Alkadhim Journal for Computer Science, Vol. 3, No. 4 (2025)

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]. More recent work adds cyclomatic complexity to measure the complexity of control flow, taking modifiability modeling even further [6].
while research has advanced metric theory and prediction models, real-world industrial projects often skip systematic metric application due to tool integration challenges, limited data access, and difficulty interpreting results [7]. Even as open-source software (OSS) ecosystems provide rich empirical data for validation, the lack of reproducible, practitioner-ready workflows limits their impact on day-to-day software practice [8,9].
A transparent, repeatable, and practical pipeline that relates object-oriented metrics to useful supportability insights for open-source Java projects is the focus of this research. Our research presents an automated, hands-on procedure for metric extraction, maintainability indicator collecting, and Spearman statistical analysis. Interpretable visuals and actionable insights rather than numerical correlations combine academic quality with practical value. A customizable framework for academic and industry research is created by validating the suggested technique using genuine open-source software project. This research applies to software quality assurance, CI/CD pipeline integration, technical debt management, software engineering education, and static code analysis tool design. Structure of the paper: Section 2 discusses relevant work and the research gap; Section 3 describes the methodology, including project selection, metric extraction, and statistical analysis; Section 4 discusses results and comments; and Section 6 closes with major conclusions and future prospects.

2) Related Work and Research Gap

In the past few years, the software engineering community has greatly increased its interest in finding answers about the practical effects of object-oriented metrics on software maintainability, driven by both the proliferation of open-source repositories and by new data mining opportunities. Another example is provided by Lee et al. (2023) [11], who formulated an automated combination of static code analysis and issue tracker to evaluate maintainability of a Java project, which resulted in high correlations among conventional measures and maintenance effort. On the same note, Park and Kim (2022) [12] studied the dynamics of software metric change over time to demonstrate how the evolution over time leads to maintainability and the propensity to defects in agile settings.

In addition, Zhang et al. (2024) [13] used hybrid machine learning models that combined object-oriented metrics with developer activity data to improve industrial system maintainability forecast accuracy. Although these research improve the profession, they also indicate numerous persisting obstacles. They often concentrate on predictive modeling without providing replicable, end-to-end processes. Most ignore their conclusions' practical interpretability, leaving developers without actionable insights. Despite the quantity of publicly accessible data, few research have directly applied such methodologies to open-source Java projects.

With an emphasis on maintainability, code complexity, and documentation quality, the present research by (Fawareh, H. J in 2025) investigates how AI-generated code affects software quality. We assessed important metrics, such as upkeep index (MI), line of code (LOC), cyclical complexity (CC), Turing volume (V), and comment ratio, by contrasting AI-generated code with open-source code from GitHub for three jobs of varied difficulty (easy, medium, and hard). The results show that while AI-generated code is typically more verbose, its cyclical complexities tends to decrease on simpler jobs, which lowers mistake rates. Maintainability greatly aids workers with machine-generated program in complex tasks and provides greater documentation based on suggestions[14].

As a result, there's continues to absence of a concise, cohesive, and supported by evidence methodology in the scientific community that methodically links real sustainability indications in freely available Java applications with object-orientated metrics[10].

A scarcity of analyses that combine measurement gathering, maintenance indicators and statistical inference into an unambiguous and repeatable procedure that can be applied to OO systems is the cause of this hole rather than the overall lack of metric investigation. This research addresses this gap in three ways. A new, hands-on methodology unifies metric extraction, data cleaning, statistical correlation, and visual interpretation in an open-source, repeatable tool chain. The study emphasizes interpretability and usability so academics and practitioners may apply the results to real-world codebases. The suggested technique is experimentally evaluated using chosen open-source Java projects, proving its relevance and applicability across academic, industrial, and open-source software settings. Table 1 compares previous work.

Table (1) : Related Work Comparison

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

This section describes a step-by-step strategy for assessing object-oriented software metrics and maintainability indicators in open-source Java projects. Automated metric extraction, focused data collecting, rigorous statistical analysis, and relevant visualization provide scientific rigor and practical applicability. Transparency, repeatability, and scalability make the technique suited for academic and industry research. The authors include accompanying UML diagrams of system components, processes, and data flows to assist readers and practitioners understand the framework's architecture.

3.1 Project Selection

Three Java-based, medium-sized, regularly maintained open-source projects on GitHub were chosen to make the dataset relevant, diverse, and current. This selection of projects covers a range of Java ecosystem services and provides a good empirical research base. Approved projects as will:

i. **JUnit** : A typical Java development and continuous integration pipeline unit testing method.
ii. **Apache Commons Lang** : A utility library used in many applications for improving standard Java classes.
iii. **Google Gson** : A robust package converts Java objects to JSON and inversely for simple information serialization and deserialization.

Selection criteria have been thoughtfully developed to ensure substantial and generalizable findings. Each project in the poll had over 1,000 GitHub stars, showing community acceptability and usefulness. Since all selected projects have had continuous commits and maintenance for two years, the data represents current development procedures. Each project has a 10,000–50,000-line codebase that balances complexity and manageability for deep analysis. These criteria focus research on technically and practically relevant projects, providing a strong foundation for software metrics 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 following integration and preprocessing was done on the extracted metrics for software and serviceability parameters:

I. Integration of Data: To guarantee proper record arrangement, software performance and maintainability signals were combined utilizing distinctive identifications like class name and package path.

II. Cleaning Data: Entries with incorrect or lacking information were eliminated. When applicable, anomalies were found and eliminated utilizing the median range (IQR) approach. Min-Max adjustment was used to normalize numerical data so that they could be compared across various measures.

III. Execution: Python 3.9 was used for each phase, along with the Pandas package for data transformation, filtering, and merging. During further investigation, the cleaned dataset was exported as structured CSV files.

A summary of the workflow is given in Table 3.

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 using Python (version 3.9) and a well described methodology to assure consistency in metric extraction, data preparation, statistical testing, and visualization. Pandas for data loading, cleaning, merging, and transformation; SciPy for Spearman rank correlations and p-values; and Seaborne and Matplotlib for heat maps, scatter plots, and histograms.

The key steps of the workflow were

i. Import Data Frames in Pandas and maintainability data and software metrics.
ii. Make data clean of incomplete or inconsistent records.
iii. Combine files based on common identifiers (e.g., file path, class names).
iv. Calculate the correlations of the metrics and indicators of Spearman correlation coefficients.
v. Calculate the statistical significance (alpha = 0.05).
vi. Picture relationships to favor interpretation and insight.

This workflow is clear, repeatable and scalable and so others find it easy to replicate or build on anything in this study in subsequent research.

3.6 **Functional and Non-functional Requirements**

A. **Functional requirements**: describe the particular behaviour and activities of the system, including processing of data, extraction of metrics, and genering of visualizations. In the case of empirical software analysis tools, these functions are vital so that the computation of metrics and the complexity of software stated in the form of maintainability can be done automatically [15-16].

B. **Non-functional requirements**: define the quality attributes of the system quality example, this might include performance, scalability and reproducibility. In an analytical tool to be used within research, modularity is paramount, since the tool is intended to be reused, validated, and flexible to different environments and databases; speed of calculation and traceability of ever-changing datasets is also important [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.

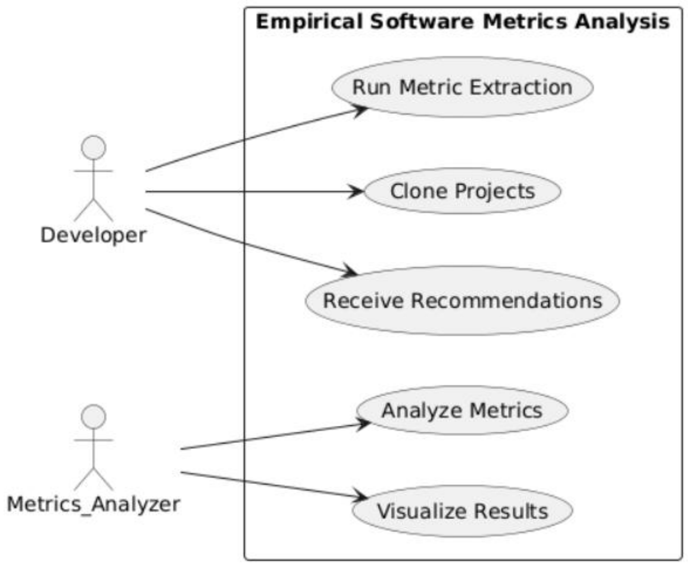


Figure (1): Use Case Diagram for Software Metrics Analysis System

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.

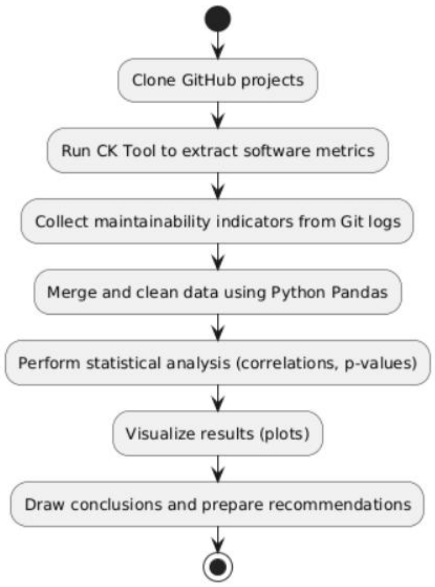

Clone GitHub projects
Run CK Tool to extract software metrics
Collect maintainability indicators from Git logs
Merge and clean data using Python Pandas
Perform statistical analysis (correlations, p-values)
Visualize results (plots)
Draw conclusions and prepare recommendations

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

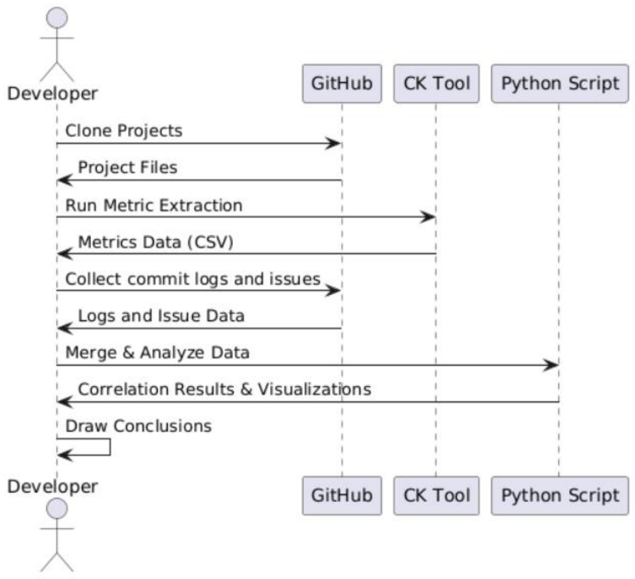


Figure (3): Sequence Diagram of Interactions Between Developer and Tools

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.

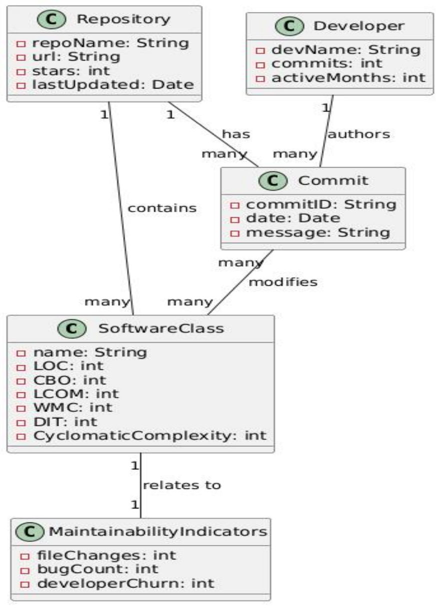


Figure (4): Class Diagram of the Proposed Analysis Framework

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.

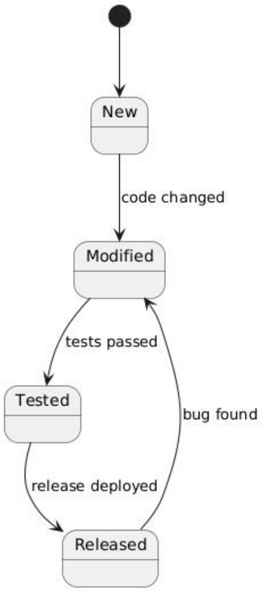


Figure 5: State Diagram of Source Code File Lifecycle

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.

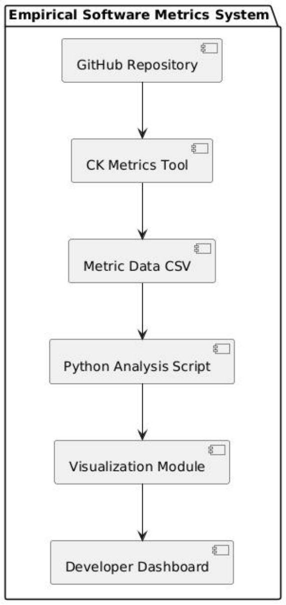


Figure (6): Component Diagram of the System Architecture

4. Results and Discussion

The statistical findings from analyzing software metrics and maintainability indicators in three Java projects are shown here. The authors assessed monotonic correlation strength and significance using Spearman's rank correlation coefficient (ρ) for non-parametric data. Table 7 shows high connections between software measures like CBO, WMC, and Cyclomatic Complexity and maintainability indicators like bug count and file modifications.

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 **

*($p < 0.05$ = *, p < 0.01 = ** → significant)*

Table 8 shows bug count correlation coefficients, p-values, and interpretations. CBO and Cyclomatic Complexity have substantial positive associations (ρ = 0.82, 0.77; $p < 0.001$), but DIT is not significantly related.

Table (8): Bug Count vs. Key Metrics Spearman Correlation and Significance Test

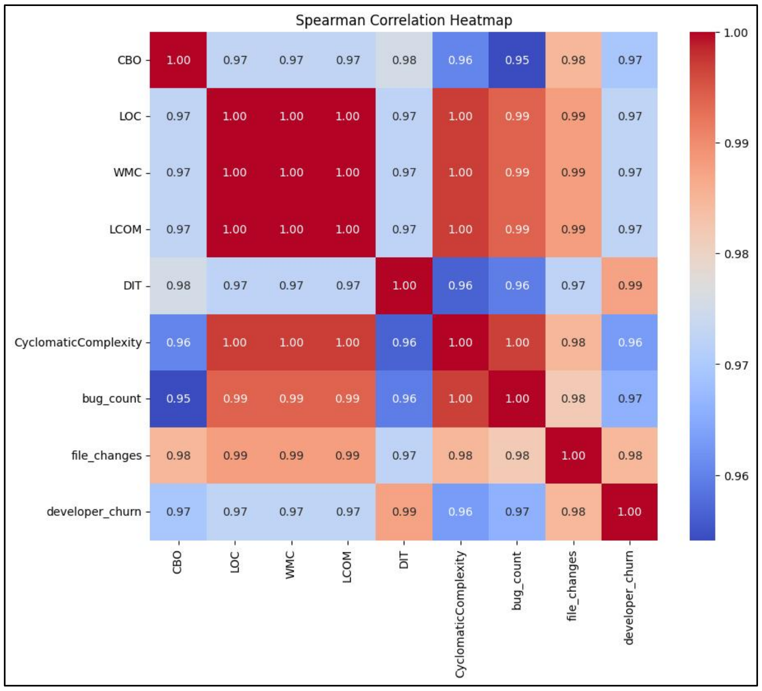
Metric	Spearman's ρ	p-value	Interpretation
Coupling Between Objects (CBO)	0.82	<0.001	Strong positive correlation
Lines of Code (LOC)	0.65	0.002	Moderate positive
Weighted Methods per Class	0.70	0.001	Moderate positive
Lack of Cohesion of Methods	0.55	0.020	Moderate positive
Depth of Inheritance Tree	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

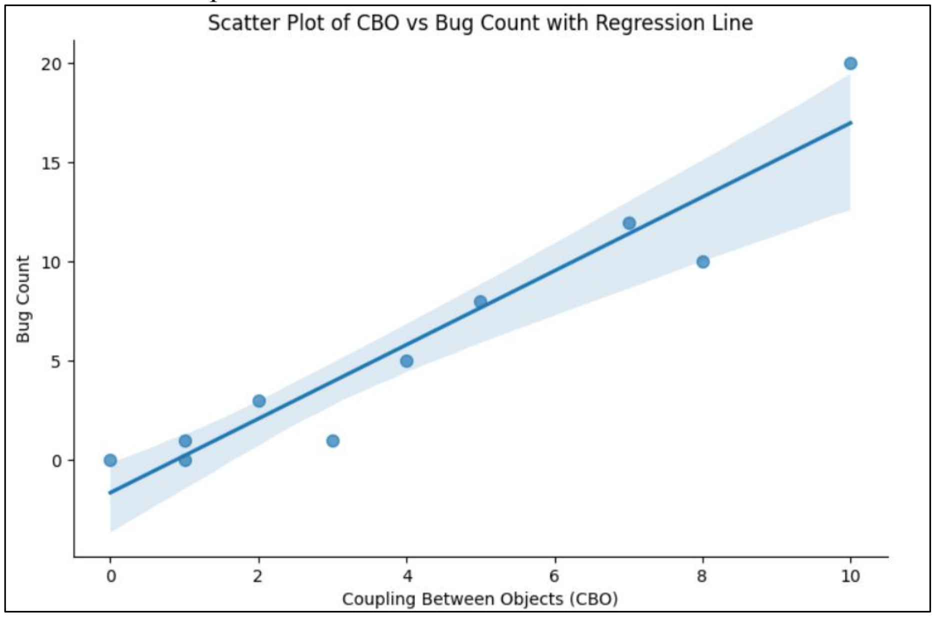
According to the heat map (Figure 7), CBO, LCOM, and WMC cluster with files with more defects or modifications. High coupling or complexity classes changes more frequently because CBO and WMC are most positively associated with defect-prone and unstable files. LCOM shows a minor relationship with bug 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.



	CBO	LOC	WMC	LCOM	DIT	CyclomaticComplexity	bug_count	file_changes	developer_churn
CBO	1.00	0.97	0.97	0.97	0.98	0.96	0.95	0.98	0.97
LOC	0.97	1.00	1.00	1.00	0.97	1.00	0.99	0.99	0.97
WMC	0.97	1.00	1.00	1.00	0.97	1.00	0.99	0.99	0.97
LCOM	0.97	1.00	1.00	1.00	0.97	1.00	0.99	0.99	0.97
DIT	0.98	0.97	0.97	0.97	1.00	0.96	0.96	0.97	0.99
CyclomaticComplexity	0.96	1.00	1.00	1.00	0.96	1.00	1.00	0.98	0.96
bug_count	0.95	0.99	0.99	0.99	0.96	1.00	1.00	0.98	0.97
file_changes	0.98	0.99	0.99	0.99	0.97	0.98	0.98	1.00	0.98
developer_churn	0.97	0.97	0.97	0.97	0.99	0.96	0.97	0.98	1.00

Figure (7): 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.

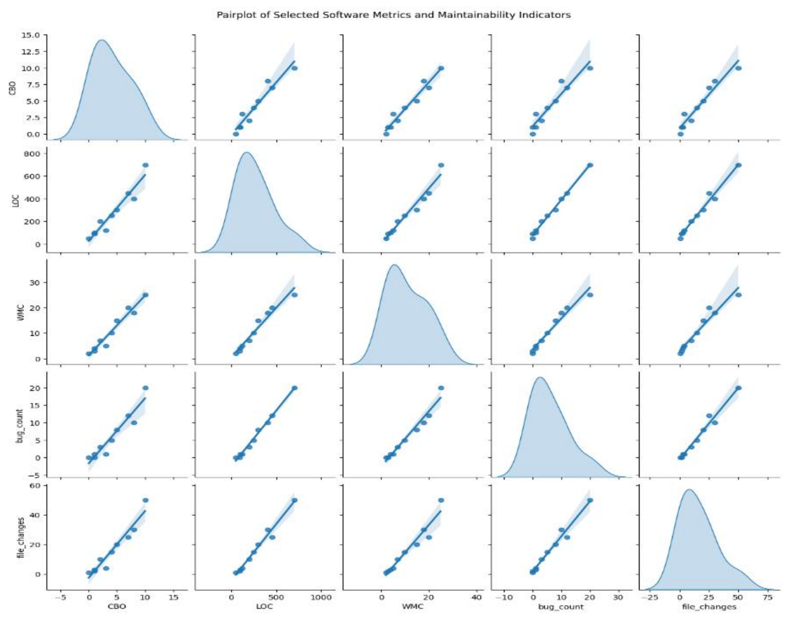


Figure (9):Pair plot of Software Metrics and Maintainability Indicators

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.

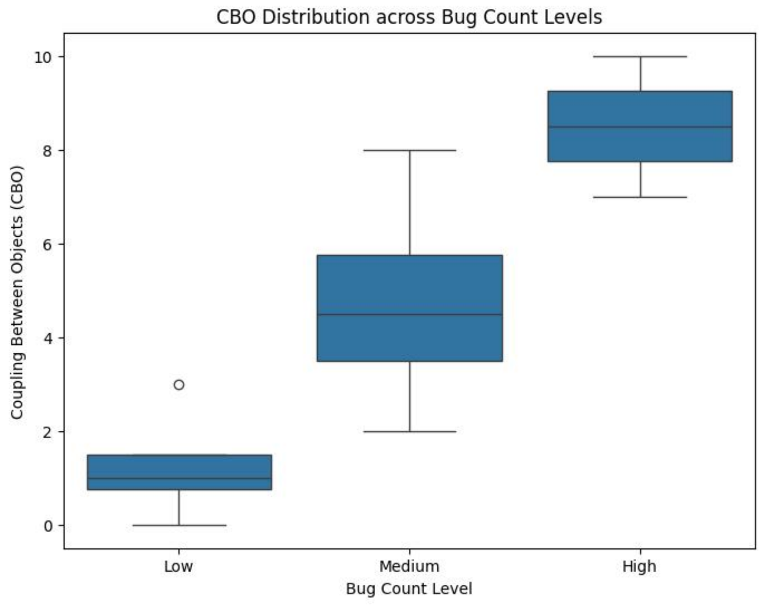


Figure (10): Box Plot of CBO Distribution Across Bug Count Levels

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

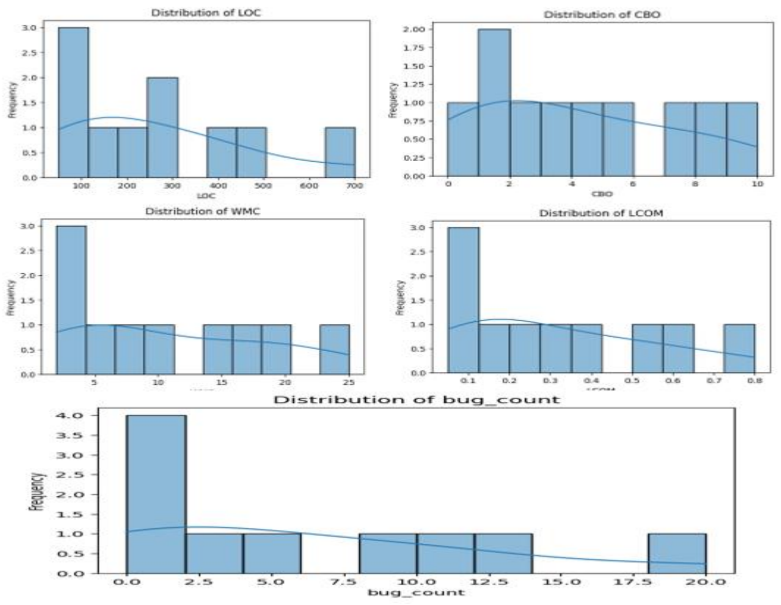


Figure (11): Distribution Plots of Key Software Metrics and Bug Count

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 mitigate these risks, development teams can use metric dashboards and automated tools for code health monitoring and smarter refactoring choices. Real-time metric monitoring, historical trend analysis, and dynamic, runtime-based metrics should be investigated in future study to better understand system maintainability. With the goal to offer more comprehensive insight into systems upkeep and to assist with improved upkeep and restructuring actions, subsequent research needs to focus on immediate software measures tracking, longitudinal analysis of measurement patterns, and the application of dynamic, runtime-specific statistics.

Acknowledgments: The authors would like to thank Mustansiriyah University (https://uomustansiriyah.edu.iq).

Conflict of Interest: No conflict of 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.