Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Інфокомунікацій_____
(повна назва)
Кафедра _____Інформаційно-вимірювальних технологій_____
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____другий (магістерський)_____
_____Забезпечення якості коду програмних застосунків_____
_____
(тема)

Виконав:
здобувач __2___ року навчання,
групи _____Зям-23-2_____
_____Гусєв І. В._____
(прізвище, ініціали)

Спеціальність_____
__175 Інформаційно-вимірювальні технології_
(код і повна назва спеціальності)
Тип програми ___освітньо-професійна _____
(освітньо-професійна або освітньо-наукова)
Освітня програма _____забезпечення якості_
_____
(повна назва освітньої програми)

Керівник ____Єгоров А.Б._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ ____Захаров І.П._____
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет_____Інфокомунікацій_____

Кафедра_____Інформаційно-вимірювальних технологій_____

Рівень вищої освіти_____другий (магістерський)_____

Спеціальність_____175 Інформаційно-вимірювальні технології_____
(код і повна назва)

Тип програми _____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____забезпечення якості _____
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві_____Гусєву Івану Віталійовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _____Забезпечення якості коду програмних застосунків_____
_____
_____

затверджена наказом університету від _12_ ___11_____20_24_ р. № _1202Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії _10_ ___01__ 2025_ р.

3. Вихідні дані до роботи
3.1 Характеристики якісного коду: зрозумілість, модульність, надійність та_____
масштабованість;
3.2 Методи забезпечення якості коду: код-рев'ю, тестування, статичний аналіз коду та впровадження принципів SOLID;

4. Перелік питань, що потрібно опрацювати в роботі
4.1 Огляд та визначення основних характеристик якісного коду;
4.2 Аналіз методів та інструментів забезпечення якості коду;
4.3 Оцінювання метрик якості коду
4.4 Визначення критеріїв оцінювання якості коду та аналіз їхньої точності та відповідності специфіці проектів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри)
5.1 Огляд та визначення основних характеристик якісного коду
5.2 Що таке зрозумілість коду та чому вона важлива?
5.3 Інщі показники якісності коду?
5.4 Аналіз методів та інструментів забезпечення якості коду
5.5 Оцінювання метрик якості коду
5.6 Аналіз відповідності метрик специфіці проекті

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Основний	Проф. Єгоров А.Б.		10.01.2025

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Огляд літератури	15.11.2024	Вик
2	Формування завдань на дослідження	17.11.2024	Вик
3	Розробка основної частини тексту	20.12.2024	Вик
4	Формулювання висновків	27.12.2024	Вик
5	Оформлення тексту	30.12.2024	Вик
6	Побудова презентації	30.12.2024	Вик
7	Представлення тексту виступу	05.01.2025	Вик
8	Подання роботи на перевірку	10.01.2025	Вик

Дата видачі завдання _30_ _____11_____2024 р.

Здобувач _____
(підпис)

Керівник роботи _____ _____Проф. Єгоров А.Б._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи містить 67 сторінки, 7 рисунків, перелік посилань з 15 назв.

Дослідження, присвячене темі забезпечення якості коду, розкриває сучасні підходи та методи, спрямовані на покращення якості програмного забезпечення, підвищення продуктивності команд розробників та зменшення ризиків, пов’язаних із помилками і збоями. Актуальність дослідження обумовлена зростаючими вимогами до стабільності, надійності та масштабованості сучасних програмних продуктів, що функціонують у складних умовах та мають задовольняти високі стандарти користувацького досвіду.

Метою цього дослідження є аналіз основних характеристик якісного коду та визначення ефективних методів і практик забезпечення його якості.

Спираючись на визначені цілі, дослідження розглядає наступні питання:

1. Що саме означає якість коду, які критерії її визначення та чому вона є критично важливою для довгострокового успіху програмного забезпечення.
2. Які методи (та інструменти) найчастіше використовуються для забезпечення якості коду, та яким чином вони впливають на різні аспекти роботи команд розробників.
3. Яким чином ці методи допомагають знизити витрати на підтримку, полегшують інтеграцію нових членів у команди розробки та сприяють покращенню користувацького досвіду.

Результати дослідження спрямовані на те, щоб допомогти розробникам та менеджерам розробки ухвалювати обґрунтовані рішення щодо вибору методів забезпечення якості коду та підвищення ефективності процесу розробки.

ЯКІСТЬ ПРОГРАМНОГО КОДУ, ЗАБЕЗПЕЧЕННЯ ЯКОСТІ КОДУ, ІНІСТРУМЕНТИ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ КОДУ.

ABSTRACT

The explanatory note for the diploma thesis consists of 67 pages, 7 figures, and a list of references with 15 titles. The research dedicated to the topic of code quality assurance explores modern approaches and methods aimed at improving software quality, increasing the productivity of development teams, and reducing risks associated with errors and failures. The relevance of the study is driven by the growing demands for stability, reliability, and scalability of contemporary software products that operate in complex environments and must meet high user experience standards.

The objective of this research is to analyze the main characteristics of quality code and identify effective methods and practices for ensuring its quality. Based on the defined goals, the study examines the following questions:

1. What exactly does code quality mean, what are the criteria for its determination, and why is it critically important for the long-term success of software?
2. What methods (and tools) are most commonly used to ensure code quality, and how do they impact various aspects of development team operations?
3. How do these methods help reduce maintenance costs, facilitate the integration of new team members into development teams, and contribute to improving user experience?

The research results aim to assist developers and development managers in making informed decisions regarding the selection of code quality assurance methods and enhancing the efficiency of the development process.

PROGRAM CODE QUALITY, CODE SECURITY, CODE QUALITY TOOLS.

ЗМІСТ

ВСТУП

Забезпечення якості коду є критичним елементом розробки програмного забезпечення, що впливає на надійність, масштабованість та продуктивність кінцевого продукту. У сучасному світі розробники прагнуть створювати продукти, що можуть витримувати високі навантаження, легко підтримуватись та вдосконалюватись, і для цього необхідний чистий, добре структурований код, що відповідає встановленим стандартам.

Суть забезпечення якості коду полягає в застосуванні комплексних методик та інструментів, що допомагають виявляти помилки, покращувати архітектуру і знижувати технічний борг проєктів. Це включає як автоматизовані методи, такі як статичний аналіз коду та модульне тестування, так і організаційні практики, наприклад, код-рев'ю та дотримання стандартів кодування.

Якість коду безпосередньо впливає на надійність, продуктивність, масштабованість та легкість підтримки програмного продукту. Висока якість коду забезпечує стабільність роботи програми, зменшуючи кількість помилок і збоїв, що є особливо важливим для критично важливих систем, таких як фінансові платформи або медичні додатки. Кожен збій у таких продуктах може коштувати компанії як фінансових, так і репутаційних втрат.

Крім того, добре структурований код дозволяє оптимізувати роботу додатка, забезпечуючи високу швидкість виконання завдань. Це особливо актуально для сервісів з великим обсягом користувачів або для додатків, що працюють у режимі реального часу, де швидкість обробки інформації є критично важливою.

Ще одним важливим аспектом є масштабованість. Високоякісний код забезпечує легке розширення проєкту та адаптацію до нових вимог, оскільки структурованість та модульність коду полегшує додавання нових функцій і

інтеграцію з іншими системами, що має особливе значення для швидко зростаючих продуктів.

Нарешті, якість коду впливає на легкість підтримки та обслуговування. Чистий, зрозумілий код значно полегшує підтримку, оскільки іншим розробникам буде простіше розібратися з ним, вносити зміни та виправляти помилки. Це знижує витрати на підтримку проєкту та підвищує ефективність роботи команди, яка підтримує і вдосконалює продукт.

1 ЩО ТАКЕ ЯКІСНИЙ КОД

Якість коду — це комплекс властивостей, які забезпечують зрозумілість, стабільність, продуктивність та легкість підтримки програмного забезпечення. Високоякісний код спрощує взаємодію між членами команди, знижує кількість помилок і підвищує ефективність розробки. У цьому розділі розглянемо основні характеристики якісного коду.

1.2 Зрозумілість коду

Зрозумілість коду є основою його якості, адже саме вона забезпечує легкість сприйняття, обслуговування та підтримки. Зрозумілий код написаний простою, логічною мовою, де кожна змінна, функція та клас мають інтуїтивно зрозумілі імена. Написання читабельного коду вимагає дотримання таких принципів, як використання простих конструкцій, дотримання єдиного стилю форматування та надання змістовних коментарів, особливо для складних фрагментів. Зрозумілість є критично важливою для великих проєктів з численними учасниками, адже вона дозволяє кожному розробнику швидко ознайомитись з кодом, зменшуючи ризик помилок. У підсумку, чіткий і зрозумілий код підвищує ефективність команди та спрощує внесення змін на всіх етапах життєвого циклу проєкту.

1.2 Модульність і структурованість

Модульність означає, що код розділений на незалежні частини (модулі), кожна з яких виконує конкретне завдання. Це робить код легким у розширенні, спрощує тестування та дозволяє одночасно працювати над різними частинами проєкту. Модулі повинні мати чіткі межі відповідальності, що дозволяє

змінювати або покращувати один модуль без впливу на інші. Наприклад, у проєкті, де використовується модульна структура, розробники можуть працювати над розробкою нових функцій паралельно, що значно пришвидшує процес розробки. Крім того, модульність забезпечує кращу масштабованість коду: нові функції можуть бути додані шляхом підключення нових модулів, що не вимагає глобальних змін у кодовій базі.

1.3 Надійність та обробка помилок

Надійність коду полягає в його здатності стійко функціонувати навіть за неочікуваних обставин, наприклад, при неправильних вхідних даних або збоях системи. Для цього у програмі передбачаються механізми обробки помилок та винятків, які дозволяють уникнути збоїв. Обробка помилок має бути продуманою і враховувати різноманітні сценарії. Наприклад, у банківських додатках ненадійна обробка помилок може призвести до втрати даних або серйозних проблем безпеки. Ретельне тестування та обробка потенційних помилок є важливими аспектами створення надійного продукту, який не тільки коректно виконує свою основну функцію, але й забезпечує стійкість до впливу зовнішніх факторів.

1.4 Легкість підтримки

Підтримка коду є складовою якісного програмного забезпечення, адже проєкти можуть існувати роками, а з ними можуть працювати різні команди. Легкість підтримки досягається завдяки документуванню коду, стандартизації стилю написання та логічній структурі коду. Це дозволяє швидко знаходити та виправляти помилки, а також легко вносити зміни. Підтримка коду важлива для проєктів, які потребують частого оновлення, адже дає змогу не тільки швидко

вирішувати проблеми, але й адаптувати програмне забезпечення до нових вимог ринку та користувачів.

1.5 Продуктивність та оптимізація

Оптимізований код не тільки працює швидше, але й споживає менше ресурсів, що є важливим для програм з великим навантаженням або для систем з обмеженими ресурсами. Написання продуктивного коду включає вибір оптимальних алгоритмів, ефективне управління пам’яттю та уникнення надмірного використання ресурсів. Наприклад, в мобільних додатках оптимізація коду є вирішальною, оскільки обмежені ресурси пристроїв вимагають раціонального використання пам’яті й процесорного часу. Продуктивний код забезпечує ефективну роботу додатка та позитивний досвід користувачів, що особливо важливо для продуктів, орієнтованих на широку аудиторію.

1.6 Відповідність стандартам кодування

Дотримання стандартів кодування забезпечує єдиний стиль написання коду в межах команди або проєкту, що значно спрощує підтримку та розвиток продукту. Стандарти кодування регулюють форматування, правила іменування змінних, написання коментарів та організацію структури класів і функцій. Вони дозволяють забезпечити послідовність, що робить код легшим для розуміння іншими розробниками. Наприклад, у великих командах стандарт кодування дозволяє новим учасникам швидко інтегруватися в проєкт, а досвідченим розробникам — підтримувати єдність стилю в кодовій базі. Використання стандартів кодування є критичним для великих і довготривалих проєктів, де постійно з’являються нові функції та фічі, і забезпечення єдності стилю стає важливим для ефективного управління кодом.

1.7 Масштабованість: майбутнє росту та розвитку

Масштабованість є одним із найбільш критичних аспектів якості коду, особливо для проєктів, які мають потенціал для активного зростання. Масштабований код легко адаптується до нових вимог, великої кількості користувачів та розширення функціоналу. Програмні системи, що не розроблялися з урахуванням масштабованості, згодом стикаються з труднощами: будь-яке нове доповнення або зміна може призвести до необхідності змін у кількох модулях, що ускладнює підтримку й подовжує час на розробку нових функцій.

Щоб забезпечити масштабованість, розробники часто використовують певні архітектурні підходи, як-от розподілена архітектура, модульна структура, кешування та балансування навантаження. Масштабованість також підтримується принципами проєктування, як-от SOLID, що допомагають структурувати код таким чином, щоб його було легко змінювати й розширювати. Наприклад, у популярних веб-сервісах, що зростають із кожним роком, масштабованість дозволяє швидко інтегрувати нові функції та адаптуватись до різних змін на ринку. Розробка масштабованого коду також має великий вплив на довгострокові витрати на підтримку продукту, адже при масштабуванні системи організація коду впливає на споживання ресурсів, а отже, і на витрати компанії.

2 СПОСОБИ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ КОДУ

Забезпечення якості коду є багатостороннім процесом, що включає різноманітні техніки, інструменти та практики. Вони допомагають уникати помилок, підвищують продуктивність команди та створюють більш стабільне і зручне для підтримки програмне забезпечення. Розглянемо основні способи забезпечення якості коду.

2.1 Код-рев'ю

Код-рев'ю (або перевірка коду) — це процес, у якому один або кілька розробників переглядають код, написаний їхнім колегою. Мета код-рев’ю — знайти потенційні помилки, забезпечити відповідність стандартам кодування та покращити читабельність. Код-рев'ю дозволяє розробникам обмінюватись знаннями, обговорювати кращі практики та виявляти слабкі місця ще до інтеграції коду в основну базу. Зазвичай код-рев'ю проводиться у системах керування версіями, таких як GitHub або GitLab, що полегшує комунікацію між розробниками і спрощує обробку коментарів.

2.2 Написання тестів

Тестування є ключовим етапом у забезпеченні якості коду, адже саме тести дозволяють переконатися, що програма працює коректно та стабільно. Існує кілька видів тестів, включаючи юніт-тести, інтеграційні тести та функціональні тести. Кожен з них має свої завдання: юніт-тести перевіряють роботу окремих модулів, інтеграційні тести — взаємодію між модулями, а функціональні — коректність роботи програми з точки зору користувача. Добре продумана система тестування допомагає швидко виявляти помилки та запобігає збоїв у майбутньому. Використання інструментів для автоматизації

тестування, таких як Jest, JUnit або Selenium, підвищує ефективність та дозволяє виконувати тести при кожній зміні в коді.

2.3 Статичний аналіз коду

Статичний аналіз коду — це автоматизований процес перевірки коду без його виконання. Цей метод використовується для виявлення потенційних помилок, вразливостей та порушень стандартів кодування. Інструменти статичного аналізу, такі як SonarQube, ESLint та Pylint, сканують код і надають звіти про виявлені проблеми, пропонуючи рекомендації щодо їх вирішення. Статичний аналіз особливо корисний для великих проєктів, оскільки дозволяє виявити проблеми на ранніх етапах розробки. Інструменти аналізу можуть бути інтегровані у конвеєр CI/CD для автоматичної перевірки кожної версії коду, що значно знижує ризик внесення помилок у основну кодову базу.

2.4 Дотримання стандартів кодування

Стандарти кодування забезпечують єдиний стиль написання коду в команді, що підвищує читабельність та спрощує підтримку коду. Стандарти можуть включати правила іменування змінних, форматування, структурування функцій та коментування. Команди використовують загальновизнані стилі, такі як PSR для PHP або PEP8 для Python, або розробляють власні стандарти, адаптовані до специфіки проєкту. Інструменти на кшталт Prettier або ClangFormat допомагають автоматизувати дотримання стилю та уникнути суперечок у команді щодо форматування. Загалом, відповідність стандартам кодування зменшує ризик помилок, підвищує узгодженість роботи та полегшує адаптацію нових розробників до проєкту.

2.5 Впровадження принципів SOLID

Принципи SOLID — це набір основних правил, які забезпечують модульність, розширюваність та гнучкість коду. SOLID включає п'ять принципів: принцип єдиного обов’язку, відкритості/закритості, підстановки Лісков, розподілу інтерфейсів та інверсії залежностей. Дотримання цих принципів дозволяє створювати код, який легше розширювати та модифікувати без шкоди для його якості. Принципи SOLID є основою об'єктно-орієнтованого дизайну, що робить їх незамінними для розробки складних проєктів з розподіленою архітектурою. У короткостроковій перспективі це може ускладнити початковий етап розробки, але в довгостроковій перспективі значно полегшує підтримку й розвиток проєкту.

2.6 Використання CI/CD

CI/CD (Continuous Integration/Continuous Delivery) — це методологія, яка дозволяє автоматизувати процес розгортання і тестування програмного забезпечення. Вона включає постійне інтегрування коду з подальшим автоматичним тестуванням, що дозволяє швидко виявляти помилки і уникати їх накопичення. Завдяки CI/CD можна налаштувати безперервний цикл інтеграції, що передбачає постійне тестування та розгортання нових версій продукту. Інструменти на кшталт Jenkins, GitLab CI/CD та CircleCI допомагають створювати конвеєри, що автоматизують весь процес. Це дозволяє прискорити вихід нових версій продукту та забезпечити стабільність і якість коду, що особливо важливо для проєктів, які постійно розвиваються та оновлюються.

2.7 Документування та коментарі

Документування є одним з базових аспектів забезпечення якості коду, оскільки чітка і зрозуміла документація дозволяє іншим розробникам швидко ознайомитись із призначенням та логікою роботи модулів. Документація може включати опис функцій, змінних, алгоритмів та методів використання програмного забезпечення. Важливо також додавати коментарі до коду, особливо у складних ділянках, де логіка може бути неочевидною. Інструменти, такі як Javadoc або Doxygen, дозволяють автоматизувати процес створення документації. Документування допомагає зменшити витрати часу на навчання нових членів команди та підвищує якість обслуговування і розвитку проєкту в цілому.

3 ІНСТРУМЕНТИ ДЛЯ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ КОДУ

Різноманітні способи забезпечення якості, описані в попередньому розділі, потребують ефективних інструментів для їх реалізації. Інструменти автоматизують і полегшують виконання процесів перевірки, тестування, документування та дотримання стандартів. Їх використання сприяє підвищенню продуктивності команд, зниженню ризиків виникнення дефектів та економії ресурсів.

Цей розділ присвячений аналізу інструментів, які дозволяють реалізувати способи забезпечення якості коду. Для код-рев'ю важливими є сервіси, що спрощують перевірку змін і полегшують командну співпрацю. Написання тестів потребує бібліотек і фреймворків, які дозволяють ефективно створювати та запускати автоматизовані тести. Статичний аналіз коду забезпечується за допомогою інструментів, які виявляють помилки та потенційні проблеми без виконання програмного коду.

Також важливим є дотримання стандартів кодування, чому сприяють лінтери та форматтери, які допомагають узгоджувати стиль написання коду між членами команди. Впровадження принципів SOLID забезпечується завдяки використанню інтегрованих середовищ розробки (IDE) та плагінів, що спрощують рефакторинг і оптимізацію. Автоматизацію CI/CD процесів забезпечують платформи, які об'єднують інтеграцію, тестування та розгортання.

Не менш важливими є інструменти для документування, які дозволяють створювати зручну й зрозумілу документацію для коду чи API. Для парного програмування використовуються сервіси, що підтримують спільну роботу розробників у реальному часі, особливо актуальні для віддалених команд.

Усі описані інструменти було підібрано з урахуванням їх ефективності та популярності, що дозволяє інтегрувати їх у сучасні процеси розробки. Цей

розділ не лише описує функціональність інструментів, але й допомагає зрозуміти, як обрати оптимальний варіант для конкретного проєкту.

3.1 Інструменти для код-рев’ю

Процес код-рев’ю є важливою складовою сучасної розробки програмного забезпечення, спрямованою на забезпечення високої якості коду та підвищення стабільності продукту. Основною метою код-рев’ю є виявлення помилок, недоліків та покращення якості коду ще на ранніх етапах розробки. Крім того, рев’ю допомагає зберігати високий рівень відповідності стандартам кодування, що сприяє підвищенню читабельності та підтримуваності програмного коду.

Процес код-рев’ю зазвичай починається з того, що розробник створює запит на перегляд змін, виконавши коміти в окремій гілці репозиторію. Цей запит, як правило, містить детальний опис змін, зокрема, що було змінено, чому ці зміни необхідні та як вони впливають на систему. Після цього інші учасники команди (рев’юери) отримують можливість ознайомитись із кодом, перевірити його та залишити коментарі або рекомендації щодо покращення. Рев’юери уважно вивчають код, перевіряючи його на наявність помилок, дотримання стандартів кодування, ефективність та можливі покращення. Вони можуть вказувати на проблеми, що стосуються логіки, читабельності коду, його продуктивності або сумісності з іншими частинами проєкту. Важливо, що код-рев’ю є не лише процесом перевірки помилок, але й механізмом навчання: менш досвідчені розробники можуть отримати зворотний зв'язок від більш досвідчених колег, що допомагає їм вдосконалювати свої навички.

Однією з головних переваг код-рев’ю є можливість оперативно виявляти та виправляти помилки до того, як код буде інтегрований у основну гілку проєкту. Це дозволяє знизити ризик виникнення багів і неочікуваних проблем у процесі подальшої експлуатації продукту. Окрім цього, код-рев’ю забезпечує

кращу командну взаємодію та комунікацію між розробниками, оскільки кожен має можливість поділитися своїм баченням вирішення проблем і отримати колективну оцінку своїх змін.

Завдяки інтеграції з інструментами для автоматизації перевірок, такими як статичний аналіз коду, тести та CI/CD, процес код-рев'ю стає ще ефективнішим, оскільки дозволяє автоматично перевіряти код на помилки та відповідність стандартам без необхідності залучати додаткові ресурси. В результаті, правильно організоване код-рев'ю допомагає значно знижувати кількість дефектів у кінцевому продукті та підтримувати високий рівень якості протягом усього циклу розробки.

Git є основним інструментом для організації процесу код-рев'ю. Це система керування версіями, яка дозволяє відслідковувати зміни в програмному коді та забезпечувати ефективну співпрацю між розробниками. Git дозволяє кожному розробнику працювати над окремою гілкою, що містить копію основного проєкту, і вносити зміни без впливу на інші частини коду. Кожен учасник команди має можливість зберігати, відновлювати та об'єднувати різні версії коду, що дозволяє працювати над проектом одночасно багатьом розробникам.

Git використовує репозиторії для зберігання історії змін, що дає можливість здійснювати контроль за всіма комітами, відслідковувати етапи розвитку проєкту та аналізувати внесені зміни. Одна з основних переваг Git — його дистрибутивна природа, яка дозволяє кожному розробнику мати повну копію проєкту з усією історією змін. Зміни можуть бути інтегровані в основну гілку через процес злиття (merge), а також можуть бути переглянуті іншими розробниками через функції Pull Requests або Merge Requests на таких платформах, як GitHub чи GitLab.

Git допомагає організувати співпрацю в команді, полегшуючи процеси синхронізації та управління версіями коду. Це важливий інструмент для

сучасної розробки програмного забезпечення, оскільки він забезпечує надійний спосіб організації змін, мінімізуючи ймовірність конфліктів між кодом, написаним різними розробниками.

Далі ми розглянемо два популярних сервіси для керування репозиторіями коду — GitHub і GitLab. Ці платформи забезпечують інтеграцію з системою керування версіями Git і пропонують широкий набір інструментів для управління розробкою програмного забезпечення, співпраці команд та автоматизації процесів.

GitHub та GitLab є важливими інструментами для командної роботи, адже вони дозволяють зручно організовувати репозиторії, вести історію змін, працювати з гілками та керувати злиттям змін у головну гілку проєкту. Крім базових функцій для роботи з Git, обидві платформи пропонують потужні можливості для організації процесу код-рев'ю.

У GitHub і GitLab реалізовані спеціальні інструменти для проведення код-рев'ю, такі як Pull Requests (на GitHub) і Merge Requests (на GitLab). Ці функції дозволяють розробникам створювати запити на злиття змін, які можуть бути переглянуті іншими учасниками команди. Інтерфейси обох платформ забезпечують зручний перегляд змін у коді, дозволяють залишати коментарі, пропонувати виправлення та обговорювати ключові рішення.

Реалізація інструментів для код-рев'ю на цих платформах забезпечує:

- Виявлення помилок і недоліків у коді ще до його інтеграції в основну гілку.
- Підтримку командного навчання через обмін знаннями та досвідом між розробниками.
- Стандартизацію та узгодженість коду завдяки спільній перевірці дотримання правил кодування.

Детально вивчивши ці інструменти, ми зможемо краще зрозуміти, як саме вони впливають на якість розробки та як їх можна ефективно використовувати в щоденній роботі.

3.1.1 GitHub Pull Requests та GitLab Merge Requests

GitHub та GitLab є лідерами серед платформ для управління репозиторіями коду, і в обох реалізовані інструменти для підтримки процесу код-рев'ю. На GitHub цей механізм називається Pull Requests, тоді як у GitLab він отримав назву Merge Requests. Обидві системи надають користувачам можливість організовувати та контролювати зміни в коді перед їх інтеграцією в основну гілку, забезпечуючи прозорість і узгодженість у командній роботі.

Загальний функціонал обох систем будується навколо одного принципу: розробник, який вносить зміни в код у своїй гілці, створює запит на їх перевірку командою. Це дозволяє виявляти помилки, оцінювати якість реалізації, а також обговорювати можливі покращення. Інтерфейси обох платформ пропонують зручні інструменти для перегляду змін, коментування коду та автоматичного запуску тестів.

Основними особливостями є можливість детального порівняння змін, коментування конкретних рядків коду, обговорення в межах окремого запиту та інтеграція з системами CI/CD для автоматизації перевірок. Користувачі також можуть налаштовувати права доступу, призначати відповідальних за рев'ю, а також вирішувати конфлікти між гілками безпосередньо через інтерфейс платформи.

Різниця між GitHub Pull Requests і GitLab Merge Requests полягає переважно у деталях реалізації та фокусі платформи. GitHub більше орієнтований на глобальну спільноту розробників, що відображається у його відкритій екосистемі. Pull Requests пропонують зручний механізм для

відкритих репозиторіїв, де участь можуть брати тисячі сторонніх контрибуторів. GitLab, своєю чергою, більше спрямований на корпоративний сегмент і часто використовується для внутрішніх командних проєктів. Merge Requests інтегровані в ширший набір DevOps-функцій, таких як управління пайплайнами CI/CD і моніторинг проєктів.

Процес роботи з цими інструментами схожий, але GitHub має спрощений підхід до налаштування рев'ю для відкритих репозиторіїв, тоді як GitLab надає гнучкіші можливості для налаштування процесу в приватних репозиторіях. Наприклад, у GitLab можна встановлювати обов'язкові перевірки з боку певних рев'юерів або автоматично блокувати злиття без проходження всіх перевірок.

Обидва інструменти є надзвичайно важливими для забезпечення якості коду в проєктах різного масштабу. Вони не лише сприяють виявленню помилок, а й створюють середовище для постійного навчання та вдосконалення команди через спільний перегляд та обговорення.

На зображенні представлений інтерфейс створення Merge Request у GitLab (рисунок.3.1), який забезпечує користувачам зручність і прозорість у роботі з командними проєктами. У процесі створення Merge Request розробник вводить заголовок і опис змін, що дає можливість пояснити контекст або мету нововведень.

Також демонструються вихідна і цільова гілки, між якими відбувається злиття. Це дозволяє легко зрозуміти, звідки надходять зміни і куди вони інтегруються. Інтерфейс надає доступ до перегляду змінених файлів або коммітів, пов'язаних із Merge Request, і можливості для ведення обговорень.

Особливістю GitLab є можливість призначення відповідальних за рев'ю, так званих Assignees. У відповідному полі вибирається один або кілька членів команди, які мають переглянути код, дати свої коментарі та затвердити Merge Request. Це дозволяє чітко розподілити відповідальність і покращує організацію процесу перевірки.

Ілюстрація відображає, як розробники можуть створювати запити на перевірку коду, комунікувати з колегами та організовувати якісний робочий процес.

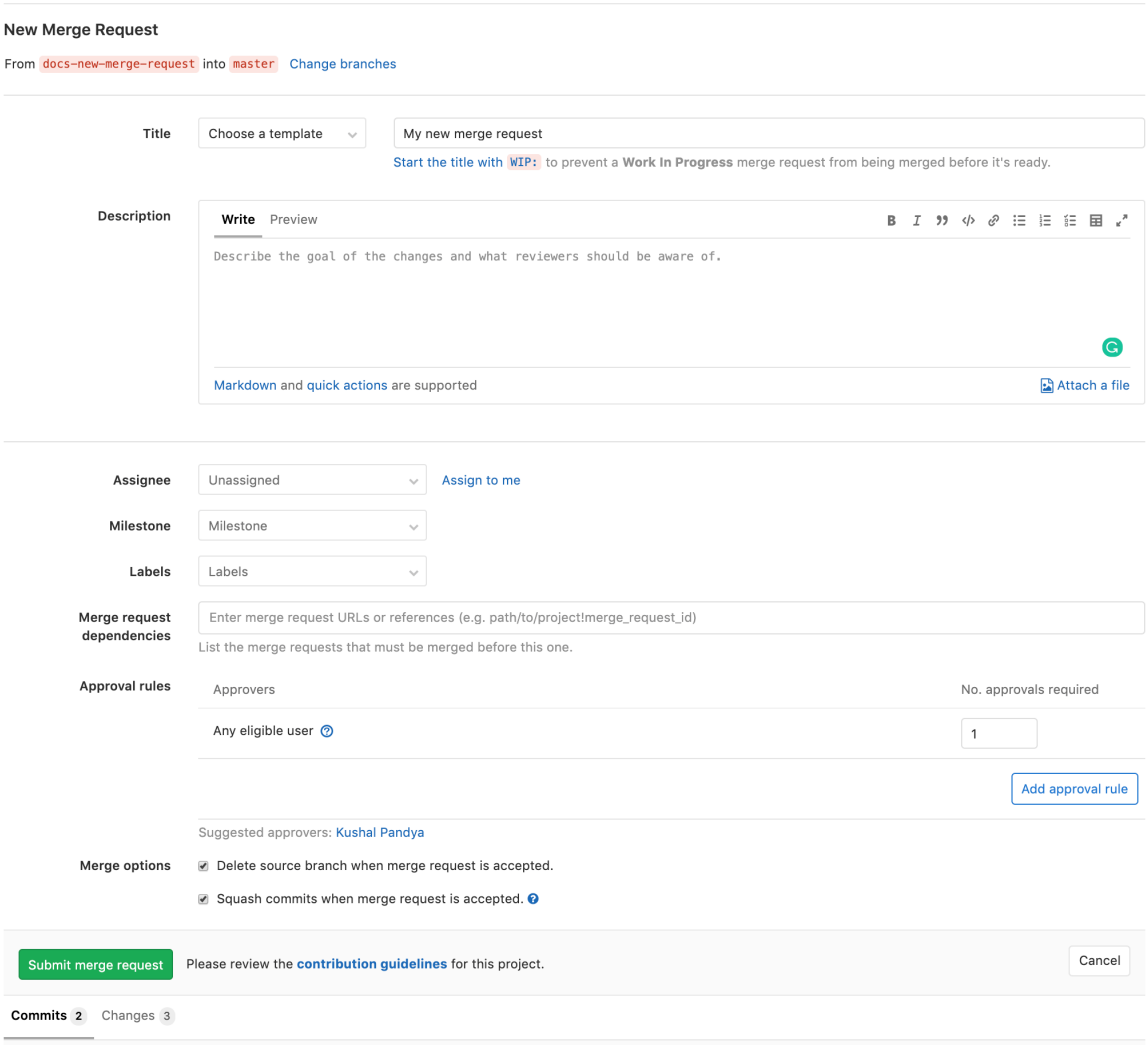

Рисунок 3.1 – Створення merge request у Gitlab

Github має схожий інтерфейс та функціонал.

3.2 Інструменти для написання тестів

Тестування коду — це невід'ємна частина розробки, яка забезпечує стабільність, надійність і прогнозованість роботи програмного забезпечення. Завдяки тестам команди розробників можуть уникнути критичних помилок, що виникають під час змін у коді або інтеграції нових функцій. Цей процес

охоплює перевірку як окремих компонентів системи, так і всієї її структури в комплексі.

Тести дозволяють розробникам автоматизувати перевірку функцій, що значно знижує витрати часу на ручне тестування. Вони також є важливим елементом процесу CI/CD, забезпечуючи швидкий зворотний зв’язок про стабільність змін у коді. Залежно від рівня перевірки, існують різні види тестування, такі як модульне, інтеграційне або e2e (end-to-end). Кожен вид тестів орієнтований на свій рівень деталізації.

Написання тестів вимагає розуміння вимог до функціоналу, вибору відповідного інструменту та правильного формування сценаріїв. У цьому розділі розглянемо, як створювати тести різних типів, які інструменти використовуються для їх автоматизації та як забезпечується їхній запуск і перевірка результатів.

3.2.1 Модульне тестування

Модульне тестування є фундаментом автоматизованого забезпечення якості коду. Його мета полягає у перевірці роботи окремих одиниць програми, таких як функції, методи або класи. Відокремлення цих одиниць дозволяє ізолювати проблеми на найнижчому рівні коду та забезпечити їх коректне функціонування, незалежно від зовнішніх залежностей чи інших частин системи.

Процес написання модульних тестів починається з розуміння того, яку функціональність потрібно перевірити. Наприклад, якщо функція відповідає за виконання математичних операцій, таких як обчислення середнього значення, тест перевіряє всі можливі сценарії: коректні вхідні дані, відсутність даних, помилкові типи даних і граничні випадки.

Для створення модульних тестів розробники використовують спеціальні інструменти — тестові бібліотеки. Вони надають функції для визначення умов виконання тесту, перевірки результатів і повідомлення про помилки. Приклади таких бібліотек включають популярні фреймворки, які дозволяють реалізовувати й запускати модульні тести автоматично, не витрачаючи час на ручну перевірку.

Модульний тест зазвичай складається з трьох основних етапів:

- Налаштування середовища. На цьому етапі створюються всі необхідні умови для виконання тесту. Наприклад, якщо функція працює із зовнішньою базою даних, для тесту можна використати "мок" (імітацію цієї бази). Це дозволяє уникнути реальних запитів до бази, зберігаючи контроль над тестовими даними.
- Виконання функції. Далі викликається функція або метод, які перевіряються. При цьому передаються заздалегідь визначені вхідні дані. Наприклад, для функції, що обчислює середнє значення, передається масив чисел [2, 4, 6].
- Перевірка результатів. Завершальним етапом є порівняння результатів роботи функції з очікуваними. Якщо результат відповідає очікуванням, тест вважається успішним. Якщо ні — повідомляється про помилку, і розробник отримує зворотний зв’язок для виправлення коду.

Розглянемо функцію, яка обчислює середнє значення з масиву чисел (рисунок 3.2):

```
function calculateAverage(numbers) {
  if (!Array.isArray(numbers) || numbers.length === 0) {
    throw new Error('Invalid input');
  }
  return numbers.reduce((sum, num) => sum + num, 0) / numbers.length;
}
```

Рисунок 3.2 – Приклад функції

Тест для цієї функції перевіряє кілька сценаріїв:

- Звичайний масив чисел ([2, 4, 6]).
- Порожній масив.
- Масив із некоректними даними, наприклад, рядками.

Розглянемо приклад тесту для такої функції (рисунок 3.3):

```
describe('calculateAverage', () => {
  it('should return correct average for a valid array', () => {
    const result = calculateAverage([2, 4, 6]);
    expect(result).toBe(4);
  });

  it('should throw an error for an empty array', () => {
    expect(() => calculateAverage([])).toThrow('Invalid input');
  });

  it('should throw an error for non-array input', () => {
    expect(() => calculateAverage('not an array')).toThrow('Invalid input');
  });
});
```

Рисунок 3.3 – Приклад модульного тесту

Для написання модульних тестів використовуються спеціалізовані інструменти. Серед найпоширеніших:

- Фреймворки для тестування. Вони забезпечують розробників базовими функціями для написання й запуску тестів. Приклади включають Jasmine, Mocha та Jest. Кожен із цих інструментів підтримує опис тестів, перевірку очікуваних результатів і зручні звіти.
- Інструменти для мокування. Для імітації зовнішніх залежностей використовуються бібліотеки, які дозволяють створювати моки. Наприклад, якщо функція взаємодіє з базою даних, можна створити імітацію, яка повертає тестові дані, не підключаючись до реальної бази. Популярні бібліотеки для цього — Sinon.js або Mockery.
- Раннери тестів. Ці інструменти автоматизують виконання тестів і виведення звітів про їх виконання. Наприклад, Karma дозволяє запускати тести в різних браузерах для перевірки сумісності.

В процесі розробки модульні тести зазвичай запускаються автоматично на етапах, коли код змінюється або додається нові функціональності. У GitLab для цього існують інструменти CI/CD, які дозволяють налаштувати автоматичний запуск тестів кожного разу, коли змінюється код. Тести можуть запускатись під час створення merge request або при кожному пуші в репозиторій.

Вищезгаданий GitLab дозволяє створювати конфіг файл, де вказується етап тестування, зокрема запуск модульних тестів. Після того, як зміни будуть завантажені в GitLab, система автоматично виконає тести, і результати будуть відображені у інтерфейсі GitLab. Розробники отримують зворотній зв'язок щодо того, чи пройшли тести, що дозволяє оперативно реагувати на проблеми та виправляти помилки до інтеграції змін у основну гілку.

Таким чином, модульні тести є важливою частиною забезпечення якості програмного коду, оскільки вони дозволяють перевіряти окремі компоненти системи на предмет коректності їх роботи. Перевагою модульних тестів є те, що вони швидко пишуться та виконуються, що дозволяє розробникам швидко отримувати зворотний зв'язок і виправляти помилки на ранніх етапах розробки.

Завдяки автоматичному запуску тестів, що налаштовується через CI/CD системи, наприклад, у GitLab, можна забезпечити безперервну перевірку коду при кожній зміні. Це значно знижує ризик інтеграції помилок у основний код, підвищує стабільність програми та забезпечує ефективний процес розробки без зупинок на виправлення помилок.

3.2.2 Інтеграційне тестування

Інтеграційне тестування є наступним етапом після модульного тестування і зосереджується на перевірці взаємодії між різними модулями або компонентами програми. Метою цього тестування є виявлення помилок, які можуть виникнути при з'єднанні окремих частин системи, що можуть не проявлятися на етапі модульного тестування. Інтеграційне тестування перевіряє, чи правильно компоненти працюють разом, обмінюються даними та виконують загальні функції в межах цілісної програми.

У рамках інтеграційного тестування розробники перевіряють, як різні модулі системи взаємодіють між собою, чи немає проблем у роботі з базами даних, зовнішніми сервісами чи іншими компонентами, з якими система повинна взаємодіяти. Цей етап особливо важливий у складних додатках, де безперервна інтеграція різних частин може призвести до непередбачуваних проблем.

Інтеграційне тестування дозволяє виявити баги, які можуть з'явитися лише після того, як окремі модулі взаємодіють один з одним. Наприклад, можуть виникнути помилки при передачі даних між модулями або при неправильному налаштуванні зовнішніх сервісів. В таких випадках завдяки інтеграційним тестам можна швидко ідентифікувати джерела помилок.

Інтеграційні тести зазвичай вимагають налаштування середовища, яке імітує реальні умови, де модулі можуть взаємодіяти між собою. Наприклад, для

тестування взаємодії з базою даних можуть використовуватись спеціальні тестові бази або мок-сервіси для зовнішніх API. У таких тестах важливо створити умови, які найбільш наближені до реальних, щоб коректно перевірити всю систему в цілому.

Існує безліч інструментів для інтеграційного тестування, які допомагають автоматизувати перевірку взаємодії між компонентами системи. Вони дозволяють розробникам створювати ефективні тести для перевірки того, як різні модулі програми працюють разом у реальному середовищі. Інструменти, як Mocha, Chai і Supertest, дозволяють легко налаштувати середовище тестування, виконувати запити до API, перевіряти відповіді і забезпечувати правильність обміну даними між модулями. Такі інструменти забезпечують зручну інтеграцію з CI/CD процесами, дозволяючи запускати тести автоматично під час кожної зміни коду, що значно підвищує ефективність і надійність розробки програмного забезпечення.

Розглянемо приклад тестування взаємодії між сервером і базою даних за допомогою Mocha та Chai. Уявімо, що ми тестуємо API для отримання користувачів з бази даних (рисунок 3.4):

```
const request = require('supertest');
const app = require('../app'); // Ваш сервер
const chai = require('chai');
const expect = chai.expect;

describe('GET /users', () => {
  it('should return a list of users', (done) => {
    request(app)
      .get('/users')
      .expect('Content-Type', /json/)
      .expect(200)
      .end((err, res) => {
        if (err) return done(err);
        expect(res.body).to.be.an('array');
        done();
      });
  });
});
```

Рисунок 3.4 – Приклад інтеграційного тесту

Інтеграційні тести пишуться після модульних, коли окремі компоненти системи вже протестовані і можна перевірити їх взаємодію в реальних умовах. Вони є більш складними, ніж модульні, оскільки тестують не лише окремі функції, а й їх сумісність між собою, що дозволяє виявити проблеми, які не проявляються на етапі модульного тестування. Порівняно з модульними тестами, інтеграційні тести охоплюють більшу частину системи, тому вони зазвичай потребують більш складного налаштування середовища, включаючи бази даних або зовнішні сервіси. Однак вони мають значну перевагу в тому, що дозволяють гарантувати правильну роботу програми в цілому, а не лише на рівні окремих функцій. Вони також дають можливість виявити помилки, які можуть виникнути через неправильну взаємодію компонентів, що є важливим аспектом при розробці складних програмних систем.

Інтеграційне тестування має важливий вплив на якість коду, оскільки дозволяє виявляти помилки, які виникають в процесі взаємодії між компонентами системи. Воно забезпечує впевненість, що не лише окремі модулі, а й вся система в цілому працює належним чином. Відсутність інтеграційних тестів може призвести до ситуацій, коли на рівні окремих функцій усе працює добре, але при інтеграції компонентів виникають неочікувані помилки, що важко виявити до цього етапу.

Завдяки інтеграційному тестуванню знижуються ризики таких помилок і забезпечується стабільність програмного забезпечення. Це також дозволяє своєчасно виявляти проблеми з сумісністю між модулями, знижує ймовірність багів при оновленні або зміні окремих частин системи і забезпечує більш високий рівень надійності кінцевого продукту. В кінцевому підсумку, інтеграційне тестування підвищує якість коду, роблячи його більш стійким, ефективним і менш схильним до помилок на етапах реальної експлуатації.

3.2.3 E2E-тестування

E2E-тестування, або тестування кінцевого шляху, є ще одним важливим етапом у забезпеченні якості програмного забезпечення, яке перевіряє всю систему в цілому. На відміну від модульного та інтеграційного тестування, яке фокусується на окремих компонентах або їх взаємодії, E2E-тестування перевіряє програму як єдину цілісну одиницю, з усіма її підсистемами та процесами. Метою таких тестів є перевірка того, чи працює програма згідно з очікуваннями, починаючи від користувацького інтерфейсу і закінчуючи внутрішніми процесами, які відбуваються за лаштунками.

E2E-тестування особливо важливе для перевірки сценаріїв використання програми, коли необхідно перевірити, чи взаємодіють всі частини системи, такі як інтерфейс, сервери, бази даних та зовнішні сервіси. Це дозволяє

переконатися, що користувач може взаємодіяти з додатком так, як передбачено, і що система коректно обробляє запити, взаємодіє з іншими компонентами і виконує всі необхідні функції. Це важливий етап тестування, оскільки він дозволяє побачити, як система працює в реальних умовах.

E2E-тестування зазвичай охоплює найбільш критичні сценарії використання додатка, такі як реєстрація, логін, оформлення замовлення або інші функціональні можливості, що активно використовуються кінцевими користувачами. Тести повинні перевіряти не лише функціональність, а й зручність використання, забезпечуючи перевірку всіх частин програми від початку до кінця.

E2E-тести зазвичай вимагають використання інструментів, які дозволяють автоматизувати взаємодію з користувацьким інтерфейсом та іншими частинами програми, такими як сервери або бази даних. Для тестування веб-додатків популярними інструментами є Cypress, Selenium і Puppeteer. Ці інструменти дозволяють автоматизувати введення даних у форми, натискання кнопок, перевірку результатів на сторінках тощо.

Сценарії E2E-тестування дозволяють перевірити, як система працює в реальних умовах і чи виконується очікувана функціональність. Ось кілька прикладів сценаріїв, які можуть бути використані в E2E-тестуванні:

- Реєстрація нового користувача:
  - Користувач відкриває веб-сторінку реєстрації.
  - Вводить свої особисті дані, такі як ім'я, електронна пошта і пароль.
  - Натискає кнопку "Зареєструватися".
  - Система перевіряє введені дані, створює новий обліковий запис і перенаправляє користувача на сторінку підтвердження реєстрації.
  - Тест перевіряє, чи користувач бачить повідомлення про успішну реєстрацію і чи отримує електронний лист для підтвердження.
- Логін користувача:

  - Користувач переходить на сторінку входу в систему.
  - Вводить свою електронну пошту та пароль.
  - Натискає кнопку "Увійти".
  - Система перевіряє правильність даних і перенаправляє користувача на головну сторінку.
  - Тест перевіряє, чи з'являється на сторінці ім'я користувача, щоб підтвердити успішний вхід.
- Додавання товару в кошик:
  - Користувач відвідує сторінку продукту на веб-сайті інтернет-магазину.
  - Вибирає товар і натискає кнопку "Додати в кошик".
  - Система підтверджує, що товар додано до кошика.
  - Тест перевіряє, чи відображається правильна кількість товарів у кошику та загальна сума.
- Процес оформлення замовлення:
  - Користувач додає товар у кошик і переходить до оформлення замовлення.
  - Вводить адресу доставки, вибирає спосіб оплати.
  - Підтверджує своє замовлення.
  - Система обробляє запит, підтверджує отримання замовлення та надсилає електронний лист.
  - Тест перевіряє, чи з'являється повідомлення про успішне оформлення замовлення і чи є в базі дан

Наприклад, тест за допомогою Cypress може виглядати так (рисунок 3.5):

```
describe('User Login Test', () => {
  it('should allow user to log in successfully', () => {
    cy.visit('https://example.com/login');
    cy.get('input[name="username"]').type('user1');
    cy.get('input[name="password"]').type('password123');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
    cy.contains('Welcome, user1');
  });
});
```

Рисунок 3.5 – Приклад E2E тесту

E2E-тестування відрізняється від модульного та інтеграційного тестування своєю цілісністю та охопленням. Якщо модульне тестування зосереджене на перевірці окремих функцій чи компонентів, а інтеграційне тестування перевіряє взаємодію між ними, то E2E-тестування охоплює весь процес роботи системи, від початку до кінця. Це дозволяє перевірити, як система працює в реальному сценарії, де всі частини взаємодіють одна з одною.

Порівняно з модульними тестами, які зазвичай швидко виконуються і охоплюють лише окремі функціональні блоки, E2E-тести потребують більшої кількості часу та ресурсів, оскільки перевіряють не лише логіку функцій, а й користувацький досвід, взаємодію з базою даних та іншими компонентами. Інтеграційні тести також обмежуються тестуванням певних компонентів, але в цілому вони не охоплюють весь шлях користувача, як це робить E2E-тестування.

E2E-тестування є критично важливим для перевірки функціональності в реальних умовах, що дозволяє гарантувати, що система працює так, як очікується, для кінцевого користувача. Водночас, завдяки вищій складності та більшим вимогам до ресурсів, його виконання може бути більш затратним порівняно з іншими типами тестів. Однак у сукупності з модульним та

інтеграційним тестуванням E2E-тести забезпечують високий рівень стабільності і надійності продукту, гарантуючи, що вся система працює безперебійно і коректно.

Іноді E2E-тестування може бути надмірним, особливо якщо воно охоплює вже перевірені функціональні можливості на рівні модульних або інтеграційних тестів. В таких випадках, проведення повноцінного E2E-тесту може бути витратним з точки зору часу та ресурсів, оскільки він перевіряє взаємодію всіх компонентів системи, включаючи ті, які вже були протестовані на більш низьких рівнях. У таких ситуаціях, коли система вже добре протестована на рівнях модулів і інтеграцій, доцільно знизити кількість E2E-тестів або обмежити їх до найбільш критичних сценаріїв, щоб уникнути зайвих витрат на виконання тестів.

3.3 Інструменти для статичного аналізу коду: можливості та обмеження

Статичний аналіз коду — це процес автоматичної перевірки програмного коду без його виконання. Це дозволяє виявити потенційні помилки, проблеми з безпекою, невідповідності стандартам кодування або навіть покращення для підвищення ефективності. Статичний аналіз використовується для оцінки якості коду на етапах розробки та забезпечення кращого управління технічним боргом. Він дозволяє виявити помилки, які можуть бути не помічені при ручному огляді коду або тестуванні, зокрема логічні помилки, відсутність необхідних перевірок та порушення структурних або стилістичних стандартів.

Інструменти для статичного аналізу коду можуть бути використані в різних контекстах. Наприклад, для перевірки наявності помилок в синтаксисі, для перевірки відповідності до стандартів кодування, для виявлення уразливостей або потенційних проблем з продуктивністю. Вони допомагають

уникнути ситуацій, коли помилки або недоліки коду, що могли бути пропущені на етапі написання, виявляються занадто пізно.

Основною перевагою інструментів статичного аналізу є здатність автоматично виявляти потенційні помилки і проблеми на ранніх етапах розробки, коли зміни до коду ще не мають великого масштабу. Це дозволяє знизити витрати на виправлення помилок, оскільки виявлені проблеми можуть бути виправлені ще до того, як вони переростуть у серйозні помилки на етапі тестування або в продакшн-середовищі.

Інструменти статичного аналізу також дозволяють забезпечити відповідність коду визначеним стандартам якості, що є важливим для підтримки узгодженості та читабельності коду, особливо в командах, де кілька розробників працюють над одним проектом. Вони можуть автоматично застосовувати правила для перевірки форматування, найменування змінних, стилю коментарів тощо.

Багато інструментів також дозволяють інтегруватися в CI/CD процеси, що дає можливість автоматично запускати перевірки на кожному етапі розробки, забезпечуючи високий рівень якості без необхідності вручну перевіряти кожну зміну.

Проте, незважаючи на численні переваги, статичний аналіз має свої обмеження. По-перше, він не може виявити проблеми, які можуть виникнути тільки під час виконання програми, такі як логічні помилки, що залежать від конкретних вхідних даних або взаємодії з іншими компонентами системи. Тому статичний аналіз не може повністю замінити тестування програмного забезпечення, особливо на етапі інтеграції або при проведенні e2e тестів.

Також важливим є те, що інструменти статичного аналізу можуть видавати помилкові сповіщення (false positives), що створює додаткову навантаження на розробників, які повинні вручну перевіряти кожне

сповіщення. Це може знижувати ефективність і призводити до ігнорування сповіщень або пропуску реальних помилок.

Оскільки інструменти для статичного аналізу зазвичай налаштовуються відповідно до конкретних стандартів і правил, вони можуть не охоплювати всі можливі проблеми, що можуть виникнути у специфічних випадках. Наприклад, вони можуть не враховувати специфічні потреби проекту або команди, що може зменшити їх ефективність у деяких випадках.

3.3.1 Використання SonarQube для статичного аналізу коду

SonarQube є одним із найвідоміших і найефективніших інструментів для статичного аналізу коду, який активно використовується командами розробників для виявлення та усунення проблем у програмному забезпеченні. Його головною метою є забезпечення високої якості коду, підвищення безпеки, продуктивності та відповідності сучасним стандартам розробки. Завдяки своїй функціональності та масштабованості цей інструмент підходить для проєктів різного розміру, від невеликих стартапів до великих корпоративних систем.

Однією з ключових функцій SonarQube є можливість оцінювання проєкту відповідно до встановлених критеріїв Quality Gate, що дозволяє командам розробників підтримувати високий рівень якості коду. Quality Gate — це набір порогових значень для різних метрик якості, таких як дублювання коду, кількість багів, уразливості або тестове покриття. Якщо будь-який із цих показників перевищує допустимий рівень, проєкт отримує статус "не пройдено" (Fail). Це дозволяє розробникам одразу визначити, чи відповідає код встановленим стандартам, і, за необхідності, вжити відповідних заходів.

Для кожної мови програмування в проєкті SonarQube призначає Quality Profile, який визначає набір правил, що аналізуються. Наприклад, для JavaScript та інших мов можуть бути налаштовані окремі профілі, що враховують

специфіку синтаксису, безпеки та продуктивності. Quality Gate, у свою чергу, об'єднує ці правила та створює узагальнену метрику, яка дозволяє оцінити стан проєкту на основі аналізу кількох ключових показників. Наприклад, у звіті може бути вказано відсоток дублювання коду, кількість заблокованих модулів через критичні помилки або відсоток покриття тестами. Завдяки цьому підходу команди отримують чіткий і зрозумілий спосіб моніторингу та забезпечення якості.

Цікавою особливістю є інтеграція Quality Gate у CI/CD-процеси. Наприклад, якщо статус Quality Gate для проєкту визначається як "не пройдено", наступні етапи конвеєра інтеграції, такі як деплой чи створення збірки, можуть бути автоматично заблоковані. Це забезпечує додатковий рівень захисту, гарантуючи, що жоден код із критичними проблемами не буде інтегрований у основну гілку проєкту або випущений у продакшн. Такий підхід не лише покращує загальну якість продукту, але й дозволяє командам зосередитися на виявленні та усуненні проблем ще до того, як вони можуть вплинути на кінцевих користувачів.

Яскравим прикладом цієї функціональності є інтерактивний репорт SonarQube, у якому відображається статус проєкту відповідно до Quality Gate. У звіті надаються дані про поточний стан та історичні тренди ключових метрик. Наприклад, у секції "Duplicated Blocks" можна побачити, наскільки часто у коді зустрічається дублювання, і як цей показник змінювався з часом. Це допомагає розробникам не лише оцінити поточний стан проєкту, але й відслідковувати динаміку покращення чи погіршення якості.

У результаті Quality Gate у SonarQube стає центральним елементом процесу забезпечення якості. Завдяки автоматизованій перевірці та чітко визначеним критеріям, команди розробників можуть швидко виявляти критичні проблеми, зменшувати технічний борг і створювати більш стабільні, безпечні та продуктивні програмні рішення.

Наприклад, загальний репорт у SonarQube (рисунок 3.6) містить основну інформацію про стан проєкту, зокрема статус Pass/Fail відповідно до встановлених критеріїв Quality Gate. Цей статус дозволяє одразу визначити, чи відповідає код встановленим стандартам якості. Крім того, звіт містить дані про поточний стан і динаміку змін ключових метрик, таких як баги, уразливості, кодові запахи, покриття тестами та дублювання коду. Окремо відображається вплив останніх змін (Leak Period) на ці метрики, що дає змогу зрозуміти, наскільки ефективно команда впоралася із завданням збереження якості на останньому етапі.

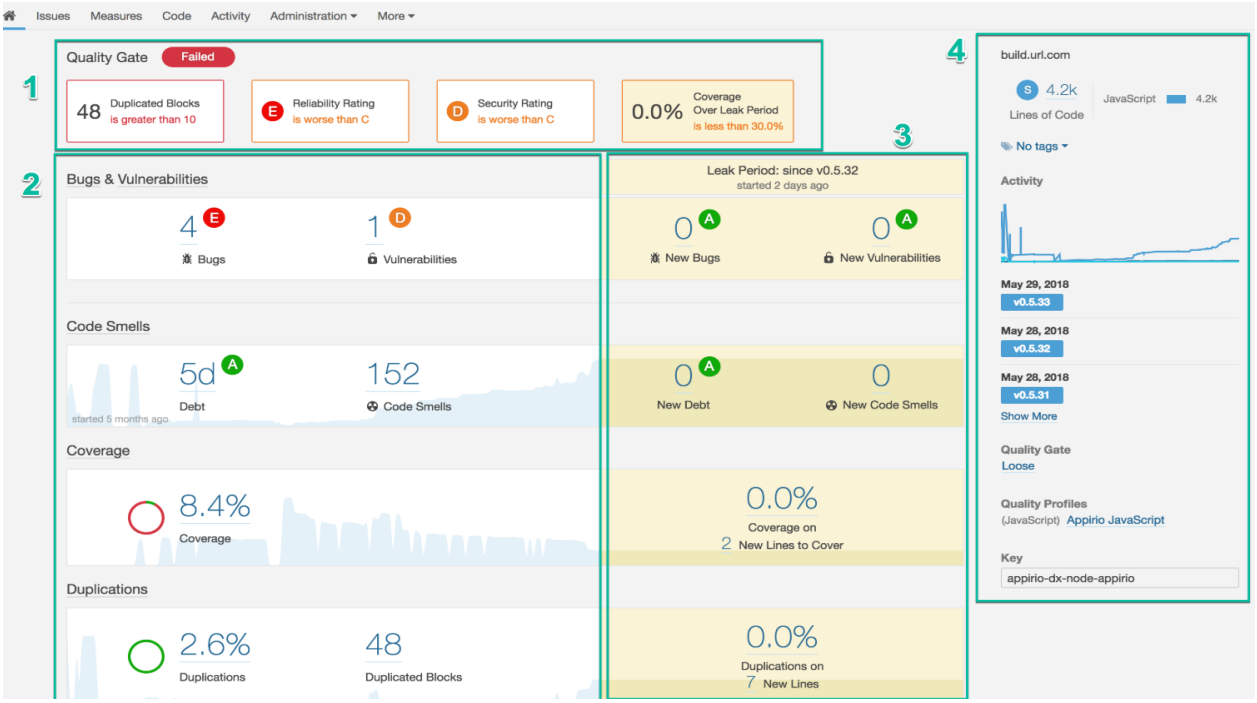

Рисунок 3.6 – Загальний репорт у SonarQube

SonarQube пропонує також детальні засоби для аналізу кожного аспекту якості коду. Вкладка Measures (рисунок 3.7) надає можливість глибокого аналізу показників надійності, безпеки та підтримуваності коду. Наприклад, у секції Reliability розробники можуть виявити помилки, що можуть призводити до відмов у роботі системи, тоді як секція Security дозволяє оцінити ризики, пов’язані з можливими вразливостями. Метрики Maintainability допомагають

визначити ділянки коду, які складно підтримувати або змінювати. Ця інформація дозволяє ефективно планувати процеси рефакторингу, забезпечуючи довгострокову стабільність і гнучкість програмного забезпечення.

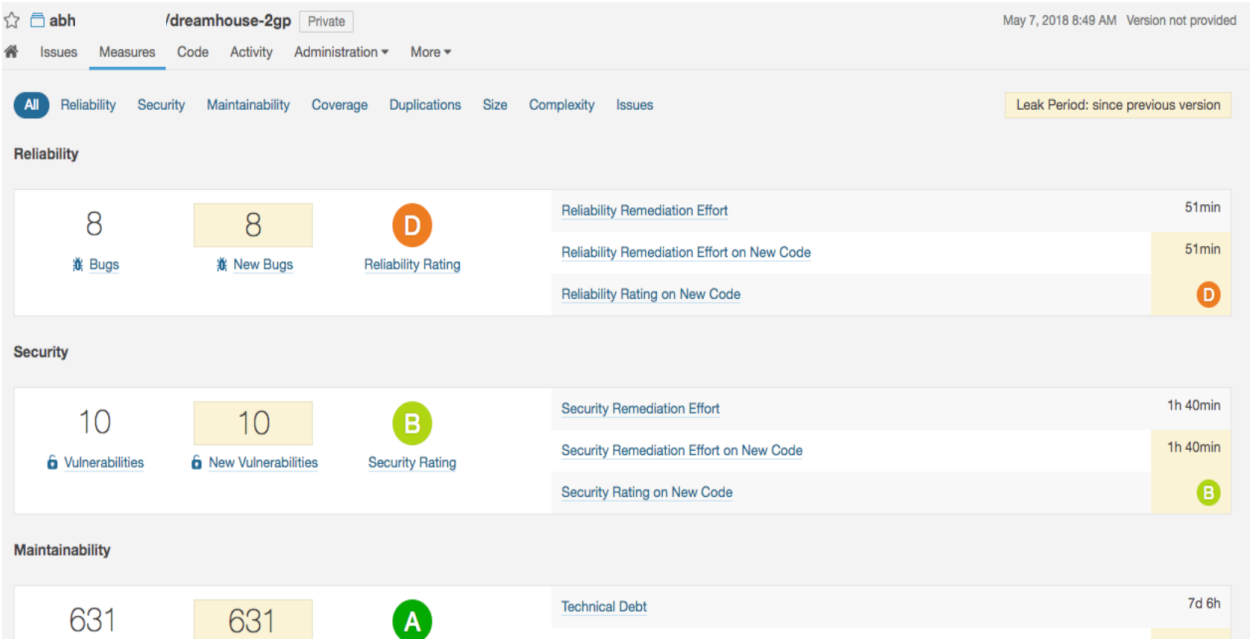

Рисунок 3.7 – Вкладка Measures

Додаткові інструменти SonarQube, такі як сторінка Code (рисунок 3.8), забезпечують зручний доступ до статистики по кожному файлу або папці в проєкті. Тут можна побачити загальну кількість проблем, пов’язаних із багами, уразливостями або кодовими запахами, а також оцінити якість тестування за допомогою показників покриття коду тестами. Ця сторінка також надає дані про дублювання коду, яке є однією з основних причин технічного боргу.

Таблиця зі статистикою по кожному файлу дозволяє швидко знайти критичні ділянки, що потребують уваги команди.

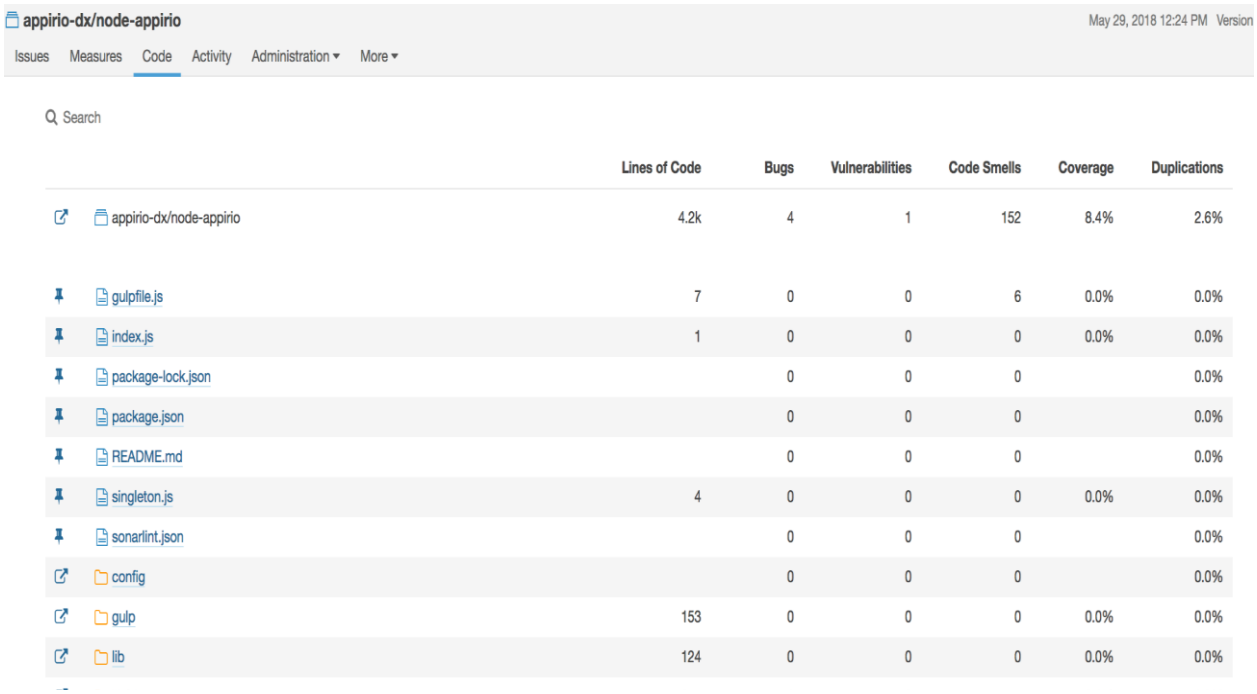

Рисунок 3.8 – Загальний репорт

Важливо відзначити, що SonarQube підтримує історичний аналіз проєкту, що є критично важливим для відстеження прогресу. Завдяки цій функції команди можуть зрозуміти, чи поліпшується якість коду з часом, та виявити проблемні аспекти процесу розробки. Зокрема, функція Leak Period дозволяє зосередитися на аналізі не всього проєкту, а лише змін, внесених у конкретний період. Це значно спрощує виявлення нових проблем, які можуть бути внесені останніми комітами, і знижує ризик накопичення технічного боргу.

SonarQube є універсальним інструментом, який дозволяє командам знаходити баланс між швидкістю розробки та якістю коду. Він допомагає уникнути критичних помилок, знижує витрати на підтримку та розширення програмного забезпечення та створює сприятливі умови для безперервного вдосконалення процесу розробки. Завдяки своєму широкому функціоналу та зручному інтерфейсу цей інструмент залишається незамінним помічником для команд, які прагнуть досягти найвищих стандартів якості.

Процес використання SonarQube у проєкті починається з інтеграції цього інструменту в робочий процес розробників. Спочатку налаштовуються Quality Profiles для кожної мови програмування, яка використовується у проєкті. Це дозволяє визначити набір правил, які будуть застосовуватися під час аналізу коду. Наприклад, для фронтенд-компонентів можуть враховуватися специфічні рекомендації щодо JavaScript, а для бекенд-частини — правила для Node.js.

Після цього проєкту призначається Quality Gate, який об'єднує всі основні метрики, включаючи рівень тестового покриття, кількість багів, уразливостей, дублювання коду та інші показники. Quality Gate виступає фільтром, що визначає загальний стан проєкту на кожному етапі розробки.

На наступному етапі, при внесенні змін у код, він автоматично сканується за допомогою SonarQube. Цей процес зазвичай інтегрований у CI/CD-конвеєр: кожен пуш або пул-реквест викликає аналіз, який генерує звіт. Спершу розробники отримують детальний репорт про статус проєкту, включаючи Pass/Fail відповідно до Quality Gate. Вони бачать, як останні зміни вплинули на ключові метрики, наприклад, чи зросла кількість кодових запахів або уразливостей. Одночасно система генерує історичний огляд, дозволяючи простежити, чи поліпшується якість коду з часом.

Коли знайдені проблеми, розробники можуть поглиблено вивчати їх через вкладку Measures, яка дозволяє деталізувати показники надійності, безпеки та підтримуваності. Вони аналізують конкретні файли та папки у вкладці Code, де кожен елемент має свій статус: кількість багів, відсоток тестового покриття, дублювання коду. Це дає змогу точково виправляти проблеми там, де вони мають найбільший вплив.

Якщо проєкт проходить усі вимоги Quality Gate, наступні етапи CI/CD-конвеєра продовжуються автоматично, наприклад, створення збірки або розгортання у тестовому середовищі. У разі невдачі розробники отримують

швидкий сигнал про необхідність виправлень, що мінімізує ризики інтеграції проблемного коду.

Завдяки цьому підходу команди не лише підвищують якість свого продукту, але й зменшують технічний борг, оскільки всі критичні проблеми вирішуються ще на етапі розробки. SonarQube стає центральним інструментом, що сприяє прозорості процесу, автоматизації перевірок і довгостроковому покращенню проєктів.

Хоча SonarQube є потужним інструментом для забезпечення якості коду, у нього є деякі мінуси та обмеження, які можуть вплинути на його ефективність залежно від специфіки проєкту та команди.

По-перше, ресурсозатратність. SonarQube може вимагати значних ресурсів для сканування великих проєктів, особливо якщо аналізуються великі кодові бази з великою кількістю файлів. Це може призвести до збільшення часу на виконання аналізу, що в свою чергу може сповільнити процес CI/CD, якщо проєкт не оптимізований або налаштований неправильно. Враховуючи вимоги до ресурсів, це може бути проблемою для невеликих команд або стартапів з обмеженими можливостями інфраструктури.

По-друге, помилки та неточності у аналізі. SonarQube, як і будь-який інший інструмент статичного аналізу, не завжди може точно визначити всі можливі проблеми у коді. Іноді він може видавати помилки, які не мають значного впливу на функціональність або безпеку програми, а іноді може ігнорувати деякі реальні уразливості чи баги, які залишаються непоміченими через обмеження в налаштуванні або специфіці коду.

Третій мінус полягає в підтримці інструменту. Для новачків у налаштуванні SonarQube це може бути складним завданням, особливо на етапі інтеграції з CI/CD. Неправильне налаштування може призвести до неповних або неякісних звітів, що знижує ефективність інструменту. Враховуючи це,

необхідно вкласти час у навчання та адаптацію, що може бути додатковим навантаженням для команди.

Також варто зазначити, що SonarQube не замінює ручне тестування чи рев'ю коду. Він є доповненням до цих процесів, а не їхньою заміною. В деяких випадках автоматичний аналіз не здатний замінити детальну перевірку розробників, особливо коли мова йде про складні логічні помилки або специфічні проблеми, що виникають лише в конкретних умовах. Тому важливо розглядати SonarQube як частину більш широкої стратегії забезпечення якості, а не як єдиний інструмент.

Загалом, SonarQube є дуже корисним інструментом для підтримки високої якості коду, але його ефективність залежить від правильного налаштування, ресурсів і доповнення до інших процесів перевірки коду.

3.4 Використання лінтерів

Лінтери — це інструменти статичного аналізу коду, які автоматично перевіряють вихідний код на відповідність встановленим правилам, стандартам або домовленостям. Головна мета лінтерів — виявляти помилки, попереджати про потенційні проблеми та забезпечувати уніфікацію коду, що особливо важливо під час командної роботи.

Лінтери виконують аналіз вихідного коду без його запуску. Вони перевіряють:

- Синтаксичні помилки. Наприклад, пропущені крапки з комою, неправильне використання дужок чи інші порушення.
- Логічні помилки. Інструмент може вказати, якщо умова завжди є хибною або істинною.

- Стандарти коду. Лінтер виявляє, чи відповідає код стилю, який встановлений у проєкті, наприклад, щодо відступів, використання лапок чи форматування.
- Результатом роботи лінтера є список попереджень або помилок, які потрібно виправити. Часто ці помилки можна усунути автоматично за допомогою функції автофіксу, яку підтримують деякі лінтери.

Приклади лінтерів:

- ESLint — популярний лінтер для JavaScript, який допомагає виявляти синтаксичні помилки, перевіряє стиль коду та сприяє покращенню його якості.
- Stylelint — інструмент для аналізу CSS та його препроцесорів (Sass, Less), який перевіряє стилі на відповідність заданим правилам.
- Pylint — лінтер для Python, що допомагає дотримуватись стандартів кодування, таких як PEP 8, і виявляє помилки у коді.
- TSLint — інструмент для перевірки коду TypeScript (нині замінений на ESLint із підтримкою TypeScript).
- Flake8 — ще один лінтер для Python, який перевіряє стиль коду та шукає помилки.

Лінтери працюють за принципом аналізу вихідного коду на відповідність заданим правилам та стандартам. Ці правила визначаються спеціальними конфігураційними файлами, які можуть бути створені вручну або базуватися на популярних стилістичних гідах, таких як Airbnb чи Google. Під час роботи лінтер сканує кожен рядок коду, перевіряючи його на синтаксичні, логічні або стилістичні помилки. Коли інструмент виявляє порушення правил, він підсвічує відповідні ділянки коду та виводить пояснення, яке допомагає розробнику зрозуміти, що саме потрібно виправити. Лінтери є особливо корисними на ранніх етапах розробки, коли ще немає можливості провести глибоке

тестування, а також у великих командах, де уніфікація стилю написання коду має вирішальне значення. Водночас є ситуації, коли правила лінтера можуть бути надто суворими або неактуальними для певного проєкту. У таких випадках лінтери дозволяють вказати виключення — наприклад, ігнорувати певні файли, рядки коду або окремі правила. Крім того, у разі виникнення необхідності можна тимчасово вимкнути лінтер, наприклад, для перевірки складного або експериментального коду, який ще не відповідає стандартам. Це забезпечує гнучкість у роботі, дозволяючи розробникам адаптувати лінтер під специфіку проєкту, зберігаючи при цьому баланс між суворістю перевірок та продуктивністю роботи.

Завдяки лінтерам кожен учасник команди може дотримуватись єдиних стандартів, що спрощує спільну роботу над одним кодовим базисом. Це зменшує кількість конфліктів у коді під час об'єднання змін, а також полегшує читання та розуміння коду іншими членами команди.

Налаштування лінтерів є важливим аспектом їх використання. Команди можуть почати з готових шаблонів правил, таких як ESLint для JavaScript чи Pylint для Python, але налаштувати їх під специфічні потреби проєкту. Наприклад, можна додати правила для перевірки іменування змінних, використання коментарів або форматування коду. Ці конфігурації зазвичай зберігаються у файлах конфігурації, що є частиною репозиторію, щоб усі члени команди мали доступ до єдиних стандартів. Це забезпечує синхронність між розробниками, незалежно від їхнього середовища розробки.

Крім того, сучасні лінтери часто інтегруються з редакторами коду, такими як Visual Studio Code чи IntelliJ IDEA, що дозволяє отримувати миттєвий зворотний зв’язок під час написання коду. Це значно підвищує ефективність роботи, адже розробник може виправляти помилки ще до того, як зберігає файл або запускає тестування. Лінтери також інтегруються з CI/CD процесами, перевіряючи код автоматично під час кожного коміту або перед

злиттям у основну гілку. Це допомагає уникнути помилок, які могли б потрапити до продуктивного середовища.

Незважаючи на користь, яку надають лінтери, їхнє використання вимагає зваженого підходу. Занадто суворі правила можуть уповільнити розробку, викликаючи роздратування у розробників. У таких випадках доцільно переглянути конфігурації, зменшити кількість обов'язкових правил або додати виключення для специфічних ситуацій. Наприклад, для деяких експериментальних функцій можна створити окрему гілку, де лінтер працюватиме у менш строгому режимі. Це дозволяє зберігати гнучкість, не втрачаючи загального контролю якості.

Загалом, лінтери не лише виявляють помилки, а й привчають розробників до дисципліни. Вони допомагають новим членам команди швидше адаптуватися до правил написання коду, що використовується у проєкті. Завдяки цьому нові співробітники швидше стають повноцінними учасниками розробницького процесу, підвищуючи загальну продуктивність команди.

Вибір спільних налаштувань для лінтерів є критично важливим для будь-якої команди розробників. Цей процес зазвичай починається з обговорення серед учасників команди, під час якого визначаються базові правила форматування та стилю коду, які відповідають специфіці проєкту. Для створення консистентного середовища всі учасники повинні дотримуватись однієї конфігурації, яка зберігається у файлі конфігурації лінтера, такому як .eslintrc.json або .pylintrc, що є частиною репозиторію. Таким чином, кожен розробник автоматично працює за тими ж правилами, незалежно від використовуваного середовища розробки чи особистих уподобань.

Консистентність у написанні коду має вирішальне значення для якості проєкту. По-перше, це забезпечує легкість читання коду. Якщо всі дотримуються однакових правил, код виглядає так, ніби його написала одна людина, навіть якщо над ним працювала велика команда. Це значно спрощує

процес рецензування, пошуку помилок та внесення змін. По-друге, консистентність знижує ризик виникнення конфліктів під час об'єднання змін із різних гілок. Коли всі слідують одним і тим же стандартам, зменшується кількість дрібних відмінностей у коді, які можуть створити проблеми.

Важливість консистентності також проявляється в довготривалій підтримці коду. Через місяці або роки після завершення основної фази розробки нові члени команди або сторонні розробники можуть легко зрозуміти, як організовано код, якщо він слідує єдиним правилам. Це зменшує час на навчання та пошук необхідної інформації.

Головна мета спільних налаштувань – не ідеальний стиль коду, а спрощення спільної роботи та забезпечення стабільності. Саме тому команда повинна враховувати реальні потреби проєкту та зосереджуватися на тих правилах, які дійсно приносять користь, не перевантажуючи процес розробки.

3.4 Використання форматерів коду

Форматери коду є важливим інструментом для забезпечення стандартизації вигляду коду в межах проєкту. Їх основна мета — автоматичне виправлення відступів, розташування дужок, пробілів та інших аспектів синтаксису, які не впливають на логіку виконання програми, але суттєво полегшують її читання та сприйняття.

Форматери, на відміну від лінтерів, не вказують на помилки чи порушення стилю, а одразу виправляють їх відповідно до попередньо налаштованих правил. Це значно спрощує роботу розробників, оскільки їм не потрібно вручну коригувати кожну дрібницю. Наприклад, у JavaScript проєктах форматери, такі як Prettier, автоматично додають або прибирають коми, вирівнюють код за необхідними відступами та забезпечують правильне розташування дужок у функціях або структурах даних.

Форматери інтегруються з редакторами коду або викликаються як окремий інструмент через командний рядок. Коли розробник зберігає файл, форматер автоматично аналізує структуру коду та приводить його у відповідність до заданих правил. Ці правила зберігаються в конфігураційних файлах, таких як .prettierrc або .editorconfig, що дозволяє легко поділитися ними з усією командою.

Автоматичне застосування форматера дозволяє уникнути конфліктів під час код-рев'ю, коли розробники витрачають час на обговорення стилістичних питань замість логічної частини коду.

Вибір і налаштування форматера є важливим процесом, оскільки кожна команда має свої стилістичні вподобання. У Prettier, наприклад, можна вказати максимальну довжину рядка, необхідність крапок з комами або форматування лапок. Незважаючи на різноманіття можливостей, важливо дотримуватись простого правила: чим менше кастомізації, тим краще. Стандартні налаштування зазвичай підходять для більшості проєктів і знижують ризик виникнення суперечок у команді.

Однією з найбільших переваг форматерів є забезпечення консистентності коду. Завдяки автоматичному застосуванню правил кожен учасник команди може бути впевнений, що його код виглядає так само, як і код його колег. Це не лише полегшує процес читання, але й сприяє швидшому виявленню логічних помилок, які могли залишитись непомітними серед неакуратного або хаотичного коду.

Також форматери допомагають уникнути зайвих змін у репозиторії. Якщо кожен файл буде автоматично відформатований перед збереженням, у комітах не буде зайвих змін, пов’язаних із форматуванням, що спрощує аналіз історії змін. Наприклад, у системах на зразок Git завдяки форматеру коміт із виправленням функції не міститиме випадкових змін у відступах, які ускладнюють читання диференціалу.

Коли варто використовувати форматери? Форматери варто використовувати в будь-якому проєкті, де працює більше одного розробника. Це дозволяє уникнути суб’єктивності у визначенні "правильного" стилю коду і забезпечує єдність підходу. Навіть у маленьких командах використання форматера скорочує час на ручне виправлення стилістичних помилок, звільняючи час для важливіших завдань.

Форматери також можуть бути інтегровані в конвеєр безперервної інтеграції. Перед виконанням тестів система може запускати перевірку або автозастосування форматування до всіх файлів. Це забезпечує відповідність усіх змін установленим правилам, навіть якщо хтось із учасників забув застосувати форматер локально. У GitLab або GitHub можна налаштувати автоматичне форматування файлів перед їх об’єднанням у основну гілку.

Використання форматерів, таких як Prettier, разом із лінтерами (наприклад, ESLint) створює потужний тандем для підтримки якості та консистентності коду в проєктах. Обидва інструменти мають різні завдання, але їхнє комбіноване застосування допомагає уникнути конфліктів стилю та забезпечує високий рівень якості коду.

Лінтери аналізують код і знаходять логічні, стилістичні або структурні помилки, зокрема використання застарілих конструкцій, неправильний порядок викликів методів чи дублювання логіки. Форматери, у свою чергу, фокусуються на зовнішньому вигляді коду: правильних відступах, розташуванні дужок, лапках і крапках з комами. Використання обох інструментів дозволяє вирішувати ці завдання автоматично, зменшуючи кількість ручної роботи.

Щоб уникнути конфліктів між форматером і лінтером, важливо правильно налаштувати їхню інтеграцію. Prettier можна налаштувати як частину ESLint за допомогою плагінів, таких як `eslint-plugin-prettier`. У такій конфігурації ESLint не лише перевіряє логіку, але й викликає Prettier для перевірки стилістичних правил. Це забезпечує єдиний підхід до перевірки коду.

Наприклад, якщо Prettier відповідає за розставляння крапок із комами, ESLint автоматично пропустить перевірку цього правила, щоб уникнути дублювання. Це зручно, оскільки розробники можуть зосередитися на логічних і структурних помилках, не хвилюючись про форматування.

Переваги використання разом:

- Автоматизація та економія часу.
  Prettier автоматично виправляє стилістичні помилки, а ESLint повідомляє про проблеми, які не можуть бути виправлені автоматично (наприклад, небезпечні виклики або помилкову логіку). Розробники менше часу витрачають на рутинні правки.
- Єдність підходу.
  Інтеграція цих інструментів допомагає уникнути суперечок у команді щодо стилю коду. Розробники працюють за єдиними правилами, а форматування й перевірка відбуваються автоматично, зменшуючи ризик "людського фактора".
- Покращення процесу код-рев’ю.
  Код, що проходить через лінтери та форматери, виглядає акуратніше й чіткіше. Це дозволяє учасникам команди більше уваги приділяти логічним і функціональним аспектам під час рев’ю, а не обговорювати форматування.

4 ОЦІНЮВАННЯ ЯКОСТІ КОДУ ТА ІНТЕРПРЕТАЦІЯ РЕЗУЛЬТАТІВ

Вимірювання якості коду є ключовим етапом у розробці програмного забезпечення, оскільки дозволяє визначити рівень стабільності, ефективності та надійності програмного продукту. Це важливий процес, що включає не лише оцінку функціональних характеристик коду, а й його структури, читабельності та можливості для подальшої підтримки. Однак якість коду є багатогранним поняттям, яке важко виміряти однозначно. Для цього використовуються різноманітні метрики та інструменти, що дозволяють оцінити як окремі аспекти, так і загальну ефективність.

Деякі аспекти якості коду можна точно виміряти у кількісному вираженні, наприклад, покриття тестами, час виконання, кількість помилок або уразливостей. Ці параметри дають чітке уявлення про рівень готовності коду до продакшн-середовища та його здатність витримувати навантаження. Вони вимірюються за допомогою спеціальних інструментів і можуть бути наочно представлені у вигляді статистичних даних, таких як відсотки, час або кількість виявлених дефектів. Такі метрики дозволяють швидко і точно оцінити певні аспекти якості коду і приймати рішення про подальшу його оптимізацію.

Проте існують також аспекти, які складно оцінити кількісно. Наприклад, читабельність коду, підтримуваність, масштабованість або надійність зазвичай потребують експертної оцінки. Хоча існують певні інструменти для перевірки стилю та структурних помилок, справжню оцінку "зрозумілості" або "підтримуваності" коду часто можуть надати лише досвідчені розробники, які мають глибше розуміння архітектури програми та її можливостей у реальних умовах. Визначення того, чи легко буде модифікувати або масштабувати систему в майбутньому, також залежить від багатьох факторів, таких як структурованість коду, використання патернів проектування та його здатність адаптуватися до нових вимог.

Таким чином, процес вимірювання якості коду є поєднанням кількісних і якісних оцінок, де автоматизовані інструменти допомагають виявити конкретні помилки і дефекти, а експертне оцінювання дає змогу побачити повну картину якості коду з огляду на його загальну архітектуру, зручність у підтримці та здатність до розширення.

Отримання чітких цифр, які оцінюють якість коду, здійснюється завдяки використанню різноманітних алгоритмів і інструментів, які аналізують код на різних рівнях. Це дозволяє зібрати статистику і метрики, які точно відображають певні аспекти якості коду, і приймати на їх основі обґрунтовані рішення. Алгоритми для отримання цих цифр зазвичай працюють на основі статичного або динамічного аналізу коду, використовуючи правила та моделі для оцінки його характеристик.

Наприклад, для вимірювання покриття тестами використовуються спеціалізовані інструменти, такі як JaCoCo для Java або Istanbul для JavaScript. Алгоритм отримання цієї метрики зазвичай полягає в тому, що інструмент запускає тести і перевіряє, які частини коду були виконані під час тестування. Збираючи ці дані, інструмент підраховує відсоток рядків або гілок коду, які були покриті тестами. Наприклад, якщо з 1000 рядків коду виконано 800 під час тестування, покриття становить 80%. В результаті цієї операції формується точне значення, яке можна використовувати для визначення якості тестування і оцінки надійності коду.

Для покращення продуктивності коду алгоритми вимірюють час виконання різних частин програми, використовуючи профайлери. Такі інструменти, як JProfiler або Xcode Instruments, аналізують витрати часу на кожну функцію чи метод, а потім створюють статистику, яка показує, яка частина коду є найбільш ресурсозатратною. Наприклад, ці інструменти можуть визначити, що певна функція займає 30% загального часу виконання програми, що дозволяє розробникам прийняти рішення про її оптимізацію.

Кількість помилок та вразливостей зазвичай визначається через статичний аналіз коду. Інструменти, такі як SonarQube, Checkstyle або PMD, використовують вбудовані алгоритми для виявлення проблем у коді. Алгоритм працює таким чином: інструмент сканує код, порівнюючи його з набором правил (наприклад, перевірка на наявність неініціалізованих змінних, використання застарілих функцій, можливі уразливості, такі як SQL-ін'єкції). Виявлені проблеми потім підраховуються та звітуються, надаючи чітке число помилок. Це дозволяє отримати точні цифри щодо якості коду в термінах наявності дефектів.

Дублювання коду визначається за допомогою алгоритмів для пошуку схожих або ідентичних фрагментів у різних частинах проєкту. Програми, як SonarQube, застосовують алгоритми для порівняння великих фрагментів коду та пошуку повторюваних блоків. Алгоритм зазвичай здійснює порівняння хешів або виводить схожість рядків коду, фрагментів функцій чи класів. Результатом є кількість дубльованих рядків або блоків коду, що дозволяє отримати точну метрику для оцінки технічного боргу.

Для вимірювання складності коду, зокрема цикломатичної складності, використовуються алгоритми, які аналізують кількість незалежних шляхів через програму. Цей метод був розроблений для визначення складності програмних алгоритмів і дозволяє отримати чітке числове значення, яке показує, скільки умовних операторів (if, while, for тощо) міститься у функції або методі. Збільшення цикломатичної складності свідчить про збільшення складності програми та ймовірність наявності помилок.

Алгоритми, що використовуються для вимірювання якості коду, спроектовані таким чином, щоб забезпечити точність результатів. Наприклад, інструменти для статичного аналізу використовують точно визначені правила для виявлення проблем, що дає змогу отримати чіткі цифри щодо кількості

помилок, потенційних уразливостей та порушень стандартів кодування. Тому ці метрики є об'єктивними та вимірюваними в точних числових значеннях.

Однак важливо зазначити, що деякі метрики, як-от читабельність або підтримуваність, не можна виміряти через алгоритми з такою точністю, оскільки вони вимагають оцінки з боку людини, яка враховує контекст, логічну організацію та зручність змін. Тим не менш, комбінація автоматизованих алгоритмів і людської експертизи дозволяє створити повну картину якості коду та допомагає розробникам досягти високого рівня стабільності та надійності програмного продукту.

4.1 Неточності оцінки показників якості коду та їх суб’єктивність

Хоча метрики якості коду, отримані за допомогою автоматизованих інструментів, можуть дати корисне уявлення про загальний стан коду, ці цифри часто не є абсолютно точними та не можуть повною мірою відобразити реальну якість коду. Це пов'язано з кількома факторами, що впливають на точність вимірювань.

По-перше, алгоритми для вимірювання складності або дублювання коду часто базуються на визначених правилах і шаблонах. Наприклад, алгоритм для вимірювання цикломатичної складності може дати високі результати, якщо в коді є багато умовних операторів, але це не завжди вказує на погану якість. Можливо, ці умови необхідні для вирішення складної задачі, і складність є обґрунтованою. Тому хоча цифри можуть бути високими, вони не враховують контексту, в якому використовуються ці конструкції.

По-друге, статичний аналіз коду може виявити численні помилки, уразливості чи порушення стандартів кодування, але він не може охопити всі можливі ситуації, які можуть виникнути під час виконання програми. Наприклад, інструменти для статичного аналізу можуть вказати на проблеми з

використанням старих API, але не завжди можуть точно виявити, як ці проблеми впливають на продуктивність або функціональність програми в реальних умовах. Вони також не можуть врахувати специфічні умови або залежності, що виникають під час виконання коду, і тому можуть пропускати деякі проблеми.

Ще однією проблемою є помилкові сповіщення (false positives), що виникають через специфічність налаштувань інструментів. Наприклад, лінтери та форматери можуть часто підсвічувати дрібні порушення, які насправді не впливають на якість коду чи його функціональність. Це може призводити до зайвих виправлень і створювати враження, що код має більше проблем, ніж є насправді. Більш того, в залежності від налаштувань лінтера або форматера, одні й ті самі частини коду можуть бути оцінені по-різному.

Ще один важливий аспект — це врахування контексту. Багато метрик, як от тестове покриття чи кількість помилок, дають лише часткову картину якості коду. Наприклад, високе покриття тестами не завжди гарантує, що код насправді працює коректно або що всі можливі сценарії тестуються. Тести можуть охоплювати лише базові випадки або бути неякісними, тому високий відсоток покриття тестами не є достатнім доказом якості. Подібним чином, наявність багів у коді не завжди означає, що вони критичні для програми, адже деякі помилки можуть бути непримітними або не мати великого впливу на загальну функціональність.

Тому важливо розглядати ці цифри як орієнтири, а не як абсолютні істини. Вони можуть допомогти виявити очевидні проблеми та області для покращення, але їх слід використовувати разом з іншими підходами, такими як код-рев'ю, тестування, аналіз з боку досвідчених розробників, щоб отримати більш точну оцінку якості коду.

Цифри, які ми отримуємо в результаті вимірювання якості коду, можуть суттєво залежати від інструменту, який використовується для цього аналізу. Це

є важливим фактором, який потрібно враховувати при інтерпретації результатів, оскільки різні інструменти можуть застосовувати різні алгоритми, правила і підходи для обробки того ж самого коду. Тому одні й ті самі метрики можуть виглядати по-різному в залежності від того, який інструмент вибраний для аналізу.

По-перше, різні інструменти можуть застосовувати різні правила для перевірки якості коду. Наприклад, один лінтер може вимагати обов’язкового використання певних стилістичних практик (наприклад, відступів або лапок), тоді як інший може мати інші вимоги. Це може призвести до того, що один і той самий фрагмент коду буде оцінений по-різному в різних інструментах. Тому, коли ми отримуємо цифри, що відображають кількість помилок або порушень, важливо пам'ятати, що вони відображають відповідність конкретним налаштуванням цього інструменту, а не абсолютну якість коду.

По-друге, різні інструменти можуть мати різний рівень чутливості до певних типів помилок. Наприклад, один інструмент може бути настроєний на виявлення лише найбільш очевидних помилок, таких як синтаксичні помилки або відсутність тестів, тоді як інший може також виявляти менш помітні проблеми, як-от логічні помилки чи порушення більш складних стандартів кодування. Це означає, що один інструмент може виявити більше проблем і, відповідно, показати більш негативну статистику, ніж інший, навіть якщо якість коду насправді не є настільки поганою.

Крім того, алгоритми аналізу та точність вимірювань можуть відрізнятися між інструментами. Наприклад, інструмент для вимірювання покриття тестами може використовувати різні підходи для визначення того, яка частина коду була протестована. Один інструмент може рахувати покриття лише на рівні рядків, а інший може враховувати покриття умов і гілок коду. Це може призвести до значних відмінностей у результатах, навіть якщо обидва інструменти аналізують один і той самий набір тестів.

Не менш важливою є інтеграція інструментів у процес розробки. Деякі інструменти можуть бути налаштовані так, щоб працювати на рівні окремих файлів або комітів, а інші — на рівні всього проєкту. Це впливає на те, як вимірюються метрики. Наприклад, інструмент, який аналізує лише певні файли або ділянки коду, може показати відмінні результати порівняно з інструментом, який оцінює увесь проєкт цілком.

Таким чином, важливо враховувати, що цифри, які отримуємо за допомогою одного інструменту, можуть відрізнятися від цифр, отриманих за допомогою іншого, через різні підходи до аналізу. Це підкреслює необхідність стандартизації та узгодження інструментів для вимірювання якості коду в команді або проєкті, щоб уникнути непорозумінь і забезпечити точність оцінок. Для об'єктивної оцінки якості коду слід використовувати комплексний підхід, що включає різні інструменти та методи, а також ручну перевірку за участю досвідчених розробників.

4.2 Інтерпретація показників залежно від специфіки проєкту

Оцінка показників якості коду завжди повинна враховувати специфіку конкретного проєкту. Чіткі метрики, такі як покриття тестами, кількість помилок або час виконання, можуть надавати корисну інформацію, але важливо розуміти, що ці цифри мають бути інтерпретовані в контексті завдань, технологій і обмежень, які існують у кожному проєкті. Кожен проєкт має свої вимоги, і стандарти якості можуть сильно відрізнятися в залежності від сфери застосування, команди та її цілей.

Наприклад, покриття тестами може бути критичним для одного проєкту, наприклад, для фінансових чи медичних додатків, де помилки можуть призвести до серйозних наслідків. В таких проєктах дуже високий рівень покриття тестами є обов'язковим, і навіть 90% покриття може виявитися

недостатнім для забезпечення високої якості продукту. У той же час, для менш критичних проєктів, таких як внутрішні інструменти або прототипи, покриття тестами може бути менш важливим, і на ньому не слід зосереджуватися, якщо це не допомагає досягти основних цілей проєкту.

Час виконання — важливий параметр для проєктів, де висока продуктивність є критичною (наприклад, для онлайн-торгівлі чи великих наукових обчислень). В таких випадках час виконання операцій і оптимізація алгоритмів може бути пріоритетом. Однак для малих або середніх проєктів, де вимоги до швидкості не є настільки жорсткими, оптимізація часу виконання може бути відкладена на більш пізні етапи, оскільки інші аспекти, такі як зручність розробки і підтримки, можуть бути важливішими.

Читабельність і підтримуваність коду є особливо важливими в командних проєктах, де код пишуть кілька розробників. Вимірювання читабельності часто включає оцінку на основі стилістичних вимог, таких як використання імен змінних, коментарів і структури коду. Однак ці аспекти можуть не бути такими важливими в малих індивідуальних проєктах, де швидкість розробки може бути важливішою за строгі стандарти стилю. В таких випадках необхідно визначити компроміс між швидкістю розробки і довгостроковою підтримуваністю коду.

Ще одним важливим аспектом є безпека. Якщо проєкт стосується обробки чутливих даних або працює в умовах високих вимог до конфіденційності (наприклад, банківські додатки, онлайн-банкінг), то наявність вразливостей і загроз повинна бути критично важливою метрикою. У такому випадку навіть незначні проблеми безпеки можуть стати серйозною загрозою для проєкту. Проте для внутрішніх інструментів, що не працюють з чутливими даними, ці проблеми можуть мати менший пріоритет, що дозволяє зосередитися на інших аспектах якості.

Технічний борг також варіюється залежно від проєкту. Для стартапів або прототипів, де важлива швидка розробка, можна відкласти вирішення деяких

проблем із технічним боргом. В інших проєктах, де є потреба в довгостроковій підтримці і масштабуванні, технічний борг повинен бути мінімізований, і робота над його усуненням є пріоритетом.

Загалом, оцінка показників якості коду повинна здійснюватися з урахуванням конкретних вимог та цілей проєкту. Тільки так можна прийняти правильне рішення щодо того, що є критично важливим для забезпечення якості, а що можна відкласти на більш пізні етапи. Визначення пріоритетів і адаптація до умов конкретного проєкту дозволяє зробити вимірювання якості коду ефективним інструментом для досягнення високих результатів при розробці програмного забезпечення.

Наприклад, у стартапах часто стикаються з ситуацією, коли швидкість розробки та запуску продукту важливіші за ідеальну якість коду. Наприклад, у стартапі, що працює над мінімально життєздатним продуктом (MVP), метою є швидке виведення продукту на ринок для того, щоб перевірити гіпотези, залучити інвесторів чи отримати зворотний зв'язок від користувачів. В такому випадку конкретні цифри, такі як покриття тестами, цикломатична складність або дотримання строгих стандартів коду, можуть бути менш важливими. Тут головне — це швидкість реалізації основних функцій продукту, навіть якщо деякі частини коду написані без оптимізації або мають низьке покриття тестами. Це може бути обґрунтованим вибором, адже важливіше провести тестування ідеї, а не витрачати багато часу на детальну перевірку якості коду, яка може бути не критичною на початкових етапах. Наприклад, якщо продукт не знайде попиту або не відповість на основні потреби користувачів, витрачені зусилля на детальну оптимізацію можуть бути марними.

Проте для довгострокового проєкту, який планується до підтримки і розвитку впродовж багатьох років, особливо у великих компаніях, важливість конкретних метрик якості коду набагато вища. Тут слід враховувати не лише функціональність, але й підтримуваність**,** масштабованість **та** відсутність

технічного боргу**.** Кожна частина коду повинна бути ретельно перевірена і оптимізована, оскільки з часом кількість помилок і проблем з підтримкою може сильно збільшитись, якщо заздалегідь не забезпечити належну якість. У таких проєктах критично важливим є забезпечення високого покриття тестами, правильне документування коду, дотримання стандартів і реалізація коду, який легко масштабувати та розвивати в майбутньому.

Отже, в стартапах можуть бути менш важливими метрики, що стосуються коду, такі як покриття тестами чи цикломатична складність, а також можуть бути допущені деякі компроміси щодо стилістичних стандартів. Водночас у великих довгострокових проєктах потрібно активно враховувати технічний борг, покриття тестами, продуктивність і масштабованість, що забезпечить стабільний розвиток і підтримку продукту в майбутньому.

ВИСНОВКИ

У роботі ми розглянули різні способи забезпечення якості коду, такі як код-рев'ю, написання тестів, статичний аналіз коду, дотримання стандартів і впровадження принципів SOLID. Кожен з цих підходів є важливим для підтримки високого рівня якості коду, однак їхня ефективність і важливість залежать від конкретного контексту проєкту, вимог до продукту та наявних ресурсів.

Також ми детально розглянули методи вимірювання якості коду, які допомагають кількісно оцінити ефективність розробки. Вимірювання покриття тестами, продуктивності, складності коду, кількості помилок і вразливостей є основними метриками для оцінки якості коду. Однак ми також з'ясували, що ці цифри можуть бути не завжди точними або можуть залежати від конкретного інструмента, що використовується для їх збору. Важливо пам'ятати, що певні метрики не враховують всі нюанси контексту та не можуть замінити експертну оцінку, яка часто є необхідною для прийняття остаточних рішень.

Нюанси оцінки якості коду можуть значно змінюватися залежно від вимог проєкту, його стадії розвитку, а також рівня експертизи команди. У стартапах чи на ранніх етапах розробки важливіше зосередитися на швидкому створенні продукту та тестуванні ідеї, ніж на дотриманні всіх можливих стандартів якості коду. Водночас у великих довгострокових проєктах, де важлива підтримка і масштабованість, якість коду повинна бути основною метою, що потребує ретельного аналізу, високого покриття тестами та дотримання стандартів.

Загалом, забезпечення та вимірювання якості коду — це складний процес, який потребує комбінування автоматизованих інструментів і експертних оцінок, а також чіткого розуміння цілей проєкту та ресурсів. Чітко визначені

стандарти і належна оцінка якості можуть значно підвищити ефективність розробки і забезпечити успіх програмного продукту на довгострокову перспективу.

ПЕРЕЛІК ПОСИЛАНЬ

1. McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction (2nd ed.). Microsoft Press.

2. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.

3. Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.

4. Soni, S. (2016). Static Code Analysis: Techniques and Applications. CRC Press.

5. Esposito, D. (2018). Architecting Modern Software Systems: Patterns and Practices. Microsoft Press.

6. Hunt, A., & Thomas, D. (2000). The Pragmatic Programmer: Your Journey to Mastery. Addison-Wesley.

7. SonarSource. (2020). SonarQube Documentation. Retrieved from https://docs.sonarqube.org

8. ESLint. (2020). ESLint Documentation. Retrieved from https://eslint.org/docs

9. Prettier. (2020). Prettier Documentation. Retrieved from https://prettier.io/docs/en/

10. Cousins, S., & Fricke, R. (2019). The Code Review Handbook. GitHub Press.

11. Soni, S. (2017). Code Quality: The Open Source Perspective. Springer.

12. Cohn, M. (2004). Agile Estimating and Planning. Prentice Hall.

13. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model.

14. McConnell, S. (1993). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.

15. Гусєв І., Захватова Т., Єгоров А. Забезпечення якості коду програмних застосунків // Collection of abstracts of the XLVII International scientific and practical conference «The Future of Scientific Discoveries: New Trends and Technologies» (November 13-15, 2024) Marseille, France. International Scientific Unity, 2024. P. 118 – 119.