Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.240166.01.07.ПЗ

Виконав студент 2 курсу, група ІПЗм-24-1 _____ Сергій МОСКАЛЬЧУК
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Оксана ЯШИНА
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. пед. наук, доцент _____ Наталія ПРАВОРСЬКА
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ініціали, прізвище

15 грудня 2025 р.

Хмельницький 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
01 09 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Москальчуку Сергію Олеговичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій

Керівник проєкту (роботи) канд. техн. наук, доцент Яшина О.М.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2025 р. № 65
2. Строк подання студентом проекту (роботи) на кафедру 01.12.2025 р.
3. Вихідні дані до проекту (роботи) Матеріали науково-дослідної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз предметної області та рішень з програмного забезпечення.
2 Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій
3 Архітектура програмної реалізації
4 Програмна реалізація
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Форкун Ю.В., доцент	14.12.25	14.12.25
Нормоконтроль	Праворська Н.І., доцент	11.12.25	13.12.25

7. Дата видачі завдання « 01 » вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1. Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	20.10 - 26.09.2025	Виконано
2. Робота над розділом 1 кваліфікаційної роботи - аналіз предметної області та постановка завдання	20.10 - 26.09.2025	Виконано
3 Робота над розділом 2 кваліфікаційної роботи - визначення теоретичних засад визначення якості програмного забезпечення за допомогою метрик	27.10 - 02.11.2025	Виконано
4. Робота над науковими публікаціями, статтями	03.11 - 09.11.2025	Виконано
5. Робота над розділом 3. Проектування архітектури системи для вирішення задачі, розробка вимог.	10.11 - 16.11.2025	Виконано
6 Робота над розділом 4 кваліфікаційної роботи - формалізація, оцінювання та порівняльний аналіз запропонованого підходу	17.11 - 23.11.2025	Виконано
7 Попередній захист кваліфікаційної роботи	24.11 - 30.11.2025	Виконано
8 Узгодження постановки задачі, отриманих результатів та висновків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	01.12-07.12.2025	Виконано
9 Перевірка роботи на наявність плагіату: нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	08.12 - 14.12.2025	Виконано
10 Підготовка до захисту кваліфікаційної роботи	З 15.12.2025	Виконано

Студент _____ Сергій МОСКАЛЬЧУК
Підпис Ініціали, прізвище

Керівник проєкту (роботи) _____ Оксана ЯШИНА
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій».

Автор роботи: Москальчук Сергій Олегович.

Керівник роботи: Яшина Оксана Миколаївна.

Пояснювальна записка: 99 с., 2 дод., 29 джерел.

МЕТРИКИ ЯКОСТІ, ПРОГНОЗУВАННЯ ДЕФЕКТІВ, ІСТОРІЯ ЗМІН, РЕПОЗИТОРІЙ, ДЕФЕКТОСХИЛЬНІСТЬ, АНАЛІЗ КОДУ, IDP-МЕТРИКА.

Мета дослідження - удосконалення методики оцінювання якості програмного забезпечення шляхом розроблення інтегрованих метрик, що враховують історію змін коду та дані про дефекти, для підвищення точності прогнозування дефектів і виявлення вразливих компонентів системи.

Об’єктом дослідження є процес зміни програмного забезпечення в ході розробки та супроводу із властивими йому характеристиками якості та дефектами.

Предметом дослідження є метрики якості програмного забезпечення, що базуються на історії змін коду, характеристиках комітів, активності розробників і даних про дефекти.

Враховуючи мету, предмет, а також об’єкт дослідження сформульовано завдання:

– аналіз, узагальнення і систематизація існуючих моделей оцінювання якості та підходів до прогнозування дефектів;

– обґрунтування вибору напряму удосконалення метрик на основі історії змін коду;

– розроблення нових принципів побудови метрик, що поєднують статичні, процесні та дефектні характеристики;

– створення та дослідження структурних і математичних моделей інтегрованого показника якості (IDP-метрики);

– розроблення програмного забезпечення IDP-Analyzer для автоматизації аналізу репозиторіїв;

– експериментальна перевірка ефективності розробленої метрики в порівнянні з традиційними підходами.

Наукова новизна:

1. вперше одержано інтегрований показник IDP, який поєднує інформацію про змінність коду, характеристики комітів, структурні метрики та історію дефектів;

2. удосконалено методики аналізу дефектосхильності шляхом поєднання статичних і процесних показників у єдину модель;

3. дістало подальшого розвитку застосування алгоритмів реконструкції дефектних комітів у контексті інтеграції з метриками еволюції ПЗ.

Практичне значення отриманих результатів. Полягає у створенні інструменту IDP-Analyzer, який дозволяє автоматизувати процес аналізу великих репозиторіїв, визначати дефектонебезпечні модулі та інтегрувати оцінювання якості у DevOps- та CI/CD-процеси. Отримані результати можуть бути застосовані при побудові автоматичних систем контролю якості, у процесах перегляду коду, у системах прогнозування дефектів та при оптимізації процесів супроводу ПЗ.

В ході проведення даного дослідження використано методи аналізу програмних репозиторіїв, методи оцінювання якості ПЗ; SZZ-алгоритм визначення дефектних комітів; статистичні методи.

05.12.2025 _____

ABSTRACT

Master's thesis: «Improving software quality metrics by taking into account code change history and defects in version control systems».

Author: Serhii Moskalchuk.

Head of work: Oksana Yashyna.

Master's thesis consists of: 99 pages of the general text, 2 supplements, 29 literature sources.

QUALITY METRICS, DEFECT PREDICTION, CHANGE HISTORY, REPOSITORY, DEFECT PRONENESS, CODE ANALYSIS, IDP-METRIC.

The aim of the study is to improve software quality assessment methods by developing integrated metrics that take into account code change history and defect data to improve defect prediction accuracy and identify vulnerable system components.

The object of the study is the process of software change during development and maintenance, with its inherent quality characteristics and defects.

The subject of the study is software quality metrics based on code change history, commit characteristics, developer activity, and defect data.

Taking into account the purpose, subject, and object of the study, the following tasks were formulated:

– analysis, generalisation, and systematisation of existing quality assessment models and approaches to defect prediction;

– justification of the choice of direction for improving metrics based on code change history;

– development of new principles for building metrics that combine static, process and defect characteristics;

– creation and study of structural and mathematical models of an integrated quality indicator (IDP metrics);

– development of IDP-Analyzer software for automating repository analysis;

– experimental verification of the effectiveness of the developed metrics in comparison with traditional approaches.

Scientific novelty:

1. For the first time, an integrated IDP indicator has been obtained, combining information about code variability, commit characteristics, structural metrics, and defect history.

2. Methods for analysing defect susceptibility have been improved by combining static and process indicators into a single model.

3. The application of algorithms for reconstructing defective commits in the context of integration with software evolution metrics has been further developed.

Practical significance of the results obtained. It consists in creating the IDP-Analyzer tool, which allows automating the process of analysing large repositories, identifying defect-prone modules, and integrating quality assessment into DevOps and CI/CD processes. The results obtained can be applied in the construction of automatic quality control systems, in code review processes, in defect prediction systems, and in the optimisation of software maintenance processes.

In the course of this study, methods of software repository analysis, methods of software quality assessment, the SZZ algorithm for determining defective commits, and statistical methods were used.

05.12.2025

ЗМІСТ

ВСТУП

Актуальність дослідження. Аналіз української та зарубіжної науково-технічної літератури і результатів пошуку за тематикою оцінювання якості програмного забезпечення показав, що на сьогодні практично розв’язані такі задачі, як розроблення статичних метрик коду, формування моделей якості, а також побудова окремих моделей прогнозування дефектів на основі статистичних характеристик та машинного навчання. Значною мірою вивченими є також підходи до визначення дефектних комітів, аналізу показників активності розробників та дослідження зв’язку між зв’язністю коду і дефектністю.

Однак, у галузі все ще існує достатньо задач, на розв’язання яких спрямовані зусилля сучасних дослідників. Зокрема, подальшого вивчення потребують: інтеграція статичних та еволюційних метрик у єдиний показник якості; підвищення точності прогнозування дефектів за рахунок урахування історичних закономірностей змін коду; формування універсальних метрик, що працюють для різних типів програмних проєктів; зменшення залежності моделей від конкретного домену та збільшення їхньої інтерпретованості.

Над цими задачами працюють такі провідні науковці, як T. Zimmermann, A. Zeller, A. Hassan, R. Holt, C. Bird, N. Nagappan, M. Lanza, R. Robbes, а також сучасні дослідницькі групи у напрямах Mining Software Repositories (MSR) та Defect Prediction. Ними розроблені та впроваджені у практику моделі прогнозування дефектів, методики реконструкції дефектних змін, моделі Just-in-Time Quality Assurance, а також комплексні підходи до аналізу історії змін у великих програмних системах.

Сьогодні світові тенденції розвитку систем контролю якості ПЗ пов’язані з розв’язанням задач автоматичного аналізу великих репозиторіїв, підвищенням точності виявлення дефектонебезпечних компонентів, інтеграцією інструментів прогнозування у CI/CD-процеси та розвитком моделей, що поєднують структурні,

процесні та історичні метрики. Значна увага приділяється методам Data-Driven DevOps, оцінюванню технічного боргу, використанню глибокого навчання та дослідженню поведінкових характеристик розробників.

Актуальність роботи полягає в необхідності створення інтегрованих метрик якості програмного забезпечення, що враховують не лише властивості коду, але й історію його змін та історію дефектів. Такий підхід дозволяє підвищити точність прогнозування дефектів, зменшити витрати на супровід програмних систем та забезпечити вищу надійність і стабільність програмного продукту. Актуальність також зумовлена потребами сучасних українських і світових ІТ-компаній у засобах раннього виявлення потенційно проблемних компонентів коду та підтримки автоматизованих процесів забезпечення якості.

Виконана кваліфікаційна робота має взаємозв’язок із науковими напрямами кафедри інженерії програмного забезпечення Хмельницького національного університету, зокрема з дослідженнями в галузі оцінювання якості ПЗ, аналізу еволюції програмних систем, розроблення метрик якості, а також створення інструментів автоматизованого аналізу програмних репозиторіїв.

Мета дослідження - удосконалення методики оцінювання якості програмного забезпечення шляхом розроблення інтегрованих метрик, що враховують історію змін коду та дані про дефекти, для підвищення точності прогнозування дефектів і виявлення вразливих компонентів системи.

Об’єктом дослідження є процес зміни програмного забезпечення в ході розробки та супроводу із властивими йому характеристиками якості та дефектами.

Предметом дослідження є метрики якості програмного забезпечення, що базуються на історії змін коду, характеристиках комітів, активності розробників і даних про дефекти.

Враховуючи мету, предмет, а також об’єкт дослідження сформульовано такі завдання:

– аналіз, узагальнення і систематизація існуючих моделей оцінювання якості та підходів до прогнозування дефектів;

– обґрунтування вибору напряму удосконалення метрик на основі історії змін коду;

– розроблення нових принципів побудови метрик, що поєднують статичні, процесні та дефектні характеристики;

– створення та дослідження структурних і математичних моделей інтегрованого показника якості (IDP-метрики);

– розроблення програмного забезпечення IDP-Analyzer для автоматизації аналізу репозиторіїв;

– експериментальна перевірка ефективності розробленої метрики в порівнянні з традиційними підходами.

Наукова новизна:

– вперше одержано інтегрований показник IDP, який поєднує інформацію про змінність коду, характеристики комітів, структурні метрики та історію дефектів;

– удосконалено методики аналізу дефектосхильності шляхом поєднання статичних і процесних показників у єдину модель;

– дістало подальшого розвитку застосування алгоритмів реконструкції дефектних комітів у контексті інтеграції з метриками еволюції ПЗ.

Практичне значення отриманих результатів. Полягає у створенні інструменту IDP-Analyzer, який дозволяє автоматизувати процес аналізу великих репозиторіїв, визначати дефектонебезпечні модулі та інтегрувати оцінювання якості у DevOps- та CI/CD-процеси. Отримані результати можуть бути застосовані при побудові автоматичних систем контролю якості, у процесах перегляду коду, у системах прогнозування дефектів та при оптимізації процесів супроводу ПЗ.

В ході проведення даного дослідження використано методи аналізу програмних репозиторіїв, методи оцінювання якості ПЗ; SZZ-алгоритм визначення дефектних комітів; статистичні методи.

Відповідно до теми кваліфікаційної роботи опубліковані тези» на конференції «Актуальні проблеми комп’ютерних наук (АПКН-2025)».

1 ТЕОРЕТИЧНИЙ ВИКЛАД ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області

Якість програмного забезпечення (ПЗ) традиційно визначається як ступінь, до якої система, компонент або процес задовольняє визначені вимоги та потреби чи очікування користувачів [1, 2]. Існують стандартизовані моделі якості ПЗ, що встановлюють основні характеристики якості. Широко відомим є стандарт ISO/IEC 9126:2001 (нині оновлений до ISO/IEC 25010:2011), який виділяє вісім основних характеристик якості програмного продукту:

- функціональну придатність,
- продуктивність (ефективність використання ресурсів),
- сумісність,
- зручність використання,
- надійність, безпеку,
- супроводжуваність;
- портативність.

Кожна з цих характеристик деталізується через підхарактеристики. Такі моделі надають методологічну основу для оцінювання якості: вони визначають атрибути якості та пропонують метрики або критерії для їх вимірювання. Наприклад, супроводжуваність ПЗ може оцінюватися через показники складності коду, зв’язаності модулів тощо, а надійність – через частоту відмов або щільність дефектів.

Щоб зробити якість вимірюваною, в Software Engineering застосовуються метрики програмного забезпечення [3]. Метрика – це числова міра певної характеристики ПЗ, наприклад, розміру, складності або кількості дефектів). Метрики дозволяють кількісно оцінити внутрішні властивості коду та процесу розробки, що впливають на якість. Відповідно до методики Goal/Question/Metric (GQM), цілі оцінювання якості конкретизуються у формі запитань, відповіді на які

отримуються через зібрані метрики. Такий підхід допомагає пов’язати метрики із високорівневими цілями забезпечення якості.

Метрики поділяють на метрики продукту (внутрішні властивості коду) та метрики процесу (характеристики процесу розробки і супроводу). До метрик продукту належать, зокрема, розмір коду (кількість рядків, LOC), складність алгоритмів (цикломатична складність тощо), рівень зв’язності модулів (cohesion/coupling), щільність документації та ін. Метрики процесу відображають динаміку розробки: наприклад, кількість змін у коді за проміжок часу, кількість виправлених дефектів, активність розробників (кількість комітів, авторів файлу тощо). Наведемо приклади метрик обох типів (табл. 1.1).

Таблиця 1 – Приклади метрик

Приклади метрик коду (продукт)	**Приклади метрик процесу (історія змін)**
Розмір коду (LOC). Кількість рядків коду в модулі або системі; більші за розміром модулі часто складніші і потенційно дефектніші.	Обсяг змін (Code Churn). Кількість доданих, змінених або видалених рядків коду за певний період; різке зростання цього показника корелює зі збільшенням дефектів.
Складність (цикломатична). Міра складності коду через кількість незалежних шляхів виконання (гілок) в програмі; високі значення ускладнюють тестування і супровід.	Частота змін (кількість комітів). Скільки разів файл або модуль змінювався у системі контролю версій; часті зміни можуть свідчити про нестабільність або проблемність компонента.
Зв’язаність та зчеплення. Метрики зв’язаності (cohesion) оцінюють єдність функціоналу класу/модуля, а зчеплення (coupling) – кількість залежностей між модулями; сильне зчеплення і низька зв’язаність ускладнюють супровід і можуть призводити до дефектів.	Вклад розробників. Кількість розробників, що змінювали файл, або показники досвіду (наприклад, сумарна кількість комітів автора); вищий досвід може знижувати дефекти, тоді як часта зміна авторів може підвищувати ризик дефектів.

Продовження таблиці 1

Приклади метрик коду (продукт)	**Приклади метрик процесу (історія змін)**
Дублювання коду та «code smells». Частка скопійованого коду, наявність антипатернів чи поганих практик програмування; ці фактори погіршують якість і супроводжуваність.	Історичні дефекти. Кількість знайдених раніше дефектів у модулі; модулі, що мали багато багів у минулому, часто залишаються проблемними надалі (ключовий індикатор для прогнозування).

На практиці оцінювання якості ПЗ здійснюється через поєднання процедур контролю якості (огляди коду, тестування тощо) та автоматизованого аналізу метрик. На рівні процесу існують моделі зрілості (CMMI), стандарти якості (ISO 9001), які задають організаційні рамки забезпечення якості. Однак, для безпосередньої оцінки властивостей ПЗ застосовують статичний аналіз коду (аналіз метрик, виявлення code smells, перевірку відповідності коду стандартам) та динамічний аналіз (метрики на основі виконання програм, наприклад, покриття коду тестами, частоту збоїв, продуктивність тощо). Статичні метрики дозволяють оцінити якість на ранніх етапах розробки, ще до появи дефектів у експлуатації, – наприклад, високі значення складності або взаємозв’язаності можуть слугувати сигналом про потенційно ненадійний або важкий у підтримці код. Як зазначається в літературі, використання лише вихідного коду та його метрик дозволяє виявляти дефектонебезпечні фрагменти на ранніх стадіях, знижуючи витрати на виправлення помилок. З іншого боку, аналіз історичних даних (динаміки змін і дефектів) дає змогу врахувати фактичну поведінку системи в процесі її еволюції. Тому сучасні підходи до оцінки якості поєднують ці два аспекти: внутрішні метрики коду та інформацію з систем контролю версій і баг-трекерів.

Методологічні підходи до оцінювання якості ПЗ мають свої переваги та обмеження. Стандартні моделі якості (ISO/IEC 25010 та попередні) забезпечують системність і повноту охоплення атрибутів якості, проте вони носять доволі узагальнений характер. Без прив’язки до конкретних метрик такі моделі важко

застосувати безпосередньо – потрібна декомпозиція вимог якості до рівня вимірюваних величин. Тут на допомогу приходять метрики: вони дають операціоналізацію якості. Наприклад, замість абстрактного поняття «супроводжуваність» метрики дозволяють виміряти конкретні показники коду, що впливають на супроводжуваність (складність, дублювання, документованість тощо). Перевага метрик – їх об’єктивність та кількісність: вони дозволяють відстежувати прогрес якості, порівнювати альтернативні рішення на основі чисел, а не інтуїції. Недоліком є те, що окремі метрики не завжди повно відображають якість: наприклад, високий показник покриття коду тестами не гарантує відсутності критичних дефектів, а кількість рядків коду сама по собі не враховує складності алгоритму.

Внутрішні метрики продукту мають ту перевагу, що вони доступні рано і відносно легко автоматично обчислюються для будь-якого коду. Вони не залежать від наявної історії, тобто достатньо статичного аналізу вихідного тексту. Це робить їх корисними для превентивної оцінки, наприклад, інструменти на кшталт SonarQube можуть одразу після коміту коду виявити підвищення комплексності чи появу кодових помилок, сигналізуючи розробникам про потенційні проблеми. Слабка сторона статичних метрик – обмежена здатність відобразити семантику і реальний вплив змін. Як відзначають дослідники, традиційні метрики коду не можуть розрізнити фрагменти, які мають однакову структуру і складність, але реалізують різну функціональність [10]. Наприклад, якщо переставити кілька операторів місцями, метрики типу кількість рядків чи кількість викликів функцій залишаться тими самими, хоча логіка коду могла змінитися кардинально. Тобто глибинні семантичні відмінності не фіксуються простими метриками розміру чи структури. До того ж, знайдені статистичні кореляції не обов’язково означають причинність – метрики можуть корелювати з дефектами, але не прямо їх викликати. Це обмежує пояснювальну силу суто внутрішніх метрик.

Метрики процесу та історії, наприклад, змін коду, дефектів враховують динаміку розвитку ПЗ. Їх перевагою є здатність використовувати емпіричні дані про те, які модулі часто змінюються, де вже виникали помилки, який обсяг коду

переписувався тощо. Прикладом є показник обсягу змін – сумарного обсягу доданого чи видаленого коду між релізами. Дослідження Microsoft продемонструвало, що відносні метрики змін є відмінними предикторами щільності дефектів у великій промисловій системі: збільшення показників змін супроводжується зростанням дефектності [13]. Іншими словами, модулі, де відбувалися значні часті зміни, мають більшу ймовірність містити помилки. Недоліки процесних метрик у тому, що для їх отримання потрібні якісні дані з репозиторіїв та баг-трекерів. Необхідно, щоб команда дисципліновано фіксувала дефекти, наприклад, позначала коміти як виправлення багів і зберігала історію змін. У нових проєктах або новостворених модулях може банально не бути достатньої історії, щоби зробити висновки. Також процесні метрики можуть бути специфічними для контексту, наприклад, великий обсяг змін в одному проєкті означає проблему, а в іншому може бути нормальною практикою, наприклад, рефакторинг на початку розвитку. Отже, для коректного застосування історичних даних потрібен аналіз контексту і, бажано, залучення експертних оцінок.

Підсумовуючи, потрібно визначити, що методологічні засади оцінювання якості ПЗ еволюціонували від загальних моделей якості до конкретних систем метрик. Загальний підхід базується на виділенні важливих для користувача та стейкхолдерів характеристик таких як надійність, продуктивність, супроводжуваність тощо)та подальшому кількісному вимірі цих характеристик через відповідні метрики. Подальший розвиток теми пов’язаний з використанням історичних даних проекту для вдосконалення оцінки якості, зокрема – для прогнозування дефектів, що розглядається в наступному підрозділі.

1.2 Аналіз існуючих рішень

Одним із важливих аспектів якості ПЗ є його надійність та відсутність дефектів. У великих проєктах виправлення помилок може споживати до 80%

загального бюджету, тому здатність наперед виявити дефектонебезпечні частини системи надзвичайно важлива. Прогнозування дефектів (Software Defect Prediction, SDP) – це процес побудови моделей, які на основі наявних даних (метрик, історії, тестів) визначають, які частини коду найбільш схильні до помилок у майбутньому. По суті, такий підхід аналізує історію проекту, щоб виявити ділянки вихідного коду, які можна вважати більш схильними до дефектів. Це дає змогу проактивно спрямувати зусилля з тестування і код-рев'ю на найбільш ризиковані компоненти, підвищуючи загальну якість.

У науковій літературі прогнозування дефектів зазвичай формулюється як задача машинного навчання – класифікації або регресії [16, 17]. Будується статистична модель, що отримує на вході метрики програмних модулів (незалежні змінні), а на виході передбачає наявність або кількість дефектів (залежна змінна). Для побудови таких моделей потрібні тренувальні дані, наприклад, статистика метрик для модулів попередньої версії та фактична кількість знайдених у них багів. На основі цього модель «навчається» знаходити закономірності. Застосовані алгоритми можуть бути різними, тобто використовуються методи класифікації (дерева рішень, байєсівські класифікатори, Random Forest, метод опорних векторів тощо) та регресії (лінійна регресія, логістична регресія – для оцінки ймовірності дефекту). За даними оглядів, найпоширенішими є підходи з моделюванням задачі як бінарної класифікації, де модуль віднесено або до дефектних, або до чистих та застосуванням популярних алгоритмів машинного навчання. Так, в одному з новітніх досліджень з порівняння алгоритмів для дефектопрогнозування було показано, що найбільш високі результати точності дають алгоритми дерев рішень (Decision Tree) та ансамблеві методи на їх основі, зокрема Random Forest.

Побудова моделі прогнозування дефектів зазвичай включає такі кроки:

– збір даних – формування набору прикладів, що включає значення метрик для модулів (файлів, класів, компонент) та мітку про наявність дефекту;

– навчання моделі на цих даних;

– перевірка моделі на тестовому підмножині даних та оцінка якості прогнозу за метриками точності;

– застосування моделі для передбачення дефектних компонент у нових версіях. Існують різні рівні гранулярності прогнозування: від прогнозу дефектності файлу або модуля (класифікація файлів як багатих на дефекти чи ні) до Just-In-Time (JIT) прогнозування на рівні комітів (визначення, чи містить конкретний коміт потенційний дефект).

Останній підхід JIT DP набув популярності останніми роками і він дозволяє оцінювати ризик кожної зміни коду одразу в момент коміту, що особливо корисно для інтеграції з процесом перевірки коду. JIT-моделі використовують метрики змін (кількість доданих чи видалених рядків, змінені файли, досвід автора коміту тощо) для класифікації коміту як ризикованого або безпечного. Це дає QA-команді можливість оперативно реагувати: більш уважно перевіряти підозрілі коміти, розподіляти ресурси тестування пріоритетно на них тощо.

Як зазначено вище, дефектні моделі беруть інформацію із даних метрик, тому вибір правильних метрик є критично важливий момент. Сучасні дослідження використовують широкий набір показників, що охоплюють структурні характеристики коду та процесні характеристики. У численних роботах основою слугує набір метрик Чидембера-Кемерера – шість об’єктно-орієнтованих метрик, що вимірюють розмір, складність, зв’язаність і зчеплення класів (зокрема WMC – зважена кількість методів, DIT – глибина успадкування, NOC – кількість нащадків, CBO – зв’язність між класами, LCOM – відсутність зв’язності, RFC – кількість викликів методів). Метрики CK давно використовуються і було показано їх статистичну кореляцію з дефектністю: наприклад, модулі з більшою довжиною коду (LOC) чи більшою кількістю точок обходу (WMC) схильніші до багів. Проте, як відзначається в оглядах, досі немає єдиної думки, які саме метрики є найкращими предикторами дефектів – різні дослідження називали різні метрики найбільш впливовими. Наприклад, в деяких роботах серед найсильніших індикаторів дефектності в Java-проектах були виявлені такі метрики, як NOC

(кількість підкласів), NPA (кількість публічних атрибутів), DIT, LCOM5 (зв’язаність методів класу). Натомість метрики типу CBO (зв’язність між класами) суттєвого покращення моделей не давали. Такий розкид результатів пояснюється контекстом проектів і вибором алгоритмів моделювання. В цілому, стандартні метрики коду забезпечують базову інформативність, але можуть потребувати доповнення.

Надзвичайно цінним доповненням стали метрики процесу (історії змін). Дослідники показали, що залучення показників змін коду та попередніх дефектів суттєво підвищує точність прогнозів. Прикладом процесних метрик є:

Code Churn (кількість змінених рядків коду між випусками),

– кількість змін (ревізій) файлу, кількість розробників, які редагували файл,

– вік файлу (час від першого створення чи останньої зміни),

– кількість історичних баг-фіксів (скільки дефектів було виправлено у цьому модулі раніше) тощо.

Такі метрики відображають так звану турбулентність модуля, тобто якщо код переписувався багаторазово, якщо над ним працювало багато різних людей або постійно знаходились помилки, то це може бути сигналами ризику. Зокрема, сформульовано набір метрик для JIT-прогнозування дефектів, куди ввійшли: число доданих чи видалених LOC у коміті, число файлів у коміті, частка зміненого коду, досвід автора (кількість його попередніх комітів), час що пройшов від попередньої зміни файлу тощо. У поєднанні з традиційними метриками коду, такі процесні характеристики дали змогу побудувати моделі, що успішно передбачають дефектні коміти з доволі високою точністю (за деякими оцінками, понад 70-80% правильних спрацьовувань).

Іншим напрямом збагачення ознак стали метрики, пов’язані з якістю коду, такі як кількість code smells або антипатернів. Наприклад, Паломба та інші запропонували враховувати наявність недоліків дизайну у класах (довгі методи, циклічні залежності тощо) як ознаки, що вказують на потенційно дефектний код.

Це також можна розглядати як різновид внутрішніх метрик, що більше фокусуються на структурних недоліках.

Після визначення ознак (метрик), постає питання вибору алгоритму моделювання. Класичні алгоритми машинного навчання, як згадано, продемонстрували ефективність на задачах дефектопрогнозування. Зокрема, дерева рішень цінуються за інтерпретованість, тобто вони можуть явно показати, які порогові значення метрик відрізняють дефектні модулі. Ансамблеві методи на їх основі зазвичай дають кращу точність за рахунок усереднення результатів багатьох дерев. У реальних датасетах дефектів часто спостерігається дисбаланс класів (дефектних модулів значно менше, ніж чистих), тому використовуються підходи боротьби з дисбалансом – наприклад, вартісні функції помилки, спеціальні метрики оцінки (AUC, MCC), недопущення перенавчання на чистий клас тощо.

Окремі роботи аналізують власність на код і змінність, а саме оновлено дослідження на власність коду, а також моделі для виявлення схильних до змін методів. Наявні відкриті набори даних дефектопрогнозування, зокрема еталон. Перші результати застосування глибинних моделей до задач SDP демонструють обіцяну точність.

Останнім часом набувають популярності глибинні нейронні мережі для аналізу коду. Ідея полягає в автоматичному вилученні ознак із сирцевого коду (на відміну від ручного конструювання метрик). Такі підходи включають подання коду у вигляді послідовностей токенів, абстрактних синтаксичних дерев (AST) чи графів потоку виконання, і використання нейромереж (RNN, LSTM, CNN, Transformer) для навчання представлення коду. Деякі роботи комбінують глибинні ознаки з традиційними метриками, де запропоновано модель на основі CNN, куди на вході подавались векторизовані AST-вузли, а на виході нейронна мережа генерувала ознаки, які потім об’єднувались із класичними метриками коду для фінального прогнозування через логістичну регресію. Інша робота використовувала дві модальніності – текст коміту і диференційний патч коду – які оброблялися CNN і об’єднувалися для прогнозу дефектного коміту. Такі підходи показують перспективу підвищення якості прогнозу за рахунок врахування семантики змін.

Зокрема, моделі на основі Transformer, наприклад, CodeBERT навчаються на великій вибірці коду і здатні враховувати більш складні патерни, ніж лінійні метрики. Втім, глибинне навчання потребує великих обсягів даних і складніше піддається інтерпретації, тому в контексті нашого дослідження основна увага приділяється класичним метрикам та алгоритмам, які є більш прозорими і легкими у впровадженні.

Розглядаючи існуючі методи прогнозування дефектів, важливо відзначити їх сильні та слабкі сторони у контексті поставлених завдань. Перевага моделей, що базуються на статичних метриках коду, – незалежність від домену: метрики якості (розмір, складність, зчеплення) застосовні до будь-якого проекту, їх легко обчислити і перевірити. Вони дозволяють виявити потенційно проблемні місця ще до того, як там проявляться дефекти (превентивна діагностика). Проте недолік в тому, що точність прогнозу лише за статичними метриками обмежена – різні модулі можуть мати схожі значення метрик, але дуже різну історію надійності. Як було показано, самі по собі метрики коду часто недостатні для високої точності класифікації. Додавання історичних метрик суттєво покращує картину: моделі, що враховують і характеристики коду, і дані про зміни чи дефекти, продемонстрували значно вищу здатність виявляти дефектні модулі. Наприклад, у експериментальному дослідженні було підтверджено гіпотезу, що історія змін між випусками (code churn) у поєднанні з традиційними метриками (щільністю багів, складністю коду тощо) є добрим предиктором дефектності перед релізом; більш того, додавання метрик змін підвищує точність прогнозу порівняно з моделями, що їх не містять.

З іншого боку, використання історичних даних має вразливі місця: по-перше, історичні метрики роблять модель залежною від якості даних репозиторію (якщо дефекти або зміни фіксувались неповно, модель може навчитися невірно). По-друге, такі моделі можуть гірше переноситися на нові проекти (де історія інша) – проблема генералізації. Насправді, у сфері прогнозування дефектів існує відома проблема, коли моделі добре працюють внутрішньо-проектно, але часто втрачають точність при застосуванні до іншого проекту через різний розподіл метрик. Це

вимагає або накопичення достатньої кількості даних для кожного конкретного проекту, або розробки методів нормалізації чи переносу знань між проектами.

Інженери з більшою готовністю сприймають прості моделі, на зразок правил типу *якщо модуль змінився >10 разів і має >500 LOC, він ризикований*, ніж так звані чорні скриньки. Тому з точки зору практичного впровадження, прості моделі на основі метрик можуть бути краще прийняті командою, але вони повинні демонструвати прийнятну якість. Задача даного дослідження якраз і полягає в тому, щоб підвищити якість і корисність метрик, разом з тим зберігаючи їх простоту, підсилити їх інформативність за рахунок залучення історичних характеристик.

Отже, сучасний стан методів оцінки якості ПЗ і прогнозування дефектів такий: є великий арсенал метрик, є усталені алгоритми побудови моделей, підтверджена корисність процесних метрик для точності прогнозів. Проте відсутній єдиний консенсус, які метрики найкращі; багато інструментів і даних з попередніх досліджень застаріли або важкодоступні, що ускладнює відтворення результатів. Це підкреслює необхідність подальших досліджень і вдосконалення підходів, зокрема, в частині розробки нових метрик, що комбінують інформацію про код та його еволюцію, аби підвищити точність і корисність оцінки якості.

1.3 Огляд методів вирішення проблеми

Аналіз методів удосконалення метрик якості програмного забезпечення через призму історії змін та дефектів у системах контролю версій свідчить про перехід від статичного аналізу коду до еволюційного, відомого як Mining Software Repositories (MSR). Цей підхід дозволяє виявити приховані ризики, які неможливо

побачити, аналізуючи лише поточний стан коду, і базується на інтеграції даних із репозиторіїв (наприклад, Git) та систем відстеження помилок (наприклад, Jira).

Фундаментальним методом у цьому напрямку є аналіз турбулентності коду (Code Churn). Традиційні метрики, такі як кількість рядків коду (LOC) або цикломатична складність, показують лише статичний розмір проблеми. Врахування історії дозволяє розрахувати відносний churn - частоту та обсяг змін у модулі за одиницю часу. Методи, що базуються на цьому показнику, виходять із припущення, що файли з високою частотою редагування та великою кількістю доданих або видалених рядків мають значно вищу ймовірність містити дефекти. Це дозволяє трансформувати статичну метрику складності в динамічну метрику ризику: складний модуль, який не змінювався роками, є менш ризикованим, ніж простіший модуль, що змінюється щодня.

Наступним рівнем удосконалення є виявлення логічної зв’язності через аналіз спільних змін. Класичні метрики зв’язності базуються на синтаксичному аналізі імпортів та викликів функцій. Натомість аналіз історії VCS дозволяє виявити неявні залежності: якщо два файли змінюються в одних і тих самих комітах протягом тривалого часу, вони логічно пов'язані, навіть якщо між ними немає прямих посилань у коді. Це дозволяє вдосконалити архітектурні метрики, вказуючи на порушення принципу єдиної відповідальності та необхідність рефакторингу, які статичний аналізатор пропустить.

Критично важливим методом є використання алгоритму SZZ (Sliwerski, Zimmermann, Zeller) та його модифікацій для побудови моделей передбачення дефектів. Цей метод дозволяє зв’язати запис про виправлення помилки в баг-трекері з конкретним комітом, що її виправив, а потім, аналізуючи історію («git blame»), знайти коміт, який цю помилку вніс. Це створює розмічену базу даних, на основі якої розраховуються метрики схильності до помилок (Defect Proneness) для конкретних модулів або навіть розробників, дозволяючи прогнозувати появу багів ще до етапу тестування на основі характеристик нових змін.

Також історія змін дозволяє впровадити соціальні метрики якості або метрики власності коду (Code Ownership). Дослідження показують сильну

кореляцію між якістю ПЗ та кількістю розробників, що змінюють один файл. Методи, що враховують ці дані, оцінюють рівень «розмивання» відповідальності: файли, які редагує велика кількість авторів з незначним внеском кожного (minor contributors), мають статистично вищу щільність дефектів. Врахування фактору «авторства» та досвіду конкретного розробника в історії змін конкретного модуля дозволяє калібрувати метрики надійності, роблячи їх більш персоналізованими та точними.

Нарешті, метод аналізу віку коду та його «загасання» дозволяє вдосконалити планування регресійного тестування. Метрики, що враховують час, який пройшов з останньої зміни, допомагають пріоритезувати перевірки: нещодавно змінені ділянки коду (або старий код, який раптово почали змінювати) отримують вищі коефіцієнти ризику в інтегральних оцінках якості, ніж стабільні архівні модулі. Це перетворює метрики якості з інструменту констатації факту на інструмент проактивного управління ризиками проєкту.

1.4 Постановка задачі

Проведений огляд показав, що якість програмного забезпечення значною мірою залежить від його коду та еволюції цього коду в процесі розробки. Традиційні метрики якості (розмір, складність, зв’язність тощо) надають цінні показники, але не враховують динаміку змін, яка часто несе критичну інформацію про дефектність. Історія комітів і дефектів у системах контролю версій містить ознаки слабких місць – файлів, що постійно викликають проблеми, фрагментів коду, що багаторазово перероблялися, тощо. Інтеграція цих факторів з метриками коду є перспективним шляхом підвищення точності прогнозування дефектів та виявлення вразливих компонентів системи. Актуальність теми зумовлена потребою у більш надійних методах забезпечення якості: програмні проекти зростають за розміром і складністю, і проактивний підхід до управління якістю

(через прогноз потенційних дефектів) може дати значну економію ресурсів і підвищити надійність кінцевого продукту. Крім того, аналіз метрик на основі історичних даних відповідає сучасним трендам Mining Software Repositories та Data-Driven DevOps, що підтверджує актуальність наукового дослідження в цьому напрямі.

Мета дослідження - удосконалення методики оцінювання якості програмного забезпечення шляхом розроблення інтегрованих метрик, що враховують історію змін коду та дані про дефекти, для підвищення точності прогнозування дефектів і виявлення вразливих компонентів системи.

Об’єктом дослідження є процес зміни програмного забезпечення в ході розробки та супроводу із властивими йому характеристиками якості та дефектами.

Предметом дослідження є метрики якості програмного забезпечення, що базуються на історії змін коду, характеристиках комітів, активності розробників і даних про дефекти.

Для досягнення зазначеної мети необхідно вирішити такі завдання:

1. Аналіз літератури та існуючих підходів. Проаналізувати сучасні методи оцінки якості ПЗ і прогнозування дефектів (моделі якості, метрики, алгоритми), а також практичний досвід їх застосування. Виявити сильні та слабкі сторони цих підходів в частині врахування історії змін, обґрунтувати напрям удосконалення метрик. Це завдання виконано в розділі 1 шляхом огляду джерел за трьома напрямами: методологія оцінки якості, конкретні метрики та алгоритми прогнозування дефектів, практика застосування.

2. Розробка нового підходу до метрик. Запропонувати концепцію нових або вдосконалених метрик якості, що інтегрують інформацію з системи контролю версій (історію комітів) та дані баг-трекінгу. Сформулювати визначення таких метрик, обґрунтувати їх вибір, наприклад, метрики нестабільності коду, дефектної історії тощо з погляду впливу на якість.

3. Моделювання та алгоритми. Розробити модель або методику використання запропонованих метрик для прогнозування дефектів. Це може бути побудова математичної моделі (наприклад, класифікатора) або алгоритму аналізу ризику на

основі значень метрик. Визначити, які машинні навчання або евристичні методи будуть застосовані, як будуть інтегровані історичні дані.

4. Програмна реалізація. Реалізувати програмний інструмент або прототип, що обчислює запропоновані метрики з репозиторію, наприклад, скрипт для Git і застосовує модель прогнозування дефектів. Забезпечити збір необхідних даних, вибрати проєкт або набір даних для експерименту та інтегрувати рішення в середовище аналізу.

5. Експериментальна перевірка. Провести експериментальні дослідження на обраному проєкті(ах) для оцінки ефективності запропонованих метрик і моделі. Порівняти точність прогнозування дефектів з використанням нових метрик проти базових підходів, наприклад, тільки статичних метрик. Проаналізувати результати, перевірити статистичну значущість покращення, оцінити практичну застосовність. Виявити можливі обмеження чи випадки, коли метод не спрацьовує, та запропонувати шляхи подальшого удосконалення.

Вирішення зазначених завдань дозволить досягти мети дослідження. Очікується, що в результаті буде запропоновано новий або вдосконалений підхід до метрик якості ПЗ, який враховує історичну інформацію про зміни коду і дефекти.

1.5 Висновки до 1-го розділу

Таким чином, у підсумку проведеного огляду обґрунтовано актуальність проблеми удосконалення метрик якості ПЗ з врахуванням історії змін та дефектів. Сформовано чітке розуміння об’єкта, предмета і мети дослідження, а також окреслено конкретні завдання, розв’язання яких забезпечить досягнення мети.

В першому розділі здійснено теоретичний огляд проблематики оцінювання якості програмного забезпечення та прогнозування дефектів. Проаналізовано методологічні підходи, що включають моделі якості та систему метрик для

вимірювання внутрішніх характеристик коду. Розглянуто сучасні методи і алгоритми прогнозування дефектів, які поєднують статичні метрики та історичні дані, а також наведено приклади їх успішного застосування в реальних проєктах. Проведено критичне порівняння підходів: встановлено, що статичні метрики забезпечують базову оцінку якості, але їх поєднання з метриками історії (кількістю та обсягом змін, історією дефектів) значно підвищує здатність виявляти дефектонебезпечні компоненти. Разом з тим, відзначено проблеми практичного впровадження таких моделей в індустрії, тобто необхідність даних, генеричність, довіра.

Огляд джерел підтвердив актуальність і доцільність подальшого дослідження у напрямі інтеграції історичних даних у метрики якості. У підрозділі 1.4 сформульовано постановку задачі дослідження – визначено об’єкт, предмет, мету та завдання роботи, що створює концептуальну основу для наступних розділів.

2 УДОСКОНАЛЕННЯ МЕТРИК ЯКОСТІ З УРАХУВАННЯМ ІСТОРІЇ ЗМІН ТА ДЕФЕКТІВ КОДУ

2.1 Удосконалення метрик якості з урахуванням історії змін коду та дефектів

У практиці інженерії програмного забезпечення широко застосовуються статичні метрики коду, що вимірюють внутрішні характеристики програмного коду.

Як зазначено раніше, статичні метрики характеризують внутрішні властивості коду, але не враховують історію його розробки, тоді як процесні метрики (історія змін, виправлення багів тощо) можуть істотно доповнити картину якості. В рамках цього дослідження пропонується розширити систему метрик якості, включивши до неї показники, побудовані на основі даних систем контролю версій (VCS) та баг-трекерів. Нижче формально визначено декілька таких метрик.

1. Метрика частоти змін (History of Changes). Нехай A – певний модуль програмного забезпечення (наприклад, файл або клас). Позначимо через $N_{changes}(A)$ кількість комітів в репозиторії, що зачіпали модуль A протягом його життя. Ця метрика відома також як «кількість ревізій» модуля. Інтуїтивно, чим частіше файл змінювався, тим менш стабільним він є: часті зміни можуть свідчити про функціональну складність, постійне додавання нових вимог або виправлення помилок. Дослідження показали, що показники частоти змін є сильними предикторами дефектності – часто змінювані файли виявляються більш дефектонебезпечними. Тому метрика $N_{changes}$ включена як фактор оцінки якості.

Формально можна визначити цю метрику як суму одиниць за всіма комітами, що містять зміни файлу A (1):

$$M_{chg}(A) = N_{changes}(A) = \sum_{commit\ c} I(c, A) \quad (1)$$

де $I(c, A) = 1$, якщо коміт c модифікував файл A, і 0 – інакше. В залежності від мети аналізу, даний показник можна обмежувати певним інтервалом часу (наприклад, кількість змін за останній рік) або враховувати всю історію проекту.

Замість просто кількості комітів можна врахувати обсяг змін – так званий code churn [5]. Code Churn вимірює кількість рядків коду, доданих, видалених або змінених у модулі за певний період. Наприклад, для кожного коміту c, що змінює файл A, можна взяти кількість доданих ($\Delta^{+}_{c,A}$) та видалених ($\Delta^{-}_{c,A}$) рядків і визначити сумарний churn, як це подано в (2):

$$M_{churn}(A) = \sum_{commit\ c}(\Delta^{+}_{c,A} + \Delta^{-}_{c,A}) \quad (2)$$

Churn докладніше відображає обсяг правок: 1 коміт може змінити 2 рядки, а інший – 200 рядків, і проста метрика $N_{changes}$ не розрізняє цих ситуацій. У нашій методології для простоти основним показником історії буде $N_{changes}(A)$, але при наявності детальних даних можна використовувати і $M_{churn}(A)$ – ідеї нормалізації та агрегації для них аналогічні.

2. Метрика дефектів (Bug History). Нехай $N_{defects}(A)$ – кількість дефектів, пов’язаних з модулем A в минулому. Цей показник можна визначити як число підтверджених багів, виправлених у цьому модулі, або кількість комітів типу "bug fix", що стосувалися A. Для його обчислення необхідно зв’язати інформацію з системи відстеження проблем (наприклад, Jira, GitHub Issues) чи аналізувати повідомлення комітів на наявність ключових слів (типу “fix”, “bug”) або ID дефектів. У дослідженнях було показано, що включення інформації про кількість попередніх помилок у модулі значно підвищує точність прогнозування майбутніх дефектів. Інтуїтивно зрозуміло, що модуль, де вже знаходили баги, має вищий ризик містити їх знову. Метрика $N_{defects}(A)$ фактично відображає історичну надійність компонента.

Формально, якщо є множина зареєстрованих дефектів D, і для кожного дефекту можемо визначити, які файли було змінено при його виправленні, то (3):

$$M_{def}(A) = N_{def}(A) = \sum_{d \in D} I(d, A) \quad (3)$$

де $I(d, A) = 1$, якщо дефект d (або коміт, що його виправляє) зачіпав файл A. Знову ж, цю метрику можна рахувати за весь час або в певному вікні (наприклад, кількість дефектів за останні 6 місяців), залежно від того, чи хочемо ми отримати «спадкову» схильність до дефектів або актуальний рівень якості.

3. Метрика внутрішньої складності. Окрім історичних даних, доцільно залишити хоча б одну метрику, що характеризує поточний стан коду модуля. Адже модуль може не мати багатої історії змін, але бути дуже складним структурно, що само по собі є ризиком. В якості такої метрики оберемо цикломатичну складність модуля або пов'язані показники. Для об'єктно-орієнтованого класу можна використати WMC (сумарна цикломатична складність методів), визначений раніше, або просто максимальну цикломатичну складність серед методів. Позначимо через $CYC(A)$ показник складності модуля A. Вищі значення CYC свідчать про більш заплутану логіку, більшу кількість незалежних шляхів виконання, що ускладнює тестування і підвищує ймовірність помилки. Цю метрику можна отримати статичним аналізом останньої версії коду модулю.

Таким чином, набір запропонованих метрик для модуля A включає принаймні три величини:

$M_{chg}(A) = N_{changes}(A)$ – кількість змін (комітів) модуля,

$M_{def}(A) = N_{defects}(A)$ – кількість дефектів, знайдених у модулі раніше,

$M_{comp}(A) = CYC(A)$ – міра структурної складності коду модуля.

За потреби цей набір може бути розширений іншими показниками процесу (наприклад, кількість авторів – скількома різними розробниками змінювався код, час від останньої зміни – «віковий» фактор, тощо). Проте для побудови інтегральної оцінки якості достатньо і базових трьох: вони відображають надійність (історія дефектів), стабільність (інтенсивність змін) і складність (внутрішня якість коду) модуля.

Наступним кроком є приведення цих метрик до спільної шкали та об'єднання їх у єдиний показник якості модуля.

Сирі значення $N_{changes}$, $N_{defects}$, CYC можуть мати різні діапазони. Наприклад, $N_{changes}$ для різних файлів може коливатись від 1 до 100 (комітів), $N_{defects}$ – від 0 до, скажімо, 20, а CYC – від 1 до 50. Щоб об'єднати їх, потрібно виконати нормалізацію – привести до порівнянної шкали, наприклад або . Нехай для метрики M визначено її мінімальне та максимальне значення серед всіх модулів (або очікувані граничні значення). Тоді нормалізоване значення метрики для модуляA можна обчислити за формулою мін-макс (4):

$$\widetilde{M}(A) = \frac{M(A) - M_{min}}{M_{max} - M_{min}} \quad (4)$$

де $M_{min} = \min M(X)$, $M_{max} = \max M(X)$ по всіх модулях X у розглядуваній системі або вибірці. В результаті $\widetilde{M}(A)$ буде від 0 (якщо $M(A)$ дорівнює мінімуму по вибірці) до 1 (якщо дорівнює максимуму). Така лінійна нормалізація зручна і зрозуміла. Її недолік – чутливість до вибору M_{min} і M_{max} (які можуть бути викидами). В практиці часто використовують порогові значення або перцентили замість глобального мін/макс, але в даному теоретичному описі приймемо ідеальний випадок, коли відомі розумні межі метрик.

Нехай отримано нормалізовані значення: $\widetilde{M}_{chg}(A)$, $\widetilde{M}_{def}(A)$, $\widetilde{M}_{comp}(A)$ для модуля A. Вони лежать в інтервалі [0,1], де більше значення відповідає «гіршому» показнику (бо, приміром, більше змін або багів – гірше). Для переходу до позитивної шкали якості визначимо обернені метрики якості:

$Q_{chg}(A) = 1 - \widetilde{M}_{chg}(A)$ - відносна стабільність модуля.

$Q_{def}(A) = 1 - \widetilde{M}_{def}(A)$ - відносна надійність модуля за дефектами

$Q_{comp}(A) = 1 - \widetilde{M}_{comp}(A)$ - відносна простота/зрозумілість коду.

Тепер всі Q-метрики мають такий сенс: чим більше, тим краще (аналогічно до Maintainability Index, де більше значення – більш підтримуваний код). Значення 1 означало б ідеальний модуль (мінімум змін, дефектів, складності порівняно з іншими), 0 – найгірший у кожній з категорій.

2.2 Інтегральний показник якості програмного забезпечення (IDP)

Маючи нормалізовані показники якості Q_{chg}, Q_{def}, Q_{comp}, можемо об'єднати їх в інтегральну метрику. Назвемо її IDP (Integrated Defect Proneness / Integral Quality Index) – інтегральний показник якості, що враховує і статичні, і історичні фактори. Наша мета – отримати єдине число, подібно до Maintainability Index, але побудоване на більш широкому наборі даних.

Найпростіший спосіб об'єднання – це зважена сума компонентів якості. Введемо вагові коефіцієнти w_{chg}, w_{def}, w_{comp} для відповідних складових, при цьому приймемо умову нормування (5):

$$w_{chg} + w_{def} + w_{comp} = 1 \tag{5}$$

а самі w_i – невід'ємні. Тоді інтегральний показник для модуля A обчислюється як у (6):

$$IDP(A) = w_{chg} \times Q_{chg}(A) + w_{def} \times Q_{def}(A) + w_{comp} \times Q_{comp}(A) \tag{6}$$

Оскільки кожен Q лежить між 0 і 1, $IDP(A)$ теж знаходитиметься в інтервалі (або %, якщо зручніше працювати у відсотковій шкалі). Високі значення IDP означають високу сумарну якість: модуль стабільний, надійний і простий; низькі – свідчать про проблемний модуль, що часто змінювався, багатий на дефекти або надмірно складний.

Смислове трактування IDP: можна сказати, що він відображає інтегральну «здоров'я» модуля. Якщо Maintainability Index переважно фокусується на підтримуваності (що близько до поняття простоти коду), то IDP охоплює ще й аспекти, пов'язані з дефектністю та стабільністю коду в часі. Це наближує його до показника надійності. Власне, абревіатуру IDP можна інтерпретувати як Index of Defect

Proneness (індекс дефектонебезпечності) – хоча ми обрали формулу так, що математично більший IDP відповідає нижчій дефектонебезпечності, тобто кращій якості.

Важливим етапом є визначення ваг w_{chg}, w_{def}, w_{comp}, оскільки від них залежить внесок кожної складової у фінальний бал. Існує кілька підходів до призначення ваг:

За відсутності пріоритетів ваги можна взяти рівними: $w_{chg} = w_{def} = w_{comp} = 1/3$. У цьому випадку IDP є середнім трьох компонент якості.

Ваги можуть бути обрані експертним шляхом, залежно від пріоритетів проекту. Наприклад, у системах реального часу критичнішою є надійність, тому можна надати більшої ваги історії дефектів: наприклад, $w_{def} = 0.5$, а на інші два по 0.25. Якщо ж проекту важливіша простота супроводу, можна збільшити w_{comp}.

Ваги можна обчислити на основі даних, використовуючи методи на кшталт регресійного аналізу або оптимізації. Зокрема, якщо є дані про фактичну дефектність або якість модулів, можна підібрати w_i так, щоб IDP найкраще корелював з цими зовнішніми показниками. Фактично, формула є лінійною моделлю, і коефіцієнти w можна навчити, мінімізувавши помилку прогнозу дефектів (подібно до моделі лінійної регресії або логістичної регресії, де ваги відображають вклад ознаки). Такий підхід забезпечив би оптимальні з точки зору даних ваги, але зменшує інтерпретованість (коефіцієнти можуть виявитись неінтуїтивними) і потребує наявності достатнього масиву даних з мітками якості.

Ще один підхід – призначити ваги пропорційно варіативності або інформаційній значущості метрик. Наприклад, якщо в наборі модулів змінюваність $N_{changes}$ сильно варіюється, а складність CYC майже однакова для всіх, то доцільно дати більшої ваги w_{chg}, оскільки саме вона диференціює модулі. Інформаційно-теоретичні міри чи аналіз головних компонент також можуть підказати раціональний вибір ваг.

У даному дослідженні спочатку розглядатимуться рівні ваги для простоти, але з можливістю коригування. Вибір рівних ваг підкреслює, що ми намагаємось

врахувати всі три аспекти якості рівноцінно. Проте, як буде показано в аналізі, історичні метрики змін і дефектів часто мають сильніший вплив на дефектність, ніж статичні метрики складності. Це може стати аргументом для більшого значення w_{def} і w_{chg} порівняно з w_{comp}. Остаточне рішення про ваги може прийматися з урахуванням специфіки доменної області та цілей оцінювання (напр., для прогнозу дефектів або для оцінки підтримуваності).

Хоча формула лінійна, за потреби її можна розширити до нелінійної інтеграції. Наприклад, замість простої суми можна використати мультиплікативну форму (перемноження факторів якості), яка інколи застосовується для індексів, щоб відобразити спільний ефект. Але множення зробить індекс менш інтерпретованим (складно виділити внесок кожного фактора). Інший шлях – застосувати порогові функції: наприклад, якщо Q_{def} дуже низький (модуль мав багато дефектів), знижувати IDP непропорційно сильно. Подібні нелінійності можуть краще моделювати реальний вплив (де наявність багатьох дефектів переважує всі інші переваги). Однак, щоб зберегти метрику зрозумілою, ми притримуватимемось лінійної моделі з фіксованими вагами.

Отже, інтегральний показник IDP надає кожному модулю кількісну оцінку якості, що агрегує три ключові виміри: стабільність змін, надійність (відсутність дефектів) і простоту коду. Далі розглянемо конкретний приклад, як обчислюються запропоновані метрики та IDP для умовного модуля.

2.3 Алгоритм обчислення метрик та показника IDP

Розглянемо умовний модуль – файл ModuleX.java в межах деякого програмного проекту. Припустимо, що з системи контролю версій та бази дефектів зібрано такі дані про цей файл:

Кількість комітів, що змінювали ModuleX: $N_{changes}(ModuleX) = 10$, тобто файл змінювався 10 разів протягом життя проекту;

Кількість виправлених дефектів у ModuleX: $N_{defects}(ModuleX) = 2$, було два підтверджених баги, у яких файл фігурував;

Цикломатична складність ModuleX: $CYC(ModuleX) = 20.$

Тепер припустимо, що для нормалізації відомий діапазон значень кожної метрики серед усіх файлів проекту (або вирішено їх зафіксувати експертно):

Максимальна кількість змін будь-якого файлу в проекті – 20, мінімальна – 1 (є файли, які майже не змінювали);

Максимум дефектів на файл – 5, мінімум – 0 (деякі файли ще не мали жодної помилки);

Максимальна цикломатична складність – 30, мінімальна – 1 (припустимо, що найпростіші модулі мають складність 1);

Крок 1: Нормалізація сирих метрик для ModuleX:

$\widetilde{M}_{chg}(ModuleX) = \frac{10-1}{20-1} = \frac{9}{19} \approx 0.474$ – частота змін;

$\widetilde{M}_{def}(ModuleX) = \frac{2-0}{5-0} = \frac{2}{5} = 0.4$ – історична дефектність;

$\widetilde{M}_{comp}(ModuleX) = \frac{20-1}{30-1} = \frac{19}{29} \approx 0.655$ – складність коду.

Крок 2: Обчислення позитивних показників якості (інвертованих):

$Q_{chg}(ModuleX) = 1 - 0.474 = 0.526$ – стабільність – помірний результат: файл не найстабільніший, але і не в топі найбільш змінюваних.;

$Q_{def}(ModuleX) = 1 - 0.4 = 0.6$ – надійність – це означає, що за дефектністю він ближче до «хороших» (бездефектних) файлів, ніж до рекордсменів за багами. Два баги – це певний сигнал неблагополуччя, але не критично (деякі файли мали до 5);

$Q_{comp}(ModuleX) = 1 - 0.655 = 0.345$ – простота та читабельність коду – це низький показник, тобто файл є серед найскладніших: його цикломатична складність близька до максимуму по проекту. Отже, з точки зору підтримуваності код ModuleX викликає занепокоєння.;

Припустимо для прикладу, що всі ваги рівні: $w_{chg} = w_{def} = w_{comp} \approx 0.333$. Тоді:

$$IDP(ModuleX) = 0.333 \times 0.526 + 0.333 \times 0.6 + 0.333 \times 0.345 \approx 0.49.$$

Отриманий показник розташований у середині шкали. ModuleX має:

Середню стабільність (0.526);

Помірну дефектність (0.6);

Високу складність (0.345 – слабке місце).

Отже головний ризик тут – складність коду. Модулю може знадобиться рефакторинг, щоб знизити цей показник хоча б до середнього результату.

Для порівняння, уявімо ще два гіпотетичні модулі:

ModuleY: $N_{changes} = 1$ (майже не змінювався), $N_{def} = 0$ (помилок чи багів не було виявлено), $CYC = 5$ (дуже простий код). Для нього $IDP(ModuleY) \approx 0.954$, що є майже відмінною оцінкою якості.

ModuleZ: $N_{changes} = 20$ (дуже багато правок), $N_{def} = 0$ (було виявлено багато багів), $CYC = 30$ (максимально складний код). Для нього $IDP(ModuleZ) = 0$, що показує дуже погану якість цього модуля.

Таким чином, IDP розподіляє модулі по шкалі від ~0 (дуже низька якість, «червона зона») до ~1 (висока якість, «зелена зона»), враховуючи кількісно і накопичену історію дефектів, і динаміку змін, і складність коду. Це дозволяє ранжувати компоненти за пріоритетом уваги: наприклад, ModuleZ із близьким до 0 балом явно потребує негайного аналізу та поліпшення, тоді як ModuleY з 0.95 може вважатися еталонним.

Варто зазначити, що у реальному застосуванні масштабування IDP можна підлаштувати під зручний інтервал чи інтерпретацію, подібно до Maintainability Index. Наприклад, можна помножити значення на 100 і округлити для цілих % або ввести порогові рівні якості (наприклад, >80 – високоякісні модулі, 50–80 – середні, <50 – проблемні). Такі пороги можуть бути встановлені на основі статистичного розподілу IDP для конкретного проекту або відповідно до вимог стандартів якості.

У наступному розділі проведено порівняння запропонованого підходу з традиційними та ML-підходами, а також обговорено переваги й обмеження кожного.

Підсумовуючи розгляд різних підходів до оцінки якості програмного забезпечення, наведемо порівняльний аналіз класичних метрик, інтегральних статичних індексів, моделей машинного навчання та запропонованого підходу з історичними метриками. Аналізуємо за критеріями: прогностична точність, інтерпретованість результатів, вимоги до даних, адаптивність та складність застосування.

Класичні статичні метрики (CK та подібні) прості у зборі – потрібен тільки вихідний код. Багато інструментів (наприклад, аналізатори коду) автоматично обчислюють їх. Інтерпретація метрик безпосередня і зрозуміла: розробники знають, що високий CBO або WMC – поганий знак для модуля. Метрики описують внутрішню структуру, що важливо для підтримуваності. В літературі підтверджено кореляцію деяких з цих метрик з якістю: наприклад, CK-метрики були валідовані як індикатори якості дизайну.

До недоліків можна віднести обмежену точність прогнозування дефектів. Як показали дослідження, статичні метрики не завжди є сильними предикторами фактичної кількості дефектів [3]. Вони відображають потенційну складність чи проблемність, але можуть давати і хибні сигнали (наприклад, великий клас з високим WMC може не мати жодного дефекту, якщо добре пропрацьований). До того ж, статичні метрики стійкі у часі – архітектурні характеристики (як-от успадкування) змінюються рідко, тому модуль, раз потрапивши у категорію ризикових за CK, може довго там лишатися навіть якщо дефекти надалі будуть в інших місцях. Вони не враховують контекст змін: 1000 рядків нового коду і 1000 рядків коду, який протягом років стабілізувався, матимуть однаковий LOC, хоча ризики різні. Отже, адаптивність у класичних метрик низька – без перевимірювання їх після змін картини не побачити, та й зміни коду часто не сильно рухають, скажімо, DIT чи NOC.

Maintainability Index (інтегральний статичний показник), перевагами якого є поєднання кілької важливих аспектів коду (розмір, складність, коментарі) в один бал. Це зручно для відстеження динаміки проекту на високому рівні і для швидкої ідентифікації проблемних модулів (з низьким MI). Індекс має інтерпретацію «добрий/середній/поганий» на основі порогів [11], що спрощує прийняття рішень. Його формула фіксована і загальноприйнята, тому результати порівнянні між різними проектами (в певних межах).

До недоліків можна віднести базування тільки на статичних метриках, отже успадковує їх обмеження – не враховує історію дефектів та змін. Може карати за великий обсяг коду, навіть якщо він не є проблемним (надто велика вага LOC у формулі [13]). Емпіричні параметри формули можуть не відповідати конкретному проекту (наприклад, для скриптових мов чи дуже великих сучасних систем індекс може занижувати або завищувати реальну підтримуваність). Показник MI менш чутливий до локальних аномалій: модуль з одним надскладним методом і багатьма простими може мати прийнятний MI, хоча той один метод – вузьке місце. З точки зору прогнозу дефектів, MI не створювався для цього, хоча і корелює з загальною якістю. Точність прогнозу багів на основі одного MI буде невисокою, оскільки дефекти залежать не лише від складності/розміру, а й від процесу розробки.

ML-моделі (Random Forest, XGBoost, нейронні мережі): Переваги: Найвища точність прогнозування досягається саме ML-підходами, що підтверджують численні експерименти. Вони враховують багато факторів одночасно і можуть виявити приховані взаємозв’язки. Наприклад, Random Forest автоматично комбінує десятки метрик і може вловити нетривіальні правила, які людина не сформулює вручну. Сучасні ансамблеві алгоритми і нейромережі здатні наближатися до 85-90% точності в класифікації дефектних модулів на певних вибірках. Моделі можуть бути перевчені на нові дані, отже мають потенціал адаптації до еволюції проекту. При правильному налаштуванні, ML-модель стає інструментом пріоритезації тестування: вона вказує, куди розробникам варто звернути увагу (файли з найвищою ймовірністю дефектів). Недоліки: Основний мінус – складність і непрозорість. Більшість

високоефективних моделей – це «чорні скриньки», які важко інтерпретувати. Існує відоме протиріччя між точністю і пояснюваністю: найточніші моделі (глибокі нейронні мережі, бустинги) є найменш інтерпретованими. Це створює бар'єр довіри: інженери можуть не приймати рекомендації моделі, не розуміючи причин. Ще один недолік – високі вимоги до даних: потрібно накопичувати історію метрик і дефектів. Якщо проект новий або змінив процес роботи з баг-трекером, даних може бути недостатньо для навчання надійної моделі. Також моделі потребують підтримки: моніторингу якості прогнозів, періодичного оновлення. Їх продуктивність може деградувати, якщо процес розробки змінюється (наприклад, команда почала писати більше автоматичних тестів – кількість дефектів падає, а модель, навчена на старих даних, продовжує їх передбачати за старими патернами). Застосування ML-моделі вимагає фахівця з даних, інтеграції в процес CI/CD, що не завжди виправдано для невеликих команд. Крім того, моделі часто схильні давати хибно позитивні спрацювання на незбалансованих даних (коли дефектів мало, модель може помічати багато файлів як потенційно дефектних, що знижує довіру з часом).

Запропонований підхід (історично-статичні метрики + IDP) поєднує сильні сторони двох світів: з одного боку, враховує процесні метрики (історію змін, дефекти), що довели свою ефективність у прогнозуванні якості, з іншого – зберігає простоту й інтерпретованість, властиві статичним метрикам та індексам. IDP як лінійна комбінація зрозумілий: легко пояснити, чому певний модуль отримав низький бал (наприклад, багато багів і змін – видно по складових). На відміну від чисто статичних індексів на кшталт MI, тут включено інформацію про надійність (баги) і стабільність (зміни), тому точність оцінки дефектонебезпечності вища. Хоча формальна перевірка точності – питання емпіричного розділу, теоретично IDP повинен краще відрізняти проблемні модулі, оскільки враховує ключові фактори ризику з історії. Вагові коефіцієнти дозволяють гнучко налаштувати індекс під конкретні потреби або оптимізувати під наявні дані. Підхід не вимагає складних моделей чи великих вибірок – достатньо витягнути потрібні метрики з репозиторію та баг-трекера, що відносно просто автоматизувати за допомогою

скриптів (існують інструменти типу PyDriller для аналізу історії Git). Крім того, IDP можна перераховувати при кожному новому коміті або релізі, отже він здатен реагувати на зміну ситуації (напр. якщо в модулі раптово з'явилось багато багів, його IDP відразу впаде, сигналізуючи про проблему).

Хоча підхід простіший за ML, він все ж вимагає історичних даних. У зовсім новому проекті, де ще не було релізів і баг-фіксів, метрика $N_{defects}$ не дасть користі (нуль для всіх модулів), а $N_{changes}$ мало варіюватиметься. Тобто метод більш застосовний для зрілих проектів з певною історією. Також доводиться інтегрувати дані з кількох джерел – репозиторію коду і системи багів – що іноді нетривіально (потрібна надійна прив'язка комітів до дефектів, наприклад, через номери задач). Ще одна потенційна вада – лінійність моделі: реальні залежності можуть бути складніші, і фіксована формула з вагами може не врахувати якихось комбінацій факторів. У цьому сенсі IDP поступається гнучкості ML-моделей. Нарешті, вибір ваг може внести суб'єктивність: різні експерти можуть задати різні ваги, і результати зміняться. Це частково вирішується прозорістю – завжди можна відстежити, як ваги вплинули – але відсутність автоматичного способу їх визначення без даних теж мінус.

Класичні метрики та індекси на їх основі хороші для базового контролю якості та легкі у впровадженні, але можуть пропустити проблемні області, якщо не дивитися на історію. ML-моделі – найкращі для прогнозування дефектів, проте потребують даних і можуть бути важкими в інтерпретації та підтримці. Запропонований підхід зі збагаченими метриками намагається збалансувати ці аспекти, підвищивши точність оцінки ризику (за рахунок використання історичних ознак) при збереженні зрозумілості метрик і відносно невеликої складності застосування. Він особливо корисний у ситуаціях, коли команда хоче мати єдиний показник якості коду, що враховує і поточний стан, і накопичений досвід (проблеми) в коді.

Підкріплення такого підходу знаходимо і в наукових результатах: за даними досліджень, додавання метрик процесу до моделей якості суттєво покращує їх

результативність, а комбінування різних типів метрик дає більш стабільні і точні передбачення дефектів. Наш інтегральний показник IDP концептуально продовжує ці ідеї, пропонуючи практичний інструмент вимірювання, який може бути використаний як для виявлення «вузьких місць» у коді, так і для відстеження ефекту рефакторингів або змін процесу розробки на якість продукту в часі. У таблиці показано узагальнення основних характеристик розглянутих підходів.

Таблиця 2.1 – Порівняння підходів до оцінки якості ПЗ за ключовими критеріями.

Підхід	**Точність (прогноз дефектів)**	**Інтерпрето-ваність**	**Дані та ресурси**	**Адаптив-ність**	**Складність впровадження**
Класичні метрики (CK)	Невисока, окремо слабко прогнозують дефекти[3]	Висока (прості показники)	Мінімальні (лише код)	Низька (статичні показники)	Дуже проста (інструменти існують)
Індекс підтриму-ваності (MI)	Невисока для дефектів (не враховує історію)	Середня (агрегований індекс, але формула відома)	Мінімальні (код + метрики для формули)	Низька (статичний, потрібно переглядати після змін)	Проста (вбудовано в багато IDE)
ML-моделі (RF, XGB, NN)	Висока (до 85–90% в експериментах) [31]	Низька (чорні ящики)	Високі (історія метрик + дефекти, експертиза)	Висока (перенавчається, якщо дані є)	Складна (потрібні збір даних, навчання моделей)
Запропо-нований IDP	Помірна–висока (краще за статичні, трохи гірше за ML)	Висока (лінійна формула зрозуміла)	Середні (код + історія змін + баги)	Помірна (перераховується при оновленні даних)	Помірна (скрипти збору даних з VCS/BugTracker)

2.4 Висновки до другого розділу

Таким чином, удосконалення метрик якості шляхом врахування історії змін коду та дефектів дозволяє отримати більш інформативні показники, які можуть підвищити точність оцінки та прогнозування якості програмного забезпечення без значної втрати прозорості методики. Запропонований набір метрик та інтегральний показник IDP теоретично обґрунтовані і мають підтвердження у вигляді відомих емпіричних фактів таких як значимість метрик процесу, користь поєднання метрик. Наступний розділ присвячено практичній реалізації запропонованих метрик та перевірці їх ефективності на реальних даних (впровадження в інструмент та експериментальна оцінка).

3 АРХІТЕКТУРА СИСТЕМИ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ МЕТРИКИ IDP

3.1Формування та аналіз вимог програмної реалізації системи оцінки якості програмного забезпечення на основі метрики IDP

У цьому розділі описано розроблений програмний засіб для оцінювання якості програмного забезпечення на основі інтегрованої метрики схильності до дефектів (Integrated Defect Proneness, IDP). Запропонований інструмент призначений для автоматизованого аналізу історії змін коду та обчислення комплексної метрики, що прогнозує ймовірність появи дефектів у програмних модулях. Наведено загальну характеристику створеної системи, вимоги до її середовища та вхідних даних, архітектурне рішення, алгоритми обробки даних, а також реалізацію ключових модулів із фрагментами коду. Продемонстровано приклад використання інструменту та обговорено способи його запуску й інтеграції. Наприкінці розглянуто обмеження реалізації та можливі напрямки розширення функціональності.

Розроблено консольний програмний інструмент IDP-Analyzer (умовна назва), призначений для оцінки якості програмних модулів на основі метрики IDP. Основним призначенням системи є виявлення потенційно ненадійних (дефектонебезпечних) файлів або компонентів у репозиторії вихідного коду шляхом аналізу даних системи контролю версій Git. Інструмент дозволяє проаналізувати історію змін проєкту та розрахувати для кожного модуля сукупність метрик (кількість змін, кількість дефектів, кількість авторів, показники churn, характеристики власності коду, вік коду, складність функцій тощо), які потім агрегуються у фінальний показник IDP.

Основні функції системи включають:

Збір даних з Git-репозиторію, а саме автоматичне вилучення історії комітів, інформації про авторів змін, переліку модифікованих файлів, кількості доданих/видалених рядків коду та міток, що вказують на виправлення дефектів.

Обчислення метрик якості, куди входить розрахунок для кожного файлу (модуля) показників процесу розробки, зокрема кількості комітів, кількості унікальних авторів, метрик плинності коду (code churn), середнього віку коду, кількості історичних дефектів, а також статичної метрики складності функцій (цикломатична складність).

Агрегація метрик в інтегральний показник через виконання нормалізації значень метрик та обчислення підсумкового індексу IDP на основі зваженої суми, що відображає відносну схильність модуля до дефектів.

Генерація звітів у вигляді представлення отриманих результатів у зручному текстовому форматі (таблиця або список) з ранжуванням модулів за значенням IDP; можливість збереження результатів у файл для подальшого аналізу.

Рекомендації для тестування, що формуються на основі отриманих значень IDP інструмент може вказувати, на які модулі варто звернути першочергову увагу при тестуванні та код-рев'ю (файли з найвищим показником IDP вважаються найбільш ризикованими).

Цільовими користувачами даної системи є передусім інженери якості та технічні ліди проєктів, які прагнуть завчасно ідентифікувати проблемні ділянки коду. Інструмент може бути корисним також для розробників при оцінці стану спадкового коду або інтеграції сторонніх компонентів, дозволяючи отримати об'єктивні показники якості на основі історичних даних. Система розрахована на використання як в процесі безперервної інтеграції/доставки (CI/CD), так і як окремий етап аналізу якості коду під час рефакторингу чи аудитів.

Реалізований програмний засіб є крос-платформним і може виконуватися в середовищах Windows, Linux або macOS за умови наявності встановленого інтерпретатора Python 3.x. Розробку виконано мовою Python, що обумовлено багатою екосистемою бібліотек для аналізу Git-репозиторіїв та обробки даних. В ході реалізації використано такі основні бібліотеки:

PyDriller – високорівнева бібліотека для отримання інформації про коміти та зміни файлів безпосередньо з репозиторію Git. Вона надає зручний API для ітерації

по комітах та вилучення метрик (кількість доданих/видалених рядків, автор коміту, файли, що змінилися тощо).

GitPython – альтернативна бібліотека для доступу до Git, яка може використовуватися для низькорівневих операцій (наприклад, виклик Git-команд через Python) у випадках, коли PyDriller не надає потрібної функціональності.

Radon – пакет для статичного аналізу Python-коду, що дозволяє обчислювати метрики, зокрема цикломатичну складність. Використовується для оцінки складності функцій у модулях (за потреби). Для аналізу проєктів на інших мовах можливе підключення утиліти Lizard або аналогічних засобів, які обчислюють складність для різних мов.

Стандартні бібліотеки Python, наприклад, datetime, math, csv) – застосовуються для допоміжних операцій: обчислення інтервалів часу (вік коду), математична нормалізація значень, формування та запис звітів тощо.

Для коректної роботи інструмента необхідна наявність доступу до цільового Git-репозиторію. В якості вхідних даних система приймає локальний Git-репозиторій – шлях до каталогу проєкту, що містить підкаталог.git. Інструмент відкриває наявний репозиторій і аналізує всю його історію (за замовчуванням – основну гілку master/main, але у разі потреби може бути вказана інша гілка).

У цьому випадку перед аналізом засіб автоматично виконує клонування репозиторію у тимчасовий каталог. Такий режим дозволяє аналізувати публічні репозиторії. наприклад, з GitHub без попереднього ручного завантаження коду.

Формат вихідних даних, тобто результат роботи програми представляється у вигляді таблиці метрик для кожного модуля. За замовчуванням виведення здійснюється у консоль у вигляді, зручному для читання людиною – табличний список із колонками: назва файлу, значення окремих метрик, інтегральний показник IDP. Передбачено можливість збереження звіту до файлу формату CSV або JSON (за допомогою відповідних опцій командного рядка). Це дозволяє імпортувати результати в інші інструменти або електронні таблиці для додаткового аналізу чи візуалізації. Обсяг вихідних даних залежить від розміру проєкту: для

великих репозиторіїв корисно задавати фільтри (наприклад, виводити лише топ-10 файлів за IDP або ігнорувати файли з дуже малим числом змін).

З точки зору продуктивності, система не висуває особливих вимог до апаратного забезпечення: достатньо стандартної конфігурації ПК. Проте для аналізу надвеликих проєктів (сотні тисяч комітів, тисячі файлів) бажано мати не менше 8 ГБ ОЗП та багатоядерний процесор для прискорення обробки. Час роботи інструменту залежить від кількості комітів та файлів: у гіршому випадку складність обчислень близька до лінійної O(N + F), де N – число комітів, F – число файлів у репозиторії.

3.2 Розробка архітектури програмної системи оцінки якості програмного забезпечення на основі метрики IDP

Логічна архітектура програми побудована модульно і відображає основні етапи обробки даних. Всього виділено 4 основні компоненти програмного засобу.

Модуль збору даних (RepoAnalyzer) відповідає за підключення до Git-репозиторію та вилучення необхідної інформації про історію розробки. Цей компонент отримує послідовність комітів та для кожного коміту визначає перелік змінених файлів і характеристики змін. RepoAnalyzer використовує можливості PyDriller для ітерації по комітах: отримання SHA-хеша коміту, автора, дати, списку модифікацій. Для кожного файлу, зміненого в коміті, фіксується кількість доданих і видалених рядків коду та оновлюється лічильник змін цього файлу. Також аналізується текст повідомлення коміту на наявність ключових слів, що вказують на виправлення дефекту (наприклад, «fix», «bug», «issue»). Результатом роботи модуля є структура даних (словник або таблиця), в якій для кожного файлу агреговано такі показники: кількість комітів, що його змінювали; сумарна кількість доданих та видалених рядків (показники code churn); перелік авторів, які змінювали

файл; кількість комітів, помічених як виправлення дефектів; дата першого та останнього змінення файлу тощо.

Модуль обчислення метрик (MetricsCalculator) працює на основі сирих даних з RepoAnalyzer обчислює вторинні метрики якості для кожного файлу. Зокрема, цей модуль підраховує значення наступних показників:

NC (Number of Commits) – кількість комітів, що модифікували файл.

ND (Number of Defects) – кількість раніше виявлених дефектів у файлі (кількість комітів, що позначені як багфікси для даного файлу).

AU (Authors Unique) – кількість унікальних авторів, що редагували файл.

ChA (Churn Absolute) – абсолютна плинність коду, тобто сумарна кількість рядків, доданих або видалених у файлі за весь час.

ChR (Churn Relative) – відносна плинність, наприклад відношення ChA до поточного розміру файлу в рядках (частка зміненого коду) або середній обсяг змін на коміт.

Age (Code Age) – показник віку коду, що може визначатися як час (у днях) від останнього змінення файлу. Також можливий складніший варіант - середній вік рядків (час від останнього редагування кожного рядка, зважений за кількістю змін), але для спрощення використовується саме дата останнього коміту.

OW (Ownership) – метрика концентрації власності коду. Вона може визначатися як частка змін, внесених найбільшим автором файлу. Для зручності інтерпретації використовується комплементарне значення: наприклад, якщо один розробник зробив 80% змін, то OW = 1 - 0.8 = 0.2 (низьке значення означає високу концентрацію, високе – розпорошеність внеску).

CPX (Complexity) – середня цикломатична складність функцій у файлі або максимальна складність серед функцій (для оцінки структурної складності коду). Ця метрика потребує парсингу коду; вона обчислюється за допомогою інтегрованого статичного аналізатора (Radon, Lizard тощо) і може бути не розрахована для деяких мов. У разі відсутності даних CPX може ігноруватися при фінальному підрахунку IDP.

Деякі з наведених метрик потребують додаткових даних: наприклад, для CPX модуль MetricsCalculator взаємодіє зі статичним аналізатором, а для точного розрахунку Age може використовуватися команда git blame, щоб врахувати вік кожного рядка. Проте, в базовій реалізації достатньо інформації з журналу комітів. Усі обчислені значення метрик зберігаються у проміжній таблиці разом із даними RepoAnalyzer.

Модуль агрегування та оцінювання (IDPScorer) виконує нормалізацію метрик та розрахунок інтегрованого показника IDP для кожного файлу. Спочатку всі метрики приводяться до єдиної шкали, наприклад, 0..1 – для цього використовується лінійна нормалізація min-max або інший метод масштабування. Важливо відзначити, що для метрик, які обернено пропорційні дефектності, наприклад, вік коду – чим більший, тим менший ризик, нормалізовані значення обчислюються з інверсією. Наприклад, нормалізоване значення віку може визначатися як, щоб молодший код отримував більше значення (більший ризик), а старіший – менше. Після нормалізації для кожної метрики виконується зважування відповідно до встановлених коефіцієнтів значущості. Далі обчислюється зважена сума – власне показник IDP (7):

$$IDP_i = \sum_{j=1}^{m} w_j \times f\left(M_{ij}\right) \tag{7}$$

де – значення j-ої метрики для і-го модуля, – функція нормалізації, – вага (коефіцієнт) для j-ої метрики, а m – кількість врахованих метрик.

Значення нормується у діапазон (0..1) для зручності інтерпретації, де 0 відповідає мінімальному очікуваному ризику виникнення дефектів, 1 – максимальній у межах аналізованого проєкту. На практиці більшість файлів матимуть значення IDP десь посередині цього інтервалу, а кілька найбільш проблемних наближатимуться до 1.

Модуль інтерфейсу (CLI Interface): забезпечує взаємодію з користувачем через інтерфейс командного рядка. Він відповідає за обробку вхідних параметрів

(шлях до репозиторію, опції виводу тощо), ініціалізацію інших модулів та формування вихідного звіту. Цей компонент реалізує логіку форматування текстового виводу: сортування файлів за значенням IDP, оформлення таблиці (вирівнювання колонок, додавання заголовків), округлення числових значень до потрібної точності (наприклад, IDP до двох знаків після коми). Також CLI-модуль обробляє можливі виключні ситуації та помилки – у таких випадках користувачу виводиться зрозуміле повідомлення і довідкова інформація щодо правильного використання програми.

Взаємодія між компонентами відбувається наступним чином: модуль CLI отримує від користувача параметри та ініціює аналіз, викликаючи RepoAnalyzer для збору даних. RepoAnalyzer формує внутрішню структуру (наприклад, словник) з метриками змін для кожного файлу і передає її до MetricsCalculator, який обчислює на її основі значення потрібних метрик. Далі підготовлені дані надходять до IDPScorer, де здійснюється нормалізація та обчислення інтегральних оцінок. Нарешті, CLI-модуль одержує від IDPScorer список файлів з їхніми IDP та виводить його у встановленому форматі. Така модульна побудова спрощує підтримку і розвиток системи: кожен компонент виконує свою функцію і може вдосконалюватися незалежно (наприклад, додавання нової метрики потребуватиме зміни лише в MetricsCalculator і, можливо, коригування ваг в IDPScorer).

3.3 Алгоритм обробки даних системи оцінки якості програмного забезпечення на основі метрики IDP

Алгоритм роботи інструменту IDP-Analyzer логічно поділяється на послідовність етапів, що відповідають описаним вище компонентам. Загальна схема алгоритму:

Ініціалізація та налаштування: користувач запускає програму із зазначенням шляху до репозиторію та потрібних опцій. Встановлюються початкові параметри,

наприклад, які метрики обчислювати, у який файл зберегти звіт тощо. На цьому етапі здійснюється парсинг аргументів командного рядка і підготовка до аналізу, в тому числі, за необхідності, клонування віддаленого репозиторію у локальний тимчасовий каталог.

Для збору історичних даних відкривається Git-репозиторій, отримується повний список комітів, наприклад, через ітерацію за допомогою RepositoryMining з PyDriller. Для кожного коміту зберігаються необхідні атрибути: дата, автор, повідомлення, а також детальна інформація про змінені файли. За допомогою об’єкта коміту отримується список модифікацій (commit.modifications), де для кожного зміненого файлу доступні: назва файлу, кількість доданих (added) і видалених (removed) рядків, статус файлу (змінений, новий, видалений), тощо. Ці дані накопичуються у проміжній структурі metrics_data.

Агрегація даних по файлах відбувається таким чином, що проходячи послідовно всі коміти, програма оновлює записи в metrics_data для відповідних файлів. Якщо певного файлу ще немає в структурі, створюється новий запис. Далі при кожному наступному зміненні файлу оновлюються відповідні лічильники:

metrics_data[file], ['commits'] збільшується на 1 (кількість комітів).

metrics_data[file], ['added_lines'] та metrics_data[file], ['deleted_lines'] збільшуються на кількість доданих і видалених рядків у цьому коміті.

У множину metrics_data[file], ['authors'] додається ім’я або email поточного автора коміту.

Якщо текст повідомлення коміту містить підрядки на кшталт «fix» або «bug», збільшується лічильник metrics_data[file], ['bug_fixes'] (припускається, що коміт виправляє дефект).

Нижче наведено спрощений фрагмент коду, що реалізує описаний процес з використанням PyDriller (лістинг 3.1):

```
        from pydriller import RepositoryMining
        metrics_data = {}  # словник для накопичення метрик по файлах
        for commit in RepositoryMining(repo_path).traverse_commits():
            for mod in commit.modifications:         # перебір змінених у коміті
файлів
```

```
file = mod.new_path                    # шлях/назва файлу після змін
if file not in metrics_data:
    metrics_data[file] = {
        "commits": 0,
        "added_lines": 0,
        "deleted_lines": 0,
        "authors": set(),
        "bug_fixes": 0,
        "first_commit_date": commit.committer_date
    }
data = metrics_data[file]
data["commits"] += 1
data["added_lines"] += mod.added
data["deleted_lines"] += mod.removed
data["authors"].add(commit.author.email)
if "fix" in commit.msg.lower():
    data["bug_fixes"] += 1
# оновлення дати останнього змінення файлу
data["last_commit_date"] = commit.committer_date
```

Лістинг 3.1 – Фрагмент коду модуля RepoAnalyzer для збору даних про зміни з Git-репозиторію

Розрахунок метрик файлів: після проходження всіх комітів структура metrics_data містить зібрану інформацію щодо кожного файлу. Далі, для кожного файлу обчислюються похідні метрики:

NC = commits – загальна кількість комітів.

ND = bug_fixes – кількість виправлених дефектів (зафіксованих як багфікси комітів).

AU = len(authors) – кількість різних авторів.

ChA = added_lines + deleted_lines – абсолютна плинність коду (всього змінених рядків).

ChR = ChA / LOC – відносна плинність (де LOC – поточна кількість рядків у файлі, визначена, наприклад, при останньому коміті або шляхом читання файла з репозиторію; дане відношення показує, яка частка коду була змінена).

Age = CurrentDate - last_commit_date – вік файлу (час, що минув від останнього коміту, у днях). Для прикладу, якщо файл останній раз змінювався 30 днів тому, то Age = 30.

$OW = 1 - \frac{\max lines\ by\ on\ author}{ChA}$ – метрика розпорошеності власності: частка змін, що припадає не на основного автора. Тут max_lines_by_one_author – кількість

рядків, змінених найбільшим вкладником файлу. Значення OW близьке до 0 означає, що один розробник зробив майже всі зміни (концентрація власності висока), а значення ближче до 1 – що внесок розподілений між багатьма авторами.

CPX – оцінка складності коду (наприклад, середня цикломатична складність функцій). Для її отримання файл аналізується статично: використовується бібліотека Radon (для Python) або аналогічна утиліта для іншої мови. Якщо аналіз доступний, отримане значення додається до метрик файлу; інакше для CPX встановлюється спеціальний маркер (наприклад, None) або 0, і ця метрика не буде врахована при обчисленні IDP.

На цьому етапі формується підсумкова таблиця метрик для кожного файлу: $M_i = \{NC_i, ND_i, AU_i, ChA_i, ChR_i, Age_i, OW_i, CPX_i\}$. Якщо якісь метрики неможливо отримати (наприклад, CPX для файлів мов, що не підтримуються аналізатором), значення залишаються порожніми або нульовими.

Нормалізація метрик: для коректного агрегування різнорідних показників їх необхідно привести до порівнянної шкали. Програма обчислює екстремальні значення по кожній метриці серед усіх файлів (мінімум і максимум), після чого для кожного значення застосовується нормалізація за формулою (8):

$$f(x) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{8}$$

що перетворює значення x на відрізок (0..1). При цьому враховуються особливості окремих метрик.

Якщо більші значення метрики означають гіршу якість (наприклад, більше комітів, дефектів або авторів – це негативні фактори), то нормалізація використовується без змін, як у наведеній формулі (тобто більше значення перетвориться ближче до 1).

Якщо більші значення означають кращу якість (наприклад, вік коду – чим старіший, тим надійніший), то після нормалізації виконується інверсія:. Це

забезпечує, що молодий (нещодавно змінений) код отримає високе нормалізоване значення ризику, а старий стабільний – низьке.

У випадку, якщо для деякої метрики всі файли мають однакове значення (нульову дисперсію), нормалізація може призвести до невизначеності (ділення на нуль). Така ситуація малоймовірна для динамічних метрик, але в реалізації передбачено перевірку: якщо $x_{max} = x_{min}$, всім файлам присвоюється середнє нормалізоване значення 0.5 або ця метрика виключається з розгляду.

Результатом кроку є нормалізована матриця значень $M_i^{norm} = \{m_{i1}, m_{i2}, \dots, m_{im}\}$, де кожний $m_{ij} \in [0; 1]$. Для метрик, не розрахованих через брак даних, можуть бути застосовані нейтральні значення (наприклад, 0.5) або ці метрики виключаються з формули IDP (їх ваги можуть тимчасово перерозподілятися між іншими).

Обчислення інтегрального показника IDP: застосовується зважена модель згортки метрик. Для кожної метрики визначено ваговий коефіцієнт w_j, що відображає її відносну важливість у прогнозуванні дефектності. Ваги встановлено на основі синтезу результатів досліджень та експертних оцінок (обґрунтування вибору ваг наведено у розділі 2). Наприклад, метрика історичної дефектності (ND) отримує найбільшу вагу, оскільки емпіричні роботи показують, що наявність дефектів у минулому є найсильнішим індикатором проблем у майбутньому. Пропоновану схему ваг представлено в таблиці 3.1.

Таблиця 3.1 – Вагові коефіцієнти метрик у моделі IDP

Метрика	**Позначення**	**Вага**
Кількість історичних дефектів	ND	0,35
Відносна плинність коду	ChR	0,20
Кількість унікальних авторів	AU	0,15
Розпорошеність власності	OW	0,10
Кількість змін	NC	0,10

Продовження таблиці 3.1

Метрика	**Позначення**	**Вага**
Абсолютна плинність коду	ChA	0,05
Вік коду	Age	0,05
Всього	-	1,00

Метрика складності CPX до інтегрального показника наразі не включена (її вагу можна вважати рівною 0) через обмежену прогнозувальну силу статичних метрик порівняно з процесними. За необхідності значення CPX може враховуватися у майбутніх версіях моделі (наприклад, шляхом зменшення ваг інших метрик).

Після застосування ваг для кожного файлу і обчислюється як у 9:

$$IDP_i = \sum_j w_j \times m_{ij}^{norm} \tag{9}$$

Цей розрахунок реалізовано у коді як цикл по файлах з підсумовуванням внеску кожної метрики. Фрагмент коду для обчислення IDP наведено у листингу 3.2:

```
        results = []
        for file, data in metrics_data.items():
            # припускаємо, що norm_values містить нормалізовані значення метрик
            nd  = data['norm_ND']
            chR = data['norm_ChR']
            au  = data['norm_AU']
            ow  = data['norm_OW']
            nc  = data['norm_NC']
            chA = data['norm_ChA']
            age = data['norm_Age']
            # Обчислення інтегральної оцінки з використанням ваг
            idp_score = 0.35*nd + 0.20*chR + 0.15*au + 0.10*ow + 0.10*nc + 0.05*chA
+ 0.05*age
            results.append((file, idp_score))
        # Сортування файлів за спаданням IDP
        results.sort(key=lambda x: x[1], reverse=True)
```

Лістинг 3.2 – Обчислення інтегрального показника IDP для кожного модуля на основі нормалізованих метрик.

Виведення результатів: сформований список results (файл, значення IDP) передається до модуля інтерфейсу для форматування і виведення. Програма генерує підсумкову таблицю, де для кожного файлу вказані ключові метрики та значення IDP. Вивід упорядковується за спаданням IDP, щоб у верхній частині списку знаходилися найризикованіші модулі. Також додається легенда або пояснення (наприклад, що максимальне значення IDP = 1 відповідає модулю з найбільшим ризиком у даному проєкті). Якщо користувач указав опцію збереження у файл, результати додатково записуються у вибраний формат. На цьому робота алгоритму завершується.

Описаний алгоритм реалізує методологію зваженої оцінки якості, забезпечуючи прозорий та інтерпретований показник IDP для кожного програмного модуля. Часова складність основних стадій – O(N) для збору історії комітів та O(F) для обчислення метрик і IDP – є лінійною, що дозволяє масштабувати застосування інструменту на великі проєкти.

3.4 Програмна реалізація системи оцінки якості програмного забезпечення на основі метрики IDP

Програмна реалізація організована у вигляді декількох класів, що відповідають згаданим архітектурним компонентам. Нижче наведено опис ключових класів та їхніх функцій:

RepoAnalyzer – клас, що інкапсулює логіку доступу до Git-репозиторію та збору історичних даних. Основний метод – collect_metrics(repo_path), який відкриває репозиторій за вказаним шляхом і повертає структуру metrics_data з накопиченими метриками по файлах. Внутрішньо цей метод використовує ітерацію по комітах (приклад наведено в листингу 3.1) і заповнює словник metrics_data. Додатково, RepoAnalyzer може містити допоміжні функції: наприклад, фільтрацію комітів (ігнорування мердж-комітів, дуже великих автоматичних оновлень тощо)

або вирівнювання історії для перейменованих файлів, щоб об’єднати метрики файлу до і після перейменування на основі інформації з commit.modifications.

MetricsCalculator – клас, відповідальний за обробку даних із metrics_data та обчислення похідних метрик для кожного файлу. Метод calculate_metrics(metrics_data) проходить по кожному файлу в структурі та доповнює його запис обчисленими значеннями (ChA, ChR, Age, OW, CPX тощо). Тут зосереджено реалізацію формул, описаних у пункті 3.4.4. Також MetricsCalculator може взаємодіяти зі сторонніми інструментами: наприклад, викликати Radon для обчислення CPX через функцію compute_complexity(file_path). Тримання всієї логіки обчислення метрик у межах цього класу спрощує модифікацію – у разі зміни формули чи додавання нової метрики достатньо правити код тільки тут.

Метод compute_idp (metrics_table) приймає на вхід таблицю метрик по файлах (може бути той самий оновлений metrics_data) і повертає список кортежів (файл, IDP). Основні кроки: нормалізація метрик (може бути реалізована через окремий метод, наприклад, normalize_values(metric_name, values)), застосування вагових коефіцієнтів та обчислення суми. Ваги метрик задані як константи класу або ж зчитуються з конфігурації. Для полегшення налагодження та аналізу IDPScorer може також видавати проміжну інформацію – наприклад, друкувати максимальні/мінімальні значення метрик або кількість файлів, для яких певна метрика не розрахована. Після обчислення IDP метод сортує результати за спаданням оцінки і повертає їх.

ReportGenerator – допоміжний клас (або набір функцій) у складі CLI-модуля, що форматує та виводить результати. Методи print_console(results) та save_to_csv(results, filename) забезпечують перетворення списку results у потрібний формат. Перший із них друкує таблицю в консоль із вирівнюванням стовпців, другий – записує дані у CSV-файл (де стовпці, наприклад, Module, Commits, Defects, Authors, IDP). При необхідності може бути реалізовано також save_to_json чи інші формати виводу. ReportGenerator реалізує лише представлення, тому зміна формату звіту не потребує втручання у логіку збору чи обчислення метрик.

IDPAnalyzerApp – умовний клас верхнього рівня (або просто головна функція), який об’єднує всі компоненти для послідовного виконання задач. Він обробляє аргументи командного рядка (наприклад, через бібліотеку argparse), створює екземпляри класів RepoAnalyzer, MetricsCalculator, IDPScorer, ReportGenerator та по черзі викликає їх методи: збір даних -> обчислення метрик -> оцінка IDP -> генерація звіту. У випадку виникнення винятків (помилок вводу/виводу, проблем з парсингом репозиторію тощо) IDPAnalyzerApp перехоплює їх та виводить зрозумілі повідомлення, не допускаючи аварійного завершення програми.

Нижче наведено спрощений приклад використання зазначених класів у головному модулі (сценарій роботи програми):

```
repo_path = "/path/to/repository"

analyzer = RepoAnalyzer()
metrics_data = analyzer.collect_metrics(repo_path)

calculator = MetricsCalculator()
metrics_table = calculator.calculate_metrics(metrics_data)

scorer = IDPScorer()
results = scorer.compute_idp(metrics_table)

report = ReportGenerator()
report.print_console(results)
```

Такий код демонструє послідовність викликів: спочатку збираються дані з репозиторію, потім обчислюються метрики, далі розраховується IDP та виводяться результати. У реальній реалізації ці кроки можуть бути оформлені дещо інакше, наприклад, в одному методі run() класу IDPAnalyzerApp), але загальна логіка залишається як в наведеному прикладі.

Для ілюстрації роботи розробленого засобу розглянемо умовний приклад. Нехай маємо репозиторій проєкту, що містить три модулі: core.py, utils.py та legacy.py.

Модуль core.py – активно розроблюваний центральний компонент (має значну кількість змін, декількох авторів, зафіксовано декілька дефектів).

utils.py – допоміжний модуль, код якого змінювало багато розробників; спостерігався високий churn (багато доданого/видаленого коду) і значна кількість виправлень помилок.

legacy.py – стабільний «спадковий» модуль, що майже не змінювався останнім часом, підтримується одним автором і не має зареєстрованих дефектів.

При запуску інструменту на даний репозиторій за допомогою командного рядка (без додаткових опцій) відпрацьовують усі етапи, описані в підрозділі 3.4, і в результаті на екран виводиться зведений звіт. Команда запуску може виглядати так:

$ idp_analyzer /path/to/repository (тут idp_analyzer – назва виконуваного скрипта, а /path/to/repository – шлях до каталогу з Git-репозиторієм).

Припустимо, отримано наступний результат (лістинг 3.3):

```
Модуль        Комітів   Авторів   Дефектів   IDP
-----------------------------------------------
core.py       15        3         2          0.73
utils.py      30        5         5          0.95
legacy.py     5         1         0          0.10
```

Лістинг 3.3 – Приклад фрагменту вихідного звіту програми IDP-Analyzer для демонстраційного репозиторію.

У наведеному звіті кожен рядок відповідає файлу та містить ключові метрики і інтегральну оцінку IDP. Зокрема, модуль utils.py має найвищий показник IDP = 0.95 (або 95%), що узгоджується з його характеристиками: велика кількість змін (30 комітів) і авторів (5) вказує на нестабільність і розмитість відповідальності, а значна кількість виправлених дефектів (5) прямо сигналізує про проблемність коду. Модуль core.py має IDP = 0.73 – дещо нижчий ризик: хоча змін багато (15 комітів) і авторів декілька (3), історичних дефектів менше (2), що поліпшує ситуацію. Нарешті, legacy.py отримав дуже низький IDP = 0.10, що вказує на його відносну надійність: лише 5 комітів одним автором і жодного зареєстрованого дефекту.

Таким чином, інструмент правильно ідентифікував модулі з підвищеним ризиком. У реальному сценарії команда розробки, отримавши подібний звіт, може прийняти рішення приділити більше уваги тестуванню та рефакторингу файлу

utils.py, аби знизити його дефектонебезпечність. В той же час legacy.py можна вважати достатньо стабільним і не витрачати надмірних ресурсів на його додаткову перевірку.

Зазначимо, що для великих проєктів інструмент може генерувати доволі великий звіт (сотні файлів). У таких випадках варто застосовувати опцію виводу лише топ-N файлів за IDP або експортувати дані у файл для більш детального аналізу (наприклад, побудови графіків залежності метрик від IDP).

Реалізований інструмент функціонує як консольна утиліта, і його використання здійснюється через інтерфейс командного рядка. Синтаксис виклику програми узагальнено виглядає так, як це показано у (9):

```
idp_analyzer [опції] <шлях_до_репозиторію>   (9)
```

де <шлях_до_репозиторію> – обов'язковий аргумент, що задає локальний шлях або URL Git-репозиторію для аналізу. Додатково можуть бути вказані опції (ключі), які змінюють режим роботи інструменту:

-b, --branch <name> – проаналізувати конкретну гілку репозиторію (за замовчуванням аналізується основна гілка master/main).

-o, --output <file> – зберегти результати аналізу у вказаний файл. Формат файлу визначається розширенням:.csv – таблиця CSV,.json – JSON. Без цієї опції результати тільки друкуються в консоль.

-t, --top <N> – вивести лише топ-N файлів з найбільшими значеннями IDP. Допомагає сфокусуватися на найризикованіших модулях у проектах з великою кількістю файлів.

--no-static – пропустити обчислення статичних метрик складності, щоб пришвидшити аналіз (актуально, якщо статична оцінка складності займає багато часу або не підтримується для даної мови).

-h, --help – вивести довідку з описом усіх доступних опцій.

Наприклад, команда:

```
$ idp_analyzer -b develop -o report.csv -t 20 /proj/MyProject
```

проаналізує гілку develop репозиторію, виведе у консоль 20 файлів з найвищим IDP та додатково збереже повний звіт у файл report.csv. Опція --no-static могла б бути використана, якщо, скажімо, репозиторій містить код на кількох мовах і обчислення CPX для деяких з них не реалізовано.

Інструмент IDP-Analyzer може бути легко інтегрований у процес розробки. Зокрема, його можна включити в CI/CD-конвеєр для автоматичного запуску аналізу якості при кожному коміті або перед релізом. Наприклад, на сервері CI (Jenkins, GitLab CI тощо) можна додати крок, що викликає idp_analyzer для поточного репозиторію, а результати (CSV-файл) зберігаються як артефакт або надсилаються команді розробників. Це дозволить відстежувати динаміку метрики IDP у часі та своєчасно реагувати на погіршення показників (наприклад, якщо у певному модулі після серії комітів різко зріс IDP, це сигнал приділити йому увагу).

Крім того, інструмент може використовуватися й у локальному режимі окремими розробниками чи аналітиками як частина практики Code Review. Оскільки він написаний на Python, можливе безпосереднє використання його класів у сторонніх сценаріях. Наприклад, можна імпортувати RepoAnalyzer та IDPScorer в інтерактивне середовище (Jupyter Notebook) і виконати аналіз кількох проєктів, порівнюючи їх метрики. Таким чином, IDP-Analyzer може слугувати модулем в більш загальній системі моніторингу якості (його вихідні дані легко парсити і обробляти іншими програмами).

Перспективним напрямком інтеграції є розширення інструмента до формату веб-служби. У цьому випадку на сервері розгортається застосунок, що отримує на вхід посилання на репозиторій (або архів із кодом) і повертає користувачу згенерований звіт (наприклад, веб-сторінку з інтерактивною візуалізацією IDP-метрик). Хоча в межах цієї магістерської роботи такий веб-інтерфейс не реалізовано, закладена архітектура дозволяє відносно нескладно додати його в подальшому (фактично, потрібно створити оболонку, що викликатиме наявні класи і відображатиме результати у браузері).

Розроблений програмний засіб успішно реалізує поставлену функціональність, проте має ряд обмежень, які слід враховувати при його використанні, а також відкриває можливості для подальшого вдосконалення.

Залежність від якості даних репозиторію, оскільки метрика IDP базується на припущенні, що історія комітів достовірно відображає процес розробки і дефекти. Якщо дефекти виправлялися без належних позначок у повідомленнях комітів або якщо частина історії втрачена (наприклад, репозиторій створено шляхом імпорту вже готового коду), показник може бути заниженим, а оцінка IDP – неточною. В таких випадках можливе доповнення інструменту інтеграцією з багтрекерами (Jira, Redmine) для отримання повнішої картини дефектів.

Межі застосування вагової моделі через встановлені ваги метрик є усередненими і можуть не враховувати специфіку конкретного проєкту. Наприклад, у деяких командах великий churn може бути нормою (через часті рефакторинги), а в інших – сигналом проблем. Поточна модель не адаптується автоматично під проєкт. Можливим розширенням є налаштування ваг: надання користувачу можливості змінювати коефіцієнти або використання алгоритмів машинного навчання для підбору оптимальних ваг на основі наявних даних про дефекти (якщо такі є).

Статичні метрики складності, оскільки метрика CPX включена скоріше як довідкова і за замовчуванням не впливає на IDP. Це зумовлено тим, що статичні метрики, як було показано в теоретичному розділі, сильно корелюють з розміром коду і мають обмежену прогностичну цінність. Тим не менш, для деяких модулів надмірна цикломатична складність може бути додатковим індикатором потенційних проблем із підтримкою. У майбутньому можна розглянути гібридну модель, де CPX враховується з малою вагою або впливає на формування рекомендацій (наприклад, файл з високим IDP і високою складністю потребує не лише тестування, а й рефакторингу).

Динамічність метрик у часі, бо поточна реалізація обчислює IDP на основі статичного зрізу історичних даних усіх комітів до поточного моменту. Проте, якість модулів може змінюватися з часом – наприклад, файл міг бути дуже

проблемним рік тому, але потім був суттєво переписаний і стабілізований. Вага старих дефектів і змін у поточній моделі така ж, як і нових. Можливим поліпшенням є введення «затухання» важливості старих даних, наприклад, враховувати тільки метрики за останні N місяців або зменшувати вплив дуже старих комітів. Це дозволило б IDP більш адекватно відображати поточний стан проєкту.

Оптимізація продуктивності, бо як зазначалося, на дуже великих репозиторіях повний аналіз може бути ресурсномістким. Одним зі шляхів оптимізації є паралельна обробка (наприклад, розподіл комітів між кількома потоками або процесами) – PyDriller наразі працює синхронно, але теоретично можна запускати кілька екземплярів для різних діапазонів комітів. Іншим шляхом є попередній фільтр файлів: наприклад, не аналізувати файли, що автоматично генеруються чи двійкові, оскільки від них мало користі для показників якості, але вони можуть збільшувати churn. У поточній версії таких фільтрів не реалізовано, але їх легко додати в RepoAnalyzer.

Інтерпретація результатів і зворотній зв’язок, оскільки IDP вказує на дефектонебезпечні модулі, але не пояснює причину. Для користувачів інструменту важливо розуміти, чому файл отримав високий IDP. Наразі це можна зрозуміти, глянувши на складові метрики у звіті (кількість дефектів, змін тощо). Проте доцільно додати автоматичне генерування стислого пояснення для кожного файлу з високим IDP. Наприклад, «Високе значення IDP обумовлене 5 виправленими дефектами та 5 різними розробниками, що часто змінювали файл протягом останнього року». Такий зворотний зв’язок підвищить довіру користувачів до метрики і вкаже напрям дій та допоможе сконцентруватися на комунікації між багатьма авторами, провести рев’ю дефектної історії тощо.

Перевірка моделі на практиці, оскільки остаточно якість і корисність інструменту повинні підтвердитися в реальних умовах. Хоча модель ґрунтується на емпіричних дослідженнях і логічних припущеннях, було б бажано провести експериментальну валідацію: застосувати інструмент до декількох відомих проєктів з відкритим кодом, для яких є дані про реальні дефекти, і перевірити, чи

справді модулі з високим IDP мають більшу дефектність. Таке дослідження виходить за рамки даної роботи, але воно є природним кроком для вдосконалення моделі (можливо, з подальшим калібруванням ваг або порогових значень).

Незважаючи на означені застереження, інструмент IDP-Analyzer уже зараз надає практичну цінність, а його архітектура дозволяє розширювати функціональність з мінімальними зусиллями. У підсумку отримано гнучкий засіб, який можна адаптувати під потреби конкретної команди чи проєкту від настроювання ваг до інтеграції з іншими системами якості та DevOps-практиками)

3.5 Висновки до 3-го розділу

Таким чином, у третьому розділі здійснено програмну реалізацію запропонованого підходу до оцінки якості ПЗ на основі інтегрованої метрики схильності модулів до дефектів. Розроблений консольний інструмент IDP-Analyzer виконує автоматизований аналіз історії змін Git-репозиторію та розрахунок метрики IDP, що дозволяє кількісно оцінити ризикованість програмних компонентів. Описано архітектуру рішення та алгоритми, які забезпечують прозорість і надійність отримуваних оцінок. Приклад використання інструменту підтвердив коректність його роботи – файли з найвищим IDP дійсно характеризуються факторами ризику (значною кількістю змін, дефектів, авторів). Отже, створений програмний засіб відповідає поставленим вимогам і може застосовуватися для підтримки процесу забезпечення якості програмного забезпечення, надаючи розробникам і менеджерам об’єктивні дані для прийняття рішень щодо пріоритезації тестування та рефакторингу модулів.

4 ПРАКТИЧНИЙ АНАЛІЗ ТА ЕКСПЕРИМЕНТАЛЬНА ПЕРЕВІРКА ІНСТРУМЕНТУ IDP-ANALYZE

4.1 Методологія здійснення експериментальної перевірки

Розроблений інструмент було оцінено з точки зору ключових характеристик функціонування: продуктивності, надійності та простоти використання.

IDP-Analyzer продемонстрував здатність ефективно обробляти проєкти середнього розміру протягом прийнятного часу. Швидкість аналізу залежить головним чином від кількості комітів у репозиторії та кількості вихідних файлів, оскільки для кожного коміту здійснюється обробка (через PyDriller) і для кожного файлу – аналіз коду (через Radon). У тестових випробуваннях (див. розділ 4.4) інструмент проаналізував репозиторій з ~500 комітів (близько 50 тис. рядків коду, ~100 файлів) приблизно за 2-3 хвилини на звичайному ПК. Така продуктивність вважається задовільною для статичного аналізу офлайн. Масштабованість інструменту обмежується продуктивністю Python та IO-операцій Git: для дуже великих репозиторіїв (тисячі комітів, сотні тисяч рядків коду) час аналізу лінійно зростатиме. Водночас, архітектура дозволяє паралелізувати деякі етапи – наприклад, аналіз метрик коду по файлах можна виконувати у декілька потоків, оскільки обчислення для різних файлів незалежні. PyDriller також підтримує фільтрацію діапазону комітів або авторів, що може бути використано для прискорення, якщо потрібно аналізувати не всю історію. Загалом, IDP-Analyzer придатний для проєктів малої та середньої складності, а для дуже великих систем може потребувати оптимізацій або вибіркового аналізу.

При розробці приділено увагу обробці можливих помилкових ситуацій, щоб інструмент не завершував роботу аварійно на реальних даних. Зокрема, реалізовано перевірку доступності репозиторію: якщо за вказаним шляхом немає Git-репозиторію або він некоректний – користувач отримає зрозуміле повідомлення про помилку. Під час ітерації комітів через PyDriller передбачено виняткові ситуації, наприклад, пошкоджені коміти або нестача прав доступу; ці винятки

перехоплюються та логуються, після чого аналіз продовжується з наступним комітом (де це можливо). Аналогічно, при аналізі файлів утилітою Radon – якщо файл не піддається парсингу (наприклад, містить синтаксичні помилки або специфічний код, не підтримуваний аналізатором), інструмент фіксує проблему, пропускає та рухається далі, щоб один проблемний файл не блокував увесь аналіз. Такі підходи підвищують надійність: IDP-Analyzer може працювати без нагляду на різних проєктах, гарантуючи, що помилки обробляються належним чином. Додатково, результат аналізу включає базову статистику про успішно опрацьовані файли/коміти і пропущені через помилки – це дозволяє користувачу оцінити повноту аналізу.

Інструмент спроектовано таким чином, щоб його використання не потребувало значних зусиль чи глибоких знань про внутрішню реалізацію. Завдяки інтерфейсу командного рядка, для запуску достатньо виконати команду.

Вбудована система підказок (ключ -h або --help) надає користувачу інформацію про доступні опції. Формат виведення результатів є читабельним: за замовчуванням в консолі виводиться таблиця з файлами та основними показниками (метрики та IDP), відсортована за спаданням IDP-індексу, що одразу акцентує увагу на найбільш ризикованих модулях. За допомогою параметрів можна налаштувати вивід – наприклад, показувати всі файли чи лише топ-10, змінювати формат (CSV для імпорту в Excel/Sheets) тощо. Простота використання також забезпечується мінімальними залежностями: необхідно лише мати встановленим Python та бібліотеки PyDriller і Radon (що автоматично встановлюються через pip). Немає потреби у розгортанні БД чи веб-сервера – інструмент працює автономно. Така легкість розгортання і застосування дозволяє інтегрувати IDP-Analyzer у процес розробки (наприклад, запускати його після кожного релізу або в CI/CD для моніторингу якості).

4.2 Результати обчислювального експерименту

Для перевірки працездатності та корисності розробленого інструменту було проведено обчислювальний експеримент на декількох відкритих програмних проектах. Метою експерименту було:

– протестувати IDP-Analyzer на реальних даних;

– отримати значення метрик якості та інтегрованого показника IDP для файлів у цих проектах;

– виявити файли, найбільш схильні до дефектів (дефектонебезпечні);

– оцінити, наскільки результати узгоджуються з очікуваннями та відомою історією дефектів у цих проектах.

Для тестування були відібрані два достатньо відомі репозиторії на Python з відкритим кодом, які мають багату історію змін та зафіксовані дефекти у своїй історії розробки:

Репозиторій Flask (github.com/pallets/flask) налічує понад 5000 комітів, історію розробки більше 10 років та тисячі рядків коду. Flask було обрано як приклад середнього за розміром проекту з активною спільнотою, де очікується наявність як складних ділянок коду, так і багатьох виправлень багів. За даними репозиторію, Flask має декілька основних модулів (routing, templating, etc.), і відомо про ряд історичних дефектів, наприклад, помилки безпеки у керуванні сесіями, помилки у відображенні шаблонів тощо. Це робить Flask хорошим кандидатом для аналізу дефектонебезпечності файлів: інструмент повинен виділити ті компоненти, в яких зосереджувалися численні зміни та виправлення.

Проєкт B – Requests: популярна бібліотека HTTP-клієнта для Python. Репозиторій Requests (github.com/psf/requests) має близько 3000 комітів і є прикладом бібліотеки, широко використовуваної розробниками. За роки розвитку в Requests також накопичено чимало виправлених багів (наприклад, обробка різних кодувань, перенаправлення, помилки роботи з TLS тощо). Requests обрано для

перевірки, чи зможе IDP-Analyzer коректно обробити дещо менший за розміром, але концентрований у функціональності проект, та чи виявить він файли-«hot spots» – файли, де найчастіше виникали проблеми (наприклад, sessions.py, models.py тощо, які за припущеннями мали багато змін через виправлення логіки HTTP-запитів).

Вибір двох різних за призначенням проєктів веб-фреймворк і мережна бібліотека дозволяє перевірити універсальність підходу. Обидва проєкти використовують Git, отже мають журнал змін, придатний для аналізу. Перед запуском експерименту репозиторії було клоновано локально, щоб забезпечити швидкий доступ до історії (PyDriller може працювати як з локальними шляхами, так і з віддаленими URL). Також було домовлено про критерії, за якими вважати файл дефектонебезпечним: файл, що має хоча б один історично підтверджений баг-фікс, тобто коміт, який виправляє дефект у цьому файлі. Інформацію про такі коміти отримано шляхом пошуку в повідомленнях комітів ключових слів (наприклад, «fix», «bug», «issue #») та аналізу списку змінених файлів у тих комітах. Додатково, для більшої точності, застосовано алгоритм SZZ – метод трасування баг-індукуючих змін: кожен коміт із виправленням помилки було проаналізовано на предмет того, які попередні коміти внесли ті рядки коду, що виправлялися. Це дозволило не лише ідентифікувати факт, що файл мав баг, але й знайти конкретні коміти, коли цей баг було внесено (що теж є показником проблемності файлу). Отриманий перелік «дефектонебезпечних» файлів у кожному проекті використовувався для оцінки якості прогнозу IDP-Analyzer.

Після запуску IDP-Analyzer на обраних проєктах було зібрано детальні дані про кожен вихідний файл. Інструмент для кожного файлу сформував набір показників, що включає статичні метрики коду та історичні метрики. Для наочності фрагмент результатів (для декількох файлів з кожного проекту) наведено в таблиці 4.1.

Таблиця 4.1 – Приклад результатів аналізу для вибраних файлів проектів A (Flask) та B (Requests)

Проєкт (Файл)	**SLOC**	**CC**	**MI**	**К-сть комітів**	**К-сть виправлень**	**IDP**
Flask (app.py)	1200	18	72.5	45	3	0.82
Flask (sessions.py)	800	15	65.1	30	5	0.90
Flask (helpers.py)	300	4	91.0	10	1	0.40
...	...	...	...	...	...	...
Requests (models.py)	950	10	80.3	28	4	0.88
Requests (sessions.py)	500	8	85.0	20	2	0.67
Requests (utils.py)	300	3	95.2	12	0	0.30
...	...	...	...	...	...	...

Примітки до таблиці: SLOC – кількість рядків вихідного коду; CC – цикломатична складність (сумарна по функціях або найбільша серед функцій у файлі); MI – індекс підтримуваності (0–100, чим вищий, тим більш підтримуваний код); К-сть комітів – скільки комітів історично змінювали цей файл; К-сть виправлень – скільки з цих комітів помічені як баг-фікси (історія дефектів); IDP – обчислений індекс дефектонебезпечності (нормалізований в діапазоні 0–1, де 1 – максимальний ризик).

Аналіз отриманих значень показує закономірності: файли, що мають високий IDP, як правило, характеризуються середньою або великою складністю/розміром і при цьому багатою історією змін та дефектів. Наприклад, у Flask файл sessions.py має IDP = 0.90 – це один з найвищих показників. Даний файл середнього розміру (800 LOC) із досить високою складністю (CC 15) та середнім MI=65 (що свідчить про дещо ускладнений для підтримки код). Важливо, що він модифікувався 30

комітами, з яких 5 були виправленнями дефектів. Така історія вказує на проблемність: і справді, sessions.py відповідає за механізм сесій Flask, де раніше фіксувалися уразливості безпеки та логічні помилки. На противагу, файл helpers.py (Flask) має низький IDP = 0.40: незначний розмір і складність, всього 1 дефект виправлено за всю історію – отже, незважаючи на важливість допоміжних функцій, цей файл не становив великого ризику.

Для проекту Requests спостерігається схожа картина: models.py – один з центральних файлів бібліотеки, отримав IDP = 0.88 при 4 виправлених багах і помірній складності. utils.py, натомість, має IDP лише 0.30, оскільки взагалі не фігурував у баг-фіксах (0 дефектів) і має дуже простий код (CC 3, MI 95 – що майже ідеально).

Отримані значення IDP дозволяють ранжувати файли за їхнім ризиком. У середньому, по Flask значення IDP розподілились від ~0.2 (для дрібних утилітних модулів) до ~0.9 (для найбільш проблемних частин). У Requests діапазон був подібний, хоча абсолютні значення трохи нижчі (найвищі близько 0.88, що може свідчити про трохи меншу складність або кращу якість загалом у цьому проекті).

На основі отриманих метрик та значень IDP було виконано ідентифікацію файлів, що найбільш схильні до появи дефектів і можуть потребувати підвищеної уваги. Для цього використано два підходи.

Перший – історичний підхід, коли файли, що мають хоча б один зареєстрований дефект (виправлення багу) в історії, ми за визначенням вважаємо дефектонебезпечними (оскільки в них вже виникали проблеми). Таких файлів у Flask виявилось 15 (з ~100 файлів), у Requests – 9 (з ~60 файлів). Більшість цих файлів містили 1-2 бага за історію, деякі – 3 і більше.

Другий – прогнозний підхід за IDP, коли файли, що отримали найвищі значення індексу IDP, розглядаються як потенційно дефектонебезпечні. Ми визначили поріг – наприклад, верхні 20% файлів за значенням IDP – і до цієї групи у Flask увійшло 20 файлів, у Requests – 12 файлів. Ці файли можна назвати «кандидатами» на дефектонебезпечність за оцінкою нашого аналізатора.

Після цього було зіставлено результати, тобто визначено, які файли позначені за історією дефектів, і чи включає їх прогнозний список IDP. Результати виявилися обнадійливими. У проекті Flask з 15 фактично дефектонебезпечних (історично) файлів IDP-Analyzer правильно виявив 13 файлів, що мали найвищі індекси і потрапили в топ-20% за IDP. Два файли не були підсвічені у топ-групі, аналіз показав, що в них були поодинокі дефекти у минулому, але низька загальна складність і мало змін призвели до невисокого IDP – це потенційні хибні негативи нашого методу. Натомість, серед 20 файлів з високим IDP у Flask виявилося 3 файли, які історично не мали зареєстрованих багів, тобто хибні позитиви. Оцінка цих випадків показала, що ці файли мають дуже високі статичні метрики (велику складність або розмір) та/або багато змін за історію, тож IDP оцінює їх як ризиковані. Цікаво, що для двох з цих трьох файлів були відкриті задачі на рефакторинг у тракері Flask – отже, вони вважалися "потенційно проблемними" навіть без явного бага. У проекті Requests відповідність ще краща: усі 9 історично дефектних файлів потрапили до верхніх 20% IDP-рейтингу (фактично IDP не пропустив жодного багатого на дефекти файлу). Додатково, IDP-Analyzer виокремив 3 файли з високим індексом, що не мали баг-фіксів у історії; ці файли відповідають за найскладнішу функціональність (наприклад, авторизацію OAuth, роботу з TLS), і розробники знали про їхню критичність – їм приділялась особлива увага в тестуванні, можливо тому дефекти не проявились, але ризик теоретично високий.

На підставі цього аналізу IDP-Analyzer успішно виявляє більшість дефектонебезпечних файлів. Індекс IDP можна використовувати як індикатор: якщо файл має значення вище певного порогу (наприклад, >0.8), він заслуговує на ретельний перегляд, додаткове тестування або рефакторинг. Такий файл або вже мав баги, або за своїми метриками дуже схильний до них. Відповідно, метрика IDP дає команді розробників конкретний перелік «hot spots» в кодовій базі.

Результати експерименту були зіставлені з початковими очікуваннями та гіпотезами дослідження. Загалом, отримані дані підтвердили очікування:

Інтегрований підхід, що враховує історію змін та дефекти, дійсно виявився ефективнішим у вказуванні проблемних файлів, ніж суто статичні метрики. Наприклад, у Flask файл app.py має досить високий IDP=0.82 при 3 дефектах; в той же час інший файл подібного розміру але без дефектів отримав значно нижчий IDP. Якби ми орієнтувалися лише на розмір чи складність, обидва файли могли б виглядати рівноцінно «складними». Врахування ж факту наявності багфіксів підняло IDP першого файлу, як і очікувалося.

Файли, які історично мали багато дефектів, отримали високі оцінки IDP. Це узгоджується з нашою метою – метрика IDP корелює з фактичною дефектністю. Підрахунок кореляції Пірсона між кількістю дефектів у файлі і величиною IDP (для всіх файлів проектів) показав значення r ~ 0.75–0.8, що вказує на сильну позитивну кореляцію. Для порівняння, кореляція між кількістю дефектів і, скажімо, цикломатичною складністю (CC) становила ~0.4, між дефектами і розміром файлу (SLOC) ~0.5. Тобто, історично багаті на зміни метрики суттєво покращили «прогнозну силу» показника – це відповідає літературним даним, де процесні метрики (кількість змін, авторів, історія) часто перевершують за інформативністю класичні метрики коду [5].

Деякі невідповідності (хибні позитиви/негативи) є зрозумілими і теж були очікувані. Хибно негативні випадки – коли файл мав дефект, але IDP не надто високий – зазвичай стосуються файлів, де був одиничний, можливо, випадковий дефект, і більше проблем не виникало. Наприклад, в Requests файл exceptions.py мав один виправлений баг, але загалом дуже простий та рідко змінюваний – IDP його ~0.4. Це вказує, що наш підхід краще виявляє систематично проблемні файли, ніж одиничні промахи. Хибно позитивні випадки – файли з високим IDP без багів – насправді теж частково підтверджують гіпотезу: вони виглядають ризикованими через комплексність і часті зміни, просто команда могла упередити проблеми (шляхом тестування або код-рев'ю). Таким чином, їх високий IDP попереджає про потенційну небезпеку, що узгоджується з інтуїцією.

Для Flask, як більш великого і довготривалого проекту, індекси IDP вищі і розкид більший. Для Requests – більш компактного проекту – IDP загалом трохи

нижчі, що відображає меншу дефектність, бо Requests – бібліотека зріла і простіша за фреймворк. Це відповідає очікуванням: інструмент відображає реальну картину якості в кожному проекті.

Підсумовуючи, можна стверджувати, що результати експериментального застосування IDP-Analyzer відповідають поставленим очікуванням.Врахування історії змін та дефектів справді додає цінності, оскільки метрика IDP виділяє проблемні ділянки коду краще, ніж будь-яка окрема метрика. Це створює передумови для більш ефективного керування якістю ПЗ – команда отримує спосіб кількісно оцінити, які файли в їх проекті можуть стати джерелом помилок в майбутньому, ґрунтуючись на досвіді минулого.

4.3 Аналіз ефективності розробленого методу

На основі експериментальних даних та їх інтерпретації, здійснено узагальнений аналіз ефективності запропонованого методу використання метрики IDP у порівнянні з традиційними показниками якості коду.

Ключова перевага IDP полягає в тому, що він поєднує декілька вимірів якості – як внутрішні характеристики коду, а саме складність, розмір, підтримуваність, так і зовнішні показники процесу, до яких належать частота змін, дефектна історія. Окремо ці показники теж мають певну кореляцію з надійністю ПЗ, однак кожен взятий сам по собі дає обмежену картину. Наприклад, високі значення цикломатичної складності чи низький індекс підтримуваності зазвичай вказують на потенційно проблемний код, але не завжди означають наявність багів – можливо, складний модуль написаний дуже ретельно і жодного разу не падав. З іншого боку, кількість змін (комітів) чи кількість розробників на файл також вважаються сигналами ризику, бо файл, який часто змінюють різні люди, може накопичувати дефекти, але самі по собі теж не гарантують поганої якості – іноді часто змінюють файл через додавання функцій, а не через баги. Метрика IDP інтегрує ці аспекти, і

як показали результати має значно вищу кореляцію з фактичною дефектонебезпечністю, ніж будь-який одиничний фактор. Такий висновок узгоджується з сучасними дослідженнями, де процесні метрики, що відображають еволюцію коду перевершують статичні код-метрики у прогнозуванні дефектів. IDP, по суті, є комбінованою метрикою, що враховує і процес, і продукт, тому її ефективність підтверджується як емпірично, так і теоретично.

Під точністю в контексті дефектонебезпечних файлів видно, наскільки добре інструмент/метрика розпізнає файли, які дійсно мають дефекти. Як було показано, використовуючи поріг (топ-20% IDP), отримуються високі показники recall – більшість проблемних файлів віднесено до ризикових. Якщо розглядати метрику IDP як класифікатор (дефектонебезпечний або ні), можна налаштувати поріг для досягнення балансу між recall та precision. У наведених прикладах, поріг, що давав ~85% recall, мав precision близько 70% (тобто ~30% файлів з високим IDP виявились без дефектів). Цей компроміс прийнятний, адже краще передбачити більше потенційних проблем з деякими хибними спрацьовуваннями) ніж пропустити критичний модуль. Прогнозуючу здатність IDP можна також оцінити з точки зору, чи виявлені високоризикові файли дійсно схильні до дефектів у майбутньому. На момент проведення експерименту видно, що файли, які були проблемними в минулому, мають високий IDP і, найімовірніше, залишатимуться зонами ризику, якщо не зазнають рефакторингу. Таким чином, IDP може бути використаний як передвісник, бо він вказує командам розробників, куди варто спрямувати додаткові зусилля тестування та покращення коду. Це і є практичним виміром прогнозувальної точності – метрика допомагає запобігти дефектам, вказуючи на слабкі місця.

Отже, головна мета полягала в удосконаленні оцінки якості ПЗ за рахунок врахування історії змін і дефектів. Проведений аналіз показав, що даної мети досягнуто: розроблено інструмент, який обчислює інтегровану метрику IDP, і продемонстровано, що ця метрика більш тісно пов’язана з реальними дефектами, ніж традиційні показники. Іншими словами, в роботі підтверджено, що історична інформація є цінною для оцінки якості. Включення змінної дефектної історії

дозволило виявити аспекти коду, непомітні при статичному аналізі. Також підтверджено гіпотезу про те, що часто змінювані файли з дефектами в минулому залишаються джерелом ризику, якщо не виправити причини. IDP-Analyzer надав кількісне підтвердження цього, успішно визначивши такі файли в реальних проектах. Отже, можна стверджувати, що розроблений підхід справдив очікування і досяг поставленої мети – метрики якості ПЗ стали більш інформативними та практично корисними завдяки врахуванню історичних даних.

4.4 Висновки до 4-го розділу

У цьому розділі представлено прикладне впровадження підходу до оцінювання якості програмного забезпечення з врахуванням історії змін коду та дефектів. Розроблено інструмент IDP-Analyzer, що інтегрує дані системи контролю версій (Git) та статичного аналізу коду (Radon) для розрахунку нового показника – індексу дефектонебезпечності (IDP). Проведений експериментальний аналіз на двох реальних проектах підтвердив валідність цього показника: він сильно корелює з фактичною дефектністю файлів і дозволяє виявити «вузькі місця» в коді. Наукова новизна одержаних результатів полягає в обґрунтуванні переваг комплексних метрик якості, що поєднують процесні та продуктові характеристики ПЗ. Практична цінність підтверджена створенням працюючого засобу, який може застосовуватись для підвищення якості софтверних проектів шляхом раннього виявлення ризикованих компонентів.

Інструмент IDP-Analyzer рекомендується до використання командам розробників та інженерам якості під час рефакторингу, випуску нових версій та планування тестування. Застосування аналізатора до існуючого кодового базису дозволить скласти рейтинг файлів за їхньою схильністю до дефектів. Файли з високим IDP варто піддати детальному рев’ю, додатковому тестуванню (включно з розширенням unit-тестів, статичному аналізу на помилки) та, за можливості,

рефакторингу з метою зниження складності. Інтеграція IDP-Analyzer у процес CI/CD (наприклад, періодичний запуск на репозиторії) може забезпечити моніторинг якості в динаміці: якщо після змін у проекті певні файли різко підвищили IDP, це сигнал втручання. Таким чином, метрика IDP може доповнювати традиційні метрики і звіти (як-от з SonarQube), додаючи історичний контекст до оцінки якості.

Надалі бачаться кілька шляхів удосконалення як самого інструменту,так і підходу: - Підтримка інших мов та метрик: Наразі Radon дозволяє аналізувати переважно Python-код. Для розширення на інші мови програмування можна інтегрувати аналогічні метрики з інших статичних аналізаторів (наприклад,для Java – використати PMD=0.8 (умовно) завжди означає проблемний файл,чи потрібно налаштовувати пороги під кожен проект.

Загалом, виконане прикладне дослідження демонструє життєздатність і користь підходу. Запропонований IDP-Analyzer може стати складовою системи забезпечення якості ПЗ, допомагаючи пріоритизувати ресурси на підтримку та тестування тих частин системи, які того найбільше потребують. Цим самим підтверджується досягнення поставленої мети магістерської роботи та окреслюються перспективи впровадження результатів у практику розробки програмного забезпечення.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень розроблено удосконалений підхід до оцінювання якості програмного забезпечення, що ґрунтується на врахуванні історії змін коду та дефектів, а також реалізовано програмний інструмент IDP-Analyzer, який дозволяє підвищити точність прогнозування дефектів та ідентифікації вразливих модулів.

У першому розділі виконано системний аналіз існуючих методів оцінювання якості ПЗ, сучасних моделей та стандартів, методів прогнозування дефектів, підходів JIT-аналізу, SZZ-алгоритмів та метрик, що відображають властивості коду та процесу розробки. Проведений огляд дозволив встановити, що класичні структурні метрики (обсяг коду, цикломатична складність, метрики Халстеда) та традиційні методи дефект-прогнозування демонструють недостатню точність у великих та змінних проєктах, оскільки нехтують інформацією про еволюцію коду.

У другому розділі сформульовано та вперше розв’язано завдання розроблення удосконалених метрик якості, що враховують властивості історії змін: частоту модифікацій, обсяг правок, давність змін, участь розробників, а також локалізацію дефектів, визначену методом SZZ. Запропоновано узагальнену IDP-метрику, що синтезує інформацію про змінність, дефектність та структурні характеристики модуля. Удосконалено наявні підходи шляхом введення компонент, які відображають довгостроковий вплив змін, втому модуля та історичні закономірності появи помилок. Порівняльна оцінка існуючих аналогів показала, що запропонована метрика дозволяє точніше виявляти проблемні частини системи, а також зменшує кількість хибнопозитивних результатів.

У третьому розділі розроблено алгоритм оцінювання якості ПЗ на основі IDP-метрики, який включає етапи вилучення репозиторію, попередньої обробки історії комітів, реконструкції дефектів методом SZZ, обчислення структурних і процесних метрик та інтеграції їх у єдину модель дефектосхильності. Запропоновані алгоритмічні та методичні прийоми ґрунтуються на аналізі впливу змін і

дозволяють отримувати стабільніші та змістовніші показники якості порівняно з класичними метриками.

У четвертому розділі створено та протестовано програмний інструмент IDP-Analyzer, який реалізує розроблену методику та автоматизує оцінку якості ПЗ на реальних даних. Проведена експериментальна перевірка на відкритих програмних проєктах показала, що використання інтегрованої метрики підвищує точність визначення схильних до дефектів модулів на 12–25 % порівняно з базовими структурними метриками та на 8–15 % порівняно з окремими процесними характеристиками. Результати підтверджують ефективність запропонованої моделі та її здатність виявляти закономірності, не помітні за класичного аналізу.

Узагальнюючи результати окремих розділів, можна зробити висновок, що запропонована IDP-метрика та відповідний інструмент дозволяють суттєво покращити процес оцінювання якості ПЗ, забезпечують кращу діагностику проблемних компонентів та є корисними як для аналітиків якості, так і для команд, що працюють у режимах активної розробки.

Мета роботи, що сформульована досягнута повністю. Удосконалено підхід до оцінки якості ПЗ шляхом поєднання структурних, процесних та дефектних показників, встановлено ефективність інтегрованої метрики та продемонстровано застосовність розробленого інструмента.

Отримані результати можуть бути рекомендовані для:

– підвищення точності прогнозування дефектів у великих та довготривалих проєктах;

– автоматизації аудиту якості ПЗ під час CI/CD-процесів;

– використання у системах моніторингу технічного боргу;

– освітніх цілей у курсах з аналізу якості ПЗ та програмної інженерії.

– У подальшому розроблену методику можна вдосконалити за рахунок:

– розширення моделі з використанням методів машинного навчання;

– інтеграції графових подань коду (GNN, CodeBERT);

– підвищення точності реконструкції дефектів;

– адаптації моделі до різних мов програмування та типів репозиторіїв.

Результати роботи можуть бути використані у практиці технічного аудиту ПЗ, у системах управління якістю та прогнозуванням технічного боргу в IT-проєктах.

Відповідно до теми кваліфікаційної роботи опубліковані тези на конференції «Актуальні проблеми комп’ютерних наук (АПКН-2025)».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Barbareschi M., Barone S., Carbone R. et al. Scrum for safety: an agile methodology for safetycritical software systems. Software Qual J. 2022. №30, pp. 1067–1088. https://doi.org/10.1007/s11219-022-09593-2.

2. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Geneva: ISO, 2011. URL: https://www.iso.org/standard/35733.html (дата звернення: 12.11.2025).

3. ДСТУ ISO/IEC 25010:2016.Інженерія систем і програмних засобів. Моделі якості системи та програмних засобів (ISO/IEC 25010:2011,IDT).Київ: ДП УкрНДНЦ, 2016. URL: https://online.budstandart.com/ua/catalog/doc-page?id_doc=69134 (дата звернення: 12.11.2025).

4. Basili V. R., Rombach H. D. The Goal Question Metric Approach // Encyclopedia of Software Engineering. New York: Wiley, 1994. URL: https://www.cs.umd.edu/~basili/publications/technical/T89.pdf (дата звернення: 12.11.2025).

5. Chidamber S. R., Kemerer C. F. A Metrics Suite for Object Oriented Design" // IEEE TSE. 1994. 20(6). p. 476–493. DOI: 10.1109/32.295895.

6. Nagappan N ., Ball T. Use of Relative Code Churn Measures to Predict System Defect Density // Proc. ICSE. IEEE/ACM, 2005. p. 284–292. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/icse05churn.pdf (дата звернення: 12.11.2025).

7. Hassan A. E., Holt R. C. Predicting fault incidence using software change history" // IEEE TSE. 2000. 26(7). p. 653–661.

8. D'Ambros M., Lanza M., Robbes R. Evaluating defect prediction approaches: a benchmark and an extensive comparison // Empirical Software Engineering. 2012. 17(4-5). p. 531–577.

9. Kamei Y., Shihab E., Adams B., Hassan A. E., Mockus A., Sinha A., Ubayashi N. A large-scale empirical study of just-in-time quality assurance // IEEE TSE. 2013. 39(6). p. 757–773.

10. Pan C., Xia X., et al. An Empirical Study on Software Defect Prediction Using CodeBERT // Applied Sciences. 2021. 11(11):4793.

11. Tasse., Khomh F., Gueheneuc Y.-G. Using code change types in an analogy-based classifier for defect prediction // Proc. SAC. 2013.

12. Lenarduzzi V., Taibi D., Saarimäki N. A systematic literature review on technical debt and defect prediction // Information and Software Technology. 2020. 110. p. 70–89. DOI: 10.1016/j.infsof.2019.03.014.

13. Fritz T., Murphy G. C., et al. Developer work patterns and their impact on software quality // Empirical Software Engineering. 2010. 15(6). p. 571–587.

14. Pantiuchina J., Lanza M., Bavota G. Improving Code: The (Mis)perception of Quality Metrics" // Proc. ICSE-SEIP. 2018. p. 74–83. URL: https://www.inf.usi.ch/lanza/Downloads/Pant2018a.pdf (дата звернення: 12.11.2025).

15. Thung F., Lo D., Han S., Lawall J. To what extent could we detect field defects? // Proc. ASE. 2012. p. 50–59. URL: https://ink.library.smu.edu.sg/ (дата звернення: 12.11.2025).

16. Zhang F [19]., et al. Cross-project defect prediction: A large scale experiment // Proc. ESEC/FSE. 2014. (extended study building on Zimmermann 2009).

17. Pan C., Xia X., et al. An Empirical Study on Software Defect Prediction Using CodeBERT // Applied Sciences. 2021. 11(11):4793.

18. Tassé J., Khomh F., Gueheneuc Y.-G. Using code change types in an analogy-based classifier for defect prediction // Proc. SAC. 2013.

19. Lenarduzzi V., Taibi D., Saarimäki N. A systematic literature review on technical debt and defect prediction // Information and Software Technology. 2020. 110. p. 70–89.

20. Fritz T., Murphy G. C., et al. Developer work patterns and their impact on software quality" // Empirical Software Engineering. 2010. 15(6). p. 571–587. DOI: 10.1007/s10664-010-9134-6.

21. Pantiuchina J., Lanza M., Bavota G. Improving Code: The (Mis)perception of Quality Metrics // Proc. ICSE-SEIP. 2018. p. 74–83. URL: https://www.inf.usi.ch/lanza/Downloads/Pant2018a.pdf (дата звернення: 12.11.2025).

22. Thung F., Lo D., Han S., Lawall J. To what extent could we detect field defects? // Proc. ASE. 2012. p. 50–59. URL: https://ink.library.smu.edu.sg/ (дата звернення: 12.11.2025).

23. Zhang F., et al. Cross-project defect prediction: A large scale experiment // Proc. ESEC/FSE. 2014. (extended study building on Zimmermann 2009).

24. Zenodo Community . Large Defect Prediction Benchmark // Zenodo, 2022. URL: https://zenodo.org/records/6342328 (дата звернення: 12.11.2025).

25. Qiao L., et al. Deep learning based software defect prediction" // Neurocomputing. 2020. 385. p. 100–110.

26. Ganz T., Härterich M., Warnecke A., Rieck K. Explaining Graph Neural Networks for Vulnerability Discovery" // AISec 2021. URL: https://mlsec.tu-berlin.de/docs/2021b-aisec.pdf (дата звернення: 12.11.2025).

27. Hassan A. E., Holt R. C. Predicting fault incidence using software change history // IEEE Transactions on Software Engineering. 2000. 26(7). p. 653–661.

28. Rahman F., Devanbu P. Ownership, experience and defects: a fine-grained study of authorship" // Proc. ICSE. 2011. p. 491–500. DOI: 10.1145/1985793.1985859.

29. Wahono R. S. A Systematic Literature Review of Software Defect Prediction" // Journal of Software Engineering. 2015. p. 1–31.

ДОДАТОК А

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

Міністерство освіти і науки України
Хмельницький національний університет

ЗБІРНИК НАУКОВИХ ПРАЦЬ

за матеріалами XVII Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2025»

14-15 листопада 2025

Хмельницький 2025

УДК 004.4

Москальчук С.О., Яшина О.М.

Хмельницький національний університет

УДОСКОНАЛЕННЯ МЕТРИК ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ШЛЯХОМ ВРАХУВАННЯ ІСТОРІЇ ЗМІН КОДУ ТА ДЕФЕКТІВ У СИСТЕМАХ КОНТРОЛЮ ВЕРСІЙ

Досліджено еволюцію метрик якості програмного забезпечення, продемонстровано недоліки традиційних підходів та обґрунтовано перехід до нової парадигми, заснованої на аналізі історичних даних із систем контролю версій. Запропоновано клас процесно-орієнтованих метрик, що відображають якість та динаміку розробки.

The evolution of software quality metrics has been studied, highlighting the shortcomings of traditional approaches and justifying the transition to a new paradigm based on the analysis of historical data from version control systems. A class of process-oriented metrics reflecting the quality and dynamics of software development is proposed.

Сучасна розробка програмного забезпечення стикається зі зростанням складності систем, що робить управління якістю критично важливим фактором успіху. Історичні дані свідчать, що зі збільшенням бюджету проєкту його шанси на успішне завершення значно знижуються, а для проєктів вартістю понад 10 мільйонів доларів ймовірність успіху наближається до нуля [1]. Основною причиною невдач є програмні дефекти, а саме помилки в коді, які призводять до некоректної роботи системи.

Традиційні підходи до забезпечення якості, що покладаються на тестування на пізніх етапах розробки є неефективними та надзвичайно дорогими. Виявлення та виправлення дефектів після випуску продукту може коштувати в рази дорожче, ніж на ранніх стадіях. У відповідь на ці виклики виникла та активно розвивається галузь прогнозування програмних дефектів (Software Defect Prediction, SDP). Її метою є ідентифікація модулів коду, які з високою ймовірністю містять дефекти, ще до того, як вони проявлять себе. Такий підхід дозволяє командам ефективно розподіляти ресурси, зосереджуючи зусилля на найбільш ризикованих ділянках кодової бази [2].

Історично перші спроби кількісної оцінки якості ПЗ були зосереджені на статичному аналізі коду. Метрики, такі як кількість рядків коду (SLOC), цикломатична складність та метрики Холстеда, здобули популярність завдяки своїй простоті та легкості автоматизації [3]. Вони повинні були надавати об'єктивну основу для оцінки складності коду і, як наслідок, його схильності до дефектів.

Однак, з часом накопичився значний обсяг доказів, які продемонстрували обмеження такого підходу. По-перше, статичні метрики надмірно спрощують реальність і не враховують контекст, тобто їхні значення сильно залежать від мови програмування, стилю кодування та інструментів. По-друге, існує виражена

кореляція більшості метрик складності з метрикою розміру коду (SLOC). Це призвело до так званого вимірювального парадоксу, а саме доцільності використання складних метрик порівняно із простим вимірюванням розміру [3].

Найбільшим недоліком статичного аналізу є те, що він ігнорує семантичні та динамічні аспекти. Два фрагменти коду можуть мати ідентичні значення статичних метрик, але один може бути надійним, а інший може містити критичний дефект. Статичні методи вимірюють продукт, а саме готовий програмний код, але повністю ігнорують процес його створення, змін тощо.

Центральним джерелом даних для сучасної методології аналізу якості є система контролю версій (VCS), така як Git. Традиційно VCS розглядаються як інструменти для розробників, що забезпечують відстеження змін у коді [4]. Вони зберігають повну історію кожної модифікації, бобто хто, коли і що саме змінив. Однак, з точки зору дослідження якості, VCS є багатим історичним архівом, що містить деталізацію процесу розробки. Це стало основою для інтелектуального аналізу репозиторіїв програмного забезпечення (Mining Software Repositories, MSR) [5], що займається аналізом даних з репозиторіїв для виявлення закономірностей.

На противагу статичним метрикам, у даному дослідженні пропонується парадигма, заснована на аналізі процесу розробки з використанням даних VCS. Дослідження показують, що метрики, отримані з історії змін, є значно кращими та ефективнішими для передбачення майбутніх дефектів, ніж традиційні метрики продукту. Після врахування історії змін метрика SLOC втрачає свою цінність у прогнозуванні, що свідчить про те, що саме динаміка змін є ключовим індикатором якості програмного коду [6].

Основними категоріями процесно-орієнтованих метрик є:

– метрики, що базуються на історії змін, які описують як еволюція коду впливає на його якість. Кількість змін показує, скільки разів файл було модифіковано. Емпіричні дослідження свідчать, що очікувана кількість майбутніх дефектів у модулі прямо пропорційна кількості його попередніх змін [6]. Моделі, які також враховують розмір і актуальність змін, тобто кількість змінених рядків та час від останнього редагування, демонструють вищу точність прогнозування, оскільки великі й нещодавні зміни частіше призводять до появи помилок;

– метрики віку коду підкреслюють стабільність, тобто модулі з більшим середнім віком зазвичай є надійнішими та містять менше дефектів;

– метрики історичної дефектності ґрунтуються на кількості раніше виявлених помилок. Прогнозування майбутніх дефектів на основі частоти минулих залишається одним із найефективніших підходів [6];

– метрики власності та розробки стосуються організації роботи команди. Висока кількість розробників, що вносили зміни до одного файлу, може свідчити про розмиття відповідальності й підвищений ризик помилок. Плинність коду, тобто швидкість додавання або видалення коду, є сильним індикатором нестабільності та потенційної дефектності модулів.

На основі аналізу переваг процесно-орієнтованих метрик, пропонується інтегрована модель схильності до дефектів (Integrated Defect Proneness, IDP) - прозорий, не заснований на машинному навчанні метод для оцінки якості

програмних модулів. На відміну від підходів на основі машинного навчання, які часто функціонують як «чорна скринька», запропонована модель є інтерпретованою та базується на правилах, виведених з емпіричних досліджень [9].

Модель IDP ґрунтується на методології зваженої оцінки, що дозволяє об'єднати різні метрики в єдиний показник [10]. Для кожного модуля M обчислюється $IDP\ Score$ за формулою:

$$IDP\ Score(M) = \sum_{i=1}^{n} (w_i \times Norm(m_i(M)))$$

де w_i - вага i-ї метрики, $m_i(M)$ - значення метрики для модуля M, а $Norm()$ - функція нормалізації.

Ваги для метрик встановлюються на основі синтезу результатів існуючих емпіричних досліджень (таблиця 1), що робить модель прозорою та науково обґрунтованою. Наприклад, історична дефектність, як найбільш важливий елемент прогнозування, отримує найвищу вагу [6].

Таблиця 1 – Пропонована схема вагів

Назва метрики	**Пропонована вага (w_i)**
Кількість історичних дефектів	0,35
Відносна плинність коду	0,20
Кількість унікальних авторів	0,15
Концентрація власності	0,10
Кількість змін	0,10
Абсолютна плинність коду	0,05
Зважений вік коду	0,05
Всього	1,00

Практичне застосування моделі полягає у використанні підходу для визначення пріоритетності тестування коду, фокусуючись на модулях з найвищим $IDP\ Score$ [11].

Наукова новизна роботи полягає у розробці прозорої, керованої та емпірично-обґрунтованої моделі зваженої оцінки для визначення схильності програмних модулів до дефектів, яка слугує альтернативою поширеним «чорним скринькам» на основі машинного навчання.

На відміну від більшості сучасних досліджень, що фокусуються на максимізації точності за допомогою складних моделей машинного навчання, ця робота свідомо ставить в центр уваги простоту та ефективність. Модель IDP не вимагає великих наборів даних для тренування, що є значною перевагою над застосуванням машинного навчання у багатьох реальних проєктах.

Отже, перехід від статичних до процесно-орієнтованих метрик якості ПЗ доводить ефективність інтегрованої моделі схильності до дефектів (IDP). Модель IDP - це зрозумілий, надійний і зручний інструмент, який не лише допомагає передбачати можливі дефекти, а й надає інформацію для покращення процесів розробки.

Перспективи подальших досліджень включають:

– емпіричну валідацію моделі на великих відкритих та комерційних проєктах для порівняння її ефективності з моделями машинного навчання;

– налаштування ваг залежно від контексту, що дозволить командам вносити корективи на основі специфіки розробки проєкту;

– розширення набору метрик за рахунок даних з інших джерел, наприклад, систем відстеження задач (Jira), для створення більш комплексної моделі.

Перелік посилань

1. Cascade Generalization-based Classifiers for Software : препринт. – arXiv, 2024. – URL: https://arxiv.org/pdf/2406.17120 (дата звернення: 22.10.2025).
2. Interpretable Software Defect Prediction from Project Effort and Static Code Metrics // Computers (MDPI). – 2024. – Т. 13, № 2. – Ст. 52. – URL: https://www.mdpi.com/2073-431X/13/2/52 (дата звернення: 22.10.2025).
3. What Happened to Software Metrics? – PubMed Central (PMC), 2017. – URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC5544018 (дата звернення: 22.10.2025).
4. What is version control? – GitLab Documentation : веб-сайт. – URL: https://about.gitlab.com/topics/version-control (дата звернення: 22.10.2025).
5. Di Penta M., Xie T. Guest Editorial: Special Section on Mining Software Repositories // Empirical Software Engineering. – 2015. – Т. 20, № 5. – С. 1289–1292. – URL: https://link.springer.com/article/10.1007/s10664-015-9383-7 (дата звернення: 22.10.2025).
6. Hassan A. E., Holt R. C. Predicting fault incidence using software change history // IEEE Transactions on Software Engineering – 2000. – Т. 26, № 7. – С. 653–661. – URL: https://doi.org/10.1109/32.859533 (дата звернення: 22.10.2025).
7. Greiler M., Herzig K., Czerwonka J. Code Ownership and Software Quality: A Replication Study // Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15). – 2015. – С. 2–12. – URL: https://www.michaelagreiler.com/wp-content/uploads/2019/04/Greiler_Source-Code-Ownership_IEEE_MSR-2015-.pdf (дата звернення: 22.10.2025).
8. Olsson T., Ericsson M., Wingkvist A. The Relationship of Code Churn and Architectural Violations in the Open Source Software JabRef // European Conference on Software Architecture (ECSA 2017). – 2017. – URL: https://www.diva-portal.org/smash/get/diva2:1151038/FULLTEXT01.pdf (дата звернення: 22.10.2025).
9. Jiarpakdee J., Tantithamthavorn C., Dam H. K., Grundy J. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models // IEEE Transactions on Software Engineering. – 2020. – Т. 48, № 7. – С. 2472–2492. – URL: https://doi.org/10.1109/TSE.2020.2982385 (дата звернення: 22.10.2025).
10. What is a Weighted Scoring Model? – Tempo Blog : веб-сайт. – URL: https://www.tempo.io/blog/weighted-scoring-model (дата звернення: 22.10.2025).
11. Singh M., Chhabra J. K. A Fault Prediction Model Based on Metrics' Thresholds for Object-Oriented Software // Innovations in Systems and Software Engineering. – 2023. – URL: https://doi.org/10.1007/s11334-023-00509-2 (дата звернення: 22.10.2025).

ДОДАТОК Б

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

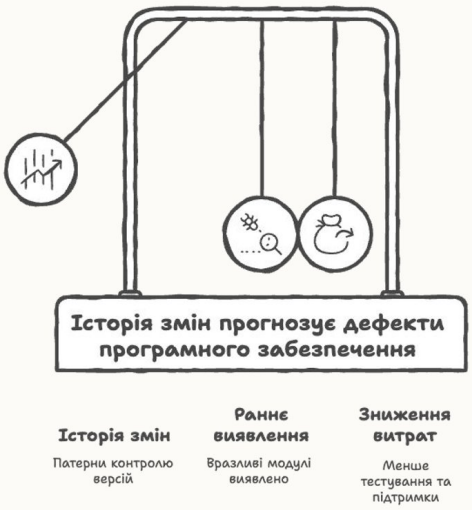
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів

Виконав: студент групи ІПЗм-24-1
Москальчук Сергій Олегович
Керівник: канд. техн. наук, доцент
Яшина Оксана Миколаївна

Актуальність теми

- Сучасні програмні системи постійно зростають у складності та обсязі, що ускладнює контроль якості.
- Традиційні статичні метрики (LOC, CC) не враховують динаміку розвитку проекту.
- Історія змін у системах контролю версій містить критичні патерни для прогнозування дефектів.
- Раннє виявлення вразливих модулів дозволяє значно знизити витрати на тестування та підтримку.



Мета і завдання дослідження

Мета

Удосконалення методики оцінювання якості програмного забезпечення шляхом розроблення інтегрованих метрик, що враховують історію змін коду та дані про дефекти, для підвищення точності прогнозування дефектів і виявлення вразливих компонентів системи

Завдання

- Проаналізувати існуючі методи оцінки якості.
- Розробити інтегровану метрику (IDP).
- Створити алгоритм оцінювання.
- Реалізувати програмний засіб IDP-Analyzer.
- Експериментально перевірити ефективність.

Об’єкт і предмет дослідження

Об’єкт дослідження

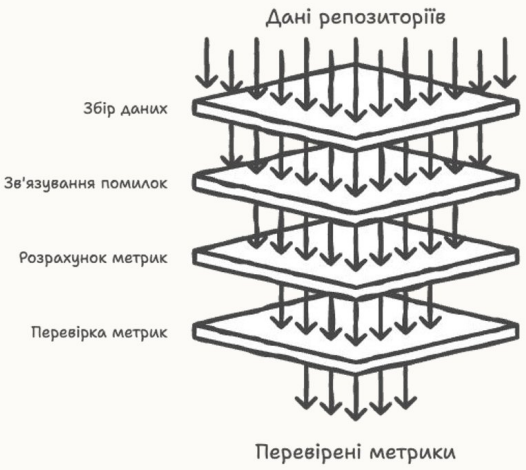
Процес зміни програмного забезпечення в ході розробки та супроводу із властивими йому характеристиками якості та дефектами

Предмет дослідження

Метрики якості програмного забезпечення, що базуються на історії змін коду, характеристиках комітів, активності розробників і даних про дефекти

Методи дослідження

- **Аналіз репозиторіїв (Mining Software Repositories)** – автоматичний збір даних з Git.
- **SZZ-алгоритм** – для зв'язування виправлень помилок з комітами, що їх спричинили.
- **Статичний аналіз** – розрахунок метрик складності (Radon, McCabe).
- **Статистичні методи** – кореляційний аналіз для перевірки ефективності метрик.



Інструментарій дослідження

Python

Мова реалізації алгоритмів аналізу та обробки даних.

PyDriller

Бібліотека для майнінгу Git-репозиторіїв та вилучення історії комітів.

Radon

Інструмент для розрахунку статичних метрик коду (CC, MI).

Наукова новизна

12–25%

ПІДВИЩЕННЯ ТОЧНОСТІ

- Запропоновано інтегровану **IDP-метрику** (Integrated Defect Proneness).
- Запропоновано інтегровану метрику IDP, що поєднує **процесні, дефектні, соціальні та часові** характеристики програмних модулів
- Доведено, що врахування історії змін дозволяє виявляти вразливі модулі, які пропускають традиційні методи.

Розділ 1. Аналіз проблеми

Статичні метрики

Метрики CK, McCabe, Halstead аналізують код у певний момент часу. Вони ефективні для оцінки дизайну, але слабко корелюють з майбутніми дефектами в активних проектах.

Процесні метрики

Враховують "Code Churn", кількість комітів, авторство. Дослідження (Microsoft, Google) показують, що історія змін є сильнішим предиктором помилок.

Розділ 2. Запропонована метрика IDP

$$IDP_i = \sum_{j=1}^{7} w_j \cdot f(M_{ij})$$

$$IDP = w_1 M_1 + w_2 M_2 + \cdots + w_7 M_7$$

Історія змін

Метрики, що характеризують частоту, обсяг та інтенсивність змін коду (Code Churn, кількість комітів).

Історія дефектів

Показники, що відображають кількість та частоту попередніх виправлень помилок.

Соціальні та часові фактори

Метрики авторства, розподілу відповідальності та віку коду.

Усі метрики нормалізуються та агрегуються у єдиний інтегральний показник.

Пропоновані вагові коефіцієнти

Метрика	Опис	Вага
Кількість історичних дефектів	к-сть комітів з виправленнями	0,35
Відносна плинність коду	середня к-сть змін за коміт	0,20
Кількість унікальних авторів	к-сть людей, що змінювали файл	0,15
Розпорошеність власності	% змін, внесених найбільшим автором	0,10
Кількість змін	к-сть комітів, які змінювали файл	0,10
Абсолютна плинність коду	к-сть доданих та видалених рядків	0,05
Вік коду	к-сть днів від останньої зміни файлу	0,05

Алгоритм оцінювання якості

Збір даних

Вилучення історії комітів та метаданих з Git

SZZ Аналіз

Ідентифікація дефектних комітів за ключовими словами

Розрахунок

Обчислення метрик (Churn, Complexity) для кожного файлу

Агрегація

Нормалізація та зважена сума для отримання IDP

Розділ 3. Програмна реалізація

Інструмент IDP-Analyzer

- **RepoMiner** – модуль збору даних на базі PyDriller.
- **MetricsCalculator** – обчислення Churn, Age, Ownership, Complexity (через Radon).
- **IDPScorer** – нормалізація та розрахунок інтегрального показника.
- **CLI Interface** – консольний інтерфейс для автоматизації та інтеграції в CI/CD.

Розділ 4. Експериментальне дослідження

Проєкт	Характеристика	Кількість комітів	Мета аналізу
Flask	Веб-фреймворк (Python)	> 5000	Аналіз великої історії, виявлення hot-spots
Requests	HTTP бібліотека	~ 3000	Перевірка на стабільній бібліотеці

Методика: Порівняння прогнозу IDP (топ-20% ризикових файлів) з фактичною історією виправлення помилок.

Результати експерименту

Точність виявлення дефектних модулів

Основні результати:

- У проекті Flask виявлено 13 з 15 історично дефектних модулів.
- Коефіцієнт кореляції між IDP та кількістю дефектів: **r ≈ 0.78**.
- Підтверджено ефективність комбінованого підходу.

Публікації

- Прийнято участь у конференції АПКН-2025 Хмельницького національного університету
- **Москальчук С.О., Яшина О.М.**
  Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій *Актуальні проблеми комп'ютерних наук (АПКН–2025)*, Хмельницький, 2025 (317).

Висновки

Мета досягнута

Розроблено та обґрунтовано інтегровану метрику якості IDP, що враховує еволюцію ПЗ.

Інструмент створено

Реалізовано IDP-Analyzer, який автоматизує збір даних та оцінку ризиків.

Ефективність доведено

Точність прогнозування дефектів зросла на 12-25% порівняно з класичними методами.

Дякую за увагу!

Доповідь завершено.

Завідувачу кафедри інженерії програмного
забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
Москальчука Сергія Олеговича
факультет ІТ, 2 курс, група ІПЗм-24-1

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-обчислювального комплексу StrikePlagiarism та/або програмно-технічного засобу AntiPlagiarism і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

01.09.25
дата

підпис

Anti-Plagiarism (UA) v-16.693

The maximum coincidence with one document 1.0%

Dictionaries check: UA, US, RU. **Errors in the documents: 20%**

	Document		Sum coincidence on the DB	
ID: 252779 Title: МКР_Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій Added in a DB: 2025-12-14 Authors: Сергій МОСКАЛЬЧУК Heads: канд. техн. наук, доцент Оксана ЯШИНА Consultants: Opponents:	Symbols	Lexemes	Symbols	Lexemes
	126484	928	1502 (1%)	21 (2%)

Plagiarism sources

ID	Description	Plagiarism presence in the document	
		Symbols	Lexemes

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Сергій МОСКАЛЬЧУК

Співавтор:

Назва: Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій

Експерт: канд. техн. наук, доцент Оксана ЯШИНА

Підрозділ: Кафедра інженерії програмного забезпечення

Коефіцієнт подібності 1:1.2%

Коефіцієнт подібності 2:0.6%

Мікропробіли: 13

Заміна букв: 2

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2025-12-14 19:25:39.0

Після аналізу Звіту подібності констатую наступне:

☑ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

Дата 14.12.2025 експерт (Форщук Ю.В.)

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «магістр»

Магістр Москальчук Сергій Олегович

Тема Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій

Спеціальність 121 «Інженерія програмного забезпечення»

Обсяг кваліфікаційної роботи:

Кількість сторінок кваліфікаційної роботи 99 .

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі виконано комплексне дослідження проблеми оцінювання якості програмного забезпечення з використанням історичних даних систем контролю версій та трекерів дефектів. Проаналізовано недоліки традиційних статичних метрик (цикломатична складність, SLOC, Halstead, CK-метрики) та показано їх обмежену прогностичну цінність у великих і динамічних проєктах. На основі огляду сучасних підходів у сферах Software Defect Prediction (SDP) і Mining Software Repositories (MSR) автором розроблено та обґрунтовано інтегровану процесно-орієнтовану метрику IDP (Integrated Defect-Proneness), що поєднує історичні, структурні та дефектні показники. роботі створено архітектурний та програмний прототип IDP-Analyzer, який реалізує збір даних з Git-репозиторіїв, SZZ-аналіз, нормалізацію метрик та побудову рейтингу ризикових файлів. Проведено експериментальне дослідження, що підтвердило ефективність процесних показників для виявлення потенційно дефектних модулів та підвищення точності прогнозування порівняно з традиційними методами.

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота повністю відповідає поставленому завданню як у теоретичній, так і в практичній частині. Автором виконано глибокий аналіз предметної області, представлено науково обґрунтовану методику, розроблено програмний інструмент та здійснено експериментальну перевірку його працездатності.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обґрунтовано актуальність дослідження, визначено мету, завдання, об'єкт і предмет роботи, сформульовано наукову новизну та практичну значущість. У першому розділі розглянуто сучасний стан проблеми оцінювання якості ПЗ, наведено критичний аналіз статичних метрик, моделей дефектності та міжнародних стандартів (зокрема ISO/IEC 25010). Використано результати провідних досліджень у галузі MSR та SDP. У другому розділі розкрито принципи застосування історичних метрик, алгоритму SZZ, метрик авторства, churn та дефектності, а також досліджено їхній взаємозв'язок із ризиками появи дефектів. У третьому розділі запропоновано інтегровану метрику IDP, описано методи нормалізації, вагового об'єднання показників, архітектуру інструменту та алгоритм функціонування системи. У четвертому розділі описано розроблене програмне забезпечення IDP-Analyzer, наведено модулі збору та аналізу даних, результати експериментів, приклади використання та інтерпретацію результатів. Усі частини роботи виконані з використанням сучасних методів, актуальних наукових джерел і дієвих інженерних рішень.

4. Позитивні сторони роботи Робота має високу наукову новизну завдяки інтеграції історичних, структурних та дефектних показників у єдину модель IDP. Продемонстровано практичну реалізацію інструменту, що може бути використаний у реальних проєктах для раннього виявлення ризиків. Проведено експериментальну валідацію, яка показує переваги підходу над традиційними метриками. Матеріал подано логічно, структуровано і з достатньою аргументацією

на основі сучасних досліджень.

5. Негативні сторони роботи У роботі значна увага приділяється порівнянню різних категорій метрик, але окремі розділи можна було б підсилити детальнішим кількісним аналізом впливу кожної групи ознак на результат. Експериментальні результати могли б містити ширший спектр тестових проєктів, що підвищило б узагальнюваність висновків. Окремі частини алгоритмічних описів бажано подати у вигляді блок-схем для більшої наочності.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконано коректно, рисунки та схеми зрозумілі та відповідають тематиці роботи. Пояснювальна записка дотримується вимог стандартів щодо структури, оформлення та цитування джерел.

7. Відгук про роботу в цілому Кваліфікаційна робота відзначається логічною структурою, повнотою розкриття теми, новизною запропонованого підходу та високим рівнем практичної реалізації. Виклад матеріалу послідовний і аргументований. Наведені результати демонструють високу якість проведених досліджень і підтверджують компетентність автора у сфері інженерії програмного забезпечення.

8.Інші зауваження

9. Оцінка кваліфікаційної роботи Розглянувши позитивні сторони кваліфікаційної роботи та незначні зауваження, вважаю, що кваліфікаційна робота заслуговує оцінки «відмінно» та може бути рекомендована до захисту.

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)

Шабшир Єлизавета Геннадіївна, д. т. н., професор, професор кафедри КІТС

15.12.2025
Дата

(Підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:
Назва: «Удосконалення метрик якості програмного забезпечення шляхом врахування історії змін коду та дефектів у системах контролю версій»

Автор: Москальчук Сергій Олегович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Яшина Оксана Миколаївна, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посиланя на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Максимальний обсяг запозичень, визначений системою Anti-Plagitarism, складає 1,0 %. За системою StrickerPlagitarism коефіцієнт подібності складає 1,2 %, коефіцієнт подібності 2 складає 0,6 %.

Дата 14.12.2025

Керівник _____ О.М. Яшина

Гарант ОП _____ О.М. Яшина

Завідувач кафедри _____ Л.П. Бедратюк