Proceedings of the
13th International Conference on Applied Informatics
Eger, Hungary, February 19–21, 2026
DOI: 10.17048/icai.2026.81
pp. 81–95

Assessing Software Metrics and Complexity in Student and Open-Source Projects Using Static Analysis

Péter János Császár, Máté Cserép

Eötvös Loránd University,
Faculty of Informatics,
Hungary
csaszarpeter@inf.elte.hu
mcserep@inf.elte.hu

Abstract. Maintaining code quality remains a significant challenge in both educational and professional software development. This paper presents the development and evaluation of static code analysis tools aimed at identifying structural issues, such as complexity, cohesion, and maintainability in C# projects. The analyzers were applied to student projects from a Software Technology course. Code metrics were computed across multiple development milestones to track trends in code quality over time.

In addition to student projects, a diverse set of publicly available open-source C# projects was analyzed to evaluate the behavior of the implemented metrics in real-world software systems. These projects vary in size, domain, architectural style, and development practices, enabling a broader assessment of the robustness and general applicability of the approach. Due to the wide range of targeted .NET versions and dependency configurations across projects, the analysis was conducted using a Docker-based environment to provide controlled and reproducible execution contexts.

The results demonstrate that metric-based static analysis can effectively support the early detection of problematic code segments and provide actionable insights to improve software quality. The approach is adaptable and shows potential for integration into continuous development workflows in both academic and real-world environments.

Keywords: static code analysis, software metrics, code quality, maintainability, cohesion, complexity, C#, student projects, open-source software

1. Introduction

The increasing size and complexity of modern software systems place growing demands on both developers and educators to ensure long-term code quality. While conventional static analysis tools have proven effective in identifying syntax errors, type mismatches, and some well-known anti-patterns [9, 15], they often fall short in capturing deeper structural issues that affect software quality, such as complexity, maintainability, and readability. By quantifying specific properties such as cyclomatic complexity [19], cohesion, and parameter count in functions, software metrics can reveal patterns that may not be immediately obvious through traditional analysis methods [24, 28].

In the context of university assignment and submission management systems – such as TMS (Task Management System)[1] and Submitty[2] – previous research has already explored the application of automatic static code analysis to support the evaluation of student submissions [3, 14, 25]. Building on this foundation, our research aims to extend these tools to enable complexity and cohesion-based analysis of student projects written in C#, using various software metrics, and focusing on the automatic detection of problematic code segments and code smells. The metrics examined and implemented during the research include *Lack of Cohesion of Methods (LCOM)* [16], *Bumpy Road* [5], *Functional Parameter Count*, *Maintainability Index* [20], *Class coupling* [1], and *Cyclomatic Complexity* [19]. The implementation is based on the well-known .NET Compiler Platform SDK (Roslyn) [21]. These metrics can be used to determine the complexity and cohesion of the code. By continuously monitoring these metrics, it becomes possible to track their fluctuations throughout the development lifecycle and to evaluate the coherence of the system at the module and type levels.

The main contributions of this paper are the implementation of code metric analyzers based on the Roslyn platform and their application to both student and open-source C# projects. Furthermore, the study provides an empirical evaluation of these metrics, including threshold calibration using the elbow method and a comparative analysis across educational and real-world software systems.

2. Related work

Static analysis has been widely used to support the automated evaluation of student programming submissions. Kaszab, Cserép demonstrated that static analyzers can effectively identify common programming errors and coding standard violations in large collections of C++ and C# student solutions [15]. Their work led to the integration of static analysis into the Task Management System (TMS), showing that automated feedback can improve submission quality while reducing instructor workload.

[1] https://tms-elte.gitlab.io
[2] https://submitty.org/

Beyond rule-based analysis, software metrics have long been used to assess code quality, complexity, and maintainability. A comprehensive review by Nuñez-Varela et al. summarizes the use of code metrics across multiple programming paradigms, highlighting the prevalence of object-oriented metrics such as CK metrics, Lines of Code, and Cyclomatic Complexity in quality assessment and fault prediction [24]. Earlier work at NASA's Software Assurance Technology Center [28] emphasizes that software defects are often introduced early in development, motivating the use of metrics to support early detection and prevention.

Software metrics have also been applied to defect prediction using machine learning techniques. Rebro, Chren, Rossi evaluated multiple object-oriented metrics on large Java codebases and found that cohesion- and structure-related metrics outperform simple size-based measures such as Lines of Code [27]. Structural metrics have been used to study software stability over time, for example, through coupling-based instability measures [18, 29].

Cohesion metrics, particularly the Lack of Cohesion of Methods (LCOM), have been extensively studied. Kirğil, Ayyildiz compared several LCOM variants and showed that LCOM5 provides a more sensitive measure of cohesion changes after refactoring [16]. However, applying cohesion metrics in practice requires careful threshold selection. Mishra [22] highlights that metric thresholds are highly context-dependent, while Filó, Bigonha, Ferreira provide empirical evidence that empirically tuned thresholds can reduce false positives in quality assessment [7].

Finally, prior work in software engineering education shows that students often struggle to refactor complex code [26], reinforcing the need for automated tools that can highlight problematic code segments early and guide targeted instructional intervention.

3. Methodology

The metrics analyzer must fulfill certain requirements: some of these are basic concepts for analyzers, others are special requirements for making a possible integration into assignment grading systems easier.

1. The tool must be able to perform code metrics calculation on a given C# solution.

2. The threshold values for these metrics should be adjustable from a configuration file.

3. The tool must be capable to produce a machine-interpretable output (e.g. a *XML* file), in addition to a summarized console output. This detailed output shall contain the metric results segmented by methods or classes, showing the corresponding metric values. The output path should also be configurable.

4. The analyzer should be applicable to arbitrary C# projects targeting the .NET platform without requiring project-specific modifications. Only a so-

lution or project file should be needed, while unsupported frameworks or custom build procedures may require additional configuration.

The analyzer relies on the Roslyn compiler platform to construct semantic models from C# projects. In most cases, providing a solution or project file is sufficient, and dependencies are restored automatically through the standard .NET build process. The approach has been successfully applied to WPF, Avalonia, Unity, and ASP.NET-based projects.

3.1. Implemented code metrics

In this section, the implemented code smells are presented.

3.1.1. Bumpy Road

This analyzer evaluates the nesting depth of statements within a method body. If the method body is empty or contains no statements, it is skipped. Otherwise, the nesting depth is computed recursively for each relevant statement. Relevant statements are *if*, *for*, *while*, *do*, *switch*, *block*, *local declaration statement* and *expression statement*. The result of the metrics is the average depth across all statements. The diagnostic is reported if this value exceeds the threshold defined in the configuration [5].

3.1.2. Functional Parameter Count

Methods with a high number of parameters are harder to read and understand, and they often indicate that the method does not have a single responsibility or that the required data is not properly encapsulated. A long parameter list also increases coupling to callers, makes call sites noisier, and complicates testing and reuse. Clean Code explicitly recommends keeping the number of function parameters low (ideally zero to two, and avoiding three or more when possible), since each additional parameter raises the cognitive load and the chance of incorrect usage [17].

This method-level analyzer counts the parameters for a given method and reports the result if it is higher than the configured threshold.

3.1.3. Lack of Cohesion of Methods

The LCOM4 metric [13] measures how well the methods in a class are connected by calling each other or accessing common fields. A cohesive class will have few independent method groups.

The LCOM5 is a stricter cohesion metric than LCOM4. It calculates cohesion based on the fields accessed by all methods of a class. If no fields are shared across methods, LCOM5 value approaches 1, indicating poor cohesion; if many fields are shared, the LCOM5 value approaches 0, indicating good cohesion. There are multiple approaches to compute the LCOM5 metric and studies have shown

[16] that the variant defined by Henderson and Sellers [11] is superior. If l is the number of attributes, k is the number of methods, and a is the sum of the number of distinct attributes, that are accessed by every method in a class, then LCOM5 is calculated as follows:

$$\text{LCOM5} = \frac{a - kl}{l - kl}$$

3.1.4. Maintainability index

Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. Higher values represent better maintainability. If HV is the Halstead Volume, CC is Cyclomatic Complexity and LC is the Lines of Code, than the maintainability index can be calculated as follows [20]:

$$\text{MI} = \max\left(0, \frac{100}{171} \times \left(171 - 5.2 \times \ln(\text{HV}) - 0.23 \times \text{CC} - 16.2 \times \ln(\text{LC})\right)\right)$$

3.1.5. Cyclomatic complexity

Cyclomatic complexity measures the structural complexity of code by counting the number of linearly independent execution paths within a method. It is primarily influenced by the presence of control-flow constructs such as conditional branches and loops. While this metric is conceptually related to the implemented *Bumpy Road* code smell, the two can yield different results in practice.

The Bumpy Road metric focuses on sudden changes in complexity across consecutive code segments, capturing irregular growth patterns within a method. In contrast, cyclomatic complexity reflects the overall decision density of a method, regardless of how that complexity is distributed. As a result, a method with evenly distributed control flow may exhibit high cyclomatic complexity without triggering the Bumpy Road smell, whereas a method with localized spikes in complexity may be flagged by Bumpy Road despite having a moderate overall cyclomatic complexity.

3.1.6. Class coupling

Class Coupling measures the degree of coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decorations. Good software design emphasizes high cohesion and low coupling. High coupling suggests a design that is difficult to reuse and maintain due to its numerous interdependencies with other types.

3.2. Determining threshold values for implemented analyzers

To minimize false positives and undetected faults, it is crucial to establish appropriate threshold values for software metrics, beyond which metric analyzers should

trigger warnings [7, 22]. Although the literature provides several recommended threshold values for commonly used software metrics [6, 17, 23], these recommendations are typically derived from observations on large-scale industrial software systems. As such, they may not be directly applicable to student projects, which differ significantly in size, development practices, and architectural complexity. For this reason, a dedicated threshold analysis was conducted on the implemented metrics based on the collected student project data.

The threshold analysis was performed for each metric by evaluating several threshold values on the student project dataset. To identify appropriate thresholds, the elbow method was applied to the cumulative number of warnings produced at each threshold value [2, 31]. The examined ranges were selected around commonly recommended values from the literature and adjusted according to the metric domain. For each metric, the total number of reported violations was plotted against the corresponding threshold values, and the knee point was automatically determined using the Kneedle algorithm [30], assuming a convex, monotonically decreasing curve. The threshold corresponding to the detected knee point was selected as the optimal value.

In contrast to other metrics, no robust threshold could be identified for LCOM5 and the Maintainability Index. For LCOM5, this is mainly due to its limited discriminative power: cohesion metrics in the LCOM family lack consensus, and prior studies have shown that they may provide inconsistent or misleading representations of class cohesion [4]. In our experiments, warning counts remained high across a wide range of thresholds, indicating that LCOM5 is overly sensitive for threshold-based classification.

For the Maintainability Index, the issue lies in its formulation and interpretation. The metric was originally derived from a limited set of procedural systems and does not explicitly capture object-oriented properties, while its threshold values are inconsistent across literature and tools. Recent work has also shown that different variants of the Maintainability Index can lead to differing assessments [10, 12].

Therefore, thresholds based on the literature were applied; however, both metrics should be interpreted with caution and are more suitable for comparative trend analysis than for strict threshold-based warning generation.

The resulting threshold values for each implemented metric are summarized in Table 1.

4. Analyzing student projects

This section presents the results of the code metrics analysis conducted on student projects from previous semesters of the Software Technology course at our university. Students implement a predefined game in groups of 3–4 members under the supervision of an instructor acting as product owner and, optionally, a senior student acting as scrum master.

Four milestones are defined, three of which involve implementation. Following

Table 1. Threshold values.

Metric	General thresholds	Applied on student projects
Bumpy Road code smell	4	2
Functional parameters	5	2
Lack of Cohesion (LCOM4)	1	4
Lack of Cohesion (LCOM5)	0.5	0.5
Maintainability index	65	65
Cyclomatic complexity	10	6
Class coupling	9	15

the design and requirements phase, students deliver two prototypes (approximately 30% and 90% of the use cases), followed by a final version conforming to clean-code principles [17], including testing and continuous integration. The last three milestones were considered in the analysis.

Although projects may be implemented in arbitrary languages, only C# projects were included. Common frameworks include *WPF*, *Avalonia*, *Unity*, and *Godot*.

The student projects are version-controlled and managed using GitLab. For each project, the last commit before each implementation milestone deadline on the default branch was retrieved using the GitLab API [8]. A total of 61 student projects were collected, of which 41 C# projects could be successfully analyzed after excluding repositories with build failures, missing dependencies, or other technical issues. Considering the three implementation milestones, up to 123 project versions were included in the analysis.

4.1. Visualization and discussion of the results

The goal of the analysis is to evaluate the effectiveness of the implemented code metrics in identifying potential code quality issues, such as high complexity, low cohesion, or maintainability concerns. Projects were analyzed at multiple milestones during their development, providing insights into how coding standards and structural requirements were followed throughout the project lifecycle. The reported values represent the number of warnings exceeding predefined thresholds, normalized by lines of code to ensure comparability across projects of different sizes.

In the line charts presented in Figure 1, and 2, selected metrics are visualized for the analyzed student projects. The trends reflect how the corresponding quality attributes evolve over time, with each line representing a single project.

Overall, the normalized summary warning counts show that most projects remain in the lower or middle range of the diagrams, indicating generally acceptable code quality across the analyzed milestones. At the same time, for most projects and metrics, an increase can be observed from the first to later milestones.

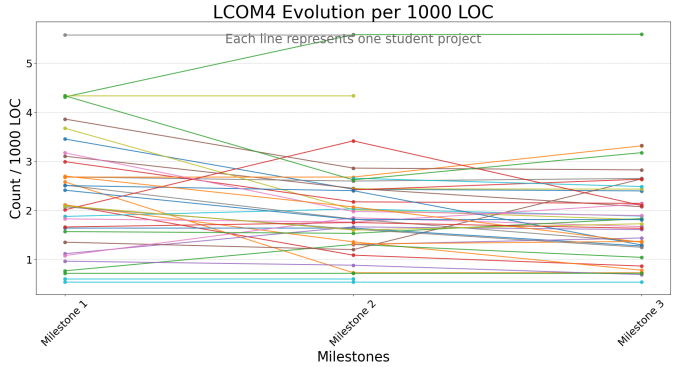

Figure 1. LCOM4 metric results on student projects.

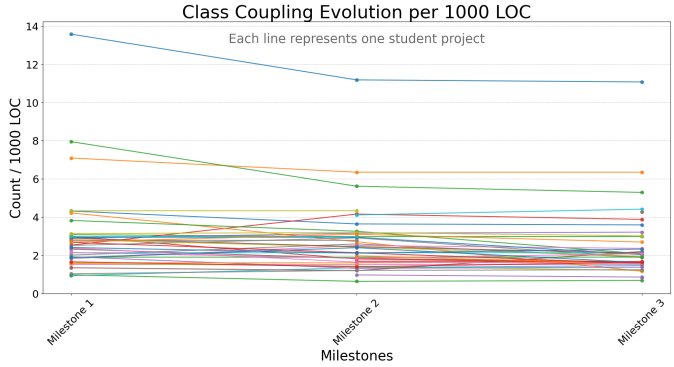

Figure 2. Class coupling metric results on student projects.

This suggests that as systems grow, the number of warnings exceeding the applied thresholds tends to increase, even after normalization by lines of code.

It is also important to note that the development effort between milestones is not uniform. Between the second and third milestones, students typically implement a significant portion of the functionality – approximately 60% of the total use cases – while later stages involve only minor additions. Consequently, a substantial deterioration in code quality in later phases should not necessarily be expected; in fact, improvements could be anticipated if teams perform refactoring and cleanup. The observed results partially align with this expectation: while some projects stabilize or slightly improve, a consistent overall improvement is not evident, suggesting that refactoring activities were limited in many cases.

The increase in warning counts is not uniform across all metrics. While cohesion- and coupling-related metrics show relatively consistent trends, the Bumpy Road metric exhibits more variability, with several projects showing fluctuations or even decreases between milestones. This indicates that certain structural characteristics

captured by Bumpy Road are less directly correlated with project growth and may be more sensitive to local code changes.

Despite these differences, the projects tend to cluster within comparable regions of the diagrams, suggesting a relatively similar overall quality level across teams. This consistency may indicate that the course structure, requirements, and supervision result in a broadly uniform development standard, even though individual projects differ in how their warning counts evolve over time.

5. Analyzing open-source projects

In addition to evaluating student projects, open-source projects were analyzed to further validate the implemented metrics and assess their applicability in a broader context. Open-source repositories provide diverse real-world codebases and enable evaluation of the metrics beyond the educational setting. In contrast to the student project analysis, which employed dataset-specific thresholds, open-source projects were evaluated using the threshold values recommended in the literature (Table 1).

Metrics were evaluated on selected major releases from five open-source .NET projects. In total, 22 tagged versions spanning several years were considered. Since the analyzer targets the .NET platform, only releases that could be successfully restored and analyzed were included. Versions requiring unsupported frameworks, failing to build, or containing unresolved dependencies were excluded. The varying number of analyzed versions across projects reflects differences in release histories and compatibility with the analysis tool.

5.1. Selected projects

The open-source projects were selected to provide diversity in terms of application domain, project size, and software architecture, while ensuring compatibility with the .NET ecosystem and the analysis tool. Preference was given to mature, actively maintained projects with multiple tagged releases available, enabling the study of metric evolution over time.

In total, 22 tagged versions spanning several years were considered. Since the analyzer targets the .NET platform, only releases that could be successfully restored and analyzed were included. Versions requiring unsupported frameworks, failing to build, or containing unresolved dependencies were excluded. The varying number of analyzed versions across projects reflects differences in release histories and compatibility with the analysis tool.

For the evaluation, the following open-source projects were selected: *i*) *OpenRA*[3], a real-time strategy game engine inspired by the classic Command & Conquer: Red Alert; *ii*) *Jellyfin*[4], a media server for organizing and streaming multimedia

[3] https://github.com/OpenRA/OpenRA
[4] https://github.com/jellyfin/jellyfin

content; *iii*) *nopCommerce*[5], an open-source e-commerce platform; *iv*) *xUnit*[6], a widely used unit testing framework for the .NET ecosystem; and *v*) *IronPython 3*[7], an open-source implementation of the Python programming language.

5.2. Visualization and discussion of the results

Figures 3 and 4 visualize the same metrics showcased and discussed for the student projects in subsection 4.1. The metric values were normalized by code size, specifically expressed as values per 1000 lines of code, to allow fair comparison between projects of different sizes.

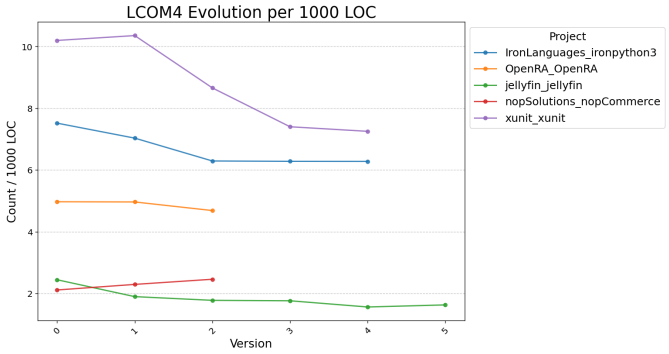

Figure 3. LCOM4 metric results on open-source projects.

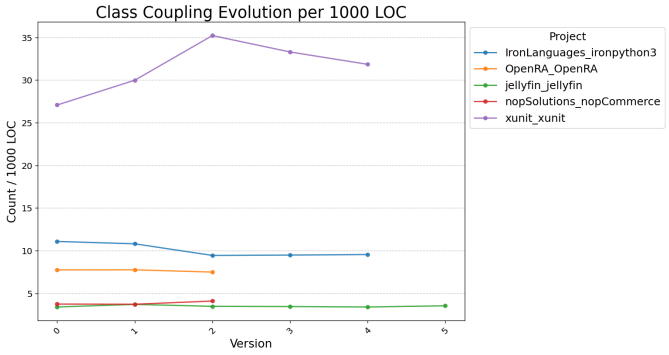

Figure 4. Class coupling metric results on open-source projects.

Overall, the results show low to moderate fluctuations in metric values across releases, with changes generally occurring gradually over time.

[5] https://github.com/nopSolutions/nopCommerce
[6] https://github.com/xunit/xunit
[7] https://github.com/IronLanguages/ironpython3

The Bumpy Road metric exhibits relatively stable behavior across most projects, with only minor variations between releases. Jellyfin shows a slight decreasing trend, while OpenRA and nopCommerce remain largely stable. IronPython maintains consistently higher values, whereas xUnit shows some fluctuation in earlier releases followed by stabilization. Overall, the metric suggests that localized structural irregularities are either stable or slightly improving over time.

The Functional Parameter Count metric remains relatively stable for most projects, with only minor variations across releases. An exception is xUnit, which exhibits a significant spike in intermediate releases, indicating a temporary increase in methods exceeding the parameter threshold. Apart from this outlier, the metric suggests consistent method-level design practices.

For cohesion, LCOM4 shows a generally decreasing or stable trend across all projects, indicating gradual improvements or controlled cohesion as the systems evolve. In contrast, LCOM5 exhibits higher values and greater variability, particularly for IronPython and xUnit, and does not provide a clear or consistent trend across projects. This further supports the observation that LCOM5 is less suitable for threshold-based evaluation.

Class Coupling shows relatively stable or slightly decreasing trends for most projects, suggesting that dependencies between classes are either maintained at a consistent level or gradually improved. xUnit again stands out with significantly higher values and noticeable fluctuations, reflecting its more complex structure.

For the Maintainability Index, IronPython exhibits consistently high values with a slight decreasing tendency, while Jellyfin remains relatively stable. In contrast, OpenRA and nopCommerce show a steady increase in warning counts across releases, and xUnit demonstrates a more pronounced upward trend with minor fluctuations. These results suggest that, for some projects, system growth is accompanied by an increasing number of maintainability-related issues.

In summary, the analyzed open-source projects exhibit largely stable or slightly improving trends for most structural metrics, with some project-specific deviations. The presence of occasional spikes (notably in xUnit) suggests localized increases in complexity, but overall the results indicate controlled evolution and generally consistent code quality across releases.

6. Comparing student and open-source projects

6.1. Descriptive statistics

Table 2 summarizes the distribution of normalized warning counts for student and open-source projects. It should be noted that the two datasets were evaluated using different threshold sets: thresholds calibrated on the student dataset were applied to student projects, whereas literature-recommended thresholds were used for open-source projects. Consequently, the reported warning counts should be interpreted primarily as descriptive indicators, and direct comparisons of absolute values should be made with caution.

Table 2. Descriptive statistics of normalized warning counts (per 1000 LOC). Values are reported as Student/Open-source.

Metric	Mean	Median	Std. Dev.
Bumpy Road	1.33 / 0.88	1.25 / 0.38	0.75 / 0.86
Functional Parameter Cont	3.22 / 5.13	2.29 / 1.14	3.04 / 8.51
LCOM5	4.95 / 5.88	4.32 / 5.60	2.15 / 3.89
LCOM4	2.10 / 4.99	1.89 / 4.97	1.10 / 2.84
Cyclomatic Complexity	4.92 / 3.95	4.90 / 2.09	1.95 / 3.06
Class Coupling	2.71 / 11.96	2.14 / 7.75	1.95 / 10.97

The larger standard deviations observed for several metrics indicate greater architectural diversity among open-source systems. Moreover, the larger differences between mean and median values for metrics such as Functional Parameter Count and Class Coupling suggest skewed distributions and the presence of outliers. In contrast, student projects exhibit smaller mean–median differences, indicating more homogeneous quality characteristics. Finally, LCOM5 yields consistently high values with relatively small mean–median differences in both datasets, supporting previous observations regarding its limited discriminative power.

6.2. Trend analysis

Despite the differences in absolute warning counts, the evolution of metric values exhibits clear differences between the two datasets. Open-source projects generally show moderate variation followed by stabilization or slight improvements, suggesting controlled evolution and ongoing refactoring. In contrast, student projects exhibit a more pronounced increase in normalized warning counts, indicating that structural issues tend to accumulate as functionality is added, with limited evidence of systematic refactoring.

For cohesion- and coupling-related metrics, open-source systems generally exhibit more stable trends, whereas student projects display a clearer increasing tendency. Cyclomatic Complexity and Functional Parameter Count show moderate fluctuations and occasional spikes in open-source projects, while student projects tend to increase more consistently. For Maintainability Index warnings, both datasets exhibit mixed behavior, indicating that maintainability-related issues are present in both cases.

Overall, the results suggest that mature open-source systems are characterized by greater architectural diversity and more controlled evolution over time. Although the use of different threshold sets limits the direct comparison of absolute warning counts, the trend analysis indicates that student projects exhibit a stronger tendency to accumulate complexity and structural issues during development.

7. Conclusion

This research aimed to identify code quality issues related to complexity, cohesion, and maintainability using static code analysis metrics. The developed analyzers were evaluated on student projects from the *Software Technology* course and on selected open-source repositories. Threshold values were determined using the elbow method, and the work extends existing Roslyn-based metrics with additional custom analyses.

The results show that student projects tend to exhibit increasing structural complexity over time, while open-source systems demonstrate more stable evolution, often supported by refactoring practices. Cohesion- and coupling-related metrics (e.g., LCOM4 and Class Coupling) highlight this difference most clearly. In contrast, maintainability-related warnings show mixed trends in both groups and do not provide a clear distinction.

Several challenges were identified, including inconsistencies in milestone progression and the context-dependent nature of metric thresholds. Additionally, the selection of suitable open-source projects was constrained by build and dependency issues.

Despite these limitations, the proposed approach proved effective in identifying problematic code segments and tracking quality trends throughout development. The analyzers can be integrated into continuous integration environments, enabling early feedback and supporting improved coding practices.

Overall, the results confirm that carefully calibrated code metrics provide a practical and scalable means for assessing and improving software quality in both educational and real-world development contexts.

8. Future work

For future work, the following aspects have been identified for further development:

- Implement additional metrics to cover a wider range of code quality attributes, for example cohesion metrics, such as Relational Cohesion.
- Analyze a broader set of student and open-source projects to further evaluate and validate the implemented metrics.
- Perform manual verification to assess whether the code segments flagged by the analyzers indeed contain problematic code, thereby validating the diagnostic accuracy.
- Provide module and submodule level metrics evaluation inside projects, to identify potentially problematic segments in a generally acceptable codebase.

Computer Code Availability

The developed source code metrics and the multitemporal analyzer tool implemented as part of this research is released under the BSD-3 license at: https://github.com/petercsaszar/code-metrics/

References

[1] E. Arisholm, L. Briand, A. Føyen: *Dynamic Coupling Measurement for Object-Oriented Software*, Software Engineering, IEEE Transactions on 30 (Sept. 2004), pp. 491–506, DOI: 10.1109/TSE.2004.41.

[2] P. Bholowalia, A. Kumar: *EBK-means: A clustering technique based on elbow method and k-means in WSN*, International Journal of Computer Applications 105.9 (2014).

[3] S. Breese, A. Milanova, B. Cutler: *Using Static Analysis for Automated Assignment Grading in Introductory Programming Classes (Abstract Only)*, in: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '17, Seattle, Washington, USA: Association for Computing Machinery, 2017, p. 704, ISBN: 9781450346986, DOI: 10.1145/3017680.3022440, URL: https://doi.org/10.1145/3017680.3022440.

[4] L. Briand, J. Daly, J. Wust: *A unified framework for cohesion measurement in object-oriented systems*, in: Proceedings Fourth International Software Metrics Symposium, 1997, pp. 43–53, DOI: 10.1109/METRIC.1997.637164.

[5] CodeScene: *The Bumpy Road Code Smell: Measuring Code Complexity by its Shape and Distribution*, 2025, URL: https://codescene.com/engineering-blog/bumpy-road-code-complexity-in-context/.

[6] CppDepend: *Understanding Code Metrics in CppDepend*, 2025, URL: https://www.cppdepend.com/documentation/code-metrics.

[7] T. Filó, M. Bigonha, K. Ferreira: *Evaluating Thresholds for Object-Oriented Software Metrics*, Journal of the Brazilian Computer Society 30 (Sept. 2024), pp. 313–346, DOI: 10.5753/jbcs.2024.3373.

[8] GitLab: *REST API*, 2025, URL: https://docs.gitlab.com/api/rest/.

[9] I. V. Gomes, P. Morgado, T. Gomes, R. M. L. M. Moreira: *An overview on the Static Code Analysis approach in Software Development*, in: 2009, URL: https://api.semanticscholar.org/CorpusID:9373127.

[10] I. Heitlager, T. Kuipers, J. Visser: *A Practical Model for Measuring Maintainability*, in: 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), 2007, pp. 30–39, DOI: 10.1109/QUATIC.2007.8.

[11] B. Henderson-Sellers: *Object-oriented metrics: measures of complexity*, USA: Prentice-Hall, Inc., 1995, ISBN: 0132398729.

[12] T. Heričko, B. Šumak: *Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems*, Applied Sciences 13.5 (2023), ISSN: 2076-3417, DOI: 10.3390/app13052972, URL: https://www.mdpi.com/2076-3417/13/5/2972.

[13] M. Hitz, B. Montazeri: *Measuring coupling and cohesion in object-oriented systems*, na, 1995.

[14] P. Kaszab: *Automated evaluation of programming assignments with static code analysis*, MA thesis, Eötvös Loránd University, 2023, URL: https://tms-elte.gitlab.io/theses/kaszab_peter_tdk.pdf.

[15] P. Kaszab, M. Cserép: *Detecting Programming Flaws in Student Submissions with Static Source Code Analysis*, Studia Universitatis Babeș-Bolyai Informatica 68 (July 2023), pp. 37–54, doi: 10.24193/subbi.2023.1.03.

[16] E. N. H. Kirğil, T. E. Ayyildiz: *Analysis of Lack of Cohesion in Methods (LCOM): A Case Study*, in: 2021 2nd International Informatics and Software Engineering Conference (IISEC), 2021, pp. 1–4, doi: 10.1109/IISEC54230.2021.9672419.

[17] R. C. Martin: *Clean code: a handbook of agile software craftsmanship*, Pearson Education, 2009.

[18] R. C. Martin, October: *OO Design Quality Metrics*, in: 1997, url: https://api.semanticscholar.org/CorpusID:18246616.

[19] T. McCabe: *A Complexity Measure*, IEEE Transactions on Software Engineering SE-2.4 (1976), pp. 308–320, doi: 10.1109/TSE.1976.233837.

[20] Microsoft: *Code metrics - Maintainability index range and meaning*, 2026, url: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning.

[21] Microsoft: *The .NET Compiler Platform SDK*, 2024, url: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/.

[22] A. Mishra, R. Shatnawi, C. Catal, A. Akbulut: *Techniques for Calculating Software Product Metrics Threshold Values: A Systematic Mapping Study*, Applied Sciences 11.23 (2021), issn: 2076-3417, doi: 10.3390/app112311377, url: https://www.mdpi.com/2076-3417/11/23/11377.

[23] NDepend: *100+ .NET and C# Code Metrics*, 2025, url: https://www.ndepend.com/docs/code-metrics.

[24] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo: *Source code metrics: A systematic mapping study*, Journal of Systems and Software 128 (2017), pp. 164–197, issn: 0164-1212, doi: https://doi.org/10.1016/j.jss.2017.03.044, url: https://www.sciencedirect.com/science/article/pii/S0164121217300663.

[25] B. D. Orosz: *Automated validation of design and architectural patterns on student assignments with static code analysis*, MA thesis, Eötvös Loránd University, 2024, url: https://tms-elte.gitlab.io/theses/orosz_balint_dominik_tdk.pdf.

[26] R. Rajapakse, C. Szabo: *Code Refactoring Strategies of Third Year Software Engineering Students*, in: Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2024, Milan, Italy: Association for Computing Machinery, 2024, pp. 562–568, isbn: 9798400706004, doi: 10.1145/3649217.3653560, url: https://doi.org/10.1145/3649217.3653560.

[27] D. A. Rebro, S. Chren, B. Rossi: *Source Code Metrics for Software Defects Prediction*, in: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC '23, Tallinn, Estonia: Association for Computing Machinery, 2023, pp. 1469–1472, isbn: 9781450395175, doi: 10.1145/3555776.3577809, url: https://doi.org/10.1145/3555776.3577809.

[28] L. H. Rosenberg, T. Hammer, J. G. Shaw: *SOFTWARE METRICS AND RELIABILITY*, in: 1998, url: https://api.semanticscholar.org/CorpusID:59798472.

[29] D. Santos, A. Resende, E. de Castro Lima, A. Freire: *Software Instability Analysis Based on Afferent and Efferent Coupling Measures*, Journal of Software 12 (Jan. 2017), pp. 19–34, doi: 10.17706/jsw.12.1.19-34.

[30] V. Satopaa, J. Albrecht, D. Irwin, B. Raghavan: *Finding a Kneedle in a Haystack: Detecting Knee Points in System Behavior*, in: July 2011, pp. 166–171, doi: 10.1109/ICDCSW.2011.20.

[31] R. L. Thorndike: *Who Belongs in the Family?*, Psychometrika 18.4 (1953), pp. 267–276, doi: 10.1007/BF02289263.