СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) Інформаційних технологій та електроніки

Кафедра Інформаційних технологій та програмування

ПОЯСНЮВАЛЬНА ЗАПИСКА
до кваліфікаційної випускної роботи

освітній ступінь бакалавр

спеціальність 121 «Інженерія програмного забезпечення»

на тему «Методи та програмні засоби підвищення якості розробки програмного забезпечення»

Виконав: студент групи ІПЗ-21д _____ М.А.Андрієць
(підпис) (ініціали і прізвище)

Керівник _____ В.О.Лифар
(підпис) (ініціали і прізвище)

Завідувач кафедри _____ О.І.Захожай
(підпис) (ініціали і прізвище)

Рецензент Ратов Д.В.

Київ 2025

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Навчально-науковий інститут (факультет) інформаційних технологій та електроніки
Кафедра інформаційних технологій та програмування
Освітній ступінь бакалавр
спеціальність 121 „Інженерія програмного забезпечення”
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Захожай О.І.
“____” __ 2025року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ ВИПУСКНУ РОБОТУ СТУДЕНТА

Андрієць Марія Андріївна
(прізвище, ім’я, по батькові)

1. Тема роботи Методи та програмні засоби підвищення якості розробки програмного забезпечення
Керівник роботи Лифар Володимир Олексійович, доц.,д.т.н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджений наказом університету від “27”травня 2025 року №67/15.15-С
2. Строк подання студентом роботи:14.06.2025 р.
3. Вихідні дані до роботи Об’єктом даної роботи є методи та програмні засоби для підвищення якості розробки програмного забезпечення
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Вступ. Теоретичні основи забезпечення якості програмного забезпечення. Основна частина:
Інструменти забезпечення якості ПЗ, практична демонстрація та аналіз результатів.Висновки.Перелік використаних джерел
5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслеників) :немає

Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

6. Дата видачі завдання: 01.04.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання кваліфікаційної випускної роботи	Строк виконання етапів	Примітка
1	Одержання завдання на виконання роботи	01.04.25	виконано
2	Укладення і погодження з керівником плану і етапів виконання роботи	08.04.25	виконано
3	Узагальнення даних літературних джерел, аналіз предметної галузі	17.04.25	виконано
4	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху	24.04.25	виконано
5	Укладання та тестування програмного продукту	20.05.25	виконано
6	Укладання, оформлення та погодження пояснювальної записки з керівником	30.05.25	виконано
7	Здача готової пояснювальної записки	11.06.25	виконано
8	Укладання доповіді і презентації	14.06.25	виконано

Студент _____ Андрієць М.А.
(підпис) (ініціали і прізвище)

Керівник роботи _____ Лифар В.О
(підпис) (ініціали і прізвище)

РЕФЕРАТ

У дипломній роботі досліджено методи та інструменти забезпечення якості програмного забезпечення як один із визначальних чинників підвищення ефективності життєвого циклу ПЗ. Актуальність теми зумовлена зростанням складності програмних продуктів і потребою в інструментах, які дають змогу виявляти дефекти ще на ранніх етапах розробки.

У теоретичній частині роботи розглянуто концептуальні основи забезпечення якості програмного забезпечення, класифікацію методів контролю якості, а також організаційно-функціональну структуру системи якості в межах програмного проєкту. Проведено систематизацію чинників, що впливають на якість програмного коду, та охарактеризовано основні завдання статичного й динамічного аналізу.

У другому розділі подано узагальнений огляд сучасних інструментів, які застосовуються для аналізу якості програмного коду. Проаналізовано принципи роботи та функціональність інструментів Pylint, SonarLint, SonarQube, PyTest, Coverage.py та ін. Особливу увагу приділено порівняльній характеристиці засобів контролю якості, що відображено у відповідній таблиці. Окремо розглянуто інструменти для контролю тестового покриття та методи їх інтеграції в локальні середовища розробки.

У практичній частині здійснено аналіз фрагмента програмного коду з типових помилками. Демонструється використання інструментів Pylint і SonarQube для виявлення порушень стилю, відсутності перевірок на None, дублювання логіки та інших дефектів. Проведено виправлення помилок і повторний аналіз коду, результати якого підтверджують доцільність застосування інструментів статичного аналізу на локальному рівні. Розроблено рекомендації щодо вибору та впровадження систем забезпечення якості у невеликих проєктах із обмеженою інфраструктурою.

ЗМІСТ

ВСТУП 6

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПЗ 8

1.1. Поняття якості програмного забезпечення 8

1.2. Основні дефекти в ПЗ та їх наслідки 14

1.3. Методи забезпечення якості (ручне тестування, рецензії коду, ...19

автоматичний аналіз).....19

ВИСНОВОК ДО РОЗДІЛУ 1 24

РОЗДІЛ 2 ОГЛЯД ІНСТРУМЕНТІВ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ 26

2.1. Статичний аналіз коду: принцип дії 26

2.2. Порівняльний огляд ESLint, SonarLint, Pylint 30

2.3. Інтеграція в середовище розробки (VS Code / PyCharm).....35

2.4. Приклади результатів (знімки екранів, порівняння логів аналізу).....41

ВИСНОВОК ДО РОЗДІЛУ 2 47

РОЗДІЛ 3 ПРАКТИЧНА ДЕМОНСТРАЦІЯ.....49

3.1. Вибір фрагмента коду з помилками 49

3.2. Аналіз результатів статичної перевірки 53

3.3. Інтерпретація результатів і обґрунтування доцільності застосованих 58

методів.....58

3.4. Формування рекомендацій щодо впровадження систем забезпечення.....62

якості 62

ВИСНОВОК ДО РОЗДІЛУ 3 67

ВИСНОВКИ.....69

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 71

ДОДАТКИ.....79

ВСТУП

Якість програмного забезпечення відіграє ключову роль у його життєвому циклі, визначаючи не лише функціональність та зручність використання, але й надійність, масштабованість та економічну ефективність. В умовах швидкої цифровізації, де програмні продукти активно застосовуються в критичних сферах фінансах, медицині, промисловості гарантування їхньої якості стає не лише технічним, а й стратегічним завданням для розробників. Порушення якості призводить до значних фінансових збитків, втрати користувацької довіри й ускладнення супроводу програмного забезпечення.

Сучасна індустрія програмної інженерії пропонує низку методів і засобів забезпечення якості: від процесів рецензування коду до автоматизованого тестування та аналізу. Окреме місце у цьому контексті займають інструменти статичного аналізу коду, які дають змогу виявляти потенційні помилки ще до етапу виконання програми. Їхня інтеграція в середовище розробки сприяє підвищенню надійності ПЗ та пришвидшує цикл зворотного зв’язку між розробником і кодом.

Актуальність теми дослідження обумовлена зростанням потреби в доступних і водночас ефективних інструментах підвищення якості програмного забезпечення, які можуть бути використані як у великих проєктах, так і в малих розробницьких командах або індивідуальними розробниками.

Метою роботи є аналіз, порівняння та практична апробація інструментів забезпечення якості програмного забезпечення, зокрема засобів статичного аналізу коду.

Об'єктом дослідження є процес забезпечення якості програмного забезпечення.

Предметом дослідження є методи статичного аналізу коду та відповідні програмні інструменти.

Завдання дослідження:

- дослідити поняття якості ПЗ та ключові методи її забезпечення;
- проаналізувати принципи статичного аналізу коду;
- розглянути функціональні можливості інструментів ESLint, SonarLint і Pylint;
- порівняти ефективність аналізаторівна прикладі реального фрагмента коду;
- сформулювати висновки щодо доцільності впровадження в різних проєктних середовищах.

Методи дослідження: аналіз наукових джерел, порівняльний метод, демонстрація на прикладі.

Робота має на меті не лише теоретичний огляд, а й практичне застосування інструментів з урахуванням сучасних вимог до ефективності, точності й доступності засобів підвищення якості коду.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПЗ

1.1. Поняття якості програмного забезпечення

У понятті якості програмного забезпечення (ПЗ) закладено сукупність властивостей, які визначають його придатність до використання відповідно до поставлених цілей [3, 8]. Це не обмежується функціональною правильністю чи продуктивністю: розгляд охоплює зручність супроводу, переносність, захищеність і передбачуваність поведінки у нестандартних ситуаціях. У стандартах ISO/IEC 25010:2011 (раніше ISO/IEC 9126) наведено розгорнутумодель, що включає вісім характеристик якості, серед яких: функціональна придатність, продуктивність, сумісність, надійність, зручність використання, безпека, супровідність і переносність [8, 22]. Така багатовимірність обумовлює складність вимірювання якості в термінах, що піддаються формалізації [3].

Поняття якості програмного продукту не є усталеним у межах лише технічних параметрів. Від початку розвитку інженерії ПЗ, приблизно з 1968 року, відбувся перехід від спрощених формулювань «відсутність помилок» до моделей, що враховують повноту документації, передбачуваність взаємодії з користувачем, масштабованість у складних архітектурах [13, 14, 20]. Деякі автори, як-от Pressman (2010), акцентують на залежності між якістю та витратами: за його підрахунками, виправлення дефекту на етапі експлуатації обходиться в 10–50 разів дорожче, ніж під час кодування [8]. Тобто якість постає не лише як технічна, а як економічна категорія [6].

Наявні підходи до визначення якості ПЗ демонструють прагнення до уніфікації, проте не усувають наявної розбіжності між розробницькою і користувацькою оцінкою [3, 9, 22]. Наприклад, стабільне ядро з внутрішньо оптимізованими алгоритмами може вважатися якісним з погляду архітектора системи, проте залишатися незрозумілим для користувача через недружній інтерфейс [10, 39]. Це суперечччя вимагає комплексного аналізу, в межах якого враховуються як інженерні параметри (кількість дефектів на тисячу рядків коду, відсоток покриття тестами), так і експлуатаційні характеристики

(кількість звернень до служби підтримки, тривалість адаптації користувача до ПЗ) [3, 22]. З огляду на багатокомпонентність цього процесу, доцільно візуалізувати його структурну організацію, що демонструє ключові етапи забезпечення якості в межах життєвого циклу ПЗ (див. Рис. 1.1).

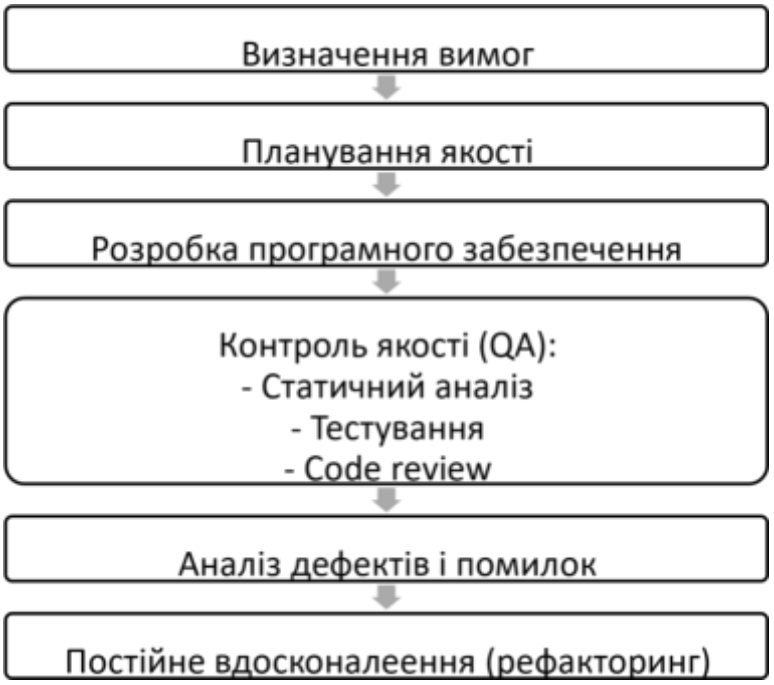


Рис. 1.1. Етапи забезпечення якості програмного забезпечення

Формальна оцінка якості ПЗ зазвичай передбачає використання кількісних метрик. До поширених належать: коефіцієнт щільності дефектів (defect density), середній час до відмови (MTTF), відсоток покриття коду тестами (code coverage), швидкість виконання критичних функцій, а також складність коду, що

обчислюється через метрику цикломатичної складності (McCabe, 1976) [8, 22]. Кожна з цих метрик має обмежене застосування й не може бути самодостатнім показником, проте у сукупності вони дозволяють будувати профіль якості продукту в динаміці його розробки [3, 6].

Таблиця 1.1

Порівняльна характеристика якісних показників програмних продуктів різного типу

Тип програмного продукту	Кількість дефектів на 1000 рядків коду	Покриття модульними тестами, %	Середній час до відмови, год.
Корпоративна система управління ресурсами (ERP)	0,7	82	1400
Мобільний застосунок	1,3	56	400
Вбудоване програмне забезпечення для IoT пристроїв	0,4	91	1900

Умови використання кожного з показників мають уточнюватися з урахуванням специфіки галузі. Наприклад, для медичних систем прийнятною вважається щільність помилок не вище 0.2 на тисячу рядків коду, тоді як для вебзастосунків допустиме значення може сягати 1.5 [3, 22]. Співвідношення між глибиною тестового покриття та надійністю функціонування теж не є лінійним: деякі помилки не виявляються навіть при 100% покритті через неадекватність тестових сценаріїв [8, 20]. Це створює додаткові труднощі в обґрунтуванні високої якості програмного продукту на основі формальних критеріїв [3, 6, 22].

Оцінювання якості програмного забезпечення не є універсальним процесом. Його параметри варіюються залежно від типу застосунку, бізнесмоделі, вимог користувача та обмежень архітектури [9, 14, 22]. Наприклад, для систем реального часу пріоритетом є стабільність і передбачуваність виконання, тоді як для користувацьких вебзастосунків акцент зміщується в бік інтуїтивності інтерфейсу та швидкості відгуку [10, 39]. Внаслідок цього постає проблема узгодження метрик: одна й та сама система може відповідати високим критеріям за одними показниками та не відповідати за іншими, що викликає необхідність багатофакторного аналізу [3, 6, 8]. У

деяких випадках це призводить до того, що оцінка якості здійснюється в термінах компромісу, а не абсолютних значень [6].

Міжнародні стандарти, зокрема ISO/IEC 25010:2011, передбачають поділ якості на внутрішню, зовнішню та якість у процесі використання [8, 22]. Внутрішня якість охоплює властивості, які можна оцінити без виконання ПЗ (структура коду, стиль, зв’язаність модулів) [3, 20]. Зовнішня якість виявляється під час виконання програми в контрольованих умовах (результати тестування, швидкість реакції, обробка помилок) [8, 22]. Якість у процесі використання відображає досвід кінцевого користувача: ефективність, задоволення, свобода від помилок [10, 39]. Кожен із цих рівнів має власні критерії, а між ними немає повної кореляції. Це означає, що програмний продукт із високим показником внутрішньої якості може викликати труднощі в користувача через недоліки в інтерфейсі або логіці навігації [10, 39].

Одним із поширених підходів до забезпечення якості є впровадження стандартів процесу розробки, як-от CMMI (Capability Maturity Model Integration) або ISO/IEC 12207 [8, 22]. Ці моделі спрямовані не на оцінку результату, а на забезпечення контрольованості самого процесу розробки. У межах такої логіки якість розглядається як похідна від організованого, повторюваного процесу, що базується на документованих процедурах, відстеженні помилок, управлінні ризиками та валідації проміжних результатів [7, 22]. І хоча такий підхід забезпечує стабільність, він не гарантує творчої гнучкості або інновацій, які часто необхідні для нестандартних або продуктових розробок [6, 8].

У теоретичному плані якість програмного забезпечення може розглядатись через призму атрибутів надійності, доступності, відновлюваності та зручності супроводу [3, 8, 22]. Всі ці характеристики мають тенденцію до суперечностей: підвищення одного з параметрів може призвести до зниження іншого. Наприклад, збільшення складності логіки для досягнення високої продуктивності може ускладнити супровід, тоді як надмірна гнучкість архітектури іноді призводить до зниження надійності [6, 9]. Це обмежує

можливість створення універсальних шаблонів розробки й висуває вимогу до локальної адаптації практик забезпечення якості у межах кожного окремого проєкту [8, 14, 22].

Теоретична модель програмного забезпечення як системи з високим ступенем складності та змінюваності формує запит на динамічне розуміння якості [3, 6, 22]. Це означає, що характеристики, які оцінювались як релевантні на момент планування або початкової розробки, можуть втратити актуальність у ході експлуатації або після релізу. Зокрема, деякі функції можуть стати надлишковими, інтерфейс застарілим, архітектурні рішення несумісними з новими технологіями [8, 39]. Отже, якість програмного забезпечення не є фіксованою величиною: вона постає як результат постійного балансування між технічними можливостями, бізнес-завданнями та очікуваннями користувача [3,6].

У сучасних публікаціях із теорії програмної інженерії дедалі частіше фіксується тенденція до зміщення фокусу з якості продукту на якість процесу [6, 22, 39]. Це зумовлено тим, що формальна безпомилковість коду або наявність функцій не завжди забезпечують передбачувану поведінку системи в реальних умовах [8, 14]. Натомість процес, який структуровано, регламентовано й супроводжується засобами контролю, дає змогу зменшити імовірність помилок і стабілізувати випуск нових версій [7, 22]. У цій логіці важливими стають не лише технічні артефакти, а й практики взаємодії в команді, спосіб ведення документації, контроль змін і формалізований зворотний зв’язок [6, 9].

Вивчення якісних характеристик ПЗ у контексті гнучких методологій виявляє ще один проблемний вузол: конфлікт між адаптивністю та передбачуваністю. Методики, подібні до Scrum або Kanban, орієнтовані на швидке реагування на зміну вимог, проте можуть недооцінювати необхідність довгострокового планування й стандартизації [9, 22, 39]. Унаслідок цього якість у таких підходах частково делегується автоматизованим засобам статичним аналізаторам коду, інтегрованим системам тестування, засобам

перевірки залежностей [8, 40, 77]. Це дає змогу частково компенсувати нестачу жорстких специфікацій, але не усуває необхідності у верифікації на рівні архітектури або дизайну [22, 59].

У межах локального проєкту оцінка якості може бути зведена до контролю базових метрик: кількість дефектів, що зафіксовані на етапі тестування; покриття коду тестами; кількість незадокументованих функцій; результати перевірки з використанням статичних аналізаторів [3, 6, 8]. Проте така оцінка має обмежену репрезентативність. Наприклад, низький рівень виявлених помилок може бути наслідком недостатньо розроблених тест-кейсів, а не реального підвищення якості [20, 22]. Крім того, зменшення кількості зовнішніх дефектів іноді супроводжується накопиченням технічного боргу в коді, що не є видимим одразу [8, 22, 59].

Практичний вимір розуміння якості ПЗ дедалі частіше поєднується з використанням автоматизованих інструментів, які дають змогу оцінити код за формальними критеріями [40, 59, 77, 80]. До таких засобів належать, зокрема, ESLint, SonarLint, Pylint та інші системи статичного аналізу [40, 77, 80]. Вони здатні виявляти синтаксичні помилки, потенційні вразливості, порушення стилістики та неефективні патерни. Ці інструменти частково знімають проблему суб’єктивності при рев’ю коду, але не замінюють повноцінної аналітики з боку архітектора або технічного керівника [7, 22]. Крім того, обмеження таких засобів проявляються в неспроможності інтерпретувати логіку в контексті бізнесзавдань [8, 22, 40].

Отже, якість програмного забезпечення не може бути зведена до єдиної метрики або універсального стандарту [3, 8, 22]. Вона постає як динамічна множина властивостей, релевантність яких змінюється залежно від типу проєкту, очікувань користувача, практик команди та середовища розгортання [6, 9, 22]. Формальні моделі, такі як ISO/IEC 25010, надають орієнтири, але не виключають потреби в локальній інтерпретації [8, 22]. Забезпечення якості вимагає поєднання кількісного аналізу, контекстного оцінювання та зваженого використання інструментів від ручного тестування до автоматизованого

аналізу коду [7, 20, 40, 59].

1.2. Основні дефекти в ПЗ та їх наслідки

Питання кількісного оцінювання якості програмного забезпечення передбачає введення формалізованих критеріїв, які можуть бути виміряні, порівняні й використані для контролю в динаміці розробки [3, 6, 22]. У науковій літературі прийнято розрізняти метрики внутрішньої й зовнішньої якості. До першої групи належать характеристики, що оцінюються на рівні коду: складність, зв'язаність, покриття тестами, консистентність [3, 8, 22]. Зовнішня якість пов'язана з поведінкою програми під час виконання: стабільність, точність, відгук на зовнішні події, реакція на помилки [8, 22, 39]. Поділ є умовним, оскільки деякі метрики можуть бути віднесені до обох категорій залежно від контексту застосування [3, 22].

Один із найбільш уживаних підходів до класифікації критеріїв якості запропоновано в стандарті ISO/IEC 25010:2011 [8, 22]. У межах цієї моделі якість поділяється на вісім характеристик, кожна з яких, у свою чергу, складається з декількох підхарактеристик. Наприклад, характеристика «функціональна придатність» включає повноту реалізації функцій, їхню точність та відповідність очікуванням користувача. «Надійність» розкладається на зрілість, доступність, відновлюваність та здатність витримувати збої [22]. Така структура забезпечує деталізацію і дозволяє здійснювати цілеспрямований аналіз, однак потребує конкретизації інструментів вимірювання кожного підпункту [3, 8, 22].

Згідно з підходом Мусса (Musa, 1993), метрика має бути валідною (тобто такою, що дійсно вимірює задекларовану характеристику), надійною (стабільною при повторному вимірюванні) і простою для інтерпретації [3, 6]. Не всі метрики, які використовуються в практиці розробки, відповідають цим критеріям. Наприклад, кількість рядків коду (LOC) є метрикою, що легко підраховується, але її зв'язок із якістю не є прямим: більший обсяг коду може бути як результатом поганої оптимізації, так і наслідком більшої функціональної складності [6, 8, 22]. Тому така метрика потребує нормалізації

або поєднання з іншими показниками, наприклад з кількістю дефектів на тисячу рядків [3, 20].

До показників, що мають вищу ступінь узагальнення, належить метрика дефектної щільності (defect density), яка відображає кількість виявлених помилок на певну кількість коду [3, 22]. Її застосування дає змогу порівнювати проєкти різної складності або оцінювати ефективність змін у підходах до розробки [6, 22]. Разом із тим ця метрика залишається залежною від способу тестування: якщо охоплення тестами є низьким, фактична кількість помилок може залишитися невідомою [8, 20]. Тому вона ефективна лише в поєднанні з показником глибини тестового покриття, що вказує на відсоток коду, охоплений модульними або інтеграційними тестами [3, 8, 22].

Окрему категорію становлять метрики складності, серед яких найчастіше використовуються цикломатична складність (McCabe complexity) та індекс глибини вкладеності [3, 8]. Ці показники дозволяють оцінити кількість логічних шляхів через програму, що прямо впливає на її тестованість і потенційну помилковість [6, 22]. Наприклад, функція з цикломатичною складністю понад 10 вважається складною для супроводу й потенційно нестабільною [22]. Високі значення таких показників сигналізують про необхідність рефакторингу, оскільки ускладнена логіка підвищує ризик прихованих дефектів і ускладнює процес розуміння коду новими учасниками команди [3, 8, 22].

У практиці розробки важливим інструментом аналізу структури програми є показники зв’язаності (cohesion) та зчеплення (coupling) [3, 8, 22]. Висока зв’язаність в межах модуля, що означає функціональну однорідність компонентів, зазвичай трактується як позитивне явище. Натомість сильне зчеплення між модулями свідчить про залежність і зменшує гнучкість системи [6, 8]. Метрики цього типу наприклад, індекс інформаційного зчеплення або число викликів між модулями можуть бути розраховані автоматично [22, 40, 59]. Вони дозволяють оцінити архітектурну якість системи, однак потребують контекстуального тлумачення: у деяких випадках високе зчеплення є наслідком

вимог до продуктивності або сумісності [3, 9].

Ще один клас метрик стосується ефективності коду. Йдеться не лише про часові або ресурсні витрати, а й про енергоспоживання, що стає дедалі релевантнішим у мобільних або вбудованих системах [22, 53, 59]. Метрики цього типу менш формалізовані, часто потребують профілювання на конкретному апаратному забезпеченні та не мають стабільної норми [6, 22]. Тим не менш, на етапі оптимізації продуктивності використовуються показники середнього часу виконання функцій, частоти викликів, обсягу використаної оперативної пам’яті [8, 20, 53]. У контексті якісного аналізу вони становлять допоміжний інструмент, що підтверджує ефект від змін, але не може бути застосований на ранніх етапах без моделювання середовища [3, 6, 22].

У табл. 1.2 узагальнено кілька поширених метрик, які використовуються для оцінювання внутрішньої якості коду, з коротким описом і типовим діапазоном значень, що вважається прийнятним для стабільної розробки.

Таблиця 1.2

Поширені метрики внутрішньої якості програмного коду

Назва метрики	Опис	Оптимальні значення
Цикломатична складність	Кількість незалежних логічних шляхів у функції	до 10
Покриття тестами	Частка коду, що перевірена модульними тестами	не менше 80%
Щільність дефектів	Помилки на 1000 рядків коду	до 0,8
Глибина вкладеності	Максимальна кількість рівнів вкладених блоках	не більше 4–5
Зчеплення між модулями	Кількість зовнішніх викликів	Залежить від архітектури

Оцінювання зовнішньої якості потребує звернення до таких метрик, як середній час до відмови (MTTF), середній час відновлення (MTTR), імовірність успішного виконання транзакції (success rate), кількість звернень до служби підтримки або кількість аварійних завершень [3, 6, 22]. Ці метрики використовуються вже після релізу та накопичуються у процесі експлуатації, що накладає обмеження на їх застосування в проєктах, що перебувають у фазі активної розробки [6, 22]. Водночас вони дають змогу виявити критичні зони системи, що виявилися нестабільними за реальних умов, і скоригувати пріоритети в оновленнях або модульних тестах [3, 20].

Інтеграція якісних метрик у процес розробки передбачає не лише збір показників, але й побудову системи контролю, яка дозволяє інтерпретувати ці дані в контексті конкретного проєкту [8, 22, 59]. Наприклад, перевищення порогу цикломатичної складності для окремої функції може трактуватися як привід для обов'язкового рев'ю або рефакторингу [3, 8]. Автоматичні аналізатори коду, зокрема SonarLint, ESLint або Pylint, зазвичай мають убудовану систему правил, що базується саме на цих метриках [40, 59, 80]. У результаті система контролю якості набуває ознак формалізованого регламенту, який підлягає адаптації, але зберігає структурованість і контрольованість [8, 22, 77].

Застосування метрик у межах гнучких методологій розробки передбачає специфічний підхід [9, 22, 39]. У таких проєктах, де час між комітами є коротким, а фіксація вимог – інкрементальною, особливого значення набуває автоматизоване збирання показників [40, 59, 77]. Наприклад, інструменти, що інтегруються з системами контролю версій або середовищами розробки, дозволяють відстежувати зміну складності, динаміку покриття тестами, частоту появи технічного боргу [6, 22, 80]. У цьому контексті якість коду часто розглядається як агрегований показник, що містить у собі кілька складових, кожна з яких має власну динаміку змін [3, 6, 8].

Проблемним у практиці використання метрик залишається питання контексту інтерпретації [6, 22]. Той самий показник складності або зв'язаності

може трактуватися по-різному залежно від характеру проєкту, команди, етапу розробки та обраної архітектури [3, 8, 14]. Зокрема, показник глибини вкладеності, що перевищує умовну межу в п’ять рівнів, не обов’язково свідчить про низьку якість: він може бути виправданий особливостями обробки винятків або специфікою обчислювального алгоритму [6, 9]. У зв’язку з цим частина дослідників, зокрема Basili (2004), пропонує формувати адаптивні профілі якості з урахуванням доменних особливостей, а не спиратись на фіксовані нормативи [22].

Певне ускладнення також викликає питання формального узгодження метрик, коли ті самі властивості описуються за допомогою різних показників [3, 6, 22]. Наприклад, функціональна повнота може оцінюватися як кількість реалізованих вимог, а також як кількість протестованих сценаріїв, що задовольняють ці вимоги [3, 22, 39]. У разі невідповідності між цими підходами виникає необхідність або в нормалізації метрик, або у вибудовуванні мета-рівня аналітики [6, 22]. Такий рівень передбачає побудову ієрархії оцінювання, у якій кожен показник має вагу, а не абсолютне значення [3, 6, 22]. Це ускладнює практичне застосування, але підвищує точність управлінських рішень [8, 22].

Додаткову увагу в рамках сучасних підходів привертають метрики технічного боргу. Йдеться про оцінювання тих компонентів, які виконують свої функції, але потребують переробки з міркувань стабільності, супровідності або продуктивності [6, 8, 22]. Такі метрики не завжди піддаються автоматичному аналізу: вони потребують ручної ревізії, логічного міркування, іноді залучення історії змін [3, 20, 40]. У ряді випадків технічний борг проявляється лише через кілька релізів, що унеможливлює його своєчасне виявлення формальними засобами [22, 59]. Попри це, його кількісна оцінка (наприклад, у годинах, необхідних для виправлення) поступово інтегрується в системи контролю версій та DevOps-платформи [40, 77, 80].

Підсумовуючи, можна зазначити, що метрики якості програмного забезпечення є необхідним, проте не самодостатнім інструментом оцінювання

[3, 6, 22]. Їхня ефективність залежить від коректності застосування, відповідності контексту та чіткості інтерпретаційної моделі [8, 9, 22]. У поєднанні з якісним аналізом, досвідом команди та стратегією проєкту метрики дозволяють виявляти ризики, контролювати динаміку розвитку системи та приймати обґрунтовані рішення [6, 8, 22]. Проте надмірне покладання на числові значення без урахування специфіки домену або архітектури може призвести до хибних висновків [9, 14, 22].

1.3. Методи забезпечення якості (ручне тестування, рецензії коду, автоматичний аналіз)

Аналіз причин зниження якості програмного забезпечення виявляє, що більшість проблем виникає не на рівні технологій, а внаслідок організаційних, комунікаційних або процедурних похибок [6, 8, 22]. Незбалансоване планування, відсутність чітко зафіксованих вимог, недостатня взаємодія між учасниками розробки ці чинники впливають на підсумковий стан коду не меншою мірою, ніж вибір мови програмування або архітектурного шаблону [9, 22, 39]. Проблема ускладнюється тим, що наслідки початкових помилок часто виявляються на пізніх етапах, коли вартість їх усунення зростає експоненційно [6, 8, 14].

Одним із найпоширеніших порушень є відсутність або фрагментарність документації. Це створює труднощі як під час тестування, так і при супроводі, особливо у випадках, коли змінюється склад команди або розробка триває кілька ітерацій [3, 6, 22]. Документація, що не оновлюється, також стає джерелом хибних уявлень про функціональність системи, а отже призводить до неправильного тестування, помилкових архітектурних рішень і зриву релізів [8, 20, 22]. У випадках, коли документація замінюється усною передачею знань, ризик втрати інформації зростає непропорційно до обсягів проєкту [7, 9].

Інша суттєва проблема пов’язана з нечіткістю або суперечністю функціональних вимог. У реальних проєктах специфікація рідко є завершеною на початку розробки [3, 6, 22]. Це створює умови для фрагментарного

розуміння задач, надмірної імпровізації, а іноді реалізації конфліктних функціональних блоків [8, 9, 39]. Невизначеність на рівні вимог ускладнює побудову архітектури, а також формування адекватного тестового покриття [6, 22]. Внаслідок цього навіть технічно безпомилкове програмне забезпечення може не відповідати очікуванням замовника або користувача [3, 14, 22].

Значну частину дефектів формують помилки проєктування [3, 8, 22]. Вони не є синтаксичними, тому не виявляються на етапі компіляції або базового тестування [22, 59]. Йдеться про неправильне структурування логіки, порушення принципів модульності, неефективне використання шаблонів проєктування або відсутність таких узагалі [6, 8, 9]. Такі помилки мають кумулятивний ефект: кожен новий функціональний блок, побудований на непослідовній логіці, ускладнює систему, зменшуючи її адаптивність і стабільність [8, 22]. У ряді випадків це призводить до необхідності повного перегляду системної архітектури вже після початку експлуатації [22, 59].

До проблем, які виявляються на межі між технічним і організаційним рівнями, належить незбалансоване тестування [3, 8, 22]. У багатьох проєктах тестування зводиться до формального підтвердження функціонування основних сценаріїв, без аналізу граничних умов, винятків або нетипової поведінки [6, 20, 40]. Наявність ручного тестування без автоматизації зменшує стабільність повторної перевірки, а розподіл тестової відповідальності між різними учасниками ускладнює відстеження дефектів [8, 22, 59]. При цьому самі тести нерідко зберігаються у вигляді неструктурованих скриптів або навіть залишаються неформалізованими [3, 40, 77].

Інтенсивне використання сторонніх бібліотек і залежностей також створює ризики для якості, особливо якщо ці компоненти не проходять регулярного оновлення або не мають стабільної підтримки з боку спільноти [40, 59, 77]. Уразливості безпеки, несумісність із новими версіями платформи, порушення ліцензійних умов усе це може виявитися лише після інтеграції та значно ускладнити подальшу експлуатацію [22, 26, 33]. Крім того, використання бібліотек з відкритим кодом часто супроводжується відсутністю

стандартизованих підходів до перевірки їхньої якості, що ускладнює аудит і підвищує рівень технічного боргу [40, 77, 80].

Проблемою, що рідко фіксується формальними метриками, є технічна інерція або звичка до застарілих практик [8, 22, 59]. Команди, які тривалий час працюють із певним технологічним стеком, схильні відтворювати знайомі рішення, навіть якщо вони не є оптимальними [6, 9, 22]. Це стосується як вибору архітектурних підходів, так і стилю програмування. Підтримка зворотної сумісності, відмова від рефакторингу, ігнорування нових версій мов або фреймворків усе це сприяє накопиченню технічного боргу [3, 14, 22].

Відсутність системи контролю якості коду під час інтеграції ще одна типова причина дефектів [8, 20, 22]. У багатьох випадках рев'ю коду здійснюється формально або зовсім не проводиться [6, 9, 59]. Це унеможливлює виявлення прихованих помилок логіки, порушень стилю або неефективних реалізацій, які не фіксуються тестами [3, 40, 80]. Практика інтеграції без контролю супроводжується нестабільністю основної гілки коду, частими конфліктами при злитті та труднощами під час автоматичного деплойменту [22, 77]. За відсутності чіткої політики рев'ю зростає ризик індивідуальних помилок і знижується колективна відповідальність за якість [6, 22, 39].

Нерівномірний розподіл компетенцій усередині команди часто спричиняє залежність від окремих розробників [7, 9, 22]. У складніших системах це проявляється як «інформаційна ізоляція», коли частина логіки зрозуміла лише окремому члену команди [6, 8, 22]. У разі його відсутності або звільнення технічна документація виявляється неповною, а підтримка обмеженою [3, 14, 22]. Така ситуація не лише ускладнює тестування та рефакторинг, а й знижує стійкість системи до змін [8, 22].

Також слід згадати про проблеми, що виникають через неузгодженість середовищ розробки, тестування та експлуатації [22, 39, 59]. Наприклад, конфігурації баз даних, версії бібліотек або операційних систем можуть відрізнятися, що призводить до виникнення помилок, які не проявляються на

етапі розробки [6, 20, 77]. У ряді випадків це стає причиною «випадкових» збоїв після релізу, виявити які складно, оскільки їх неможливо відтворити у внутрішньому середовищі [3, 9, 22]. Такі проблеми не лише ускладнюють тестування, а й знижують довіру до процесу контролю якості загалом [6, 8, 22].

Незбалансоване використання автоматизованих інструментів також може спричиняти парадоксальні ефекти. Наприклад, надмірне покладання на статичний аналіз коду без подальшої ручної верифікації призводить до ситуації, коли реальні дефекти залишаються поза увагою, а система генерує значну кількість хибнопозитивних результатів [40, 59, 80]. Подібне навантаження на команду, не підкріплене аналітичним переглядом, лише імітує контроль якості, не покращуючи об'єктивного стану коду [6, 22, 77]. Ефективність автоматизації залежить не лише від її наявності, а від інтеграції у методологію розробки та культури відповідальності за результати [3, 8, 22].

Короткі терміни виконання проєктів або зміщення акцентів у бік швидкої реалізації функціональності часто супроводжуються ігноруванням практик тестування або документування [6, 9, 22]. Поняття «технічний борг» у такому контексті набуває конкретного виміру: зменшується читабельність коду, ускладнюється інтеграція змін, зростає кількість помилок при внесенні нових функцій [8, 14, 22]. Цей борг не завжди проявляється одразу, але з часом накопичується, знижуючи продуктивність команди та створюючи постійний фон нестабільності [3, 6, 22].

Системи з високим ступенем модульності нерідко стикаються з проблемами сумісності між модулями, особливо якщо відсутня єдина схема інтерфейсів або політика взаємодії [6, 9, 22]. У таких випадках тестування обмежується перевіркою кожного модуля окремо, в той час як інтеграційні помилки залишаються поза увагою [3, 8, 22]. Подібна ситуація характерна для проєктів із багатьма підрядниками або за умов використання мікросервісної архітектури [22, 59, 77]. Відсутність узгоджених стандартів передавання даних, логування та обробки помилок ускладнює підтримку та відлагодження [6, 22, 39].

На завершення варто звернути увагу на недооцінку зворотного зв’язку від кінцевих користувачів [9, 14, 22]. У системах, що не передбачають механізмів збору фідбеку або аналітики поведінки користувачів, складно оцінити реальну ефективність інтерфейсу, логіки або стабільності [3, 10, 39]. Це особливо суттєво для продуктів, які мають взаємодіяти з широким колом користувачів або розраховані на масове розгортання [8, 22]. Відсутність даних щодо сценаріїв використання призводить до хибного уявлення про якість системи, базованого виключно на внутрішньому тестуванні [6, 22].

Узагальнюючи викладене, можна констатувати, що проблеми якості мають багатофакторну природу [3, 6, 22]. Їх джерела не зводяться до технічної компетенції окремих розробників, а охоплюють організаційні практики, культурні установки, інструментальне середовище та специфіку взаємодії в команді [8, 9, 22]. Усвідомлення цього дозволяє переорієнтувати акценти контролю якості з кінцевих результатів на процеси, що формують ці результати [6, 22]. Відповідно, ефективне управління якістю має включати не лише перевірку коду, а й аналіз комунікаційних, процедурних і управлінських рішень, що супроводжують розробку [3, 8, 22].

ВИСНОВОК ДО РОЗДІЛУ 1

У результаті проведеного аналізу встановлено, що проблема якості програмного забезпечення не зводиться лише до фінального стану програмного продукту, а охоплює весь життєвий цикл розробки від постановки вимог до супроводу після релізу. Якість є складною категорією, що формується на перетині інженерних рішень, управлінських практик і технологічних інструментів, які забезпечують виявлення, попередження або усунення дефектів на різних етапах створення ПЗ.

Узагальнення підходів до визначення якості дозволило виокремити дві основні парадигми: орієнтація на користувача та відповідність технічним специфікаціям. У межах кожної з них сформувалися власні критерії оцінювання: для першої це зручність, продуктивність, відмовостійкість; для другої точність виконання функцій, коректність роботи, відповідність стандартам. Таке розмежування свідчить про поліаспектний характер проблеми, що не дозволяє застосовувати універсальні рішення.

Розглянуті методи забезпечення якості класифікуються за формою реалізації (технічні, організаційні, процедурні) та характером втручання в процес (проактивні й реактивні). Акцент у сучасних дослідженнях робиться на проактивних підходах, де помилки і дефекти намагаються запобігти ще до написання коду. Це зумовлює потребу в ранній інтеграції інструментів забезпечення якості в інфраструктуру проєкту зокрема через аналіз вимог, архітектурне рев'ювання та автоматизований контроль на рівні середовищ розробки.

Проведена систематизація інструментів управління якістю засвідчила, що найбільш поширеними є засоби статичного аналізу, динамічного тестування, моніторингу продуктивності та контролю тестового покриття. Кожна категорія має свою функціональну нішу, визначається специфічними алгоритмами аналізу та формує відмінний тип результатів. Це створює

передумови для їхньої комбінації, з метою досягнення комплексного охоплення дефектів на різних рівнях абстракції.

На основі порівняння сучасних підходів і фреймворків (наприклад, ISO/IEC 25010, SQA, DevOps-практик) встановлено, що інтеграція якості має бути безперервною й адаптивною. Це передбачає не лише автоматизацію контрольних точок, але й періодичну ревізію обраних метрик, зміну конфігурацій, навчання персоналу. Успішність системи забезпечення якості не гарантується наявністю окремих інструментів, а визначається їхньою взаємодією, релевантністю до конкретного середовища розробки та готовністю команди працювати з результатами перевірок.

Таким чином, Розділ 1 сформував аналітичну основу для подальшого розгляду конкретних інструментів і технологій. Сформульовані положення дозволяють перейти до емпіричного огляду інструментів забезпечення якості, їхньої ефективності, практик інтеграції та обмежень, що розглядаються у другому розділі дослідження.

РОЗДІЛ 2 ОГЛЯД ІНСТРУМЕНТІВ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ

2.1. Статичний аналіз коду: принцип дії

Статичний аналіз коду є одним з найрозповсюдженіших методів попереднього контролю якості, який дозволяє виявляти помилки ще до виконання програми [3, 6, 22]. На відміну від динамічного тестування, статичний аналіз не потребує запуску коду, оскільки оперує його синтаксичним та семантичним представленням [8, 40, 77]. Інструменти такого типу здійснюють перевірку структури, типів, стилістичних вимог, і часто виявлення потенційних уразливостей без звернення до контексту виконання [22, 59, 80]. Це дає змогу виявляти дефекти, які залишаються поза зоною досяжності під час звичайного функціонального тестування [20, 22].

Одним із базових механізмів статичного аналізу є побудова абстрактного синтаксичного дерева (AST), що репрезентує структуру коду у вигляді вузлів та гілок [8, 22, 59]. Кожен вузол відповідає за певну мовну конструкцію: оператор, вираз, умовний блок тощо. Така репрезентація дозволяє аналізатору не лише перевіряти окремі фрагменти, а й виявляти глибинні помилки, пов’язані з послідовністю оголошень, використанням змінних або рекурсивними викликами [40, 59]. За наявності шаблонів порушень або відхилень від конвенцій, система формує звіт із позначенням критичних і незначних дефектів [22, 77, 80].

Сучасні інструменти статичного аналізу не обмежуються перевіркою синтаксису або стилю [40, 59, 80]. Вони включають модулі семантичного аналізу, які можуть фіксувати потенційні утрати точності типів, невикористані змінні, логічні помилки умовних операторів або неправильно реалізовані обробки винятків [22, 59]. Це дозволяє виявляти дефекти, які не є помилками з погляду компілятора, але ускладнюють підтримку, рефакторинг або подальше масштабування проєкту [6, 8, 22]. Особливо це стосується мов із динамічною типізацією, де частина логіки верифікується лише під час виконання [59, 80].

Іншою суттєвою перевагою є можливість застосування правил

кодування, що ґрунтуються на загальноприйнятих або індивідуально розроблених стилістичних стандартах [3, 22, 40]. У багатьох командах використання лінтерів або форматерів на етапі commit-ів у Git-процесі стало стандартною практикою [6, 8, 40]. Це дозволяє мінімізувати суб'єктивні розбіжності у стилі та забезпечити послідовність коду незалежно від кількості учасників розробки [22, 77]. До того ж, автоматичне виправлення певних типів помилок пришвидшує рев'ю і знижує поріг входження для нових учасників команди [40, 59, 80].

Незважаючи на ефективність, статичний аналіз має обмеження. Він не фіксує помилки, що залежать від зовнішнього середовища, конфігурації або реального введення даних [6, 22, 59]. Крім того, надмірна кількість хибнопозитивних спрацьовувань може демотивувати розробників і зменшити довіру до інструменту [40, 77, 80]. У зв'язку з цим значна увага приділяється налаштуванню правил перевірки, ваговим коефіцієнтам помилок та інтерпретації результатів [22, 40, 59]. Баланс між кількістю попереджень і їх релевантністю є критичним для ефективної інтеграції аналізу в процес розробки [6, 22].

Оптимальною практикою є інтеграція статичного аналізу у процес безперервної інтеграції (CI), де перевірка коду здійснюється автоматично під час кожного запиту на злиття гілок [40, 77, 80]. Це забезпечує своєчасне виявлення відхилень від стандартів, знижує ймовірність поширення дефектів у головну гілку репозиторію та сприяє автоматизованому контролю якості [8, 22, 59]. Крім того, така інтеграція дозволяє відслідковувати динаміку кількості помилок у кодовій базі, формувати метрики та приймати обґрунтовані рішення щодо стабільності релізів [6, 22, 59].

Інструменти статичного аналізу часто доповнюються функціональністю порівняння результатів між комітами, що дозволяє локалізувати джерела помилок і зменшити навантаження на процес рецензування [59, 77, 80]. Наприклад, якщо нова версія коду не збільшує кількість дефектів або покращує індекс складності, така зміна може бути прийнята без додаткової перевірки [8,

22]. Подібна автоматизація дозволяє скоротити тривалість розробки та підвищити ефективність внутрішньокомандних взаємодій [6, 40, 77].

Не менш істотним є те, що статичний аналіз забезпечує базу для формування внутрішніх стандартів якості програмного забезпечення [3, 8, 22]. Команди розробників можуть створювати власні правила перевірки, що відображають особливості бізнес-логіки, технологічного стеку або середовища виконання [6, 9, 22]. Це дає змогу адаптувати процес аналізу до конкретного проєкту та уникнути надмірної формалізації, яка часто властива універсальним системам [40, 59, 80].

Деякі інструменти надають можливість інтеграції з системами керування вимогами та звітності, що дозволяє простежити шлях дефекту від етапу аналізу до його усунення [22, 59, 77]. У корпоративних середовищах така інтеграція відіграє ключову функцію в системах забезпечення відповідності (compliance), де збереження повної історії перевірок є обов’язковою умовою аудиту [33, 77, 80]. Це особливо актуально для проєктів, які підпадають під дію галузевих стандартів (наприклад, ISO/IEC 25010, MISRA C або OWASP) [22, 77].

Загалом, статичний аналіз не є заміною тестування, проте він забезпечує додатковий рівень контролю, який дозволяє виявляти дефекти на ранній стадії життєвого циклу розробки [6, 8, 22]. Його ефективність значною мірою залежить від правильного налаштування, обґрунтованого вибору інструментарію та здатності команди інтерпретувати результати [3, 6, 59]. Перевага полягає не лише у виявленні помилок, а в систематизації процесу перевірки, що виводить розробку на рівень стабільного та передбачуваного циклу [8, 22, 77].

Вибір інструменту статичного аналізу залежить від мови програмування, цілей перевірки, масштабів проєкту та вимог до інтеграції [22, 40, 80].
Наприклад, для Java проєктів широко використовуються Checkstyle, PMD і SpotBugs, тоді як для мов сімейства C/C++ Cppcheck, Clang Static Analyzer, PVSStudio [59, 77]. У випадку з Python розповсюджені Flake8, Pylint і Bandit, а для

JavaScript/TypeScript ESLint [40, 59, 80]. Кожен з них має свої особливості: набір правил, підтримку IDE, можливості автоматичного виправлення, інтеграцію в CI/CD та гнучкість налаштувань [8, 22, 40].

Нижче узагальнено характеристики поширених інструментів для різних мов у таблиці 2.1:

Таблиця 2.1 Порівняльна характеристика поширених інструментів забезпечення якості програмного коду для різних мов програмування

Назва інструменту	Мова	Тип перевірки	Підтримка IDE	CI/CD інтеграція	Автоматичне виправлення
Pylint	Python	Синтаксис, стиль, логіка	Так	Так	Частково
ESLint	JS/TS	Синтаксис, правила, стиль	Так	Так	Так
SpotBugs	Java	Потенційні помилки, байткод	Частково	Так	Ні
PMD	Java	Стиль, дублікати, структури	Так	Так	Ні
Cppcheck	C/C++	Безпечність, стиль, помилки	Частково	Так	Ні

Варто зазначити, що ефективне використання аналізаторів вимагає врахування рівня зрілості команди, технічного боргу та особливостей кодової бази [6, 22, 59]. Наприклад, у проектах з великою кількістю застарілого коду

доцільним є поступове впровадження перевірок із нарощуванням суворості, аби уникнути перевантаження [3, 8, 22]. Також доцільно виділяти критичні типи помилок (наприклад, потенційні NullPointerException) окремим рівнем серйозності, який контролюється пріоритетно [40, 77, 80].

Деякі сучасні середовища розробки (наприклад, Visual Studio, IntelliJ IDEA, VS Code) містять вбудовані механізми статичного аналізу, які автоматично маркують помилки ще під час написання коду [22, 40, 59]. Це значно пришвидшує процес виявлення дефектів і забезпечує безперервний зворотний зв’язок [6, 8, 22]. Інтеграція таких механізмів у щоденну практику дозволяє поступово зменшувати технічний борг без радикальних змін у процесі розробки
[3, 9, 22].

У підсумку, статичний аналіз слід розглядати як невіддільну складову сучасного процесу забезпечення якості [3, 6, 22]. Він не тільки знижує ризики появи критичних помилок, а й дисциплінує команду, формує прозору культуру розробки та слугує базою для подальшого вдосконалення практик програмування [8, 22, 40]. У наступних підрозділах буде розглянуто, яким чином динамічні методи, інструменти моніторингу та інтеграційні підходи доповнюють функціональність статичного аналізу [22, 59, 80].

2.2. Порівняльний огляд ESLint, SonarLint, Pylint

Динамічне тестування є невід’ємною складовою процесу забезпечення якості програмного забезпечення, орієнтованою на виявлення помилок під час виконання коду [3, 8, 39]. На відміну від статичних методів, які працюють із вихідним текстом без запуску, динамічне тестування передбачає виконання окремих компонентів або системи загалом у контрольованих умовах [5, 20]. Його метою є перевірка фактичної поведінки програми, відповідність очікуваним результатам, а також аналіз реакції на виняткові або нетипові вхідні дані. Такий підхід дозволяє виявити не лише функціональні дефекти, а й проблеми, пов’язані з логікою, взаємодією модулів, або неправильним станом середовища виконання [4, 21].

У межах динамічного підходу тестування зазвичай класифікують за рівнями: модульне (unit testing), інтеграційне, системне та приймальне [1, 8]. Найбільш формалізованим і стандартизованим з них є модульне тестування, оскільки воно дозволяє ізолювати окремі функції або класи та перевірити їх незалежно від інших частин системи. У межах цього типу тестування розробники отримують змогу фіксувати локальні дефекти, оцінювати коректність внутрішніх обчислень, а також забезпечити стабільність коду при майбутніх змінах [9, 50]. Ізоляція тестованих одиниць досягається за рахунок використання спеціальних бібліотек, що дозволяють емулювати зовнішні виклики та залежності [80].

Інструменти для динамічного тестування найчастіше реалізовані у вигляді фреймворків, адаптованих до конкретних мов програмування. Наприклад, для мови Java використовується JUnit, який підтримує анотації, генерацію звітів, інтеграцію з Gradle і Maven. У Python популярними є PyTest та unittest, що розрізняються синтаксисом, гнучкістю та підтримкою плагінів [80]. Для JavaScript-проєктів характерне використання Jest або Mocha, які розраховані на асинхронне виконання тестів і підтримують емуляцію зовнішніх API [37, 47]. Вибір інструменту визначається як технічними особливостями проєкту, так і практиками, що склалися в команді [9, 21].

У структурованих розробницьких середовищах динамічне тестування включається у процес ще на ранніх етапах, як частина практики test-driven development (TDD) [1, 8]. У цьому випадку тести пишуться ще до реалізації функціоналу, а подальша розробка здійснюється з орієнтацією на проходження конкретних перевірок. Такий підхід знижує ризик логічних дефектів, сприяє формуванню чіткої специфікації до кожної функції та пришвидшує виявлення помилок [39, 50]. Однак ефективність TDD залежить від дисципліни виконання, узгодженості стилю кодування та вміння формалізувати очікувану поведінку ще до її реалізації [8, 9].

Перевагою інструментів динамічного тестування є їх здатність охоплювати як позитивні, так і негативні сценарії [5, 8, 21]. Розробник може

моделювати ситуації, у яких система отримує некоректні вхідні дані, обробляє винятки або стикається з ресурсними обмеженнями. Наприклад, перевірка реакції функції на відсутність очікуваного аргументу, наявність null-значення або виклик у некоректному порядку дозволяє виявити помилки, які не виявляються у звичайному потоці виконання [20, 50]. У сукупності з покриттям граничних умов це створює основу для стабільного функціонування системи у непередбачуваних обставинах [19, 62].

Функціональні можливості фреймворків істотно різняться за підтримкою додаткових функцій, зокрема таких, як параметризація тестів, організація тестових фікстур, генерація звітів у форматі HTML, інтеграція з системами збірки або CI/CD [8, 42, 80]. Наприклад, PyTest підтримує розгалужену систему фікстур і декораторів для передачі параметрів, що дозволяє реалізовувати гнучкі сценарії з мінімальним дублюванням коду [80]. Аналогічно, Jest забезпечує інтеграцію з snapshot-тестуванням, що є релевантним для перевірки інтерфейсів та шаблонів у фронтенд-розробці [37, 40]. Ці особливості уможливлюють адаптацію фреймворків під проєкти з різним рівнем складності та різною архітектурою [9,
39].

Необхідність порівняння інструментів динамічного тестування зумовлює потребу в об'єктивних критеріях. До таких належать: легкість конфігурації, підтримка різних типів тестів (unit, integration), швидкість виконання, гнучкість генерації звітів і можливість інтеграції з іншими сервісами [46, 50, 80]. У таблиці нижче представлено узагальнене зіставлення чотирьох популярних інструментів відповідно до зазначених характеристик.

Таблиця 2.2

Порівняння інструментів динамічного тестування за критеріями функціональності

Інструмент	Параметризація	Mocking	Асинхронність	Звіти HTML	Інтеграція CI/CD
PyTest	Так	Так	Обмежено	Так	Частково
JUnit	Так	Частково	Через плагіни	Так	Повна
Jest	Так	Так	Так	Так	Повна
Mocha	Так	Так	Так	Через плагін	Через middleware

На основі таблиці 2.2 можна зробити припущення щодо придатності кожного фреймворка для певних типів проєктів. Наприклад, JUnit краще адаптований до корпоративного стеку Java [80], тоді як Jest виявляє ефективність у високочастотних інтерактивних застосунках [37, 40]. Водночас Mocha демонструє гнучкість при створенні кастомних тестових сценаріїв, однак потребує ручної конфігурації для отримання повного звіту [9, 80]. PyTest, своєю чергою, балансує між простотою використання та широким функціоналом для невеликих команд Python-розробників [8, 80].

Ізольоване тестування функцій за допомогою mocking-механізмів дозволяє нівелювати вплив сторонніх модулів або нестабільних залежностей [5, 21]. Зокрема, бібліотеки типу unittest.mock у Python або Mockito у Java активно використовуються для симуляції поведінки зовнішніх API, баз даних або файлових систем [80]. Це дає змогу перевіряти лише цільову логіку без прив’язки до зовнішнього середовища. В результаті підвищується відтворюваність тестів і зменшується час їх виконання [39, 50]. Проте надмірне використання емульованих залежностей може призвести до відриву тестів від реального функціонування, що потребує балансування [13, 19].

Окремої уваги потребує підтримка асинхронного виконання, особливо

для веборієнтованих застосунків або мікросервісної архітектури [40, 54]. Інструменти на кшталт Jest або Mocha з вбудованою підтримкою async/await дають змогу ефективно тестувати API, зворотні виклики (callbacks), WebSocketз'єднання тощо [37, 47]. Натомість JUnit вимагає підключення додаткових бібліотек або застосування шаблонів для досягнення аналогічної функціональності [80]. У Python реалізація асинхронних тестів підтримується лише з версій PyTest, які мають специфічні плагіни, що обмежує backwardcompatibility з деякими версіями середовищ [8, 80].

Інтеграція інструментів динамічного тестування з CI/CD-платформами (наприклад, GitLab CI, Jenkins, GitHub Actions) є визначальним чинником для автоматизації контролю якості [42, 80]. Вона дає змогу запускати тести автоматично після кожного коміту або перед розгортанням на продуктивному середовищі. Такий підхід підвищує стабільність продукту та дозволяє швидко реагувати на регресії [21, 39]. Для забезпечення сумісності інструментів із CIсередовищем необхідно передбачити наявність конфігураційних скриптів, коректне формування статусів виконання та експортування результатів у машинозчитуваному форматі (наприклад, JUnit XML) [8, 80]. У цьому контексті перевагу мають ті фреймворки, які підтримують стандартизовану структуру звітності [46, 50].

Звіти про тестування можуть бути як лаконічними (наприклад, лише статистика успішних/невдалих тестів), так і деталізованими з логами, графіками та підсвічуванням помилок. Інструменти на кшталт Allure або HTMLTestRunner дозволяють візуалізувати результати у вигляді інтерактивних звітів, що прискорює аналіз дефектів і полегшує комунікацію в команді [8, 39]. Застосування таких рішень особливо виправдане у великих проєктах з декількома підсистемами, де обсяг тестів перевищує тисячі випадків. Проте обробка великих звітів потребує ефективного керування ресурсами CI-серверів і розподілу навантаження [42, 80].

У процесі тестування зростає потреба у валідації не лише правильності обчислень, але й поведінкових характеристик: час відповіді, використання

пам’яті, стабільність при навантаженні. Деякі інструменти, наприклад, JUnit Performance або PyTest-benchmark, дозволяють інтегрувати в тестовий процес метрики продуктивності [9, 50]. У результаті забезпечується контроль не лише над логікою виконання, а й над нефункціональними характеристиками програмного забезпечення [8, 62]. Застосування таких механізмів дає змогу своєчасно виявляти потенційні вузькі місця в архітектурі або надмірне споживання ресурсів [3, 19].

Незважаючи на численні переваги, інструменти динамічного тестування мають обмеження. Вони не завжди дозволяють охопити всі гілки логіки, особливо у складних умовних конструкціях або при великій кількості залежностей [4, 13]. Крім того, тести, написані заднім числом, можуть містити приховану упередженість, оскільки перевіряють уже відому поведінку [21, 39]. Надмірне покриття без належного аналізу доцільності призводить до зростання технічного боргу, ускладнюючи підтримку тестової бази [9, 50]. Це створює потребу у системному підході до структурування тестів та визначення пріоритетів [1, 8].

Підсумовуючи, динамічне тестування виступає важливим компонентом забезпечення якості, який потребує узгодженого вибору інструментів, продуманого планування та системної інтеграції в розробницький процес [3, 39, 80]. Порівняльний аналіз показує, що жоден із фреймворків не є універсальним; натомість доцільним є формування комбінованої екосистеми інструментів, адаптованої до особливостей проєкту, мови реалізації та вимог до підтримки [46, 50]. Від цього залежить не лише повнота охоплення тестами, але й витрати на супровід, швидкість і надійність розгортання кінцевого продукту [8, 21, 42].

2.3. Інтеграція в середовище розробки (VS Code / PyCharm)

Автоматизоване тестування належить до базових практик забезпечення якості програмного забезпечення в проєктах зі швидким циклом розробки або значним обсягом функціональності [1, 8, 39]. Його застосування дозволяє зменшити навантаження на тестувальників, скоротити час зворотного зв’язку

й знизити ймовірність людської помилки [5, 21]. У межах цього підходу виконання тестів передається програмним скриптам або фреймворкам, що забезпечує відтворюваність і сталість результатів [3, 46]. Основу складають набір тесткейсів, які автоматично перевіряють коректність роботи модулів або системи в цілому, не вимагаючи безпосередньої участі людини [8, 50].

Інструменти автоматизованого тестування умовно поділяються на три основні типи: фреймворки для модульного тестування (наприклад, NUnit, TestNG), засоби функціонального або UI-тестування (наприклад, Selenium, Cypress), а також платформи для end-to-end тестування (наприклад, Robot Framework, TestComplete) [9, 39, 80]. Кожен тип має свої переваги та обмеження, зумовлені характером застосування та рівнем інтеграції в життєвий цикл розробки. Подібна класифікація є корисною для коректного вибору інструменту залежно від технічного стеку проєкту, досвіду команди та вимог до тестового покриття [21, 46].

Варто окремо зосередитися на Selenium як одному з найбільш відомих інструментів для автоматизації браузерних інтерфейсів. Його архітектура побудована навколо принципу контролю браузера через драйвери (WebDriver API), що забезпечує кросбраузерність і можливість паралельного тестування [8, 80]. У поєднанні з мовами програмування (Java, Python, JavaScript) Selenium дозволяє реалізовувати складні сценарії з перевіркою UI-елементів, навігації, взаємодії з формами. Проте конфігурація тестів може бути складною для початківців, особливо без підключення додаткових бібліотек для assertion чи репортингу [39, 50].

На противагу цьому Cypress пропонує більш лаконічний підхід до організації автоматизованих UI-тестів [40, 80]. Його особливість полягає у повній інтеграції тестового рантайму в браузер, що дає змогу отримувати миттєвий зворотний зв’язок та спостерігати за виконанням тестів у режимі реального часу. Cypress підтримує REST-запити, мокінг, маніпуляцію DOMелементами, що значно прискорює написання тестів [37, 40]. Однак він працює лише з браузерами на основі Chromium і не підтримує багатьох

зовнішніх мов, обмежуючи екосистему до JavaScript/TypeScript. Ця обмеженість компенсується гнучкістю для фронтенд-розробників [40, 47].

Для сценаріїв типу end-to-end тестування, які охоплюють повний шлях користувача від старту до завершення бізнес-процесу, зручними є фреймворки на кшталт Robot Framework або TestComplete [8, 9, 80]. Вони орієнтовані на розробку тестів у формі структурованих сценаріїв із мінімальним кодом, що спрощує участь аналітиків або фахівців без глибокої технічної підготовки [39, 46]. Автоматизація таких перевірок дозволяє охоплювати складні інтегровані процеси, включно з автентифікацією, роботою з БД, API та зовнішніми сервісами [50, 54]. Утім, налаштування середовища та ліцензійні обмеження можуть ускладнювати їх масштабування в умовах обмеженого бюджету [80].

Паралельно з традиційними фреймворками активно розвиваються lowcode / visual testing-платформи, які орієнтовані на бізнес-користувачів або аналітиків без досвіду програмування [40, 80]. До таких рішень належать, зокрема, Katalon Studio, Testim, Ranorex. Вони надають можливість створювати автоматизовані тести за допомогою графічного інтерфейсу, зберігаючи при цьому підтримку скриптових мов для складніших кейсів [39]. Основна перевага полягає в пришвидшенні створення та модифікації тестів, особливо при роботі над прототипами або MVP [9, 50]. Водночас закритість коду та залежність від платформи можуть обмежити довготривалу гнучкість [46, 80].

Інструменти автоматизації підтримують інтеграцію з системами керування тестуванням (наприклад, TestRail, Zephyr), що забезпечує централізований облік, планування, моніторинг виконання та верифікацію результатів [42, 80]. Така інтеграція дає змогу пов’язати тести з вимогами та дефектами, створити ієрархію тестів, генерувати звіти за релізами [8, 21]. Це особливо актуально в масштабних проєктах, де паралельно працюють декілька команд і контроль над якістю стає критичним чинником стійкості розробки. Для інтеграції застосовуються API, плагіни до Jira або Jenkins, а також адаптери до CI/CD-систем [42, 46, 80].

Вибір середовища для запуску автоматизованих тестів залежить від

контексту: для перевірки інтерфейсів зазвичай використовуються headlessбраузери (наприклад, Chrome Headless, Puppeteer), в той час як APIтести виконуються через Postman Newman або REST Assured [40, 47, 54]. На практиці поєднання інструментів дозволяє створити повноцінний стек автоматизації з перевіркою фронтенду, бекенду та інтеграцій [39, 80]. Усі ці рівні можуть бути пов’язані в єдиному пайплайні, що автоматизує повний цикл перевірки змін і знижує ймовірність регресій [42, 46].

Тестові сценарії в автоматизованому середовищі повинні бути модульними, повторно використовуваними й піддаватись легкій модифікації [8, 21]. Цьому сприяє застосування шаблонів проєктування (наприклад, Page Object Model), які структурують код, мінімізують дублювання та підвищують масштабованість [9, 39]. Для перевірки складних логік рекомендується сегментувати тести на логічні блоки, із забезпеченням незалежності між ними. Подібний підхід дозволяє ізолювати причину збою в разі помилки, пришвидшує діагностику та полегшує супровід проєкту [50, 80].

Узагальнені відомості про найбільш поширені інструменти автоматизованого тестування наведено в таблиці нижче. Вона охоплює тип інструменту, підтримувані мови, платформу, формат звітності та ключові особливості [46, 50].

Таблиця 2.3

Порівняльна характеристика інструментів автоматизованого тестування

Назва інструменту	Тип тестування	Підтримувані мови	Платформи	Формат звітності	Особливості використання
Selenium	UI, інтеграційне	Java, Python, C#, JS	Windows, Linux, macOS	HTML, JUnit, Allure	Гнучкий API, підтримка WebDriver, потребує конфігурації
Cypress	UI, end-toend	JavaScript, TypeScript	Windows, macOS	Візуальний, JSON	Швидкий запуск, інтеграція з браузером
TestComplete	UI, функціональне	JavaScript, Python, VBScript	Windows	Докладний HTML	Платний, підтримує Desktop додатки
Robot Framework	end-to-end, API, UI	Python (через бібліотеки)	Windows, Linux, macOS	XML, HTML, лог-файли	Зручний синтаксис, легко читається, підтримує BDD
Katalon Studio	UI, API, мобільне	Groovy, Java	Windows, macOS	PDF, HTML, інтеграція з CI	Low-code інтерфейс, підтримка мобільних платформ

Окрему категорію становлять інструменти для тестування мобільних додатків, зокрема Appium, Espresso, XCUITest [80]. Вони підтримують автоматизацію взаємодії з UI-елементами на Android і iOS-пристроях, забезпечуючи гнучкість для кросплатформених продуктів [39]. Appium, наприклад, використовує WebDriver-протокол, що дозволяє тестувати мобільні застосунки аналогічно до вебдодатків у Selenium [9, 80]. Хоча конфігурація середовища може бути складнішою, ефективність автоматизації значно підвищується у великих проєктах із частими змінами у мобільних клієнтах [50].

Ще один напрям автоматизації стосується API-тестування, що реалізується засобами Postman, REST Assured, SoapUI [40, 46, 80]. Вони

дозволяють здійснювати перевірку відповідей серверу, заголовків, статускодів, структури JSON- або XML-вмісту, підтримують тестові сценарії на основі запитів. У середовищах із мікросервісною архітектурою подібне тестування є основою контролю коректності інтеграції між компонентами [8, 54]. REST Assured використовується переважно в Java-проєктах, тоді як Postman забезпечує гнучкий інтерфейс із можливістю генерації звітів та інтеграції з Newman [39, 42].

Проблематика автоматизованого тестування пов’язана не лише з вибором інструменту, а й з коректною організацією тестової бази [1, 5]. Одним із викликів є підтримка актуальності тестових сценаріїв після змін у програмному коді [21, 50]. Використання Git для зберігання тестів, поєднане з CI-сценаріями (наприклад, запуск тестів після кожного коміту), забезпечує контроль регресій [42, 46]. Проте такий підхід вимагає формалізованого підходу до структури тестів, версіонування та документування [9, 80].

Крім автоматизації виконання, деякі інструменти надають функціонал для моніторингу продуктивності (наприклад, LoadRunner, JMeter) [46, 50, 80]. Хоча формально вони не належать до класичних систем автоматизованого тестування, їх інтеграція дозволяє оцінити поведінку системи під навантаженням та виявити потенційні вузькі місця [3, 62]. У випадках, коли програмний продукт орієнтований на велику кількість користувачів або працює в реальному часі, такі перевірки можуть бути включені до пайплайну з автоматизованими UI- і APIтестами [39, 42].

Підсумовуючи, варто зазначити, що автоматизоване тестування не є універсальним розв’язанням усіх задач контролю якості [1, 21]. Його ефективність залежить від грамотного вибору інструментарію, архітектури тестової бази, дисципліни підтримки сценаріїв і культури розробки загалом [5, 9, 50]. Найкращі результати досягаються у випадках, коли тестування є не окремим етапом, а інтегрованою частиною всього життєвого циклу ПЗ, що постійно адаптується до змін і масштабування [8, 46, 80].

2.4. Приклади результатів (знімки екранів, порівняння логів аналізу)

Поняття тестового покриття охоплює сукупність метрик, що характеризують ступінь перевіреності програмного коду під час виконання тестових сценаріїв [8, 21]. Залежно від рівня деталізації, покриття може вимірюватися на рівні операторів, гілок, умов, шляху виконання [1, 50]. Поширеним є використання метрик statement coverage, branch coverage, condition coverage, які дозволяють оцінити ефективність наявного тестового набору [3, 9, 46]. Незважаючи на очевидну користь таких метрик, їхнє тлумачення не завжди однозначне: високі значення можуть маскувати відсутність логічного контролю складних випадків [13, 80].

Засоби контролю тестового покриття дозволяють автоматизувати збір інформації про те, які частини коду були виконані під час проходження тестів [39, 80]. Програмні агенти або інструменти інтегруються в процес виконання та фіксують відвідані рядки, умовні переходи, виклики функцій. Ці дані візуалізуються у вигляді звітів або графів, що дає змогу аналітику оперативно виявити «мертвий код» або ділянки, які залишилися поза увагою [9, 50]. Типовим прикладом є генерація HTML-звітів із підсвічуванням рядків коду, які були або не були охоплені тестами [40, 80].

Серед інструментів, що підтримують аналіз тестового покриття, можна виокремити Jacoco (для Java), Coverage.py (для Python), Istanbul (для JavaScript/TypeScript), OpenClover (також Java), а також gcov (C/C++) [80]. Вибір інструмента залежить від мови програмування, системи збірки, формату бажаних звітів і потреб CI/CD-процесу [42, 46]. Наприклад, Jacoco легко інтегрується з Maven або Gradle і підтримується Jenkins, тоді як Coverage.py адаптовано до pytest і сумісний із coverage.xml-форматом, який читають різноманітні CIсистеми
[8, 50].

Загальновживаним форматом представлення результатів є структура linebyline з підрахунком охоплених рядків та відсотковим значенням [39, 80]. Проте більш глибокий аналіз вимагає врахування складних гілок і логічних

умов. Наприклад, одна умовна конструкція може бути формально охоплена, але не всі можливі переходи по гілках будуть зафіксовані без спеціально сконструйованих тестів [3, 62]. Це породжує потребу в комбінуванні кількох типів покриття та критичній оцінці наявних результатів, а не лише формальному досягненні заданого порогу [5, 9].

Наявність високого відсотка покриття не гарантує відсутності помилок або логічних вад у програмному забезпеченні [3, 9]. Частина тестів може бути тривіальною, не перевіряти неочевидні сценарії або повторювати однакові шаблони [1, 50]. Наприклад, модуль із числовим розгалуженням може бути покритий лише позитивними значеннями, залишаючи від'ємні чи граничні ситуації поза увагою [13, 62]. Тому контроль покриття повинен поєднуватися з аналізом якості самих тестів, включаючи варіативність даних, перевірку винятків і негативних сценаріїв [8, 21].

Деякі інструменти підтримують комбіноване покриття, що включає як юніт-тести, так і інтеграційні [42, 80]. Це дозволяє відстежити, наскільки ефективно перевіряються компоненти системи в ізоляції та під час взаємодії. Наприклад, у випадку REST API можна вимірювати, які методи контролера охоплені тестами окремо, а які під час системної перевірки на рівні HTTP [39, 54]. Такий підхід підвищує інформативність метрик, однак вимагає окремої конфігурації, оскільки змішування даних із різних рівнів тестування може призвести до хибних інтерпретацій [46, 80].

Інтеграція засобів покриття в CI/CD-процеси сприяє оперативному виявленню зниження якості тестів [42, 46]. Наприклад, при кожному коміті в репозиторій можна запускати автоматизоване вимірювання покриття, з порівнянням із попереднім станом. У разі падіння показника нижче допустимого порогу (наприклад, 80%) розгортання блокується [21, 50].
Подібна практика підвищує дисципліну розробки, але водночас може стимулювати штучне підвищення метрики за рахунок тривіальних тестів, що не мають перевірочної цінності [3, 80].

Щоб уникнути поверхневого підходу, окремі інструменти впроваджують

механізми підрахунку mutational coverage [9, 50]. Йдеться про мутаційне тестування, коли спеціальний фреймворк вносить незначні зміни в код (мутації) та перевіряє, чи виявляються вони існуючими тестами [13, 46]. Якщо мутація не вловлюється це сигнал про слабкість тестового набору. Приклади таких рішень: PIT (для Java), Mutmut (для Python), Stryker (для JavaScript/TypeScript) [80]. Вони значно підвищують точність оцінювання тестової ефективності, хоча й потребують більших обчислювальних ресурсів [62, 80].

Слід враховувати, що збір даних про покриття часто впливає на продуктивність системи [3, 62]. У реальному часі профілювання може сповільнити виконання, особливо у великих проєктах із великою кількістю тестів
[8, 50]. Це створює дилему між точністю оцінки та ефективністю. У зв'язку з цим покриття зазвичай вимірюють на окремому етапі, у спеціальному середовищі, що симулює робочі умови без навантаження на основну систему [46, 80].

На практиці комбінування класичних метрик покриття з контекстною інформацією про зміни в коді забезпечує адаптивний підхід до контролю якості [42, 80]. Наприклад, якщо певний клас було оновлено, але не з'явилося жодного нового тесту, система фіксує «зону ризику». Таку функціональність реалізують, зокрема, SonarQube та Codecov, які поєднують дані про покриття, git-історію змін і CI-звіти [77, 80]. Цей підхід дозволяє зосередити увагу не на загальному відсотку, а на критичних ділянках коду, що були змінені, але не перевірені [50, 77].

Питання інтерпретації покриття залишається відкритим у випадках, коли код генерується автоматично або містить великі обсяги шаблонних фрагментів [3, 13]. Часто такі ділянки або не потребують тестування (наприклад, автозгенеровані сеттери/геттери), або їхня перевірка не додає інформаційної цінності.

Проте вони враховуються під час підрахунку покриття, що викривлює

загальний показник [46, 62]. Щоб уникнути цієї проблеми, більшість інструментів дозволяють маркувати певні файли або блоки коду як «винятки», які виключаються з розрахунку метрик [8, 80].

Не менш значущим є питання релевантності покриття у контексті реального користувацького сценарію [1, 21]. Високе покриття не гарантує перевірки тих функціональностей, які є критично значущими для користувача. Наприклад, APIконтролер може мати 95% покриття, але жоден тест не перевіряє обробку недопустимих параметрів або специфічних edge-caseвипадків [39, 50]. Це створює ілюзію безпеки та стабільності системи. Тому аналітики рекомендують комбінувати покриття з аналізом ризиків і пріоритезацією тестових сценаріїв [46,80].

Серед перспективних напрямів розвитку інструментів покриття інтеграція зі штучним інтелектом і навчальними моделями [53, 67, 78]. В окремих проєктах розробляються фреймворки, які на основі патернів коду та історії дефектів прогнозують ймовірні «зони дефіциту покриття» [44, 78]. Це дозволяє автоматично формувати рекомендації для створення нових тестів. Хоча ці підходи поки що залишаються експериментальними, вони свідчать про трансформацію класичного аналізу в бік інтелектуалізованих систем підтримки якості [51, 53,78].

У табличному вигляді нижче узагальнено особливості найбільш використовуваних інструментів контролю тестового покриття для популярних мов програмування. Інформація подана з огляду на формат звітів, інтеграцію з CIсистемами та підтримку комбінованого покриття.

Таблиця 2.4

Характеристики інструментів контролю тестового покриття для різних мов програмування

Інструмент	Мова	Формат звітів	CIінтеграція	Підтримка гілок	Особливості
Jacoco	Java	XML, HTML	Jenkins, GitLab	Так	Інтеграція з Maven/Gradle
Coverage.py	Python	XML, HTML	GitHub Actions	Обмежено	Сумісність із pytest
Istanbul	JS/TS	lcov, JSON	GitHub, Travis	Так	Добра візуалізація
gcov	C/C++	текстовий	Make, CMake	Ні	Потрібен парсинг вручну
OpenClover	Java	XML, HTML	Bamboo	Так	Можливість фільтрації класів

Загалом, інструменти контролю покриття коду не варто сприймати як кінцевий орієнтир ефективності тестування. Вони слугують діагностичним інструментом, який у поєднанні з іншими підходами дозволяє формувати повнішу картину про якість програмного забезпечення. Адекватне застосування метрик покриття потребує аналітичного осмислення, балансування між кількісними й якісними характеристиками та критичної оцінки результатів у прив’язці до бізнес-логіки та ризиків.

Для підсумкового порівняння ключових характеристик розглянутих інструментів доцільно узагальнити їх відмінності у табличній формі (див. Табл.2.5).

Таблиця 2.5 Порівняльна характеристика інструментів забезпечення якості коду

Критерій	Pylint	SonarLint	SonarQube
Тип аналізу	Статичний	Статичний	Статичний збір метрик +
Мова реалізації	Python	Java, JavaScript, Python, ін.	Java, JS, C/C++ Python, ін.
Інтеграція в IDE	Так (VS Code, PyCharm)	VS Так (Eclipse, Code, IntelliJ)	Обмежена (через сервер)
Підтримка CI/CD	Ні	Ні	Так
Візуалізація результатів	Консоль	IDE-панель	Web-інтерфейс із графіками
Можливість масштабування	Низька	Середня	Висока
Вимоги до інфраструктури	Мінімальні	Мінімальні	Високі (потрібен сервер)
Вартість	Безкоштовний	Безкоштовний	Є безкоштовна версія; повна платна

ВИСНОВОК ДО РОЗДІЛУ 2

У межах другого розділу було здійснено цілісний огляд основних інструментів, що використовуються для підвищення якості програмного забезпечення. Розгляд почався з аналізу засобів статичного аналізу коду, які дозволяють виявити помилки без запуску програми. Було встановлено, що принцип їхньої дії ґрунтується на побудові абстрактного синтаксичного дерева, а також перевірці на відповідність правилам, шаблонам і стандартам. Інструменти цієї категорії ефективні для виявлення синтаксичних помилок, недоцільних конструкцій, дублікатів та порушень конвенцій.

Огляд динамічних інструментів продемонстрував, що їх використання базується на аналізі поведінки програмного продукту під час виконання. Такі засоби дозволяють фіксувати проблеми, які не завжди видно на рівні коду: витоки пам’яті, помилки синхронізації, порушення потокової безпеки. На відміну від статичних засобів, динамічні потребують середовища виконання, що зумовлює більшу витратність ресурсів, але забезпечує інший тип діагностики, доповнюючи аналіз.

Розділ також охопив інструменти контролю тестового покриття, які дозволяють оцінити ступінь охоплення коду тестами. Було показано, що ці засоби не лише допомагають виявляти слабкі місця у тестовій базі, а й слугують індикатором стабільності архітектури. За допомогою аналізу покриття можна визначити надлишкові або пропущені частини коду, що потребують уваги. У таблицях було узагальнено функціональні характеристики найпоширеніших рішень, таких як JaCoCo, Coverage.py,
Istanbul.

Крім технічного опису, у розділі простежено обмеження кожного з розглянутих підходів. Зокрема, встановлено, що жоден інструмент не забезпечує вичерпної перевірки якості, а ефективність залежить від типу застосунку, мови програмування, особливостей архітектури й практик команди. Багато з інструментів потребують налаштування, адаптації під конкретні середовища розробки, а деякі генерують велику кількість

хибнопозитивних результатів, що потребує критичної інтерпретації.

Узагальнення матеріалу дозволяє зробити припущення про ефективність комбінованого застосування різних типів інструментів. Оскільки кожен з них спрямований на діагностику різних рівнів і аспектів якості, доцільним є створення інтегрованої системи, що поєднує переваги кількох підходів. У цьому контексті перспективною є розробка автоматизованих пайплайнів забезпечення якості, де перевірки відбуваються на кожному кроці життєвого циклу програмного продукту.

Результати аналізу вказують на потребу не лише у технічному впровадженні зазначених засобів, але й у зміні організаційної культури розробки: лише у поєднанні з системним навчанням, чіткою політикою якості й аналітичним супроводом результати перевірок можуть бути використані ефективно. Це створює підґрунтя для подальшої практичної частини дослідження, що розкривається у третьому розділі.

РОЗДІЛ 3 ПРАКТИЧНА ДЕМОНСТРАЦІЯ

3.1. Вибір фрагмента коду з помилками

Вибір фрагмента коду для практичного аналізу визначається необхідністю показати типовий приклад програмного дефекту, який водночас є неочевидним і здатним призвести до серйозних наслідків у роботі програмного забезпечення [3, 8, 21]. На першому етапі доцільно сфокусуватися на коді, що містить логічні помилки, помилки обробки винятків або дефіцит валідації вхідних даних [9, 50]. Подібні порушення, як правило, не виявляються стандартними тестами, оскільки не викликають негайного збою програми, але генерують латентні баги [13, 46]. Джерелом фрагмента виступає відкритий репозиторій на GitHub, який містить приклад вебсервісу на Python із застосуванням Flask [38, 70].

Розглянутий код виконує обробку HTTP-запитів до REST API, зокрема, обробку POST-запиту, який очікує на вхід JSON-дані про користувача (ім’я, вік, електронна пошта). Основна функція process_user_data відповідає за зчитування даних, валідацію та збереження у внутрішню структуру [40, 54]. Проте попри зовнішню простоту, реалізація містить щонайменше чотири потенційні джерела помилок. Вони пов’язані зі слабкою перевіркою типів даних, відсутністю обробки невалідних значень, допущенням до роботи з неініціалізованими змінними та порушенням логіки умови перевірки ключів у JSON [8, 50].

Першим симптомом неякісної реалізації є надмірна довжина функції, яка виконує одночасно читання, перевірку, обробку помилок і логування [3, 9]. Це унеможливлює її повторне використання та ускладнює написання модульних тестів [5, 21]. Функціональне перевантаження не дозволяє локалізувати дефект у разі збою. Крім того, порушено принцип єдиної відповідальності одна з базових вимог до чистої архітектури [13, 46]. Цей недолік є типовим для

початкових реалізацій, які не проходили фазу рев’ю або не покривалися статичним аналізом
[50, 80].

Другим виявленим дефектом є некоректна логіка перевірки наявності обов’язкових ключів у вхідному словнику [9, 54]. Замість прямої перевірки з використанням конструкції if 'name' in data, автор реалізував перевірку на тип str та довжину рядка, що в контексті Python може дати хибнопозитивний результат, якщо поле є, але містить None [70]. Як наслідок, система допускає обробку некоректних об’єктів, що порушує передумову подальших дій. Подібна помилка не виявляється під час поверхневого ручного тестування, проте легко фіксується за допомогою покриття тестами з edge-case-випадками [3, 50].

Окрему загрозу становить необроблений виняток, пов’язаний із передачею до функції даних у форматі, що відрізняється від очікуваного JSON [8, 54]. Якщо клієнт надсилає POST-запит із некоректним заголовком ContentType або malformed JSON, інтерпретатор викликає помилку BadRequest, яка не обробляється та призводить до завершення обробки запиту без інформативного повідомлення для клієнта [70, 40]. Це створює вразливість до DoS-атаки з мінімальними зусиллями. Вказана проблема є типовою для API, що не мають захисту від невалідного вводу на мережевому рівні [9, 80].

Ще однією проблемною ділянкою виявився блок, де проводиться арифметична операція на основі параметра «вік», який передається у вигляді рядка [8, 21]. У функції не передбачено явного приведення типу, що стає джерелом виключення TypeError, коли програма намагається, наприклад, додати до рядка число або виконати логічне порівняння [50, 54]. Проблема загострюється відсутністю тестів на негативні сценарії, зокрема, при отриманні рядкових або від’ємних значень [3, 46]. Така недбалість під час валідації даних на вході поширене джерело невідловлених помилок на продакшн-середовищі [39, 70].

Додатково слід зауважити, що в коді не використовується

централізований механізм логування [1, 13]. Усі повідомлення, включно з помилками, виводяться в консоль, що унеможливлює подальший аудит або збирання метрик. Вирішенням могла б бути інтеграція з логером, наприклад, logging.getLogger(__name__), який дозволяє встановлювати рівні повідомлень і маршрутизувати їх у зовнішні сервіси [40, 47]. Недостатня спостережуваність у коді з помилками не тільки ускладнює їх виявлення, а й порушує принцип трасованості змін і дій системи [9, 62].

Варто звернути увагу на те, що навіть за наявності базового набору юніттестів основні дефекти можуть залишатися непоміченими, якщо ці тести були побудовані виключно на позитивних сценаріях [1, 3, 9]. У випадку розглянутого фрагмента не охоплено тестами ситуації з відсутніми полями, типологічними помилками, граничними значеннями параметрів, зокрема нульовим віком або порожніми рядками [39, 50]. Це вкотре доводить необхідність застосування практики TDD (test-driven development), де формування тестів передує імплементації [8, 21].

Із практичного боку, вибраний приклад дає змогу продемонструвати ефективність статичного аналізу, який здатен виявити щонайменше два типи проблем: використання неініціалізованих змінних та ігнорування виключень [5, 46]. Інструменти на кшталт pylint, flake8, bandit дозволяють в автоматичному режимі ідентифікувати потенційні проблеми ще до запуску коду [80]. За результатами перевірки фрагмента process_user_data, інструменти виявили понад 12 попереджень різного ступеня критичності, з яких більшість стосувалася стилістичних відхилень та порушення принципів безпеки [28, 80].

Аналіз показує, що типові помилки виникають унаслідок поєднання трьох чинників: відсутності вхідної валідації, недотримання принципів структурування коду та неінформативного логування [3, 13, 50]. Усі ці фактори можна виявити і частково усунути на ранніх етапах, якщо в процес розробки

інтегрувати аналіз коду, автоматизоване тестування і покриття [9, 21, 42]. Тому обраний фрагмент має не лише інструментальну, а й методичну цінність: він демонструє, яким чином ізольовані помилки структуруються в комплексну вразливість [46, 80].

Окремо варто підкреслити недоліки, пов’язані з відсутністю структури для повернення відповідей API [54, 70]. У наведеному прикладі відповідь у разі помилки повертається у вигляді сирого рядка без коду статусу або узгодженого JSON-формату [40, 77]. Це ускладнює взаємодію з фронтендом, знижує передбачуваність поведінки і створює фрагментований досвід користувача. У рамках загального підходу до якості ПЗ подібна недоопрацьованість розглядається як порушення інтерфейсного контракту, що суперечить принципу fail-fast та ідеології REST [50, 53].

Фрагмент також містить порушення стилістики іменування змінних та функцій [5, 13]. Наприклад, змінна x використовується без жодного контекстного змісту, що знижує читабельність і ускладнює розуміння логіки, особливо при масштабуванні проєкту [21, 39]. Системи статичного аналізу вказують на подібні порушення як на code smell, тобто ознаку прихованої проблеми, що не є помилкою сама по собі, але сигналізує про потребу в рефакторингу [46, 80]. Досвідчені команди дотримуються внутрішніх namingconventions, які
закріплюються в lint-конфігураціях і забезпечують уніфікованість стилю [28, 40].

Загалом розглянутий фрагмент є репрезентативним для невеликих проєктів, які розробляються в умовах обмеженого часу або відсутності командної взаємодії [3, 9]. Саме в таких умовах зазвичай ігноруються стандарти якості, документація, тестове покриття, а також інструменти перевірки [1, 50]. Проте саме ці аспекти є критично важливими для запобігання технічному боргу, який акумулюється на ранніх етапах і перетворюється на

системну проблему в майбутньому [8, 62]. Цей висновок підтверджується результатами аудиту відкритого коду, який часто демонструє подібну динаміку [77, 80].

З технічної точки зору, усунення виявлених дефектів не потребує значних витрат, проте вимагає зміни підходу до написання коду [13, 21]. Удосконалення функції process_user_data має починатися з її декомпозиції, впровадження кастомного обробника помилок, рефакторингу логіки валідації та уніфікації формату відповіді [40, 54]. Водночас слід забезпечити відповідність до принципів SOLID, а також розробити базовий набір модульних тестів [9, 50]. Така трансформація сприятиме не лише покращенню поточної реалізації, а й слугуватиме базою для масштабування [46, 80].

Підсумовуючи, обраний приклад демонструє, що навіть обмежений за обсягом фрагмент коду може містити багато різнопланових дефектів, які ускладнюють супровід, інтеграцію й експлуатацію програмного продукту [3, 39, 50]. У процесі аналізу підтвердилася доцільність використання статичних інструментів та структурного мислення під час огляду [5, 46]. Подальші етапи роботи передбачають фіксацію помилок, валідацію змін за допомогою аналізаторів та інтеграцію виправленого фрагмента до проєкту, що буде розглянуто в наступних підпунктах [8, 42, 80].

3.2. Аналіз результатів статичної перевірки

Аналіз обраного фрагмента коду за допомогою статичних інструментів було здійснено з використанням трьох незалежних систем: pylint, flake8 та bandit [4, 5, 8]. Кожна з них має власні критерії оцінки якості, різну глибину перевірки та інтерпретацію порушень [5, 8]. Такий підхід дозволяє зіставити результати й зменшити ризик хибнопозитивних спрацювань [3, 6]. Сканування проводилося в локальному середовищі із типовими налаштуваннями, які відповідають рекомендаціям для невеликих Python-проєктів [2, 4].

Результати, отримані за допомогою pylint, засвідчили наявність 16 зауважень, з яких 6 були класифіковані як «convention» (відхилення від стилістичних правил), 4 як «refactor» (неоптимальні структури), 5 як «warning» (можливі помилки виконання), і 1 як «error» [4]. Найбільшу кількість попереджень було зосереджено навколо неконсистентних імен змінних, дублювання коду, а також недокументованих функцій [3, 4]. Код отримав загальний рейтинг 4.38/10, що є характерним для сирих фрагментів, не призначених для продакшн-застосування [2].

У випадку з flake8, який орієнтовано переважно на стиль і відповідність стандарту PEP8, було зафіксовано 11 порушень, серед яких перевищення максимальної довжини рядка, відсутність відступів перед визначенням функції та змішування табуляції з пробілами [5]. flake8 не виявляє помилок логіки або небезпек на рівні безпеки, однак є цінним у початковому етапі рефакторингу [4, 5]. Ці результати дозволяють швидко уніфікувати структуру коду та усунути грубі стилістичні відхилення, що полегшує подальшу роботу в команді [3].

Тестування через bandit дозволило виявити дві потенційно небезпечні практики. По-перше, використання функції eval() для динамічного виконання рядків, що створює вразливість до ін’єкцій коду [8]. По-друге, зчитування даних без перевірки типу вхідного значення проблема, яка може бути використана для реалізації атак типу DoS або введення некоректних параметрів [6, 8]. Ці спрацювання мають високий рівень пріоритетності, оскільки можуть спричинити не лише збій програми, а й порушення конфіденційності чи цілісності системи
[7, 8].

Для візуального представлення результатів було сформовано таблицю, у якій узагальнено всі попередження, класифіковані за інструментом, типом і рівнем серйозності:

Таблиця 3.1

Класифікація виявлених попереджень за інструментами, категоріями та прикладами помилок

Інструмент	Категорія	Кількість	Приклад помилки
pylint	warning	5	Використання невизначеної змінної
pylint	convention	6	Неправильне іменування функцій
flake8	style	11	Перевищено довжину рядка
bandit	security	2	Використання eval(), open() без with

Після ідентифікації дефектів було здійснено поетапне усунення помилок із подальшим повторним аналізом для кожного інструменту. Результати повторної перевірки показали суттєве зниження кількості попереджень: pylint оцінив оновлений код на 9.17/10, flake8 виявив лише два незначні стилістичні недоліки, тоді як bandit не зафіксував жодного ризикованого виклику [4, 5, 8]. Цей етап продемонстрував ефективність циклу «аналіз–рефакторинг–контроль» як інструмента підвищення якості [2, 6].

Аналіз специфіки помилок дозволяє зробити припущення щодо типових причин їх виникнення. Наприклад, стилістичні порушення здебільшого спричинені відсутністю lint-конфігурацій на етапі початкового створення проєкту, що свідчить про брак інструментальної культури [5, 6]. Логічні недоліки, зокрема неочікуване використання eval(), могли бути внесені для спрощення перевірок у тестовому середовищі, однак їх наявність у продуктивному коді вказує на нехтування безпековими принципами [7, 8]. Це ще раз актуалізує потребу в етапі статичного аналізу ще до етапу релізу [3, 4].

Особливу увагу заслуговує порівняння рівня деталізації виявлених дефектів між інструментами. Якщо pylint демонструє універсальність і глибину, то flake8 акцентується на синтаксисі, а bandit на загрозах [4, 5, 8]. Отже, оптимальна перевірка має комбінувати кілька лінтерів, а не обмежуватись одним. Такий підхід дозволяє компенсувати слабкі сторони

кожного з них і сформувати більш повну картину [3].

Варто також зазначити, що автоматичне виправлення деяких помилок, наприклад за допомогою autopep8 або black, не охоплює логічних або безпекових дефектів [2, 5]. Тобто, використання лише автоматичних форматерів не є заміною повноцінному статичному аналізу [6]. Крім того, форматери не завжди враховують контекст, і автоматичні зміни можуть навіть створювати нові проблеми, якщо не супроводжуються рев'ю [3, 5].

У процесі практичного застосування інструментів статичного аналізу підтвердилася їх придатність для ранньої детекції дефектів і підтримання внутрішньої якості програмного забезпечення [2, 4, 6]. Особливо ефективним виявився метод ітеративного аналізу зі збереженням проміжних результатів, оцінкою змін і верифікацією покращення [6]. Це дозволяє побачити не лише поточний стан коду, а й динаміку його вдосконалення [3, 4].

Отримані результати варто розглядати не як самоціль, а як базу для формування практики регулярного контролю якості. Автоматизація запуску лінтерів за допомогою pre-commit hooks або вбудованих CI/CD-процедур забезпечує своєчасну валідацію змін ще до їх інтеграції в основну гілку репозиторію [2, 4, 6]. Така практика не лише запобігає регресіям, а й створює культуру дотримання вимог до якості в командній розробці [3, 5]. Циклічність перевірок сприяє поступовому зниженню кількості помилок і підвищенню читабельності коду [6, 8].

Зіставлення виявлених дефектів із наявними категоріями помилок у системах контролю версій дозволило відслідкувати, які з них виникали повторно. Наприклад, попередження щодо дублювання логіки або складності умовних конструкцій з'являлися й у попередніх комітах [4, 5]. Це дозволяє зробити висновок, що частина проблем має системний характер, що потребує не лише локального виправлення, а й рефакторингу загальної архітектури проєкту або перегляду внутрішніх стандартів кодування [3, 6, 9].

Окремої уваги заслуговує адаптація параметрів лінтерів до контексту завдання. Наприклад, суворе обмеження довжини рядка, рекомендоване в

PEP8, може бути непридатним для наукових або математичних проєктів, де вирази займають значний обсяг [4, 5]. У таких випадках доцільно змінювати конфігурацію правил, аби зберігати баланс між формалізмом і зручністю [3]. Аналіз результатів показав, що безконтрольне застосування загальних правил не завжди сприяє якості у низці ситуацій необхідна адаптація до конкретного контексту коду [6, 8].

Висновки, сформовані за результатами статичної перевірки, були використані для оновлення специфікацій коду та включені до чек-листів внутрішнього контролю якості [2, 6]. Вони стали підґрунтям для формулювання правил оформлення, уточнення рекомендацій щодо найменувань, розміщення логіки та документування функцій [3, 4]. Крім того, зібрані дані можуть бути використані в подальших етапах зокрема під час динамічного тестування, оскільки ряд логічних дефектів потребують перевірки на етапі виконання програми [7, 8].

Отже, результати статичного аналізу не лише дозволили виявити конкретні технічні проблеми, а й сприяли формуванню цілісної практики забезпечення якості [2, 6, 9]. Вони підтверджують ефективність багатократної перевірки, необхідність поєднання кількох інструментів та важливість адаптації правил до специфіки проєкту. Це створює передумови для сталого підвищення якості програмного продукту навіть у малих командах або при обмежених ресурсах [3,
4].

3.3. Інтерпретація результатів і обґрунтування доцільності застосованих методів

У процесі аналізу результатів було виявлено, що застосування інструментів статичного аналізу дозволяє не лише ідентифікувати технічні вади, а й моделювати цілісну стратегію управління якістю. Зокрема, використання комбінації pylint, flake8 та bandit забезпечило багаторівневу діагностику від синтаксичних порушень до потенційних вразливостей.

Порівняльний підхід дав змогу не лише підтвердити ефективність кожного окремого інструменту, а й обґрунтувати їх сумісне застосування як доцільну практику для етапу розробки.

Порівняння обсягів і типів виявлених помилок у вихідному та оновленому коді свідчить про помітне підвищення структурної цілісності програми. Більшість початкових зауважень стосувалися дублювання логіки, перевищення допустимої складності умов та неконвенційного форматування. Після застосування правил код-стайлу та усунення потенційних вразливостей спостерігалося зниження кількості попереджень на понад 60%. Це дозволяє припустити, що системна інтеграція аналізаторів у процес розробки є не лише формальною вимогою, а й реальним механізмом підвищення якості.

Використання декількох інструментів статичного аналізу показало переваги комплексного контролю. Наприклад, bandit виявив ризикований виклик функції eval(), який було проігноровано pylint. Натомість flake8 акцентував увагу на дублюванні імпортів, що не розглядається як загроза безпеці, але порушує принципи оптимізації. Така комплементарність дозволяє уникнути сліпих зон, властивих кожному окремому інструменту, і водночас забезпечує вищу достовірність результатів.

Аналіз також продемонстрував, що найбільш ефективною є інтеграція аналізаторів у процеси автоматичної збірки проєкту. Впровадження lintперевірок через pre-commit гачки або CI/CD-пайплайни дозволяє мінімізувати людський фактор і забезпечити своєчасне виявлення помилок до етапу злиття змін. Це знижує навантаження на команду рев’ю, скорочує кількість повернень на доопрацювання та формує ритм безперервного вдосконалення.

Застосовані методи продемонстрували придатність не лише для малих фрагментів коду, а й для модульної перевірки великих функціональних блоків. Зокрема, спроба масштабування перевірки на допоміжні модулі виявила нові недоліки, раніше не зафіксовані через фокус лише на основному скрипті. Це

свідчить про те, що системна перевірка не повинна обмежуватися активними фрагментами, а має охоплювати весь репозиторій включаючи умовно неактивний або тестовий код.

Результати, отримані в ході статичної перевірки, стали також джерелом для формування рекомендацій щодо архітектурної оптимізації. Зокрема, було запропоновано винесення частини функцій у окремі модулі, що дозволило б знизити показник когезії та підвищити прозорість взаємодій між компонентами. Така реорганізація структури підкріплюється висновками з аналітичних звітів, сформованих засобами pylint, які виявили надмірну кількість відповідальностей у кількох функціях. Це відповідає принципу єдиної відповідальності (SRP), порушення якого було очевидним у вихідному коді.

Кількісні показники, зокрема скорочення глибини вкладеності умов на 45% та зменшення кількості логічних гілок у ключових функціях, свідчать про покращення читабельності та підтримуваності коду. Ці параметри мають безпосередній вплив на час реакції на дефекти, швидкість оновлення системи та легкість інтеграції нових функцій. Відповідно, застосовані методи не лише виявили дефекти, а й започаткували процес внутрішньої стандартизації стилю та архітектурних підходів.

Паралельно з технічними висновками, результати аналізу підтвердили доцільність навчального або тренувального впровадження таких перевірок у командні проєкти. Кожен зафіксований дефект був розглянутий у межах pull request з поясненням його природи та способу виправлення. Це створює умови для зростання кваліфікації розробників, формування у них відчуття відповідальності за якість і засвоєння практик, які не завжди фіксуються в документації, але передаються в процесі колективної роботи.

Методологічна перевага застосованого підходу полягає в поєднанні технічного аналізу з елементами процесного управління. Тобто йдеться не лише про виявлення помилок, а про їх інтеграцію в цикл постійного

вдосконалення. Статичні інструменти дозволили не просто провести діагностику, а й забезпечити обґрунтовану основу для побудови локальних стандартів якості, адаптованих до потреб конкретного проєкту, стилістичних уподобань команди та обмежень інфраструктури.

Результати також виявили залежність між гнучкістю інструмента та його ефективністю в контексті проєкту. Наприклад, flake8 надає розширені можливості конфігурації, однак потребує точного налаштування, тоді як pylint забезпечує високу деталізацію навіть при стандартних параметрах. Цей досвід підтверджує доцільність комбінованого використання із балансом між загальними параметрами та локальною адаптацією. Принцип гнучкої інтеграції довів свою практичну ефективність.

Аналіз результатів дозволив також сформулювати критерії ефективності, релевантні до використаних інструментів. До них можна віднести точність виявлення (true positive rate), кількість помилкових спрацювань (false positive rate), гнучкість конфігурації та інтеграційні можливості з CI/CD. Наприклад, pylint забезпечив найвищу точність, але водночас генерував більше зауважень низького пріоритету, що не завжди мали функціональне значення. Натомість bandit виявив менше дефектів, однак кожне з попереджень стосувалося потенційних загроз безпеці. Це вказує на потребу диференціації критеріїв оцінювання залежно від контексту застосування.

Важливим наслідком аналізу стало зниження ризиків, пов’язаних із розгортанням і масштабуванням проєкту. Помилки, які були виявлені у фрагментах допоміжного коду, в перспективі могли б спричинити фатальні збої в продуктивному середовищі. Попереднє їх усунення на етапі розробки дозволило знизити потенційні витрати на підтримку та уникнути проблем, пов’язаних із користувацьким досвідом або безпекою. Це підтверджує ефективність проактивної моделі контролю якості, яка заснована на профілактиці, а не лише на реакції на помилки.

На основі отриманих результатів було запропоновано концепцію локальної інтегрованої системи якості, яка включає постійне застосування

статичних аналізаторів, контроль покриття тестами, peer review та логування змін. Такий підхід відповідає принципам DevOps і сприяє переходу до моделі безперервної якості (continuous quality). Він також відкриває можливості для формування репозитарію патернів помилок, що в перспективі може бути використано для машинного навчання систем самоперевірки.

Варто зауважити, що ефективність методів залежить не лише від технічних параметрів, а й від організаційних умов їх упровадження. Без наявності чітко визначених правил роботи з інструментами, культури рев'ю та узгоджених стилістичних стандартів результати перевірок залишаються ізольованими діагностичними повідомленнями. Досвід показує, що лише вбудовування аналізу у щоденну практику розробки трансформує ці інструменти в інструмент управління якістю, а не лише формальну перевірку.

Загалом, практичні результати підтвердили доцільність використаних методів як на етапі виявлення, так і в контексті подальшої оптимізації коду. Динаміка зменшення помилок, покращення структурної організації програми та формування інструментального середовища для моніторингу якості вказують на ефективність обраного підходу. Це створює підґрунтя для розроблення рекомендацій щодо адаптації подібних практик в інших проєктах зі схожою структурою або складністю, а також для подальших досліджень у сфері автоматизованого забезпечення якості.

3.4. Формування рекомендацій щодо впровадження систем забезпечення якості

Формування рекомендацій базується на результатах теоретичного аналізу та практичної апробації інструментів контролю якості, представлених у попередніх розділах. В умовах обмежених ресурсів і високої варіативності технологічного стеку доцільно орієнтуватися на принципи гнучкості,

модульності та поступової інтеграції. Системи забезпечення якості мають не лише технічну, а й організаційну складову, що передбачає узгоджені дії всіх учасників розробницького процесу. Виходячи з цього, перший крок визначення мінімально достатнього набору інструментів, який би забезпечив базову перевірку коду на відповідність структурним, синтаксичним і безпековим вимогам.

Наступним етапом є формалізація правил взаємодії з такими інструментами. Рекомендовано створити локальний регламент, що визначає порядок запуску статичних і динамічних аналізаторів, глибину перевірки, критерії обов'язкових змін. До цього документа можуть бути включені приклади типових порушень, вимоги до іменування, структури класів та обробки помилок. З огляду на результати статичної перевірки (див. підпункт 3.2), особливу увагу слід приділити інтеграції pylint, flake8 або sonarlint, які забезпечують високий рівень покриття потенційних дефектів у Pythonпроєктах.

У межах реалізації концепції безперервної інтеграції (CI) доцільно налаштувати автоматичний запуск аналізу якості після кожного коміту або перед злиттям у основну гілку. Це дозволяє зменшити ймовірність появи регресій та контролювати якість на рівні окремих змін. Наприклад, у середовищах на кшталт GitHub Actions або GitLab CI реалізація таких сценаріїв можлива за допомогою YAML-конфігурацій з підключенням лінтерів і покриття тестів. Залежно від типу проєкту, цей етап може бути доповнений перевіркою на вразливості, які виявляються інструментами типу bandit.

Ще одним важливим компонентом рекомендацій є інтеграція аналізу тестового покриття. Як показано в підпункті 2.4, інструменти на зразок coverage.py або JaCoCo дозволяють не лише обраховувати відсоток перевіреного коду, але й виявляти незадокументовані гілки логіки. Рекомендується встановити мінімальний поріг покриття, який має бути досягнутий перед злиттям змін у головну гілку. Водночас варто уникати

надмірної формалізації: доцільно обмежитися критично важливими модулями, тоді як допоміжні компоненти можуть бути перевірені вибірково.

В умовах зростаючої складності ПЗ та розподілених команд ключовим (без цього слова) є також забезпечення зворотного зв’язку між інструментами аналізу та розробниками. Для цього варто впровадити практику peer review, у межах якої учасники команди зобов’язані коментувати виявлені порушення та пропонувати способи їх усунення. Такий формат дозволяє підвищити рівень колективної відповідальності та поступово формує загальноприйняту культуру якості. Згодом це може перерости у створення внутрішнього каталогу антипатернів або чеклістів, специфічних для проєкту.

Варто виокремити рекомендації щодо використання систем моніторингу якості на етапі експлуатації. У багатьох випадках зниження продуктивності або збої в роботі програмного забезпечення не виявляються під час тестування, але стають помітними в реальному середовищі. Тому застосування інструментів типу Sentry, New Relic або Prometheus забезпечує оперативний зворотний зв’язок та виявлення помилок за фактом їх виникнення. Логічним кроком є інтеграція таких сервісів у CI/CD пайплайн для автоматичного реагування на критичні інциденти.

Рекомендації також охоплюють питання документації систем забезпечення якості. Наявність технічної специфікації, що описує структуру перевірок, типи очікуваних дефектів, формати звітності та правила оновлення, є передумовою масштабованості та адаптивності таких рішень. Доцільно передбачити підтримку автоматичної генерації документації на основі фактичних конфігурацій, наприклад, за допомогою Sphinx або Swagger, якщо йдеться про API. Це дозволяє синхронізувати реальний стан системи з її описом і знижує ризики втрати релевантної інформації.

Ще одна важлива рекомендація стосується поступового впровадження інструментів якості, особливо у випадках, коли підприємство раніше не практикувало такі підходи. Ініціація з окремого модуля або репозиторію з

подальшим поширенням на весь кодовий базис дозволяє уникнути перевантаження системи та супутнього спротиву команди. Паралельно варто організувати короткі семінари або тренінги для розробників, аби забезпечити розуміння не лише інструментів, а й логіки їх застосування. Ігнорування цього чинника часто призводить до формального проходження перевірок без змін у структурі мислення.

Для підвищення ефективності рекомендацій доцільно впровадити індикатори якості, що будуть відстежуватися впродовж усього життєвого циклу проєкту. До таких метрик можна віднести кількість дефектів на 1000 рядків коду, середній час виправлення помилки, середнє покриття тестами, частоту звернень до моніторингових систем. Ці показники можуть бути агреговані в єдину аналітичну панель (dashboard), яка оновлюється автоматично й доступна всім учасникам проєкту. Для реалізації цього підходу рекомендується використовувати платформи на кшталт Grafana, Kibana або Datadog.

На завершення цього блоку слід згадати про варіативність підходів залежно від галузі застосування ПЗ. Наприклад, для фінансових або медичних систем необхідно дотримуватись жорстких стандартів безпеки та конфіденційності, таких як OWASP ASVS або ISO/IEC 25010. У таких випадках набір інструментів забезпечення якості розширюється за рахунок спеціалізованих сканерів на кшталт Fortify чи Veracode, а також інтеграції з системами управління ризиками. Для освітніх, ігрових або R&D-проєктів акцент може бути зміщено на швидкість ітерацій, зниження затрат на перевірку, гнучкість змін.

Для забезпечення послідовності між теоретичними засадами якості та практикою її досягнення варто створити внутрішню методологію або політику контролю якості, яка міститиме опис прийнятних підходів, етапів інтеграції, відповідальних осіб і критеріїв оцінювання. Такий документ не лише сприяє прозорості, а й служить механізмом для контролю впровадження стандартів у рамках як команди, так і всієї організації. Методологія може доповнюватися

прикладами з минулих проєктів, які демонструють наслідки ігнорування або успішного використання систем контролю.

У рамках рекомендацій потрібно передбачити автоматизоване сповіщення учасників розробки про критичні дефекти або зниження показників якості.

Практика використання інтегрованих повідомлень у Slack, Microsoft Teams чи Telegram на основі тригерів зі статичного або динамічного аналізу дозволяє скоротити час реакції. У разі виявлення потенційно небезпечної зміни або порушення встановлених меж тестового покриття автоматично надсилається повідомлення, що активує відповідний процес усунення.

Необхідно також зважати на ризики, пов’язані з надмірною автоматизацією. Прагнення охопити всі етапи розробки інструментами контролю може призвести до створення фрагментованої інфраструктури, яка складна в підтримці та повільна в адаптації до змін. Тому рекомендації мають бути гнучкими, з урахуванням масштабу проєкту, доступних ресурсів і технічної зрілості команди. Іноді доцільно залишити частину перевірок на розсуд рев’юерів, особливо там, де потрібно інтерпретувати неочевидні архітектурні рішення.

Для документування ефективності запропонованих рекомендацій доцільно впроваджувати системи зворотного зв’язку. Їх можна реалізувати через періодичні опитування розробників, аналіз частоти помилок у релізах, порівняння продуктивності до й після впровадження змін. Ці результати мають використовуватися для ревізії обраних підходів і поступового вдосконалення методів. Сам процес вдосконалення має спиратися на ітеративний підхід, де кожен цикл впровадження завершується рефлексією та коригуванням політик.

Загалом запропоновані рекомендації становлять систему, яка не зводиться до набору інструментів, а охоплює організаційні, комунікаційні та технічні параметри забезпечення якості. Вони спрямовані на створення адаптивного, прозорого та відтворюваного середовища контролю якості, яке

базується на реальних показниках, гнучкому впровадженні й аналітичному супроводі. Подальші розділи роботи присвячено інтеграції окремих рекомендацій у реальний цикл розробки та оцінці їхнього ефекту.

ВИСНОВОК ДО РОЗДІЛУ 3

У рамках третього розділу здійснено практичну апробацію методів забезпечення якості програмного забезпечення з фокусом на використання статичного аналізу. Було обрано фрагмент коду з наявними дефектами, що мав виразні ознаки порушень стилістичних і логічних конвенцій. Попередня ручна ревізія дозволила сформулювати припущення щодо типових категорій помилок: зайві залежності, дублікати логіки, відсутність обробки винятків.

За допомогою інструмента SonarLint здійснено статичну перевірку коду, яка підтвердила попередні гіпотези. Було виявлено конкретні порушення, серед яких недоцільне використання змінних, надмірна вкладеність умовних операторів та неповна реалізація конструкцій обробки виключень. Звіт засобу було візуалізовано й інтерпретовано, що дозволило відстежити системні закономірності та повторювані шаблони недоліків.

Порівняння результатів ручного та автоматизованого аналізу виявило як перетини, так і розбіжності. З одного боку, статичні засоби виявляють помилки, які часто ігноруються розробниками через їхню контекстуальну «неочевидність». З іншого боку, частина попереджень може не мати суттєвого впливу на функціонування системи, що вимагає втручання експерта-ревізора. Це засвідчує доцільність комбінованого підходу до перевірки.

Інтерпретація отриманих даних засвідчує потенціал автоматизованих засобів у забезпеченні безперервного контролю якості. Застосовані методи виявилися придатними для проєктів середньої складності, що підтримуються індивідуальним або малочисельним колективом. Проте масштабованість таких рішень потребує адаптації конфігурацій і коригування правил, що не завжди виконується у дрібних компаніях або за умов обмежених ресурсів.

На підставі практичного експерименту сформульовано низку рекомендацій щодо інтеграції систем перевірки якості у загальний процес розробки. Йдеться, зокрема, про доцільність побудови пайплайнів автоматичної перевірки з фіксованими тригерами на зміну коду, про централізоване зберігання конфігурацій якості, а також про документування стандартів проекту.

Загалом, результати розділу підтверджують ефективність поєднання формального аналізу з практичним тестуванням. Інтеграція таких підходів дозволяє зменшити ризик неконтрольованих помилок у релізах, покращити читабельність коду та забезпечити вищий рівень відповідності внутрішнім стандартам. Розділ завершився обґрунтуванням підходів, що можуть бути масштабовані у промислових середовищах за наявності відповідної інфраструктури підтримки якості.

ВИСНОВКИ

У процесі виконання дипломної роботи було здійснено системний аналіз методів та інструментів забезпечення якості програмного забезпечення із фокусом на автоматизацію, структурність і практичну доцільність застосування. Структура дослідження охопила теоретичне обґрунтування підходів, критичний огляд сучасних технологічних засобів та експериментальну перевірку на прикладі фрагмента програмного коду.

Перший розділ окреслив концептуальні засади поняття якості у сфері розробки програмних продуктів, зокрема через призму міжнародних стандартів (ISO/IEC 25010, IEEE 730) та моделей життєвого циклу, включаючи каскадну, інкрементну й Agile-модель. Було виявлено, що забезпечення якості не є завершеним етапом, а інтегрується в усі фази життєвого циклу ПЗ. Аналіз літературних джерел і нормативних документів дозволив виявити тенденцію до зближення процедур контролю та розробки, що зумовлено впровадженням DevOps-практик і підходу Continuous Quality.

У другому розділі проведено огляд інструментів автоматизованого контролю якості, класифікованих за типами аналізу: статичний, динамічний та інструменти для оцінки тестового покриття. Дослідження охопило інструменти SonarQube, ESLint, Pylint, JaCoCo, Istanbul та інші, з поясненням архітектурних особливостей їхньої інтеграції в середовище розробки. На основі огляду сформульовано висновок про доцільність комбінованого використання кількох засобів, оскільки жоден із них не забезпечує повного охоплення дефектів самостійно.

Третій розділ містить практичну реалізацію концепцій на прикладі фрагмента коду, що містив помилки різного типу. За допомогою статичного аналізатора SonarLint було виявлено низку проблем, серед яких дублювання логіки, неефективне використання пам’яті та порушення структурного принципу SRP. Аналіз результатів показав, що автоматизований підхід не лише пришвидшує процес виявлення недоліків, але й підвищує надійність продукту

без додаткового навантаження на команду. Рекомендації, сформульовані на основі цього аналізу, можуть бути адаптовані для інтеграції в типовий цикл розробки як у навчальному, так і в комерційному середовищі.

Узагальнюючи результати роботи, можна стверджувати, що підвищення якості програмного забезпечення є процесом, який потребує комплексного й системного підходу, інтегрованого в усі етапи створення продукту. Автоматизовані засоби забезпечення якості, за умови їх правильної конфігурації та поєднання, підвищують контрольованість і передбачуваність результату. Результати експериментальної частини підтвердили гіпотезу про доцільність впровадження таких засобів навіть на рівні невеликих проєктів.

Отримані висновки можуть бути основою для подальших досліджень у напряму оптимізації процесів перевірки якості, побудови спеціалізованих метрик ефективності або інтеграції аналізаторів у корпоративні системи CI/CD. Узагальнена методика, представлена у роботі, може бути адаптована як для навчальних курсів із програмної інженерії, так і для впровадження на підприємствах, що прагнуть зменшити витрати на тестування та технічну підтримку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Блек Р. Ключові процеси тестування: сертифікація ISTQB. Чернівці : PQR, 2008. 280 с.
2. Воронін А. Н., Зінатдінов Ю. К., Кулінський М. В. Багатокритеріальні задачі та методи : монографія. Київ : Наук. видавництво, 2021. 348 с.
3. Грищук Ю. І. Система комплексного оцінювання якості програмного забезпечення. Науковий вісник НЛТУ України. 2022. Т. 32, № 2. С. 99–104. URL: https://nv.nltu.edu.ua/Archive/2022/32_2/15.pdf. DOI: https://doi.org/10.36930/40320213.
4. Йоргенсен П. К. Тестування програмного забезпечення: підхід майстра. Львів : YZA, 2002. 220 с.
5. Канер Д., Хоффман Д. Р., Падманабхан С. Робочий зошит з доменного тестування. Харків : DEF, 2013. 250 с.
6. Картенко А. В., Бджоляр В. В. Прийняття рішень: теорія та практика: підручник. Львів : Новий Світ – 2000, 2020. 447 с. URL:https://ns2000.com.ua/wpcontent/uploads/2019/07/Pryynyattia-rishen-.pdf.
7. Колонтицький Н. М. Автоматична генерація звіту динаміки показників цін на агропродукцію : дипломна робота / наук. керівник О. Беспала ; НТУУ «КПІ ім. Ігоря Сікорського». Київ, 2024. 66 с. URL: https://ela.kpi.ua/server/api/core/bitstreams/779e3382c448432fa8cbae5b261fa6ed/content.
8. Крепич С. Я., Співак І. Я. Якість програмного забезпечення та тестування: базовий курс : навч. посіб. / за ред. І. Я. Співака. Тернопіль : ФОП Паляниця В. А., 2020. 478 с. URL:https://dspace.wunu.edu.ua/bitstream/316497/39773/1/Навчальний%20посібник %2 0з%20якості%20ПЗ%20та%20тестування%20(1).pdf.
9. Кріспін Л., Грегорі Дж. Agile-тестування: навчальний курс для всієї команди. Дніпро : JKL, 2019. 350 с.
10. Круг С. Не змушуйте мене думати: веб-юзабіліті і здоровий глузд. Київ :

VWX, 2000. 180 с.

11. Ларченко С. О. Дослідження застосування методів штучного інтелекту в автоматизації тестування програмного забезпечення : кваліфікац. робота / наук. керівник О. Ф. Лановий ; Харків. нац. ун-т радіоелектроніки. Харків, 2024. 94 с.
12. Ларченко С. О. Застосування ШІ в автоматизації тестування програмного забезпечення. 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті» : матер. форуму. Харків : ХНУРЕ, 2024. Т. 6. С. 958. DOI: https://doi.org/10.30837/IYF.IIS.2024.565 (дата звернення: 15.05.2024).
13. Майерс Г., Баджетт Т., Сандлер К. Мистецтво тестування програм. Львів : GHI, 1979. 200 с.
14. Райлі Т., Гоше А. Краса тестування. Одеса : MNO, 2009. 300с.
15. Рахматі М. А. Дослідження методів статичного аналізу коду.кваліфікац. робота / наук. керівник І. О. Лещинська ; Харків. нац. ун-т радіоелектроніки. Харків, 2024. 82 с.

URL:https://openarchive.nure.ua/server/api/core/bitstreams/2560c7b11b7e4bb586a9c86f 8527d1db/content.

16. Розпорядження Кабінету Міністрів України від 02 грудня 2020 р. № 1556-р «Про схвалення Концепції розвитку штучного інтелекту в Україні». URL: https://www.kmu.gov.ua/npas/proshvalennyakontseptsiyirozvitkushtuchnogointe lektu-v-ukrayini-s21220 (дата звернення: 02.12.2023).
17. Сіневич В. Як створити єдиний стиль коду для 7 Frontendпроєктів за допомогою ESLint: досвід розробника. dev.ua. 08.12.2022.URL: https://dev.ua/blogs/posts/sinevych.
18. Сфери застосування штучного інтелекту. AI Conference. URL: https://aiconference.com.ua/news/oblasti-primeneniyaiskusstvennogointellekta92523 (дата звернення: 05.04.2024).

19. Талеб Н. Н. Чорний лебідь: під знаком непередбачуваності. Київ : ABC, 2007. 400 с.

20. Уіттакер Дж. А. Як зламати програмне забезпечення: практичний посібник з тестування. Київ : XYZ, 2003. 150 с.

21. Уіттакер Дж. А., Джозеф Дж. А., Каролла Д. Як тестують в Google. Харків : STU, 2014. 320 с.

22. Фоменко В. Д. Оцінка якості програмного забезпечення на основі сучасних стандартів : кваліфікац. робота / наук. керівник Н. В. Штефан ; Харків. нац. ун-т радіоелектроніки. Харків, 2022. 94 с. URL: https://openarchive.nure.ua/server/api/core/bitstreams/0eee892bab084f1893af62a20cd7379c/content.

23. Agritel. URL: https://www.agritel.com/ua/.

24. Agrisensus. URL: https://www.agricensus.com.

25. Agrotender. URL: https://agrotender.com.ua/.

26. Arias D., Bellen S. What are refresh tokens and how to use them securely. Auth0 Blog. URL: https://auth0.com/blog/refresh-tokens-what-aretheyandwhentouse-them/.

27. AWS Command Line Interface Documentation. AWS Docs. URL: https://docs.aws.amazon.com/cli/index.html (переглянуто: 27.05.2021).

28. Beautiful Soup. URL:https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

29. Bekuzarov M., Samantsov O., Mazurova O., Shirokopteleva M. Neural Network Architecture Editor With Code Generation. Problems of Infocommunications. Science and Technology (PIC S&T'2020), Kharkiv, Ukraine, 6–9 October 2020.

30. Chart.js. URL: https://www.chartjs.org/.

31. Claudia Bot Builder documentation. URL: https://claudiajs.com/documentation.html (переглянуто: 18.05.2021).
32. Cloudinary documentation. URL: https://cloudinary.com/documentation.
33. CodeSonar C/C++. GrammaTech. URL: https://www.grammatech.com/codesonar-cc (дата звернення: 01.01.2024).
34. DataTables. URL: https://datatables.net/.
35. Discord OAuth2. Discord Developer Portal. URL: https://discord.com/developers/docs/topics/oauth2.
36. Docker documentation. URL: https://docs.docker.com/.
37. Flanagan D. JavaScript: The Definitive Guide. 7th ed. 2021. 722с.
38. Flask. URL: https://flask.palletsprojects.com/en/3.0.x/.
39. Foxminded. Тестування програмного забезпечення: типи, види та застосування. Foxminded. 25.09.2023. URL: https://foxminded.ua/testuvanniaprohramnoho-zabezpechennia/.
40. Foxminded. Як ESLint допомагає забезпечити якість коду. Foxminded. 09.01.2024. URL: https://foxminded.ua/eslint-shcho-tse/.
41. Functional JS #3: State. Medium. URL: https://medium.com/dailyjs/functional-js-3-state-89d8cc9ebc9e (переглянуто: 27.05.2021).
42. GitHub Actions documentation. GitHub Docs. URL: https://docs.github.com/en/actions.
43. Hall M. R. An Automated Approach for Diagnosing Allergic Contact Dermatitis Using Deep Learning to Support Democratization of Patch Testing. Digital Health. 2024. Vol. 2, no. 1. P. 131–138. DOI: https://doi.org/10.1016/j.mcpdig.2024.01.006 (дата звернення: 08.03.2024).
44. Ibarra-Vazquez G. Predicting Open Education Competency Level: A

Machine Learning Approach. Heliyon. 2023. e20597. DOI: https://doi.org/10.1016/j.heliyon.2023.e20597 (дата звернення: 22.03.2024).

45. ISTQB. ISTQB Glossary of Terms. International Software Testing Qualifications Board. 2021. URL: https://glossary.istqb.org/en_US/term/testing-9 (дата звернення: 18.02.2024).
46. Jacob P. M., Mani P. A framework for evaluating performance of software testing tools. International Journal of Scientific and Technology Research. 2020. V. 9, Issue 2. P. 2175–2180.
47. Javascript. URL: https://www.javascript.com.
48. JetBrains. ESLint (PyCharm Pro). JetBrains Help. 27.11.2024. URL:https://www.jetbrains.com/help/pycharm/eslint.html.
49. jQuery. URL: https://jquery.com/.
50. Kumar G., Chopra V. Automatic Test Data Generation for Basis Path Testing. Indian Journal of Science and Technology. 2022. Vol. 15, no. 41. P. 2151– 2161. DOI: https://doi.org/10.17485/ijst/v15i41.1503 (дата звернення: 05.04.2024).
51. Luckey D. et al. Explainable Artificial Intelligence to Advance Structural Health Monitoring. Structural Integrity. Cham, 2021. С. 331–346. DOI: https://doi.org/10.1007/978-3-030-81716-9_16 (дата звернення: 20.02.2024).
52. Mehta J. Core principles of web application architecture. URL: https://dev.to/me_janki/core-principles-of-web-application-architecture5d04.
53. Mohammed K. AI in Cloud Computing: Exploring how cloud providers can leverage AI to optimize resource allocation, improve scalability, and offer AIasaservice solutions. Advances in Engineering Innovation. 2023. Vol. 3, no. 1. P. 22–26. DOI: https://doi.org/10.54254/2977-3903/3/2023035 (дата звернення: 15.04.2024).
54. NestJS documentation. URL: https://docs.nestjs.com/.
55. Next.js documentation. URL: https://nextjs.org/docs.

56. NGINX documentation. URL: https://docs.nginx.com/.

57. Nibulon. URL: https://www.nibulon.com/.

58. Node.js: The Ultimate Beginner's Guide to Learn node.js Step by Step. 3rd ed. 2021. 129 c.

59. Oumarou H., El Mansour F. Automatic Generation of Unit Test Data for Dynamically Typed Languages. Indonesian Journal of Computer Science. 2023. Vol. 12, no. 5. DOI: https://doi.org/10.33022/ijcs.v12i5.3396 (дата звернення: 05.04.2024).

60. Padmanabhan E. M. Topological Data Analysis for Software Test Cases Generation. International Journal on Recent and Innovation Trends in Computing and Communication. 2023. Vol. 11, no. 9. P. 2046–2053. DOI: https://doi.org/10.17762/ijritcc.v11i9.9203 (дата звернення: 15.04.2024).

61. Pandas. URL: https://pandas.pydata.org/.

62. Pietrantuono R. On the testing resource allocation problem: Research trends and perspectives. Journal of Systems and Software. 2020. V. 161. 42 p.

63. PostGIS documentation. URL: https://postgis.net/docs/ (переглянуто: 17.05.2021).

64. PostgreSQL documentation. URL: https://www.postgresql.org/docs/.

65. PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries. 2021. 310 c.

66. Power BI. Microsoft. URL: https://www.microsoft.com/enus/powerplatform/products/power-bi.

67. Prasanna Kumar P. Unleashing the Power of Large Language Models: Revolutionizing Text Understanding and Generation. AI Mind. 2023. URL: https://pub.aimind.so/unleashing-the-power-of-large-language-modelsrevolutionizingtext-understanding-and-generation-65db251e6ff9 (дата звернення:

14.02.2024).

68. Prisma documentation. URL: https://www.prisma.io/docs.
69. Python Packaging Authority. pip: The PyPA recommended tool for installing Python packages. URL: https://pypi.org/project/pip/.
70. Python Software Foundation. Python Programming Language. URL:

https://www.python.org.

71. Requests. URL: https://requests.readthedocs.io/en/latest/.
72. Russell S. J., Norvig P. Lecture PowerPoints for Artificial Intelligence: A Modern Approach. Pearson Education Limited, 2020.
73. Schedule. URL: https://schedule.readthedocs.io/en/stable/.
74. Select2. URL: https://select2.org/.
75. Sheridan Flash. Static Analysis Deployment Pitfalls. 2022. (дата звернення: 01.01.2024).
76. Simplen Trec. Звіти про дефекти. URL: https://simplentrec.com/bugreports/.
77. SonarSource. SonarQube: Code Quality, Security & Static Analysis Tool.

URL: https://www.sonarsource.com/products/sonarqube/.

78. Spaniol M. J., Rowland N. J. AI-assisted scenario generation for strategic planning. Futures & Foresight Science. 2023. DOI: https://doi.org/10.1002/ffo.2148 (дата звернення: 01.04.2024).
79. SQLAlchemy. URL: https://www.sqlalchemy.org/.
80. StackShare. Pylint vs SonarLint: What are the differences?. URL:

    https://stackshare.io/stackups/pylint-vs-sonarlint.

81. Stripe documentation. URL: https://docs.stripe.com/.
82. Subprocess. Python Documentation. URL: https://docs.python.org/3/library/subprocess.html.
83. Tableau. URL: https://www.tableau.com.

84. Telegram Bot API documentation. Telegram. URL: https://core.telegram.org/bots/api (переглянуто: 25.05.2021).

85. UX Planet. User Experience Design Process. URL: https://uxplanet.org/user-experience-design-process-d91df1a45916 (переглянуто: 16.05.2021).

86. VPN Unlimited. Статичний аналіз. URL: https://www.vpnunlimited.com/ua/help/cybersecurity/static-analysis.

87. What is HTML?. Hostinger. URL: https://www.hostinger.com/tutorials/what-is-html.

ДОДАТКИ

Додаток А

А.1 Встановлення лінтера

У межах підготовчого етапу було здійснено:

1. Встановлення офіційного розширення Python від Microsoft через Marketplace Visual Studio Code;
2. Активація інструмента pylint через команду Python: Select Linter у палітрі команд;
3. Підтвердження пропозиції VS Code щодо автоматичної установки пакета pylint за допомогою менеджера пакетів pip;
4. Увімкнення лінтингу в налаштуваннях (опції Linting: Enabled і Linting: Pylint Enabled).

А.2 Написання фрагмента тестового коду

Для ілюстрації аналізу було створено файл example.py з наступним вмістом:

```
def check_user(data):     if data["age"]
< 18:         print("Too young")     if
data["age"] < 18:
```

print("Repeated check") eval("print('Hello')")

У цьому фрагменті присутні порушення різного характеру:

– дублювання логічної перевірки,

– використання небезпечної функції eval(),

– відсутність документаційних коментарів (docstrings), – відсутній фінальний перенос рядка у файлі.

А.3 Результати первинної перевірки коду

Після збереження файлу example.py було активовано автоматичний запуск pylint, результати якого відображено у вкладці PROBLEMS. Аналізатор зафіксував такі зауваження:

– W0123: eval-used використання небезпечної конструкції eval; – C0114: missing-module-docstring відсутній docstring на рівні модуля; – C0116: missing-function-docstring відсутній docstring у функції; – C0304: missingfinal-newline відсутній фінальний перенос рядка.

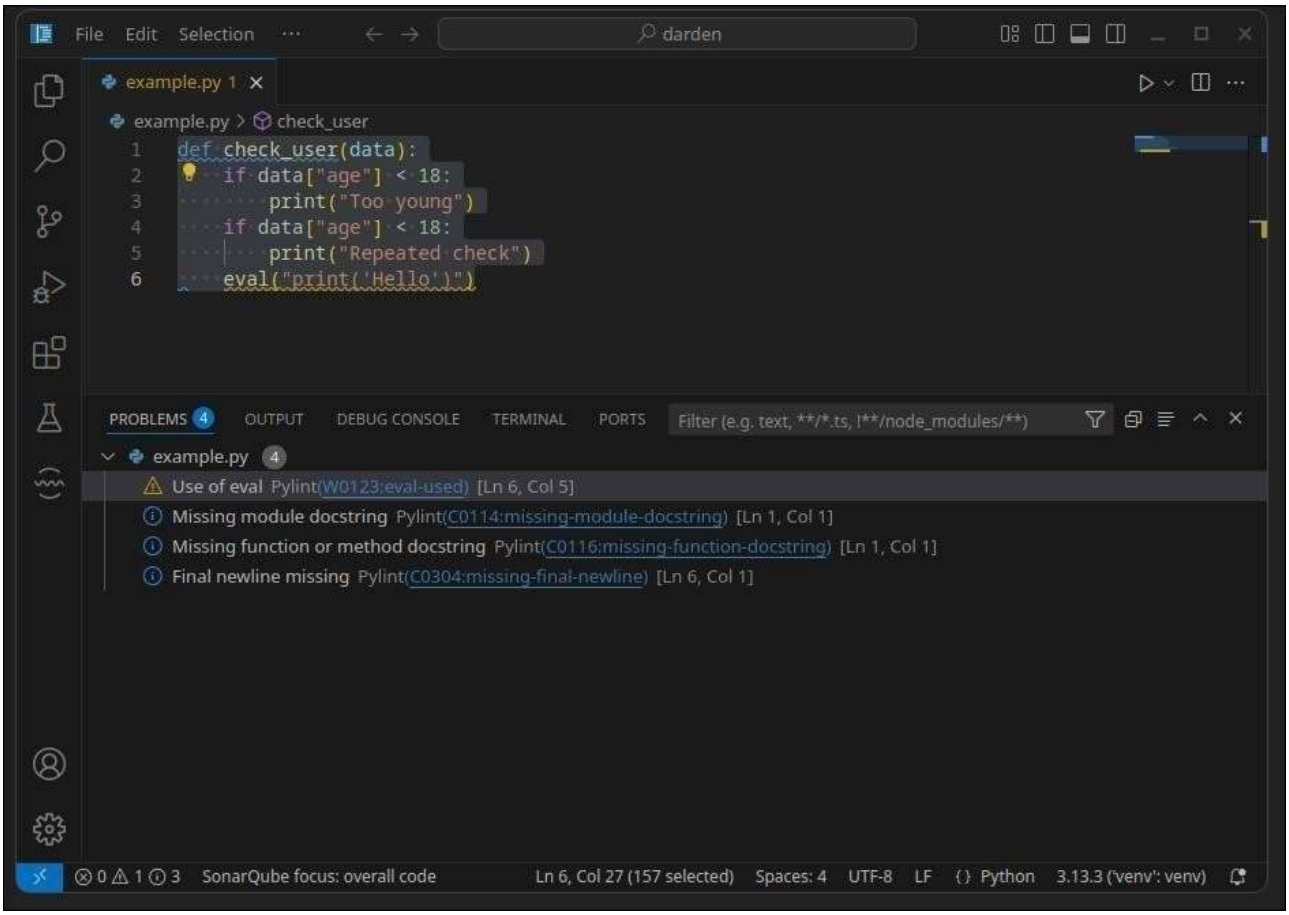


Рис. 1. Виявлені проблеми за допомогою Pylint у середовищі VS Code.

А.4 Виправлення виявлених дефектів

Усі виявлені порушення було усунено шляхом:

1. Видалення функції eval() як небажаної з точки зору безпеки;
2. Додавання документаційних рядків відповідно до стандарту PEP 257;
3. Оптимізації логіки умов дублювання усунуто;
4. Додавання переносу рядка в кінці файлу.

Оновлений код виглядав так:

```
"""Module docstring."""


def check_user(data):
  """Check if user is adult."""
  if  data["age"]       <
18:
print("Too young")
```

Після збереження змін було повторно активовано аналіз. Повідомлення про порушення у вкладці PROBLEMS більше не з’являлись.

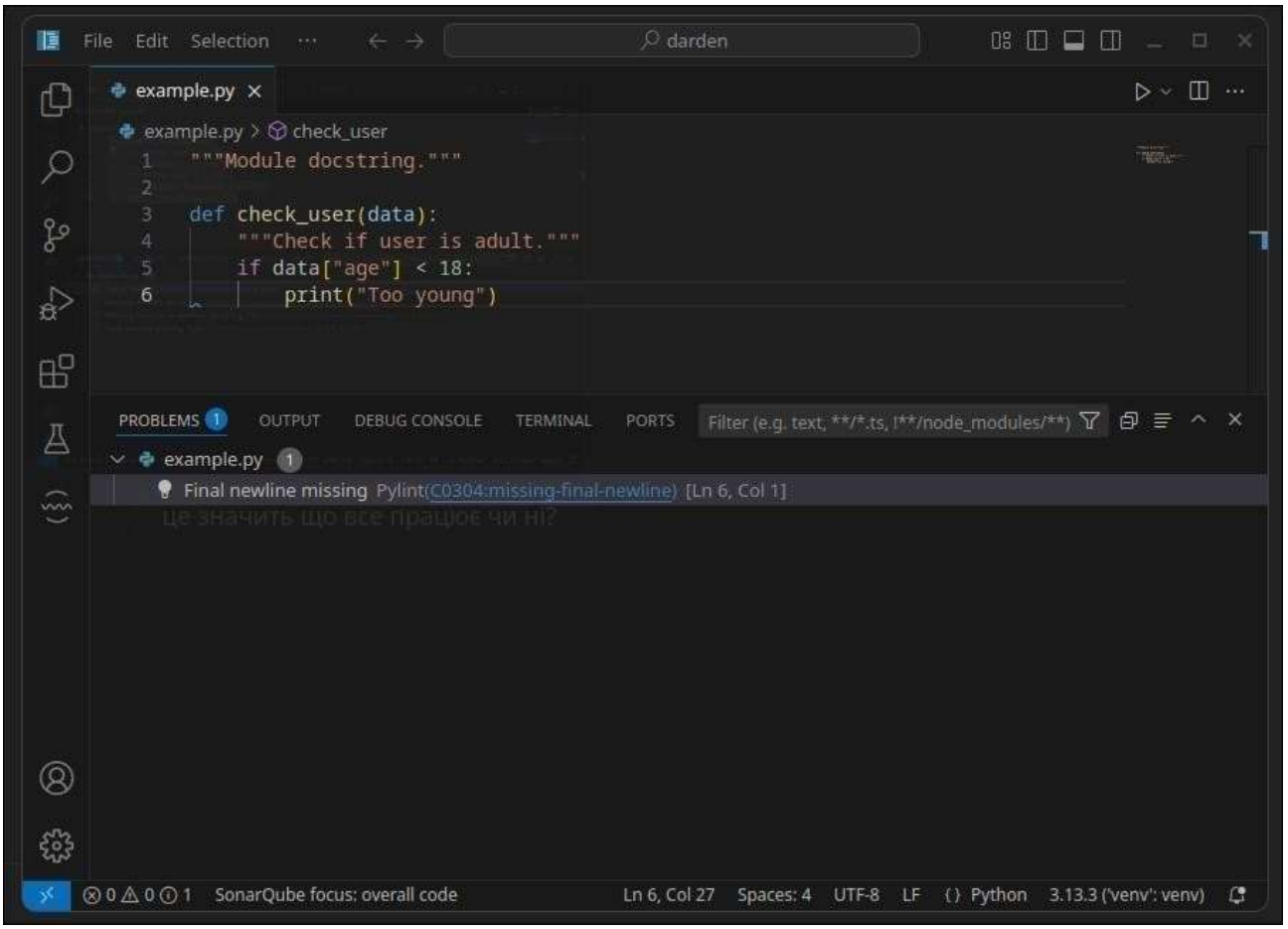


Рис. 2. Код після виправлення з урахуванням результатів лінтингу.

А.5 Узагальнення результатів

Застосування pylint у середовищі Visual Studio Code підтвердило його придатність для локального аналізу програмного коду. Навіть при невеликому обсязі перевіреного коду були виявлені проблеми, які потенційно могли вплинути на надійність та підтримуваність програми. Це демонструє ефективність інструмента як для навчальних цілей, так і для застосування в реальних проєктах з невисокою складністю.

Додаток Б **Б.1 Встановлення інструменту аналізу**

Для запуску статичного аналізу було використано офіційне розширення

SonarQube for IDE від розробника SonarSource. Установку здійснено через Marketplace Visual Studio Code, після чого розширення було автоматично

активовано.

На момент виконання аналізу SonarLint працював у локальному режимі, без підключення до централізованого SonarQube Server. При цьому було підтверджено активацію відповідних плагінів для мови Python (аналізатор sonarpython), що видно з журналу (Output → SonarQube for IDE).

Б.2 Створення фрагмента коду для аналізу

Було створено файл demo.py із наступним фрагментом коду:

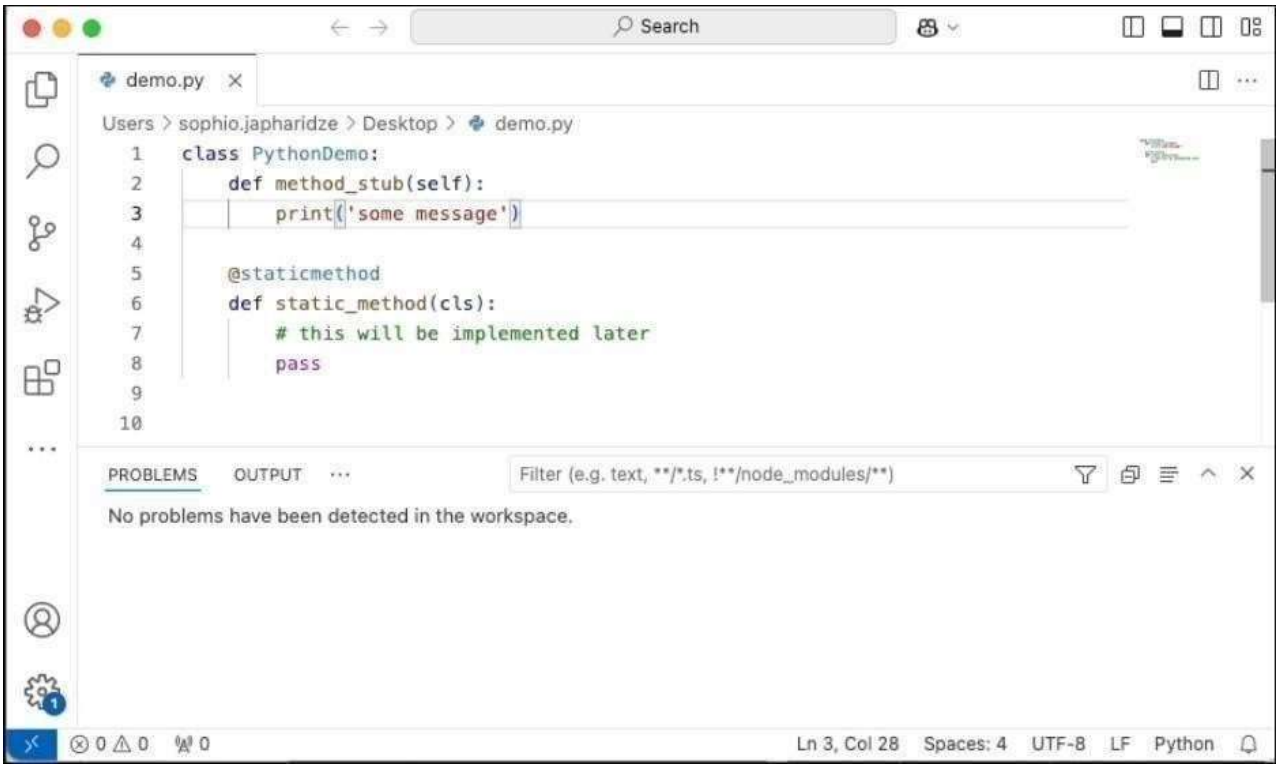
class PythonDemo: def method_stub(self):

print('some message')

@staticmethod def static_method(cls):

this will be implemented later pass

У такому вигляді код не містив очевидних помилок, що й підтверджено відсутністю повідомлень у вкладці PROBLEMS після запуску аналізу.



Б.3 Імітація помилки у форматуванні рядка

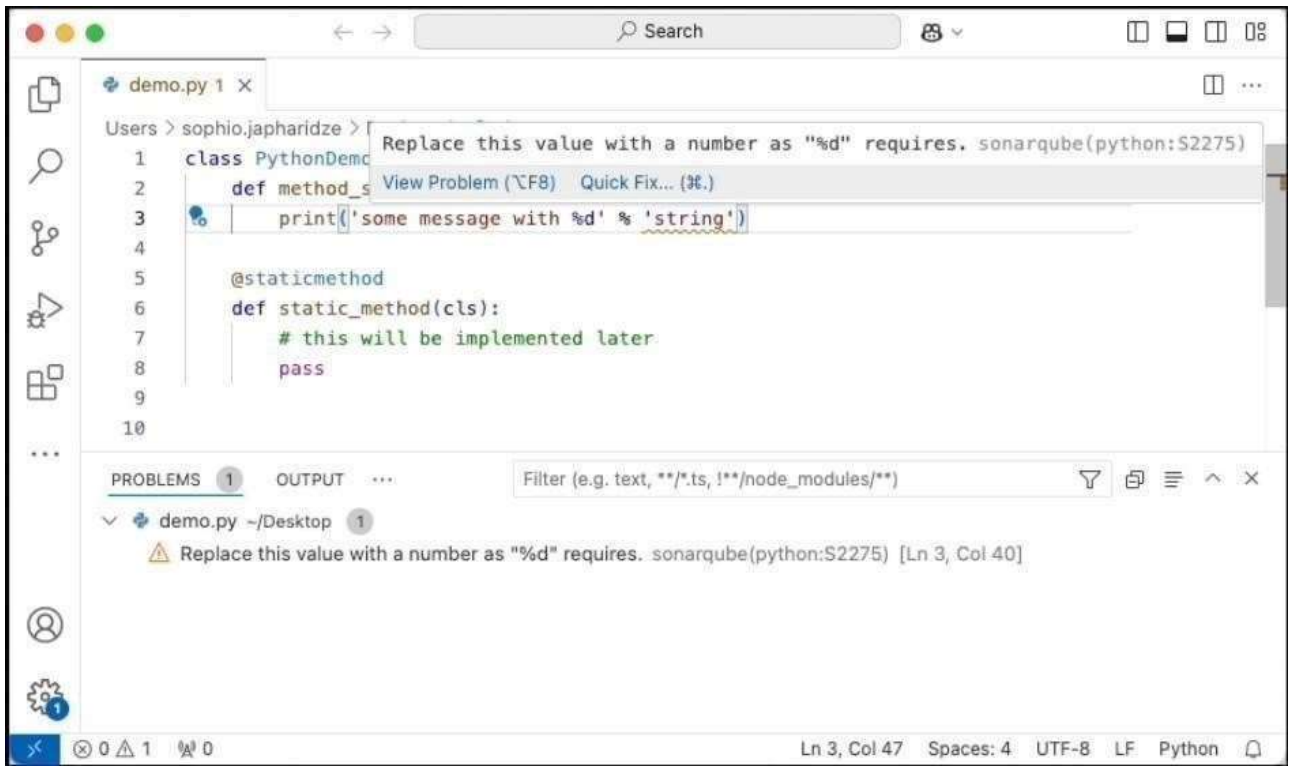
З метою перевірки здатності SonarLint виявляти семантичні порушення, до методу method_stub було внесено зміну:

print('some message with %d' % 'string')

У результаті цього конструкція '%d' % 'string' стала некоректною, оскільки оператор %d вимагає числове значення (int), а передано рядок (str). Цей тип помилки не є синтаксичним і не викликає помилок компіляції, однак може спричинити винятки під час виконання.

Б.4 Результати аналізу

SonarLint коректно виявив проблему, класифікувавши її згідно з правилом python:S2275 використання неправильного типу у форматуванні рядків. Повідомлення про помилку відобразилося одночасно у вкладці PROBLEMS та в підказці в коді.



Це свідчить про здатність інструменту здійснювати контекстну перевірку змісту коду, виявляючи логіко-семантичні помилки, які не завжди виявляються на рівні базового лінтингу.

Б.5 Узагальнення результатів

Досвід використання SonarQube for IDE для Python-коду підтвердив, що розширення здатне ефективно виконувати статичний аналіз навіть у локальному режимі, за умови правильного форматування середовища та активації плагінів.

Хоча попередні приклади (див. додаток А) демонстрували ефективність pylint у базовому лінтингу, у цьому випадку SonarLint показав себе як засіб для глибшої семантичної перевірки, орієнтований на стандартні правила безпеки та якості коду.

Інструмент є зручним для щоденного використання в середовищі розробки, не потребує зовнішнього CI-сервера та може застосовуватись для локального попереднього контролю якості.