УДК 004.415.2:004.42

Ю.В. КУЄВДА, Є.О. ТАТАРІН, Ю.М. СЕЛІН
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

СИСТЕМА ОЦІНЮВАННЯ ЯКОСТІ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМНОГО КОДУ НА ОСНОВІ СТАТИЧНОГО АНАЛІЗУ ТА ВІЗУАЛІЗАЦІЇ МЕТРИК

Статтю присвячено розробленню системи статичного аналізу програмного коду, яка об'єднує аналіз, візуалізацію та інтерпретацію метрик якості програмного забезпечення. Актуальність дослідження зумовлена недостатньою ефективністю існуючих інструментів у забезпеченні комплексної оцінки коду та виявлення проблемних ділянок на ранніх етапах розроблення. Більшість доступних рішень не пропонують інтерактивної візуалізації результатів, що ускладнює контроль якості та підтримку архітектурних рішень.

Метою дослідження є створення системи Code Analyzer, яка здійснює статичний аналіз коду, надає інтерактивну візуалізацію та допомагає у розумінні метрик якості коду. Ця система сприяє своєчасному виявленню проблем і покращенню прийняття рішень архітекторами програмного забезпечення. Поточна версія підтримує мову Python та обробляє різні групи метрик: загальні метрики, метрики Холстеда, метрики складності й підтримуваності, а також об'єктно-орієнтовані та архітектурні метрики.

Code Analyzer використовує бібліотеки Radon для базових метрик, аналіз абстрактного синтаксичного дерева (AST) для більш глибокого аналізу, а також розроблені алгоритми для обчислення специфічних метрик, таких як RFC, LCOM, Ca, Ce. Система надає користувачу можливість завантажити код у форматах .py або .zip і в результаті отримати візуалізацію метрик у вигляді графіків, таблиць та рейтингів. Програмна система може експортувати звіти у формати CSV та Excel для подальшого аналізу.

Одним із ключових компонентів системи є інтерактивна візуалізація результатів, яка дає змогу зручно аналізувати складність коду та виявляти потенційно проблемні ділянки. Візуалізація надає не лише кількісні метрики, а й інтерпретує їх значення для покращення процесу рефакторингу коду та прийняття рішень. Інтерфейс дає змогу сортувати та фільтрувати дані, що забезпечує ефективне управління великими проєктами. Окрім того, система пропонує інтерпретацію результатів через радарні графіки та дає конкретні рекомендації щодо покращення якості коду.

Code Analyzer реалізує комплексний підхід до статичного аналізу та візуалізації метрик якості програмного забезпечення, що дає змогу знижувати ризики помилок, оптимізувати процес розроблення та підтримки коду, а також підвищувати ефективність командної роботи в рамках великих проєктів.

Ключові слова: *якість програмного коду, статичний аналіз, візуалізація метрик, об'єктно-орієнтоване програмування.*

IU.V. KUIEVDA, YE.O. TATARIN, YU.M. SELIN
National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute»

QUALITY ASSESSMENT SYSTEM FOR OBJECT-ORIENTED PROGRAM CODE BASED ON STATIC ANALYSIS AND METRICS VISUALIZATION

The article is devoted to the development of a system for static analysis of program code that integrates analysis, visualization, and interpretation of software quality metrics. The relevance of the research is determined by the insufficient efficiency of existing tools in providing a comprehensive code assessment and identifying problematic areas at the early stages of development. Most available solutions do not offer interactive visualization of results, which complicates quality control and the maintenance of architectural decisions.

The aim of the research is to develop a system named Code Analyzer, which performs static code analysis, provides interactive visualization, and assists in understanding code quality metrics. This system facilitates the timely detection of issues and supports software architects in decision-making. The current version supports the Python language and processes various groups of metrics: general metrics, Halstead metrics, complexity and maintainability metrics, as well as object-oriented and architectural metrics.

Code Analyzer uses the Radon library for basic metrics, Abstract Syntax Tree (AST) analysis for deeper inspection, and custom algorithms for calculating specific metrics such as RFC, LCOM, Ca, and Ce. The system allows users to upload code in .py or .zip formats and obtain visualized metric results in the form of graphs, tables, and ratings. The program can also export reports in CSV and Excel formats for further analysis.

One of the key components of the system is the interactive visualization of results, which enables convenient anal-

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

ysis of code complexity and identification of potentially problematic areas. Visualization provides not only quantitative metrics but also interprets their meaning to improve the processes of refactoring and decision-making. The interface allows data sorting and filtering, ensuring efficient management of large-scale projects. In addition, the system provides interpretation of results using radar charts and offers specific recommendations for improving code quality.

Code Analyzer implements a comprehensive approach to static analysis and software quality metrics visualization, which helps reduce error risks, optimize code development and maintenance processes, and increase the efficiency of teamwork within large projects.

Key words: *software code quality, static analysis, metrics visualization, object-oriented programming.*

Постановка проблеми

Необхідність розроблення системи для статичного аналізу програмного коду виникає через обмеженість існуючих інструментів у забезпеченні комплексної оцінки якості коду та своєчасного виявлення проблемних ділянок на ранніх етапах розроблення. Більшість сучасних рішень не пропонує інтерактивної візуалізації метрик або можливості для гнучкої інтерпретації результатів, що ускладнює ефективний контроль за кодовою базою. Це може призводити до труднощів у підтримці архітектурних рішень та вчасному реагуванні на потенційні проблеми, що, своєю чергою, негативно впливає на стабільність та якість програмного забезпечення. Відсутність адекватних інструментів підтримки прийняття рішень для архітекторів програмних систем створює потребу в новому рішенні, яке поєднує статичний аналіз, візуалізацію та інтерактивну допомогу в інтерпретації метрик якості.

Аналіз останніх досліджень і публікацій

Система статичного аналізу коду базується на основі розрахунку, візуалізації та аналізу декількох груп метрик статичного аналізу програмного коду, а саме: загальних метрик, метрик Холстеда (Halstead), метрик складності та підтримуваності, а також ООП та архітектурних метрик. Ці набори метрик були введені та досліджені в різні роки розвитку програмування, але не втрачають актуальності й сьогодні [1]. Розглянемо більш детально призначення та бажані значення цих метрик.

Група **загальних метрик** включає базові кількісні характеристики програмного коду. Ці прості метрики з'явилися найпершими в історії, у 1960–1970-х роках [2]. Серед них:

– *LOC* (Lines of code) – загальна кількість рядків коду, включаючи порожні рядки та коментарі, ця метрика вказує на загальний розмір файлу;

– *LLOC* (Logical Lines of Code) – кількість логічних рядків коду, виключаючи коментарі та порожні рядки, метрика дає точніше уявлення про обсяг виконуваної роботи;

– відсоток коментарів (Comment Ratio) – показує частку коментарів у коді відносно загальної кількості рядків, не повинен бути ні високим, ні занадто малим, оптимальний варіант – це середній рівень коментування;

– кількість функцій – відображає кількість функцій у кодовій базі, велика кількість функцій може вказувати на високу деталізацію або, навпаки, на надмірну фрагментацію, що потребує подальшого аналізу;

– кількість класів – відображає кількість класів у кодовій базі, велика кількість класів може свідчити про складну архітектуру, слід оцінити, чи не є їх кількість надмірною.

Метрики Холстеда (Halstead metrics) – це набір програмних метрик, розроблених Морісом Х. Холстедом у 1977 р. [1], які дотепер використовуються для кількісної оцінки складності та зусиль, необхідних для розроблення програмного забезпечення. Вони базуються на кількості унікальних операторів та операндів у коді. Базовими елементами є унікальні оператори ($h1$) та їх загальна кількість ($N1$), а також унікальні операнди ($h2$) та їх загальна кількість ($N2$), вони формують основу для всіх метрик Холстеда:

– довжина програми (N) є сумою всіх операторів та операндів $N – N1+N2$;

– словник програми (n) – кількість унікальних операторів та операндів $n – h_1 + h_2$;

– об'єм програми (V) відображає інформаційний розмір коду $V – N \cdot log_2(n)$;

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

– кількість імовірних багів (*B*) є прогнозом кількості помилок у коді, ця метрика містить емпіричну оцінку якості програми *D – V/3000*;

– складність програми (*D*) вказує на зусилля для розуміння коду $D = \left(\frac{h1}{2}\right) \cdot \left(\frac{N2}{h2}\right)$;

*** зусилля розроблення (*E*) та час, необхідний для реалізації (*T*), оцінюють трудомісткість розроблення $E = D \cdot V$, $T = E/18$.

Метрики складності та підтримуваності:

– цикломатична складність (*CC*) – була введена Thomas McCabe [3] та вимірює кількість незалежних шляхів у коді функції, високе значення *CC* (більше 10–15) указує на складну логіку, що може ускладнити тестування та розуміння;

– індекс підтримуваності (*MI*, Maintainability Index) – це композитна метрика, запропонована Oman and Hagemeister [4], яка показує, наскільки легко підтримувати та змінювати код, і базується на метриках *CC*, *LLOC* та *V*. Вона має багато різновидів формули обчислення, одна з них така:

$$MI = \max\left(0, \left(171 - 5.2 * log_2(V) - 0.23 * CC - 16.2 * log_2(LLOC)\right) * \frac{100}{171}\right).$$

Високе значення *MI* (ближче до 100) означає, що код легше читати, розуміти та змінювати, що свідчить про його високу якість та підтримуваність.

ООП-метрики оцінюють якість об'єктно-орієнтованого дизайну [5]. Вони допомагають виявити проблеми з когезією, зв'язністю та ієрархією класів:

– *CBO* (Coupling Between Objects) – кількість інших класів, із якими взаємодіє даний клас. Високий CBO вказує на сильну зв'язність, що може ускладнити зміни;

– *RFC* (Response For a Class) – кількість методів, які можуть бути викликані у відповідь на повідомлення, надіслане об'єкту класу. Високий *RFC* може свідчити про надмірну зв'язність класу;

– *DIT* (Depth of Inheritance Tree) – глибина ієрархії успадкування. Глибокі ієрархії можуть ускладнити розуміння та тестування коду;

– *LCOM* (Lack of Cohesion in Methods) – вимірює відсутність когезії у методах класу. Діапазон *LCOM* від 0 до 1. Низьке значення *LCOM* бажане, оскільки вказує на те, що методи класу тісно пов'язані між собою

$$LCOM = 1 - \frac{\textit{кількість пар методів зі спільними атрибутами}}{\textit{загальна кількість пар методів}};$$

– *NOC* (Number of Children) – кількість безпосередніх підкласів для даного класу. Високий *NOC* може свідчити про невідповідне використання успадкування або про занадто загальний дизайн класу;

– *WMC* (Weighted Method Count) – вимірює складність класу шляхом підсумовування цикломатичних складностей усіх його методів

$$WMC = \sum_{i=1}^{n} CC_i,$$

де CC_i – цикломатична складність окремої функції (методу) у класі, n – кількість функцій (методів) у класі. Нижчий *WMC* указує на меншу складність класу та кращу структурованість методів, високий *WMC* може свідчити про те, що клас є надто складним і може потребувати рефакторингу.

Архітектурні метрики [6] оцінюють якість структури системи на рівні модулів та компонентів. Вони допомагають забезпечити стабільність та гнучкість архітектури.

– *Ca* (Afferent Couplings) – кількість класів за межами модуля, які залежать від класів всередині цього модуля. Високий *Ca* означає, що модуль є дуже залежним.

– *Ce* (Efferent Couplings) – кількість класів усередині модуля, які залежать від класів за межами цього модуля. Високий *Ce* означає, що модуль сильно залежить від інших.

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

– *Abstractness (A)* – визначає ступінь абстрактності компонентів. Показує гнучкість архітектури до розширення без змін. Висока абстрактність сприяє адаптивності.

$$A = \frac{\text{кількість абстрактних класів}}{\text{кількість усіх класів}}.$$

– *Instability (I)* – оцінює стабільність модуля. Вище значення вказує на часті зміни, що потенційно впливають на інші частини системи.

$$I = \frac{Ce}{Ca + Ce}.$$

– *Distance (D)* – відстань до ідеального балансу абстрактності (A) та нестабільності (I). Значення близько до нуля є бажаним, що вказує на збалансовану архітектуру.

$$D = |A + I - 1|.$$

Мета дослідження

Метою дослідження є проведення аналізу предметної області, проєктування архітектури та розроблення системи, яка не лише здійснює статичний аналіз програмного коду, а й надає інтерактивну візуалізацію та інтерпретацію широкого спектру статичних метрик якості (далі – Code Analyzer). Така система сприятиме підвищенню ефективності контролю за станом кодової бази проєкту, забезпечуватиме своєчасне виявлення можливих проблемних ділянок на початкових етапах розроблення, а також стане інструментом для підтримки процесу прийняття рішень архітекторами програмного забезпечення.

Виклад основного матеріалу дослідження

Сьогодні Code Analyzer підтримує статичний аналіз коду на мові Python. Архітектуру програмної системи представлено на рис. 1. Code Analyzer забезпечує ефективний аналіз коду шляхом функціонування за таким ланцюгом обробки: завантаження даних – користувач ініціює процес, завантажуючи файл у форматі .py або .zip для аналізу; обробка та аналіз – система автоматично аналізує завантажені модулі коду, обчислюючи метрики складності, підтримуваності та архітектури; відображення результатів – після обробки, результати візуалізуються у вигляді інтерактивних графіків, таблиць, рейтингової оцінки та рекомендацій; експорт звітів – користувачі можуть експортувати повні звіти з аналізом у зручних форматах, таких як CSV- та Excel-таблиці, для подальшого використання.

Архітектура системи (рис. 1) складається з чотирьох взаємопов'язаних рівнів. На рівні користувацького інтерфейсу (User Interface) реалізовано вебсторінки та скрипти для завантаження .py- та .zip-файлів і візуалізації результатів. Рівень Web-Server&Routing (app.py, metrics_analyser.py) відповідає за маршрутизацію запитів і передачу даних до ядра системи. Основні аналітичні модулі (Analysis Modules) виконують обчислення різних метрик програмного коду включно зі статистичними, структурними та об'єктно-орієнтованими показниками. Отримані результати експортуються у формати CSV або Excel, що забезпечує подальше використання даних для спеціального аналізу.

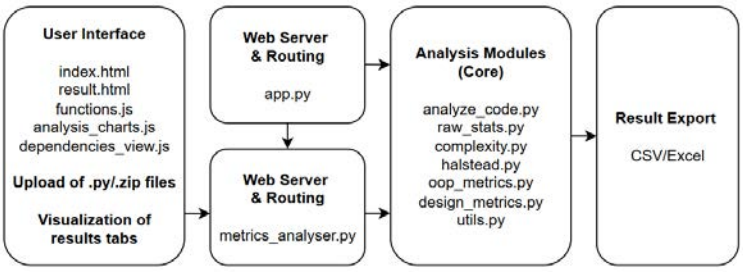

Рис. 1. Архітектура програмної системи статичного аналізу коду Code Analyzer

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

У процесі розроблення Code Analyzer ключове значення мала ефективна інтеграція з інструментами статичного аналізу [1], оскільки саме вона відкриває можливість отримання як кількісних, так і структурних показників програмного коду. Замість використання одного готового рішення було вибрано підхід, що об'єднує три різні джерела аналізу: бібліотеку Radon [7], безпосередню роботу з AST (Abstract Syntax Tree) [8], а також власноруч реалізовані метрики, що базуються на аналізі синтаксичних структур. Така комбінація дала змогу охопити ширший спектр метрик, включаючи об'єктно-орієнтовані та архітектурні, а також адаптувати систему до масштабування під різні Python-проєкти.

Інструмент Radon забезпечує високу продуктивність та зручність у застосуванні, однак його функціональність обмежується здебільшого базовими метриками. Натомість вбудований механізм аналізу синтаксичного дерева (AST) надає більш гнучкі можливості для точного дослідження структури коду, що є особливо важливим під час обробки об'єктно-орієнтованих та архітектурних характеристик. Розроблені алгоритми, що не залежать від сторонніх бібліотек, поєднують точність AST-аналізу з можливістю адаптації під специфічні цілі, наприклад для обчислення таких метрик, як *RFC* або *LCOM*, у контексті нестандартних архітектурних патернів.

Інтеграція з Radon реалізована двома шляхами: через пряме використання його API (наприклад, analyze, cc_visit) для отримання структурованих Python-об'єктів, а також через виклики командного інтерфейсу CLI, що дає змогу запускати аналіз на окремих модулях або пакетах без зміни основного коду. За допомогою Radon виконуються обчислення таких показників: *LOC*, *LLOC*, кількість і відсоток коментарів – через radon.raw.analyze; індекс підтримуваності (*MI*) – за допомогою radon.metrics.mi_visit; цикломатична складність – через radon.complexity.cc_visit. Ці значення становлять основу загального статистичного профілю коду, який подається користувачеві через візуальні елементи: шкали, діаграми та текстові блоки.

Для більш глибокого аналізу, включаючи метрики Холстеда, ООП-показники (*RFC*, *CBO*, *LCOM*) та архітектурні залежності (*Ca*, *Ce*, *Instability*), використовується парсинг абстрактного синтаксичного дерева за допомогою ast.parse(). Далі система рекурсивно обходить дерево, отримуючи повну картину щодо: структури класів та методів; ієрархій спадкування; використання атрибутів; міжкласових викликів; імпортів та зовнішніх залежностей.

Зокрема, *LCOM* визначається на основі перетину змінних екземпляра, які використовуються в методах; RFC ураховує кількість методів та зовнішніх викликів зсередини класу; *DIT*/*NOC* формуються через побудову дерева спадкування; *Ca*/*Ce* обчислюються шляхом аналізу імпортів (import, from ... import), що дає змогу визначити напрямки залежностей.

Цей підхід забезпечує гнучке налаштування обробки: можна ігнорувати службові методи, визначати типи зв'язків між компонентами або будувати графи залежностей, які відображаються в інтерфейсі за допомогою Vis.js на вкладці «Залежності».

Архітектурне рішення Code Analyzer передбачає поєднання переваг зовнішніх бібліотек та внутрішньої логіки, що дає змогу як охопити широкий набір метрик, так і оперативно адаптувати механізми обробки до змінних вимог, зберігаючи сумісність та розширюваність системи.

Під час первинного завантаження проєкту користувач працює через вебінтерфейс (файл index.html), де може вибрати для аналізу окремий Python-файл або архів формату .zip, що містить повну структуру проєкту. Після того як файли завантажено, серверна частина системи, реалізована за допомогою Flask (файл app.py), передає отримані дані до модуля обробки. У разі якщо передано архів, він автоматично розпаковується, після чого запускається рекурсивний пошук усіх .py-файлів, при цьому ігноруються технічні каталоги на кшталт venv, __pycache__ та .git. Уся логіка цього етапу зосереджена в окремому модулі metrics_analyser.py.

Після попередньої обробки кожен знайдений файл передається до модуля analyze_code.py, який виступає центральним елементом системи обчислення метрик. Його логіка побудована як послідовність кроків.

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

На першому етапі вміст файлу зчитується як текстовий рядок, після чого виконується побудова абстрактного синтаксичного дерева (AST) через конструкцію tree = ast.parse(code). У результаті формується повна структура коду у вигляді дерева, що охоплює класи, функції, виклики, наслідування, імпорти тощо. Використання AST забезпечує можливість детального структурного та семантичного аналізу без потреби виконання програми, що критично важливо для статичного аналізу.

На другому етапі для обчислення базових метрик на рівні окремого файлу (наприклад, кількість рядків коду чи індекс підтримуваності) застосовується функція radon.raw.analyze(), яка надає такі показники: *LOC*, *LLOC*, *Comment Ratio*, *MI*.

Усі отримані результати формуються у вигляді словників Python і передаються далі для агрегації й візуалізації. Ілюстрацію прикладу загального аналізу проєкту наведено на рис. 2.

Для аналізу цикломатичної складності функцій у програмному коді застосовується метод cc_visit() з модуля radon.complexity. Цей інструмент формує перелік об'єктів, кожен з яких містить оцінку складності для конкретної функції або методу. Окрім індивідуальних значень, ці дані дають змогу обчислити інтегральний показник *WMC* – сумарну цикломатичну складність для всього модуля. Отримані результати використовуються у візуалізації (зокрема, у вигляді гістограм) та для впорядкування функцій за рівнем складності.

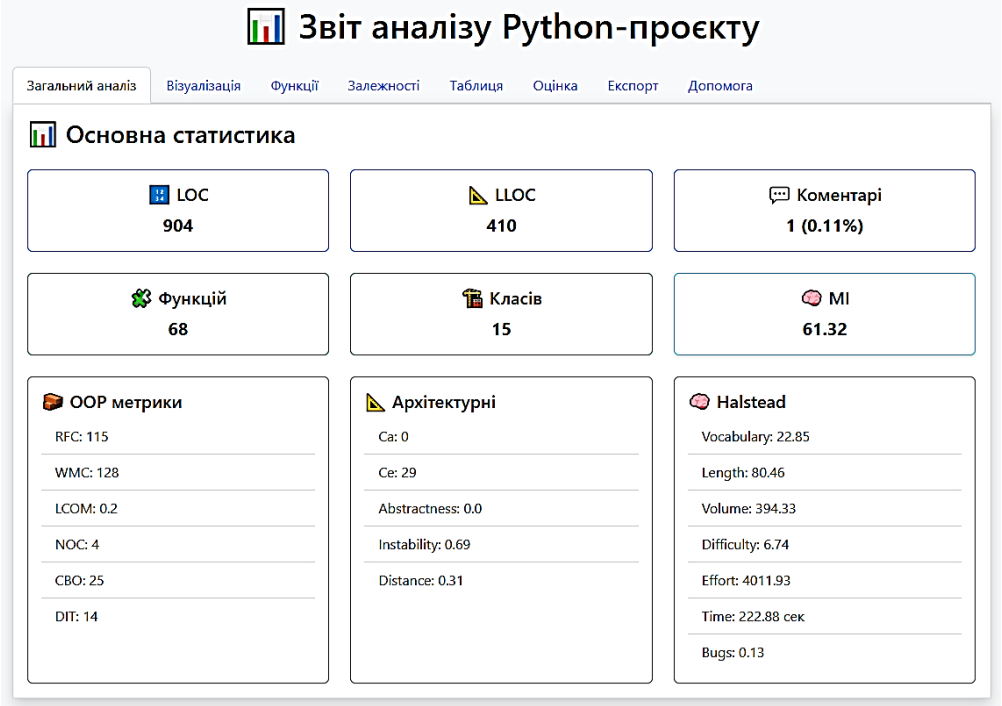

Рис. 2. Приклад загального аналізу проєкту за допомогою Code Analyzer

Обчислення метрик Холстеда реалізовано за допомогою окремого модуля halstead.py, який аналізує абстрактне синтаксичне дерево (AST) файлу. Цей аналіз включає: визначення кількості та унікальних типів операторів і операндів; розрахунок *Volume* як оцінки інформаційного обсягу; показник *Effort*, що інтерпретується як зусилля, необхідні для розуміння коду; прогностичне оцінювання кількості можливих помилок (*Bugs*). Ці метрики дають змогу оцінити складність не лише з погляду логіки виконання, а й із погляду когнітивного навантаження на розробника.

Для отримання ООП-метрик застосовується модуль oop_metrics.py, який також працює з AST. У результаті такого аналізу визначаються такі показники: *RFC*, *LCOM*, *CBO*, *NOC* та *DIT*. Зокрема, для оцінки зв'язності (*CBO*) аналізуються конструкції import і from ... import, що дає змогу визначити зовнішні залежності класу.

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

На рівні модулів система проводить архітектурний аналіз за допомогою модуля design_metrics.py. Побудову графа залежностей між модулями базовано на обчисленні таких характеристик: *Ca*, *Ce*, *Instability*, *Abstractness* та *Distance*.

Усі зібрані метрики зберігаються у структурованому форматі. Граф залежностей інтерактивно візуалізується за допомогою JavaScript-бібліотеки Vis.js у скрипті dependencies_view.js. Далі ці дані передаються в модуль metrics_analyser.py, який відповідає за обчислення агрегованих показників, формування візуалізацій через analysis_charts.js, а також присвоєння фінальної оцінки якості у вигляді літерної шкали (від A до D).

Поєднання автоматизованого аналізу AST, інтеграції з бібліотекою Radon та власних алгоритмічних рішень забезпечує системі Code Analyzer можливість здійснювати комплексну оцінку якості програмного забезпечення – від аналізу окремих функцій до вивчення архітектурних залежностей між модулями. Такий рівень деталізації робить інструмент універсальним рішенням як для індивідуального використання, так і для команд, які потребують регулярного технічного аудиту коду.

Однією з ключових характеристик Code Analyzer є структурована та інформативна візуалізація метрик програмного коду. Застосування графіків, таблиць і показників дає змогу не лише здійснювати кількісну оцінку технічного стану програмного забезпечення, а й своєчасно виявляти проблемні ділянки без необхідності безпосереднього аналізу сирцевого коду. Візуальна репрезентація результатів відіграє роль інтерпретаційного інструмента, який трансформує аналітичні дані у практичну відповідь на запитання: що саме, де саме і з якої причини потребує оптимізації.

Інтерфейс аналітичної системи реалізовано за допомогою HTML-шаблонів, створених на базі Jinja2, у поєднанні з JavaScript-бібліотеками Chart.js та Vis.js. Основна логіка візуалізації реалізується через три окремі скрипти, що відповідають за побудову графіків метрик, реалізацію механізмів фільтрації та сортування функцій, а також формування графів залежностей між програмними модулями.

Усі візуальні компоненти системи об'єднані на сторінці result.html, яка містить вісім функціональних вкладок: загальний огляд, візуалізація, функції, залежності, таблиця, оцінка, експорт та довідкова інформація.

Гістограми та лінійні графіки, представлені на рис. 3, використовуються для ілюстрації розподілу таких метрик, як *MI*, *Halstead Volume* та *WMC*, у межах окремих функцій або модулів. Кожен елемент графіка відповідає окремій функціональній одиниці, а кольорова диференціація використовується для кодування рівня потенційного ризику: зелений колір відповідає допустимому рівню складності, жовтий указує на необхідність додаткової перевірки, а червоний сигналізує про критичний рівень складності або підтримуваності. Графіки є інтерактивними, під час наведення на елемент графіка з’являється спливаюче вікно, у якому відображаються назва та значення вибраної метрики, а також назва модуля.

Застосування такого типу візуалізації значно полегшує процес виявлення атипових значень серед великої кількості функцій. Зокрема, сегменти з підвищеним значенням метрик, наприклад *Halstead Volume*, легко ідентифікуються завдяки яскравим візуальним ознакам – змінам кольору або різкій зміні ширини сектора або висоти стовпця на діаграмі.

У вкладці «Функції» представлено детальну інформацію щодо кожної окремої функції програмного коду, зокрема її назву, кількість рядків коду (*LOC*), цикломатичну складність (*CC*), індекс підтримуваності (*MI*), кількість потенційних помилок (*Bugs*) та *Halstead Volume*. Уміст таблиці автоматично візуально маркується залежно від значень показників: комірки змінюють колір відповідно до рівня складності чи ризику. Подібне подання даних значно спрощує процес виявлення функцій, які потенційно є складними для тестування, модифікації або супроводу, даючи змогу швидко локалізувати найбільш проблемні ділянки коду.

Інтерактивна вкладка «Залежності» (рис. 4) забезпечує візуальне подання модульної структури проєкту у вигляді графа. У цій візуалізації кожен модуль репрезентовано у формі

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

вузла, тоді як зв’язки імпорту між модулями відображаються спрямованими стрілками. Такий підхід дає змогу ефективно аналізувати архітектуру програмної системи, зокрема виявляти ділянки з надмірним рівнем залежності, ідентифікувати критичні компоненти (модулі з великою кількістю вхідних зв’язків) та розпізнавати модулі, які самі по собі є залежними, але не використовуються іншими.

Завдяки інтерактивності графа користувач може досліджувати структуру проєкту в динаміці, що сприяє глибшому розумінню взаємозв’язків між компонентами та полегшує прийняття архітектурних рішень.

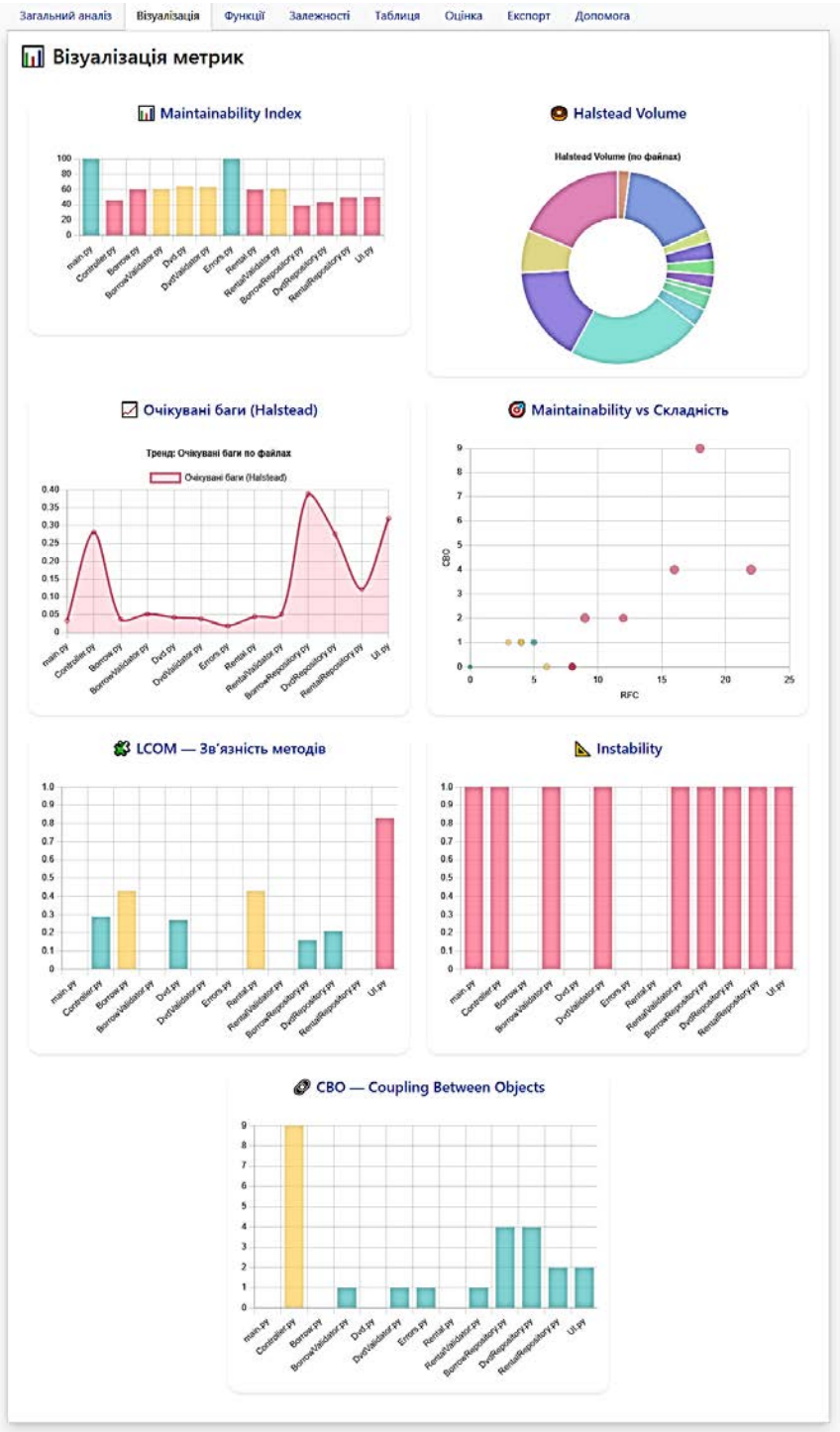

Рис. 3. Візуалізація метрик у Code Analyzer

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

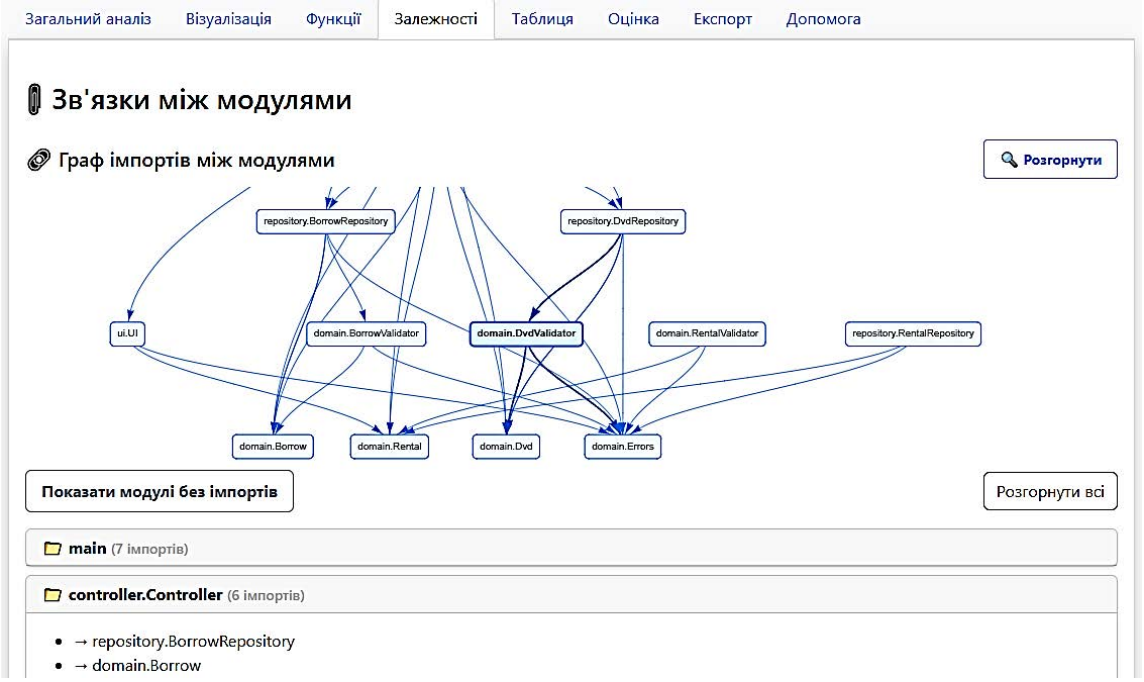

Рис. 4. Інтерактивна візуалізація «Залежності» у Code Analyzer

У вкладці «Таблиця» можна побачити всі метрики у розрізі окремих файлів. Застосування стандартних HTML-таблиць із вбудованою підтримкою JavaScript-сортування (реалізованого у скрипті functions.js) забезпечує гнучке й ефективне опрацювання даних про функції. Інтерфейс дає змогу виконувати сортування за будь-яким із доступних параметрів, таких як індекс підтримуваності (*MI*), цикломатична складність (*CC*), кількість викликів (*RFC*) тощо. Окрім того, реалізовано можливість фільтрації функцій за заданими критеріями, зокрема для виділення критичних ділянок коду (наприклад, функції з $MI < 50$ або $WMC > 10$). Кожен запис у таблиці має гіперпосилання або ідентифікатор, що дає змогу швидко перейти до відповідного фрагмента сирцевого коду для подальшого аналізу. Варто зазначити, що таблиці формуються динамічно, що забезпечує масштабованість підходу й дає змогу ефективно працювати з великими проєктами без утрати продуктивності.

Для підсумкової інтерпретації результатів аналізу застосовуються радарні графіки на вкладці «Оцінка» (рис. 5). Кожна вісь такого графіка відповідає окремому показнику якості програмного коду, зокрема MI, кількості потенційних помилок (*Bugs*), когезії (*LCOM*), зв’язності (*CBO*), обсягу (*Halstead Volume*) тощо. Центральна точка графіка позначає гранично критичне значення (0), тоді як зовнішній контур відображає ідеальний стан (1). Цей формат візуалізації є особливо ефективним для комплексної оцінки коду. Він забезпечує швидке виявлення слабких місць у структурі метрик, даючи змогу візуально виділити ті з них, де значення відхиляються від прийнятних меж і потребують уваги з боку розробника або аналітика.

Також на цій вкладці формується сумарна оцінка коду у вигляді літерної шкали A–D та окремі рекомендації щодо значень метрик на базі наступних умов: $MI < 60$ – низька підтримуваність, покращити читабельність і коментування коду; $WMC > 50$ – висока цикломатична складність, спростити логіку методів, розбити великі функції; *Instability* > 0.7 – надмірна залежність модулів, зменшити кількість зовнішніх імпортів; $CBO > 20$ – надмірна зв’язаність класів, зменшити зв’язаність між класами, переглянути залежності; *Halstead Volume* > 1000 – висока складність модулів, розбити великі модулі на менші, зменшити обсяг коду; *Halstead Bugs* > 0.5 – високий ризик помилок у коді, перевірити найскладніші ділянки коду, провести додаткове тестування.

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

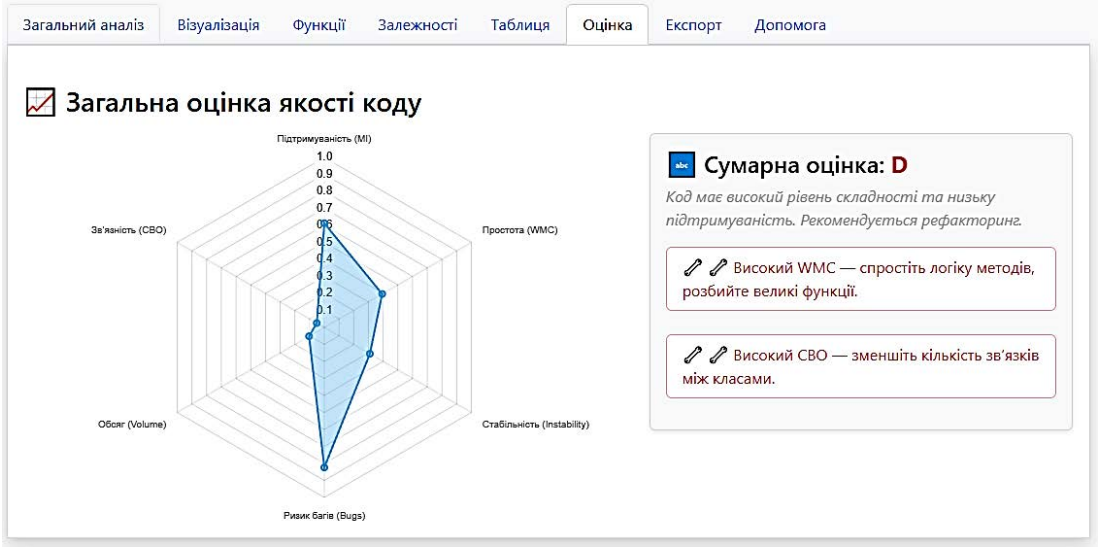

Рис. 5. Інтерактивна візуалізація «Залежності» у Code Analyzer

Також у Code Analyzer на вкладці «Експорт» реалізовано можливість експорту результату у формати CSV та Excel та додано розділ «Допомога» з описом усіх використаних у системі метрик.

У цілому побудова візуальних компонентів у системі Code Analyzer базується на прагненні до оптимального поєднання інформативності, швидкодії та візуальної привабливості. Такий підхід забезпечує не лише наочне представлення кількісних характеристик програмного коду, а й сприяє глибшому розумінню їх практичного значення. Це, своєю чергою, відповідає фундаментальному завданню будь-якої системи оцінювання якості програмного забезпечення – надати користувачеві інструменти для усвідомленого прийняття рішень щодо підтримки, рефакторингу або модернізації коду.

Висновки

У результаті дослідження проведено аналіз предметної області, спроєктовано архітектуру та розроблено систему Code Analyzer для статичного аналізу програмного коду, яка поєднує у собі аналіз широкого набору метрик якості та інтерактивну візуалізацію результатів. Завдяки інтеграції з бібліотеками Radon, а також механізмам обробки абстрактного синтаксичного дерева (AST) система забезпечує комплексну оцінку програмного коду, охоплюючи такі важливі метрики, як складність, підтримуваність, архітектурні залежності та об’єктно-орієнтовані показники. Візуалізація результатів у вигляді графіків, таблиць і діаграм дає змогу ефективно оцінювати стан коду, зосереджуючи увагу на найбільш проблемних ділянках і забезпечуючи своєчасне реагування на потенційні проблеми.

Подальші дослідження можуть бути спрямовані на підтримку інших мов програмування, покращення точності метрик для складних архітектурних патернів, а також на інтеграцію з іншими інструментами для автоматичного тестування та безпеки програмного забезпечення. Окрім того, доцільним є впровадження алгоритмів машинного навчання для прогнозування можливих дефектів на основі історії змін у коді.

Розроблення Code Analyzer надає нові можливості для програмістів та архітекторів програмного забезпечення у процесі підтримки, рефакторингу та оптимізації коду. Система дає змогу не лише здійснювати технічний аудит, а й активно підтримувати процес прийняття рішень на всіх етапах розроблення, покращуючи якість програмних продуктів і знижуючи ризик виникнення помилок у коді.

https://doi.org/10.32782/mathematical-modelling/2025-8-2-16 ISSN 2618-0332 (print), 2618-0340 (online)

Список використаної літератури

1. Ardito L., Coppola R., Barbato L. Verga D. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming*. 2020. № 2020(1). P. 1–26. DOI: https://doi.org/10.1155/2020/8840389
2. Rashid J., Mahmood T., Nisar M.W. A Study on Software Metrics and its Impact on Software Quality. *Technical Journal, University of Engineering and Technology (UET) Taxila, Pakistan*. 2019. № 24(1). P. 1–14. DOI: https://doi.org/10.48550/arXiv.1905.12922
3. Kafura D. Reflections on McCabe's Cyclomatic Complexity. *IEEE Transactions on Software Engineering*. 2025. № 51(3). P. 700–705. DOI: https://doi.org/10.1109/TSE.2025.3534580
4. Heričko T., Šumak B. Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Applied Sciences*. 2023. № 13(5). P. 2972. DOI: https://doi.org/10.3390/app13052972
5. Filó T.G.S., Bigonha M.A.S., Ferreira K.A.M. Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society*. 2024. № 30(1). P. 313–346. DOI: https://doi.org/10.5753/jbcs.2024.3373
6. Santos D., Resende A., de Castro Lima, E., Freire A. Software Instability Analysis Based on Afferent and Efferent Coupling Measures. *Journal of Software*. 2017. № 12(1). P. 19–34. DOI: https://doi.org/10.17706/jsw.12.1.19-34
7. Radon Project. Radon: Code metrics in Python. PyPI. URL: https://pypi.org/project/radon/ (дата звернення: 15.09.2025).
8. Python Software Foundation. AST – Abstract Syntax Trees. In Python documentation (version 3). URL: https://docs.python.org/3/library/ast.html (дата звернення: 15.09.2025).

References

1. Ardito, L., Coppola, R., Barbato, L. & Verga, D. (2020). A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming*, 2020(1), 1–26. https://doi.org/10.1155/2020/8840389 [in English].
2. Rashid, J., Mahmood, T. & Nisar, M.W. (2019). A Study on Software Metrics and its Impact on Software Quality. *Technical Journal, University of Engineering and Technology (UET) Taxila, Pakistan*, 24(1), 1–14. https://doi.org/10.48550/arXiv.1905.12922 [in English].
3. Kafura, D. (2025). Reflections on McCabe's Cyclomatic Complexity. *IEEE Transactions on Software Engineering*, 51(3), 700–705. https://doi.org/10.1109/TSE.2025.3534580 [in English].
4. Heričko, T., & Šumak, B. (2023). Exploring Maintainability Index Variants for Software Maintainability Measurement in Object-Oriented Systems. *Applied Sciences*, 13(5), 2972. https://doi.org/10.3390/app13052972 [in English].
5. Filó, T.G.S., Bigonha, M.A.S. & Ferreira, K.A.M. (2024). Evaluating Thresholds for Object-Oriented Software Metrics. *Journal of the Brazilian Computer Society*, 30(1), 313–346. https://doi.org/10.5753/jbcs.2024.3373 [in English].
6. Santos, D., Resende, A., de Castro Lima, E. & Freire, A. (2017). Software Instability Analysis Based on Afferent and Efferent Coupling Measures. *Journal of Software*, 12(1), 19–34. https://doi.org/10.17706/jsw.12.1.19-34 [in English].
7. Radon Project (2025). Radon: Code metrics in Python. PyPI. Retrieved from https://pypi.org/project/radon/ [in English].
8. Python Software Foundation (2025). AST – Abstract Syntax Trees. In Python documentation (version 3). Retrieved from https://docs.python.org/3/library/ast.html [in English].

Куєвда Юлія Валеріївна – к.т.н., доцент кафедри математичних методів системного аналізу Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». E-mail: kuievda.iuliia@lll.kpi.ua, ORCID: 0009-0001-6630-1215.

Татарін Євгеній Олегович – бакалавр Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». E-mail: tatarine8@gmail.com, ORCID: 0009-0001-0452-2389.

Селін Юрій Миколайович – к.т.н., старший викладач кафедри математичних методів системного аналізу Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». E-mail: selinyurij1963@gmail.com, ORCID: 0000-0002-7562-8586.

Kuievda Iuliia Valeriivna – Candidate of Technical Sciences, Associate Professor at the Department of Mathematical Methods of System Analysis of the National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute». E-mail: kuievda.iuliia@lll.kpi.ua, ORCID: 0009-0001-6630-1215.

Tatarin Yevhenii Olehovych – Bachelor of the National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute». E-mail: tatarine8@gmail.com, ORCID: 0009-0001-0452-2389.

Selin Yurii Mykolaiovych – Candidate of Technical Sciences, Senior Lecturer at the Department of Mathematical Methods of System Analysis of the National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute». E-mail: selinyurij1963@gmail.com, ORCID: 0000-0002-7562-8586.

Отримано: 15.10.2025
Рекомендовано: 05.12.2025
Опубліковано: 30.12.2025