https://doi.org/10.31891/2219-9365-2025-83-17
УДК 004.056:004.852:004.75

ПРОКОФ'ЄВ Ігор
Хмельницький національний університет
https://orcid.org/0009-0005-6560-3587
e-mail: prokofjev.igor@gmail.com

МЕТОД СТАТИЧНОГО АНАЛІЗУ ЯКОСТІ КОДУ З ДОПОМОГОЮ МАШИННОГО НАВЧАННЯ

У статті представлено модель для оцінювання якості програмного коду на основі існуючих атрибутів якості. Проведено огляд основних методів статичного аналізу коду, а також їхніх обмежень. Якість програмного забезпечення може визначатися різними підходами, зокрема: ручним та автоматизованим тестуванням, рецензією коду розробниками, виявленням дублікатів, а також вимірюванням ключових метрик за допомогою статичних аналізаторів. Оцінка якості коду є складним завданням, яке потребує комплексного підходу. Для перевірки ефективності запропонованої моделі було проведено експеримент із використанням набору C# файлів з реальних програмних проєктів, що охоплюють різні рівні складності. Розроблений інструмент автоматично сканує файли у вибраному каталозі, обробляє їх та експортує результати у форматі CSV. Результати експерименту підтвердили ефективність обраного підходу: було успішно ідентифіковано складні фрагменти коду та потенційні проблеми в архітектурі програмного забезпечення. Отримані дані можуть бути корисними для розробників під час оптимізації та рефакторингу коду. У подальших версіях інструменту передбачено розширення переліку аналізованих показників та підвищення точності оцінювання. Крім того, планується інтеграція механізмів розпізнавання антишаблонів та аномалій у коді за допомогою методів машинного навчання.

Ключові слова: якість коду, атрибути якості коду

PROKOFIEV Ihor
Khmelnytskyi National University

METHOD OF STATIC CODE QUALITY ANALYSIS USING MACHINE LEARNING

This paper introduces a comprehensive model for assessing the quality of source code by leveraging a combination of established code quality attributes and modern analysis techniques. The study begins with an overview of the fundamental methods of static code analysis, outlining their capabilities as well as inherent limitations, and situates them within the broader context of software quality assurance practices. While software quality can be examined through multiple complementary approaches—such as manual inspection, automated unit and integration testing, peer developer code reviews, duplicate code detection, and metric-based evaluation via static analysis tools—none of these approaches alone is sufficient for a reliable and holistic evaluation. Instead, effective code quality assessment requires a multifaceted strategy that integrates diverse perspectives and tools.

The core contribution of this work lies in the design and experimental validation of a novel assessment tool. To verify its effectiveness, an empirical study was conducted using a dataset of C# source code files extracted from real-world software projects of varying scale and complexity. The tool automatically scans source files in a designated directory, processes them, and generates detailed reports in CSV format for further analysis. Experimental results demonstrated the ability of the model to successfully identify complex code fragments, redundant constructs, and potential architectural deficiencies. This not only provides actionable insights for developers but also supports informed decisions during refactoring and long-term codebase maintenance.

The results confirm that the proposed approach significantly enhances the accuracy and efficiency of code quality evaluation, offering advantages over traditional methods by combining structured analysis with extensibility. Future research will focus on expanding the range of supported metrics, improving the precision of quality assessments, and incorporating advanced detection of anti-patterns and anomalies through machine learning techniques. These enhancements are expected to further strengthen the applicability of the model in diverse software engineering contexts, making it a valuable resource for developers, project managers, and quality assurance teams alike.

Keywords: code quality, code quality attributes

Стаття надійшла до редакції / Received 19.07.2025
Прийнята до друку / Accepted 22.08.2025

ПОСТАНОВКА ПРОБЛЕМИ У ЗАГАЛЬНОМУ ВИГЛЯДІ ТА ЇЇ ЗВ'ЯЗОК ІЗ ВАЖЛИВИМИ НАУКОВИМИ ЧИ ПРАКТИЧНИМИ ЗАВДАННЯМИ

Якість коду відіграє вирішальну роль у формуванні розробки програмного забезпечення з 1990 року до сьогодні. На початку 1990-х років розробка програмного забезпечення зіткнулася з такими проблемами, як недостатня документація, недостатнє тестування та часті вразливості безпеки. З розвитком галузі такі найкращі методи, як автоматичне тестування, статичний аналіз коду та рекомендації щодо безпечного кодування, підвищили надійність і безпеку програмного забезпечення. Застосування гнучких методологій і DevOps додатково підкреслює безперервну інтеграцію та автоматизовану перевірку, покращуючи зручність обслуговування та зменшуючи кількість дефектів у виробництві. Сьогодні вдосконалені методи перевірки, включаючи статичний [1-2] і динамічний аналіз, допомагають завчасно виявляти проблеми, знижуючи ризики безпеки та витрати на розробку. Високоякісний код не тільки покращує сумісність системи, але й зміцнює

довіру клієнтів, підкреслюючи його важливість у сучасній розробці програмного забезпечення. Фреймворки та бібліотеки значно спрощують роботу програмістам та бізнесу і дозволяють сфокусуватись на вирішенню конкретних проблем без витрачання часу на інфраструктурні речі і наслідком є створення складних розподілених системи реального часу з мільйонами користувачів. В цьому контексті якість коду є не побічною проблемою а мейнстрімом, оскільки втрати якості ПЗ може відбутися внаслідок досить не суттєвих змін у коді що може призвести до повної зупинки критичних сервісів роботи.

Для виявлення якості коду недостатньо заміряти основні метрики коду оскільки зазвичай метрики за звичай не залежать одна від одної або залежать опосередковано. У даній роботі для оцінювання якості програмного коду запропоновано використання методу машинного навчання Random Forest (випадковий ліс). Цей метод дозволяє ефективно здійснювати класифікацію чи регресію на основі багатовимірних даних за рахунок побудови ансамблю незалежних рішень (дерев рішень), які в сукупності забезпечують високу точність і стійкість моделі до перенавчання.

АНАЛІЗ ДОСЛІДЖЕНЬ ТА ПУБЛІКАЦІЙ

Статичний аналізатор коду [1-2] — це програмне забезпечення, призначене для аналізу вихідного чи скомпільованого коду без його запуску. Ці інструменти перевіряють код під час розробки, щоб виявити потенційні проблеми, такі як помилки кодування, вразливості безпеки, порушення стилю та проблеми з якістю коду. Виконуючи аналіз на ранніх етапах життєвого циклу розробки програмного забезпечення (SDLC), статичний аналіз допомагає запобігти ескалації проблем, заощаджуючи час і гроші. Це також забезпечує дотримання стандартів кодування, таких як угоди про іменування, відступи та правила форматування. Сучасні статичні аналізатори, такі як SonarQube [3] і ESLint, також можуть вимірювати ключові показники, включаючи цикломатичну складність, індекс ремонтопридатності, дублювання коду та виявлення мертвого коду. Якість програмного коду не є статичною величиною; його можуть щодня змінювати десятки розробників, особливо при використанні програмного забезпечення, яким користуються мільйони людей, наприклад Gmail, Microsoft Office 365, X (раніше Twitter) і Netflix. Іншими словами, процес визначення якості коду є постійним, і кожна нова зміна коду ініціює нову ітерацію, починаючи з компіляції. Код повинен бути не тільки швидким, добре протестованим, але й здатним протистояти кібератакам [4]

Показники або атрибути якості програмного забезпечення зазвичай відповідають таким показникам якості коду:

- цикломатична складність (CC) [5]
- рядки коду (LOC)
- відсоток покриття коду автоматизованими тестами
- дублювання коду
- порушення правил статичного аналізу коду
- індекс ремонтопридатності
- витік коду
- технічний борг
- Метрика Холстеда [6-7]
- Метрика Джилба

Цикломатична складність вимірює складність програми шляхом підрахунку кількості лінійно незалежних шляхів у вихідному коді програми. Висока цикломатична складність може вказувати на складний код, який важко підтримувати.

Рядки коду (LOC) [8] — це проста, але базова метрика, яка вимірює розмір кодової бази шляхом підрахунку кількості рядків у файлі вихідного коду. Хоча це не є прямим показником якості, надмірно довгі методи або класи можуть вказувати на погану організацію коду. LOC не містить порожніх рядків і коментарів, а лише корисний код, який компілюється в машинний код.

Покриття коду [9,10] вимірює відсоток коду, який виконується автоматизованими тестами. Більше покриття коду зазвичай вказує на більш повне тестування коду.

Дублювання коду [11] вимірює кількість рядків дублюючого коду. Дубльований код може призвести до проблем з обслуговуванням і невідповідностей. У разі великої кількості дублюючого коду розробникам потрібно витрачати більше часу на зміни.

Порушення правил статичного аналізу коду вимірюють кількість порушень стандартів кодування або найкращих практик, виявлених інструментами статичного аналізу коду. Поширені проблеми включають неправильне іменування та стиль, створення змінних без значень, неправильну обробку помилок і порушення правил кодування. Кожна мова програмування має певні правила, такі як форматування, які ігноруються та викликають статичні помилки аналізатора. За бажанням деякі правила можна вимкнути для статичних аналізаторів.

Індекс ремонтопридатності — це загальний показник, який поєднує різні атрибути коду, такі як складність, дублювання коду та зміни коду, щоб оцінити загальну ремонтопридатність бази коду.

Витік коду - вимірює частоту змін коду з часом. Значні зміни коду можуть свідчити про

нестабільність або часті рефакторинги, що може вплинути на якість програмного забезпечення.

Термін «технічний борг» [12-13] був введений Уордом Каннінгемом, одним із авторів Agile Manifesto. Він порівняв невдалі рішення щодо розробки програмного забезпечення з фінансовим боргом. Технічна заборгованість стосується зусиль, необхідних для вирішення неоптимальних варіантів проектування чи реалізації в базі коду. Високий технічний борг може уповільнити швидкість розробки та збільшити ймовірність помилок. Причина високого технічного боргу зазвичай є результатом поганих архітектурних рішень, поганого кодування, частих змін коду та зазвичай вимірюється статичними аналізаторами. Різке збільшення технічної заборгованості зазвичай свідчить про проблеми з якістю коду, і його слід переглянути.

Метрики Холстеда засновані на кількості унікальних операторів і операндів і обчислюють розмір програми, зусилля, необхідні для написання програми. Метрики Холстеда допомагають оцінити складність і якість коду. Однак метрики не враховують ієрархію.

Поряд із загальними метриками існують також метрики ООП Кемерера [14-15]:

- WMC (Weighted Methods per Class) [16] кількість методів у класі з урахуванням їх цикломатичної складності. Тобто це сума цикломатичної складності всіх методів у класі. Високе значення дає підстави вважати необхідним рефакторинг
- DIT глибина успадкування
- NOC кількість нащадків
- CBO (Coupling Between Objects) - зв'язок між об'єктами, тобто скільки методів класу використовується. Це просте число
- RFC (Response for a Class) - відповідь класу, або кількість методів, які можна викликати в результаті одного виклику
- LCOM (Lack of Cohesion in Methods) – відсутність узгодженості між методами

Метрики Гілба — це набір показників, які використовуються для оцінки якості програмного забезпечення та ефективності процесів розробки. Хоча ці метрики не так широко відомі, як метрики Чидамбера або Кемерера, вони все одно допомагають розробникам оцінити технічний стан проекту. Метрику Jilba можна використовувати для аналізу різних аспектів програмного забезпечення, таких як складність, узгодженість, модульність і зручність обслуговування. Вони дозволяють оцінити рівень якості коду, а також допомагають у виявленні потенційних проблем на ранніх етапах розробки. Основні показники Jilba включають складність, узгодженість, ремонтопридатність, повторне використання, тестування, масштабованість і гнучкість.

Керування залежностями вимірює складність і стабільність залежностей у базі коду. Надлишкові залежності або застарілі бібліотеки можуть створювати вразливості та додаткові витрати на обслуговування. Показники перевірки коду пов’язані з процесами кодування, такими як час виконання перевірки, участь рецензента та кількість проблем, виявлених під час перевірки. Розрахунок цикломатичної складності в C++ і мовах об'єктно-орієнтованого програмування (ООП) передбачає аналіз потоку керування кодом, який може включати як процедурні, так і об'єктно-орієнтовані конструкції. У мовах ООП, таких як C++, необхідно враховувати додаткові фактори, такі як методи класу та зв’язки класів. Методи класу в межах класу можуть містити логічні оператори, такі як оператори if або цикли, які сприяють загальній цикломатичній складності класу. Якщо використовуються успадкування та поліморфізм, потік керування може стати більш складним через перевизначення методу та динамічну диспетчеризацію. Кожен перевизначений метод додає пункти прийняття рішення до складності. Якщо код передбачає взаємодію між класами, наприклад виклики методів або співпрацю, ці взаємодії також необхідно враховувати при обчисленні цикломатичної складності.

Іноді якість програмного забезпечення також включає атрибути, пов'язані з безпекою. Сьогодні найпоширенішим видом додатків є інтернет-сайти через відносну простоту створення - дають про себе знати системи управління контентом і фреймворки. І при використанні навіть досить простої розподіленої бот-атаки більшість простих сайтів просто зупиняться і не будуть працювати. І якщо раніше, в 2000-х, DDOS-атаки можна було зупинити, встановивши додаткове мережеве обладнання, то з появою ботнетів для забезпечення функціонування сайту роль якості програмного забезпечення стає однією з перших і мова йде не про прості ін'єкції sql і відомі правила OWASP, а в першу чергу продумування правильної архітектури.

ВИКЛАД ОСНОВНОГО МАТЕРІАЛУ

Для дослідження розрахунку якості коду були обрані наступні метрики:

- цикломатичної складності
- охоплення коду тестами
- технічної заборгованості
- метрики Кемерера (WMC, DIT, NOC, RFC)
- метрики Чідамбера (коефіцієнт зчеплення, коефіцієнт згуртованості, фактор поліморфізму)
- метрики Холстеда
- метрики Гілба
- Виявлення антишаблонів дизайну [17-18]

Cyclomatic complexity is calculated by formula 1:

$$M = E - N + 2P \quad (1)$$

де
M - цикломатична складність
E - кількість ребер у графі
N - кількість вершин у графі
P - кількість компонент зв'язності

Хоча цикломатична складність вимірює кількість точок прийняття рішень (гілок) у коді, вона не враховує його якість або чіткість. Код, який є складним, але також зрозумілим, читабельним і добре задокументованим, може мати високу цикломатичну складність. Ця метрика зосереджується лише на коді й не враховує зовнішні фактори, такі як архітектура програмного забезпечення, зв'язки між модулями чи залежності в коді, що не дає повної картини загальної якості програми. У деяких випадках висока цикломатична складність може бути природним наслідком вирішення складних проблем ефективними способами. Наприклад, це може вказувати на правильне використання алгоритмів для певної проблеми, але не обов'язково вказувати на низьку якість коду. Зменшення цикломатичної складності іноді призводить до надмірного спрощення коду, що може призвести до дублювання або неефективних рішень. Тому нижче значення складності не завжди є кращим, якщо воно ставить під загрозу логіку чи продуктивність програми. Цикломатична складність вимірює лише код, ігноруючи нефункціональні аспекти коду, такі як продуктивність, масштабованість і безпека, що може призвести до відсутності критичних проблем, які впливають на якість системи. У невеликих або простих програмах з мінімальними розгалуженнями цикломатична складність може не надати корисної інформації. У таких випадках високі значення можуть бути штучно завищені, навіть якщо кодова база добре організована та підтримується. Крім того, цикломатична складність не враховує дублювання коду, що може бути значною проблемою для майбутніх змін і обслуговування. Наприклад, два ідентичних блоки коду можуть мати таку саму цикломатичну складність, як і один, навіть якщо існує надмірність. Цей показник також не відображає складності бізнес-логіки: деякі програми можуть мати прості структури керування, але дуже складну логіку, яку цикломатична складність не охоплює.

Метрики Кемерера зосереджені на технічних аспектах коду, включаючи кількість розгалужень, циклів та інструкцій. Давайте розглянемо кожен з них.

Вага методів у класі (Weight Method per Class) обчислюється за формулою 2:

$$\mathrm{WMC} = \sum_{i=1}^{n} \mathrm{Ci} \quad (2)$$

де
n - кількість методів у класі
Ci - цикломатична складність кожного методу

Однак вони не завжди враховують контекст конкретного проекту або функціональні вимоги, які можуть істотно вплинути на складність програмного забезпечення. Зосередження на зменшенні значень метрик може призвести до надмірного спрощення коду, що може погіршити логіку або створити неефективні рішення, якщо розробники намагатимуться покращити метрики за рахунок функціональності. Показники Кемерера вимірюють здебільшого структурну та поведінкову складність коду, але не враховують такі важливі аспекти, як продуктивність, масштабованість або безпека, які мають вирішальне значення для загальної якості програмного забезпечення. Високі значення цих показників можуть вказувати на складність коду, але це не завжди корелює з реальними помилками або дефектами в програмі. Код із низькою складністю згідно з цими показниками все ще може містити серйозні проблеми, які не виявляються за допомогою статистичних заходів. Ці показники зосереджуються на локальних характеристиках коду, таких як розмір функції або кількість умовних операторів, і не враховують зв'язки між модулями чи їхні залежності. Це може призвести до неповної оцінки складності системи, оскільки взаємодія між компонентами коду також важлива для загального розуміння складності. Крім того, показники Кемерера вимірюють лише статичну складність, ігноруючи динамічні аспекти, такі як поведінка програми під час виконання, складність алгоритмів або взаємодія модулів. Зосередження на зменшенні складності також може негативно вплинути на продуктивність системи, оскільки оптимізація коду для покращення показників може призвести до надмірного спрощення або недооптимізації критичних частин коду. Для деяких типів програм, таких як невеликі проекти або коли складність коду не завжди корелює з проблемами реального світу, метрики Кемерера можуть бути менш корисними, оскільки вони зосереджуються на неважливих аспектах і не відображають реальних проблем у розробці.

Метрика LCOM Чидамера обчислюється за формулою:

$$LCOM = |P| - |Q| \quad (3)$$

Якщо |P| > |Q|, тоді 0
|P| це кількість пар методів, які не мають спільних змінних.
|Q| є кількістю пар методів, які мають принаймні одну спільну змінну.

Одним із провідних підходів до кількісної оцінки складності програмного коду є використання метрик Холстеда, розроблених Морісом Холстедом у 1977 році. Ця методика базується на концепції, згідно з якою будь-який програмний код можна представити у вигляді послідовності операторів та операндів, а рівень його складності визначається кількісними характеристиками зазначених елементів. Метрики Холстеда дають змогу оцінити не лише структурну складність коду, а й когнітивне навантаження на розробника, що виникає у процесі розуміння, супроводу та тестування програмного забезпечення.

Словник (n) це загальна кількість унікальних операторів (n_1) та унікальних операндів (n_2) та обчислюється за формулою 4:

$$n = n1+n2 \quad (4)$$

де
n1 - кількість унікальних операторів
n2 – кількість унікальних операндів

Довжина програми (N) це загальна кількість операторів (N_1) та операндів (N_2) у програмному коді та обчислюється за формулою 5:

$$N = N1+N2 \quad (5)$$

Об’єм (V) це оцінка розміру програми у бітах та вираховується згідно формулу 6:

$$V = N \times \log2(n) \quad (6)$$

Складність реалізації (E) це міра складності коду, яка оцінює зусилля, необхідні для його написання та обчислюється за формулою 7:

$$E = V/L \quad (7)$$

Метрика Холстеда оцінює програму на основі кількості унікальних і загальних входжень операторів і операндів, забезпечуючи математичний підхід до вимірювання складності програмного забезпечення. Аналізуючи ці елементи, метрика обчислює різні атрибути, такі як розмір словника, довжина програми, обсяг і зусилля, необхідні для впровадження. Область представляє розмір програми в бітах, тоді як зусилля оцінюють роботу, необхідну для написання або розуміння коду. Ці розрахунки допомагають передбачити витрати на обслуговування, ймовірність помилок і загальну якість програмного забезпечення.

Однією з головних переваг метрики Холстеда є її об’єктивність, оскільки вона базується на математичних формулах, а не на суб’єктивних судженнях. Оскільки метрика базується виключно на синтаксисі, вона не враховує логічну складність або архітектуру програмного забезпечення. Крім того, різні стилі кодування можуть призвести до варіацій у результатах, що робить його менш надійним для порівняння структурно різних реалізацій однієї і тієї ж функціональності. Крім того, для великих або багатомодульних програм метрика не завжди може точно відображати фактичну складність. Незважаючи на ці недоліки, метрика Холстеда залишається цінним інструментом у поєднанні з іншими методами оцінки якості програмного забезпечення.

Метрика складності Гілба відрізняється від метрики Холстеда і Маккейба тим, що вона не базується на одній стандартизованій формулі. Натомість він базується на наборі принципів, які дозволяють оцінювати складність програмного забезпечення з якісної та практичної точки зору. Том Гілб запропонував підхід, який враховує атрибути системи, такі як точки прийняття рішень, залежності та рівні абстракції, на відміну від суто математичних обчислень. Оскільки метрика Гілба не має фіксованої формули, її конкретна реалізація може змінюватися залежно від контексту використання. Він часто враховує такі фактори, як кількість окремих компонентів системи, їхні взаємозв’язки та когнітивні зусилля, необхідні для розуміння чи модифікації програмного забезпечення. Деякі варіації цього показника використовують зважені кількісні показники

елементів програмного забезпечення, таких як класи, функції та модулі, для оцінки ремонтопридатності коду та ризиків його обслуговування. Основною перевагою підходу Гілба є його гнучкість, яка дозволяє адаптувати метод до різних програмних проектів. Однак його залежність від суб'єктивних суджень при визначенні параметрів складності ускладнює стандартизацію його застосування в середовищах розробки. Порівняно зі структурованими показниками, такими як метрика Халстеда або цикломатична складність Маккейба, метод Гілба може бути менш точним, особливо під час аналізу великих програмних систем, де об'єктивність і повторюваність оцінювання є критичними.

Існує кілька популярних комерційних програмних інструментів, таких як SonarQube, DeepSource або Embold, але їхні алгоритми недоступні, навіть якщо вони засновані на атрибуті Cyclomatic Complexity і Code Coverage.

SonarQube є одним із найпоширеніших інструментів для статичного аналізу коду [19]. Він підтримує широкий спектр мов програмування (більше 25) і надає показники на основі якості коду, такі як цикломатична складність, рядки коду (LOC), тестове покриття, дублювання коду, виявлення помилок. Окрім статичного аналізу, SonarQube також може виконувати динамічний аналіз під час тестування програмного забезпечення.

Розглянемо детальніше архітектуру статичного аналізатора. Статичний аналізатор коду зазвичай складається з кількох ключових компонентів, які разом аналізують вихідний код без його виконання. Ось розбивка типової архітектури: рівень аналізу вхідних даних, лексичний аналіз, синтаксичний аналіз, семантичний аналіз.

Рівень аналізу вхідних даних — це компонент, який читає вихідний код із файлової системи або систем контролю версій (наприклад, Git). Він може обробляти різні типи вихідних файлів, наприклад .java, .cpp, .py тощо. Деякі інструменти вимагають етапів попередньої обробки, таких як токенізація або аналіз коментарів коду, макросів або специфічних для мови конструкцій перед аналізом. Цей крок може обробляти такі речі, як директиви препроцесора в C/C++ або імпорт в Java. Лексичний аналіз зчитує необроблений вихідний код і перетворює його на лексеми, які є меншими значущими одиницями (такими як ключові слова, оператори, змінні та типи даних). Токени використовуються на наступних етапах для розуміння структури та логіки коду. Синтаксичний аналіз, або парсер, бере лексеми, згенеровані лексером, і створює синтаксичне дерево (часто абстрактне синтаксичне дерево або AST). AST представляє граматичну структуру коду відповідно до правил синтаксису мови. Тут інструмент визначає правильний синтаксис і структуру коду. Семантичний аналізатор виконує більш глибоку перевірку значення коду. Це гарантує відповідність коду семантичним правилам мови програмування (наприклад, перевірка типу, перевірка оголошення змінних). Він також може перевіряти, чи змінні або функції правильно використовуються в контексті, забезпечуючи логічну коректність і виявляючи проблеми, такі як невідповідність типів.

Механізм обробки правил і шаблонів є основою інструментів статичного аналізу. Набір попередньо визначених правил або шаблонів (часто заснованих на стандартах кодування, найкращих практиках, інструкціях із безпеки або шаблонах оптимізації продуктивності) застосовуються до коду. Правила можуть бути такими: правила стилю коду (наприклад, відступи, домовленості про іменування), потенційні виявлення помилок (наприклад, розіменування нульового вказівника, неініціалізовані змінні), уразливості безпеки (наприклад, ризики впровадження SQL-ін'єкцій, міжсайтовий сценарій), показники складності (наприклад, цикломатична складність, дублювання коду)

Для оцінки ефективності розробленого методу було проведено експеримент. Було відібрано набір файлів C# з реальних програмних проектів, що містять різні рівні складності коду. Інструмент аналізував кожен файл, визначаючи показники, включаючи кількість рядків коду, цикломатичну складність, кількість батьківських і дочірніх класів і кількість зовнішніх посилань. Результати експортовано у формат CSV для подальшого аналізу. Після аналізу були виявлені закономірності між складністю коду та його структурою. Це дозволило нам оцінити ефективність інструменту та визначити потенційні напрямки його вдосконалення.

У рамках запропонованої інформаційної технології поставлено задачу автоматизованого оцінювання необхідності рефакторингу окремих функцій або класів на основі кількісних метрик. Метою є побудова моделі, яка на підставі структурних, когнітивних і зв'язних характеристик коду видає числове значення R∈[0,1]R \in [0, 1]R∈[0,1], що інтерпретується як ступінь (ймовірність) потреби у рефакторингу. Значення, близькі до нуля, свідчать про відсутність необхідності змін, тоді як значення, близькі до одиниці, вказують на критичну потребу у покращенні структури.

Формально задача моделюється як регресія згідно формули 8:

$$f : \mathbb{R}^n \to [0, 1] \tag{8}$$

де Rn — простір вхідних ознак, що описують клас чи функцію, а f(x) — оцінка необхідності рефакторингу.

Для експерименту було створено програмне забезпечення на базі Microsoft .NET та мови C#. Програма містить наступні блоки:

- FileParser: пошук файлів у папці за потрібним шаблоном або розширенням. Дозволяє здійснювати пошук у вкладених папках. У майбутньому можуть підтримуватися кластери даних [25]
- CodeParser: пошук класів і методів. Підтримує різні мови програмування, такі як C++, C# і Java. Цей модуль пропускає коментарі в коді та повертає лише чистий код програми
- CodeAnalyzer повертає список атрибутів якості програмного коду. Правила можна встановити на рівні класу або методу. Також є можливість створювати власні правила. Аналізує як методи, так і класи
- Звіт експортує інформацію як файл CSV

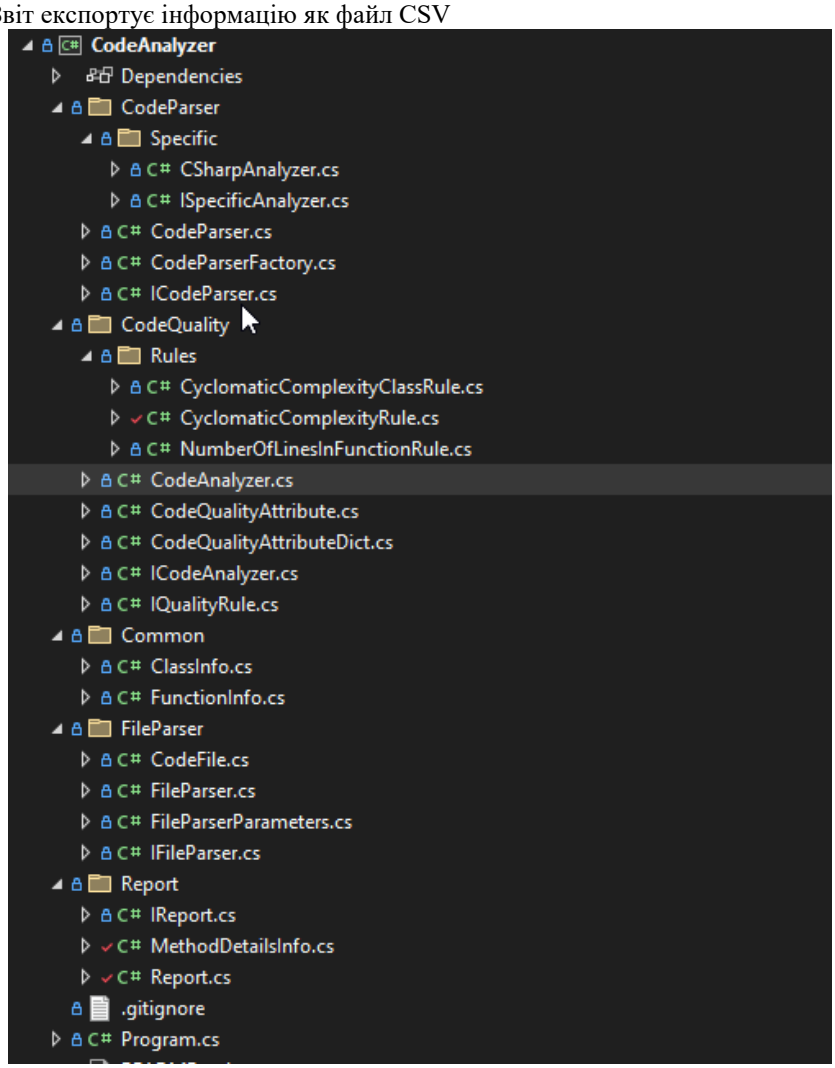

Рис 1. Структура програми обчислення метрик коду

Кожний метод аналізується за такими показниками:

- Рядки коду (LOC) – кількість рядків виконуваного коду
- Циклічна складність (CC) – міра кількості незалежних шляхів у методі
- Weighted Number of Methods in a Class (WMC) – обчислення загальної складності всіх методів у класі.
- Глибина дерева успадкування (DIT) – показник рівня ієрархії успадкування
- Відповідь на клас (RFC) – кількість методів, які можна викликати у відповідь на дії класу
- Кількість батьківських класів (NOP) – міра кількості прямих суперкласів для кожного класу.
- Кількість дочірніх класів (NOC) – міра кількості прямих підкласів для кожного класу
- Child-to-Parent Ratio (CPR) – показник відношення кількості підкласів до кількості суперкласів.

Модель обрано у вигляді Random Forest Regressor, що дозволяє не лише врахувати складні нелінійні залежності між ознаками, а й зберегти інтерпретованість результату через механізм оцінки важливості ознак. Для навчання моделі Random Forest було підготовлено вибірку, де кожен метод або клас представлено вектором вхідних ознак (LOC, CC, WMC, DIT, RFC, NOP, NOC, CPR). Як цільова змінна використовувався

індикатор якості коду, який може бути визначений експертним шляхом (наприклад, бінарна мітка "потребує рефакторингу" / "не потребує") або за допомогою існуючих метрик якості, таких як технічний борг або щільність дефектів.

Під час навчання Random Forest створює ансамбль дерев рішень, де кожне дерево приймає рішення на основі підмножини ознак і навчальної вибірки. Остаточне рішення приймається шляхом голосування (у випадку класифікації) або усереднення (у випадку регресії) результатів усіх дерев. Це дозволяє моделі враховувати складні взаємозв'язки між метриками коду і дає високу узагальнюючу здатність. В результаті експериментів було показано, що використання методу Random Forest для аналізу вказаних метрик дозволяє з високою точністю прогнозувати якість коду і підтримувати процеси управління якістю у великих програмних системах.

Алгоритм навчання базується на зведенні результатів бутстреп-множин у вигляді середнього прогнозу:

$$\mathrm{f(x)} = \frac{1}{T}\sum_{t=1}^{T} h_t(x) \tag{9}$$

де h_t — регресійне дерево, побудоване на випадковій підвибірці та підмножині ознак.

Для навчання моделі використовується еталонний набір прикладів, де значення $R \in [0,1]$ отримане на основі експертної оцінки.

Оцінка якості регресійної моделі здійснюється за допомогою таких метрик, як середньоквадратична помилка (Mean Squared Error, MSE) та середня абсолютна помилка (Mean Absolute Error, MAE). Середньоквадратична помилка визначає середнє значення квадратів різниць між фактичними та передбаченими значеннями і чутливо реагує на великі відхилення. Тому вона широко використовується у тих задачах, де важливо мінімізувати саме великі похибки, наприклад, у прогнозуванні навантаження на енергосистему, фінансовому моделюванні ризиків або прогнозуванні витрат на медичне страхування. У цих сферах навіть рідкісні великі помилки можуть мати серйозні наслідки, тому застосування MSE дозволяє моделі ефективніше їх уникати. Водночас середня абсолютна помилка оцінює середнє абсолютне відхилення між передбаченими та фактичними значеннями і менш чутлива до викидів у даних. MAE зазвичай використовується у тих випадках, коли важлива інтерпретованість результатів у зрозумілих одиницях виміру, наприклад, у прогнозуванні цін на нерухомість, строків доставки товарів або рівня продажів. У таких задачах MAE дозволяє безпосередньо оцінити середню похибку моделі у гривнях, днях чи кількості одиниць товару, що полегшує розуміння ефективності моделі для бізнес-користувачів. Таким чином, вибір між MSE та MAE залежить від специфіки задачі: чи важливіше мінімізувати великі помилки, чи забезпечити зрозумілу середню оцінку похибки.

Середньоквадратична помилка розраховується за формулою 10 (Mean Squared Error, MSE):

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - z_i)^2 \tag{10}$$

Середня абсолютна помилка обчислюється згідно формули 11 (MAE):

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - z_i| \tag{11}$$

Побудована модель дозволяє впровадити автоматизовану систему рейтингування якості класів і функцій. Значення понад 0.7 може трактуватися як сигнал до обов'язкового перегляду коду. Інтервал 0.4–0.7 — як помірний ризик, а менше 0.4 — як стабільний клас, що не потребує рефакторингу.

Таким чином, запропонована регресійна модель на основі Random Forest дає змогу кількісно оцінювати архітектурну та структурну якість коду без участі людини, що відкриває шлях до інтеграції таких підходів у CI/CD-процеси, автоматичний аудит та навіть вказівки для авто-рефакторингу.

ВИСНОВКИ З ДАНОГО ДОСЛІДЖЕННЯ І ПЕРСПЕКТИВИ ПОДАЛЬШИХ РОЗВІДОК У ДАНОМУ НАПРЯМІ

У даному підході було обґрунтовано доцільність використання методу Random Forest для автоматизованої оцінки якості програмного коду на основі структурних метрик. Проведений аналіз показав, що такі показники, як LOC, CC, WMC, DIT, RFC, NOP, NOC, CPR, забезпечують достатню інформативність для побудови ефективної моделі машинного навчання. Модель Random Forest продемонструвала високу точність прогнозування якості коду та стійкість до варіацій у вихідних даних. Важливою перевагою цього методу є можливість виявлення найбільш впливових показників, що на практиці дозволяє розробникам сфокусувати зусилля на ключових аспектах якості. Застосування даного підходу у процесі розробки програмного забезпечення дозволяє автоматизувати виявлення фрагментів коду, що потребують рефакторингу, а також підвищити ефективність контролю якості у великих та складних проектах.

Експеримент показав, що отримані дані дозволяють оцінити складність коду та потенційні проблемні області програмного забезпечення. Автоматичний аналіз спрощує процес перевірки якості коду та може бути корисним для розробників під час рефакторингу або вдосконалення архітектури. Отримані результати свідчать про перспективи подальшого розвитку інструменту, зокрема розширення його функціональних можливостей та підвищення точності оцінок. Також було вирішено додати розпізнавання антишаблонів та аномалій коду з використанням машинного навчання.

Література

1. E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati. 2024. An empirical study on bug severity estimation using source code metrics and static analysis; https://doi.org/10.1016/j.jss.2024.112179
2. V. Lenarduzzi, F. Pecorelli. 2022. A critical comparison on six static analysis tools: Detection, agreement, and precision. https://doi.org/10.1016/j.jss.2022.111575
3. R. Alfayez, R. Winn, W. Alwehaibi, E. Venson, B. Boeh. 2022. How SonarQube-identified technical debt is prioritized: An exploratory case study; https://doi.org/10.1016/j.infsof.2023.107147
4. Sergii Lysenko, Kira Bobrovnikova, Roman Shchuka, Oleg Savenko. A Cyberattacks Detection Technique Based on Evolutionary Algorithms. 11th International Conference on Dependable Systems, Services and Technologies (DESSERT), 2020. Vol.1. pp. 127-132
5. A. Abualkishik, L. Lavazza. 2021. An empirical evaluation of the “Cognitive Complexity” measure as a predictor of code understandability; https://doi.org/10.1016/j.jss.2022.111561
6. Dongjin Yu, Quanxin Yang, Xin Chen. 2023. Actionable code smell identification with fusion learning of metrics and semantics. https://doi.org/10.1016/j.scico.2024.103110
7. Yuanfang Cai, Rick Kazman. 2022. Software design analysis and technical debt management based on design rule theory. https://doi.org/10.1016/j.infsof.2023.107322
8. K. Ge, Q. Han. 2024. Hidden code vulnerability detection: A study of the Graph-BiLSTM algorithm. https://doi.org/10.1016/j.infsof.2024.107544
9. A. Aghamohammadi, S. Mirian-Hosseinabadi. 2021. Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness; https://doi.org/10.1016/j.infsof.2020.106426
10. M. Zakeri-Nasrabadi, S. Parsa, S. Jafari. 2023. Measuring and improving software testability at the design level; https://doi.org/10.1016/j.infsof.2024.107511
11. M. Lei, H. Li, N. Aundhkar. 2021. Deep learning application on code clone detection: A review of current knowledge; https://doi.org/10.1016/j.jss.2021.111141
12. A. Aldaeej, C. Seaman. 2023. A rule-based decision model to support technical debt decisions: A multiple case study of web and mobile app startups; https://doi.org/10.1016/j.infsof.2024.107542
13. G. Recupito, F. Pecorelli, G. Catolino. 2023.Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture; https://doi.org/10.1016/j.jss.2024.112151
14. U. Iftikhar, N. Ali, J. Börstler, M. Usman. 2022. A tertiary study on links between source code metrics and external quality attributes. https://doi.org/10.1016/j.infsof.2023.107348
15. N. Kaur, H. Singh. 2020. An empirical assessment of threshold techniques to discriminate the fault status of software. https://doi.org/10.1016/j.jksuci.2021.03.003
16. F. Huang, H. Madeira. 2023. Advancing modern code review effectiveness through human error mechanisms; https://doi.org/10.1016/j.jss.2024.112060
17. Cezar Sas, Andrea Capiluppi. 2021. Antipatterns in software classification taxonomies; https://doi.org/10.1016/j.jss.2022.111343
18. K. Alkharabsheh, S. Alawadi, K. Ignaim, N. Zanoon, Y. Crespo, E. Manso, J. Taboada. 2022. Prioritization of god class design smell: A multi-criteria based approach; https://doi.org/10.1016/j.jksuci.2022.09.011
19. Piërre van de Laar, Rosilde Corvino, Arjan J. Mooij. 2023. Custom static analysis to enhance insight into the usage of in-house libraries. https://doi.org/10.1016/j.jss.2024.112028
20. M. Alkhomsan, M. Alshayeb, M. Baslyman. 2024. Toward a novel taxonomy to capture code smells caused by refactoring. https://doi.org/10.1016/j.scico.2024.103120
21. C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y. Guéhéneuc, A. Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. https://doi.org/10.1016/j.infsof.2020.106278
22. Kashtalian, A., Lysenko, S., Savenko, O., Nicheporuk, A., Sochor, T., & Avsiyevych, V. (2024). Multi-computer malware detection systems with metamorphic functionality. Radioelectronic and Computer Systems, 2024(1), 152-175. doi:https://doi.org/10.32620/reks.2024.1.13
23. Savenko O. / Dynamic signature-based malware detection technique based on API call tracing / Savenko, O., Nicheporuk, A., Hurman, I., Lysenko, S. - CEUR-WS. – 2019. – Vol. 2393. – P.633-643, ISSN: 1613-0073
24. Markowsky G. The technique for metamorphic viruses' detection based on its obfuscation features analysis / G. Markowsky, O. Savenko, S. Lysenko, A. Nicheporuk // CEUR-WS, ISSN: 1613–0073. – 2018. – Vol. 2104. – Pp. 680–687.

25. KYRYCHENKO, O. 2022. INFORMATION TECHNOLOGY FOR STATISTICAL CLUSTER ANALYSIS OF INFORMATION IN COMPLEX NETWORKS. Computer systems and information technologies. 4 (Dec. 2022), 47–51. DOI:https://doi.org/10.31891/csit-2022-4-7.

26. B. Savenko, A. Kashtalian, S. Lysenko and O. Savenko, "Malware Detection By Distributed Systems with Partial Centralization," 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Dortmund, Germany, 2023, pp. 265-270, doi: 10.1109/IDAACS58523.2023.10348773.

27. Nicheporuk, A., Savenko, O., A. Nicheporuk, and Y. Nicheporuk. 2020. An android malware detection method based on CNN mixed-data model CEUR Workshop Proceedings Kharkiv, Ukraine. 2732:198–213.

28. D. Denysiuk, O. Savenko, S. Lysenko, B. Savenko and A. Kashtalian, "Method for Detecting Steganographic Changes in Images Using Machine Learning," 2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 2023, pp. 1-6, doi: 10.1109/DESSERT61349.2023.10416453.

29. Khalid Alkharabsheh. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class. 2021. https://doi.org/10.1016/j.infsof.2021.106736

30. Mansi Agnihotri. Application of machine learning algorithms for code smell prediction using object-oriented software metrics. 2020. https://doi.org/10.1080/09720510.2020.1799576

References

1. E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati. 2024. An empirical study on bug severity estimation using source code metrics and static analysis; https://doi.org/10.1016/j.jss.2024.112179

2. V. Lenarduzzi, F. Pecorelli. 2022. A critical comparison on six static analysis tools: Detection, agreement, and precision. https://doi.org/10.1016/j.jss.2022.111575

3. R. Alfayez, R. Winn, W. Alwehaibi, E. Venson, B. Boeh. 2022. How SonarQube-identified technical debt is prioritized: An exploratory case study; https://doi.org/10.1016/j.infsof.2023.107147

4. Sergii Lysenko, Kira Bobrovnikova, Roman Shchuka, Oleg Savenko. A Cyberattacks Detection Technique Based on Evolutionary Algorithms. 11th International Conference on Dependable Systems, Services and Technologies (DESSERT), 2020. Vol.1. pp. 127-132

5. A. Abualkishik, L. Lavazza. 2021. An empirical evaluation of the “Cognitive Complexity” measure as a predictor of code understandability; https://doi.org/10.1016/j.jss.2022.111561

6. Dongjin Yu, Quanxin Yang, Xin Chen. 2023. Actionable code smell identification with fusion learning of metrics and semantics. https://doi.org/10.1016/j.scico.2024.103110

7. Yuanfang Cai, Rick Kazman. 2022. Software design analysis and technical debt management based on design rule theory. https://doi.org/10.1016/j.infsof.2023.107322

8. K. Ge, Q. Han. 2024. Hidden code vulnerability detection: A study of the Graph-BiLSTM algorithm. https://doi.org/10.1016/j.infsof.2024.107544

9. A. Aghamohammadi, S. Mirian-Hosseinabadi. 2021. Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness; https://doi.org/10.1016/j.infsof.2020.106426

10. M. Zakeri-Nasrabadi, S. Parsa, S. Jafari. 2023. Measuring and improving software testability at the design level; https://doi.org/10.1016/j.infsof.2024.107511

11. M. Lei, H. Li, N. Aundhkar. 2021. Deep learning application on code clone detection: A review of current knowledge; https://doi.org/10.1016/j.jss.2021.111141

12. A. Aldaeej, C. Seaman. 2023. A rule-based decision model to support technical debt decisions: A multiple case study of web and mobile app startups; https://doi.org/10.1016/j.infsof.2024.107542

13. G. Recupito, F. Pecorelli, G. Catolino. 2023.Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture; https://doi.org/10.1016/j.jss.2024.112151

14. U. Iftikhar, N. Ali, J. Börstler, M. Usman. 2022. A tertiary study on links between source code metrics and external quality attributes. https://doi.org/10.1016/j.infsof.2023.107348

15. N. Kaur, H. Singh. 2020. An empirical assessment of threshold techniques to discriminate the fault status of software. https://doi.org/10.1016/j.jksuci.2021.03.003

16. F. Huang, H. Madeira. 2023. Advancing modern code review effectiveness through human error mechanisms; https://doi.org/10.1016/j.jss.2024.112060

17. Cezar Sas, Andrea Capiluppi. 2021. Antipatterns in software classification taxonomies; https://doi.org/10.1016/j.jss.2022.111343

18. K. Alkharabsheh, S. Alawadi, K. Ignaim, N. Zanoon, Y. Crespo, E. Manso, J. Taboada. 2022. Prioritization of god class design smell: A multi-criteria based approach; https://doi.org/10.1016/j.jksuci.2022.09.011

19. Piërre van de Laar, Rosilde Corvino, Arjan J. Mooij. 2023. Custom static analysis to enhance insight into the usage of in-house libraries. https://doi.org/10.1016/j.jss.2024.112028

20. M. Alkhomsan, M. Alshayeb, M. Baslyman. 2024. Toward a novel taxonomy to capture code smells caused by refactoring. https://doi.org/10.1016/j.scico.2024.103120

21. C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y. Guéhéneuc, A. Maiga. 2020. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. https://doi.org/10.1016/j.infsof.2020.106278

22. Kashtalian, A., Lysenko, S., Savenko, O., Nicheporuk, A., Sochor, T., & Avsiyevych, V. (2024). Multi-computer malware detection systems with metamorphic functionality. Radioelectronic and Computer Systems, 2024(1), 152-175. doi:https://doi.org/10.32620/reks.2024.1.13

23. Savenko O. / Dynamic signature-based malware detection technique based on API call tracing / Savenko, O., Nicheporuk, A., Hurman, I., Lysenko, S. - CEUR-WS. – 2019. – Vol. 2393. – P.633-643, ISSN: 1613-0073

24. Markowsky G. The technique for metamorphic viruses' detection based on its obfuscation features analysis / G. Markowsky, O. Savenko, S. Lysenko, A. Nicheporuk // CEUR-WS, ISSN: 1613–0073. – 2018. – Vol. 2104. – Pp. 680–687.

25. KYRYCHENKO, O. 2022. INFORMATION TECHNOLOGY FOR STATISTICAL CLUSTER ANALYSIS OF INFORMATION IN COMPLEX NETWORKS. Computer systems and information technologies. 4 (Dec. 2022), 47–51. DOI:https://doi.org/10.31891/csit-2022-4-7.

26. B. Savenko, A. Kashtalian, S. Lysenko and O. Savenko, "Malware Detection By Distributed Systems with Partial Centralization," 2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Dortmund, Germany, 2023, pp. 265-270, doi: 10.1109/IDAACS58523.2023.10348773.

27. Nicheporuk, A., Savenko, O., A. Nicheporuk, and Y. Nicheporuk. 2020. An android malware detection method based on CNN mixed-data model CEUR Workshop Proceedings Kharkiv, Ukraine. 2732:198–213.

28. D. Denysiuk, O. Savenko, S. Lysenko, B. Savenko and A. Kashtalian, "Method for Detecting Steganographic Changes in Images Using Machine Learning," 2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 2023, pp. 1-6, doi: 10.1109/DESSERT61349.2023.10416453.

29. Khalid Alkharabsheh. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class. 2021. https://doi.org/10.1016/j.infsof.2021.106736

30. Mansi Agnihotri. Application of machine learning algorithms for code smell prediction using object-oriented software metrics. 2020. https://doi.org/10.1080/09720510.2020.1799576