YILDIZ TECHNICAL UNIVERSITY
FACULTY OF ELECTRICAL AND ELECTRONICS
DEPARTMENT OF COMPUTER ENGINEERING

3D VISUALIZATION OF OBJECT ORIENTED SOFTWARE METRICS

Eray Gökçe, Asst. Prof. Dr. Yunus Emre Selçuk
eray.gokce@std.yildiz.edu.tr, yselcuk@yildiz.edu.tr

Özet

Bu proje, nesne yönelimli yazılım sistemlerinin yapısal kalitesini yazılım metrikleriyle analiz edip, bu verileri 3 boyutlu ortamda görselleştirmeyi amaçlar. Java projeleri, BCEL kütüphanesiyle bytecode üzerinden analiz edilir. Hesaplanan metrikler veritabanına kaydedilir ve Unity motoru ile görselleştirilir. Her sınıf bir bina olarak modellenir; renk, yükseklik ve genişlik metrikleri yansıtır. Spring Boot ile geliştirilen backend, React arayüz ve Unity görselleştirme modülüyle entegre çalışır. Sistem, sınıf düzeyinde mimari sorunları sezgisel olarak tespit etmeyi kolaylaştırır.

Abstract

This project aims to evaluate the structural quality of object-oriented software systems by analyzing software metrics and visualizing them in a 3D environment. Java projects are analyzed via bytecode using the BCEL library. Calculated metrics are stored in a database and visualized using the Unity engine. Each class is represented as a building, with color, height, and width reflecting different metrics. The system integrates a Spring Boot backend, a React-based frontend, and Unity for visualization. It enables developers to intuitively detect architectural issues at the class level, making structural analysis more interactive and insightful.

I. Introduction

As software systems evolve, maintaining quality becomes a major challenge [1]. Without proper initial design, code may become complex and hard to manage. Object-Oriented Programming structures—such as class relationships, inheritance, and method distribution—directly impact software quality. To measure these properties, Chidamber and Kemerer proposed metrics like WMC, DIT, LCOM, and RFC [2][3]. However, interpreting metrics solely as numbers is often insufficient. Visualization provides intuitive understanding. Lanza and Ducasse introduced “polymetric views” to represent cohesion and complexity visually [4]. Wettel and Lanza's CodeCity presents classes as buildings, helping detect flaws effectively.

II. System Design

The system follows a three-layer architecture: backend, frontend, and visualization. The backend, built with Spring Boot, analyzes Java bytecode using the BCEL library and computes software metrics. These results are stored in a database and exposed via RESTful APIs. The React-based frontend allows users to upload files, select metrics, and view past analyses. The Unity-based visualization module receives JSON data from the backend and represents each class as a 3D building in a virtual city. Visual attributes like height, width, and color reflect metric values, allowing intuitive exploration of software quality and architectural structure.

A. Backend Design

The project's backend structure is designed according to a layered architecture. The controller layer hosts API endpoints and receives requests from the client. The service layer contains the business logic, and metric analysis is performed here. The Repository & Model layer provides database access; analysis and metric results are stored using JPA. DTO structures simplify data transfer, enabling effective communication with the client. The Util layer contains utility classes that manage metric calculations. This structure improves code readability and provides a sustainable development environment.

B. Database Design

The database structure consists of two main tables: analysis and metric_result. The analysis table stores the project name, analysis type (CLASS/JAR), target path, and creation time for each analysis process. The metric_result table stores the metric results (WMC, TCC, CBO, LCOM, etc.) for each class and is linked to the relevant analysis via analysis_id. This allows a single analysis to be associated with multiple classes. Thanks to the JSON structure, data is both readable and easily processable via API. This structure enables efficient storage and retrieval of past analysis data.

C. Visualization Design

Unity converts metric data from the backend into a 3D “code city” metaphor [5], representing each class as a building. The three metrics selected by the user are visualized as the height, width, and color of the building. As shown in Table 1, metrics are normalized before visualization; each is converted to a specific range (e.g., WMC: 0-30). In most metrics, lower values indicate better quality. This makes metric density in complex projects intuitively understandable. The map width is dynamically determined based on the number of classes, and the numerical values of metrics are visible in building details.

Table 1. Normalization Ranges

	Normalization	Is Lower Better
DIT	0 – 5	Yes
WMC	0 – 30	Yes
LCOM	0 – 5	Yes
CBO	0 – 12	Yes
TCC	0 – 1	No
MAXCYCLO	1 – 30	Yes
AVGCYCLO	1 – 7	Yes

III. Experimental/Application Results

The project was tested on Java projects of different sizes to evaluate the accuracy of metric analysis and system stability. As shown in Figure 2, the application can display analysis results both intuitively and numerically. The application worked consistently in both small-scale class analyses and large-scale JAR files. Three metrics selected by the user were matched with building height, width, and color, allowing the software architecture to be examined visually and intuitively. Normalizing metric values ensures consistent visualization across different projects. As shown in Figure 1, based on the principle that low metric values are considered “good,” problematic classes stand out more prominently in the city.

The application results were not limited to instantaneous analyses; past data could also be stored and accessed again. Thanks to database support, users could view previous analysis records and compare the structural evolution of the project over time. The visualization module provided historical analysis capabilities by projecting these old records back into the 3D scene. The Unity scene dynamically expanded based on the number of classes, and clicking on details allowed users to access numerical metric information for each class. This structure provided a significant advantage in terms of quality traceability.

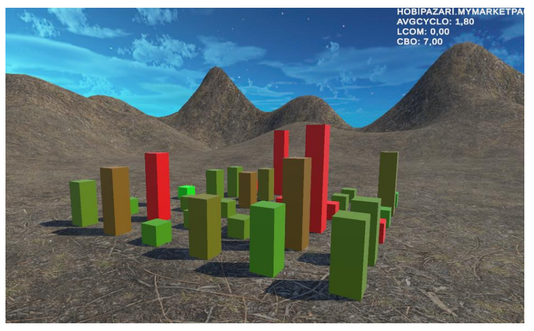

Figure 1. Visualization.

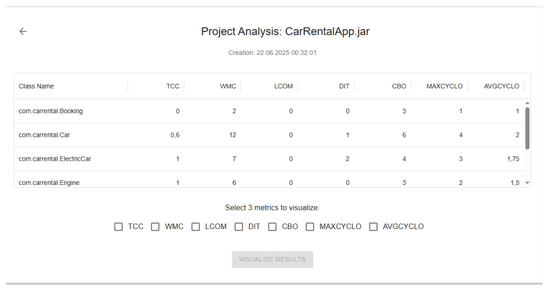

Figure 2. Metric Result

Conclusion

This project has made it possible to analyze software metrics not only through numerical data but also through three-dimensional visual representations. Especially in large and complex software systems, it provides a powerful alternative to traditional text-based analysis methods by enabling architectural problems to be identified intuitively. Through visualization, developers can identify quality issues at the class level more quickly, significantly simplifying sustainability, restructuring, and maintenance processes.

The application can be effectively used in quality control processes, code review meetings, and software architecture analysis for training purposes by enterprise software teams. In addition, project managers can measure the refactoring performance of teams by comparing past analysis records over time. It also contributes to the more concrete and understandable transfer of abstract concepts such as cohesion, complexity, and dependency in the context of software engineering education.

However, the current system has been developed specifically for the Java language and compiled bytecode structure. Software written in different languages such as C# and Python is not supported. Furthermore, the inability of some metrics to reflect dynamic behavior is one of the limitations of visualization. In very large projects, visual performance may occasionally weaken, requiring optimization.

In future work, support for different programming languages can be added to enable cross-platform analysis. The range of metrics can be expanded to enable more in-depth quality analysis (e.g., MPC, FAN-IN/OUT). Integration with GitHub can enable evolutionary analysis based on commit history and automatic tracking of code change–quality relationships. Additionally, intelligent modules such as “code smell prediction” supported by machine learning can provide developers with recommendation-based improvements. The visualization infrastructure can be enhanced with new-generation technologies like augmented reality, WebGL, or mobile platforms, enabling software quality analysis not only on desktops but also across various environments and devices.

References

[1] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060–1076.
[2] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6), 476–493.
[3] Briand, L. C., Daly, J. W., & Wüst, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, 25(1), 91–121.
[4] Lanza, M., & Ducasse, S. (2003). Polymetric views—A lightweight visual approach to reverse engineering. IEEE Transactions on Software Engineering, 29(9), 782–795.
[5] Wettel, R., & Lanza, M. (2008). CodeCity: 3D visualization of large-scale software. In Proceedings of the 30th International Conference on Software Engineering (ICSE) (pp. 921–922). IEEE.