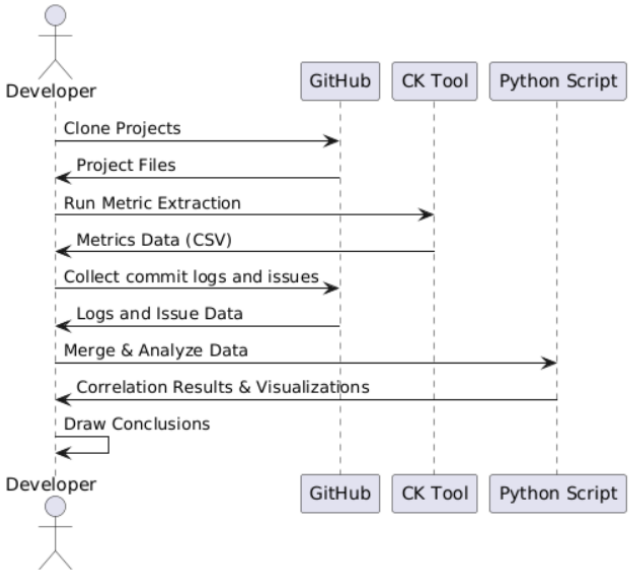


Figure (3): Sequence Diagram of Interactions Between Developer and Tools

4- **Class diagram:** The class diagram is a static structure diagram that visualizes the system's architecture by showing its classes, attributes, operations, and the relationships between objects. Table 4 provides the detailed breakdown of the components depicted in the diagram.

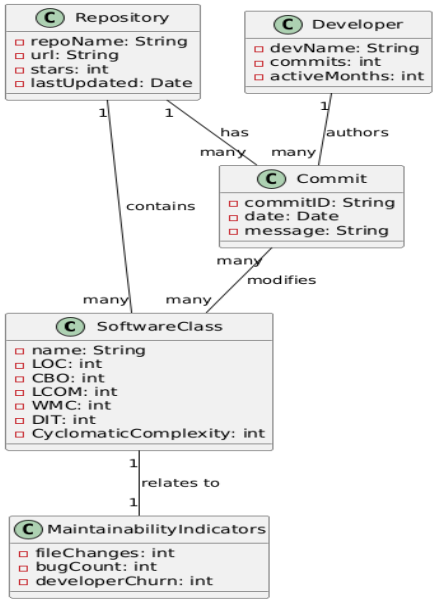


Figure (4): Class Diagram of the Proposed Analysis Framework

Table 6 shows the main classes, their objectives, and important properties in the proposed class diagram. Figure 6 depicts the analytical framework's structural basis.

Table (6): Summary of Main Classes in the Analysis Framework

Class	Purpose	Key Attributes
Software Class	Stores software metrics for a code unit.	name, LOC, CBO, LCOM, WMC, DIT, Cyclomatic Complexity
Maintainability Indicators	Tracks change frequency, bugs, developer churn.	File Changes, bug Count, developer Churn
Repository	Represents the project	Repo Name, url, stars, last Updated

	repository.	
Commit	Details a single code change.	Commit ID, date, message
Developer	Represents a project contributor.	devName, commits, active Months

5- State diagram

The state diagram (Figure 5) shows a source code file's duration from creation to modification, testing, release, and possibly change after problem identification.

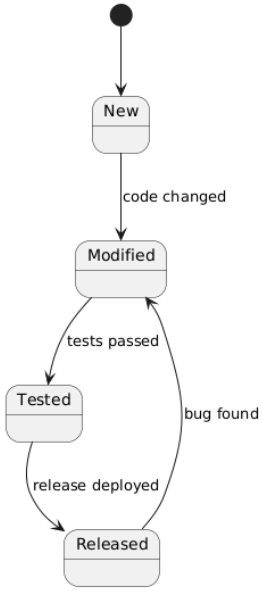


Figure (5): State Diagram of Source Code File Lifecycle

6- Component program

Component diagram (Figure 6) shows the modular structure of the software system and data flow across code repositories, metric extraction tools, analysis scripts, and display interfaces.

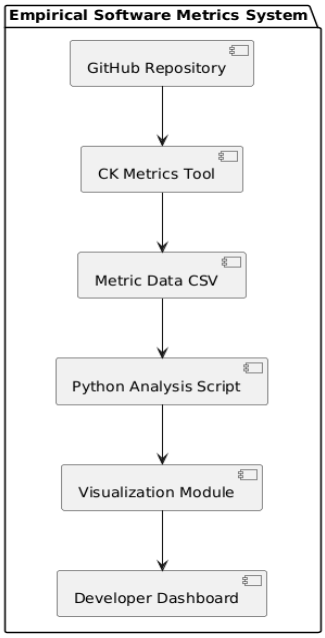


Figure (6): UML Component Diagram of the System.

4. Results and Discussion

These are the statistical results of software metrics and maintainability indicators analysis in three Java projects. A non-parametric test of monotonic correlation strength and significance was conducted by the authors with Spearman rank correlation coefficient (R). The high relationships between software measures such as CBO, WMC, and Cyclomatic Complexity and indicators of maintainability such as the number of bugs and files modified are illustrated in Table 7.

Table (7): Spearman Correlation Matrix Between Software Metrics and Maintainability Indicators

Metric	Bug Count	File Changes	Developer Churn
Coupling Between Objects (CBO)	0.82 **	0.75 **	0.68 **
Lines of Code (LOC)	0.65 **	0.72 **	0.60 **
Weighted Methods per Class (WMC)	0.70 **	0.66 **	0.62 **
Lack of Cohesion of Methods (LCOM)	0.55 **	0.48 *	0.44 *
Depth of Inheritance Tree (DIT)	0.30	0.25	0.20
Cyclomatic Complexity	0.77 **	0.70 **	0.65 **

The statistical significance is denoted in terms of asterisk notation in which a star**(*)** represents that $p = < 0.05$, and 2 stars**(**)** represent that $p = < 0.01$, which are statistically significant and highly significant values, respectively. Table 8 shows bug count correlation coefficients, p-values, and interpretations. There are strong positive correlations between CBO and Cyclomatic Complexity (rho = 0.82, 0.77; $p < 0.001$) but no significant values between DIT and any of them.

Table (8): Bug Count Correlation Coefficients.

Metric	Spearman's ρ	p-value	Interpretation
Coupling Between Objects (CBO)	0.82	<0.001	Strong positive correlation
Lines of Code (LOC)	0.65	0.002	Moderate positive
Weighted Methods per Class	0.70	0.001	Moderate positive
Lack of Cohesion of Methods	0.55	0.020	Moderate positive
Depth of Inheritance Tree	0.30	0.120	Not statistically significant
Cyclomatic Complexity	0.77	<0.001	Strong positive correlation

Table 9 provides summary statistics for all analyzed variables, offering context on their mean, median, variability, and range.

Table (9): Summary Statistics of Software Metrics and Maintainability

Metric	Mean	Median	Std. Dev.	Min	Max
Coupling Between Objects (CBO)	4.7	4	2.8	0	10
Lines of Code (LOC)	265	250	180	50	700
Weighted Methods per Class	10.9	10	7.3	2	25
Lack of Cohesion of Methods	0.37	0.30	0.25	0.05	0.80
Depth of Inheritance Tree	1.6	1	1.3	0	4
Cyclomatic Complexity	7.6	7	5.5	1	18
Bug Count	6.0	5	7.0	0	20
File Changes	16.0	15	16.5	1	50
Developer Churn	3.3	3	2.2	1	8

According to the heat map (Figure 7), CBO, LCOM, and WMC cluster with files with more defects or modifications. High coupling or complexity classes changes more frequently because CBO and WMC are most positively associated with defect-prone and unstable files. LCOM shows a minor relationship with bug frequency, validating earlier research that poor cohesion may affect maintainability. Cyclomatic Complexity is a little connected. with developer turnover, suggesting that developers may avoid or rewrite complicated classes to decrease maintenance.

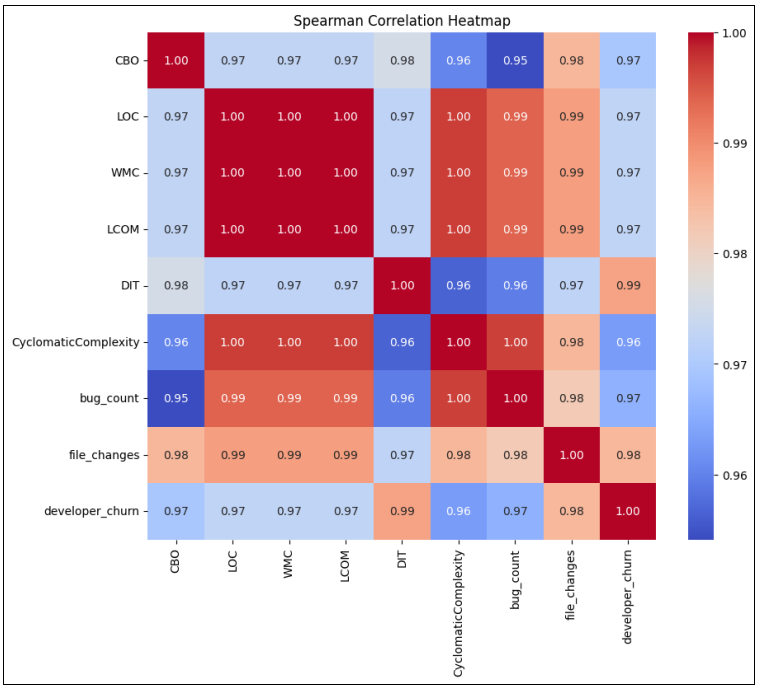
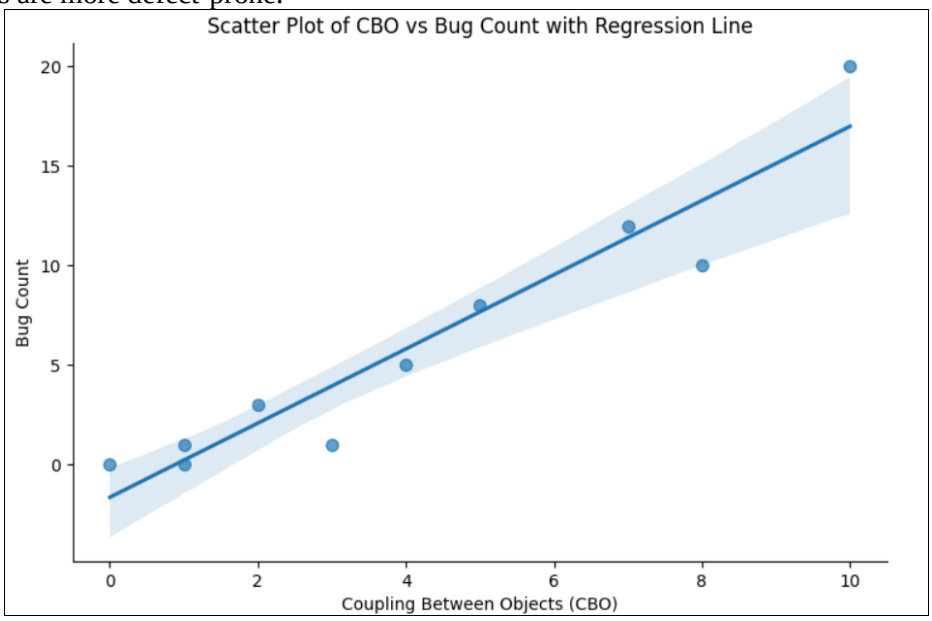


Figure (1): Heat map of Spearman Correlation Coefficients

In Figure 8, the scatter plot between CBO and bug count reveals a rising trend, confirming that strongly connected classes are more defect-prone.



Figure(8) : Scatter Plot – CBO vs Bug Count

The pair plot (Figure 9) shows diagonal distributions and paired scatter plots off-diagonal for LOC, CBO, WMC, LCOM, DIT, Cyclomatic Complexity, and maintainability indices.

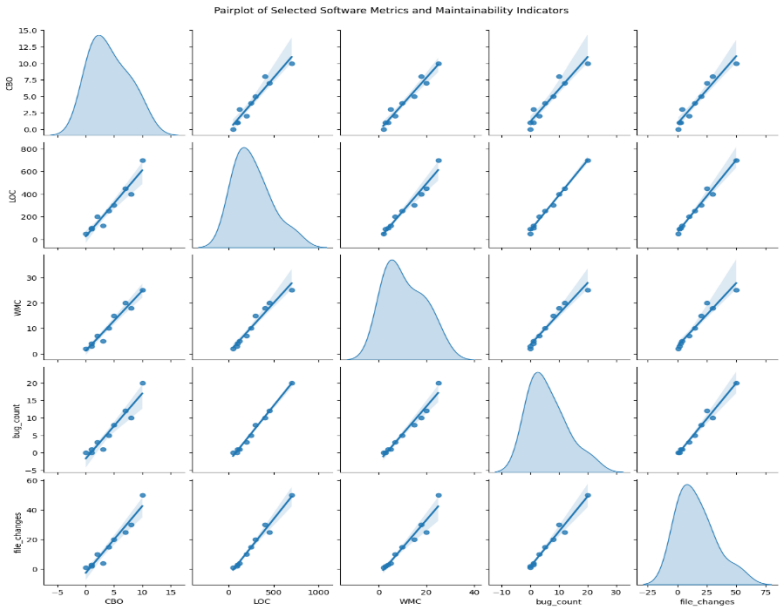


Figure (2):Pair plot of Software Metrics and Maintainability Indicators

The box plot (Figure 10) shows that CBO values grow throughout low, medium, and high bug counts, supporting the relationship between strong coupling and defect rates.

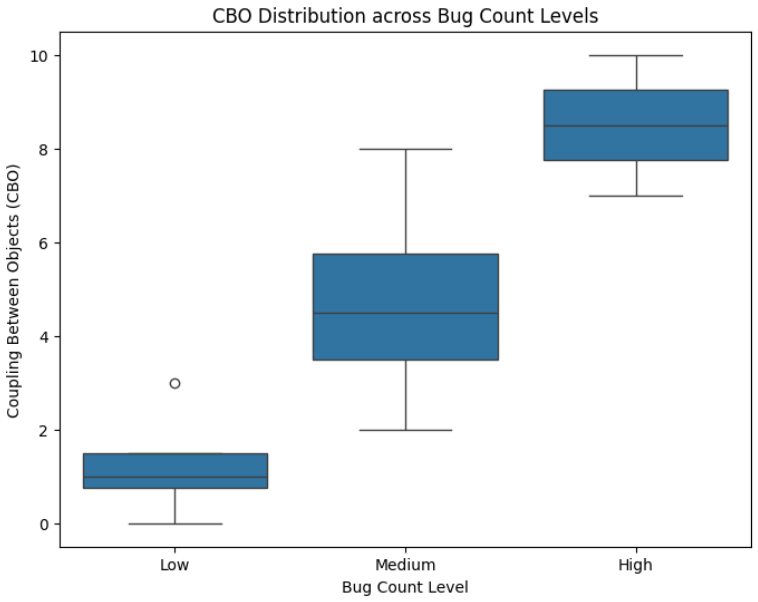


Figure (3): Box Plot of CBO Distribution Across Bug Count Levels

In Figure 11, histograms with kernel density curves show the dispersion and variability of important metrics and bug counts throughout the studied Java.

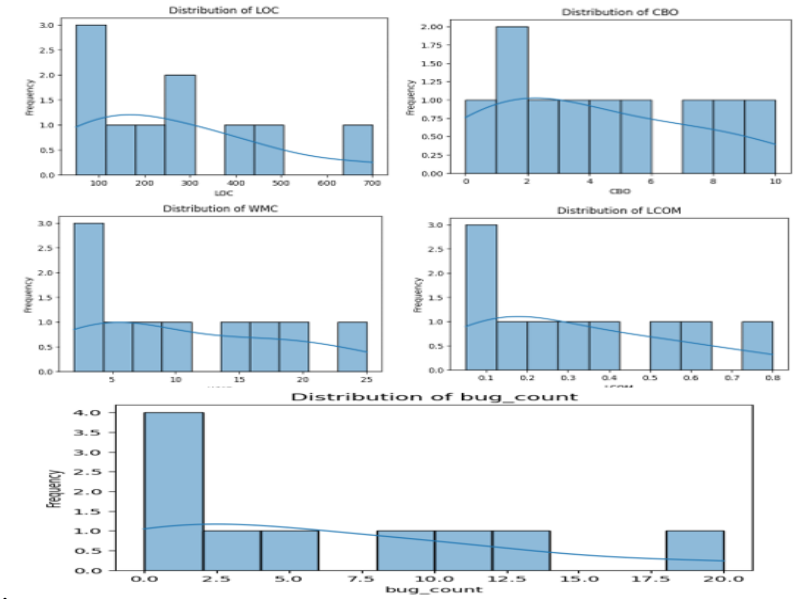


Figure (11): Distribution Plots of Key Software Metrics and Bug Count

4.1 Discussion

Results confirm past studies and provide new evidence from real-world software initiatives. Early coupling analysis during system design is crucial since CBO was the largest indicator of poor maintainability. Among the first metrics, LOC and LCOM still help identify maintenance-intensive modules. High structural complexity may raise maintenance and discourage developer participation, since cyclomatic complexity is linked to developer turnover. DIT's inadequate prediction ability supports past research that questioned its maintainability assessment efficacy.

4.2 Implications for Practice

Dashboards for real-time CBO, LCOM, and WMC monitoring, alarms for dangerous metrics during reviews, and refactoring to target classes with high LOC, CBO, and LCOM.

4.3 Threats to Validity

Tool inaccuracies may influence internal validity, although CK Tool and Java Parser reduced such dangers. Due to Java-based open-source systems, conclusions may not apply to other systems. Bug count, file modifications, and developer churn may not represent all maintainability factors, limiting construct validity. Finally, correlation does not indicate causality and implies monotonic variable connections, limiting statistical validity.

5. Conclusion

Object-oriented software metrics are dependable maintainability indicators, as shown by this research. CBO, LCOM, and Cyclomatic Complexity consistently lower code quality and increase maintenance effort. To address such risks, the development teams may rely on metric dashboards and code health monitoring tools based on automated tools as well as smarter refactoring choices. Further research should focus on real-time monitoring of metrics, historical trend analysis, and dynamic, runtime-driven metrics to gain a better insight into the maintainability of systems. To provide deeper understanding of systems maintenance and to help in the better maintenance and reorganization activities, further studies should be conducted on the immediate software measures monitoring, longitudinal analysis of measurement patterns, and the use of dynamic and runtime specific statistics.

Acknowledgments: The authors would like to give a warm welcome to Mustansiriyah University (https://uomustansiriyah.edu.iq).

Conflict of Interest: None of conflict interest.

References

[1] A. E. Hassan and Z. Xing, "Software quality metrics: A survey and analysis," IEEE Transactions on Software Engineering, vol. 46, no. 9, pp. 930–945, 2020.

[2] S. Ali, E. Shihab, and A. E. Hassan, "A comprehensive study on software maintainability prediction," Journal of Systems and Software, vol. 157, p. 110388, 2019.

[3] M. Alsaqa'aby, A. Ouni, and H. Sahraoui, "Assessing the effectiveness of object-oriented metrics for defect prediction: A systematic literature review," Information and Software Technology, vol. 129, p. 106424, 2021.

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[5] H. D. Nguyen and T. T. Nguyen, "Revisiting CK metrics for modern Java applications: Empirical validation and extension," Empirical Software Engineering, vol. 27, no. 3, p. 42, 2022.

[6] C. Catal and B. Diri, "A systematic review of software fault prediction studies," Expert Systems with Applications, vol. 37, no. 4, pp. 1105–1116, 2019.

[7] Y. Zhang et al., "Challenges in adopting software quality metrics in industrial practice: A survey study," IEEE Software, vol. 38, no. 2, pp. 55–62, 2021.

[8] H. Borges et al., "Mining open-source software repositories for empirical studies: Challenges and solutions," Journal of Systems and Software, vol. 163, p. 110555, 2020.

[9] S. Kim and T. Zimmermann, "An empirical study of code churn and fault prediction in open-source projects," Information and Software Technology, vol. 111, pp. 39–54, 2019.

[10] L. Jiang et al., "Machine learning-based software maintainability prediction: A systematic literature review and framework," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 150–175, 2023.

[11] S. Lee, J. Choi, and H. Park, "Automated framework for assessing software maintainability using combined static analysis and issue tracking data," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp. 1560–1574, 2023. doi: 10.1109/TSE.2022.3154210.

[12] K. Park and D. Kim, "Temporal analysis of software metrics evolution in agile Java projects," Journal of Systems and Software, vol. 189, p. 111352, 2022. doi: 10.1016/j.jss.2022.111352.

[13] Y. Zhang, J. Li, and F. Wang, "Hybrid machine learning approach for maintainability prediction combining code metrics and developer activity," Empirical Software Engineering, vol. 29, no. 1, p. 21, 2024. doi: 10.1007/s10664-023-10324-7.

[14] Fawareh, H. J., Al-Shdaifat, H. M., & Samara, G. (2025). Comparing Open Source with Software Code Generated by AI Tools from Software Maintainability Quality Factor Perspective. *WSEAS Transactions on Computer Research*, *13*, 653-659.

[15] Y. Huang and X. Chen, "Explainable machine learning for software quality prediction: Challenges and opportunities," ACM Computing Surveys, vol. 55, no. 2, p. 34, 2023. doi: 10.1145/3551234.

[16] T. Gorschek, S. Fricker, and C. Wohlin, Requirements Engineering: Fundamentals, Principles, and Techniques. Springer, 2023.

[17] B. Kitchenham, D. Budgen, and O. Brereton, Evidence-Based Software Engineering and Systematic Reviews. CRC Press, 2022.