МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки та програмної інженерії
Кафедра інженерії програмного забезпечення

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

Катерина НЕСТЕРЕНКО
“____”_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСНИКА ОСВІТНЬОГО СТУПЕНЯ
“МАГІСТР”

Тема: “Методика визначення якості коду із використанням засобів штучного інтелекту”

Виконавець: Родічева Дарина Олексіївна

Керівник: к.т.н. Горський Олексій Миколайович

Консультант: д.т.н., доцент Чебанюк Олена Вікторівна

Нормоконтролер: асистент Ольга Сергіївна Кравченко

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра інженерії програмного забезпечення
Освітній ступінь магістр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Катерина НЕСТЕРЕНКО
"___" _____ 2023 р

ЗАВДАННЯ
на виконання кваліфікаційної роботи студента
Родічевої Дарини Олексіївни

1. Тема кваліфікаційної роботи: «Методика визначення якості коду із використанням засобів штучного інтелекту»
затверджена наказом ректора від 29.09.2023 р. № 1994/ст.

2. Термін виконання проекту: з 17.09.2023 р. по 31.12.2023 р.

3. Вихідні дані до роботи: розробити методику автоматичного визначення якості коду з засобами штучного інтелекту.

4. Зміст пояснювальної записки:
   1. Аналіз проблематики визначення якості коду.
   2. Методика визначення якості коду із використанням засобів штучного інтелекту.
   3. Експериментальне дослідження методики.
   4. Опис впровадження методики.

5. Перелік обов'язкових слайдів презентації:
   1. Переваги та недоліки мануальної перевірки якості коду.
   2. Загальний процес оцінки якості коду.
   3. Алгоритм створення нейронної мережі.
   4. Схема роботи програмного забезпечення.
   5. Демонстрація роботи створеного програмного забезпечення.
   6. Порівняльний аналіз оцінок респондентів та методологічних результатів.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1.	Розробка та затвердження графіка роботи	25.10.2023 – 27.10.2023	
2.	Підготовка та написання 1 розділу. Відсилка керівнику	28.10.2023 – 01.11.2023	
3.	Підготовка та написання 2 розділу. Відсилка керівнику	02.11.2023 – 15.11.2023	
4.	Підготовка та написання 3 розділу. Відсилка керівнику	16.11.2023 – 01.12.2023	
5.	Підготовка та написання 4 розділу. Відсилка керівнику	02.12.2023 – 08.12.2023	
6.	Редагування та друк пояснювальної записки, графічного матеріалу Відсилка ПЗ для перевірки на плагіат одним файлом.	13.12.2023 – 19.12.2023	
7.	Проходження нормо-контролю, перепліт пояснювальної записки. Отримання відгуку керівника. Підготовка презентації та тексту доповіді.	20.12.2023 – 26.12.2023	
8.	Передзахист (наявність віддрукованої ПЗ, презентації, позитивного відгуку керівника).	13.12.2023 – 27.12.2023	
9.	Отримання рецензії.	13.12.2023 – 25.12.2023	
10.	Здати секретарю ДЕК: ПЗ, ГМ, CD-R з електронними версіями ПЗ, ГМ, презентацію, відгук керівника, рецензію, довідку про успішність, 2 папки, 2 конверта).	25.12.2023 – 26.12.2023	
11.	Захист дипломної роботи перед ЕК	26.12.2023 – 27.12.2023	

Дата видачі завдання 17.09.2023 р.

Керівник дипломної роботи: к.т.н. Олексій ГОРСЬКИЙ

Завдання прийняв до виконання: Дарина РОДІЧЕВА

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Методика визначення якості коду із використанням засобів штучного інтелекту»: 98 с., 21 рис., 6 табл., 33 інформаційні джерела.

ЯКІСТЬ КОДУ, ШТУЧНИЙ ІНТЕЛЕКТ, СЕМАНТИЧНИЙ ПОШУК, ЗАПАХИ КОДУ, ЦИКЛОМАТИЧНА СКЛАДНІСТЬ, СТАТИЧНИЙ АНАЛІЗ КОДУ, СТРУКТУРНІ МЕТРИКИ КОДУ.

Об’єкт дослідження – процес автоматичного визначення якості програмного коду з використанням засобів штучного інтелекту.

Предмет дослідження – методи та засоби штучного інтелекту для автоматичного аналізу та визначення якості програмного коду.

Мета роботи – розробити методику автоматичного визначення якості коду, засновану на засобах штучного інтелекту, спрямовану на підвищення ефективності та точності процесу аналізу вихідного коду.

Методи дослідження – метод аналізу та евристичні методи для розгляду та порівняння наявних методик; метод синтезу для узагальнення здобутих знань; метод експериментального аналізу для вибору оптимальної архітектури моделей штучного інтелекту; методи статистичного аналізу для об’єктивного порівняння отриманих результатів з мануальними оцінками.

В процесі роботи було розглянуто існуючі методики та інструменти для автоматичного оцінювання та створено комплексну методику оцінки якості коду, результати якої є наближеними до дійсних.

Результом цієї роботи є комплексна методика оцінювання якості коду, основна особливість якої полягає у її інтегративному підході, використанні штучного інтелекту та розширенні концепції оцінки якості коду, що у свою чергу дає більш точні, наближені до експертних результати.

Розробку програмного забезпечення було проведено на мові програмування Python 3 у середовищі програмування JetBrains PyCharm.

ABSTRACT

Explanatory note to the thesis "Approach to determining code quality using artificial intelligence tools": 98 p., 21 figures, 6 tables, 33 information sources.

CODE QUALITY, ARTIFICIAL INTELLIGENCE, SEMANTIC SEARCH, CODE ODORS, CYCLOMATIC COMPLEXITY, STATIC CODE ANALYSIS, STRUCTURAL CODE METRICS.

Object of research - the process of automatic determination of program code quality based on artificial intelligence.

Subject of research - methods and tools of artificial intelligence for automatic analysis and evaluation of program code quality.

The purpose of the work - to develop an approach to automatic code quality determination based on artificial intelligence tools aimed at improving the efficiency and accuracy of the source code analysis process.

Methods of the research: analysis and heuristic methods for reviewing and comparing existing methods; synthesis method for generalizing the acquired knowledge; experimental analysis method for choosing the optimal architecture of artificial intelligence models; statistical analysis methods for comparing the results with manual estimates.

Throughout the study, an examination was conducted to elucidate the challenges perceived by developers in evaluating code quality. Existing methodologies and tools for automated assessment were scrutinized, culminating in the formulation of a comprehensive code quality assessment approach. The outcomes of this methodology closely align with empirical observations.

The result of this work is a comprehensive code quality assessment approach, incorporating artificial intelligence to enhance precision, resulting in assessments closely aligned with human expertise.

The software was developed using Python 3 programming language in the JetBrains PyCharm development environment.

ЗМІСТ

ПЕРЕЛІК ПРИЙНЯТИХ СКОРОЧЕНЬ

CR – code review (код рев’ю, перевірка коду)

IDE – integrated development environment (інтегроване середовище розробки)

IEC – International Electrotechnical Commission (Міжнародна електротехнічна комісія)

IEEE – Institute of Electrical and Electronics Engineers (Інститут інженерів з електротехніки та електроніки)

ISO – International Organization for Standardization (Міжнародна організація зі стандартизації)

НМ – нейронна мережа

ПЗ – програмне забезпечення

ШІ – штучний інтелект

ВСТУП

Актуальність запропонованої методики. У сучасному світі, коли програмне забезпечення є рушієм технологічного прогресу, його якість стає дедалі важливішим чинником. Від початку до завершення розробки, якість коду безпосередньо визначає успіх кінцевого продукту. Неякісний код може викликати програмні помилки та знизити продуктивність. Сучасні компанії використовують різні методи перевірки якості коду, такі як статичний і динамічний аналіз, автоматизоване тестування та інструменти для виявлення проблем. Інтеграція штучного інтелекту може поліпшити точність цих методів та розширити їхні можливості, що у свою чергу може сприяти підвищенню конкурентоспроможності компаній.

Основна мета цієї роботи – запропонувати інноваційний підхід до визначення якості програмного забезпечення, використовуючи ресурси штучного інтелекту для більш повного та об'єктивного аналізу кодової бази.

Задачі дипломної роботи:

1) Проведення комплексного огляду підходів до оцінки якості вихідного коду програмного забезпечення, аналіз встановлених стандартів та вивчення поширених практик.

2) Інтеграція засобів штучного інтелекту з традиційними методами оцінювання з метою створення новітнього комплексного підходу до визначення якості коду.

3) Реалізація запропонованої методики у програмне забезпечення, що включає в себе розробку та інтеграцію алгоритмів штучного інтелекту для автоматизованого аналізу якості коду.

4) Аналіз ефективності та результатів процесу оцінювання якості вихідного коду з використанням штучного інтелекту та порівняння його з традиційними методами.

Об’єкт дослідження – процес автоматичного визначення якості програмного коду з використанням засобів штучного інтелекту.

Предмет дослідження – методи та засоби штучного інтелекту для автоматичного аналізу та визначення якості програмного коду.

Методи дослідження:

- метод аналізу та евристичні методи для розгляду наявних методик автоматичного визначення якості коду, порівняння їх переваг та обмежень.
- метод синтезу для узагальнення здобутих знань для використання їх у розробці методики визначення якості коду;
- метод експериментального аналізу для визначення найкращої для визначеної мети архітектури моделей штучного інтелекту;
- методи статистичного аналізу для об’єктивного порівняння отриманих результатів з мануальними оцінками.

Наукова новизна роботи заключається у створенні новітнього підходу до розгляду якості програмного забезпечення. Використання цього підходу надає більш точні результати, ніж ті підходи, що існують і дозволяє оцінити такі аспекти кодової бази, які раніше не розглядалися інструментами автоматизації.

Практичне значення отриманих результатів. Результати роботи відкривають нові перспективи в галузі програмної інженерії, зокрема у процесах розробки та верифікації програмного забезпечення. У сучасних компаніях розглядається питання затратності часу та ресурсів на взаємоперевірки коду між розробниками та на повторні перевірки після внесення змін. Нова методика дозволяє провести оцінку якості коду перед ініціацією процесу перевірки співробітниками. Це сприяє значному скороченню часового циклу перевірки коду, забезпечуючи ефективніші взаємодії завдяки вищій якості коду. Зазначена методика може також слугувати важливим ресурсом для студентів під час вивчення мов програмування, сприяючи покращенню якості їх кодової бази.

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ ВИЗНАЧЕННЯ ЯКОСТІ КОДУ

1.1. Доменний аналіз області визначення якості коду

Для створення методики визначення якості коду спершу визначити поняття вихідного коду, які є критерії якісного коду і як якість коду впливає на загальну якість створюваного програмного забезпечення.

Тож, **вихідний код програми –** це набір інструкцій, написаних мовою програмування, які може виконати комп'ютер. Ці інструкції призначені для виконання конкретних завдань або операцій.

Програмне забезпечення (ПЗ), також відоме як комп'ютерне програмне забезпечення, відноситься до набору програм, даних та інструкцій, які дозволяють комп'ютерній системі виконувати певні завдання або функції. Воно охоплює широкий спектр додатків, операційних систем, утиліт та інших програм, призначених для вирішення різноманітних обчислювальних потреб.

1.1.1. Якість коду як критичний аспект розробки програмного забезпечення

Розробка програмного забезпечення – це складний і багатогранний процес, який включає створення, підтримку та управління програмними продуктами. Він охоплює широкий спектр діяльності, від початкового планування та аналізу вимог до розгортання, супроводу і, зрештою, виведення програмного забезпечення з експлуатації.

Якість розробки програмного забезпечення тісно пов'язана з якістю кінцевих програмних продуктів. Це важливо для забезпечення успішної розробки та постачання програмних продуктів, які відповідають вимогам користувачів та галузевим стандартам.

При цьому ремонтопридатність програмного забезпечення вважається однією з найважливіших характеристик якості програмного забезпечення, що

підкреслює важливість розробки програмного забезпечення, яке легко супроводжувати і підтримувати в довгостроковій перспективі.

Крім того, гнучкість програмного забезпечення має вирішальне значення для розробників, дизайнерів та спеціалістів з контролю якості, що лише підкреслює важливість адаптивності та реагування на мінливі вимоги та середовище.

І, якщо оцінювання архітектури або зрозумілості кінцевого продукту для користувача мають проводити спеціалісти з оцінки якості продукту, все ж існує практична частина ПЗ, якість якої прямо впливає на можливу кількість помилок, ремонтопридатність, підтримуваність, тощо. Цією частиною програмного забезпечення є написаний розробниками вихідний код.

Якість коду означає відповідність коду певним стандартам, конвенціям та найкращим практикам, які забезпечують його надійність, ремонтопридатність та ефективність.

Ключові аспекти якості коду:

Читабельність відіграє ключову роль у якості коду. Чіткість коду є важливою, щоб інші розробники могли легко зрозуміти мету, логіку та функціональність. Досягнення зрозумілості коду передбачає використання узгоджених стилів і практик кодування в усій кодовій базі. Послідовний підхід не тільки покращує читабельність, але й допомагає створити більш узгоджену та зрозумілу структуру коду.

Придатність до покращення означає легкість, з якою код можна оновлювати та змінювати, не порушуючи роботу всієї системи. Модульність відіграє тут вирішальну роль; добре структурований код із модульними компонентами забезпечує легке обслуговування, дозволяючи розробникам вносити зміни, не викликаючи каскадних ефектів. Крім того, вичерпна документація, коментарі та змістовні назви змінних і функцій значно сприяють зручності обслуговування коду, гарантуючи, що він залишається зрозумілим з часом.

Ефективність передбачає написання коду, який оптимально виконує завдання без непотрібних витрат ресурсів. Оптимізація коду має вирішальне значення для загальної продуктивності системи. Досягнення цього передбачає використання ефективних алгоритмів, відповідних структур даних і методів кодування, які мінімізують витрати на обчислення.

Надійність є фундаментальною для високоякісного коду. Надійна кодова база може протистояти помилкам, крайнім випадкам і неочікуваним вводам, забезпечуючи послідовну та передбачувану поведінку. Широкі методології тестування, включаючи модульні тести, інтеграційні тести та автоматизоване тестування, відіграють важливу роль у забезпеченні надійності коду шляхом виявлення та усунення потенційних проблем на ранніх стадіях циклу розробки.

Безпека є критично важливим аспектом якості коду. Написання захищеного коду має першочергове значення для запобігання вразливостям, таким як ін’єкційні атаки або витоки даних. Впровадження надійних засобів контролю доступу, методів шифрування та найкращих практик обробки конфіденційних даних є обов’язковим для зміцнення безпеки кодової бази.

Значення якості коду полягає в його глибокому впливі на весь життєвий цикл розробки програмного забезпечення. Високоякісний код зменшує кількість помилок і, відповідно, підвищує стабільність і надійність програмного забезпечення. Крім того, це призводить до економічності в довгостроковій перспективі, роблячи кодову базу легшою та менш дорогою для обслуговування. Акцентування уваги на якості коду сприяє покращенню співпраці між розробниками, полегшенню командної роботи та обміну знаннями в групах розробників.

На якість коду впливають певні фактори, зокрема наявність "запахів коду". “**Запахи коду”** - це індикатори потенційних проблем у програмному коді, які можуть призвести до проблем з підтримкою та зниження якості програмного забезпечення. Наявність "запахів коду" вказує на необхідність рефакторингу, більш того їх своєчасне та ефективне виявлення може допомогти розробникам пришвидшити та полегшити сам процес рефакторингу.

Крім того важливим аспектом якості є стратегії найменування різних частин коду. Дослідження показують, що правильне іменування функцій та змінних має вирішальне значення для покращення читабельності та розуміння коду [15]. Це впливає на портативність кодової бази, надійність та зручність супроводу. Послідовність стандартам кодування та своєчасний зворотній зв'язок щодо іменування ідентифікаторів прямо впливає на покращення читабельності та зрозумілості програми.

Завдяки приоритезації якості коду процеси розробки програмного забезпечення стають ефективнішими, створюючи більш продуктивні, безпечні та придатні для обслуговування програмні системи.

Загальний процес оцінки якості коду зображено на рис. 1.1:

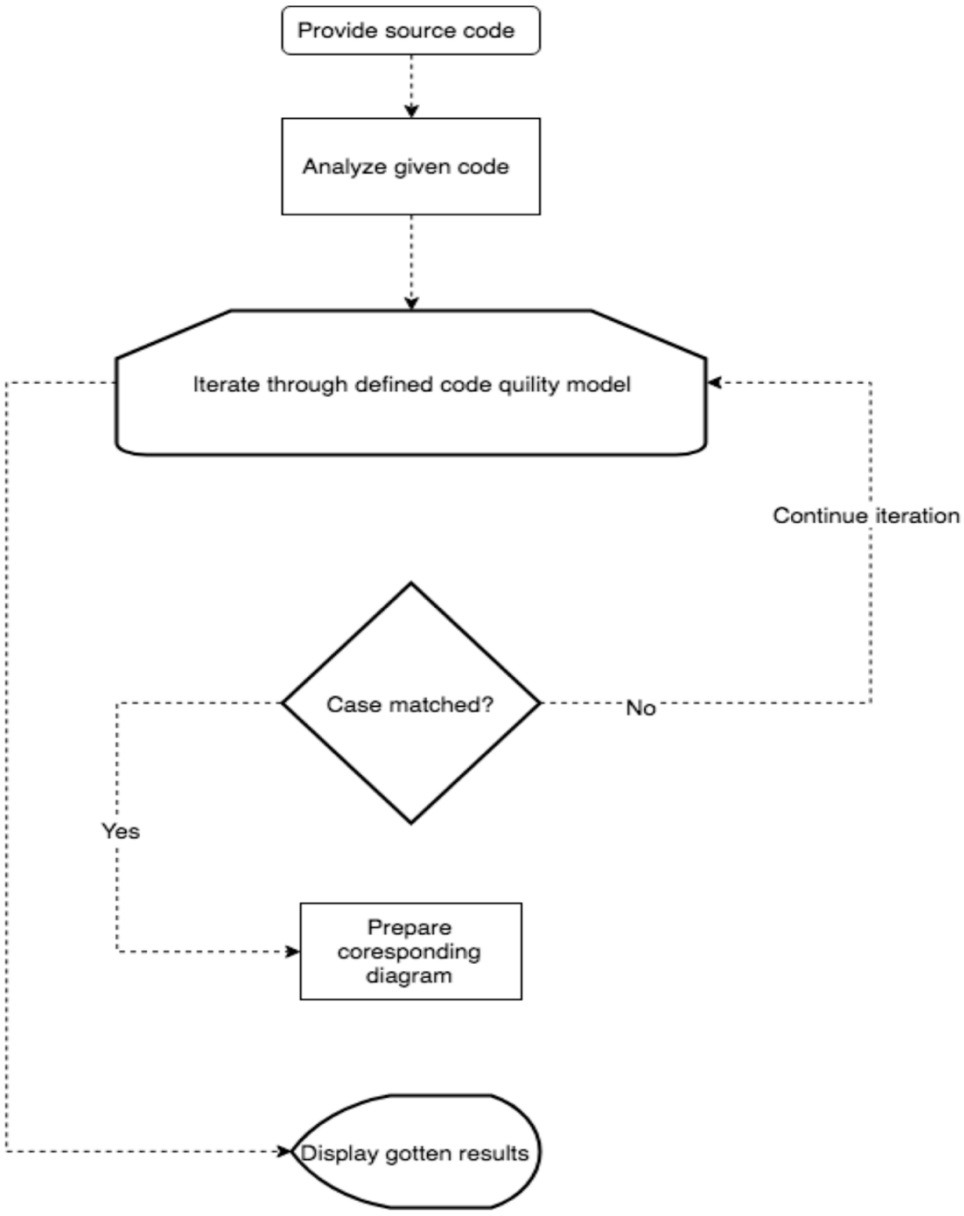

Рис. 1.1. Схема процесу оцінки якості вихідного коду.

1.1.2. Запахи коду

У програмуванні **"запах коду"** - це будь-яка характеристика у вихідному коді програми, яка, можливо, вказує на глибшу проблему.

Вони вважаються симптомами недоліків проектування або проблем у вихідному коді [25]. До поширених "запахів коду" належать дублікат коду, клас даних, клас бога, клас шизофреніка та довгий метод. Ці запахи коду можуть негативно вплинути на якість програмної системи та ускладнити її підтримку та розвиток.

Виявлення “запахів” має вирішальне значення для підтримки якості коду. "Запах коду” можна виявити різними методами, включаючи структурний аналіз коду, інформацію про історію змін та статичні аналізатори, які виявляють запах коду на основі метрик вихідного коду. Крім того, запах коду можна виявити за допомогою метрик якості, які представляють певні симптоми.

1.1.2.1. Довгий метод

У розробці програмного забезпечення **"довгий метод"** вважається "запахом коду", що може вказувати на те, що функція або метод є надмірно довгим і може бути складним для розуміння, супроводу та налагодження.

Довгі методи порушують принцип єдиної відповідальності (Single Responsibility Principle, SRP) з принципів SOLID, оскільки вони, як правило, охоплюють кілька обов'язків в рамках однієї функції. Це може призвести до зниження модульності та посилення зв'язків між різними частинами коду.

Складність визначення цього “запаху” полягає у тому, що немає певного граничного значення, який метод є довгим, а який ні. Тим не менше, є певні способи визначити його наявність:

1. Перевірити кількість рядків у методі. Якщо метод надмірно довгий, він може бути кандидатом на рефакторинг. Складність полягає у тому, що не існує суворого правила щодо того, яка кількість рядків вважається завеликою, але методи з кількома сотнями рядків часто вважаються проблематичними.

2. Визначити цикломатичну складність. **Цикломатична складність** - це програмна метрика, яка використовується для вимірювання складності програми. Такі інструменти, як складність МакКейба, можна використовувати для виявлення методів з високою цикломатичною складністю, що може вказувати на необхідність рефакторингу.

3. Перевірити, яку кількість завдань виконує метод. Якщо метод служить декільком цілям, це може бути ознакою того, що він занадто довгий. У якісному коді метод повинен мати єдину відповідальність, це робить код більш модульним і зручним для супроводу.

1.1.2.2. Великий клас

Запах "великого класу" є ознакою потенційних проблем у вихідному коді. Клас стає занадто великим з точки зору рядків коду, методів або обов'язків, що робить його складним для розуміння, підтримки та розширення.

З наукової точки зору, великий клас порушує SRP, охоплюючи занадто багато обов'язків в межах одного класу. Це може призвести до збільшення складності, зменшення модульності та труднощів у розумінні коду. Великі класи часто асоціюються з відсутністю належної абстракції і можуть вказувати на те, що клас намагається зробити занадто багато.

Великий клас - це клас, який має надто багато методів і атрибутів та несе на собі безліч неспоріднених обов'язків. Визначення великого класу здійснюється за допомогою метрик WMC (Weighted Methods per Class) і NOC (Number of Children) [20]. Клас вважається великим, якщо його WMC перевищує певний поріг. Додатково, клас, який має велику кількість підтипів, також може бути ознакою великого класу.

Так, за М. Ланца, великі класи визначаються за допомогою метрик WMC і NOC. Метрика WMC вимірює загальну складність класу, рахуючи кількість його методів, зважених їх складністю. З іншого боку, метрика NOC вимірює кількість безпосередніх підкласів класу. Клас вважається великим, якщо він перевищує поріг для будь-якої з цих метрик. Значення порогу може варіюватися, проте високі значення WMC або висока кількість NOC можуть вказувати на те, що клас є занадто складним та важким для обслуговування [20].

Тим не менш сама варіативність цих метрик не дає точного розуміння, чи має певний клас запах коду, , тому визначення за цими параметрами, хоча і є математично розрахованим, інтерпретація результатів є інтуїтивною і залежить від того, хто проводить оцінювання.

Існують і інші критерії, що вказують на наявність цього “запаху”:

1. Кількість рядків. Для визначення цього “запаху” коду необхідно перевірити кількість рядків у класі. Клас з надмірно великою кількістю рядків може бути ознакою “великого класу”.

2. Кількість методів. Велика кількість методів в одному класі може вказувати на те, що намагається впоратися з декількома обов'язками.

3. Завелика кількість обов'язків. Якщо клас відповідає за декілька різних завдань, він може бути кандидатом на рефакторинг.

4. Зв'язок. Необхідно перевірити, чи є зв'язок між класом та іншими класами в системі. Великі класи часто мають сильний зв'язок, що може призвести до недостатньої гнучкості та ремонтопридатності.

5. Згуртованість. Для визначення цього запаху треба оцінити зв'язність всередині класу. Якщо в класі бракує зв'язності, а методи всередині класу не є тісно пов'язаними, це може бути ознакою великого класу.

1.1.3. Структурні метрики коду

Структурні метрики коду — це числові величини, які використовуються для вимірювання різноманітних аспектів структури програмного коду. Їхня основна мета полягає в оцінці складності, розміру, та інших характеристик програмного забезпечення на основі його логічної та фізичної структури. Ці метрики надають об'єктивні критерії для оцінки, порівняння та вдосконалення якості коду, а також допомагають приймати рішення щодо його підтримки та розвитку.

Серед найпоширеніших структурних метрик можна виокремити кількість рядків коду, цикломатичну складність, кількість вузлів у структурі програми, та інші показники, які допомагають розуміти та вдосконалювати архітектуру та якість програмного забезпечення.

1.1.3.1. Цикломатична складність коду

Цикломатична складність - це метрика програмного забезпечення, яка вимірює складність програми шляхом аналізу структури потоку управління. Вона

визначається як кількість лінійно незалежних шляхів через вихідний код програми. Поняття цикломатичної складності було введено Томасом МакКейбом (Thomas J. McCabe) у 1976 році як засіб кількісної оцінки складності програми шляхом аналізу її графа потоку управління [28].

Як метрику, цикломатична складність може бути розрахована за допомогою наступних формул:

1. Базова формула цикломатичної складності:

Цикломатична складність = E - N + 2P, *де E відповідає кількості ребер, N – кількості вершин, а P - кількості з'єднаних компонентів у графі потоку управління.*

Ця формула базується на теорії графів та рахує кількість лінійно незалежних шляхів у програмі. Чим вище ця величина, тим складніше програма.

2. Формула підрахунку точок рішення:

Цикломатична складність = Кількість точок рішення + 1.

Точки рішення відображають різні точки прийняття рішень в програмі. Додавання одиниці компенсує базову складність.

3. Формула підсумовування предикатних вершин:

Цикломатична складність = Сума всіх предикатних вершин + 1.

Ця формула визначає складність на основі кількості умовних операторів у програмі. Кожна предикатна вершина є окремою точкою прийняття рішень, і додавання одиниці враховує базову складність.

Кожна з цих формул пропонує унікальний підхід до вимірювання цикломатичної складності, враховуючи різні аспекти структури програми. Використання конкретної формули може залежати від вимог та контексту аналізу програмного коду.

Цикломатична складність широко використовується як метрика програмного забезпечення вже понад 40 років і вважається однією з найпопулярніших метрик ПЗ. Вона застосовується в різних областях, включаючи компонентне програмне забезпечення, розробку програмного забезпечення та сервіс-орієнтовану архітектуру [28].

Дослідники вивчали зв'язок між цикломатичною складністю та іншими метриками програмного забезпечення, такими як кількість рядків коду, індекс супроводжуваності та розміри модулів [11] [9]. Були дискусії щодо кореляції між цикломатичною складністю МакКейба (CC) та кількістю рядків коду (LOC) [11]. Крім того, дослідження підкреслили взаємозв'язок між цикломатичною складністю, кількістю рядків коду, об'ємом Холстеда та індексом ремонтопридатності [9].

Незважаючи на широке використання, існує критика та дискусії щодо обмежень та інтерпретації цикломатичної складності як метрики програмного забезпечення. Деякі дослідники запропонували альтернативні метрики, такі як NPATH, спрямовані на подолання обмежень цикломатичної складності МакКейба [27]. Крім того, ведуться дискусії щодо чутливості теоретико-графових метрик, включаючи цикломатичну складність, до граничних напрямків для структурованих і неструктурованих програм.

Так, цикломатична складність є важливою метрикою програмного забезпечення, яка дає уявлення про складність структури потоку управління програми. Вона широко використовується в розробці програмного забезпечення, архітектурному аналізі та інших сферах. Хоча вона є цінним інструментом для оцінки складності програм, постійні дискусії та дослідження спрямовані на усунення її обмежень та вивчення альтернативних метрик для більш повного розуміння складності програмного забезпечення.

1.1.3.2. Метрики Холстеда

Метрики програмного забезпечення Холстеда, запроваджені Морісом Говардом Холстедом у 1977 році, є набором показників, призначених для кількісної оцінки різних аспектів програмного забезпечення. Ці метрики мають на меті надати уявлення про складність і розмір програмного забезпечення, а також про зусилля, необхідні для його розробки та підтримки.

Ці метрики включають довжину програми (N), яка кількісно визначає загальну кількість входжень операторів та операндів у програмі, та розмір словника (n), що представляє загальну кількість окремих операторів та операндів.

Об'єм програми (V) визначається як добуток довжини програми (N) на логарифм розміру словника (n), виражений як V = N * log2(n).

Інша метрика, рівень програми (L), вимірює співвідношення входжень операторів (n1) до входжень операндів (n2) у програмі, що формулюється як L = n1/n2. Крім того, складність програми (D) визначається відношенням кількості унікальних операторів до загальної кількості операторів у програмі, тобто D = (n1/2) * (N2/n2).

Для всебічної оцінки зусиль, залучених до процесу розробки, вводиться показник трудомісткості програми (E) як добуток обсягу програми (V) і складності програми (D), що позначається як E = V * D. Ця метрика дає уявлення про тонкощі і проблеми, пов'язані з побудовою програмного забезпечення.

Час реалізації (T) слугує оцінкою тривалості, необхідної для розробки програми. Він є похідним від програмних зусиль (E) і постійною величиною, на яку впливають мова програмування та середовище розробки. Включення міркувань, пов'язаних з часом, забезпечує більш повне розуміння практичних наслідків і потреб у ресурсах для реалізації програми.

Метрики Холстеда використовуються в різних контекстах, включаючи оцінку складності програмного забезпечення, супровід програмного забезпечення, оцінку якості програмного забезпечення та об'єктно-орієнтований системний аналіз.

Крім того, метрики Холстеда використовуються у кореляційних дослідженнях з іншими метриками програмного забезпечення та надійності програмного забезпечення. Наприклад, вони були співвіднесені з внутрішніми метриками програмного забезпечення, такими як кількість рядків коду та цикломатична складність МакКейба, для оцінки надійності програмного забезпечення [24] та для аналізу кореляції між зусиллями з тестування та метриками складності програмного забезпечення [1].

Окрім застосування в інженерії програмного забезпечення, метрики Холстеда були предметом статистичних методів аналізу даних програмних метрик, а також були адаптовані для вимірювання складності архітектури [21]. Їх

також використовували для виявлення плагіату та оцінки науковості програмного забезпечення [8].

1.1.4. Вплив найменування на якість коду

Найменування у програмуванні — це процес присвоєння зрозумілих та описових імен (назв) одиницям коду. Це важлива практика, що дозволяє програмістам та іншим учасникам проекту швидко розуміти призначення та функціонал без докладного вивчення вихідного коду.

Правильне іменування функцій та змінних суттєво впливає на читабельність та ремонтопридатність коду. Рішення щодо іменування, разом із форматуванням, відіграють вирішальну роль у визначенні читабельності вихідного коду програми, впливаючи на переносимість кодової бази, доступність для новачків, надійність та ремонтопридатність.

В ході емпіричних досліджень читабельності коду було виявлено, що покращення назв змінних та методів було однією з найчастіших пропозицій щодо покращення читабельності коду [15]. Крім того, було визнано роль ланцюжків методів та коментарів у читабельності та розумінні програмного забезпечення, що підкреслює важливість іменування для читабельності та розуміння програмного забезпечення [7].

Важливість правильного іменування ще більше підкреслюється проблемами, пов'язаними з обслуговуванням коду, коли доречність назви може змінюватися з часом, ще більше впливаючи на читабельність і зрозумілість коду. Послідовні стандарти написання коду, включаючи рекомендації щодо іменування класів, методів та змінних, покращують читабельність коду.

Дослідження демонструють, як різні фактори, такі як програмні конструкції та угоди про імена, можуть впливати на читабельність і розбірливість коду та який вплив мають угод про імена на легкість читання і розуміння коду. Крім того, надання своєчасного зворотного зв'язку щодо іменування ідентифікаторів програмістам-початківцям було визначено як спосіб значно покращити читабельність програми [22].

Отже, правильне іменування функцій та змінних має вирішальне значення для покращення читабельності та розуміння коду. Послідовні стандарти кодування та своєчасний зворотній зв'язок щодо іменування ідентифікаторів є важливими для покращення читабельності та зрозумілості програми.

1.1.5. Можливості штучного інтелекту у сфері оцінки якості коду

Штучний інтелект (ШІ) представляє собою вершину технологічного прогресу, що містить створення комп’ютерних систем, здатних виконувати завдання, які традиційно потребують людського інтелекту. Ця широка сфера охоплює такі дисципліни, як машинне навчання, обробка природної мови, комп’ютерне бачення тощо. Особливість штучного інтелекту полягає в його здатності вчитися на даних, розпізнавати закономірності, приймати обґрунтовані рішення та постійно вдосконалювати свою продуктивність без явного програмування для кожного завдання.

За своєю суттю **машинне навчання** є основоположним елементом штучного інтелекту, що дозволяє комп’ютерам навчатися на основі даних і екстраполювати прогнози чи рішення, позбавлені чітких інструкцій. Ця сфера охоплює різноманітні парадигми, такі як контрольоване навчання, неконтрольоване навчання та навчання з підкріпленням, що сприяє розвитку систем ШІ.

Нейронні мережі є фундаментальною концепцією в галузі штучного інтелекту та машинного навчання. Це обчислювальні моделі, натхненні структурою та функцією людського мозку, які складаються з взаємопов'язаних вузлів, або "нейронів", які спільно обробляють складну інформацію та розпізнають патерни. Нейронні мережі широко використовуються в різних галузях, включаючи розпізнавання зображень та мови, медичну діагностику та фінансові прогнози.

У контексті машинного навчання нейронні мережі відіграють важливу роль, оскільки їх використовують для навчання та прогнозування на основі вхідних даних. Концепція машинного навчання обертається навколо ідеї надання

можливості машинам вчитися з даних та покращувати свою продуктивність з часом, не будучи явно програмованими. Це досягається різними методами навчання, включаючи механічне навчання, індуктивне навчання, аналогічне навчання та навчання на основі нейронних мереж [9].

Процес створення нейронної мережі зображено на рис. 1.2.

Крім того, застосування нейронних мереж розширилося на глибинне навчання, підгалузь машинного навчання, яка акцентує на вивченні представлення даних через кілька шарів взаємопов'язаних вузлів. Глибоке навчання здобуло значну увагу завдяки його здатності автоматично виявляти витончені патерни та особливості від сирих даних, що призвело до прогресу в галузях, таких як комп'ютерне зорове сприйняття та обробка природної мови.

Глибинне навчання, відгалуження машинного навчання, використовує складні нейронні мережі для дешифрування представлень даних. Його успіхи у вирішенні складних завдань, від розпізнавання зображень до обробки природної мови, позиціонує глибоке навчання як лідера серед штучного інтелекту.

Обробка природної мови (NLP), ще один аспект штучного інтелекту, дозволяє машинам розуміти та генерувати людську мову. Ця сфера охоплює такі завдання, як класифікація тексту, аналіз настроїв, мовний переклад і розпізнавання мовлення, що підкреслює вправність штучного інтелекту в інтерпретації людського спілкування.

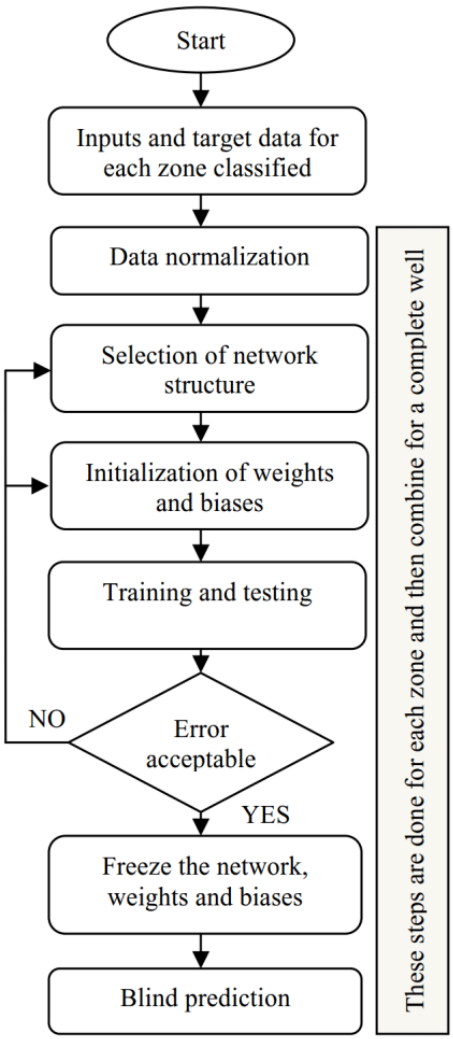

Рис. 1.2. Алгоритм створення нейронної мережі

У сфері розробки програмного забезпечення ШІ започаткував трансформаційну епоху в тестуванні якості коду. Інструменти автоматизованого аналізу коду використовують методи штучного інтелекту, щоб швидко переглядати кодові бази, розпізнавати шаблони, виявляти аномалії та оцінювати якість коду. Ці інструменти значно прискорюють такі завдання, як виявлення потенційних помилок, вразливостей або відхилень від встановлених норм кодування.

1.1.6. Роль код рев'ю в забезпеченні якісного ПЗ

Код рев’ю (code review, рецензування коду) є важливою складовою частиною процесу розробки ПЗ і вимагає від учасників значної кількості часу та

зусиль. У нещодавньому опитуванні серед програмістів відкритого ПЗ було виявлено, що розробники витрачають у середньому шість годин на тиждень на перегляд коду [23]. Це більше шести відсотків від загального часу розробки, що може є суттєвою часткою. Схоже дослідження показує, що час на розгляд змін в коді може становити і значно більше (рис. 1.3). Однак, незважаючи на витрати часу, перегляд коду залишається важливою практикою у розробці програмного забезпечення з кількох причин.

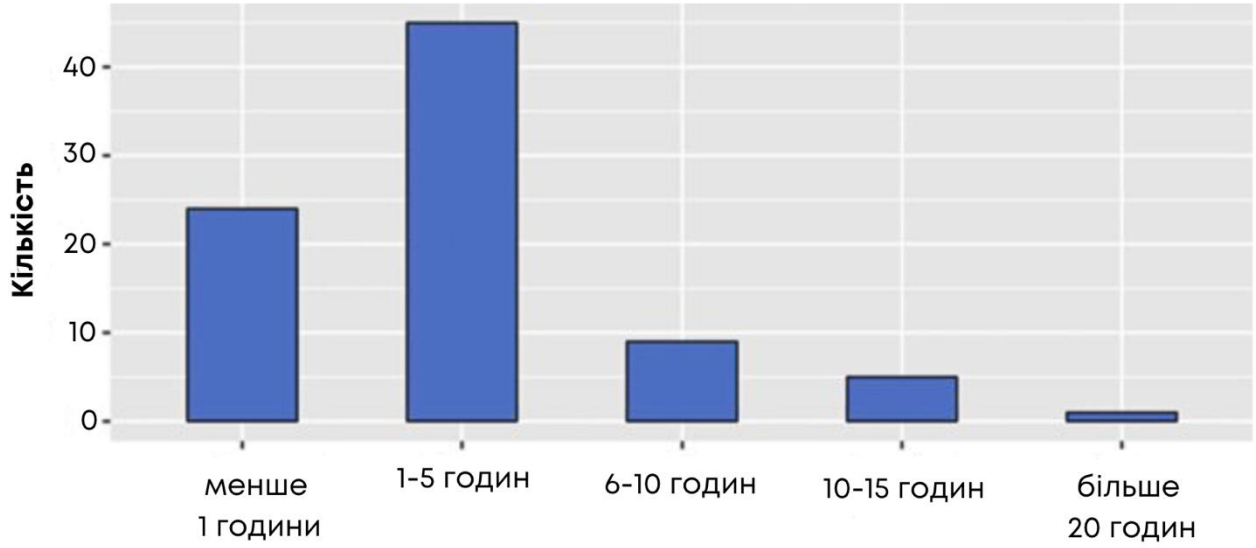

Рис. 1.3. Кількість часу, витрачена розробниками на перевірку коду

По-перше, перегляд коду слугує засобом забезпечення якості програмного забезпечення та виявлення дефектів на ранніх стадіях процесу розробки.

Крім того, рецензування коду є цінною практикою для обміну знаннями та навчання в командах розробників. Він дозволяє розробникам отримати уявлення про різні стилі кодування, найкращі практики та потенційні покращення, тим самим сприяючи їхньому професійному зростанню та вдосконаленню навичок.

Код рев'ю відіграє життєво важливу роль у підтримці ремонтопридатності програмного забезпечення та зменшенні технічного боргу. Виявляючи та вирішуючи проблеми в процесі перегляду, розробники можуть запобігти накопиченню технічного боргу, що призводить до створення більш зручної та

стійкої кодової бази. Крім того, перегляд коду сприяє підвищенню загальної надійності та безпеки програмного забезпечення шляхом виявлення вразливостей і забезпечення дотримання стандартів кодування та практик безпеки [13]. Це продемонстровано на рис. 1.4.

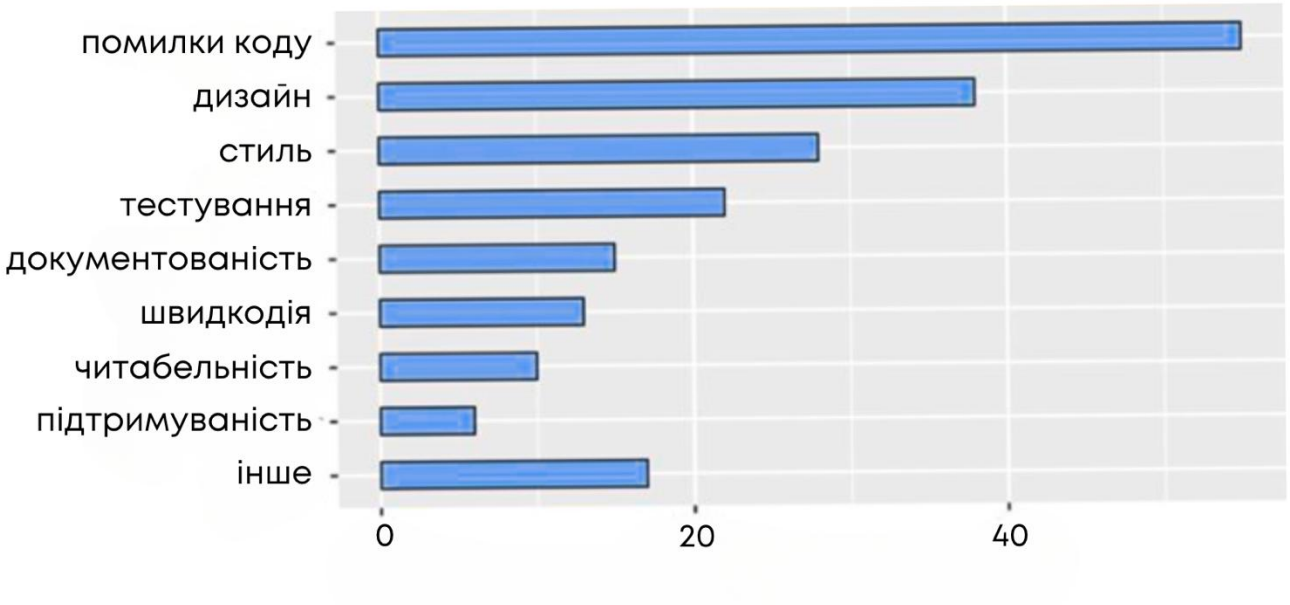

Рис. 1.4. Проблеми, які допомагає знайти та вирішити код рев'ю

Незважаючи на часові витрати, результати перегляду коду позитивно впливають на програму. Дослідження показали, що перегляд коду допомагає покращити якість програмного забезпечення, зменшити кількість дефектів та підвищити загальну надійність програмного забезпечення. Також було помічено, що перегляд коду сприяє ранньому виявленню помилок і потенційних недоліків дизайну, що призводить до покращення зручності обслуговування програмного забезпечення та його довгострокової стабільності [18].

Однак важливо визнати, що код рев'ю може також створювати проблеми, такі як часові обмеження та потенційні конфлікти під час процесу перегляду (рис. 1.5). Крім того, практична застосовність формальних оглядів коду в гнучкій розробці є предметом дискусій, що підкреслює потребу в ефективних та дієвих

процесах перегляду в сучасних методологіях розробки програмного забезпечення [32].

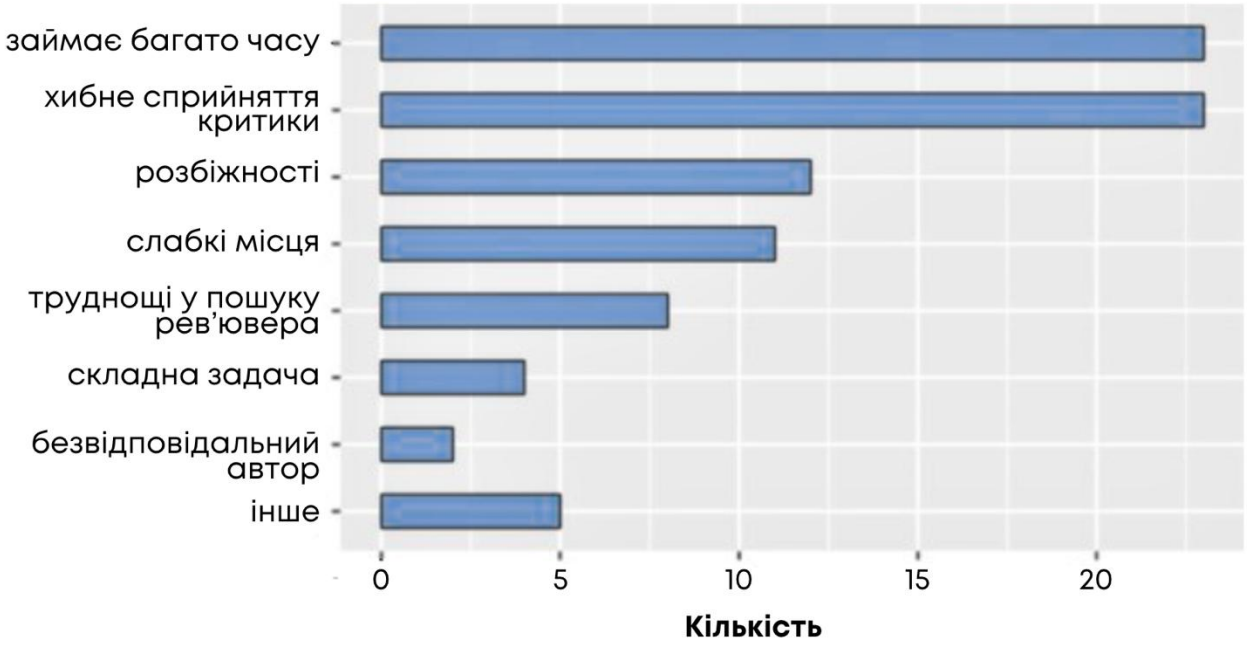

Рис. 1.5. Проблеми, пов’язані з проведенням код рев’ю

Отже, перегляд коду є трудомісткою, але незамінною практикою в розробці програмного забезпечення. Код рев’ю відіграє важливу роль у забезпеченні якості програмного забезпечення, обміні знаннями, виявленні дефектів та загальній надійності програмного забезпечення. Незважаючи на часові витрати, позитивний вплив перегляду коду на якість програми та її супроводжуваність підкреслює його важливість у життєвому циклі розробки програмного забезпечення.

1.2. Огляд відомих підходів визначення якості коду автоматично

Для забезпечення якості коду можна використовувати різні підходи в залежності від мети.

Статичний аналіз передбачає перевірку вихідного коду без його виконання з метою виявлення потенційних проблем і покращення якості коду [25]. Цей метод працює повністю автоматизовано без участі користувача та не потребує

складних налаштувань тестування, що робить його корисним для раннього виявлення проблем із якістю коду [6]. Він може забезпечити ранні індикатори дефектів програмного забезпечення. Однак інструменти статичного аналізу можуть не охоплювати всі проблеми, пов’язані з виконанням, і їх ефективність потрібно постійно вдосконалювати [16].

Динамічний аналіз оцінює поведінку коду під час виконання, допомагаючи виявити проблеми під час виконання та підвищити якість коду. Це може надати докази динамічної цілісності системи, розширюючи підхід статичного аналізу. Однак динамічний аналіз може зайняти багато часу та не охоплювати всі можливі шляхи коду, що потенційно обмежує його ефективність у певних сценаріях [29].

Структурний аналіз зосереджується на архітектурі та дизайні коду, сприяючи його супроводжуваності, верифікації та гнучкості. Він широко використовується в розробці програмного забезпечення, особливо в контексті критичних для безпеки систем. Цей метод має вирішальне значення для розуміння архітектури системної програми та може впливати на такі атрибути якості, як модульність і можливість перевірки. Цей метод, зокрема, може передбачати розрахунок цикломатичної складності [12] та індекс придатності до покращень [31]

Аналіз безпеки включає такі методи, як аналіз забруднень, має вирішальне значення для виявлення та усунення вразливостей безпеки в коді. Ця методика включає в себе аналіз використовуваних бібліотек та рішень на присутність вразливостей. Однак у статичному аналізі, спрямованому на виявлення вразливостей безпеки, можуть виникнути проблеми, а емпіричні докази, що підтверджують твердження в цій області, можуть бути обмеженими [5].

Емпіричний аналіз, наприклад перевірка коду, відіграє важливу роль у покращенні якості програмного забезпечення завдяки використанню колективного досвіду та знань розробників. Це має важливе значення для розуміння якості коду та може зменшити зусилля при перевірці коду. Однак може виникнути потреба в забезпеченні якості конкретного програмного забезпечення, і слід застосовувати специфічні підходи.

ШІ можна використовувати для автоматизації оцінки якості коду, дозволяючи ідентифікувати шаблони та аномалії, які впливають на якість коду. Він має практичні наслідки для трансформації DevOps і підтримки бізнесу в їхній діяльності. ШІ надає нові підходи до тестування якості коду, автоматизуючи аналіз для швидкого виявлення шаблонів, аномалій і потенційних вразливостей у кодових базах. Застосування алгоритмів машинного навчання забезпечує швидку послідовну оцінку показників коду, пропонуючи розуміння для вдосконалення та оптимізації процесів розробки, тим самим зміцнюючи програмні продукти підвищеної якості та надійності.

Підсумовуючи, кожен підхід до тестування якості коду має свої переваги та недоліки. Статичний аналіз надає ранні індикатори дефектів, але може не охопити всі проблеми під час виконання. Динамічний аналіз оцінює поведінку коду під час виконання, але може зайняти багато часу. Структурний аналіз зосереджується на архітектурі та дизайні, сприяючи ремонтопридатності та перевірці. Аналіз безпеки має вирішальне значення для виявлення вразливостей, але може зіткнутися з проблемами під час статичного аналізу. Емпіричний аналіз використовує колективний досвід, але може потребувати спеціальних підходів. ШІ може автоматизувати оцінку якості коду та має практичні наслідки в різних областях, що робить цей метод найперспективнішим серед перелічених, і в комбінації із вже перевіреними методами ШІ може ввести значне покращення в існуючі підходи до тестування якості програмного забезпечення.

1.3. Огляд відомих інструментів визначення якості коду

SonarLint є одним з найпопулярніших інструментів для визначення якості коду, який використовується в інтегрованих робочих середовищах розробки програмного забезпечення, таких як IntelliJ IDEA, Eclipse, або Visual Studio. Цей інструмент розроблений компанією SonarSource та базується на відкритому програмному забезпеченні.

Він використовує методики статичного структурного аналізу.

SonarLint використовує ряд аналізаторів для оцінювання різних аспектів коду з точки зору його читабельності, безпеки, ефективності та інших критеріїв якості. Ці аналізатори виявляють потенційні проблеми та порушення стандартів програмування, що допомагає розробникам виявляти та виправляти проблеми на ранніх етапах розробки.

Плюси використання SonarLint полягають у здатності підвищити якість коду, зменшити кількість помилок та покращити загальну продуктивність розробників, що при цьому відбувається автоматично. Інструмент надає рекомендації щодо поліпшення коду, що сприяє створенню більш безпечного, ефективного та читабельного програмного забезпечення.

SonarLint визначає запахи коду за допомогою вбудованих аналізаторів, які перевіряють джерело коду на відповідність різним критеріям якості та стандартам програмування. Запахи коду визначаються на основі певних шаблонів або антипатернів, які можуть свідчити про недоліки в дизайні або реалізації коду.

Основні види запахів коду, які можуть визначатися за допомогою SonarLint та подібних інструментів, включають:

Дублювання коду (Code Duplication): інструменти, такі як SonarLint, виявляють схожі частини коду та попереджають про можливі проблеми, пов'язані з не підтримкою та складністю обслуговування дубльованого коду.

Довгі методи та функції: інструменти аналізують розмір методів та функцій, і якщо вони виявляються надто довгими, це може свідчити про погану структуру коду.

Завелика складність методу: вимірюється складність коду за допомогою різноманітних метрик, таких як цикломатична та когнітивна складність, і попереджується про можливі проблеми, пов'язані з розумінням та тестуванням коду.

Зайва складність умов: перевіряється чи використовуються зайві або заплутані умови у логіці коду.

Неправильне використання мовних конструкцій: попередження про невірне використання мовних можливостей, що може призвести до помилок або непередбачених наслідків.

Потенційно небезпечні конструкції: виявлення коду, який може бути потенційно небезпечним або має можливість призвести до помилок чи вразливостей безпеки.

SonarLint також включає аналізатори, які визначають відповідність коду до стандартів програмування.

Однак, слід зазначити найбільшу проблему SonarLint. Зокрема, інструмент може виявляти певні фальшиві позитиви, тобто ситуації, коли код визначається як проблематичний, хоча насправді це не так. Або навпаки, коли код, який є проблематичним, інструмент визначає як такий, що не мітить проблем (докладніше про це описано у розділі 4.1). Така ситуація може виникнути через обмежену здатність інструмента розуміти контекст конкретного додатка чи відсутність специфічних знань про деякі фреймворки чи бібліотеки.

Іншим популярним інструментом є Codacy. Codacy – це інструмент для автоматизованого аналізу коду, який спрямований на виявлення потенційних проблем та поліпшення якості коду в проектах програмного забезпечення. Основна мета Codacy – це спростити процес контролю якості коду, зменшити кількість помилок та виявляти проблеми ще на ранніх етапах розробки.

Він використовує методики статичного структурного аналізу та аналізу безпеки.

Проблеми, які може виявити Codacy:

1. Аналізує код для виявлення повторень та дублювань, що може призвести до погіршення обслуговування та збільшення ймовірності помилок.
2. Визначає відсутність або неповність коментарів та документації, що може ускладнити розуміння коду для інших розробників.
3. Аналізує код на предмет можливих вразливостей, таких як SQL-ін'єкції, виклики з мінливим джерелом та інші проблеми безпеки.

4. Визначає та надає рекомендації щодо проблем, які є характерними для конкретної мови програмування.

5. Виявляє невідповідності до стандартів оформлення коду та інші стилістичні недоліки.

6. Codacy також може виявляти низьке покриття коду тестами, що може свідчити про недостатню тестування деяких частин програми.

Тим не менш, є ряд проблем, які Codacy не може визначити: зрозумілість та читабельність коду, його ефективність та проблеми зі структурою.

Висновки

У сучасному світі, де швидкість розробки та випуску програмного забезпечення стають все більш критичними, питання актуальності та ефективності процесів оцінки та забезпечення якості коду залишаються одним з найбільш важливими.

Хоча код рев'ю визнаний як невід'ємна практика для забезпечення якості та безпеки коду, його мануальний характер є причиною значних витрат часу та людських ресурсів. Вирішення цієї проблеми стає ключовим завданням для забезпечення ефективності процесів розробки.

Тому необхідно врахувати, які характеристики коду впливають на його якість та чи можливо вирахувати їх автоматично. Ремонтопридатність програмного забезпечення, його легкість супроводу та можливість адаптації до змін є критично важливими чинниками. Код, що легко читається та піддається покращенням, забезпечує ефективну та безпечну розробку.

Запахи коду, такі як "довгий метод" чи "великий клас", вказують на можливі проблеми, що можуть ускладнити супровід і розширення програмного продукту.

Використання структурних метрик коду, які вимірюють його складність та розмір, дозволяє об'єктивно оцінити якість та підтримуваність. Інтеграція штучного інтелекту в процес тестування дозволяє виявляти аномалії та робити оцінку якості коду ефективніше.

SonarLint і Codacy виокремлюються серед інструментів для аналізу коду, але мають певні обмеження та можливі фальшиві результати.

Таким чином, комплексний підхід до аналізу коду, що поєднує статичний та динамічний аналіз, а також використання структурних метрик та інтелектуальних інструментів, є найбільш перспективним для досягнення високої якості коду та підтримки програмного забезпечення в довгостроковій перспективі.

РОЗДІЛ 2
МЕТОДИКА ВИЗНАЧЕННЯ ЯКОСТІ КОДУ ІЗ ВИКОРИСТАННЯМ ЗАСОБІВ ШТУЧНОГО ІНТЕЛЕКТУ

2.1. Опис теоретичного підґрунтя, яке забезпечує реалізацію методики

Цей розділ присвячений висвітленню теоретичного підґрунтя, на основі якого була розроблена та реалізована методика оцінки якості програмного коду. Детальне розуміння теоретичних концепцій та методів грає ключову роль у розробці ефективних стратегій оцінки, що сприяють поліпшенню програмного коду та процесів розробки.

2.1.1. Цикломатична та когнітивна складність коду

За Томасом МакКейбом, оцінка складності розробки програмного забезпечення є важливим елементом, оскільки вона безпосередньо впливає на якість створеного продукту. Якість програмного забезпечення визначається такими параметрами, як зрозумілість та вимірюваність, які стали ключовими в контексті теорії програмних метрик.

У своїй роботі він стверджує, що програміст повинен враховувати складність проектування та розуміти його вплив, перш ніж переходити до фази розробки ПЗ. При масштабуванні ПЗ зростає і загроза для якості цього ПЗ, що призводить до серйозних проблем з його обслуговуванням.

Один із підходів до вимірювання складності в архітектурному дизайні - це використання цикломатичної складності. МакКейб описує цей метод як такий, що застосовується до архітектурного ієрархічного проектування для врахування кількості лінійно незалежних шляхів через програму. Це не лише вимірює, але і моделює граф потоку управління, використовуючи математичні техніки для модуляризації програм та ефективного модульного тестування.

Крім цикломатичної складності існує і інша більш точна характеристика коду – когнітивна складність ПЗ. Вона вважається більш комплексним та

ефективним показником порівняно з цикломатичною складністю з кількох теоретичних та практичних причин.

Когнітивна складність - це рівень розумових зусиль, розуміння та обробки, необхідних для розуміння та роботи з певною системою, завданням або частиною інформації. У контексті програмного забезпечення когнітивна складність - це міра когнітивного навантаження та зусиль, пов'язаних із розумінням, підтримкою та роботою з програмними системами. Вона враховує когнітивні процеси, пов'язані з розробкою програмного забезпечення, включаючи розуміння коду, архітектури системи та загальної функціональності програмного забезпечення.

Один з теоретичних поглядів на когнітивну складність в інженерії програмного забезпечення представлений Й. Ванг і В. Чів, які описують когнітивну складність як продукт архітектурної та операційної складності ПЗ на основі дедуктивної семантики [30]. Вони підкреслюють зв'язок між функціональною складністю програмного забезпечення та людською когнітивною складністю. Вони стверджують, що обчислювальна складність програмного забезпечення - це мікропроблема, що стосується алгоритмічного аналізу, тоді як функціональна складність і когнітивна складність - це макропроблеми, що стосуються семантичних властивостей програмного забезпечення і людської когнітивної складності.

Учені визнають популярність метрики цикломатичної складності, запропонованої Томасом МакКейбом у 1976 році, вказуючи на її історичне значення в інженерії програмного забезпечення [4].

Однак теоретичні основи когнітивної складності припускають, що вона виходить за межі структурного потоку програми, беручи до уваги когнітивні процеси, пов'язані з розумінням, проектуванням та підтримкою програмного забезпечення.

Так, Д. Елшофф та М. Маркотті [14] обговорюють обмеження цикломатичної складності, виявляючи недоліки в її здатності ефективно охоплювати когнітивні аспекти складності програм. Це підкреслює теоретичну

прогалину в здатності цикломатичної складності всебічно відображати когнітивні виклики та зусилля, пов'язані з розробкою та супроводом ПЗ.

У свою чергу, Й. Ванг і В. Чів описують додаткові характеристики мір когнітивної складності, які дають уявлення про людське пізнання в інженерії програмного забезпечення, підкреслюючи теоретичну релевантність когнітивної складності для розуміння когнітивних процесів, пов'язаних з розробкою та розумінням програмного забезпечення [30].

Крім того, дослідження зосереджується на формальному вимірюванні когнітивної складності текстів у когнітивній лінгвістиці, підкреслюючи фундаментальну роль когнітивної складності в розумінні, обробці та пошуку текстів. Ця теоретична перспектива підкреслює важливість когнітивної складності в розумінні та обробці текстової інформації, яка може бути поширена на розуміння програмних систем.

Популярною також є метрика когнітивної складності, що надається SonarSource, та її інтеграція в SonarLint і SonarQube, що використовується для оцінки якості програмного забезпечення. Таке практичне застосування вимірювання когнітивної складності відображає його теоретичне підґрунтя в оцінюванні та розумінні програмних систем.

Загалом, теоретичні дослідження підтверджують думку, що когнітивна складність забезпечує більш повне та детальне розуміння складності програмного забезпечення порівняно з цикломатичною складністю. Враховуючи когнітивні процеси, зусилля розуміння та операційні тонкощі, когнітивна складність пропонує більш цілісний погляд на складність програмного забезпечення, що відповідає багатогранній природі розробки та супроводу програмного забезпечення.

2.1.3. Семантичний пошук

Семантичний пошук - це передова пошукова технологія, призначена для розуміння підтексту слів і словосполучень. Його особливість полягає в тому, що

він видає результати пошуку, які відповідають передбачуваному значенню запиту, а не просто буквальному збігу слів у запиті.

Ця методологія пошуку охоплює набір можливостей пошукової системи, яка вміє розпізнавати слова відповідно до намірів користувача і контекстуальних аспектів його пошуку.

Основна мета семантичного пошуку - підвищити точність результатів пошуку шляхом точної інтерпретації природної мови в контексті. Це досягається шляхом узгодження пошукового наміру з семантичним значенням, чому сприяють машинне навчання та штучний інтелект [17].

Семантичний пошук працює на основі векторного пошуку, що дозволяє йому надавати і класифікувати контент на основі відповідності як контекстних, так і цілеспрямованих елементів. Векторний пошук передбачає кодування деталей пошукової інформації у вектори, які є полями пов'язаних термінів або елементів. Згодом ці вектори порівнюються, щоб визначити ступінь їхньої схожості.

У рамках парадигми семантичного пошуку векторний пошук організовано подвійним чином: після запуску запиту пошукова система перетворює запит на вкладення (embeddings), числове представлення даних і пов'язаних з ними контекстів, що зберігаються у векторах. Потім використовується алгоритм k-найближчих сусідів (kNN) для зіставлення векторів наявних документів (що відносяться до тексту в семантичному пошуку) з векторами запиту. Результати генеруються і класифікуються на основі їхньої концептуальної релевантності.

Контекст у рамках семантичного пошуку охоплює додаткову інформацію, таку як географічне розташування пошукача, текстовий контекст слів запиту або історію пошуку пошукача. Таке контекстуальне розуміння допомагає розшифрувати значення слова у великому наборі даних, виявляючи слова, які мають схоже контекстуальне використання.

Основна мета семантичного пошуку - покращити користувацький досвід шляхом ефективної інтерпретації та відповідності намірам користувача. Завдяки глибокому розумінню потреб користувача, що випливають із запиту та його контексту, семантичний пошук упорядковує результати в порядку релевантності.

Крім того, семантичний пошук пропонує можливість персоналізації за допомогою налаштувань категоризації запитів, що дозволяє адаптувати представлення результатів.

Основні пошукові системи, такі як Google і Bing, вже містять елементи семантичного пошуку [19]. Оскільки простори даних стають дедалі складнішими, можливість семантичного пошуку за даними з використанням власного словника стає фундаментальною. Поєднання структурованих запитів з інтерфейсами ключових слів та багатофакторного пошуку було представлено як підхід до семантичного пошуку [3].

Семантичний пошук вже дозволяє значно підвищити якість коду, надаючи розробникам можливість ефективно шукати та знаходити потрібні фрагменти коду на основі запитів природною мовою. Ця технологія дозволяє знаходити семантично пов'язаний код, що може пришвидшує та підвищує ефективність програмування. Моделі пошуку коду на основі глибокого навчання були розроблені для зменшення втручання зашумлених ключових слів і вивчення особливостей коду шляхом векторизації коду, тим самим розпізнаючи семантично пов'язані слова і підвищуючи точність пошуку коду [33]. Крім того, для покращення семантичного пошуку коду було запропоновано використовувати моделі глибокого навчання, такі як Path-based Semantic Code Search (PSCS), що ще раз підкреслює важливість використання передових методів для ефективного пошуку коду [22].

Інтеграція семантичних і структурних збігів, релевантних цілям, у пошук коду, як було показано, підвищує продуктивність пошуку коду за рахунок включення наміру розширення, тим самим підвищуючи релевантність знайденого коду. Крім того, було запропоновано використання навчання з підкріпленням у моделях пошуку коду, таких як QueCos, для збагачення семантики запиту для більш ефективного пошуку коду, що підкреслює потенціал передових методів навчання для покращення семантичних аспектів пошуку коду [23].

Отже, семантичний пошук вже дозволяє розробникам ефективно знаходити семантично пов'язані фрагменти коду на основі запитів природною

мовою. Інтеграція передових моделей глибокого навчання, навчання з підкріпленням і методів зіставлення на основі уваги підкреслює потенціал семантичного пошуку коду для значного підвищення точності, релевантності та ефективності процесів пошуку коду.

Використання семантичного пошуку для забезпечення якості коду має більший функціонал в контексті когнітивної зрозумілості вихідного коду, а значить може напряму впливати не тільки на швидкість, але й на читабельність розробленого ПЗ.

2.1.4. Стандарт ISO/IEC 25010

При визначенні якості ПЗ необхідно розглянути стандарти, які регулюють, який програмний продукт можна вважати якісним.

ISO/IEC 25010 – це стандарт інженерії, який фокусується на вимогах та оцінці якості систем та програмного забезпечення (SQuaRE). Він забезпечує всеосяжну основу для оцінювання якості програмно-містких систем, охоплюючи різні характеристики та підхарактеристики якості. Стандарт призначений для керівництва оцінюванням програмних продуктів і систем, пропонуючи структурований підхід до оцінювання їхніх атрибутів якості.

ISO/IEC 25010 базується на наборі моделі якості та факторів, які є важливими для оцінювання якості програмних продуктів. Він охоплює низку характеристик якості, зокрема функціональну придатність, надійність, продуктивність, сумісність, зручність використання, безпеку, ремонтопридатність і портативність. Ці характеристики далі поділяються на підхарактеристики, забезпечуючи детальну та всеосяжну основу для оцінювання якості програмного забезпечення.

Модель якості продукції, визначена в ISO/IEC 25010, складається з восьми характеристик якості, показаних на рис. 2.1.

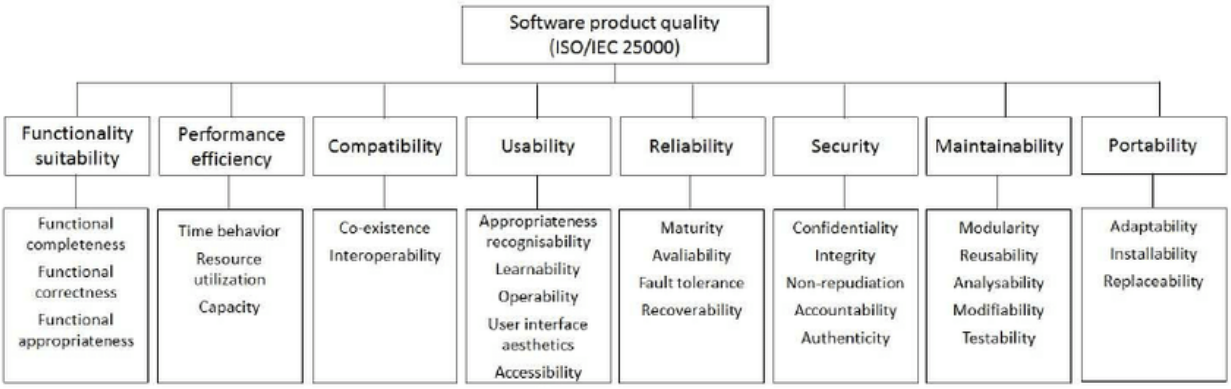

Рис. 2.1. Характеристики якості систем та програмного забезпечення

Опишемо ці характеристики якості:

Функціональна придатність (Functional Suitability) – це ступінь, до якого продукт або система задовольняє як заявлені, так і неявні потреби під час використання у визначених умовах. Ця загальна характеристика охоплює кілька підхарактеристик:

1. Функціональна повнота (Functional Completeness) оцінює ступінь, до якого набір функцій всебічно охоплює всі визначені завдання та цілі користувача.

2. Функціональна коректність оцінює ступінь, до якого продукт або система дає точні результати з необхідною точністю.

3. Функціональна придатність (Functional Appropriateness): оцінює ступінь, до якого функції сприяють виконанню визначених завдань і досягненню поставлених цілей.

Ефективність функціонування (Performance Efficiency) вимірює продуктивність продукту або системи відносно використаних ресурсів за заздалегідь визначених умов. Ця характеристика складається з наступних підхарактеристик:

1. Часова поведінка (Time Behavior) розглядає ступінь, до якого час відгуку і обробки, а також пропускну здатність продукту або системи при виконанні своїх функцій.

2. Використання ресурсів (Resource Utilization) оцінює ступінь відповідності кількості та типів ресурсів, що використовуються продуктом або системою, встановленим вимогам.

3. Продуктивність (Capacity) оцінює ступінь, до якого максимальні межі параметрів продукту або системи відповідають визначеним вимогам.

Сумісність оцінює ступінь, до якого продукт, система або компонент може обмінюватися інформацією з іншими та виконувати необхідні функції, використовуючи те саме апаратне або програмне середовище. Ця характеристика окреслюється наступними підхарактеристиками:

1. Співіснування (Co-existence) оцінює ступінь, до якого продукт може ефективно виконувати свої функції, використовуючи спільне середовище та ресурси з іншими продуктами, без негативного впливу на будь-який інший продукт.
2. Операційна сумісність (Interoperability) вимірює ступінь, до якого дві або більше систем, продуктів або компонентів можуть обмінюватися інформацією та ефективно використовувати спільну інформацію.

Придатність для використання (Usability) вимірює ступінь, до якого продукт або система можуть бути використані конкретними користувачами для досягнення визначених цілей з ефективністю, результативністю та задоволенням у заздалегідь визначеному контексті використання. Ця характеристика складається з наступних підхарактеристик:

1. Доречність та зрозумілість (Appropriateness recognizability) оцінює ступінь, до якого користувачі можуть розпізнати, чи відповідає продукт або система їхнім потребам.
2. Здатність до навчання (Learnability) вимірює ступінь, до якого продукт або система можуть бути використані визначеними користувачами для досягнення цілей навчання з ефективністю, результативністю, відсутністю ризику та задоволенням у визначеному контексті використання.
3. Оперативність (Operability) оцінює ступінь, до якого продукт або система має атрибути, що полегшують його експлуатацію та контроль.
4. Захист від помилок користувача (User Error Protection) вивчає ступінь, до якого система захищає користувачів від помилок.

5. Естетика інтерфейсу користувача (User Interface Aesthetics) враховує ступінь, до якого користувацький інтерфейс робить взаємодію з користувачем приємною і такою, що задовольняє його.
6. Доступність (Accessibility) оцінює ступінь, до якого продукт або система можуть бути використані людьми з широким спектром характеристик і можливостей для досягнення певної мети в заздалегідь визначеному контексті використання.

Надійність (Reliability) вимірює ступінь, до якого система, продукт або компонент виконує визначені функції за визначених умов протягом визначеного періоду часу. Ця характеристика охоплює наступні підхарактеристики:

1. Зрілість (Maturity) оцінює ступінь, до якого система, виріб або компонент відповідає вимогам надійності в умовах нормальної експлуатації.
2. Доступність (Availability) вимірює ступінь, до якого система, виріб або компонент є робочим і доступним, коли це необхідно для використання.
3. Відмовостійкість (Fault Tolerance) оцінює ступінь, до якого система, виріб або компонент працює за призначенням, незважаючи на наявність апаратних або програмних збоїв.
4. Відновлюваність (Recoverability) вивчає ступінь, до якого в разі переривання або збою продукт або система може відновити пошкоджені дані та відновити бажаний стан системи.

Безпека (Security) оцінює ступінь, до якого продукт або система захищає інформацію та дані, гарантуючи, що окремі особи, інші продукти або системи мають відповідний доступ до даних на основі їх типів та рівнів авторизації. Ця характеристика складається з наступних підхарактеристик:

1. Конфіденційність (Confidentiality) вимірює ступінь, до якого продукт або система гарантує, що дані доступні лише тим, хто має на це дозвіл.
2. Цілісність (Integrity) оцінює ступінь, до якого система, продукт чи компонент запобігає несанкціонованому доступу до комп'ютерних програм чи даних або їх модифікації.

3. Невідмовність (Non-repudiation) вимірює ступінь, до якого можна довести, що дії або події мали місце, запобігаючи подальшому запереченню цих подій або дій.
4. Підзвітність (Accountability) оцінює ступінь, до якого дії суб'єкта можуть бути однозначно віднесені до цього суб'єкта.
5. Автентичність (Authenticity) вимірює ступінь, до якого ідентичність суб'єкта або ресурсу може бути доведена як така, що заявлена.

Ремонтопридатність (Maintainability) являє собою ступінь ефективності та результативності, з якою продукт або систему можна модифікувати для покращення, виправлення або адаптації до змін у навколишньому середовищі та вимог. Ця характеристика складається з наступних підхарактеристик:

1. Модульність (Modularity) оцінює ступінь, до якого система або комп'ютерна програма складається з окремих компонентів, що мінімізує вплив змін в одному компоненті на інші компоненти.
2. Можливість багаторазового використання (Reusability) вимірює ступінь, до якого актив може бути використаний у більш ніж одній системі або при створенні інших активів.
3. Аналітичність (Analysability) вимірює ефективність та результативність, з якою можна оцінити вплив запланованих змін на продукт чи систему або діагностувати недоліки чи причини збоїв.
4. Модифікованість (Modifiability) оцінює ступінь, до якого продукт або систему можна модифікувати без внесення дефектів або погіршення існуючої якості продукту.
5. Випробуваність (Testability) вимірює ефективність і результативність, з якою можна встановити критерії тестування для системи, продукту або компонента, а також провести випробування, щоб визначити, чи були виконані ці критерії.

Переносимість (Portability) оцінює ефективність і результативність, з якою система, продукт або компонент можуть бути перенесені з одного апаратного,

програмного забезпечення або іншого операційного чи експлуатаційного середовища в інше. Ця характеристика складається з наступних підхарактеристик:

1. Адаптивність (Adaptability) вимірює ступінь, до якого продукт або система можуть бути ефективно та результативно адаптовані до різних апаратних засобів, програмного забезпечення або інших операційних середовищ чи середовищ використання, що розвиваються.
2. Встановлюваність (Installability) оцінює ступінь ефективності та результативності, з якою продукт або систему можна успішно встановити та/або видалити у визначеному середовищі.
3. Замінність (Replaceability) оцінює ступінь, до якого продукт може замінити інший визначений програмний продукт для тієї ж мети в тому ж середовищі.

ISO/IEC 25010 широко застосовується в різних сферах, зокрема в академічних дослідженнях, промисловості та практиці розробки програмного забезпечення. Він забезпечує спільну мову та структуру для обговорення та оцінювання якості програмного забезпечення, полегшуючи комунікацію та співпрацю між зацікавленими сторонами, які беруть участь у програмних проектах.

2.2.Опис методики визначення якості коду з використанням штучного інтелекту на основі структурного та емпіричного аналізу

Сучасна парадигма розробки програмного забезпечення вимагає від дослідників та практиків найвищого рівня підходів для об'єктивного визначення якості коду. Ця методика базується на синтезі структурних математичних методів та емпіричного статистичного аналізу для визначення ключових аспектів якості програмного коду.

Діаграму методики зображено на рис.2.2:

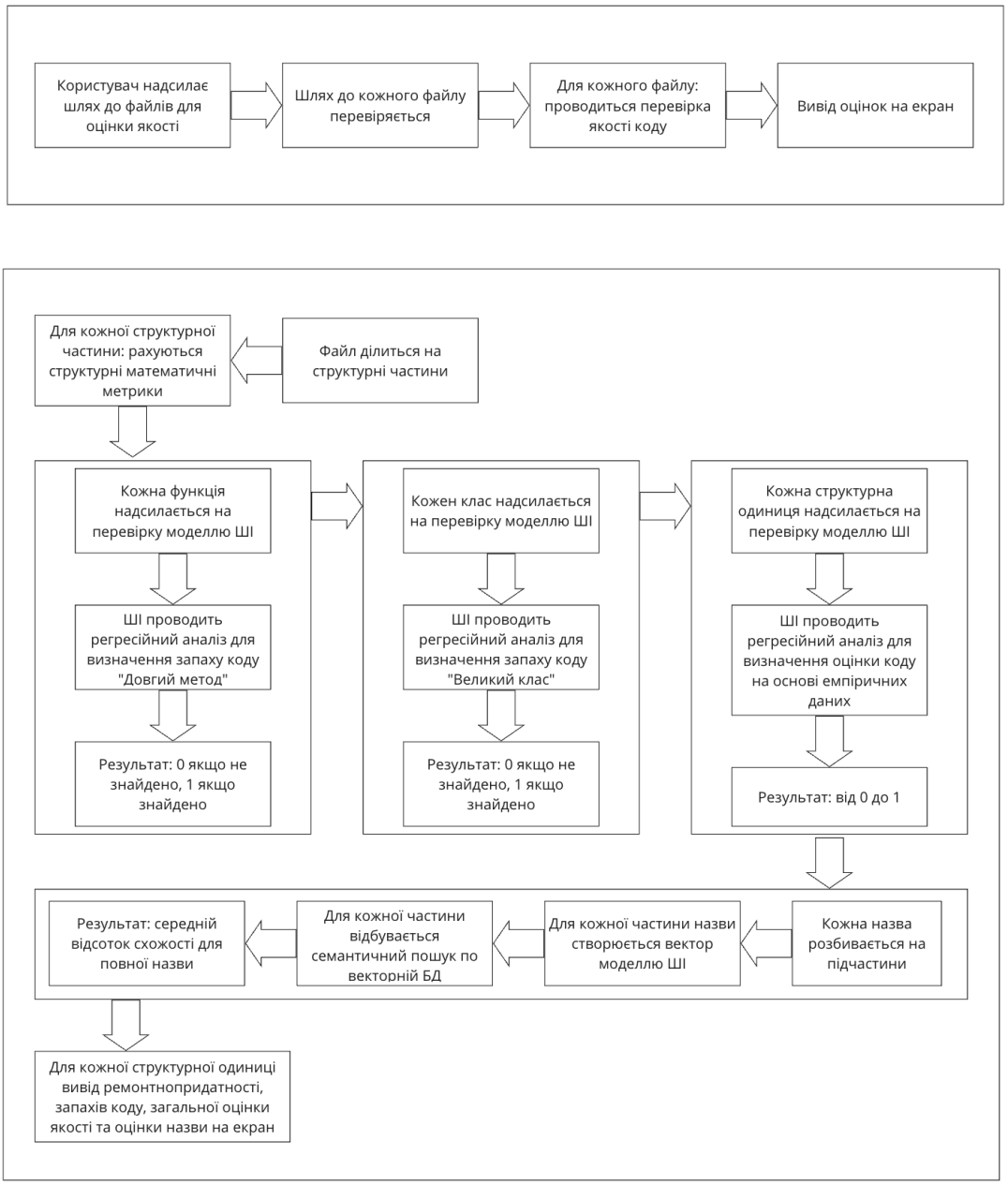

Рис.2.2. Діаграма створеної методики

Центральною частиною методики є використання структурних математичних методів, зокрема цикломатичної та когнітивної складності, а також метрик Холстеда. Ці параметри служать об'єктивними мірниками структурної складності та ефективності кодової бази, дозволяючи систематично аналізувати та порівнювати програмний код.

Додатково до математичних методів, використовуються емпіричні дані, отримані з результатів реальних досліджень. Датасети включають в себе оцінки

від програмістів, що дозволяє враховувати людський фактор та надає емпіричну основу для аналізу якості коду.

До загального аналізу якості коду додається визначення запахів коду, таких як “Довгий Метод” та “Великий Клас”. Для цього використовуються метрики коду та спеціально введені або автоматично визначені ознаки, які слугують підставою для класифікації кодових артефактів.

Однією з ключових особливостей методики є використання семантичної схожості для визначення зрозумілості назв змінних, функцій та класів. За допомогою датасету, який включає сучасні слова та їх семантичну структуру, штучний інтелект оцінює ступінь схожості назв до загальноприйнятих семантичних конструкцій.

Висновки

У ході досліджень, на яких базується розроблена методика визначення якості коду з використанням штучного інтелекту, було проведено аналіз значущих наукових праць в галузі семантичного пошуку в коді, використання підсиленого навчання для покращення семантичної схожості запитів у кодовому пошуку, емпіричні дослідження щодо функціональної складності великих програмних систем, автоматизацію зменшення когнітивної складності програмного коду, використання цикломатичного числа для вимірювання складності програм та інших.

Отримані результати дозволяють стверджувати, що точне оцінювання якості програмного коду неможливе без врахування різноманітних аспектів його структури та семантики. Підхід, який комбінує математичні метрики, емпіричні дані та аналіз семантики, надає більш об'єктивні та повнісні результати, сприяючи розумінню інженерами розробки програмного забезпечення ключових аспектів їхнього коду.

Розроблена методика відзначається великим перевагам у контексті реальних завдань, враховуючи результати досліджень із суміжних галузей та

враховуючи вагомий внесок у вирішення завдань автоматичного визначення якості програмного коду.

Використання інтегрованого підходу, який поєднує структурно-математичні та емпіричні методи засобами штучного інтелекту, надає об'єктивну та науково обґрунтовану оцінку якості коду. Цей метод дозволяє не лише аналізувати та порівнювати кодову базу, але й враховувати її взаємодію зі справжнім середовищем розробки та сприяє покращенню процесу програмної інженерії.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ

3.1. Розрахунок математичних структурних метрик

У визначенні та оцінці якості програмного коду важливо мати доступ до математичних структурних метрик, які дозволяють оцінити його складність, обсяг та інші ключові характеристики. В цьому контексті розроблено клас MetricCalculator, який використовує бібліотеку Radon для автоматизованого розрахунку математичних метрик коду.

Radon - це пакет для аналізу програмного коду на мові програмування Python. Вона надає інструменти для обчислення різноманітних метрик, включаючи цикломатичну складність, метрики Холстеда та інші параметри, які допомагають оцінити важливі характеристики коду.

Опис методів класу MetricCalculator:

1. Конструктор (__init__): ініціалізує об'єкт класу, приймаючи на вхід фрагмент коду та тип фрагменту.
2. Встановлення метрик (set_metrics_for_code): обчислює різні математичні метрики для заданого коду, використовуючи функціонал бібліотеки Radon.
3. Отримання словника метрик (get_metrics_dict): повертає словник, який містить розраховані математичні метрики коду.
4. Розрахунок рядків коду (calculate_lines_of_code): визначає кількість ліній коду в програмі, використовуючи функціонал бібліотеки Radon.
5. Розрахунок складності (calculate_complexity): визначає середню цикломатичну та когнітивну складність коду за допомогою Radon.
6. Розрахунок об'єднаної кількості операторів та операндів (calculate_total_operand_operators): визначає загальну кількість операторів і операндів за метриками Холстеда.
7. Розрахунок обсягу (calculate_volume): визначає обсяг коду за метриками Холстеда.

8. Розрахунок довжини програми (calculate_program_length): визначає довжину програми за метриками Холстеда.
9. Розрахунок складності (calculate_difficulty): визначає складність коду, використовуючи параметри Холстеда, при цьому враховуючи умову, що складність не повинна дорівнювати 0. У випадку, якщо це так, повертається значення 1.
10. Розрахунок інтелекту (calculate_intelligence): визначає інтелектуальну складність коду, яка обчислюється як відношення обсягу коду до його складності.
11. Розрахунок зусиль (calculate_effort): визначає кількість зусиль, необхідних для написання коду, з використанням параметрів Холстеда.
12. Розрахунок кількості помилок (calculate_bugs): визначає кількість очікуваних помилок у коді за допомогою метрик Холстеда.
13. Розрахунок часу розробки (calculate_time): визначає очікуваний час розробки коду на основі параметрів Холстеда.
14. Розрахунок загальної кількості рядків коду (calculate_total_lines): визначає загальну кількість ліній коду, включаючи порожні рядки та коментарі.
15. Розрахунок кількості коментарів (calculate_comments): обчислює кількість ліній, які містять коментарі.
16. Розрахунок однорядкових коментарів (calculate_single_comments): визначає кількість однорядкових коментарів у коді.
17. Розрахунок багаторядкових коментарів (calculate_multi_comments): обчислює кількість багаторядкових коментарів у програмному коді.
18. Розрахунок порожніх рядків (calculate_blanks): визначає кількість порожніх рядків у коді.
19. Розрахунок унікальних операторів (calculate_unique_operators): визначає кількість унікальних операторів за метриками Холстеда.
20. Розрахунок унікальних операндів (calculate_unique_operands): обчислює кількість унікальних операндів, використовуючи параметри Холстеда.

21. Розрахунок загальної кількості операторів (calculate_total_operators): визначає загальну кількість операторів за метриками Холстеда.
22. Розрахунок змістовної лексичної групи (calculate_vocabulary): обчислює розмір лексичної групи, включаючи оператори та операнди.
23. Розрахунок довжини коду (calculate_length): визначає загальну довжину коду за метриками Холстеда.
24. Розрахунок розрахованої довжини (calculate_calculated_length): обчислює розраховану довжину коду за параметрами Холстеда.
25. Розрахунок загальної кількості операндів (calculate_total_operands): визначає загальну кількість операндів за метриками Холстеда.

Ці функції надають широкий набір математичних метрик, які можуть бути використані для більш детального та комплексного аналізу якості програмного коду.

3.2. Визначення “запахів коду” за допомогою засобів штучного інтелекту

В сучасному програмуванні виявлення та вирішення проблем, пов'язаних з архітектурними та структурними аномаліями в програмному коді, визначається як одна з ключових задач для забезпечення високої якості програмного продукту. Запахи коду, що є проявом цих проблем, можуть призводити до збоїв, погіршення підтримки та загроз безпеці програм.

3.2.1. Використання штучного інтелекту в розробці та навчанні моделей у Python

Штучний інтелект займає центральне місце в розробці та навчанні моделей у мові програмування Python. Процес включає етапи від підготовки даних до тренування та оцінки моделей.

Перший етап - отримання та підготовка даних. Зазвичай це включає в себе завантаження датасетів, їх обробку та розділення на тренувальний та тестовий набори.

Після підготовки даних визначається архітектура моделі. Зазвичай використовуються глибокі нейронні мережі, і вибір архітектури залежить від конкретної задачі.

Модель тренується за допомогою оптимізатора та функції втрат. У розглядуваному випадку використовуються “mean_squared_error” як функція втрат та “adam” як оптимізатор. Процес тренування відбувається на тренувальних даних з використанням зазначених гіперпараметрів, таких як кількість епох та розмір пакета.

Після тренування модель оцінюється на тестових даних для визначення її точності. Модель зберігається для подальшого використання.

3.2.2. Інструменти для роботи зі штучним інтелектом

TensorFlow є відкритою бібліотекою машинного навчання та глибинного навчання, розробленою компанією Google. Вона надає інфраструктуру для побудови та навчання різноманітних моделей штучних нейронних мереж. Його графовий підхід до обчислень дозволяє оптимізувати використання ресурсів та прискорювати навчання на графічних процесорах.

Keras — це високорівневий інтерфейс для нейронних мереж, який сприяє швидкості та легкості розробки моделей. У випадку розглядуваного проекту, Keras використовується як інтерфейс до бібліотеки TensorFlow. Його інтуїтивно зрозумілий API дозволяє легко визначати та навчати складні моделі, використовуючи мінімальну кількість коду.

Pandas — це бібліотека для обробки та аналізу даних, яка надає структури даних, такі як DataFrame, і функції для роботи з ними. У процесі навчання моделей та обробки даних, Pandas використовується для ефективного зчитування та обробки даних з CSV-файлів, а також для підготовки вхідних даних для моделей. Його зручний інтерфейс дозволяє виконувати широкий спектр операцій з даними, що є важливим для належної підготовки даних перед навчанням моделей.

3.2.3. Створення та тренування моделей для визначення “запахів коду”

Опис даних.

У даному дослідженні використовуються два набори даних, для визначення запаху довгий метод та запаху великий клас. Кожен рядок містить різні метрики коду та мітку результату - наявність або відсутність запаху коду. За допомогою бібліотеки pandas дані зчитуються, перемішуються та розділяються на тренувальний та тестовий набори.

Створення та навчання моделей.

Для визначення запахів коду використовується метод машинного навчання з використанням нейронних мереж. Для визначення запахів коду створено та навчено дві моделі: одну для виявлення "запахів" у великих класах (large_class), іншу - у довгих методах (long_method). Модель складається з трьох шарів: вхідного, прихованого та вихідного. У вхідному шарі враховуються різні метрики коду. Після певної кількості епох тренування моделі оцінюються на тестовому наборі.

В якості функції втрат обрано mean_squared_error (середнє квадратичне відхилення), що зазвичай використовується для задач регресії. У випадку визначення запахів коду, де вихідна змінна представляє ймовірність наявності запаху, використання цієї функції визначається можливістю оцінки точності моделі за допомогою середнього квадратичного відхилення між прогнозованими та фактичними значеннями.

У якості оптимізатора обрано алгоритм Adam (Adaptive Moment Estimation), який поєднує в собі методи градієнтного спуску та методу спуску моменту. Цей алгоритм дозволяє ефективно адаптувати швидкість навчання для кожного параметра окремо, забезпечуючи стабільну та ефективну збіжність моделі. Вибір Adam обумовлено його успішним застосуванням у різноманітних завданнях машинного навчання та нейронних мереж.

Збереження та оцінка моделей.

Після тренування моделі оцінюються на тестовому наборі, і втрата визначається як показник ефективності.

Після завершення процесу навчання моделі зберігаються для подальшого використання. Під час тренування виводиться інформація про параметри моделі та її ефективність на тестовому наборі.

Науковий контекст та обґрунтування.

Методи машинного навчання та нейронні мережі зазначаються як ефективні інструменти для вирішення завдань класифікації, включаючи виявлення аномалій у програмному коді. Застосування цих методів до визначення запахів коду може значно полегшити процес виявлення та усунення дефектів у програмах.

3.2.4. Використання моделей ШІ для визначення “запахів коду”

В цьому розділі розглянуто використання засобів ШІ для оцінки наявності запахів коду в мові програмування Python. Використовується модель, навчена на даних, що містять інформацію про "запахи коду" та метрики коду.

Для тестування функціоналу використовуються реальні фрагменти коду.

Загальний алгоритм взаємодії з моделями виглядає наступним чином:

1. У файлі з кодом, який необхідно перевірити, виділяються фрагменти коду – класи, функції та методи.

2. Для кожного фрагмента коду визначаються метрики за допомогою MetricCalculator.

3. В залежності від того, є фрагмент класом чи методом, отримані метрики подаються на вхід потрібної моделі.

4. Модель видає оцінку "запаху коду" – 1, якщо “запах коду” знайдено і 0, якщо ні.

5. Результати обробляються, і, в кінцевому підсумку, виводиться інформація про коду.

Таким чином, використання засобів ШІ дозволяє автоматизувати процес оцінки та виявлення "запахів коду", що сприяє покращенню якості та підтримці читабельності програмного коду.

3.3. Визначення якості коду за допомогою засобів штучного інтелекту

Дослідження з оцінки якості коду на основі результатів опитувань програмістів включає в себе використання глибокого навчання для аналізу технічних метрик коду та їх співвідношення з суб'єктивними відгуками експертів. У цьому підрозділі детально розглядається підготовка датасету, архітектура моделі, процес тренування та взаємодія з моделлю для оцінки нових фрагментів коду.

Датасет складається з технічних метрик коду та суб'єктивних відгуків програмістів. Використовується бібліотека Keras для побудови моделі глибокого навчання. Вхідний шар має розмірність, що відповідає кількості технічних метрик. Після цього слідують приховані шари, здатні визначати внутрішні залежності між метриками. Вихідний шар має один нейрон із сигмоїдальною активацією для адаптації до результатів опитувань.

Для тренування моделі використовується стохастичний градієнтний спуск за допомогою оптимізатора “Adam”. Функцією втрат обрана середньоквадратична помилка. Додатково використовується масштабування метрик для покращення стабільності та уникнення перенавчання.

Після тренування модель може оцінювати якість нових фрагментів коду. Вхідні метрики масштабуються та передаються моделі для отримання оцінки. Результат порівнюється із суб'єктивними відгуками програмістів, що дозволяє визначити, наскільки технічні характеристики впливають на сприйняття якості коду.

Перед використанням збереженої моделі необхідно підготувати тестові фрагменти коду. Для цього використовується модуль ast для вилучення фрагментів коду (класів та функцій) з вихідного коду. Отримані фрагменти коду піддаються розрахунку технічних метрик за допомогою MetricCalculator.

Після підготовки тестових фрагментів та розрахунку метрик, використовується попередньо навчена модель для передбачення оцінок якості коду. Вхідні дані для моделі формуються на основі розрахованих технічних метрик.

Цей підхід до оцінки якості коду дозволяє систематизувати технічні та суб'єктивні аспекти. Використання глибокого навчання виявляє складні зв'язки між метриками та експертними відгуками, що сприяє автоматизованій та об'єктивній оцінці якості коду.

3.4. Визначення якості найменування за допомогою засобів семантичного пошуку

Даний розділ присвячений розгляданню методу семантичного пошуку для визначення якості найменувань змінних та функцій у програмному коді.

3.4.1. Інструменти для семантичного пошуку

У ході проведення дослідження використовується модель векторизації тексту, а саме SentenceTransformer зі зразком "intfloat/e5-small-v2". Визначено для генерації числових векторів, які інкапсулюють семантичні характеристики текстових фрагментів. Отримані вектори є представленням семантичного значення тексту, забезпечуючи можливість семантичного аналізу та порівняння текстових даних.

Pinecone виступає в якості інструменту для створення та керування індексами векторів, що призначені для ефективного зберігання та вилучення векторів, отриманих внаслідок векторизації тексту. У даному випадку Pinecone використовується для побудови індексу, спрямованого на вдосконалення швидкодії пошуку та аналізу схожості векторів.

Нижче наведено ключові характеристики та застосування Pinecone у семантичному пошуку:

1. Ефективність: Pinecone пропонує високоефективний механізм пошуку та аналізу векторів, що дозволяє швидко виконувати операції пошуку та порівняння навіть в умовах великої кількості даних.

2. Індексація: сервіс надає можливість побудови індексів, що забезпечують оптимізований доступ до векторів за допомогою унікальних ідентифікаторів.

3. Легкість використання: Pinecone має простий та зрозумілий API, що дозволяє легко використовувати його в різноманітних проектах.

4. Масштабованість: сервіс розроблений з урахуванням можливостей масштабування, що дозволяє використовувати його в проектах різної величини та обсягу.

Pinecone часто використовується для реалізації семантичного пошуку в текстах, зображеннях та інших типах об'єктів. Векторизовані дані, що представляють семантичні характеристики об'єктів, зберігаються та оптимально вилучаються з індексів Pinecone.

Для текстових даних, сервіс може використовуватись для порівняння схожості текстових фрагментів, що спрощує завдання аналізу та категоризації.

Pinecone дозволяє ефективно обробляти великі масиви векторів, забезпечуючи високу швидкодію операцій.

У контексті дослідження семантичного пошуку найменувань змінних та функцій у програмному коді, Pinecone використовується для створення індексу векторів, що представляють семантичні аспекти текстових фрагментів. Це дозволяє використовувати швидкий та ефективний семантичний пошук для аналізу та порівняння найменувань коду.

Стандартні бібліотеки Python, такі як csv та itertools, використовуються для роботи з датасетами та обробки файлів CSV. Окрім цього, використано стандартні інструменти Python для завантаження словників та виконання операцій з мовою програмування.

3.4.2. Створення та завантаження словника даних

Перший етап полягає у завантаженні словника, що містить найбільш часто вживані англійські слова. Словник зчитується з CSV-файлу, та використовується для формування бази порівняння.

Отримані слова піддаються обробці для вилучення непотрібних даних. Застосовується відсіювання слів за довжиною та перевірка на наявність лише

літер у складі слова. Кінцевий словник формується, враховуючи лише перші 30 000 слів за популярністю.

У контексті семантичного пошуку, векторизація використовується для перетворення слів або текстових фрагментів у вектори числового представлення. Це важливий аспект при обробці текстової інформації, оскільки вектори можуть ефективно представляти семантичні аспекти слів та фраз. В цьому випадку, векторизація використовується для створення числових представлень англійських слів. Ці вектори слугують основою для семантичного пошуку, де схожість векторів вказує на семантичну схожість слів чи фраз. Це забезпечує можливість використання семантичного пошуку для порівняння та аналізу найменувань, що допомагає в оцінці якості програмного коду.

Для створення векторів, які будуть використовуватися для семантичного пошуку, використано підготовлений SentenceTransformer з моделі "intfloat/e5-small-v2". Кожне слово було векторизовано та додано до індексу системи Pinecone.

З метою ефективності обробки великої кількості слів, дані було поділено на частини з фіксованим розміром. Для кожної частини векторизовані дані було індексовано та виведено інформацію про кількість оброблених слів та їх успішність.

З метою аналізу та відстеження результатів, слова, що були успішно чи неуспішно оброблені, зберігалися у відповідні файли.

3.4.3. Застосування семантичного пошуку для оцінювання якості іменування

Реалізовано засіб автоматизованої оцінки якості найменувань змінних у програмному коді. Використання сучасних методів обробки природної мови та семантичного аналізу забезпечує об'єктивний підхід до аналізу структури коду та найменувань змінних.

Використовується модель Sentence Transformer, яка забезпечує векторизацію текстових фрагментів. Це дозволяє конвертувати текстові дані в числовий векторний простір, зберігаючи семантичну інформацію.

Використано сервіс Pinecone для зберігання та ефективного використання векторів. Pinecone забезпечує швидкий доступ до векторизованих даних та покращує продуктивність семантичного пошуку.

Алгоритм оцінювання найменувань:

1. Здійснюється конвертація назв змінних до формату snake_case для подальшого аналізу та спрощення інтерпретації.
2. Використовуючи векторизовані дані, здійснюється семантичний аналіз окремих слів. При використанні Pinecone проводиться пошук схожих семантичних сутностей.
3. На основі семантичного аналізу формується оцінка неймінгу змінних. Оцінка виражається у вигляді середнього показника схожості слів. Чим більше назва змінної або функції схожа на існуюче слово, тим більш зрозумілим воно є і навпаки.

Система забезпечує автоматизовану оцінку якості найменувань змінних, що полегшує процес аудиту програмного коду. Застосування методів обробки природної мови та семантичного аналізу забезпечує об'єктивний підхід до оцінки, уникаючи суб'єктивності. Відзначається можливістю поліпшення читабельності коду через однозначні та доречні найменування змінних.

3.5. Опис архітектури та структури розробленого програмного забезпечення

Структура програмного забезпечення визначається ієрархічним розподілом функцій та відповідальностей в межах пакетів та модулів. Кожен елемент цієї структури глибоко інтегрований для досягнення збалансованої та ефективної системи оцінки якості програмного коду. Нижче подано детальний аналіз кожного компонента програми.

Підпакет "general_score" налаштований для загальної оцінки якості коду та включає в себе такі ключові компоненти:

- datasets: модуль для обробки та підготовки даних для навчання та оцінки моделей;
- models: модуль, де розташовані моделі ШІ, відповідальні за загальну оцінку якості коду;
- code_checker: основний модуль, який використовує навчені моделі для оцінки якості загального коду;
- train.py: сценарій для тренування моделей та підготовки даних для навчання.

Підпакет "naming" фокусується на аналізі та оцінці найменувань у коді та включає наступні складові:

- datasets: модуль для підготовки та обробки даних, пов'язаних з найменуваннями;
- code_checker: модуль, який використовує дані та моделі для об'єктивної оцінки якості найменувань у програмному коді;
- dictionary: модуль, що містить словники та ресурси, використовувані для оцінки найменувань;
- vectorizer: модуль для векторизації слів та текстової інформації для використання в навчених моделях;
- writer: модуль для запису словника слів у базу даних pinecone для подальшої оцінки найменувань.

Підпакет "smells" призначений для виявлення архітектурних запахів коду і включає такі самі модулі, як і підпакет “general_score”. Це зроблено для уніфікації інтерфейсу і зрозумілості коду.

Пакет також має в собі багато допоміжних скриптів у корені:

- check_code_quality: головний модуль, який узагальнює функціональність усіх компонентів для повноцінної оцінки якості коду;
- code_extractor: модуль, що відповідає за отримання сутностей певного типу з програмного коду;

- metrics: модуль для розрахунку структурних метрик коду;
- cli: модуль для взаємодії з програмою через командний рядок, що надає зручний інтерфейс користувачеві.

Кожен компонент системи виконує свої унікальні завдання та взаємодіє з іншими для створення інтегрованого середовища для оцінки та покращення якості програмного коду.

В рамках програмного забезпечення визначено кілька ключових класів, що виконують важливі функції в системі оцінки якості програмного коду.

1. Клас "CodeExtractor" відповідає за витягування сутностей з програмного коду. Включає методи для виділення фрагментів коду, класів, функцій та інших програмних елементів.

2. Клас "CodeChecker" є ключовим клас для оцінки загальної якості коду. Використовує різні моделі та метрики для визначення рівня обслуговування та інших аспектів якості кодової бази.

3. Клас "MetricCalculator" відповідає за розрахунок структурних метрик коду, таких як цикломатична складність, обсяг, та інші. Використовується для кількісної оцінки характеристик програмного коду.

4. Клас "NamingCodeChecker" існує для оцінки найменувань у коді. Використовує словникові дані та векторизацію для аналізу та надання балів назвам змінних та функцій.

5. Клас "PineconeWriter" відповідає за запис результатів оцінки найменувань у систему управління векторами Pinecone, забезпечуючи зручний інтерфейс для збереження та використання результатів.

6. Клас "SmellCodeChecker" реалізує функціональність для виявлення "запахів" коду, таких як великі класи чи довгі методи. Використовує моделі та вхідні метрики для визначення проблемних місць у коді.

Ці класи взаємодіють уніфіковано для створення комплексної системи оцінки та вдосконалення якості програмного коду. Це можна побачити на діаграмі класів (рис. 3.1)

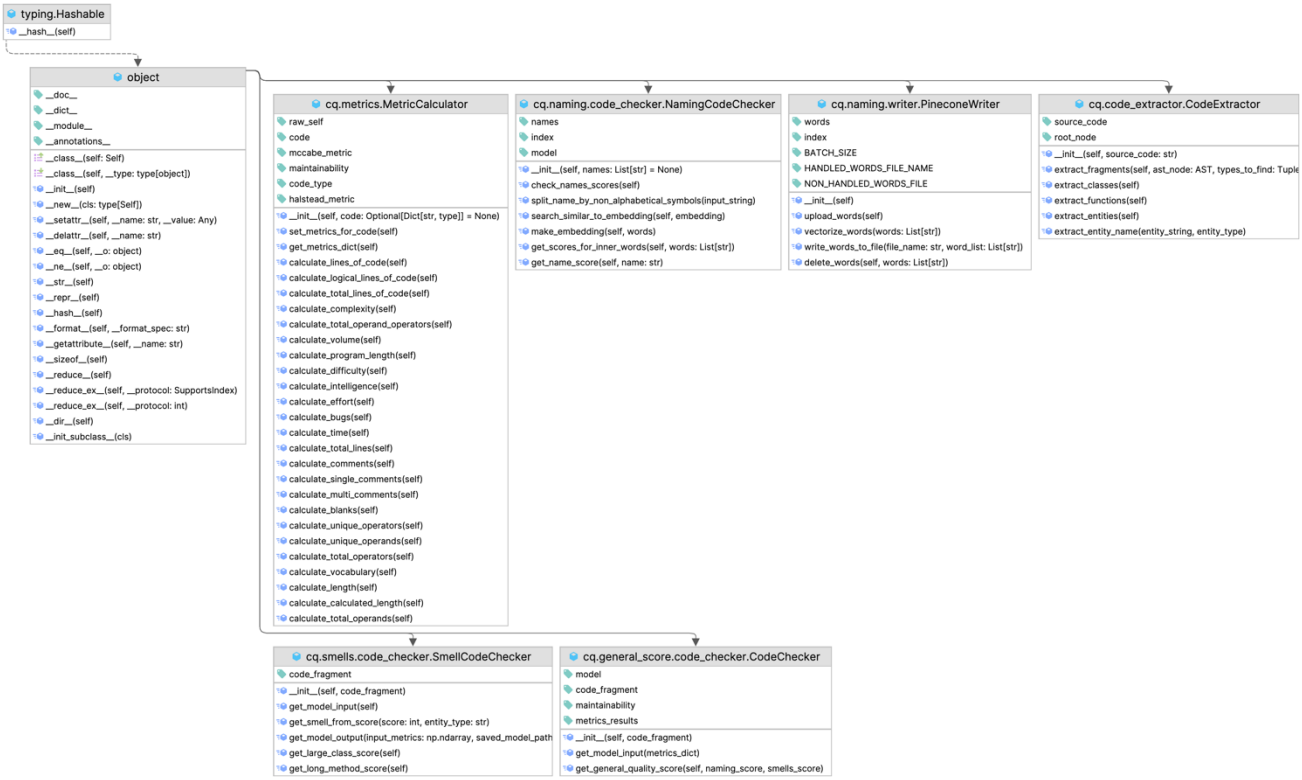

Рис. 3.1. Діаграма класів створеного програмного забезпечення

Висновки

У цьому розділі було розглянуто та впроваджено комплексний підхід до оцінки якості програмного коду з використанням методів машинного навчання, аналізу семантики та векторизації тексту. Оцінювання величезного обсягу інформації, що міститься у програмному коді, стало більш об'єктивним та швидким завдяки використанню сучасних технологій та сервісів.

Один із ключових аспектів - впровадження метрик Холстеда та їх використання для побудови моделей, які оцінюють різні аспекти коду, такі як складність, обсяг та часові витрати. Розроблені моделі дозволяють не лише кількісно оцінювати характеристики коду, але і виявляти можливі "запахи" коду, асоційовані з великими класами та довгими функціями.

У контексті семантичного аналізу та векторизації тексту, використання Sentence Transformer та сервісу Pinecone дозволило реалізувати автоматизовану оцінку найменувань змінних. Цей підхід забезпечує аналіз семантики та визначення ступеня відповідності найменувань загальноприйнятим стандартам.

Проведений аналіз та реалізація функціоналу для оцінки найменувань дозволяє покращити читабельність коду та допомагає розробникам у збереженні стандартів найменувань. Результати використання розроблених моделей та функціоналу є перспективними для інтеграції в сучасні системи контролю версій та редактори коду для надання рекомендацій та підтримки у процесі розробки програмного забезпечення.

РОЗДІЛ 4
ОПИС ВПРОВАДЖЕННЯ МЕТОДИКИ ЯКОСТІ КОДУ ІЗ ВИКОРИСТАННЯМ ЗАСОБІВ ШТУЧНОГО ІНТЕЛЕКТУ

4.1. Оцінювання ефективності ПЗ

Досягнення високої якості програмного коду є критичним завданням у розробці будь-якого програмного забезпечення. Однак, для досягнення цієї мети, необхідно мати ефективні засоби оцінювання якості коду. У цьому контексті, розроблена програма для класифікації якості коду виступає важливим інструментом, який дозволяє розробникам отримувати об'єктивні та комплексні оцінки якості свого коду.

Враховуючи мету проекту, важливо оцінити ефективність створеного ПЗ.

Створена програма запакована у Python-пакет та встановлюється як бібліотека. Для цього використано пакет конфігурації залежностей для мови Python - Poetry.

Використання Poetry для виклику програми забезпечує зручну управління залежностями та використання віртуального середовища.

Команда “poetry run cqa <шлях_до_1_файлу> <шлях_до_n_файлу>“ “ активує процес оцінювання якості коду для визначеного файлу (або будь-якої кількості файлів). Команда “cqa” - абревіатура до code quality assessment (оцінювання якості коду), після неї можна вказати всі файли, які треба оцінити (рис. 4.1).

```
(.venv) ➜  QualiCode git:(main) ✗ poetry run cqa test_files/fill_data.py test_files/hierarchy.py
Started Class Quality Assessment

1/1 [==============================] - 0s 108ms/step
1/1 [==============================] - 0s 32ms/step

Name: FillTestData
----------------------
```

Рис. 4.1. Запуск програми для перевірки декількох файлів

Якщо файлу за шляхом не існує – користувач бачить помилку (рис. 4.2).

```
(.venv) ➜  QualiCode git:(main) ✗ poetry run cqa test_files/nonexistent.py
Usage: cqa [OPTIONS] [FILES]...
Try 'cqa --help' for help.

Error: Invalid value for '[FILES]...': Path 'test_files/nonexistent.py' does not exist.
```

Рис. 4.2. Помилка, яка виводиться на екран, якщо файлу не існує

Програма виводить результати аналізу для об'єктів коду (класів та функцій) вказаного файлу. Кожен об'єкт піддається комплексній оцінці, включаючи тип елементу, показник обслуговування, виявлені "запахи" коду, оцінку імені та загальну якість. Це дозволяє розробникам отримувати детальні результати аналізу для подальшого вдосконалення їхнього коду.

4.1.1. Перевірка ефективності методики для оцінювання якісно написаного методу

Для перевірки ефективності методики визначимо якість коду методу ініціалізації. Особливостями такого методу є легкість виконання, прямий потік програми, бо є ініціалізацією об’єкту і має в собі тільки присвоювання. Ще одною особливістю є стандартна назва методу.

4.1.1.1. Перевірка якості коду створеною методикою

Розглянемо результати виконання для функції “__init__” (рис. 4.3).

Метод “__init__” визначений як функція, має показник обслуговування 10.0/10, виявлені "запахи" відсутні, оцінка імені також 10/10, а загальна якість становить 10.0/10.

Цей вивід свідчить про високу ефективність функції, де всі показники досягають максимальних значень, що підкреслює відповідність функції стандартам та найкращим практикам.

```
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 32ms/step

Name: __init__
-----------------------
Entity type:             function
Maintainability:         10.0/10
Smell Found:             No
Naming score:            10/10
General Quality score:   10.0/10
```

Рис. 4.3. Вивід роботи програми для методу “__init__”

4.1.1.2. Перевірка якості коду статичним аналізатором SonarLint

У даному підрозділі проводиться порівняльний аналіз результатів двох методик оцінки коду: SonarLint та розробленої методики. Метою цього порівняння є визначення відмінностей у підходах до оцінки якості коду та встановлення переваг та недоліків кожного методу.

Результат роботи SonarLint для функції зображено на рис. 4.4.

Рис. 4.4. Вивід роботи SonarLint для методу “__init__”

Звіт по “__init__” функції:

1. Немає виявлених проблем.
2. Відсутність потенційних проблем безпеки.

Порівняльний аналіз:

1. SonarLint не надав конкретної оцінки, тоді як ПЗ розробленої методики поставило максимальну можливу оцінку 10.0/10.
2. Обидві методики не виявили запаху коду в аналізованій функції.
3. За розробленою методика, що ім'я функції має максимально можливу оцінку 10/10, тоді як SonarLint не здійснив оцінювання імені.
4. SonarLint не визначив загальну якість коду, але враховуючи те, що не було виявлено жодних проблем, можна припустити, що оцінка SonarLint – максимально можлива. Розроблена ПЗ за методикою виводить максимальну оцінку 10.0/10.

4.1.1.1. Порівняльний аналіз якості коду з результатами опитування експертів у області розробки ПЗ

Одним зі способів оцінки ефективності методики є порівняння результатів, отриманих від експертів-розробників, і результатів, отриманих в ході використання самої методики. Це порівняння є важливим етапом, оскільки дозволяє оцінити ефективність та об'єктивність використання методики у порівнянні з традиційними підходами до вимірювання якості коду.

У даному підрозділі проводиться аналіз результатів порівняння оцінок коду, отриманих від розробників з різним досвідом, та оцінок, отриманих від автоматизованої методики на основі штучного інтелекту та машинного навчання. Аналіз такого порівняння дозволяє зрозуміти, наскільки точно та адекватно розроблена методика визначає якість коду та чи можна їй довірятися в процесі розробки програмних продуктів.

Отримані оцінки від респондентів та розробленого ПЗ за методикою в практично кожному випадку є ідентичними чи мають дуже високу ступінь збігу.

Таблиця 4.1.

Оцінки експертів та методики для функції з гарною якістю коду.

Рецензент	**Оцінка якості**
Респондент 1	10
Респондент 2	10
Респондент 3	10
Респондент 4	10
Респондент 5	10
Респондент 6	10
Респондент 7	10
Респондент 8	10
Респондент 9	10
Респондент 10	9
Середнє значення відповідей серед респондентів	9,9
Методика	10,0

Важливо відзначити, що середній бал обох наборів оцінок дуже близький, що свідчить про ефективність та точність розробленої методики в оцінці якості коду.

Це порівняння підтверджує, що розроблена методика демонструє високий рівень узгодженості та збігу оцінок з експертним вирішенням реальних розробників, що підкреслює її потенціал у покращенні процесів оцінки якості коду.

4.1.2. Перевірка ефективності методики для оцінювання методу з ознаками “запаху коду”

Мета перевірки ефективності методики для оцінювання методу з ознаками запаху коду полягає в оцінці її здатності вчасно та точно ідентифікувати аномалії в коді, які можуть призвести до недоліків у функціонуванні програмного продукту.

4.1.2.1. Перевірка якості коду створеною методикою

Визначимо якість коду задовгої функції (рис. 4.5):

```
Name: create_users
-----------------------
Entity type:            function
Maintainability:        6.6/10
Smell Found:            Long Method
Naming score:           10/10
General Quality score:  7.1/10
```

Рис 4.5. Вивід роботи програми для методу “create_users”

Функція "create_users" представляє собою блок коду, відповідальний за створення користувачів в системі. Згідно з проведеним аналізом, функція отримала оцінку обслуговуваності на рівні 6.6/10, що може вказувати на певні аспекти, які потребують поліпшення для забезпечення легкої та ефективної обслуговуваності в майбутньому.

Запах коду, виявлений у функції, визначений як "Довгий метод", що може сигналізувати про те, що функція може бути занадто об'ємною та складною для розуміння. Рекомендується розглянути можливості рефакторингу для зменшення довжини функції та поліпшення її читабельності.

Оцінка імені функції становить 10/10, що свідчить про вдале та зрозуміле ім'я, що відображає призначення функції.

Загальна якість коду для даної функції оцінюється на 7.1/10. Ця оцінка враховує різноманітні аспекти якості, і рекомендується в подальшому зосередитися на покращенні обслуговуваності та оптимізації функції для досягнення кращих результатів у загальній якості кодової бази.

4.1.2.2. Перевірка якості коду статичним аналізатором SonarLint

У даному підрозділі проводиться порівняльний аналіз результатів двох методик оцінки коду: SonarLint та розробленої методики. Метою цього порівняння є визначення відмінностей у підходах до оцінки якості коду та встановлення переваг та недоліків кожного методу.

Проведемо аналіз функції за допомогою SonarLint (рис. 4.6)

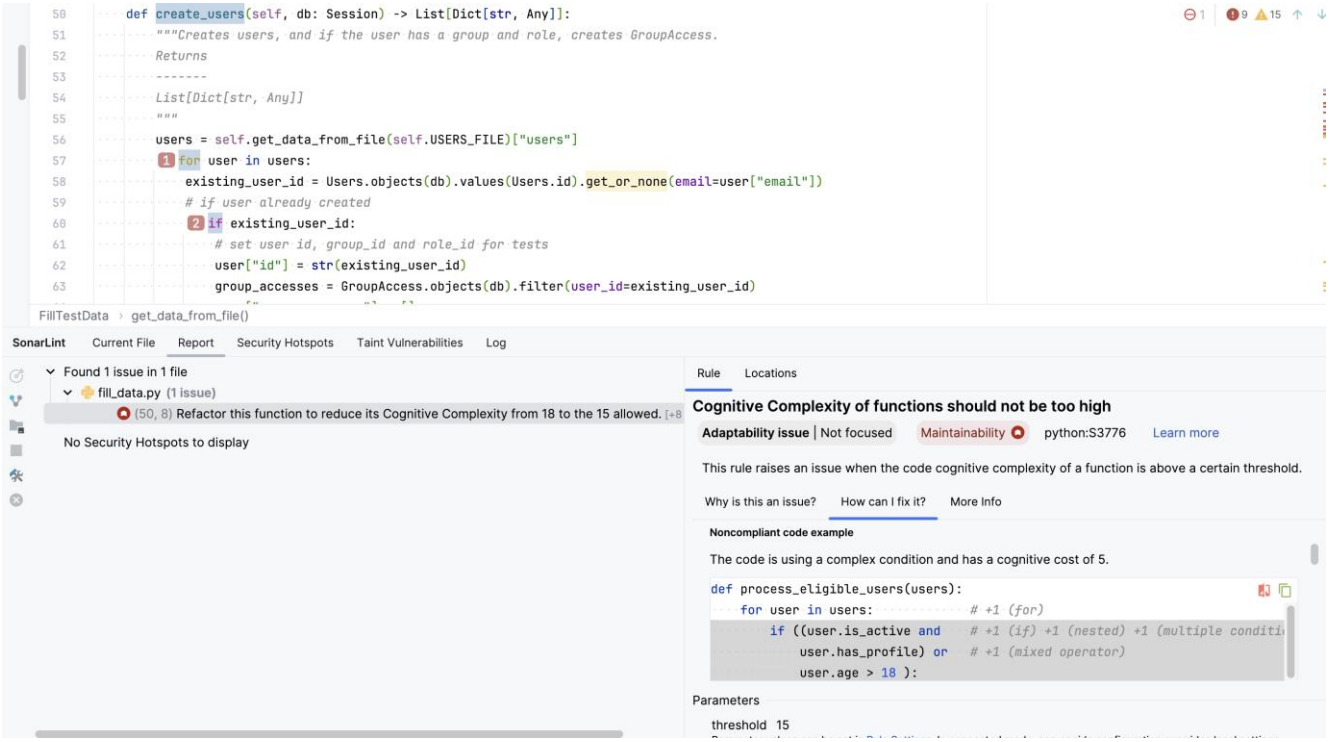

Рис. 4.6. Вивід роботи SonarLint для методу “create_users”

SonarLint виявив, що функція create_users має високий рівень когнітивної складності, який становить 18, що перевищує допустимий ліміт в 15.

Власна методика виявила запах коду "Long Method," що вказує на об'ємність функції та можливу необхідність рефакторингу.

Обидві методики виявили проблему довжини та складності функції.

SonarLint, як інструмент статичного аналізу коду, пропонує конкретні рекомендації щодо зменшення когнітивної складності функції “create_users”. Однак його рекомендації базуються на власно встановлених межах, не враховуючи конкретних вимог чи особливостей проекту.

Створена методика, у свою чергу, аналізує функцію ‘create_users’ і визначає проблему "Long Method," вказуючи на те, що функція може бути важко обслуговувати та розуміти через свою об'ємність. Методика надає підстави для подальшого аналізу та оптимізації коду.

SonarLint надає конкретний підхід до зменшення складності та пропонує рекомендації по покращенню коду, але без урахування контекстуальних особливостей проекту. З іншого боку, створена методика виявляє проблему та вказує на необхідність подальшого аналізу, дозволяючи розробникам приймати індивідуалізовані рішення відповідно до потреб конкретного проекту.

4.1.2.3. Порівняльний аналіз якості коду з результатами опитування експертів у області розробки ПЗ

Порівняємо результати, отримані від експертів-розробників, і результати, отримані в ході використання самої методики.

Таблиця 4.2.

Оцінки експертів та методики для функції з ознаками “запахів коду”.

Рецензент	**Оцінка якості**
Респондент 1	6
Респондент 2	8
Респондент 3	8
Респондент 4	7
Респондент 5	7
Респондент 6	7
Респондент 7	7
Респондент 8	6
Респондент 9	7
Респондент 10	7
Середнє значення відповідей серед респондентів	6,8
Методика	7,1

Середня оцінка якості коду, отримана від експертів, становить 6,8, в той час як створена методика вказує на оцінку 7,1. За загальними показниками, результати дослідження та методика визначення якості коду подібні, при цьому методика дозволяє отримати результати без суб'єктивного впливу експертної оцінки.

4.1.3. Перевірка ефективності методики для оцінювання методу з неточним найменуванням

Мета перевірки ефективності методики для оцінювання методу з неточним найменуванням полягає в оцінці її здатності ідентифікувати когнітивно незрозумілі та нечитабельні назви, які можуть призвести до складнощів у підтримці та слабшій ремонтопридатності програмного коду .

4.1.3.1. Перевірка якості коду створеною методикою

Виміряємо якість для методу з неповними словами в імені (рис. 4.7).

Функція "_srch_thrgh_tree_nds" відповідає за пошук через вузли дерева. Згідно з проведеним аналізом, функція отримала досить високий рівень обслуговуваності – 7.8/10, що свідчить про те, що код функції, ймовірно, добре організований та легко змінюється для подальшого удосконалення.

Запах коду не виявлено (No), що є позитивним показником. Відсутність запахів коду свідчить про те, що функція відповідає базовим стандартам якості.

```
Name: _srch_thrgh_tree_nds
-----------------------
Entity type:            function
Maintainability:        7.8/10
Smell Found:            No
Naming score:           5/10
General Quality score:  7.2/10
```

Рис. 4.7. Вивід роботи програми для методу “_srch_thrgh_tree_nds “

Оцінка імені функції становить 5/10, що може вказувати на те, що ім'я не є досить зрозумілим або не відображає призначення функції повністю. Рекомендується розглянути можливості покращення імені функції для забезпечення більшої ясності та зрозумілості.

Загальна якість коду для даної функції оцінюється на 7.2/10. Ця оцінка враховує різноманітні аспекти якості, і вказує на те, що необхідно удосконалення функції для забезпечення ще більшої чіткості та оптимізації кодової бази.

4.1.3.2. Перевірка якості коду статичним аналізатором SonarLint

SonarLint оцінив функцію “_srch_thrgh_tree_nds” та не виявив жодних проблем. Отже, за результатами аналізу SonarLint вважає, що дана функція не містить потенційних проблем чи неефективностей в коді (рис. 4.8).

В порівнянні з цим, створена методика видає оцінку цього методу, призначаючи йому значення ремонтопридатності на рівні 7.8/10 з загальною оцінкою якості на рівні 7.2/10. Ця різниця в оцінках може бути пояснена різними критеріями, які використовуються SonarLint та створеною методикою.

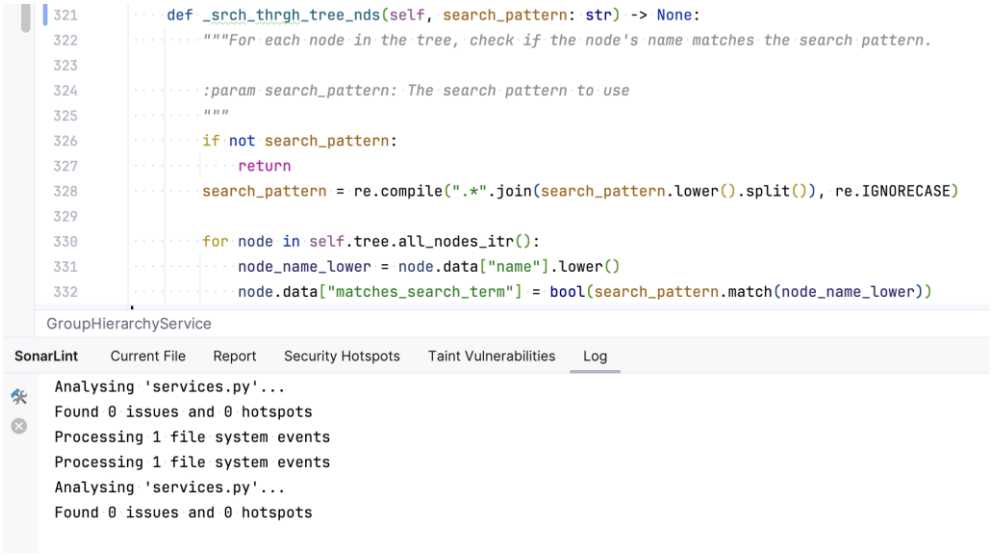

Рис. 4.8. Вивід роботи SonarLint для методу “_srch_thrgh_tree_nds”

Можливі причини розбіжностей в оцінках можуть включати різні методології визначення обслуговуваності, інші аспекти якості коду, або власні налаштування SonarLint. Наприклад, SonarLint фокусується на визначенні конкретних проблем, таких як когнітивна складність чи стандартні патерни, тоді як створена методика є більш комплексною та враховувати різні аспекти якості коду, такі як іменування, розмір функції, загальна структура тощо.

4.1.3.3. Порівняльний аналіз якості коду з результатами опитування експертів у області розробки ПЗ

Порівняємо результати, отримані від експертів-розробників, і результати, отримані в ході використання самої методики.

Таблиця 4.3.

Оцінки експертів та методики для функції з неточним найменуванням.

Рецензент	**Оцінка якості**
Респондент 1	6
Респондент 2	10
Респондент 3	8
Респондент 4	9
Респондент 5	7

Респондент 6	6
Респондент 7	8
Респондент 8	8
Респондент 9	8
Респондент 10	6
Середнє значення відповідей серед респондентів	7,6
Методика	7,1

Середня оцінка якості коду, отримана від експертів, складає 7,6, у той час як створена методика вказує на оцінку 7,1. Як бачимо, результати дослідження та методики визначення якості коду є трохи розбіжними, що може бути зумовлено різним сприйняттям експертами певних аспектів коду.

4.1.4. Перевірка ефективності методики для оцінювання якості коду класу з ознаками “запахів коду”

Цей підрозділ присвячений оцінці ефективності розробленої методики в контексті виявлення та аналізу "запахів коду" у класах програмного коду. Важливим етапом у розробці програмного продукту є забезпечення його стабільності та легкої розширюваності, що можливо лише за умови високої якості коду класів, як сутностей, що зазвичай мають великий функціонал і часто є завеликими та проблемними.

4.1.4.1. Перевірка якості коду створеною методикою

Зробимо оцінку класу (рис. 4.9):

```
Name: FillTestData
-----------------------
Entity type:            class
Maintainability:        5.0/10
Smell Found:            Large Class
Naming score:           10/10
General Quality score:  5.8/10
```

Рис. 4.9. Вивід роботи програми для класу “FillTestData”

Клас "FillTestData" відповідає за наповнення тестових даних. Згідно з проведеним аналізом, клас отримав оцінку обслуговуваності на рівні 5.0/10, що може вказувати на певні аспекти, які потребують уваги та подальшого вдосконалення для поліпшення його обслуговуваності.

Запах коду, виявлений у класі, визначений як "Великий клас" (Large Class), що може свідчити про те, що клас має занадто багато методів та атрибутів і несе на собі занадто багато відповідальностей. Рекомендується розглянути можливості рефакторингу для розподілу функцій та покращення структури класу.

Оцінка імені класу становить 10/10, що свідчить про вдале та зрозуміле ім'я, що відображає призначення класу.

Загальна якість коду для даного класу оцінюється на 5.8/10. Ця оцінка враховує різноманітні аспекти якості, і рекомендується удосконалення структури класу для забезпечення більшої читабельності та зменшення його обсягу.

4.1.4.2. Перевірка якості коду статичним аналізатором SonarLint

Проаналізуємо код класу за допомогою статичного аналізатора SonarLint (рис.4.10).

SonarLint-ом знайдено тільки проблему з методом “create_users”, яку було описано у розділі 4.1.2.2. Жодних проблем зі структурою класу не знайдено, або цей аналізатор не включає клас у список сутностей для оцінки якості.

Оцінка класу окремо враховує особливості його реалізації та функціональності, забезпечуючи більш об'єктивні результати. Оскільки кожен клас може мати унікальні властивості та виклики, оцінка їх відокремлено сприяє збалансованому погляду на якість коду.

Оцінка для окремих класів також полегшує подальший аналіз та впровадження рекомендацій. Розглядання результатів на рівні класів дозволяє команді розробників зосередитися на конкретних виправленнях, спрощуючи процес рефакторингу.

Створена методика у класі базується на розпізнаванні конкретних аномалій, таких як "Великий клас". Це дозволяє точно визначити особливості класу, що можуть впливати на його читабельність та обслуговуваність. Оцінка методики також враховує ім'я класу та загальну якість коду.

З іншого боку, SonarLint спрямований на знаходження конкретних проблем, але оцінка не здійснюється для окремих класів. Такий підхід може бути менш точним, оскільки не враховує унікальні аспекти кожного класу.

Оцінка якості коду на рівні класів з використанням створеної методики дозволяє отримати більш деталізовану інформацію та керувати процесом рефакторингу.

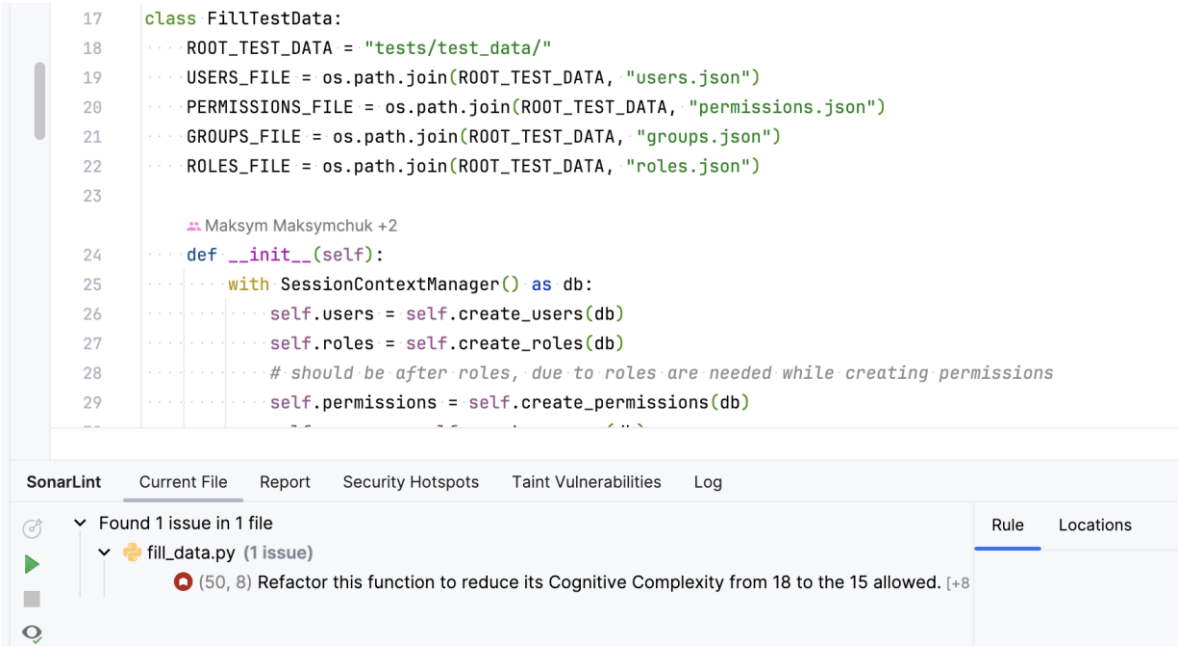

Рис. 4.10. Вивід роботи SonarLint для класу “FillTestData”

4.1.4.3. Порівняльний аналіз якості коду з результатами опитування експертів у області розробки ПЗ

Порівняємо результати, отримані від експертів-розробників, і результати, отримані в ході використання самої методики.

Таблиця 4.4.

Оцінки експертів та методики для класу з ознаками “запахів коду”.

Рецензент	**Оцінка якості**
Респондент 1	4
Респондент 2	6
Респондент 3	6
Респондент 4	5
Респондент 5	6
Респондент 6	5
Респондент 7	5
Респондент 8	5
Респондент 9	5
Респондент 10	5
Середнє значення відповідей серед респондентів	5,2
Методика	5,8

Порівняння з оцінкою експертів-розробників показало відносну консистентність результатів. Середнє значення відповідей респондентів становить 5,2, тоді як методика отримала оцінку 5,8. Це може свідчити про те, що “запах” великий клас може впливати на сприйняття і треба це враховувати при створенні оцінки.

Отже, враховуючи рекомендації методики та порівняльний аналіз з експертною думкою, можна визначити шляхи подальшого удосконалення класу "FillTestData" для забезпечення більшої обслуговуваності та відповідності стандартам коду.

4.1.5. Перевірка часової ефективності методики

Для ефективної роботи програмного забезпечення, особливо фукціонального, важливо мати найменший можливий час виконання.

Ці значення можуть служити ключовими метриками для визначення продуктивності методики в реальних умовах використання.

Для аналізу результатів важливо розглянути кількість символів коду, час обробки та ефективність програми. Звернемо увагу на основні параметри та їх взаємозв'язок.

З таблиці А.1 (див. Додаток А) можна побачити, що середній час обробки для різних сутностей значно коливається.

Дивлячись на дані, можна підтвердити, що кількість символів коду не є ключовим фактором визначення часу аналізу (рис. 4.11). Це може бути результатом ефективного алгоритму обробки.

Тим не менш швидкість виконання аналізу 1-3 секунди для однією сутності може бути проблемою для користувача при великому масштабі проекту або великій кількості сутностей.

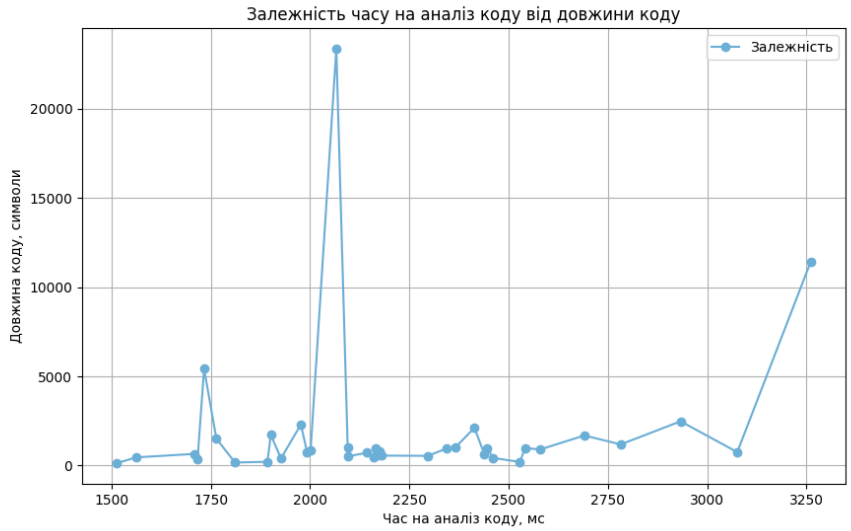

Рис. 4.11. Графік залежності часу на аналіз коду від його довжини

Хоча можливість використання штучного інтелекту для аналізу якості коду вказує на високий рівень автоматизації та інтелектуальної обробки та може допомогти у виявленні складних патернів та потенційних проблем у коді, аналіз

за допомогою ШІ є ресурсо- та часозатратним процесом. Час на виконання аналізу прямо залежить від апаратного забезпечення кінцевого користувача.

Схожу проблему має семантичний пошук. Використання семантичного пошуку через сторонній API дозволяє отримати додаткові контекстуальні дані. Однак у свою чергу може вимагати додаткових ресурсів та бути вразливим до доступності цього зовнішнього сервісу.

Швидкість виконання семантичного пошуку залежить від завантаженості стороннього API, а також від швидкості Інтернету, який має кінцевий користувач.

Тим не менш, час виконання програми залишає певний резерв для оптимізації, але знаходиться на рівні, яке може задовольнити вимоги багатьох реальних застосувань (якщо врахувати, що 3-5 секунд для отримання результату є прийнятним при створенні ПЗ).

4.2. Експериментальне застосування методики

В даному розділі представлено емпіричне дослідження якості програмного коду з використанням розробленої методики оцінювання. Дослідження базується на оцінках 10 експертів у галузі програмування мовою Python із різним рівнем кваліфікації та аналізі результатів, отриманих в ході проведення дослідження.

Таблиця А.2 (див. Додаток А) відображає результати оцінок для різних сутностей. Кожен рядок містить назву сутності, оцінки від кожного респондента, середню оцінку серед респондентів, та оцінку, отриману за розробленою методикою.

4.2.1 Середнє стандартне відхилення

Для аналізу відхилення оцінки, отриманою за методикою, від оцінок експертів було використано статистичний аналіз.

Для порівняння ефективності результатів було розраховано середнє значення оцінки серед респондентів та стандартне відхилення.

Результати аналізу виведені нижче (рис. 4.12), де для кожної сутності порівнюється середнє значення оцінок експертів з отриманою оцінкою. Якщо

відхилення оцінки ПЗ виявляється більшим за стандартне відхилення, то вважається, що оцінка ПЗ вибивається з вибірки.

Так, низьке середнє відхилення свідчить про консистентність оцінок.

```
Відхилення оцінки сутності `mark_nodes_as_disabled` більше за стандартне.
Стандартне відхилення: 0.4216. Відхилення оцінки за методикою: 1.0

--------------------------------------------------------------------------------
Середнє відхидення для всіх сутностей: 0.4056

Process finished with exit code 0
```

Рис. 4.12. Вивід на екран результатів розрахунку стандартного відхилення отриманих результатів

Для сутності "mark_nodes_as_disabled" виявлено відхилення за методикою, яке перевищує нормативне відхилення. Це може свідчити про те, що саме для цієї сутності методика не досить точна або ефективна. Необхідно ретельно вивчити цю сутність та внести корективи до методики для поліпшення її точності.

Загалом сутності не виявили аномалій, і їх відхилення в межах норми, що можна побачити на графіку розподілу оцінок респондентів та методологічних результатів (рис. 4.13).

Середнє відхилення оцінки методики для всіх сутностей: 0.405, що, враховуючи область розробки, кількість оцінок та вплив суб’єктивності, не є показником неефективності методики. Тим не менш, вона вимагає оптимізації та подальшого вивчення.

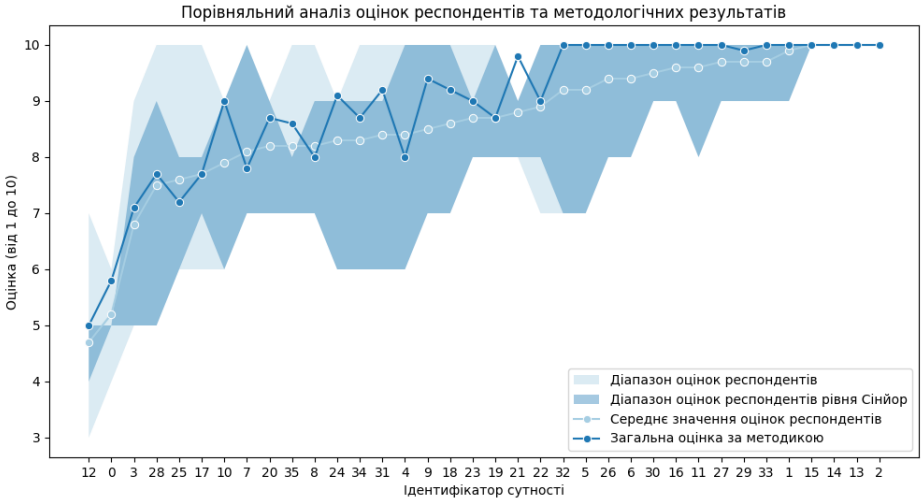

Рис. 4.13. Порівняльний аналіз оцінок респондентів та отриманих результатів

4.2.2. Одновибірковий t-тест

Одновибірковий t-тест є універсальним і широко використовуваним статистичним методом, який застосовується в різних дисциплінах. Його адаптивність до різних характеристик даних, надійність і здатність надавати всебічну інформацію роблять його цінним інструментом для дослідників при перевірці гіпотез і формулюванні значущих висновків на основі отриманих даних.

Одновибірковий t-тест - це статистичний тест, який дозволяє порівнювати середнє значення вибірки з відомим теоретичним (або історичним) середнім значенням.

Порівняємо середні оцінки експертів із оцінкою ПЗ методики для кожної сутності.

Найбільш важливим значенням у розрахунку одновибіркового t-тесту є p. Значення p обчислюється на основі розрахованого t-відношення та кількості наявних ступенів свободи (що дорівнює розміру вибірки мінус 1).

Результатом тесту є значення P, яке вимагає уважної інтерпретації.

Значення p відповідає наступному запитанню: якщо дані відібрані з гаусівської генеральної сукупності із середнім значенням, що дорівнює введеному гіпотетичному значенню, яка ймовірність того, що випадковим чином обрані N

дані матимуть середнє значення, настільки віддалене (або близьке) від гіпотетичного значення, як у даному випадку?

Якщо значення p велике (зазвичай визначається як більше 0,05), то немає достатніх даних для того, щоб стверджувати, що середнє значення вибірки відрізняється від порівнюваного гіпотетичного значення. Це свідчить не про те, що справжнє середнє дорівнює гіпотетичному значенню, а скоріше про відсутність переконливих доказів на користь відмінності.

Якщо значення p невелике (зазвичай менше або дорівнює 0,05), ймовірність того, що спостережувана різниця між вибірковим середнім і гіпотетичним середнім є результатом випадкового вибору, мала. З'являються обґрунтовані підстави відкинути припущення, що різниця є випадковою, і зробити висновок, що генеральна сукупність має середнє значення, відмінне від гіпотетичного.

Так, $p < 0.05$ вказує на те, що є статистично значущі докази на користь того, що середні значення вибірки відрізняються від теоретичного (або історичного) середнього.

А $p > 0.05$ вказує на те, що немає статистично значущих доказів відмінності між середніми значеннями вибірки та теоретичним (або історичним) середнім.

В результаті аналізу для більшості сутностей немає статистично значущих відмінностей між оцінками експертів і створеною методикою (рис. 4.14).

```
Кількість сутностей без статистично значущих відмінностей: 28
Кількість сутностей зі статистично значущими відмінностями: 8
```

Рис. 4.14. Результат аналізу за одновибірковим t-тест-ом

Однак для окремих сутностей (init, get_data_from_file, update_node_data, mark_nodes_as_disabled, тощо) виявлені статистично значущі відмінності, що може вказувати на неспівпадіння між експертними оцінками і методикою для цих конкретних випадків.

Таким чином, дослідження показало, що розроблена методика виявилася ефективною в оцінці якості програмного коду для більшості сутностей, підтверджуючи консистентність оцінок респондентів. Однак для окремих сутностей були виявлені статистично значущі відмінності.

Втім необхідно розширення дослідження на більший обсяг програмного коду та залучення більшої кількості експертів для отримання більш репрезентативних результатів. Також передбачається постійне вдосконалення методики на основі отриманих висновків та відгуків учасників дослідження.

4.2.3. Відповідність стандарту ISO/IEC 25010

Функціональна придатність:

1. Функціональна повнота:

a. Найменування:

- методика оцінює якість назв різних сутностей
- розроблене ПЗ не оцінює якість назв змінних;
- методика не оцінює контекст та співпадіння написаного коду з назвами сутностей.

b. Оцінювання за загальними характеристиками на основі емпіричних даних: працює.

c. Визначення запахів коду:

- визначаються запахи коду: великий клас та довгий метод;
- не визначаються інші запахи коду.

2. Функціональна коректність: у розділах 4.1-4.2 було визначено, що у абсолютній більшості випадків методика дає точні результати.

3. Функціональна придатність: методика показує оцінки різних аспектів якості коду, що є симптомом проблеми і може вказати користувачам, що у коді необхідно внести зміни. Методика не дає рекомендацій, що може бути незрозумілим для користувача.

Ефективність функціонування:

1. Часова поведінка: при створенні методики не було створено вимог з приводу ефективності, тим не менш аналіз часу відповіді було проведено у розділі 4.1.5 та визначено, що, хоча і час відповіді знаходиться у прийнятному проміжку до 3 секунд, він все ж має місце для оптимізації.

2. Використання ресурсів: ПЗ вимагає доступу до мережі інтернет, а також встановлення бібліотек для роботи з нейронними мережами, що може займати велику кількість ресурсів.

3. Продуктивність: ПЗ було випробовувано на різних за розміром та складністю сутностях, на різній кількості файлів, було проведено тести для граничних та помилкових значень і ПЗ не дало збій, тому його можна вважати продуктивним.

Сумісність:

1. Співіснування: ПЗ може ефективно виконувати свої функції, використовуючи спільне середовище та ресурси з іншими продуктами, без негативного впливу на будь-який інший продукт.

2. Операційна сумісність: ПЗ не має можливості обмінюватися інформацію між різними середовищами, і не має необхідності у цьому.

Придатність для використання:

1. Доречність та зрозумілість: ПЗ є зрозумілим та легким для використання.

2. Здатність до навчання: ПЗ може бути використано користувачами для досягнення цілей навчання з ефективністю.

3. Оперативність: ПЗ має опис можливих операцій у консолі, воно легко встановлюється та приймає тільки аргументом список файлів для аналізу. ПЗ не має можливості виключити певні функції, що може вважатися незручним.

4. Захист від помилок користувача: ПЗ захищає користувачів від помилок.

Надійність:

1. Зрілість: ПЗ вимагає подальшого тестування, щоб вказати, наскільки воно відповідає вимогам надійності в умовах нормальної експлуатації.

2. Доступність: ПЗ є робочим і доступним, коли необхідно для використання.

3. Відмовостійкість: ПЗ необхідні ресурси для роботи як-от доступ до Інтернету та встановлені залежності.

4. Відновлюваність: ПЗ не може відновитися, якщо користувач перерве виконання. ПЗ продовжить працювати після відновлення доступу до мережі Інтернет, якщо станеться розрив.

Безпека:

1. Конфіденційність: ПЗ не збирає дані користувачів та має відкритий вихідний код.

2. Цілісність: створене ПЗ не дає зробити змін у кодових даних користувачів і нічого не додає, тому не впливає на сутності, які аналізує.

3. Невідмовність та підзвітність: ПЗ має логування.

Ремонтопридатність:

1. Модульність: ПЗ складається з окремих компонентів, класів, модулів та навіть підпрограм, що мінімізує вплив змін в одному компоненті на інші компоненти.

2. Можливість багаторазового використання: ПЗ має багато спільних перевикористовуваних модулів та класів.

3. Аналітичність: ПЗ має алгоритми для аналізу ефективності, для якого йому необхідні реальні оцінки.

4. Модифікованість: ПЗ може бути легко модифікована. Тим не менш, модифікація моделей (тренування їх на інших датасетах або зміна архітектури може як покращити, так і погіршити результати).

5. Випробуваність: ПЗ складно піддається тестуванню, так як сфера оцінювання у більшості є суб’єктивною і неможливо точно визначити, де система видає неправильний результат.

Переносимість:

1. Адаптивність: через використання ШІ ПЗ залежить від того, у якому середовищі запускається.

2. Встановлюваність: ПЗ встановлюється однією командою через інструмент для роботи з залежностями у мові Python.

3. Замінність: ПЗ може замінити продукти, які були розглянуті у цій роботі. Тим не менш, як було сказано вище, ПЗ вимагає оптимізації та покращень.

Висновки

Запропонована методика якості коду, що ґрунтується на використанні засобів штучного інтелекту, була вичерпно описана та успішно впроваджена в реальному середовищі програмної розробки.

Використання одновибіркового t-тесту послужило підтвердженням того, що для більшості сутностей відсутні статистично значущі відмінності між результатами методики та експертними оцінками. Тим не менше, виявлені випадки статистично значущих відмінностей вказують на необхідність подальшої оптимізації та удосконалення методики.

Висновки дослідження підтверджують ефективність методики у виявленні аномалій та підвищенні якості програмного коду. Однак для досягнення більшої об'єктивності та надійності необхідно проведення додаткових експериментів та розширення досліджень на більший обсяг програмного коду. Доступні результати слід розглядати як стартову платформу для подальшого вдосконалення методики, що дозволить їй ще точніше визначати якість коду та надавати корисні рекомендації для його вдосконалення.

ВИСНОВКИ

Ця робота описує підхід до оцінки якості програмного коду, який визначається аналізом сучасних тенденцій та використанням передових методів, зокрема штучного інтелекту. Актуальність цього підходу обумовлена швидким розвитком технологій та зростанням складності програмного забезпечення, що вимагає ефективних засобів для визначення та поліпшення його якості.

Запровадження штучного інтелекту в оцінку якості коду відкриває нові перспективи завдяки здатності алгоритмів машинного навчання виявляти патерни та залежності великих обсягів даних. Це особливо важливо в контексті сучасних вимог до програмного забезпечення, де важливу роль відіграє не лише синтаксична коректність коду, але й його семантика та здатність адаптуватися до змін у вимогах індустрії.

Ця методика визначення якості програмного коду вирізняється комплексним підходом до оцінки. Її основні складові ефективно інтегрують структурні математичні методи та емпіричний статистичний аналіз для об'єктивного визначення ключових аспектів якості програмного коду.

Центральним елементом методики є використання структурних математичних методів, зокрема цикломатичної складності, а також метрик Холстеда. Ці параметри служать об'єктивними метриками структурної складності та ефективності кодової бази, дозволяючи систематично аналізувати та порівнювати програмний код.

До математичних методів додається використання емпіричних даних, отриманих з результатів реальних досліджень. Оцінки від програмістів включаються в датасети, що дозволяє враховувати людський фактор та надає емпіричну основу для аналізу якості коду.

Для загального аналізу якості коду використовується визначення "запахів" коду, таких як "Довгий Метод" та "Великий Клас". Метрики коду та введені ознаки слугують підставою для класифікації кодових артефактів.

Однією з ключових особливостей методики є використання семантичної схожості для визначення зрозумілості назв функцій та класів. За допомогою датасету, який включає сучасні слова та їх семантичну структуру, штучний інтелект оцінює ступінь схожості назв до загальноприйнятих семантичних конструкцій.

Розроблений підхід дозволяє оцінити не лише технічні аспекти, але і адекватність коду відповідно до функціональних вимог та його можливість підтримувати та розвиватися у майбутньому, що робить його відмінним від традиційних методик.

Отримані результати дослідження підтверджують відсутність статистично значущих відмінностей для більшості сутностей між оцінками, отриманими методикою, та експертними оцінками. Це свідчить про те, що методика ефективно працює для широкого спектру програмних компонентів.

Аналіз відповідності методики стандарту ISO/IEC 25010 підкреслює її відповідність ключовим характеристикам функціональної придатності, ефективності, сумісності, придатності для використання, надійності та безпеки. Це підтверджує об'єктивність та наукову обґрунтованість отриманих результатів, роблячи методику надійним інструментом для оцінки різноманітних аспектів програмного коду.

Таким чином, комплексний підхід забезпечує вдосконалення якості програмного коду, а отже, підвищення продуктивності розробки та зменшення ймовірності виникнення помилок у програмному забезпеченні. Тим самим, цей підхід визначається як ефективний та перспективний у контексті сучасних тенденцій у розробці програмного забезпечення.

Хоча розроблена методика визначення якості програмного коду з використанням штучного інтелекту виявилася ефективною, важливо враховувати деякі вдосконалення, у яких вона потребує.

Інтеграції методики з іншими інструментами розробки, такими як інтегровані середовища розробки (IDE) або системи контролю версій, може сприяти зручності та використанню методики в реальних проектах.

Методика може бути додатково вдосконалена шляхом врахування специфічного контексту вирішення завдань у конкретному проекті. Врахування вимог та особливостей розробки може підвищити адаптивність методики.

Для ще більшої ефективності, методика може потребувати розширення масштабу аналізу для обробки великих обсягів коду та більш широкого спектру проектів.

В цілому, незважаючи на успішність методики, постійне вдосконалення в напрямку врахування нових технологій, розширення аналізу та покращення точності алгоритмів може зробити її ще більш ефективною та універсальною.

Важливим аспектом для подальшого вдосконалення методики є розвиток та покращення використовуваних датасетів. Розширення та удосконалення датасетів, що включають в себе різноманітні типи проектів та галузевих особливостей, може позитивно вплинути на точність результатів. Збільшення репрезентативності датасетів дозволить методиці краще адаптуватися до різноманітних умов та специфікацій проектів.

Додатково, розробка механізмів адаптації до різних ресурсів на комп'ютері користувача може зробити методику більш незалежною та доступною для широкого кола розробників. Можливість автоматично адаптуватися до обмежень апаратного забезпечення та ресурсів може полегшити впровадження методики в різноманітних програмних проектах, незалежно від їхньої складності чи розміру.

Також, розгляд можливостей реалізації частини функціоналу методики у вигляді веб-сервісу чи хмарного рішення може дозволити використовувати методику без значних обчислювальних витрат на стороні користувача. Це сприятиме розповсюдженню та використанню методики в умовах обмежених ресурсів і допоможе зробити її більш еластичною та доступною.

Узагальнюючи, розроблена методика та програмне забезпечення є перспективними напрямками для автоматизованої оцінки якості програмного коду. Процес їхнього вдосконалення та розширення може сприяти подальшому покращенню процесів програмної інженерії та розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Afriyie D. An empirical study investigating the predictors of software metric correlation in application code and test code. [Electronic resource] – Access mode: https://doi.org/10.22215/etd/2020-13954.
2. Arxiv. Choice reviews online. 2007. Vol. 45, no. 02. P. 45–0602–45–0602. [Electronic resource] – Access mode:: https://doi.org/10.5860/choice.45-0602.
3. A semantic search engine in the cloud / F. Amato et al. 2013 eighth international conference on P2P, parallel, grid, cloud and internet computing (3PGCIC), COMPIEGNE, France, 28–30 October 2013. 2013. [Electronic resource] – Access mode:: https://doi.org/10.1109/3pgcic.2013.73 (date of access: 09.12.2023).
4. Automatizing software cognitive complexity reduction / R. Saborido et al. IEEE access. 2022. Vol. 10. P. 11642–11656. [Electronic resource] – Access mode: https://doi.org/10.1109/access.2022.3144743 (date of access: 09.12.2023).
5. Badgett T., Myers G. J., Sandler C. Art of software testing. Wiley & Sons, Incorporated, John, 2011.
6. Balachandran V. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. 2013 35th international conference on software engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013. 2013. [Electronic resource] – access mode: https://doi.org/10.1109/icse.2013.6606642 (date of access: 09.12.2023).
7. Bavishi R., Pradel M., Sen K. Context2Name: a deep learning-based approach to infer natural variable names from usage contexts. 2018. [Electronic resource] – access mode: https://arxiv.org/abs/1809.05193.
8. Berghel H. L., Sallach D. L. Computer program plagiarism detection: the limits of the halstead metric. Journal of educational computing research. 1985. Vol. 1, no. 3. P. 295–315. [Electronic resource] – access mode: https://doi.org/10.2190/0yum-n42t-bw7g-6d5e (date of access: 09.12.2023).

9. B.Senousy M., Sh. Mazen T. Correlations and weights of maintainability index (MI) of open source linux kernel modules. International journal of computer applications. 2014. Vol. 91, no. 7. P. 30–37. [Electronic resource] – Access mode: https://doi.org/10.5120/15894-5052 (date of access: 09.12.2023).
10. Code quality analysis in open source software development / I. Stamelos et al. Information systems journal. 2002. Vol. 12, no. 1. P. 43–60. [Electronic resource] – access mode: https://doi.org/10.1046/j.1365-2575.2002.00117.x (date of access: 09.12.2023).
11. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship / G. Jay et al. Journal of software engineering and applications. 2009. Vol. 02, no. 03. P. 137–143. [electronic resource] – access mode: https://doi.org/10.4236/jsea.2009.23020 (date of access: 09.12.2023).
12. Cyclomatic complexity / C. Ebert et al. IEEE software. 2016. P. 27–29. [Electronic resource] – access mode: https://ieeexplore.ieee.org/document/7725232/.
13. Biase M., Bruntink M., Bacchelli A. A security perspective on code review: the case of chromium. 2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM), Raleigh, NC, USA, 2–3 October 2016. 2016. [electronic resource] – access mode: https://doi.org/10.1109/scam.2016.30 (date of access: 09.12.2023).
14. Elshoff J. L., Marcotty M. On the use of the cyclomatic number to measure program complexity. ACM SIGPLAN Notices. 1978. Vol. 13, no. 12. P. 29–40. [Electronic resource] – access mode: https://doi.org/10.1145/954587.954590 (date of access: 09.12.2023).
15. Exploring the influence of identifier names on code quality: an empirical study / S. Butler et al. 14th european conference on software maintenance and reengineering (CSMR 2010), Madrid, 15–18 March 2010. 2010. [Electronic resource] – access mode: https://doi.org/10.1109/csmr.2010.27 (date of access: 09.12.2023).

16. Incorporating code structure and quality in deep code search / H. Yu et al. Applied sciences. 2022. Vol. 12, no. 4. P. 2051. [Electronic resource] – access mode: https://doi.org/10.3390/app12042051 (date of access: 09.12.2023).
17. Kasenchak R. T. What is semantic search? And why is it important?. Information services & use. 2019. Vol. 39, no. 3. P. 205–213. [Electronic resource] – access mode: https://doi.org/10.3233/isu-190045 (date of access: 09.12.2023).
18. Kim J., Lee E. Understanding review expertise of developers: a reviewer recommendation approach based on latent dirichlet allocation. Symmetry. 2018. Vol. 10, no. 4. P. 114. [Electronic resource] – access mode: https://doi.org/10.3390/sym10040114 (date of access: 09.12.2023).
19. Liu Y., Wang Y., Yang D. DLOSSS: an improved ontology-based semantic web search system. 2014 fifth international conference on intelligent control and information processing (ICICIP), Dalian, China, 18–20 August 2014. 2014. [Electronic resource] – access mode: https://doi.org/10.1109/icicip.2014.7010300 (date of access: 09.12.2023).
20. Mann C. J. H. Object-Oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Kybernetes. 2007. Vol. 36, no. 5/6. [Electronic resource] – access mode: https://doi.org/10.1108/k.2007.06736eae.001 (date of access: 09.12.2023).
21. Mayer A., Sykes A. M. Statistical methods for the analysis of software metrics data. Software quality journal. 1992. Vol. 1, no. 4. P. 209–223. [Electronic resource] – access mode: https://doi.org/10.1007/bf01885771 (date of access: 09.12.2023).
22. Metric and tool support for instant feedback of source code readability. Tehnicki vjesnik - technical gazette. 2020. Vol. 27, no. 1. [Electronic resource] – access mode: https://doi.org/10.17559/tv-20181030091239 (date of access: 09.12.2023).
23. Process aspects and social dynamics of contemporary code review: insights from open source development and industrial practice at microsoft / A. Bosu et al. IEEE transactions on software engineering. 2017. Vol. 43, no. 1. P. 56–75.

[Electronic resource] – access mode: https://doi.org/10.1109/tse.2016.2576451 (date of access: 09.12.2023).

24. Revilla M. A. Correlations between internal software metrics and software dependability in a large population of small C/C++ programs. 2007 18th IEEE international symposium on software reliability engineering, Trollhattan, 5–9 November 2007. 2007. [Electronic resource] – access mode: https://doi.org/10.1109/issre.2007.12 (date of access: 09.12.2023).
25. Sae-Lim N., Hayashi S., Saeki M. An investigative study on how developers filter and prioritize code smells. IEICE transactions on information and systems. 2018. E101.D, no. 7. P. 1733–1742. [Electronic resource] – access mode: https://doi.org/10.1587/transinf.2017kbp0006 (date of access: 09.12.2023).
26. Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis. 2017 the 7th international workshop on computer science and engineering. 2017. [Electronic resource] – access mode: https://doi.org/10.18178/wcse.2017.06.030 (date of access: 09.12.2023).
27. The ACPATH metric: precise estimation of the number of acyclic paths in c-like languages / R. Bagnara et al. 2016. [Electronic resource] – access mode: https://doi.org/10.48550/arxiv.1610.07914.
28. Tiwari U., Kumar S. Cyclomatic complexity metric for component based software. ACM SIGSOFT software engineering notes. 2014. Vol. 39, no. 1. P. 1–6. [electronic resource] – access mode: https://doi.org/10.1145/2557833.2557853 (date of access: 09.12.2023).
29. Venkatasubramanyam R. D., G. R. S. Why is dynamic analysis not used as extensively as static analysis: an industrial study. The 1st international workshop, Hyderabad, India, 1 June 2014. New York, New York, USA, 2014. [electronic resource] – access mode: https://doi.org/10.1145/2593850.2593855 (date of access: 09.12.2023).
30. Wang Y., Chiew V. Empirical studies on the functional complexity of software in large-scale software systems. International journal of software science and computational intelligence. 2011. Vol. 3, no. 3. P. 23–42. [ELECTRONIC

RESOURCE] – ACCESS MODE:: https://doi.org/10.4018/ijssci.2011070103 (date of access: 09.12.2023).

31. Welker K. D. Software maintainability index revisited. Crosstalk - the journal of defense software engineering. 2001. P. 18–21. [electronic resource] – access mode: https://www.osti.gov/biblio/912059.
32. Wurzel Gonçalves P., Çalikli G., Bacchelli A. Interpersonal conflicts during code review. Proceedings of the ACM on Human-Computer Interaction. 2022. Vol. 6, CSCW1. P. 1–33. [Electronic resource] – access mode: https://doi.org/10.1145/3512945 (date of access: 09.12.2023).
33. Zhao W., Liu Y. Utilizing edge attention in graph-based code search. The 34th international conference on software engineering and knowledge engineering. 2022. [Electronic resource] – access mode: https://doi.org/10.18293/seke2022-078 (date of access: 09.12.2023).

ДОДАТОК А

Таблиця А.1.

Залежність часу обробки від довжини сутності, яка піддається аналізу.

Назва сутності	**Кількість символів**	**Час, мкс**
FillTestData	11436	3261194
__init__	463	1561859
get_data_from_file	450	2159134
create_users	5418	1732894
create_permissions	756	1992299
create_groups	658	1709430
create_roles	335	1716333
_get_action_and_expected_result	548	2296712
get_permission_test_data	571	2170703
_crt_grp_accss	790	2174459
crtGrp1212acsss	1021	2094616
HierarchyPermissionsResponse	870	2001188
GroupHierarchyService	23358	2065978
no_permissions	174	1810650
read_users_and_groups_permissions	206	2528453
read_groups_permissions	214	1892433
__init__	138	1513049
update_node_data	403	1926531
get_group_access_nodes	722	2142844
find_or_create_parent_in_tree	1689	2690791
get_groups_by_tree_level	2118	2413904
add_users_to_nodes	1046	2366974
mark_nodes_as_disabled	988	2541617
get_filtered_data	1731	1900907
get_group_tree_as_dict	990	2445762
_make_tree_dict_iterable	958	2343556
_srch_thrgh_tree_nds	561	2179554
_create_tree_from_group_accesses	429	2460822
build_tree_from_db_completely	645	2438730
set_current_children	516	2094784
get	1512	1763847
get_hierarchy_for_navigation	981	2164861
_build_tree_for_assign_hierarchy_groups	905	2579190
_find_user_role_permissions_to_get_hierarchy	1188	2782371
_check_if_group_permissions_to_read_users_exist	756	3075955
_find_permissions_for_group_hierarchy_user_has	2483	2933008
get_assignee_hierarchy	2303	1976676

Таблиця А.2.

Порівняння оцінок, отриманих від респондентів та в ході аналізу методикою.

№	Назва сут-ті	Оц. Р-та № 1	Оц. Р-та № 2	Оц. Р-та № 3	Оц. Р-та № 4	Оц. Р-та № 5	Оц. Р-та № 6	Оц. Р-та № 7	Оц. Р-та № 8	Оц. Р-та № 9	Оц. Р-та № 10	Сер. оц.	Оц. (за мет-ю)
0	FillTestData	4	6	6	5	6	5	5	5	5	5	5.2	5.8
1	__init__	10	10	10	10	10	10	10	10	10	9	9.9	10.0
2	get_data_from_file	10	10	10	10	10	10	10	10	10	10	10.0	10.0
3	create_users	5	9	7	8	7	5	7	7	8	5	6.8	7.1
4	create_permissions	7	8	9	9	9	6	8	10	10	8	8.4	8.0
5	create_groups	8	10	10	10	10	7	10	10	9	8	9.2	10.0
6	create_roles	8	10	10	9	10	9	10	10	10	8	9.4	10.0
7	_get_action_and_expected_result	7	8	9	8	9	7	8	8	10	7	8.1	7.8
8	get_permission_test_data	7	10	8	9	8	7	8	9	8	8	8.2	8.0
9	_crt_grp_accss	7	10	8	10	9	7	9	8	10	7	8.5	9.4
10	crtGrp1212accss	6	9	8	9	9	7	9	8	8	6	7.9	9.0

Продовження таблиці А.2.

11	HierarchyPermissionsResponse	10	10	10	9	10	8	10	10	10	9	9.6	10.0
12	GroupHierarchyService	3	7	5	5	5	4	5	5	4	4	4.7	5.0
13	no_permissions	10	10	10	10	10	10	10	10	10	10	10.0	10.0
14	read_users_and_groups_permissions	10	10	10	10	10	10	10	10	10	10	10.0	10.0
15	read_groups_permissions	10	10	10	10	10	10	10	10	10	10	10.0	10.0
16	update_node_data	9	9	10	10	10	9	9	10	10	10	9.6	10.0
17	get_group_access_nodes	6	10	8	8	8	7	8	8	7	7	7.7	7.7
18	find_or_create_parent_in_tree	8	10	9	8	9	7	9	10	9	7	8.6	9.2
19	get_groups_by_tree_level	8	8	9	8	10	8	8	9	10	9	8.7	8.7
20	add_users_to_nodes	7	8	9	9	9	7	9	8	9	7	8.2	8.7

Продовження таблиці А.2.

21	mark_nodes_as_disabled	8	9	9	9	9	9	9	9	9	8	8.8	9.8
22	get_filtered_data	7	10	9	8	9	8	9	10	10	9	8.9	9.0
23	get_group_tree_as_dict	8	9	9	10	9	8	9	8	8	9	8.7	9.0
24	_make_tree_dict_iterable	7	8	9	9	9	6	9	9	8	9	8.3	9.1
25	_srch_thrgh_tree_nds	6	10	8	9	7	6	8	8	8	6	7.6	7.2
26	_create_tree_from_group_accesses	8	10	10	10	10	8	9	9	10	10	9.4	10.0
27	build_tree_from_db_completely	9	10	10	10	10	10	10	10	9	9	9.7	10.0
28	set_current_children	7	10	8	7	9	5	7	9	7	6	7.5	7.7
29	get	9	10	10	10	10	9	10	10	10	9	9.7	9.9
30	get_hierarchy_for_navigation	9	9	10	9	10	9	10	10	10	9	9.5	10.0

Закінчення таблиці А.2.

31	_build_tree_for_assign_hierarchy_groups	8	10	9	8	9	6	9	8	8	9	8.4	9.2
32	_find_user_role_permissions_to_get_hierarchy	8	10	10	10	10	7	9	9	10	9	9.2	10.0
33	_check_if_group_permissions_to_read_users_exist	9	10	10	9	10	10	10	10	10	9	9.7	10.0
34	_find_permissions_for_group_hierarchy_user_has	7	9	9	9	10	6	8	9	9	7	8.3	8.7
35	get_assignee_hierarchy	8	8	9	9	10	7	8	8	8	7	8.2	8.6