Міністерство освіти і науки України
Український державний університет науки і технологій

Факультет Комп'ютерні технології та системи
(назва факультету)

Кафедра Комп'ютерні інформаційні технології
(повна назва кафедри)

Пояснювальна записка
до кваліфікаційної роботи
магістра

на тему:
Дослідження можливостей супроводу та масштабування програмного коду
за освітньою програмою Інженерія програмного забезпечення
зі спеціальності: 121 Інженерія програмного забезпечення
Виконала: студентка групи ПЗ2321:

(підпис студента) / Дарія ЗЕЛЕНЬКО /
(Ім'я ПРІЗВИЩЕ)

Керівник: (підпис) / Олександра ГОРБОВА /
(посада, Ім'я ПРІЗВИЩЕ)

Нормоконтролер: (підпис) / Світлана ВОЛКОВА /
(посада, Ім'я ПРІЗВИЩЕ)

Засвідчую, що у цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент (підпис)

Дніпро – 2025 рік

Ministry of Education and Science of Ukraine
Ukrainian State University of Science and Technologies

Faculty of Computer technologies and systems
(faculty)

Department of Computer information technology
(department)

Explanatory Note
to Master's Thesis

on the topic: «Researching the possibilities of maintaining and scaling the source code»

according to educational curriculum Software engineering
in the Speciality: 121 Software engineering

Done by the student of the group PZ2321: / Dariia ZELENKO /
(name, surname)

Scientific Supervisor: / Olexandra GORBOVA /
(position, name, surname)

Normative controller: / Svitlana VOLKOVA /
(position, name, surname)

Dnipro – 2025

Міністерство освіти і науки України

Український державний університет науки і технологій

Факультет Комп'ютерних технологій і систем

Кафедра Комп'ютерні інформаційні технології

Рівень вищої освіти: магістр

Освітня програма: Інженерія програмного забезпечення

Спеціальність: Інженерія програмного забезпечення

«ЗАТВЕРДЖУЮ»

Завідувач кафедри КІТ

_____ Вадим ГОРЯЧКІН

«____» грудня 2024 р.

ЗАВДАННЯ

На кваліфікаційну роботу Магістра

студенту Зеленько Дарії Миколаївні

1. Тема дипломної роботи: «Дослідження можливостей супроводу та масштабування програмного коду»

   Керівник роботи: Горбова Олександра Вікторівна

   затверджені наказом № 1186 ст від 29.12.2023
2. Строк подання студентом роботи 12.01. 2025 року
3. Вихідні дані до дипломної роботи: _____
4. Зміст пояснювальної записки (перелік питань до розробки):

   4.1. Аналітична частина: Огляд проблеми, аналіз наукової та методологічної бази;

   4.2. Основна частина: Збір та аналіз метрик, пошук та розробка аналітичних інструментів, підбір матеріалу для проведення дослідження, дослідження рівня підтримки проектів з відкритим кодом, проведення аналізу та критичного оцінювання результатів дослідження;

5. Перелік демонстраційного матеріалу

5.1. Презентація;

5.2. Доповідь.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами	01.08.24 – 01.09.24	10%
2	Аналіз сучасного стану методів оцінки супроводжуванності та масштабування коду, які потребують вдосконалення для вирішення проблем дослідження	01.09.24 – 10.09.24	
3	Постановка задачі, технічне завдання	10.09.24 – 15.09.24	30%
4	Розробка інструментальних засобів дослідження	15.09.24 – 25.09.24	
5	Виконання досліджень	26.09.24 – 13.12.24	60%
6	Оформлення тез доповідей		
7	Оформлення статті у монографію		
6	Оформлення пояснювальної записки	01.11.25 – 05.01.25	
7	Розробка демонстраційних матеріалів	06.01.25 – 12.01.25	100%
8	Подання кваліфікаційної роботи до кафедри	12.01.25	
9	Захист кваліфікаційної роботи на засіданні Екзаменаційної комісії	26.01.25	

Студент _____ Дарія ЗЕЛЕНЬКО

Керівник роботи _____ Олександра ГОРБОВА

РЕФЕРАТ

Об’єктом дослідження є якість коду та засоби автоматизованого аналізу коду.

Предметом дослідження є ефективність методології «Чистого коду» на основі огляду реальних великих програмних продуктів.

Метою роботи є дослідження та пошук можливих кореляцій між кодом за стандартами "чистого коду" та складністю підтримки сучасного програмного забезпечення.

Методи дослідження та апаратура: методи системного аналізу та кількісні показники, які допомагають оцінити характеристики програмного коду та сукупність яких дозолить зробити висновки про ефективність такої моделі оцінки в умовах реальних великих проектів та складності їх супроводу та розвитку.

Результати та їх новизна: Результати та їх новизна: вперше, на базі дослідження вибірки великих проектів з відкритим вихідним кодом розроблено методи супроводу програмного забезпечення з акцентом на фактичні кількісні виміри для реалізації принципів написання якісного коду.

Значення роботи – розробка набору методів для оцінки програмного забезпечення з точки зору довгострокової підтримки коду що дозволить і далі розвивати важливу для індустрії тему та покращити методи оцінки якості коду за допомогою запропонованої методології..

Пояснювальна записка складається зі вступу, 4 розділів, висновків, бібліографічного списку та 3 додатків:

- у вступі описується сутність проблеми та її актуальність, приведені мета та практичне значення роботи. Складається із 5 сторінок;
- у першому розділі проведено аналіз сучасного стану дослідження проблеми за науковими літературними джерелами та висвітлено головну проблематику дослідження. Складається з 9 сторінок;
- у другому розділі надано обґрунтування обраних методів дослідження та сформовано вектор дослідження. Складається з 14 сторінок;

- у третьому розділі представлене проектування й розробка інструментального забезпечення для проведення дослідження. Складається з 12 сторінок;
- у четвертому розділі описано всі етапи дослідження та проведена заключна аналітика результатів. Складається з 69 сторінок;
- у додатках містяться: технічне завдання, текст програми, опис програми та аналітика дослідницьких даних.

Таблиць – 6, рисунків – 80, бібліографія – 53 джерела.

Ключові слова: супровід програмного забезпечення, масштабованість коду, дослідження якості коду, чистий код, дослідження методології чистого коду, якість коду у відкритих проектах.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ОЗНАК, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

Лінтер — програма або програмне забезпечення для аналізу, що аналізує вихідний код для виявлення потенційних помилок, проблем стилю, невірного синтаксису, потенційних проблем безпеки та інших проблем у коді. Його часто використовують у розробці мобільного додатка, веб-сайту та програмного забезпечення для забезпечення відповідності коду певним стандартам і його вірному написанню.

Технічний борг — концепція розроблення програмного забезпечення, яка відображає передбачувані витрати на додаткове доопрацювання, спричинене вибором простого (обмеженого) рішення замість використання кращого підходу, який зайняв би більше часу.

Git — розподілена система керування версіями, яка використовується для відстеження змін у файлах та координації роботи над проєктами між кількома розробниками. Вона дозволяє зберігати історію змін, працювати паралельно в гілках, об'єднувати зміни та ефективно вирішувати конфлікти. Git широко використовується у розробці програмного забезпечення.

Репозиторій Git — сховище де зберігається проєкт, що використовується для фіксації та відстеження змін у проєкті, координації роботи команди розробки.

Рефакторинг коду — процес, який передбачає редагування та очищення раніше написаного програмного коду без зміни його функцій.

Парсинг — це процес аналізу та обробки вхідних даних з метою їх перетворення в структуру, яка є більш зрозумілою та зручною для програмного використання. Парсинг широко застосовується в програмуванні для роботи з текстами, файлами, веб-сторінками, кодом програм або іншими даними.

FindBugs — інструмент для статичного аналізу коду, який спеціалізується на виявленні помилок у програмах, написаних на мові Java. Він аналізує байт-код Java для виявлення потенційних помилок, вразливостей і антипаттернів, які можуть призвести до помилок під час виконання програми.

NetBeans — інтегроване середовище розробки (IDE), яке підтримує кілька

мов програмування, зокрема Java, PHP, HTML5, JavaScript, C/C++ та багато інших. Це потужний інструмент, який надає розробникам функціональність для створення, налагодження та розгортання програмного забезпечення.

JUnit — популярний фреймворк для тестування програмного забезпечення, призначений для написання та виконання автоматизованих тестів у Java. JUnit є основою для написання юніт-тестів, які перевіряють окремі частини коду (зазвичай методи або класи) на предмет коректності їх виконання.

TestNG — потужний фреймворк для тестування, призначений для написання і виконання автоматизованих тестів, особливо для Java-додатків. TestNG (Test Next Generation) розроблено для покращення можливостей тестування і надає більш гнучкі та зручні функції, ніж традиційні фреймворки, такі як JUnit.

IntelliJ IDEA — інтегроване середовище розробки (IDE), розроблене компанією JetBrains, яке спеціалізується на розробці програмного забезпечення, переважно на мовах Java, але також підтримує багато інших мов, таких як Kotlin, Groovy, Scala, Python, PHP, JavaScript, TypeScript та інші.

Eclipse IDE — безкоштовне інтегроване середовище розробки (IDE), яке використовується для розробки програмного забезпечення на різних мовах, зокрема Java, C/C++, Python, PHP, JavaScript та ін. Eclipse є одним з найпопулярніших середовищ розробки завдяки своїй відкритій архітектурі та великій кількості доступних плагінів.

Stack Overflow — веб-сайт, що слугує платформою для обміну знаннями і вирішення проблем у сфері програмування та розробки програмного забезпечення. Він є частиною мережі сайтів Stack Exchange і є одним з найбільших і найпопулярніших ресурсів для розробників, фахівців з IT та студентів

GitHub — веб-платформа для зберігання, управління та спільного використання коду, яка використовує систему контролю версій Git. GitHub дозволяє розробникам з усього світу співпрацювати над проектами, відслідковувати зміни в коді та організовувати роботу над програмним

забезпеченням.

Посібник зі стилів коду — документ, що містить набір рекомендацій і правил щодо форматування, структурування та написання коду в певній мові програмування чи проекті. Мета такого посібника — забезпечити єдність і послідовність в коді, що спрощує його читання, розуміння та обслуговування.

Комміт (commit) — зафіксовані зміни в репозиторії системи керування версіями (наприклад, Git). Це одиниця історії змін, яка містить інформацію про внесені правки, автора, час і опис (commit message). Комміт дозволяє відстежувати розвиток проєкту і повертатися до попередніх станів коду за потреби.

SonarLint – плагін для інтегрованих середовищ розробки, таких як IntelliJ IDEA, Eclipse чи Visual Studio, який виконує статичний аналіз коду безпосередньо під час написання. Він допомагає виявляти помилки, потенційні вразливості та порушення стандартів написання коду в реальному часі, забезпечуючи розробнику швидкий зворотний зв’язок для виправлення проблем.

GitStats – це інструмент для збору статистичних даних про історію змін у репозиторії Git. Він генерує звіти у вигляді HTML-сторінок з інформацією про кількість комітів, активність розробників, розмір репозиторію, зміну кількості рядків коду з часом та інші метрики, що дозволяють аналізувати розвиток проєкту.

Qodana – плагін, розроблений JetBrains, який дозволяє виконувати локальний аналіз коду безпосередньо в IntelliJ IDEA та інших IDE від JetBrains. Qodana надає можливість налаштовувати правила перевірки, допомагає знаходити технічний борг, складність коду, потенційні дефекти та інші проблеми якості, забезпечуючи підтримку багатьох мов програмування.

Docker – це відкрита платформа для контейнеризації програмного забезпечення. Вона дозволяє упаковувати програми та їх залежності у стандартизовані контейнери, які забезпечують переносимість та ізоляцію додатків у різних середовищах виконання. Контейнери Docker включають всі необхідні компоненти для виконання програми, такі як код, залежності та

конфігураційні файли, ізолюючи їх від операційної системи хоста.

Пулл-реквест (pull request) — запит на внесення змін до основного репозиторію, який надсилається контриб'ютором. Використовується в системах керування версіями, таких як Git, для пропозиції змін (наприклад, додавання нового функціоналу чи виправлення помилок) до основної кодової бази проєкту.

Контриб'ютор (contributor) — учасник проєкту, який робить внески, наприклад, додає код, виправляє помилки, оновлює документацію чи тестує функціонал. Це може бути будь-хто, хто сприяє розвитку проєкту, навіть якщо не є його основним розробником.

Форк репозиторія — це копія існуючого репозиторія, яка створюється на обліковому записі користувача для незалежної роботи з вихідним кодом. Форк дозволяє розробникам модифікувати проект, додавати нові функції або виправляти помилки, не впливаючи на оригінальний репозиторій.

ВСТУП

Якісний код - є зрозумілим, простим, добре протестованим, без помилок, таким, що пройшов детальний рефакторинг, задокументований і продуктивний. Але основне мірило якісного коду - відповідність специфікації, залежить від потреб проекту.

Схоже програмне забезпечення може мати очевидні відмінності в коді від розробника до розробника, але найважливіше те, що якість коду повинна залишатися незмінною протягом усього часу. Якість коду не є новим параметром для коду, і насправді про нього говорять ще з 1970 року. Зараз необхідність приділяти більше уваги якості коду стала настільки актуальною, що близько 90% компаній використовують інструменти аналізу коду для покращення його якості. Вимірювання якості коду, з іншого боку, може бути складним, оскільки параметри змінюються відповідно до потреб проектів, тому не існує єдиного способу його вимірювання. Ключовими метриками, що визначають якість програмного коду, є надійність, ремонтопридатність, тестованість, портативність та можливість повторного використання. Ці характеристики відіграють вирішальну роль у довготривалій підтримці та масштабуванні програмних систем, а також забезпечують їх ефективну експлуатацію в різних умовах.

Надійність програмного забезпечення полягає в його здатності забезпечувати стабільну роботу навіть в умовах непередбачених ситуацій або помилок під час виконання. Високоякісне програмне забезпечення повинно бути здатним обробляти виняткові ситуації, зберігаючи при цьому загальну функціональність. Ця здатність важлива як для кінцевих користувачів, так і для розробників, оскільки вона дозволяє зберігати довіру до програмного продукту. У разі виникнення помилок користувач повинен отримати зрозуміле і детальне повідомлення про помилку, що дозволяє йому усвідомити, що саме пішло не так і як можна виправити ситуацію. Відсутність таких зрозумілих повідомлень призводить до зниження довіри користувачів до продукту.

Одним з найважливіших аспектів якісного програмного забезпечення є

читабельність коду. Чітко структурований, добре коментований і зрозумілий код не тільки знижує ймовірність виникнення помилок, але й суттєво спрощує процес його супроводу. Коли код легко читати, його легше підтримувати, тестувати та змінювати. Ремонтопридатність коду дозволяє швидко адаптувати програмне забезпечення до нових вимог або виправляти недоліки без необхідності переписування великих частин програмного забезпечення. Це особливо важливо для масштабних проектів, де змінюються не тільки функціональні вимоги, але й технологічні платформи, на яких працює продукт.

Деталізовані коментарі, структуровані блоки коду та використання загальноприйнятих стилів програмування дозволяють іншим членам команди швидко орієнтуватися в коді та вносити корективи без ризику порушити існуючу логіку. Це також полегшує передачу коду між різними командами, сприяючи колективному навчанню та знижуючи залежність від конкретних розробників.

Тестованість — це здатність програмного забезпечення бути перевіреним на наявність помилок у процесі його розробки та експлуатації. Якісний код, спроектований з урахуванням принципів модульності та з дотриманням високих стандартів написання тестів, дозволяє проводити ефективне тестування кожної окремої складової програми. Це знижує кількість критичних помилок у фінальному продукті та полегшує підтримку системи у випадку внесення змін чи доповнень. Тестованість напряму впливає на стабільність і безперервність роботи програмного забезпечення в умовах зміни зовнішніх факторів або нових викликів ринку.

З огляду на різноманітність платформ та середовищ, на яких працює сучасне програмне забезпечення, портативність стає ключовим чинником якості коду. Програмний продукт повинен бути легко адаптованим до нових операційних систем або апаратних рішень з мінімальними змінами у вихідному коді. Якісно написаний код спрощує цей процес, дозволяючи швидко масштабувати програмне забезпечення на нові платформи або інтегрувати його в різні системи. Відсутність необхідності кардинальних змін у коді під час перенесення забезпечує ефективність міграційних процесів та знижує витрати на

адаптацію.

Можливість повторного використання компонентів коду сприяє економії часу та ресурсів при розробці нових функцій або продуктів. Добре спроектовані модулі або бібліотеки можна використовувати в різних частинах одного проекту чи в зовсім нових проектах, що знижує потребу в написанні коду з нуля та підвищує загальну продуктивність команди розробників.

Актуальність роботи:

Якість програмного коду прямо впливає на здатність до масштабування та довготривалого супроводу програмного забезпечення. Масштабованість означає здатність системи справлятися зі збільшенням навантаження або інтеграцією нових функцій без істотних змін у її архітектурі. Для цього програмне забезпечення має бути добре структурованим і модульним, що дозволяє додавати нові компоненти або змінювати існуючі без негативного впливу на інші частини системи.

Що стосується супроводу, то підтримка великомасштабних програмних систем може займати значні ресурси, якщо код складний для розуміння або його важко змінювати. Якісний код знижує час, необхідний для виправлення помилок або адаптації системи до нових умов, що, своєю чергою, знижує технічний борг. Технічний борг — це накопичені проблеми або застарілі рішення, які ускладнюють подальший розвиток програмного забезпечення. Низькоякісний код може швидко збільшити технічний борг, змушуючи розробників витрачати більше часу та ресурсів на усунення недоліків, тоді як код високої якості допомагає уникнути цих проблем, забезпечуючи стабільність і гнучкість системи у довгостроковій перспективі.

Таким чином, якість програмного коду є фундаментом для ефективного супроводу та масштабування програмного забезпечення. Добре спроектований, тестований і документований код забезпечує гнучкість при внесенні змін, підвищує стабільність системи та сприяє довготривалому функціонуванню продукту. Суттєве значення для подальшого розвитку програмної інженерії. Ця робота та проведене дослідження сприяють розвитку теоретичних основ

програмування завдяки тому, що формують наукову базу у напрямку, який має загалом невелику кількість досліджень та не має стабільної наукової бази.

Доцільність роботи, її відмінність з відомими розв'язаннями проблеми. Ця робота є доцільною, тому що сфера вивчення якості коду особливо у рамках "Чистого коду" майже не має наукових емпіричних досліджень, з якими можна було б працювати. Це створює проблему непідтвердженої гіпотези.

Об'єктом дослідження є якість коду та засоби автоматизованого аналізу коду.

Предметом дослідження є ефективність методології «Чистого коду» на основі огляду реальних великих програмних продуктів.

Мета роботи полягає у дослідженні та пошуку можливих кореляцій між кодом за стандартами "чистого коду" та складністю підтримки сучасного програмного забезпечення.

Згідно з поставленою метою слід вирішити такі завдання:

- Розробити та описати метрики оцінки якості коду, обґрунтувавши вибір метрик;
- провести збір інформації за допомогою автоматизованих методів збору даних;
- дослідити взаємозв'язок між принципами якісного коду та покращенням якості підтримки програмного забезпечення за раніше описаними метриками;
- запропонувати методологію оцінки якості коду.

Методи дослідження:

Для дослідження були використані кількісні показники, які допомагають оцінити характеристики програмного коду та сукупність яких дозолить зробити висновки про ефективність такої моделі оцінки в умовах реальних великих проектів та складності їх супроводу та розвитку.

Наукова новизна:

Вперше, на базі дослідження вибірки великих проектів з відкритим вихідним кодом розроблено методи супроводу програмного забезпечення з

акцентом на фактичні кількісні виміри для реалізації принципів написання якісного коду.

Практичне значення полягає в розробці і уніфікації принципів оцінки програмного забезпечення з метою спрощення процедури супроводження та підтримки кодової бази. Результати виконаних досліджень та запропонованої методології можуть бути застосовані на етапах: розробки технічного завдання, формування узгоджень та рефакторингу коду.

Апробація результатів дослідження.

Основні положення магістерської роботи доповідалися та були схвалені на Всеукраїнській науково-технічній конференції студентів і молодих учених “Наука і сталий розвиток транспорту 2024” (Дніпро, УДУНТ, 27 листопада 2024 року), XVIII міжнародній науково-практичній конференції «Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості та освіті» (Дніпро, УДУНТ, 12-13 грудня 2024 року), на семінарах кафедри КІТ впродовж вересня-грудня 2024 року та на міжнародному науковому симпозіуму “Wissenschaftliche Forschung unter modernen Bedingungen der Instabilität /Scientific research in modern conditions of instability '2024”(Карлсруе, Німеччина, листопад 2024 року).

Публікації за темою роботи.

За результатами виконаних досліджень опубліковано 3 наукові праці, у тому числі: одна наукова стаття у європейському виданні в міжнародній монографії ”Scientific research in modern conditions of instability '2024”(Карлсруе, Німеччина, листопад 2024 року)[30] та двоє тез доповідей на міжнародних наукових конференціях[28, 29].

1 АНАЛІЗ МЕТОДОЛОГІЧНИХ ЗАСАД СУПРОВОДУ ПРОГРАМНОГО ЗАБАЗПЕЧЕННЯ

1.1 Проблематика підтримки та масштабування коду

Супровід коду та масштабованість вже давно є основними проблемами в інженерії програмного забезпечення, що розвиваються разом зі зростанням складності та обсягу програмних систем. Історично склалося так, що ранні програмні системи часто розроблялися ізольовано, з обмеженою увагою до довговічності або розширюваності кодової бази. Однак у другій половині 20-го століття, коли обчислювальні потреби розширилися, стали зрозумілими обмеження таких підходів. Системи часто були тісно пов'язані між собою, погано задокументовані і схильні до збоїв, коли з'являлися нові функції або коли потрібно було масштабування для більшої кількості користувачів або більших наборів даних. Це заклало основу для більш формалізованого фокусу на якості коду, обслуговуванні та масштабованості як в академічних дослідженнях, так і в промисловій практиці.

У цьому контексті книга Роберта Мартіна «Чистий код»[1], яка була написана у 2008, є важливою віхою в еволюції філософії розробки програмного забезпечення. Розглянута через наукову та історичну призму, «Чистий код» містить багато уроків, винесених з десятиліть невдач та успіхів програмної інженерії. Книга не є революційним текстом у сенсі введення абсолютно нових концепцій, а радше консолідацією найкращих практик, спрямованих на подолання поширених пасток у супроводі та масштабуванні коду. Він відображає зростаюче визнання, яке почалося в 1960-х і 1970-х роках, що програмні системи не є статичними об'єктами. Вони повинні розвиватися, адаптуватися і масштабуватися з часом, а це вимагає дисциплінованого підходу до кодування, який полегшує довгострокове обслуговування.

З наукової точки зору, потреба в підтримуваному і масштабованому коді була предметом все більшого вивчення з перших днів комп'ютерної науки. Дослідження в галузі програмної інженерії, особливо в таких сферах, як розуміння програм і програмні метрики, підкреслюють, як структура і

читабельність коду суттєво впливають на його супроводжуваність. У ранніх роботах, таких як «Міфічний людино-місяць» Фредеріка П. Брукса[27], де аналізувались проблеми великомасштабної розробки програмного забезпечення, стверджувалося, що складність зростає нелінійно зі збільшенням розміру системи. Це безпосередньо пов'язано з підтримкою коду, оскільки більші системи зі складною кодовою базою стають експоненціальною складнішими для оновлення чи виправлення.

Наукова література про когнітивні процеси, пов'язані з розумінням коду, наприклад, робота Пеннінгтона[18], дає глибше розуміння того, чому чистий код має значення. Пеннінгтон дослідив, як програмісти засвоюють код і міркують про нього, зазначивши, що погано структурований код вимагає більше когнітивних зусиль для розуміння. Це безпосередньо пов'язано з підтримкою коду, оскільки чим складніше зрозуміти фрагмент коду, тим складніше його модифікувати і тим більша ймовірність помилок. Пропагуючи стиль кодування, який надає пріоритет читабельності та простоті, Clean Code відповідає цим висновкам, припускаючи, що такі практики зменшують когнітивне навантаження на розробників, тим самим покращуючи як індивідуальну продуктивність, так і командну масштабованість.

Крім того, питання масштабованості полягає не лише в додаванні нових функцій чи роботі з більшою кількістю користувачів, але й у тому, як системи можуть зростати в складності, не стаючи некерованими. Історичні збої програмного забезпечення, такі як сумнозвісна аварія ракети Ariane 5 у 1996 році, проблема з перевантаженням в Patriot Missile у 1991, помилка Y2K (2000) та навіть помилка у програмному забезпечені від CrowdStrike у липні 2024 року підкреслили, як некерований і погано зрозумілий код може призвести до катастрофічних збоїв, коли системи виходять за рамки своїх початкових можливостей. Ці приклади ілюструють, як складність коду, що зростає з часом, разом з недостатньою якістю тестування і відсутністю належної архітектури, можуть призвести до критичних помилок, що іноді коштують життя або величезних економічних збитків. Це підкреслює потребу в масштабованих

системах, які можна легко модифікувати і тестувати в міру їх розвитку, що є центральним принципом філософії Мартіна в «Чистому коді».

Якщо поглянути на літературу з архітектури програмного забезпечення, то масштабованість вже давно вивчається з точки зору як технічних, так і організаційних аспектів. Ранні комп'ютерні системи були розроблені для конкретних, часто статичних завдань, і масштабованість рідко викликала занепокоєння. Однак з появою корпоративних систем у 1980-х роках та розвитку інтернету у 1990-х роках потреба в масштабованих архітектурах стала очевидною. У науковому контексті масштабованість часто стосується того, наскільки добре система може впоратися зі зростанням, не лише з точки зору продуктивності (тобто, обробки більшої кількості користувачів або більших наборів даних), але й з точки зору здатності додавати нові функції без погіршення надійності системи або можливості її обслуговування. Робота Мартіна знаходиться в рамках цієї розширеної дискусії, виступаючи за код, який можна адаптувати до змін - одна з ключових вимог до масштабованої системи.

Більше того, дослідження метрик програмного забезпечення - кількісних показників складності програмного забезпечення - показали, що певні практики кодування можуть передбачити складність обслуговування. Наприклад, висока цикломатична складність асоціюється зі складнішим в обслуговуванні кодом, оскільки він представляє більш заплутані та менш передбачувані потоки управління. Аналогічно, дослідження рівня дефектів у великих кодових базах показали, що добре структурований, читабельний код, як правило, призводить до меншої кількості помилок і швидшого вирішення проблем, коли вони все ж виникають. У зв'язку з цим акцент Мартіна на чистому, простому коді можна розглядати як науково обґрунтований підхід до зменшення дефектів програмного забезпечення та покращення довгострокової масштабованості, навіть якщо його робота в першу чергу орієнтована на промисловість, а не на академічну науку.

Історично склалося так, що прагнення до легкого в обслуговуванні та масштабованого коду збігається з розвитком гнучких методологій наприкінці

1990-х - на початку 2000-х років. Зосередженість Agile на ітеративній розробці, співпраці з клієнтами та швидкій адаптації до змін створила потребу в кодових базах, які можна було б часто модифікувати, не порушуючи цілісності коду. Clean Code чітко вписується в цю парадигму, пропонуючи настанови для написання коду, який може бути легко зрозумілим і модифікованим різними розробниками, що працюють в коротких циклах. Він відображає ширший рух у розробці програмного забезпечення до гнучкості та адаптивності, що стало відповіддю на невдачі жорстких, водоспадних процесів розробки, які домінували у попередні десятиліття.

На завершення, «Чистий код»[1] можна розуміти не лише як посібник з написання кращого програмного забезпечення, але й як частину історичної та наукової еволюції в тому, як ми думаємо про код. Його акцент на читабельності, простоті та адаптивності говорить про давні проблеми програмної інженерії, від когнітивних викликів розуміння складних систем до технічних викликів масштабування програмного забезпечення для зростаючої кількості користувачів та наборів функцій. З огляду на це, «Чистий код» є одночасно кульмінацією десятиліть важких уроків у розробці програмного забезпечення та практичним посібником для вирішення поточних проблем, пов'язаних з підтримкою та масштабуванням сучасних систем.

1.2 Аналіз сучасного стану проблеми за науковими літературними джерелами

Сучасний стан проблеми супроводу та масштабованості коду залишається актуальною проблемою в інженерії програмного забезпечення, незважаючи на прогрес у практиках та інструментах розробки. Наукова література продовжує досліджувати нюанси цього питання, відображаючи зростаючу складність програмних систем та підвищення вимог до гнучкості, продуктивності та надійності в сучасному життєвому циклі розробки програмного забезпечення. Дослідження на ці теми стосуються як технічних, так і когнітивних факторів, що впливають на супровід коду та масштабованість, і існує визнання того, що, незважаючи на десятиліття досліджень, ці проблеми

зберігаються у нових формах, оскільки системи зростають у розмірі та складності.

1.2.1 Супровід коду

Згідно з останніми науковими дослідженнями, підтримка коду все ще становить значну частину загальної вартості розробки програмного забезпечення. Дослідження 2018 року, проведене [15], показало, що на підтримку програмного забезпечення витрачається до 50-75% від загальної вартості життєвого циклу програмної системи. Основна частина роботи з обслуговування полягає в розумінні існуючого коду, виправленні помилок і невеликих адоптаціях до нових вимог. Проблема ускладняється тим, що сучасні програмні системи часто підтримуються розробниками, які не писали оригінального коду, що призводить до збільшення часу на розуміння та модифікацію коду. Дослідження когнітивного навантаження при супроводі програмного забезпечення підкреслюють, що погано написаний або недостатньо задокументований код збільшує час, необхідний для внесення змін, а отже, збільшує витрати і ймовірність появи нових помилок.

Дослідження, такі як [16, 32, 33], підкреслюють, що хоча читабельність коду є сильним предиктором його супроводжуваності, інструменти, які підтримують кращу документацію, рефакторинг та автоматизоване тестування, також можуть пом'якшити труднощі з обслуговуванням. Однак ці дослідження виявляють значний розрив між найкращими практиками та реальним впровадженням. Наприклад, хоча безперервний рефакторинг широко рекомендується в літературі, його впровадження в індустрії є непослідовним. На практиці команди часто стикаються зі стислими термінами або не мають інституційних стимулів для пріоритизації якості коду, що призводить до накопичення технічного боргу з часом. Цей борг проявляється в тому, що зі зростанням складності систем їх стає дедалі важче підтримувати. Нещодавні дослідження кількісної оцінки технічного боргу, як наприклад робота [17, 34 - 36], показує, що нездатність керувати цим аспектом розробки може призвести до експоненціального зростання витрат на обслуговування в міру старіння системи.

Більше того, емпіричні дослідження застарілих систем показують, що багато організацій продовжують експлуатувати велике, критично важливе програмне забезпечення, розроблене десятиліття тому, часто на застарілих мовах або архітектурах. Тягар підтримки цих систем посилюється дефіцитом розробників, знайомих із застарілими технологіями, та зростанням вартості міграції на сучасні архітектури. Такий стан справ створює значні проблеми для організацій, які прагнуть йти в ногу з сучасними вимогами до програмного забезпечення, зберігаючи при цьому стабільність існуючої інфраструктури.

1.2.2 Масштабованість

Проблема масштабованості є особливо актуальною, оскільки сучасні системи повинні пристосовуватися до швидко зростаючих баз користувачів, обсягів даних та наборів функцій. Сучасні дослідження в галузі архітектури програмного забезпечення часто зосереджені на стратегіях проектування систем, які можуть масштабуватися як вертикально (шляхом підвищення продуктивності системи), так і горизонтально (шляхом додавання додаткового обладнання або розподілу робочих навантажень між серверами). Масштабованість часто вивчають з точки зору архітектури програмного забезпечення, при цьому дослідники вивчають різні архітектурні патерни - такі як мікросервіси, архітектури, керовані подіями, та безсерверні обчислення - як рішення проблем масштабованості. У статті [20], підкреслюється, що архітектура мікросервісів стає все більш популярною як спосіб роз'єднання компонентів системи, що полегшує масштабування окремих сервісів, не впливаючи на всю систему в цілому.

Однак, хоча мікросервіси та подібні архітектурні патерни пропонують багатообіцяючі рішення проблем масштабованості, дослідження також вказують на їхні обмеження. Ключовою проблемою є компроміс між масштабованістю та складністю. Хоча ці архітектури підтримують масштабованість, дозволяючи незалежне розгортання та масштабування сервісів, вони призводять до значної операційної складності. Згідно з дослідженням [19], проведеним у 2019 році, управління системою на основі мікросервісів вимагає складного інструментарію

для розгортання, моніторингу та обробки помилок, що може призвести до виникнення нових проблем в обслуговуванні. Аналогічно, такі роботи, як [26,37. 38], стверджують, що організації часто недооцінюють складність розподілених систем і можливість виникнення таких проблем, як узгодженість даних, мережеві затримки та обробка збоїв, що стають вузькими місцями в масштабуванні.

Управління складністю коду у великомасштабних програмних системах є серйозною проблемою, яка може перешкоджати масштабованості та супроводжуваності[21]. У міру розширення кодових баз вони часто стають складними, що ускладнює реалізацію нових функцій або виправлення помилок без внесення помилок. Така складність може призвести до погіршення читабельності коду та збільшення часу розробки. Щоб вирішити ці проблеми, важливо застосовувати такі стратегії, як модульний дизайн, чітка документація та регулярний рефакторинг, щоб підтримувати якість коду та забезпечувати ефективний розвиток системи. Управління складністю коду у великомасштабних програмних системах є серйозною проблемою, яка впливає на масштабованість та зручність обслуговування.

По мірі зростання кодові бази стають все більш складними, що ускладнює впровадження нових функцій, виправлення помилок або підтримку стабільності. Така складність знижує читабельність, сповільнює розробку і збільшує ризик внесення помилок при внесенні змін. Масштабування великих систем створює додаткові труднощі, такі як міжкомандні залежності та фрагментарність знань[39]. Ці проблеми сприяють зростанню складності коду, що ускладнює його розробку та підтримку. Щоб вирішити ці проблеми, необхідно впроваджувати модульний дизайн, покращувати комунікацію та ефективно керувати залежностями. Ці стратегії допомагають підтримувати якість коду і гарантують, що система може адаптуватися до зростання, не стаючи некерованою.

Оскільки додатки стають все більш складними, оптимізувати продуктивність і масштабованість стає все складніше. Виявлення проблем з продуктивністю за допомогою інструментів профілювання та моніторингу є ключем до оптимізації використання ресурсів. Цей процес включає в себе точне

налаштування інструментів моніторингу та оптимізацію інфраструктури для досягнення цілей продуктивності. Для забезпечення масштабованості та продуктивності великих систем є необхідність постійного моніторингу їхнього стану, цілеспрямованих удосконалень та глибокого розуміння контексту роботи системи [40]. Зосередившись на модульному дизайні, чіткій комунікації та постійній оптимізації, організації можуть ефективно управляти зростаючою складністю, гарантуючи, що системи залишатимуться масштабованими та підтримуваними протягом тривалого часу.

1.2.3 Виклики у вирішенні проблеми

Однією з важливих проблем, на яку вказують сучасні дослідження, є дихотомія між короткостроковим тиском на поставку та довгостроковими проблемами підтримки. Багато команд розробників стикаються з необхідністю швидко надавати функції, часто за рахунок якості коду. Ця напруга добре задокументована в літературі про технічний борг, де короткострокові компроміси в якості програмного забезпечення призводять до довгострокових проблем з підтримкою та масштабуванням. У дослідженні [22] йдеться про те, що накопичення технічного боргу часто є невидимою проблемою - його вплив відчувається лише тоді, коли системи стає надто складно розширювати чи модифікувати. Хоча існують інструменти для вимірювання певних аспектів технічного боргу, такі як «запах коду» або метрики складності коду, ці інструменти не є загальноприйнятими, а розробникам часто бракує часу або стимулів для проактивного вирішення проблеми боргу.

Крім того, сучасні програмні системи дедалі більше взаємопов'язані із зовнішніми сервісами та платформами, що ускладнює масштабування та обслуговування. Наприклад, хмарні системи часто покладаються на сторонні API, бази даних і сервіси, які з часом можуть змінюватися або вносити докорінні зміни. Це створює нові форми залежності, якими потрібно ретельно керувати, щоб уникнути перебоїв у роботі системи або погіршення продуктивності. Нещодавнє дослідження [23] обговорює проблему управління зовнішніми залежностями у великомасштабних програмних системах, наголошуючи на

необхідності надійних стратегій тестування, моніторингу та версійності для зменшення ризиків, пов'язаних з масштабуванням.

1.2.4 Якість коду

Окрім зовнішніх залежностей та технічного боргу, сучасні програмні системи стикаються з внутрішніми проблемами, пов'язаними з якістю коду. Діомідіс Спінелліс [24] пропонує всебічний аналіз того, що робить код зручним для супроводу та масштабування, досліджуючи найкращі практики у проектах з відкритим кодом. Спінелліс підкреслює, що якісний код характеризується модульністю, зрозумілими інтерфейсами та можливістю тестувати і розширювати функції без регресу. Його робота спирається на великі, успішні системи з відкритим кодом, пропонуючи докази того, що читабельний, добре задокументований і модульний код призводить до створення більш масштабованих і зручних для обслуговування систем.

Спінелліс виступає за такі методи, як захисне програмування, дотримання стандартів кодування та автоматизоване тестування як життєво-важливі стратегії для забезпечення довгострокової якості коду. Його аналіз[41, 42] узгоджується з більш широким колом досліджень з супроводу програмного забезпечення, зокрема, демонструючи, що ясність і структура коду важливі не лише для безпосередньої функціональності, але й для масштабування та адаптації системи з часом. Погляд Спінелліса, заснований на спільноті відкритого коду, є особливо актуальним у сучасному програмному ландшафті, де багато систем інтегрують компоненти з відкритим кодом і, як очікується, будуть швидко розвиватися, зберігаючи при цьому високу надійність.

Висновки до розділу 1

Отже, поточний стан супроводу та масштабування коду залишається багатогранною проблемою, тісно переплетеною з технічними, організаційними та людськими аспектами розробки програмного забезпечення. Незважаючи на розвиток інструментів і передових практик, наукова література підкреслює, що ці проблеми залишаються актуальними, особливо в міру того, як системи стають

все складнішими і масштабнішими. Хоча сучасні архітектурні моделі, такі як мікросервіси та хмарні технології, надають можливості для більш ефективного масштабування систем, вони також створюють новий тягар в обслуговуванні. Дослідники продовжують вивчати стратегії збалансування вимог швидкої розробки функцій з довгостроковою стійкістю, але проблема залишається далекою від вирішення. Вирішення цих проблем вимагає постійної уваги як до технічних аспектів проектування систем, так і до когнітивних та організаційних факторів, які впливають на те, як розробники підтримують і масштабують код з часом.

2 МЕТОДИ ДОСЛІДЖЕННЯ

Основними методами дослідження є стандарти коду різних типів. Особливо у рамках парадигмами «Чистого коду", основними критеріями є зрозумілість, стабільність та загальна структура коду. Роздивимось особливості моделі стандартизації коду, на основі чого та як вони створились.

Вміння писати код, придатний для супроводу, є однією з ключових навичок будь-якого програміста. Це означає, що код повинен бути легким для читання, розуміння, модифікації та повторного використання як автором, так і іншими фахівцями. Такий підхід допомагає зменшити кількість помилок та уникнути технічних боргів.

2.1 Необхідність домовленостей щодо коду

Кодекс правил та стандартів розробки коду має важливе значення для підтримки якості програмного забезпечення. Близько 80% загальних витрат на життєвий цикл програмного забезпечення пов’язано з його підтримкою, при цьому мало ймовірно, що програмний продукт буде підтримуватися автором протягом усього терміну його використання. Угоди про стиль написання коду, підвищують читабельність коду, що дозволяє інженерам швидше орієнтуватися у нових проектах та ефективніше взаємодіяти з кодовою базою.

Стандарти оформлення коду - це набір правил і домовленостей, які визначають, як писати і форматувати код. Вони можуть охоплювати такі аспекти, як відступи, іменування, коментарі, структура та синтаксис. Стандарти написання коду допомагають зробити код послідовним, чітким та організованим. Вони також полегшують співпрацю з іншими програмістами, які можуть дотримуватися тих самих стандартів і уникати плутанини. Використання загальноприйнятих або власних стандартів сприяє підвищенню якості коду та спрощенню його підтримки.

2.2 Складність оцінки

Супроводжуваність коду є фундаментальною частиною якісної розробки програмного забезпечення. Це важливо, тому що коли код стає менш придатним для супроводу, вартість підтримки коду зростає. Код, який важко підтримувати,

значно підвищує вартість його подальшого обслуговування, оскільки виникає більше помилок, складніше додавати нові функції, а новим розробникам важче адаптуватися до проекту.

Графічне представлення залежності між витратами та супроводжуваністю коду :

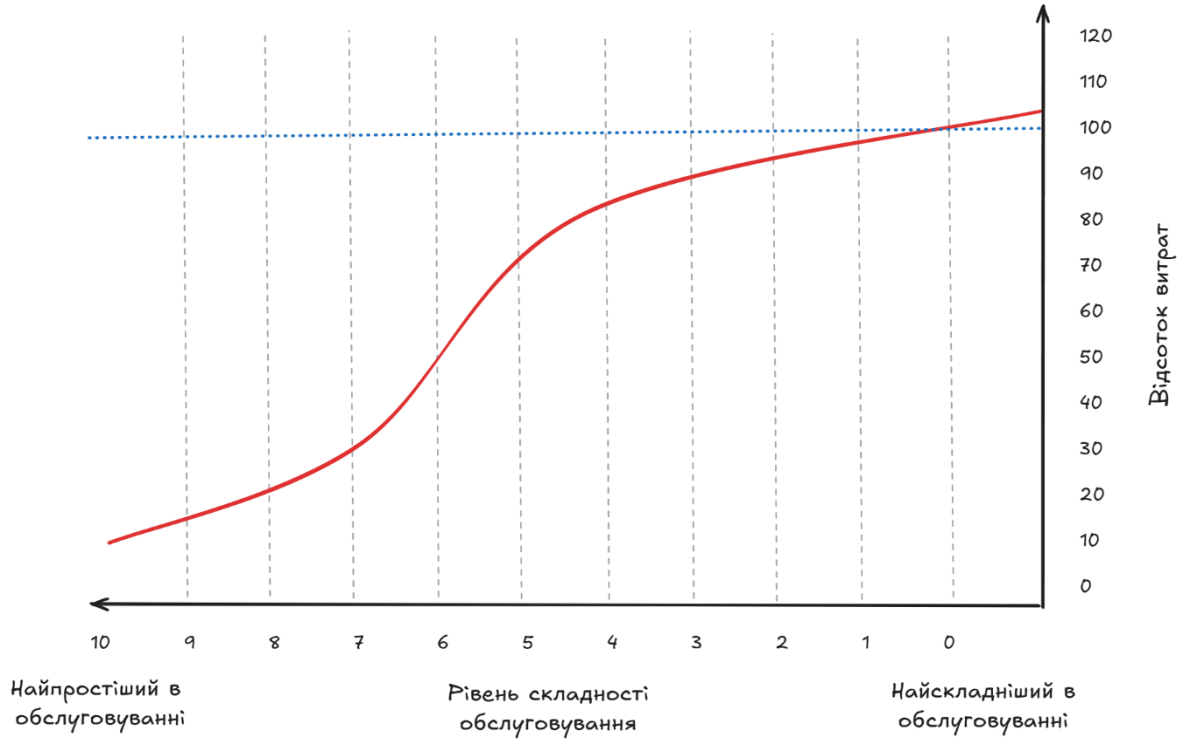


Рисунок 2.1 – Відношення між витратами та рівнем легкості підтримки коду

Зі збільшенням зручності обслуговування витрати знижуються. Витрати на підтримку ніколи не падають до нуля, але вони значно зменшуються з покращенням кодової бази.

Одна з проблем супроводжуваності коду полягає в тому, що її важко оцінити кількісно. На приклад, тест Джоела Спольського[43] є відомим методом оцінки якості роботи команди розробників програмного забезпечення, але існує кілька інших методологій та метрик, призначених для оцінки якості коду та його ремонтопридатності. До них належать цикломатична складність, яка вимірює складність програми шляхом кількісної оцінки кількості лінійно незалежних шляхів у вихідному коді, покриття коду, яке вимірює відсоток коду, що виконується автоматизованими тестами, інструменти статичного аналізу коду, такі як SonarQube, PMD та Checkstyle, які аналізують код без його виконання,

щоб знайти потенційні помилки, вразливості безпеки та "запахи" коду, а також огляди коду, які являють собою ручні перевірки коду колегами з метою виявлення помилок, покращення дизайну та дотримання стандартів написання коду. Крім того, індекс супроводжуваності об'єднує кілька метрик, щоб дати загальну оцінку, яка показує, наскільки легко підтримувати код. Технічний борг відображає вартість додаткового доопрацювання, спричинену вибором легкого рішення зараз замість використання кращого підходу, який зайняв би більше часу. Лінтери коду аналізують вихідний код, щоб позначити помилки програмування, стилістичні помилки та підозрілі конструкції. А показники відстеження та усунення помилок вимірюють, наскільки ефективно виявляються та усуваються помилки в кодовій базі.

Незважаючи на доступність і корисність цих методологій, їх часто вважають неефективними або недостатніми з кількох причин. Багато метрик фокусуються на одному аспекті якості коду, не даючи цілісного уявлення про кодову базу. В основному, їх можливо пристосувати тільки для невеликих проектів, такі методі дуже часто не дають точної відповіді стосовно якості коду, вони занадто загальні, а критерії дуже часто не надійні.

Наприклад, високе покриття коду не гарантує, що тести ефективні або що код добре написаний. Інструменти статичного аналізу коду можуть генерувати велику кількість хибних спрацьовувань, перевантажуючи розробників і ускладнюючи виявлення справжніх проблем. Рецензування коду, хоча і є цінним, але займає багато часу і значною мірою залежить від навичок і старанності рецензентів, що призводить до непослідовності в його ефективності. Індекс супроводжуваності, хоча і є всеосяжним, може бути складним для інтерпретації і не завжди відображає реальні виклики, пов'язані з підтримкою кодової бази. Технічний борг часто важко точно оцінити кількісно і він може бути суб'єктивним, що ускладнює його визначення пріоритетів та вирішення. Кодові ворсинки, хоча і корисні для забезпечення дотримання стандартів кодування, іноді можуть запроваджувати занадто жорсткі правила, що стримує творчість та інновації. Показники відстеження та усунення помилок можуть вводити в оману,

якщо основні процеси неефективні або якщо помилки не мають належного пріоритету.

Зрештою, ці методології та метрики часто вважаються недостатніми, оскільки вони не можуть охопити всю складність розробки програмного забезпечення. Вони можуть надати корисну інформацію, але не можуть замінити тонке розуміння та судження досвідчених розробників. Як наслідок, вони часто не повністю інтегровані в процес розробки програмного забезпечення, використовуючись скоріше як додаткові інструменти, а не як остаточні показники якості коду.

І саме це створює проблему не повного покриття коду, тому що не має розробленої систематики оцінки, дуже багато автоматизованих засобів, але їх майже ніхто не порівнює за якістю та надійністю і повторно не комбінує. І ця робота сконцентрована саме на вирішенні цього питання, а також на підборі оптимальної комбінації інструментів, з метою розробки оптимальної методології оцінювання.

2.3 Важливість правил та стандартів у рамках розробки програмного забезпечення

Розробка програмного забезпечення у сучасному світі вже дійшла до того рівня, коли для багатьох компонентів, архітектурних рішень та структур - вже є розроблені системи правил та метрик, які допомагають у процесі розробки тим, що завдяки ним розробник може бути впевнений, що дотримується правил та створює систему за "канонами". Це гарантує, що нарівні будови системи загалом, вона буде зрозуміла не тільки самому розробнику, але й іншим людям, таким як інші розробники, менеджери або користувачі.

Ідея полягає в тому, що розробник не зобов’язаний суворо дотримуватися всіх правил, він має свободу прийняття рішення стосовно архітектури та структури продукту. Але при дотримані правил та постулатів - загальна якість та зрозумілість кінцевого продукту буде вище, ніж без їх дотримання. І це правило розповсюджується не тільки на архітектуру чи стандарти розробки систем з конкретним призначенням чи використанням але й для самого коду.

Описані вище думки викликають багато обговорень у спільноті розробників[48-52]. Є дуже багато ідей, думок та пропозицій як можна покращити код, зробити його більш зрозумілим та ефективним. Але все це завжди знаходилось у полі суб'єктивних дискусій, не маючих під собою серйозної доказової або фактичної бази.

У процесі розвитку галузі інформаційних технологій, з'явилася потреба у розробці мінімального набору правил та стилю кодування. Фахівці висловлювали думку та намагалися розвивати описаний напрям через проблему пов'язану з нерозбірливим та незрозумілим кодом, з метою його покращення чи підтримання. Пошук рішення погіршувався тим, що можливостей та інструментів, щоб це зробити, було недостатньо. До того ж програмісти мали багато рудиментарних звичок, які вони отримали у процесі роботи з низькорівневими мовами та зберегли їх навіть переходячи до нових більш "синтаксично дружніх" мов.

Отримуючи цей досвід емпіричним шляхом та через метод проб та помилок, програмісти того часу почали розуміти краще, що призводить до ускладнень в роботі, які є паттерни та ознаки поганого коду. З годом була написана книга "Чистий код" та багато інших книг і робот у цьому напрямку.

Протягом багатьох років ця проблема залишається актуальною та активно обговорюваною. Тому що чистий код та його стандарти дуже часто є суто суб'єктивними. До того ж, дотримання будь яких правил чи конвенцій, має іншу проблематичну сторону. Стандарти кодування існують для того, щоб зробити команди розробників більш продуктивними. Теоретично, вони полегшують розуміння, зміну та тестування коду. На практиці вони можуть створювати небезпечну кількість мета-роботи; команди переписують існуючий код знову і знову в пошуках найбільш правильного та елегантного рішення. Та навіть пошук "елегантних рішень" призводить до великої кількості конфліктів та обговорень.

Хоча якість коду є одним з найважливіших критеріїв при розробці програмного забезпечення, його часто ігнорують. Це призводить до розробки програмного забезпечення низької якості. Таке програмне забезпечення може

функціонувати подібно до програмного забезпечення з належними стандартами кодування, але з часом воно застаріє і призведе до значного технічного боргу.

Отже, першими кроками до розробки були посібники зі стилю кодування на різних мовах. Розробка посібників зі стилів коду, зокрема для Java, - це захоплююча подорож, яка відображає еволюцію практик програмування, співпрацю спільноти та зростаючу увагу до якості та супроводжуваності програмного забезпечення.

Наведемо поглиблений погляд на історію та причини розробки посібників зі стилів Java:

1. Ранні дні Java та початкові посібники зі стилів.
    - Виникнення Java. Java була представлена компанією Sun Microsystems у 1995 році як високорівнева об'єктно-орієнтована мова програмування, розроблена як незалежна від платформи. З перших днів свого існування синтаксис Java зазнав значного впливу мов C та C++, які вже мали усталені правила кодування та рекомендації щодо стилю. Однак Java потребувала власного набору угод, пристосованих до її унікальних особливостей та парадигм.
    - Потреба у стандартизації. У міру того, як Java швидко набувала популярності, стала очевидною потреба в стандартизованому підході до написання Java-коду. Різноманітні стилі кодування призводили до неузгодженостей, що ускладнювало читання, розуміння та підтримку коду. Щоб вирішити цю проблему, Sun Microsystems опублікувала перші офіційні рекомендації щодо стилю кодування Java, які забезпечили основу для написання чіткого, послідовного та зручного для підтримки коду.
2. Розробка формальних посібників зі стилів:
    - Sun Microsystems' Java Coding Conventions

      У 1997 році компанія Sun Microsystems опублікувала "Угоди про кодування мови програмування Java", які стали стандартом де-факто для розробників Java. У цьому документі було висвітлено широкий

спектр тем, зокрема стандартну структуру вихідних файлів Java, послідовну практику відступів для покращення читабельності, правила іменування класів, методів, змінних і констант, найкращі практики для вбудованих коментарів і використання Javadoc, а також настанови щодо використання дужок, пробілів і керуючих структур.

Ці домовленості спрямовані на зменшення когнітивного навантаження на розробників, гарантуючи, що код, написаний різними людьми, буде відповідати єдиному формату.

Вплив відкритого коду. Зростання кількості проектів з відкритим кодом ще більше посилило потребу в стандартизованих посібниках зі стилів. Великі проекти з відкритим кодом, такі як Apache, Eclipse, а пізніше Spring Framework, прийняли і часто розширили конвенції Sun, щоб задовольнити свої специфічні потреби. Ці проекти сприяли колективному розумінню найкращих практик у розробці Java.

3. Еволюція та стандарти, керовані спільнотою:
   - Google's Java Style Guide

     Оскільки популярність Java продовжувала зростати, такі організації, як Google, розробили власні посібники зі стилю, щоб задовольнити свої внутрішні потреби. Google's Java Style Guide, вперше опублікований у середині 2000-х років, став впливовим завдяки авторитету Google у спільноті розробників програмного забезпечення. Цей посібник наголошував на читабельності, простоті та послідовності, а також містив детальні приклади та обґрунтування для кожної настанови.
   - Вплив Oracle

     Після того, як компанія Oracle придбала Sun Microsystems у 2010 році, вона продовжила підтримувати та просувати конвенції кодування Java. Хоча Oracle підтримувала основоположні конвенції, встановлені Sun, вони також визнали еволюційний характер розробки програмного забезпечення і заохочували спільноту до

участі в поширенні передового досвіду.

- Роль інтегрованих середовищ розробки (IDE)

  Розвиток передових IDE, таких як Eclipse, IntelliJ IDEA та NetBeans, також відіграв значну роль у прийнятті стандартів кодування. Ці інструменти пропонували вбудовану підтримку посібників зі стилів і могли автоматично форматувати код відповідно до визначених угод, що полегшувало розробникам дотримання рекомендацій.

- Сучасні розробки та найкращі практики

  Внесок спільноти: спільнота Java за допомогою таких платформ, як Stack Overflow, GitHub та різних груп користувачів Java (JUG), продовжує розвивати та вдосконалювати стандарти кодування. Розроблені спільнотою посібники зі стилів та лінери (наприклад, Checkstyle, PMD) стали важливими інструментами для дотримання стандартів кодування та забезпечення якості коду як у відкритих, так і в корпоративних проектах.

- Специфікація мови Java (JLS).

  JLS, що підтримується компанією Oracle, слугує офіційним довідником з синтаксису та семантики мови програмування Java. Хоча JLS більше фокусується на технічних специфікаціях мови, вона опосередковано впливає на конвенції кодування, визначаючи, як слід писати та інтерпретувати код Java.

4. Посібники зі стилів Java сьогодні

Сьогодні посібники зі стилю Java є більш повними та доступними, ніж будь-коли. Вони охоплюють широкий спектр тем, включаючи сучасні методи програмування, такі як функціональне програмування, паралелізм і використання нових можливостей мови, представлених в останніх версіях Java (наприклад, лямбда-вирази, потоки).

Причини розробки посібників зі стилів мови Java:

- Узгодженість: забезпечення однакового написання коду різними командами та проектами.

- Читабельність: полегшення читання та розуміння коду як для нових, так і для досвідчених розробників.
- Супроводжуваність: спрощення супроводу та оновлень шляхом дотримання загальних практик.
- Співпраця та домовленості: покращення співпраці шляхом зменшення непорозумінь та узгодження з єдиним стандартом кодування.
- Якість: покращення загальної якості коду шляхом заохочення найкращих практик та зменшення ймовірності помилок.
- Інструментальна підтримка: уможливлення використання автоматизованих інструментів для форматування коду, лінкування та статичного аналізу.

Розробка посібників зі стилів Java була зумовлена потребою в узгодженості, читабельності, підтримуваності, співпраці та якості при розробці програмного забезпечення. Починаючи з початкових конвенцій Sun Microsystems і закінчуючи сучасними рекомендаціями спільноти, ці керівництва зі стилю відіграли вирішальну роль у формуванні способів написання та супроводу коду на Java, гарантуючи, що Java залишається надійною та адаптованою мовою для розробників у всьому світі.

5. Зв’язок посібників зі стилю та книги "Чистой код"

Посібники зі стилів коду та принципи, викладені в книзі Роберта Мартіна[1], тісно пов'язані між собою, оскільки обидва мають на меті покращити якість, читабельність та ремонтопридатність коду. Посібники зі стилів коду надають конкретні конвенції та стандарти для написання коду в узгоджений спосіб у команді або організації. Ці настанови охоплюють такі аспекти, як угоди про імена, відступи, інтервали та організацію файлів, гарантуючи, що всі пишуть код, який виглядає і поводиться однаково.

"Чистий код", з іншого боку, заглиблюється у філософію написання коду, який є не лише читабельним, але й є ефективним, гнучким та легким для розуміння. Він наголошує на таких принципах, як осмисленість імен, невеликі функції та уникнення запаху коду, який є індикатором глибших проблем у коді.

У той час як посібники зі стилів більше зосереджені на зовнішньому вигляді та форматуванні коду, "Чистий код" звертається до базових практик та мислення, необхідних для створення високоякісного програмного забезпечення.

Зв'язок між ними полягає в їх спільній меті - сприяти кращій комунікації та співпраці між розробниками. Послідовне форматування коду, як це передбачено посібниками зі стилів, полегшує розробникам читання та рецензування коду один одного, зменшуючи кількість непорозумінь та помилок. Принципи "чистого коду" ще більше посилюють цей ефект, гарантуючи, що код логічно організований, чітко сформульований і вільний від непотрібної складності. Таке поєднання послідовної стилізації та практик чистого кодування призводить до створення кодової бази, яку легше підтримувати та розширювати з часом.

У сучасному програмуванні неможливо переоцінити важливість дотримання як посібників зі стилів коду, так і принципів "чистого коду". Оскільки програмні проекти зростають у розмірі та складності, а команди стають більш розподіленими, здатність швидко розуміти та змінювати код стає вирішальною. Високоякісний код скорочує час, необхідний для адаптації нових розробників, сприяє більш плавному перегляду коду та мінімізує ризик внесення помилок під час змін. Крім того, він забезпечує кращу масштабованість та адаптивність до мінливих вимог, гарантуючи, що програмне забезпечення залишається надійним протягом усього терміну експлуатації.

Зрештою, посібники зі стилів коду та принципи "чистого коду" разом сприяють створенню програмного забезпечення, яке є не лише функціональним, але й елегантним та стійким. Вони заохочують дисциплінований підхід до кодування, який цінує ясність, простоту і відповідальність, що веде до більш продуктивного і гармонійного середовища розробки.

6. Стандарти "Чистого коду".

Стандарти "чистого коду" - це принципи та найкращі практики написання коду, який не лише функціональний, але й зручний для читання, обслуговування та розширення.

Загальні стандарти коду та посібники зі стилів зазвичай наголошують на правилах іменування, відступів, інтервалів та використання специфічних особливостей сови програмування. Вони можуть диктувати, чи використовувати табуляцію або пробіли для відступів, як форматувати коментарі або як називати змінні та функції. Ці настанови гарантують, що код, написаний різними членами команди, виглядатиме узгоджено, що допомагає підтримувати та перевіряти код. Однак вони зазвичай не розглядають проблеми більш високого рівня, наприклад, як структурувати код для зручності читання та супроводу, і саме тут вступають у гру принципи чистого коду.

Стандарти чистого коду включають кілька ключових концепцій та практик. Один з фундаментальних принципів полягає в тому, що код слід писати в першу чергу для людей, а вже потім для комп'ютерів. Це означає, що код повинен бути максимально читабельним і зрозумілим, що часто передбачає використання осмислених імен для змінних, функцій і класів, а також написання невеликих одноцільових функцій. Іншою важливою практикою є уникнення дублювання коду, що часто узагальнюється принципом DRY (Don't Repeat Yourself). Дублювання коду може призвести до проблем з обслуговуванням, оскільки зміни потрібно вносити в декількох місцях.

Крім того, чистий код виступає за чіткі та стислі коментарі до коду, але тільки там, де це необхідно. Сам код повинен бути зрозумілим, що зменшує потребу в коментарях. Якщо коментарі використовуються, вони повинні надавати додатковий контекст або пояснення, які сам код не може передати. Чистий код також підкреслює важливість написання тестів. Юніт-тести та інші форми автоматизованого тестування гарантують, що код працює так, як очікується, і допомагають запобігти регресу при внесенні змін.

Принципи SOLID - ще один важливий аспект чистого коду. Ці принципи надають вказівки для об'єктно-орієнтованого проектування, які сприяють гнучкості та ремонтопридатності:

- Принцип єдиної відповідальності (Single Responsibility Principle, SRP): Клас повинен мати лише одну причину для зміни, тобто він

повинен мати лише одну роботу або відповідальність.

- Принцип відкритості/закритості (Open/Closed Principle, OCP): Програмні об'єкти повинні бути відкритими для розширення, але закритими для модифікації. Це означає, що нову функціональність слід додавати шляхом розширення існуючого коду, а не шляхом його зміни.
- Принцип заміщення Ліскова (LSP): Об'єкти суперкласу повинні бути замінені об'єктами підкласу без порушення коректності роботи програми.
- Принцип розділення інтерфейсів (ISP): Клієнти не повинні залежати від інтерфейсів, якими вони не користуються. Це означає створення спеціальних інтерфейсів для різних клієнтів замість одного універсального інтерфейсу.
- Принцип інверсії залежностей (Dependency Inversion Principle, DIP): Високорівневі модулі повинні залежати не від низькорівневих модулів, а від абстракцій. Це заохочує роз'єднання компонентів.

Ці принципи допомагають створити кодову базу, яку легко зрозуміти, модифікувати та розширювати з часом. Вони доповнюють ширшу філософію чистого коду, надаючи конкретні рекомендації щодо структурування та проектування коду.

Таким чином, стандарти чистого коду зосереджені на загальній якості та ремонтопридатності коду, наголошуючи на читабельності, простоті та ефективних принципах проектування, таких як SOLID. Це на відміну від загальних стандартів коду або посібників зі стилів, які більше зосереджені на узгодженості форматування та синтаксису коду. Практики чистого коду спрямовані на те, щоб зробити код більш зрозумілим і керованим, що в кінцевому підсумку призводить до створення більш надійних і адаптивних програмних систем.

Тому, загальна ідея дослідження - це поєднання різних методів з метою оцінки великих проектів з відкритим кодом. Та надання базового набору

інструментів для подальшого розвитку наукового напрямку.

2.4 Вектор дослідження

Питання доцільності дотримання принципів чистого коду, стилістичних рекомендацій та інших стандартів програмування викликає численні дискусії в професійній спільноті. Хоча ці принципи розроблені для підвищення якості коду, їх практична ефективність залишається предметом дебатів. Одним із ключових аргументів критиків є те, що ці принципи також можуть мати свої обмеження та недоліки.

Дослідження реальних проектів є важливим етапом для оцінки того, наскільки ефективно застосовуються стандарти кодування в практичних умовах. Велика частина сучасних досліджень базується на аналізі вузьких або специфічних випадків, які не дозволяють зробити узагальнення щодо стану кодової бази у великих або складних проектах.

Таким чином, основна мета дослідження полягає у вивченні існуючих проектів, оцінці їхньої поточної якості та відповідності загальноприйнятим принципам програмування. Це дозволить перевірити, чи дійсно дотримання принципів чистого коду має суттєвий вплив на підтримуваність та надійність програмного забезпечення. Це допоможе визначити, чи потребують ці стандарти адаптації до реальних умов розробки

Вектор дослідження орієнтований на комплексне вивчення ефективності принципів чистого коду та стилів програмування в умовах реальної розробки. Основна ідея полягає в аналізі існуючих проектів з метою оцінки того, наскільки дотримання цих принципів впливає на якість програмного забезпечення, його підтримуваність, а також ефективність роботи команд розробників. Для цього необхідно здійснити емпіричне дослідження, що включає збір даних про кодову базу реальних проектів, їхню структуру, складність, кількість помилок і тестове покриття. Особлива увага приділяється порівнянню проектів, що дотримуються принципів чистого коду, із тими, де ці принципи не застосовуються, з метою виявлення реальних переваг або обмежень кожного підходу.

Результати таких досліджень можуть бути доповнені

експериментальними підходами, наприклад, моделюванням розробки одного проекту за різними методологіями. Завершальним етапом є формулювання висновків і практичних рекомендацій щодо того, як адаптувати існуючі принципи до реальних умов розробки, щоб максимально врахувати їхні сильні сторони та мінімізувати можливі недоліки.

Висновки до розділу 2

Принципи чистого коду відіграють ключову роль у розвитку та зростанні кодових баз, оскільки вони спрямовані на забезпечення читабельності, підтримуваності та гнучкості програмного забезпечення. Ці принципи виникли як відповідь на історичні виклики, з якими стикаються розробники, особливо в умовах масштабування проектів та підвищення їхньої складності. На ранніх етапах розвитку програмування багато проектів потерпали через неконтрольований ріст кодової бази, що призводило до технічного боргу, труднощів у підтримці та високих витрат на внесення змін.

Історичне обґрунтування принципів чистого коду ґрунтується на спостереженнях за еволюцією великих систем, де погано структурований код перешкоджав їхньому ефективному розвитку. Наприклад, без чіткого розділення відповідальностей у коді (принципи SOLID), навіть незначні зміни в одній частині системи могли спричиняти неочікувані помилки в інших. Це стало стимулом для створення підходів, які забезпечують мінімізацію зв’язності між компонентами та покращують модульність.

Дослідження у цій сфері важливі, оскільки сучасні проекти стають дедалі складнішими, і без належного дотримання принципів чистоти коду зростають ризики створення неефективних систем, що швидко застарівають. Окрім того, актуальність таких досліджень обумовлена необхідністю перевірки, чи є існуючі принципи універсальними та наскільки вони пристосовані до сучасних умов розробки, наприклад, до роботи з хмарними обчисленнями, мікросервісною архітектурою або DevOps-підходами.

Вивчення принципів чистого коду дозволяє не лише забезпечити довгострокову ефективність розробки, а й сприяє підвищенню якості навчання нових фахівців. Такі принципи створюють основу для розуміння того, як має виглядати добре структурований код, який легко підтримувати та розширювати. Це особливо важливо в умовах постійної зміни вимог бізнесу, коли можливість швидкої адаптації стає конкурентною перевагою для програмного забезпечення.

3 РОЗРОБКА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ДОСЛІДЖЕННЯ ЯКОСТІ КОДУ

У життєвому циклі програмного забезпечення обслуговування часто є найбільш ресурсоємною фазою, яка поглинає значну частину загального бюджету. У міру розширення програмних систем вони, як правило, стають більш складними через взаємозалежності між компонентами, такими як функції, модулі та об'єкти. Ця складність, якщо нею не керувати ефективно, може призвести до появи коду, який важко зрозуміти, модифікувати та тестувати, що, в свою чергу, збільшує витрати на обслуговування.

Одна з головних проблем полягає в тому, що складність часто зростає поступово під час розробки. Початкові проектні рішення або швидкі виправлення, впроваджені для дотримання термінів або вимог користувачів, можуть призвести до заплутаних структур. Такі погано спроектовані системи, які іноді називають «спагеті-кодом», значно ускладнюють супровід. Розробники, яким доручено підтримувати такі системи, часто стикаються з проблемою розшифровки складних взаємозв'язків між елементами коду, що збільшує час і зусилля, необхідні для таких завдань, як виправлення помилок, оновлення функцій і тестування.

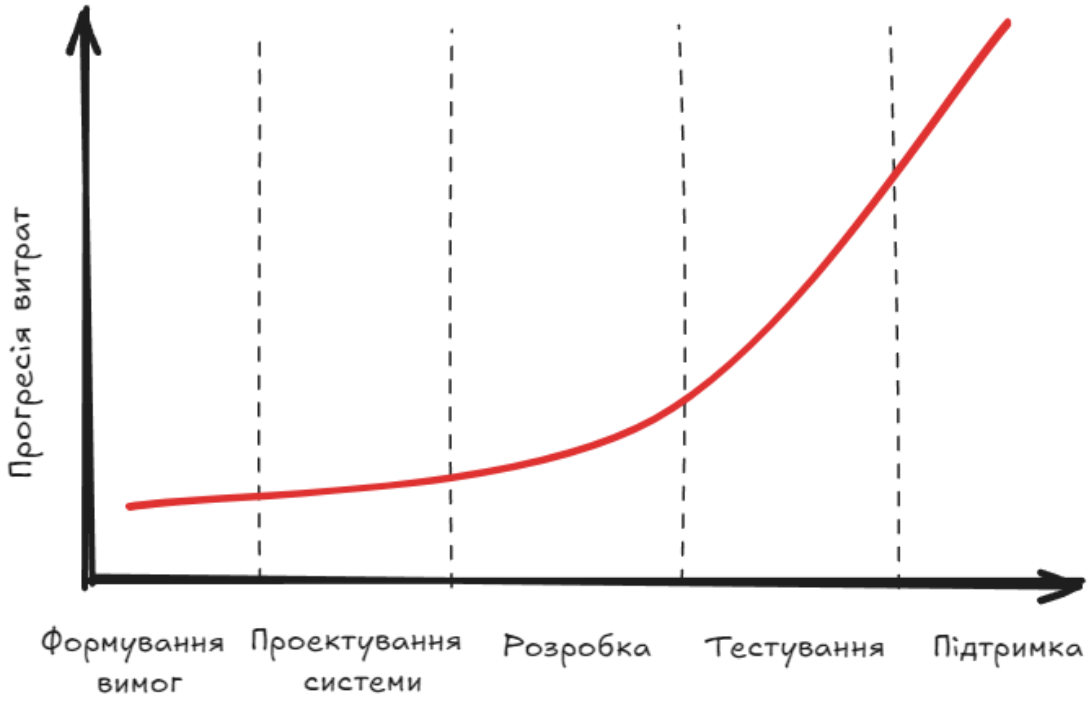


Рисунок 3.1 – Графік росту проектних витрат на різних етапах розробки програмного забезпечення

Інша ключова проблема полягає в тому, що традиційних інженерних методів недостатньо, щоб впоратися з довільною складністю, притаманною програмним системам. На відміну від фізичних систем, де складність часто є передбачуваною і закономірною, складність програмного забезпечення є дуже абстрактною і може бути непередбачуваною. Така природа програмного забезпечення робить його схильним до погіршення структури в міру розвитку, якщо не докладати постійних зусиль для спрощення та рефакторингу кодової бази. Однак такі зусилля часто забирають багато часу і коштів, що ще більше збільшує витрати на обслуговування.

Недостатня увага до супроводжуваності на початку процесу розробки програмного забезпечення призводить до того, що системи складно розширювати або адаптувати до нових вимог. Це не лише збільшує витрати, але й знижує загальну надійність та продуктивність системи, оскільки підтримка складного програмного забезпечення збільшує ймовірність появи нових помилок. Отже, управління складністю стає критично важливим фактором у забезпеченні довгострокової стійкості та якості програмних систем.

Роздивмось проблему на прикладі реальних великих проектів, базуючись на даних з дослідження[25], де проводиться аналіз вартості розвитку різних операційних систем. Це цікаво тим, що код ОС складається з мільйонів рядків, які відповідають за роботу різних підсистем — від ядра до драйверів, файлових систем, мережевих протоколів і графічних інтерфейсів. Така багатошаровість вимагає чіткого планування, особливо з точки зору підтримки, модифікації та оновлення функціоналу. Великі проекти такого типу передбачають розробку й використання складних механізмів контролю версій, тестування, управління змінами та метриками коду. До того ж, такі проекти мають великі бюджети через велику критичність та мають довгий історичний розвиток. На рис. 3.2 – 3.4 наведені дані відношення орієнтовних витрат на розробку та підтримку різних версій операційних систем.

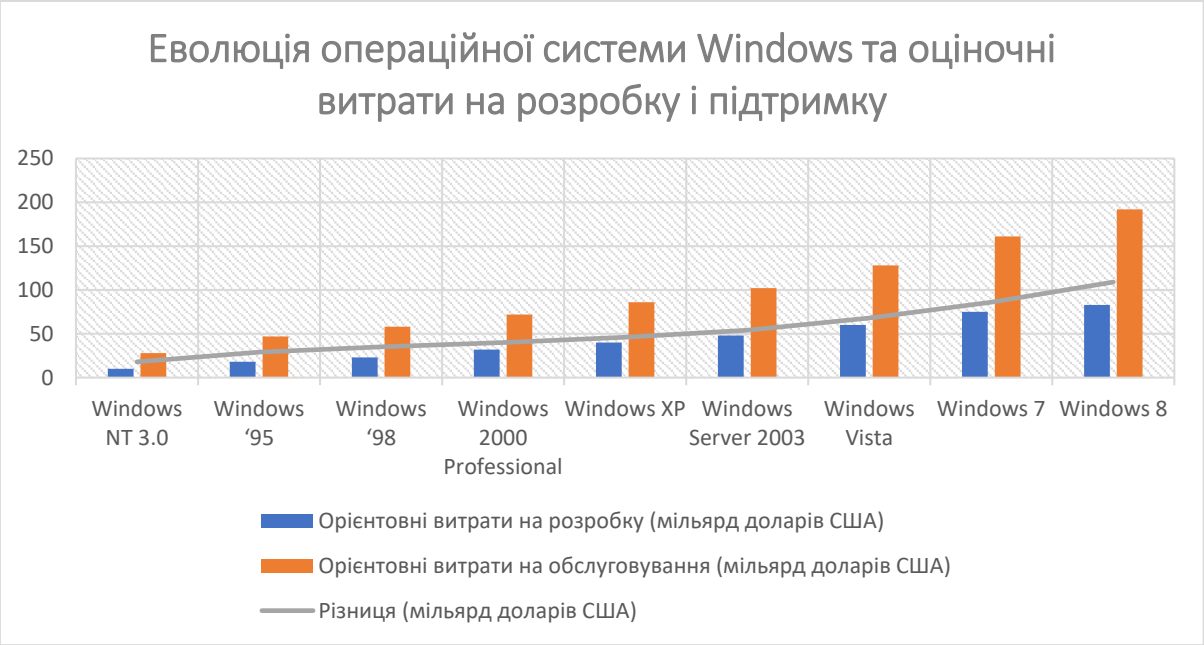


Рисунок 3.2 – Графік відношення орієнтовних витрат на розробку та підтримку різних версій ОС Windows

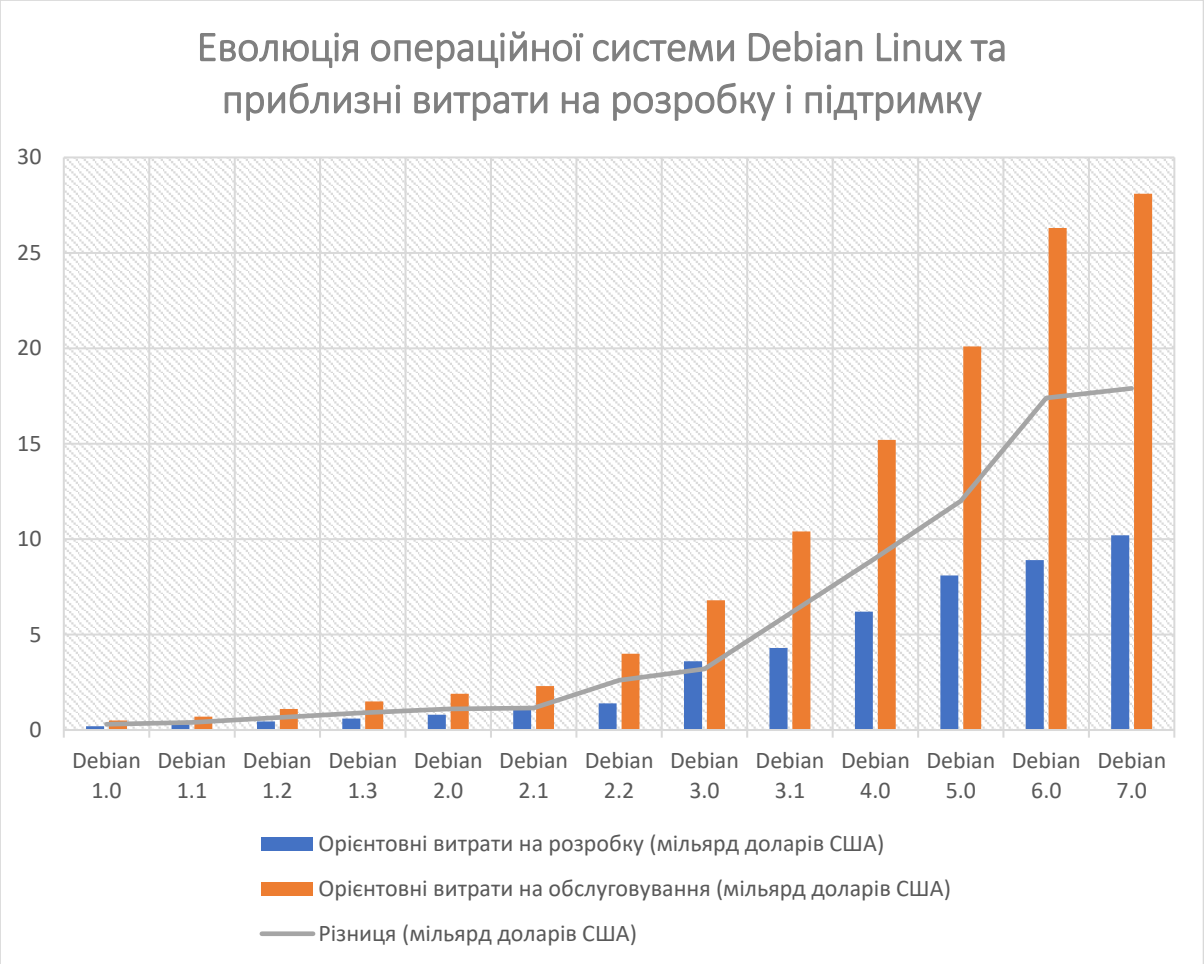


Рисунок 3.3 – Графік відношення орієнтовних витрат на розробку та підтримку різних версій ОС Debian Linux

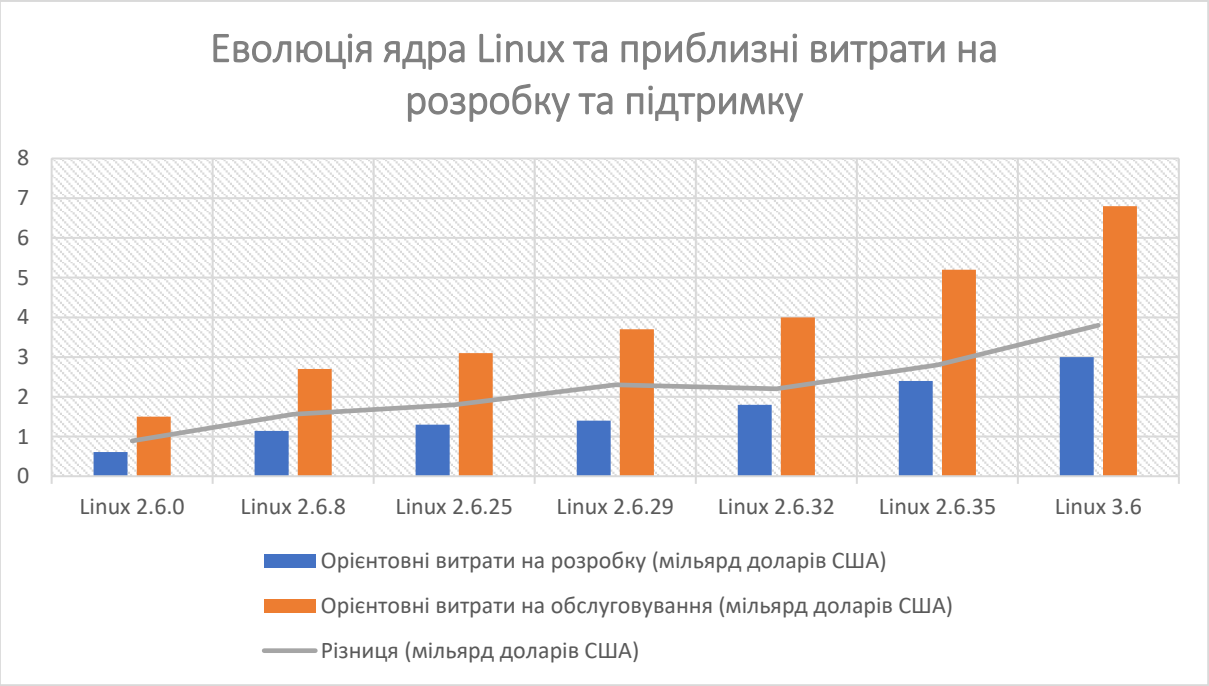


Рисунок 3.4 – Графік відношення орієнтовних витрат на розробку та підтримку різних версій ядра ОС Linux

Витрати на обслуговування значно зростають зі збільшенням розміру та складності загаданих ОС. Також зі збільшенням складності та розміру системи частка зусиль, що спрямовуються на її обслуговування, неухильно зростає. Якщо на початкових етапах на підтримку витрачається близько 60% загальних зусиль, то на пізніх стадіях розвитку програмного забезпечення ця частка може зрости до 80% і більше. Це відображає той факт, що підтримка великих, складних систем вимагає більше ресурсів і часу порівняно з початковою розробкою.

3.1 Формалізація задачі

Першим етапом для постановки завдання є етап формування метрик та чинників, за якими буде виконуватися оцінка характеристик коду. Це етап є найскладнішим, але дозволяє створити чіткі границі у рамках дослідження

Метрики оцінки коду допомагають виміряти якість коду, надаючи об'єктивні дані про різні аспекти кодової бази. Такі метрики, як цикломатична складність, показують, наскільки складним є код, допомагаючи виявити області, які можуть бути схильні до помилок або складні в обслуговуванні. Метрики покриття коду демонструють, наскільки добре код протестований, забезпечуючи надійність і стійкість. Метрики технічного боргу виділяють частини коду, які

можуть потребувати рефакторингу, вказуючи на потенційні довгострокові проблеми з обслуговуванням. Кількісно оцінюючи ці фактори, метрики дають чітке, практичне уявлення про якість і працездатність коду, спрямовуючи поліпшення і забезпечуючи дотримання високих стандартів. Взагалі, навіть просте відстеження якості коду допомагає виявити потенційні проблеми на ранній стадії розробки, запобігаючи їхньому перетворенню на критичні проблеми пізніше.

Є дуже багато вже розроблених метрик, тому мета цієї роботи - зібрати і проаналізувати набір цих метрик та запропонувати стартову модель оцінки якості. Тому в процесі збору інформації, використані доволі відомі метрики оцінки, через те, що багато з них стали основою первинної оцінки коду та показують високу ефективність.

Дуже важливим моментом є той факт, що метрики повинні бути розроблені з урахуванням характеристик та особливостей мови програмування, на якій написаний код для оцінки. З першого погляду здається, що запропонований метод у цьому дослідженні не є повноцінним, тому що метрики не універсальні. Але основна проблематика цієї задачі полягає у тому, що не можливо розробити метрики, тому що різні мови програмування мають різні парадигми та синтаксис. Так, весь код, який розглядається у цій роботі, написаний на мові програмування Java, тому такі метрики можуть бути застосовані або адаптовані для інших мов програмування об’єктно-орієнтованої парадигми.

3.1.1 Обґрунтування вибору мови програмування для оцінки якості

Мова програмування, якою буде написаний проектний код для огляду - є дуже важливим фактором. І для зручного та наочного порівняння треба, що б усі проєкти мали одну мову програмування, тому що це не тільки особливості синтаксису але й методи реалізації бізнес логіки та архітектурної структури коду. Це дуже не просте питання, тому що вибір дуже широкий. І зараз велика кількість мов мають дуже гарну підтримку з точки зору інструментів для підвищення якості коду. Java є чудовим вибором для оцінки якості коду в рамках принципів

чистого коду з наступних причин:

1. Надійна типізація та об'єктно-орієнтовані функції: надійна типізація та надійна модель об'єктно-орієнтованого програмування (ООП) забезпечують чітку структуру та шаблони проектування, які добре узгоджуються з принципами чистого коду, такими як SOLID. Це сприяє розробці модульного, підтримуваного та масштабованого коду.
2. Широкий інструментарій та екосистема: Java має багату екосистему інструментів та бібліотек, які підтримують оцінку якості коду. Такі інструменти, як SonarQube, Checkstyle, PMD та FindBugs забезпечують всебічний статичний аналіз, виявляючи запахи коду, потенційні помилки та дотримання стандартів кодування. Ці інструменти добре інтегровані в робочий процес розробки, забезпечуючи безперервний зворотній зв'язок.
3. Читабельність та зручність супроводу: код на Java відомий своєю читабельністю та узгодженістю, які мають вирішальне значення для чистоти коду. Синтаксис і домовленості мови сприяють написанню коду, який легко читати і розуміти, зменшуючи складність і покращуючи зручність супроводу.
4. Спільнота та найкращі практики: Java має велику, активну спільноту, яка просуває найкращі практики та стандарти якості коду. Існує велика кількість документації, навчальних посібників та спільних знань про написання чистого коду на Java. Підтримка спільноти допомагає розробникам дотримуватися високих стандартів якості коду.
5. Зрілі фреймворки для тестування: Java може похвалитися зрілими та досконалими фреймворками тестування, такими як JUnit та TestNG, які мають вирішальне значення для забезпечення високої якості коду. Ці фреймворки полегшують написання комплексних модульних тестів, дозволяючи розробникам перевіряти правильність функціонування коду та дотримання принципів чистого коду. Добре протестований код за своєю суттю є більш надійним і легшим в обслуговуванні.
6. Підтримка рефакторингу: інтегровані середовища розробки (IDE), такі як

IntelliJ IDEA, Eclipse та NetBeans, пропонують надійні інструменти рефакторингу, спеціально розроблені для Java. Ці інструменти допомагають розробникам реструктурувати існуючий код, не змінюючи його зовнішню поведінку, що полегшує застосування принципів чистого коду, таких як зменшення складності та покращення читабельності.

7. Статичний аналіз та лінтери: екосистема Java включає потужні інструменти статичного аналізу та лінтери, які автоматично перевіряють дотримання принципів чистого коду. Ці інструменти можуть виявляти «запахи» коду, забезпечувати дотримання стандартів кодування та пропонувати покращення, гарантуючи, що кодова база залишається чистою та придатною для обслуговування.
8. Документація та анотації: підтримка Java Javadoc та анотацій дозволяє створювати чітку документацію безпосередньо у коді. Це підтримує практику чистого коду, гарантуючи, що кодова база добре задокументована, а призначення та використання класів і методів зрозумілі.
9. Послідовність та стандартизація: усталені конвенції Java та широке використання в індустрії забезпечують високий рівень узгодженості та стандартизації в проектах. Ця узгодженість допомагає підтримувати принципи чистоти коду, роблячи кодову базу простішою для розуміння та управління.

Таким чином, потужна типізація, надійна модель ООП, широкий інструментарій, читабельність, підтримка спільноти, зрілі фреймворки тестування, інструменти рефакторингу, можливості статичного аналізу, підтримка документації та узгодженість роблять Java ідеальною мовою для аналізу якості коду.

3.2 Базова методологія

Методологія дослідження включає низку підходів і методів, які дозволяють забезпечити надійність і обґрунтованість результатів. Кожне завдання вимагає специфічних інструментів та підходів, щоб забезпечити

глибоке і всебічне вивчення проблеми.

1. Аналіз існуючих принципів та рекомендацій

Для огляду основних концепцій чистого коду використовується методологія літературного огляду. Це передбачає:

- Огляд основних концепцій чистого коду (Clean Code, SOLID, DRY, KISS, тощо).
- Розгляд поширених стандартів стилю програмування (Google Style Guides, Java Style Guide та інші).
- Аналіз книг, статей, наукових публікацій, що стосуються принципів чистого коду (наприклад, робіт Роберта Мартіна та досліджень у галузі програмної інженерії).
- Визначення переваг, недоліків і взаємозв’язків між різними принципами та рекомендаціями.

2. Вивчення реальних проектів:
    - Збір даних про програмне забезпечення з відкритим кодом (Open Source) з GitHub.
    - Вибір проектів із різними рівнями дотримання принципів чистого коду.
    - Створення набору метрик для опису кожного проекту (розмір коду, кількість розробників, вік проекту, частота змін).
3. Оцінка стану кодової бази:
    - Використання інтрументів для оцінки коду за такими параметрами, як цикломатична складність, дублікати, кількість Code Smells.
    - Оцінка складності та обсягу роботи для реалізації змін у коді та опис характеристики кодової бази.
4. Порівняльний аналіз:
    - Порівняння показників та стриктурних особливостей проектів.
    - Аналіз впливу дотримання принципів на процеси розробки, частоту помилок і загальний рівень підтримки.

- Групування проектів: Розділення на групи залежно від рівня дотримання принципів чистого коду.
- Кількісний аналіз: Порівняння груп за метриками (частота помилок, кількість змін коду, тестове покриття).
- Якісний аналіз: Оцінка структурної організації коду для визначення, наскільки вона відповідає принципам чистого коду.

5. Експериментальний підхід

Для моделювання впливу різних підходів до розробки використовується:

- Контрольовані експерименти: Організація процесу розробки однієї функціональності різними командами, де одні працюють із суворим дотриманням принципів, а інші — без них.
- Тестування: Оцінка якості отриманого коду, швидкості розробки та рівня помилок.
- Документування процесу: Детальне фіксування результатів для подальшого аналізу.

6. Інтерпретація результатів

Для забезпечення валідності даних застосовуються методи статистичного аналізу. Python та Excel використовуються як інструменти для візуалізації результатів та виявлення закономірностей, які допоможуть сформувати висновки.

Така методологія дозволяє детально дослідити вплив принципів чистого коду, забезпечуючи надійність та обґрунтованість отриманих висновків.

3.3 Технологічна платформа

Під час реалізації експерименту використовувалася така технологічна платформа.

SonarQube — інструмент для автоматизованого аналізу коду, що використовується для виявлення проблем із якістю програмного коду. Він аналізує код для знаходження потенційних помилок, вразливостей, кодів, що важко підтримувати (code smells), дубльованого коду та проблем із дотриманням стилістичних стандартів.

PMD — інструмент для статичного аналізу коду, який використовується для виявлення проблем з якістю коду в проектах на мовах програмування, таких як Java, JavaScript, XML, і багатьох інших. PMD допомагає розробникам виявляти різноманітні проблеми, включаючи:

Checkstyle — інструмент для статичного аналізу коду, спеціально розроблений для перевірки стилю коду в проектах, написаних на мові Java. Він допомагає розробникам дотримуватися визначених стандартів кодування, виявляючи відхилення від цих стандартів у коді.

SpotBugs — це інструмент для статичного аналізу коду Java, який використовується для виявлення потенційних багів у коді. Він є продовженням та модернізацією проекту FindBugs, який припинив активну підтримку у 2016 році.

SonarQube, PMD, Checkstyle та SpotBugs є потужними інструментами для підтримки високої якості кодової бази Java-проектів. SonarQube виконує комплексний аналіз коду, охоплюючи такі аспекти, як вразливості, технічний борг, відповідність стандартам, підтримуваність, надійність та тестове покриття. Він дозволяє оцінити стан проекту за допомогою метрик якості, таких як безпека та продуктивність, і надає зручний веб-інтерфейс для відстеження змін. PMD фокусується на виявленні проблем стилю та кодування, таких як дублювання коду, використання застарілих конструкцій та порушення кращих практик програмування. Він пропонує гнучкі можливості налаштування правил і може інтегруватися з багатьма інструментами розробки. Checkstyle спеціалізується на перевірці дотримання стилю кодування відповідно до заздалегідь визначених стандартів, що сприяє підтримці читабельності та узгодженості коду в команді. SpotBugs є інструментом статичного аналізу, що дозволяє виявляти потенційні баги в коді, такі як некоректне використання ресурсів, можливі витоки пам’яті та логічні помилки.

Згадані інструменти дозволять всебічно оцінити стан кодової бази в цілях експерименту.

3.4 Використані принципи

Оцінка якості програмного забезпечення є важливим завданням сучасної розробки, оскільки від неї залежить надійність, підтримуваність і загальна ефективність роботи програмного продукту. Для цього використовуються різні підходи, які дозволяють оцінити стан коду, визначити його сильні та слабкі сторони, а також вплив процесів розробки на результат.

У даному дослідженні застосовуються три основні підходи: емпіричний аналіз, кількісні та якісні методи, а також експериментальний підхід. Емпіричний аналіз зосереджений на використанні інструментів статичного аналізу коду та вивченні змін у репозиторіях, що дозволяє будувати об'єктивні висновки на основі реальних даних. Кількісні методи, такі як розрахунок метрик цикломатичної складності або ідентифікація Code Smells, допомагають сформувати вимірювані критерії якості. Якісні методи, у свою чергу, розкривають ширший контекст, оцінюючи продуктивність і командну динаміку.

Експериментальний підхід забезпечує перевірку методів на реальних проектах, дозволяючи провести порівняння різних умов розробки.

3.4.1 Емпіричний аналіз

Емпіричний підхід до оцінки якості коду забезпечує об'єктивність за рахунок використання фактичних даних із репозиторіїв. Інструменти історичного аналізу змін, такі як Git blame та Git log, дозволяють вивчати історію змін у коді, відстежувати внесок розробників та аналізувати причини технічного боргу. Цей підхід акцентує увагу на реальних аспектах розробки, що робить оцінку точнішою.

3.4.2 Кількісні методи

Кількісний аналіз дає можливість обчислити ключові метрики, що визначають якість коду. Наприклад, розрахунок цикломатичної складності допомагає оцінити логічну складність програм, тоді як виявлення Code Smells вказує на потенційні проблеми в архітектурі. Крім того, аналіз продуктивності команд розробників дозволяє об'єктивно виміряти ефективність роботи,

створюючи основу для порівнянь і вдосконалення.

3.4.3 Якісні методи

Доповнення кількісного підходу якісними методами сприяє глибшому розумінню проблем і пошуку їх рішень. Аналіз продуктивності команд у контексті метрик якості, таких як зручність підтримки коду, забезпечує комплексний підхід до оцінки. Це допомагає знайти баланс між структурною складністю та практичною ефективністю.

3.4.4 Експериментальний підхід

Для перевірки ефективності методів аналізу створюється контрольоване середовище, в якому оцінюються 5 різних проектів з відкритим кодом, активно підтримуваних Java-спільнотою. Порівняння цих проектів з урахуванням їхніх особливостей дозволяє не лише тестувати інструменти, але й виявляти специфічні закономірності, що впливають на якість коду.

Усі ці підходи інтегруються для створення багатовимірної моделі аналізу, яка поєднує переваги кількісних і якісних методів, підкріплених емпіричними спостереженнями та експериментами.

Висновки до розділу 3

У третьому розділі була проведена формалізація поставленої задачі оцінки якості коду, а також описані аналітичні та технічні методи, з допомогою яких буде проводитися дослідження. Поєднання різних підходів до аналізу коду дозволяє отримати комплексну оцінку його якості. Емпіричні методи забезпечують об'єктивність завдяки аналізу реальних змін у репозиторіях, тоді як кількісні метрики дають змогу виміряти технічні характеристики коду. Якісні методи доповнюють аналіз, враховуючи динаміку роботи команд розробників. Експериментальний підхід, що включає порівняння реальних проектів, дозволяє перевірити ефективність різних методів оцінки. Якісні методи аналізу доповнюють традиційні кількісні підходи, враховуючи динаміку співпраці та комунікаційні процеси в командах розробників. Експериментальний підхід,

заснований на порівнянні реальних проектів, дозволяє емпірично оцінити ефективність різних методів.

Інтеграція якісних і кількісних підходів створює надійну основу для оцінювання якості програмного забезпечення та вдосконалення процесів розробки. Це сприяє виявленню оптимальних практик, що підвищують ефективність процесу розробки та підтримки і якість кінцевого продукту.

4 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ПІДТРИМКИ ТА МАСШТАБУВАННЯ КОДУ

Метою цього розділу є детально описати хід і процес виконання експериментів, спрямованих на пошук оптимальних методів оцінки рівня підтримки програмного забезпечення. Особливу увагу приділено визначенню критеріїв та методик, які дозволяють об'єктивно аналізувати якість програмного забезпечення на основі емпіричних даних.

У межах цього дослідження буде запропоновано нову методологію оцінки, яка враховує специфіку великих програмних проектів, їхню складність і динаміку розвитку. Представлений підхід забезпечить надання важливої та унікальної інформації про поточний стан таких проектів, включно з їхньою підтримуваністю, що у підсумку сприятиме вдосконаленню процесів розробки та підвищенню загальної якості програмного забезпечення.

4.1 Підготовка до експерименту

Підготовка до експерименту була виконана наступним чином:

- розробка метрології оцінки та формування метрик,
- підбір проектів,
- підбір та конфігурація інструментів оцінки,
- розробка додаткових методів збору та обробки інформації.

4.1.1 Опис підходу до аналізу проектів

Експеримент складається з трьох ключових етапів, кожен з яких спрямований на аналіз різних аспектів розробки програмного забезпечення та активності спільноти.

1. Збір інформації про проекти з допомогою інструментів статичного аналізу
   На першому етапі здійснюється збір даних про технічні характеристики програмних проектів. Для цього використовуються інструменти статичного аналізу, які допомагають оцінити якість коду, виявити потенційні проблеми або уразливості, а також надати загальну характеристику коду.
2. Збір статистики про активність розробки через систему контроля версій за допомогою проекту GitStats

Другий етап зосереджений на аналізі динаміки розробки проектів. Застосовуючи інструмент GitStats, здійснюється збір статистики про зміни в коді: частоту комітів, кількість залучених розробників, їхню активність, тривалість та інтенсивність робочих періодів. Це дозволяє зрозуміти, наскільки активно ведеться розробка і наскільки стабільно розвивається проект.

3. Збір статистики про рівень активності, підтримки та залученості спільноти у різних проектах на GitHub

   Третій етап спрямований на аналіз соціальної активності навколо проектів. Зокрема, розглядаються такі ключові показники, як кількість зірок (stars), форків (forks), відкритих і закритих питань (issues), обговорень, а також активність у створенні pull-запитів. Для автоматизованого збору та аналізу цієї інформації були розроблені Python-скрипти. Вони виконують парсинг даних із GitHub і надають змогу отримати структуровану статистику про те, як активно підтримується проект, наскільки він популярний і які зусилля залучає для свого розвитку.

Ці три етапи в сукупності дозволяють сформувати цілісну картину про технічний стан проекту, активність його розробки та рівень підтримки в спільноті.

4.1.2 Опис використаних проектів для оцінки

До експерименту було включено п’ять відомих Java-бібліотек та фреймворків, які широко використовуються в розробці: Dubbo, Guava, JUnit5, Mockito та JUnit 4. Ці проекти є популярними і широко використовуваними в Java-спільноті, що робить їх ідеальними для оцінки якості стану кодової бази.

1. Висока якість коду
   - Відповідність стандартам Java: Ці проекти використовують найкращі практики Java, включаючи сучасні підходи до написання коду, модульність, читабельність та використання стандартних конструкцій.
   - Чистий код: Вони мають зрозумілу структуру, використовують

зрозумілі імена змінних та методів, і дотримуються принципів SOLID.

- Рефакторинг: Часто такі проекти проходять регулярний рефакторинг для покращення якості коду та адаптації до нових стандартів Java (наприклад, використання лямбда-виразів у Guava або нових фіч Java 8/11 у JUnit5).

2. Висока популярність та використання у промислових умовах
    - Guava використовується майже в кожному другому Java-проєкті для роботи з колекціями, кешування та утиліт.
    - JUnit (4 і 5) — це де-факто стандарт для тестування Java-коду.
    - Mockito популярний у сфері тестування, завдяки простоті створення "заглушок" (mock-об'єктів) і інтеграції з JUnit.
    - Dubbo є одним з основних фреймворків для розподілених систем і мікросервісів в екосистемі Java.
    - Популярність цих проєктів забезпечує великий пул розробників і активне використання в реальних програмах, що підтверджує їх якість і адаптацію до реальних потреб.
3. Активне підтримання та розвиток
    - Ці проєкти мають активні спільноти розробників, які підтримують їх актуальність та стабільність.
    - Регулярні оновлення слідують за розвитком Java (наприклад, перехід на нові версії JDK або інтеграція з іншими популярними бібліотеками).
    - Код бази таких проєктів постійно покращується завдяки внескам від досвідчених розробників із відкритої спільноти та корпоративного сектору.
4. Використання кращих практик тестування
    - JUnit та Mockito самі по собі є інструментами тестування, і тому їх код є чудовим прикладом використання методологій тестування,

таких як TDD (Test-Driven Development) та BDD (Behavior-Driven Development).

- Ці проекти мають високий рівень покриття тестами, що дозволяє легко відслідковувати якість і стабільність.

5. Приклади вирішення складних задач
    - Guava вирішує складні задачі роботи з потоками, кешами, колекціями та іншими проблемами, які часто зустрічаються у Java-розробці.
    - Dubbo — приклад складної інфраструктури для RPC, що охоплює багато аспектів: від серіалізації до балансування навантаження.
    - Ці проєкти ілюструють принципи роботи з продуктивними, розподіленими та масштабованими системами. Вони демонструють підходи до оптимізації обчислень, управління ресурсами та забезпечення стабільності роботи в умовах значного навантаження.
6. Відкрита кодова база
    - Відкритий код цих проєктів дозволяє всім охочим переглядати та вивчати їх.
    - Це забезпечує прозорість і дає можливість оцінити архітектурні рішення, стилі програмування, патерни проєктування (наприклад, використання Dependency Injection в Mockito або модульного дизайну в JUnit5).
7. Репрезентативність для Java-екосистеми
    - Ці проекти покривають широкий спектр задач Java-розробки: від бібліотек утиліт (Guava) до тестування (JUnit/Mockito) і мікросервісів (Dubbo).
    - Вони підходять для оцінки різних аспектів: архітектури, продуктивності, тестування, відповідності стандартам Java та загальної культури написання коду.

Таким чином, ці проекти є гарними прикладами для проведення аналізу, адже вони поєднують в собі найкращі практики, великий досвід спільноти та

високу актуальність у Java-екосистемі.

4.1.3 Опис використаного програмно-апаратного середовища

Для оцінки якості коду Java-проектів використовувалося програмно-апаратне середовище, яке включає IntelliJ IDEA як основне інтегроване середовище розробки. Ця IDE забезпечує ефективну підтримку процесу перевірки коду завдяки інтеграції з різними інструментами статичного аналізу. SonarLint і SonarQube застосовувалися для виявлення помилок, потенційних вразливостей та забезпечення дотримання стандартів якості коду. SpotBugs дозволив аналізувати Java-байт-код для виявлення типових помилок і проблем. Інструмент Qodana використовувався для локального аналізу коду безпосередньо в IntelliJ IDEA з можливістю налаштування правил перевірки. PMD і Checkstyle слугували для оцінки відповідності коду стандартам стилю, дотримання правил написання коду та виявлення дублювання. Використання цих інструментів дозволило комплексно оцінювати та підвищувати якість коду на всіх етапах розробки Java-проектів.

Окрім інструментів статичного аналізу, для експерименту застосовувалися додаткові методи аналізу проектів. Зокрема, плагін Statistic[3] для IntelliJ IDEA використовувався для збору метрик щодо структури та динаміки коду. GitStats забезпечував аналіз історії змін у репозиторії, дозволяючи відстежувати еволюцію коду та активність розробників. Додатково був використаний Python-скрипт, створений для розрахунку кастомних метрик і візуалізації даних про якість коду. Такий підхід дозволив отримати комплексне уявлення про якість коду, враховуючи як статичний аналіз, так і додаткові метрики проектної діяльності.

4.1.4 Конфігурація та розробка програмних інструментів збору даних

Як було описано вище для оцінки якості коду були виростанні такі інструменти як: лінтер IntelliJ IDEA, SonarLint, SonarQube, SpotBugs, Qodana, PMD, Checkstyle. SonarLint, PMD, Checkstyle та іноді SpotBugs використовувалися у форматі плагінів для зручної інтеграції проектів та

уникнення додавання додаткових залежностей у код. Для збору інформації за допомогою локального сервера використовувалися SonarQube, Qodana та іноді SpotBugs. Залежність SonarQube була додана у кожний проект, та сам сервер запускався у віртуальному середовищі від Docker за допомогою наступної конфігурації (docker-compose.yml):

```
version: "3"

services:
  sonarqube:
    image: sonarqube:lts-community
    depends_on:
      - sonar_db
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://sonar_db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    ports:
      - "9000:9000"
    volumes:
      - sonarqube_conf:/opt/sonarqube/conf
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
      - sonarqube_temp:/opt/sonarqube/temp
  sonar_db:
    image: postgres:13
    environment:
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
      POSTGRES_DB: sonar
    volumes:
      - sonar_db:/var/lib/postgresql
      - sonar_db_data:/var/lib/postgresql/data

volumes:
  sonarqube_conf:
  sonarqube_data:
  sonarqube_extensions:
  sonarqube_logs:
  sonarqube_temp:
  sonar_db:
  sonar_db_data:
```

GitStats – це відкритий проект, тому збір інформації для кожного з репозиторіїв проводився локально.

Для додаткового збору інформації були використані програмні

компоненти, розроблені на Python, специфічно для потреб цього дослідження. Технічне завдання та текст програм(див. Додаток А та Додаток Б).

4.2 Проведення експерименту

Кожен з етапів експерименту був проведений для кожного з п’яти проектів незалежно від особливостей структури проектів, щоб забезпечити незалежність експерименту від зовнішніх факторів.

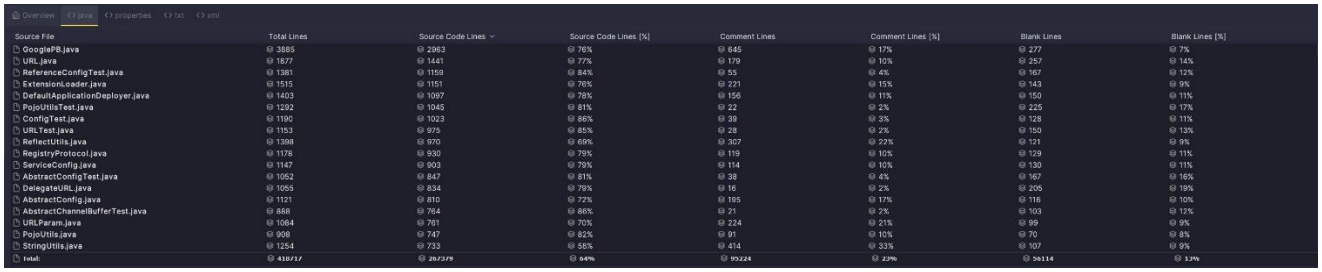
4.2.1 Apache Dubbo

Apache Dubbo — це RPC-фреймворк, створений Alibaba у 2011 році для побудови розподілених систем і мікросервісів. Завдяки високій продуктивності та гнучкості швидко став популярним, особливо в Китаї. У 2018 році проект був переданий до Apache Software Foundation, де продовжує активно розвиватися як частина глобальної екосистеми відкритого коду.

Структурна характеристика проекту:

Проект має зручну модульну структуру. Однак проекті все ще присутні деякі старі пакеті від Alibaba, що не є гарним з точки зору залежностей, але спільнота веде активну роботу над видаленням старих елементів. У проекті присутня базова та детальна документація(для роботи з кодом, або для контриб’юторів та навіть шаблон для пулл-реквкстів). Має свій шаблон для стандартів кодування проекту. Все описано дуже детально, але коротко та зрозуміло. З точки зору коду, код є дуже добре задокументованим та структурованим, його легко читати та розуміти, однак присутні деякі старі пояснювальні коментарі або мертвий код (навіть у нових пакетах). Загалом, проект має читальний та зрозумілий код, так, у проекту все ще присутні завеликим файли, багато непотрібних пояснень та мертвого коду, але над цим ведеться активна, особливо беручі до уваги, що проекту вже 13 років. Dubbo слідує загальним стандартам написання коду та має чудовий рівень підтримки.

Роздивимось загальну статистику рядків коду у проекті(рис 4.1):



Overview | java | properties | txt | xml

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
GooglePB.java	3885	2963	76%	845	17%	277	7%
URL.java	1877	1441	77%	179	10%	257	14%
ReferenceConfigTest.java	1381	1159	84%	55	4%	167	12%
ExtensionLoader.java	1515	1151	76%	221	15%	143	9%
DefaultApplicationDeployer.java	1403	1097	78%	156	11%	150	11%
PojoUtilsTest.java	1292	1045	81%	22	2%	225	17%
ConfigTest.java	1190	1023	86%	39	3%	128	11%
URLTest.java	1153	975	85%	28	2%	150	13%
ReflectUtils.java	1398	970	69%	307	22%	121	9%
RegistryProtocol.java	1178	930	79%	119	10%	129	11%
ServiceConfig.java	1147	903	79%	114	10%	130	11%
AbstractConfigTest.java	1052	847	81%	38	4%	167	16%
DelegateURL.java	1055	834	79%	16	2%	205	19%
AbstractConfig.java	1121	810	72%	195	17%	116	10%
AbstractChannelBufferTest.java	868	764	86%	21	2%	103	12%
URLParam.java	1084	761	70%	224	21%	99	9%
PojoUtils.java	906	747	82%	91	10%	70	8%
StringUtils.java	1254	733	58%	414	33%	107	9%
Total:	418717	287379	64%	95224	23%	36114	13%

Рисунок 4.1 – Загальна статистика проекту Dubbo

Рядки коду Dubbo становлять приблизно 64% від загальної кількості рядків проекту. Це типовий показник для великих open-source проектів, де значний обсяг займають тестування, документація та файли налаштувань, що критично важливі для підтримки якості та використання проекту.

Проведемо сканування проекту за допомогою інструментів статичного аналізу (рис. 4.2 — 4.9):

Рисунок 4.2 – Результати сканування проекту Dubbo з InteliJ

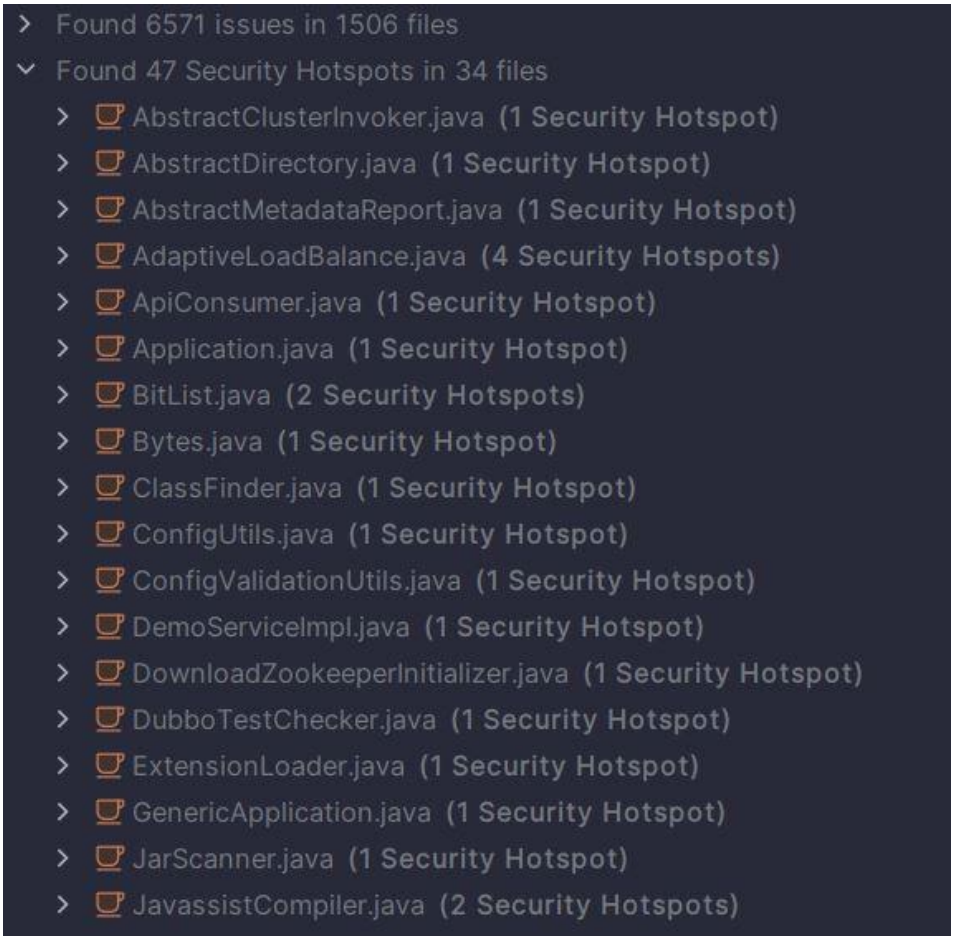


Рисунок 4.3 – Результати сканування проекту Dubbo з SonarLint

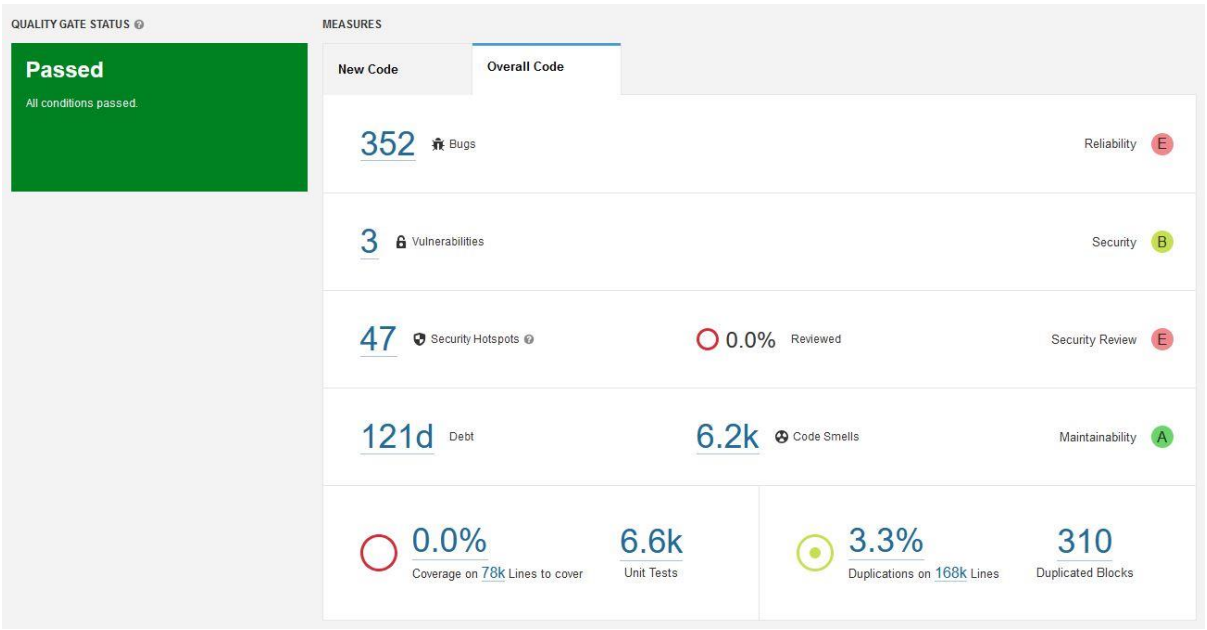


Рисунок 4.4 – Результати сканування проекту Dubbo з SonarQube

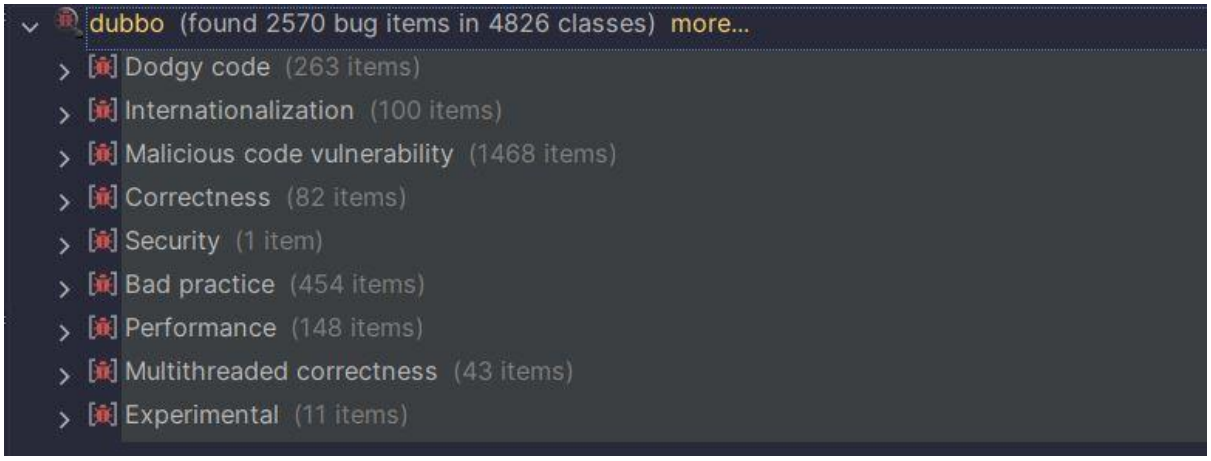


Рисунок 4.5 – Результати сканування проекту Dubbo з SpotBugs

Рисунок 4.6 – Результати сканування проекту Dubbo з Qodana

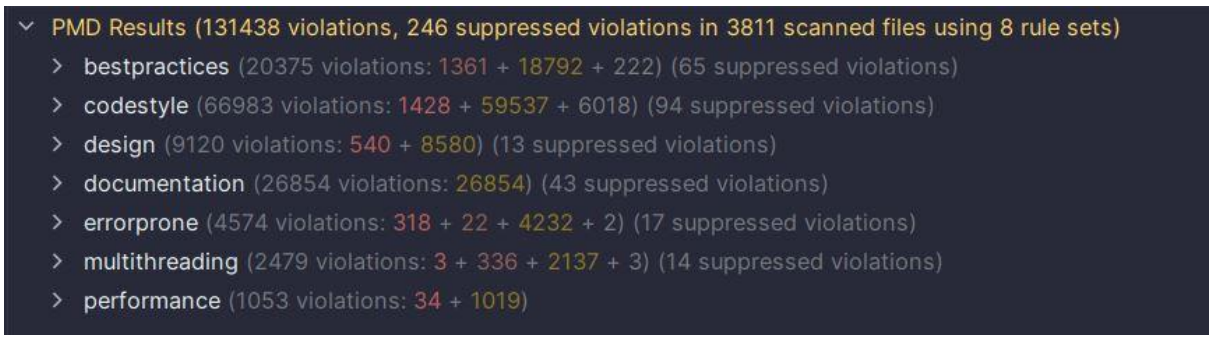


Рисунок 4.7 – Результати сканування проекту Dubbo з PMD

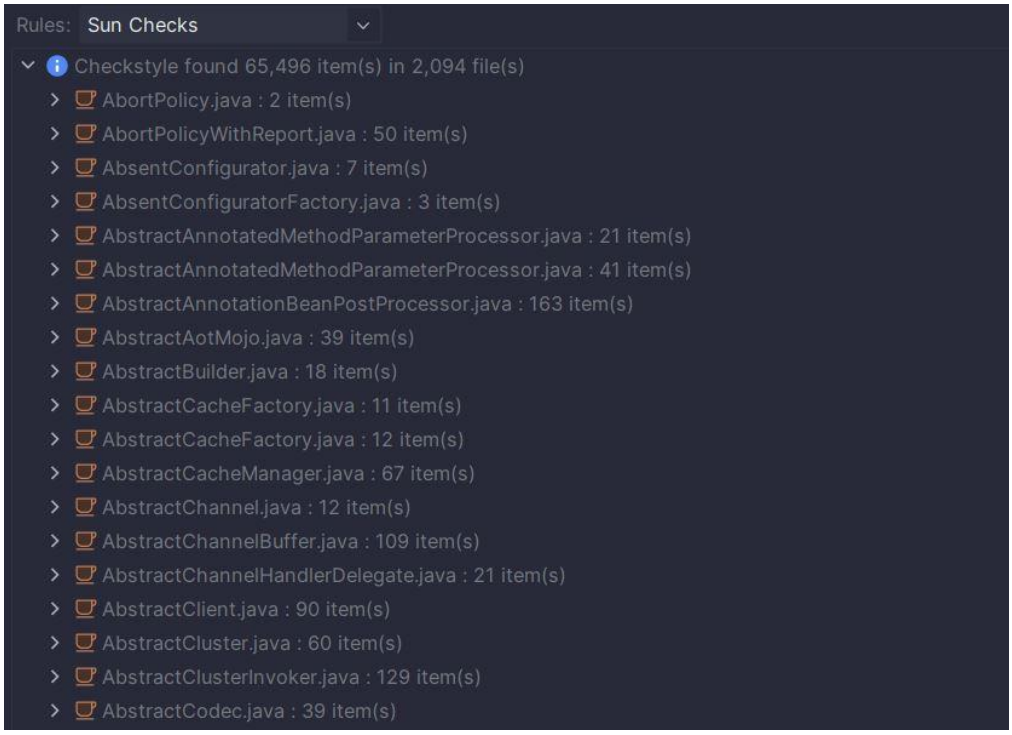


Рисунок 4.8 – Результати сканування проекту Dubbo з CheclStyle та посібником зі стилю від Sun

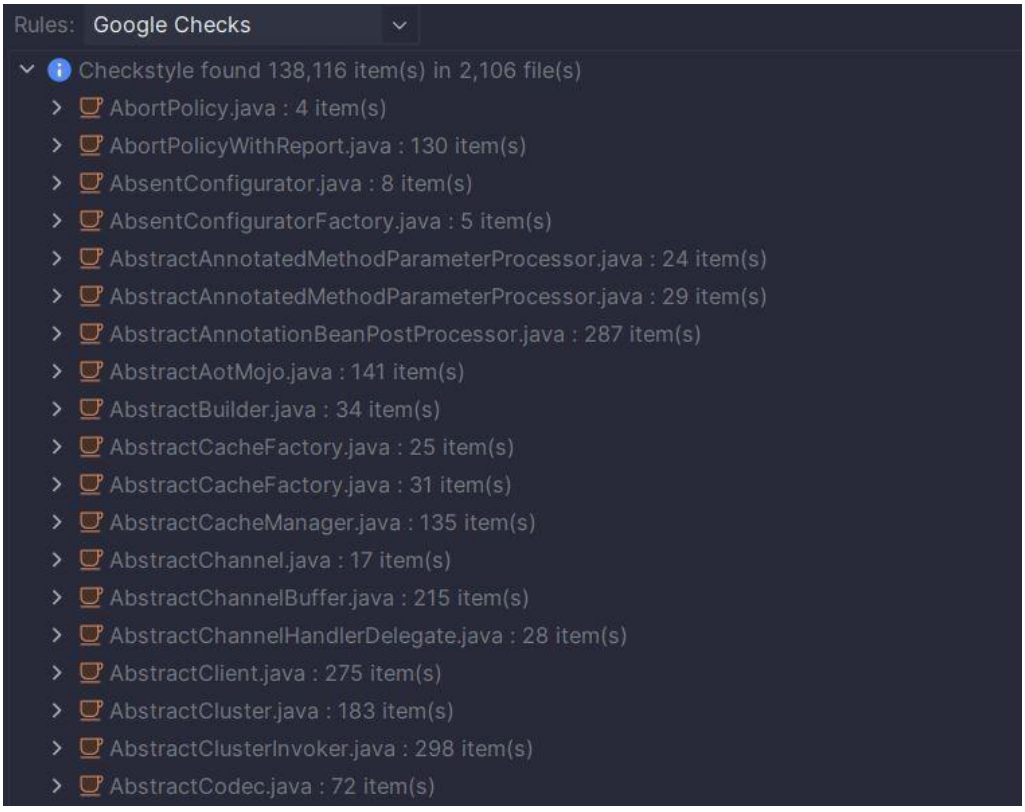


Рисунок 4.9 – Результати сканування проекту Dubbo з CheclStyle та посібником зі стилю від Google

Найбільша кількість помилок на рядок коду виявлена аналізаторами

CheckStyle Sun (0.19) та PMD (0.5). IntelliJ показує помірну кількість помилок (0.028), а інші аналізатори, такі як SonarQube, CheckStyle Google та SonarLint, виявляють значно меншу кількість помилок (0.07 або менше). Це свідчить про те, що різні статичні аналізатори використовують різні підходи до виявлення проблем у коді, і деякі з них є більш суворими.

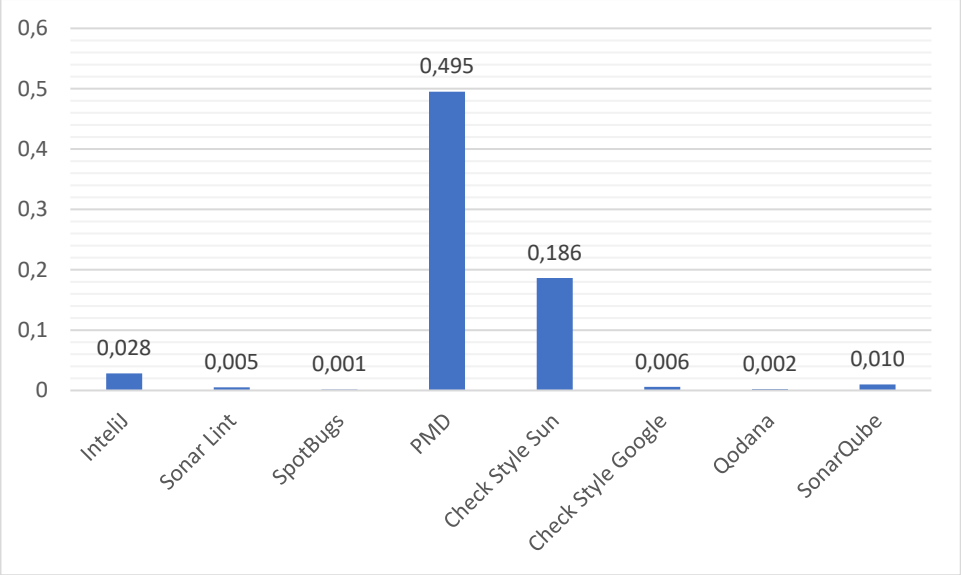


Рисунок 4.10 – Кількість помилок на рядок коду Dubbo

Базуючись на даних від Statistic виберемо десять найбільших файлів та проведемо сканування кожного з них(табл. 4.1) та побудуємо графік порівняння кількості помилок від різних статичних аналізаторів(рис. 4.10):

Таблиця 4.1 – Результати сканування десяти найбільших файлів проекту Dubbo

Назва файлу	Кількість рядків вихідного коду	InteliJ	Sonar Lint	Spot Bugs	PMD	Check Style Sun	Check Style Google
org.apache.dubbo.common.URL	1441	50	30	15	878	961	1534
org.apache.dubbo.common.extension.ExtensionLoader	1151	90	46	4	422	374	1090
org.apache.dubbo.config.deploy.DefaultApplicationDeployer	1097	60	33	7	338	223	968
org.apache.dubbo.common.utils.ReflectUtils	970	200	91	2	439	203	961
org.apache.dubbo.registry.integration.RegistryProtocol	930	69	32	6	385	343	782

org.apache.dubbo.config.ServiceConfig	903	122	22	7	263	245	815
org.apache.dubbo.config.AbstractConfig	810	81	39	8	347	268	784
org.apache.dubbo.common.url.component.URLParam	761	43	18	3	300	218	735
org.apache.dubbo.common.utils.PojoUtils	747	99	41	5	321	180	720
org.apache.dubbo.common.utils.StringUtils	733	86	15	1	361	220	799

Файл org.apache.dubbo.common.URL має найбільшу кількість рядків коду (14412), а також найбільше помилок за аналізаторами CheckStyle Google та Sun та PMD (878). Помітно, що аналізатори CheckStyle Google та CheckStyle Sun завжди фіксують більшу кількість помилок порівняно з іншими інструментами.

Інші файли з великою кількістю помилок:

- org.apache.dubbo.common.extension.ExtensionLoader (11518 рядків)
- org.apache.dubbo.config.deploy.DefaultApplicationDeployer (10973 рядків)
- org.apache.dubbo.config.ServiceConfig (9033 рядків).

Високі значення для цих файлів вказують на їх складність і необхідність додаткового рефакторингу чи оптимізації.

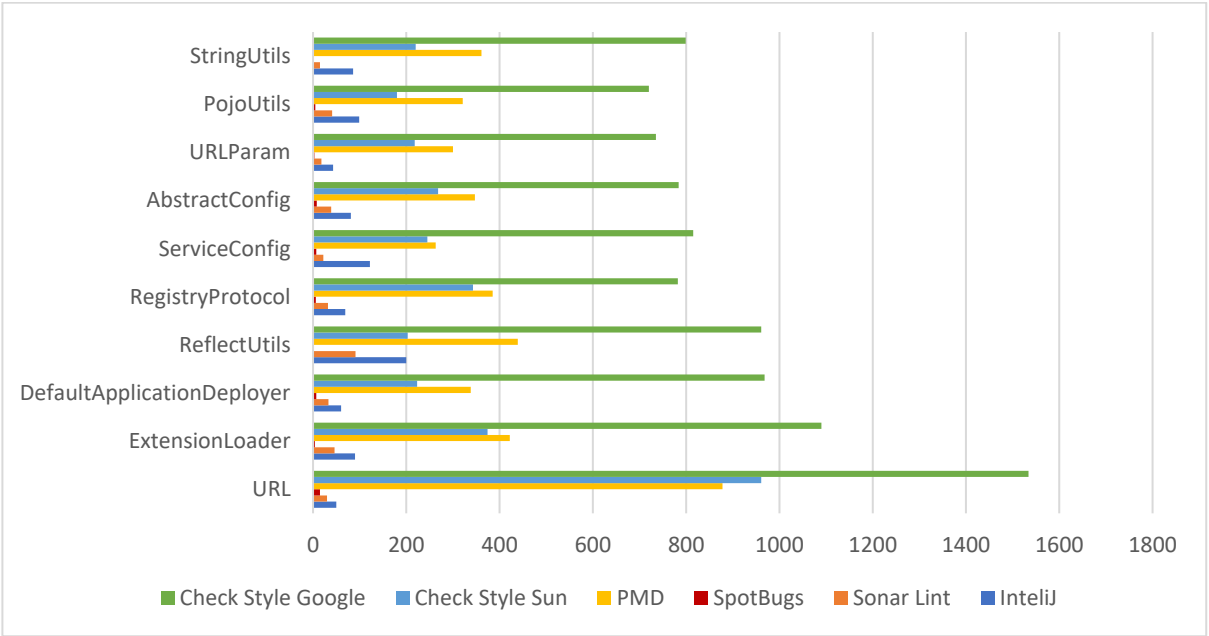


Рисунок 4.11 – Порівняння результатів сканування десяти найбільших файлів

проекту Dubbo

Графік (рис 4.11) підтверджує, що CheckStyle Google та CheckStyle Sun знаходять найбільшу кількість проблем у файлах. Інструменти SpotBugs, SonarLint та IntelliJ виявляють значно менше помилок, що може вказувати на їхню фокусованість на певних типах дефектів (наприклад, логічні помилки або помилки продуктивності).

Найбільші файли проекту містять більше помилок, що є очікуваним через їх складність. Аналізатори CheckStyle є найбільш суворими, що вказує на потребу у дотриманні стандартів кодування. Фокусування на великих файлах, як URL, ExtensionLoader та ServiceConfig, може значно зменшити кількість помилок у проекті.

Розглянемо Git статику:

Проекту 4608 днів, з них 1734 дні були активними (приблизно 37.63% часу). Всього коммітів: 7549, що в середньому становить 4.4 комміта на активний день або 1.6 комміта на день протягом усього часу існування проекту. Це свідчить про помірну, але нерівномірну активність протягом тривалого періоду. У проекті взяли участь 633 автори, але в середньому на одного автора припадає 11.9 коммітів. Це вказує на досить велику спільноту з активною меншістю, яка здійснює основний обсяг роботи. Загалом 4391 файл було змінено, а кількість доданих рядків коду (119015) значно поступається кількості вилучених рядків (731710), що свідчить про оптимізацію чи рефакторинг проекту.

З рис. 4.12 видно, що пік активності припадає на кілька періодів, особливо на 2011-2013 роки та 2019-2023 роки. Місяці з дуже високою активністю свідчать про інтенсивні етапи розробки чи впровадження нових функцій.

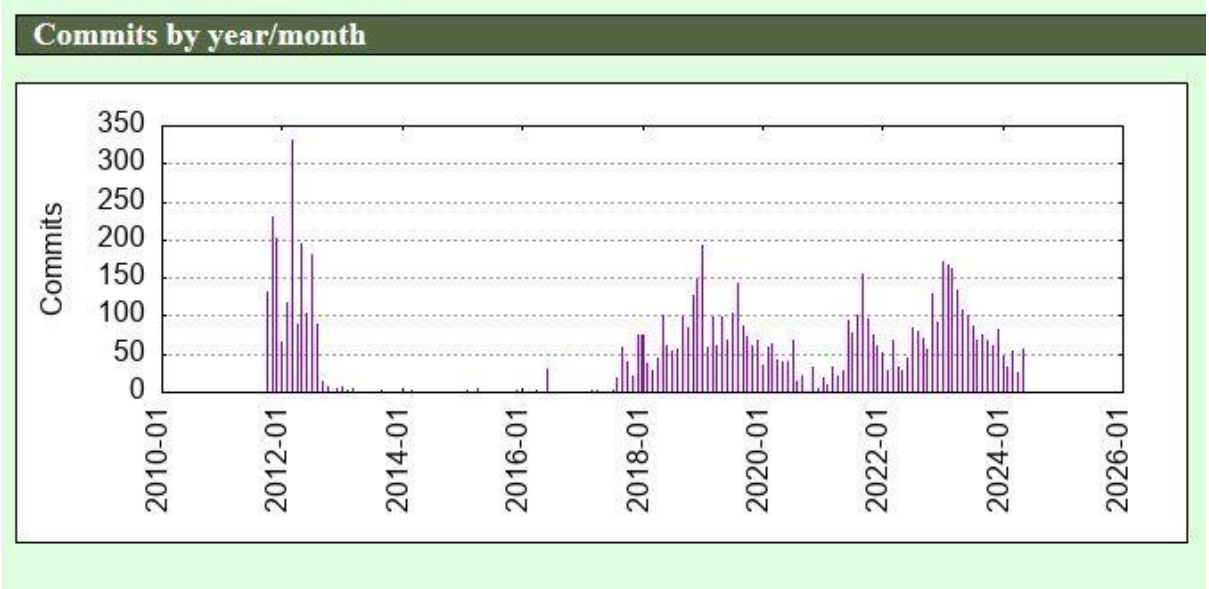
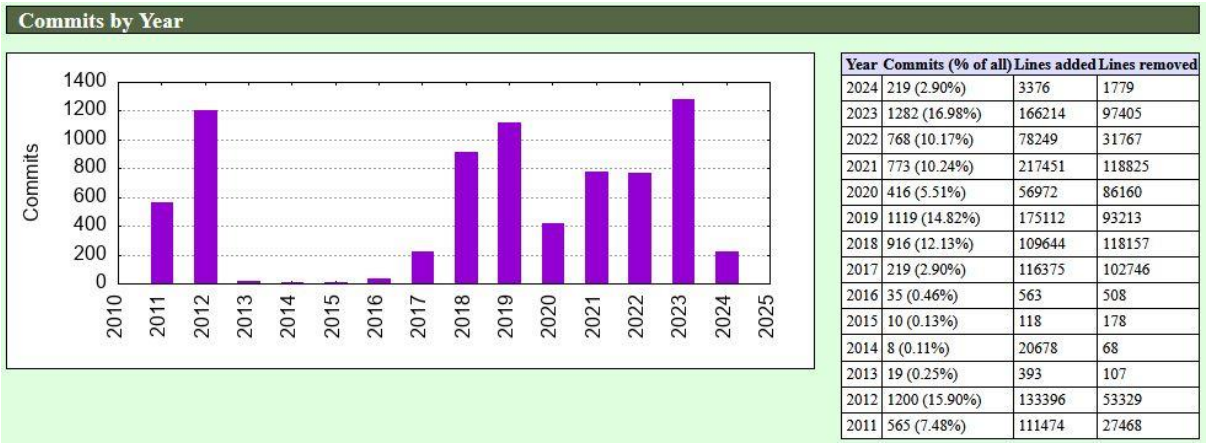


Рисунок 4.12 – Статистика коммітів за місяцями у проекті Dubbo

З рис. 4.13 видно, що найбільша активність спостерігалась у 2023 році з 1282 коммітами (17%) та у 2012 році. Активність знизилася між 2012-2018 роками, але знову відновилася у 2018-2023 роках. Це вказує на циклічність проекту з періодами зростання активності, можливо, пов’язаними з новими релізами чи змінами у команді.



Year	Commits (% of all)	Lines added	Lines removed
2024	219 (2.90%)	3376	1779
2023	1282 (16.98%)	166214	97405
2022	768 (10.17%)	78249	31767
2021	773 (10.24%)	217451	118825
2020	416 (5.51%)	56972	86160
2019	1119 (14.82%)	175112	93213
2018	916 (12.13%)	109644	118157
2017	219 (2.90%)	116375	102746
2016	35 (0.46%)	563	508
2015	10 (0.13%)	118	178
2014	8 (0.11%)	20678	68
2013	19 (0.25%)	393	107
2012	1200 (15.90%)	133396	53329
2011	565 (7.48%)	111474	27468

Рисунок 4.13 — Статистика коммітів за роками у проекті Dubbo

Кількість файлів у проєкті Dubbo (рис. 4.14) поступово збільшувалася з 2011 року. Різке зростання кількості файлів спостерігалося у 2012-2013 роках та 2020-2022 роках, що може свідчити про активний розвиток і додавання нових функціональних можливостей. Після 2022 року приріст кількості файлів продовжується, але сповільнюється, стабілізуючись на рівні приблизно ~4300 файлів.

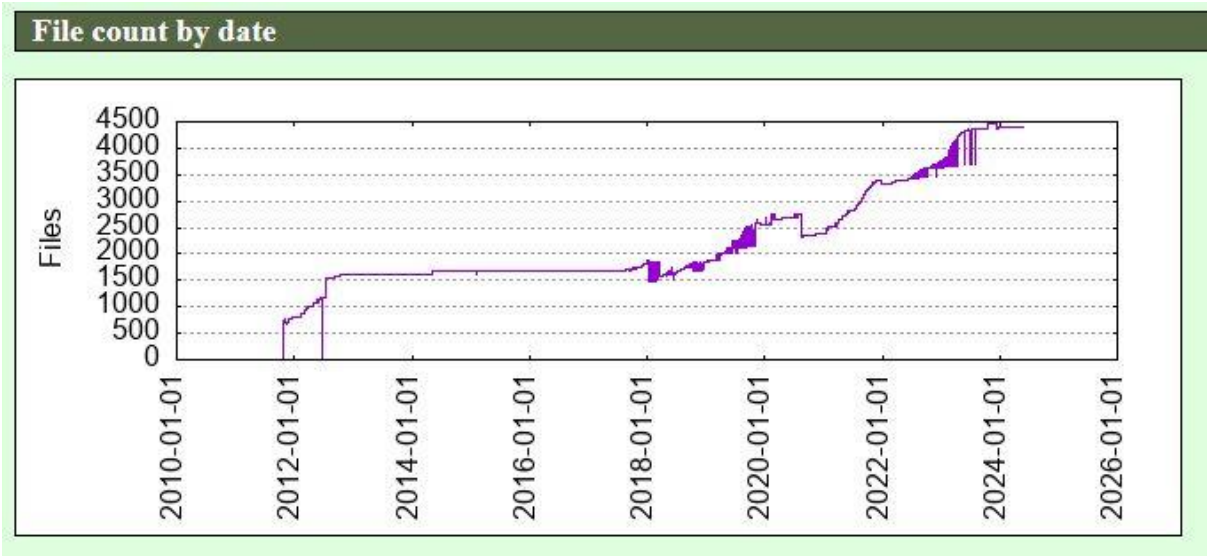


Рисунок 4.14 – Статистика кількості файлів у проекті Dubbo

Кількість рядків коду(рис 4.15) також постійно зростає, що підтверджує активну розробку проекту протягом усього періоду. Значне зростання рядків коду помітне у 2012-2013 роках, а потім у 2019-2022 роках, що корелює з періодами збільшення кількості файлів. У 2023 році кількість рядків коду перевищила 450 000, що вказує на масштабність і зрілість проекту. Помітні "плато" на графіку свідчать про періоди стабілізації, рефакторингу або зупинки активного розвитку.

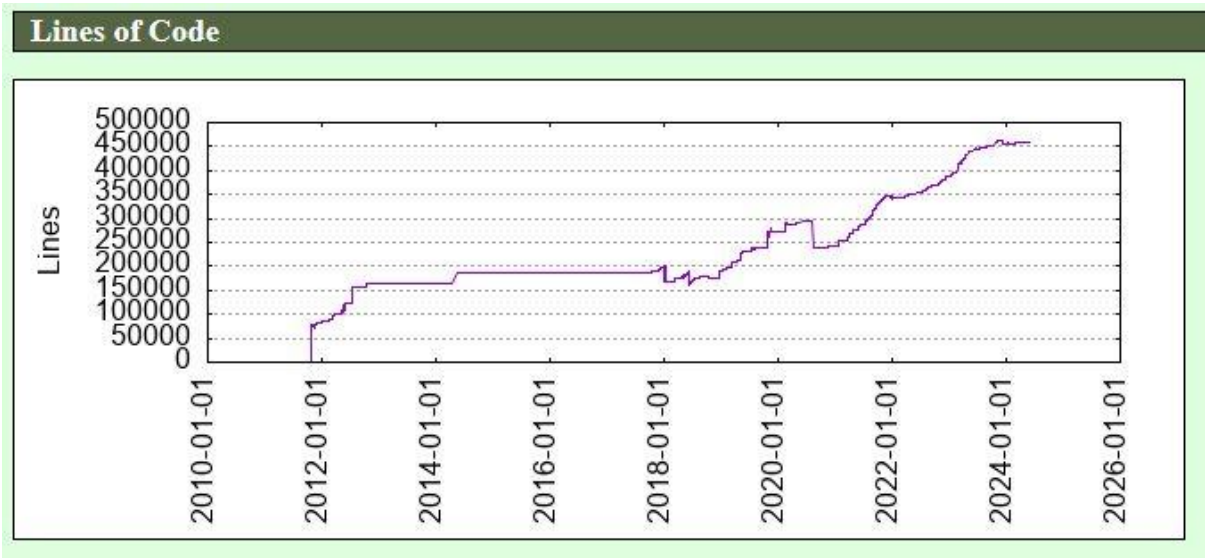


Рисунок 4.15 – Статистика росту рядків коду у проекті Dubbo

Проект розвивається довгостроково з періодами високої активності, має велику кількість учасників, але основну роботу виконують найактивніші з них. В останні роки активність значно зросла, що може свідчити про нові цілі чи

відродження інтересу до проекту. Dubbo демонструє тривалий та стабільний розвиток із кількома етапами різкого зростання активності (2012-2013 та 2019-2022 роки). Збільшення кількості файлів і рядків коду вказує на впровадження нових функцій і масштабування проекту. В останні роки проект досяг стабільного рівня, з мінімальними різкими змінами.

Розглянемо загальну інформацію про репозиторій:

Репозиторій Dubbo демонструє високий рівень активності та популярності серед розробників і користувачів. Загальна кількість комітів становить 8223, що свідчить про активну підтримку та розвиток проєкту. Кількість пул-реквестів – 7604, що майже дорівнює кількості комітів і підтверджує структурований підхід до внесення змін через процес рев'ю. Велика кількість задач, а саме 14683, свідчить про активний фідбек від спільноти та регулярну взаємодію з користувачами. У проекті бере участь 403 контриб'ютори, що вказує на широку спільноту розробників. Популярність проекту підтверджується високою кількістю зірок (40562) та форків (26441), що вказує на його використання для створення власних рішень.

Репозиторій Dubbo демонструє високу активність спільноти та розробників, маючи найбільшу кількість задач та пулл-реквестів. Високий рівень закритих завдань та змін свідчить про ефективну роботу команди, тоді як наявність відкритих завдань показує, що проект активно розвивається. Процеси внесення змін добре налагоджені: більшість пулл-реквестів закриваються швидко з мінімальним обговоренням. Важливі або складні зміни отримують більше уваги та коментарів, підтверджуючи відкриту комунікацію в команді. Великі оновлення проходять ретельнішу перевірку, що свідчить про збалансований підхід до якості та розвитку функціоналу. Додаткові дослідження активності проекту, відношення додавань та видалень у пулл-реквестах, відношення загальної кількості змін у коді до кількості коментарів від розробників, відношення часу, який пулл-реквест був відкритим до кількості коментарів від розробників в пулл-реквесті, та відношення часу, який пулл-реквест був відкритим до загальної кількості змін у коді в пулл-реквестах

приведені нижче (див. Додаток В).

Аналіз графіка(4.16) активності показує, що сплески кількості комітів, пул-реквестів та задач відповідають періодам розробки нових функцій, тестування або релізів. Такі тенденції свідчать про динамічний розвиток та регулярне оновлення проекту. Зниження активності у певні періоди можна пов’язати з фазами стабілізації, коли основна увага приділяється усуненню помилок і вдосконаленню функціоналу. Загалом, активний процес вирішення задач через пул-реквести забезпечує високу якість коду, а значна кількість контриб'юторів підтримує безперервний розвиток.

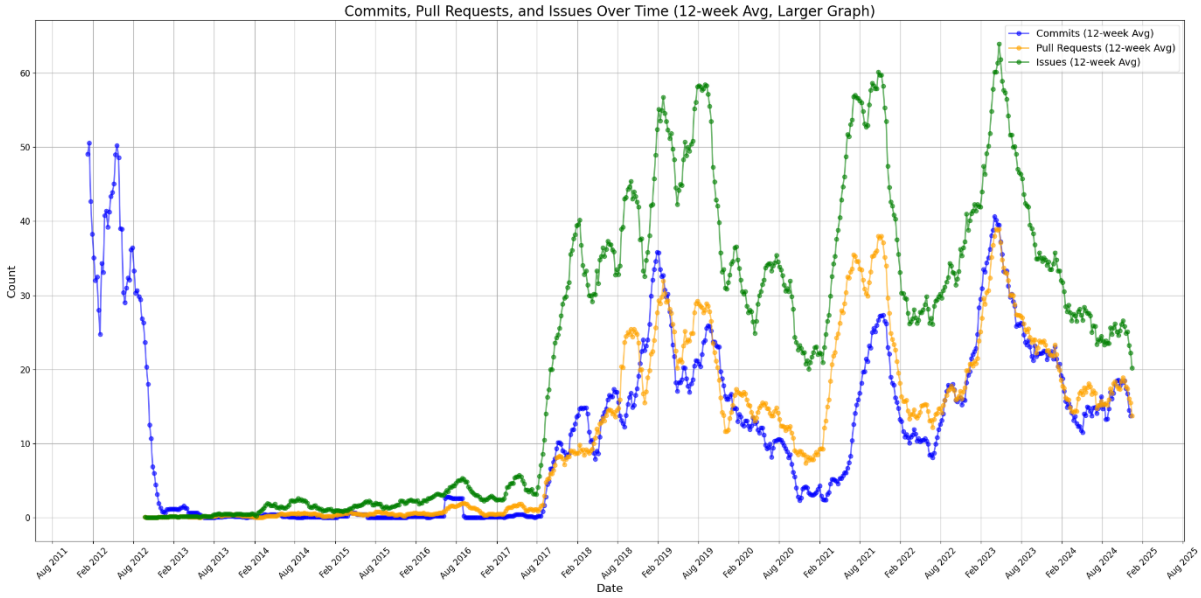


Рисунок 4.16 – Відношення коммітів/пулл-реквестів/задач репозиторію Dubbo

Проект демонструє високий рівень організованості та відкритості до зовнішніх внесків, що сприяє його популярності й сталому розвитку. Щоб зберегти цей рівень, важливо підтримувати взаємодію зі спільнотою, аналізувати ключові періоди активності та забезпечувати швидке вирішення задач для підвищення задоволеності користувачів. Dubbo є прикладом успішного проекту з відкритим кодом, який активно розвивається та відповідає потребам сучасних розробників.

4.2.2 Google Guava

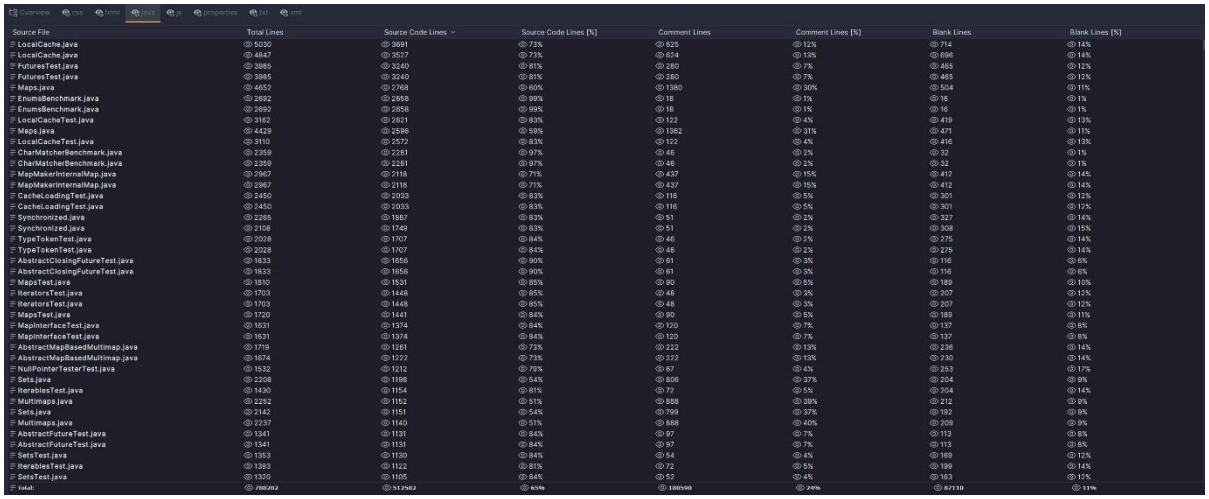
Guava — це набір бібліотек для Java, створений компанією Google для вирішення поширених завдань програмування, таких як робота з колекціями,

кешування, обробка рядків і базові утиліти. Проект розпочався як внутрішній інструмент Google, але в 2010 році став відкритим. Завдяки своїй універсальності та високій якості швидко здобув популярність у Java-спільноті.

Структурна характеристика проекту:

Guava має невелику кількість модулів, але доволі прозору структуру пакетів, класи легко шукати за їх логічним призначенням та назвою. Рівень документації трохи нижчий ніж у Dubbo, але вважається задовільним. Проект слідує стилістичним принципам «Google's Java style guide», що є дуже добре. Рівень задокументованості коду дуже високий, але все одно присутні непотрібні пояснювальні коментарі, деякі елементи документації є також не дуже потрібними. Іноді зустрічаються заглушені порушення коду, що є не дуже гарним знаком, але їх кількість доволі низька. У коді зустрічаються багато «TODO» це теж не є гарним показником. Також у проекті присутні багато дерикторій, ціль та організація яких не дуже зрозуміла. Загалом Guava має хороший рівень коду, але потребує рафакторінгу, очистки та покращень.

Роздивимось загальну статистику рядків коду у проекті(рис 4.17):



Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
LocalCache.java	5030	3691	73%	625	12%	714	14%
LocalCache.java	4847	3527	73%	624	13%	696	14%
FuturesTest.java	3985	3240	81%	280	7%	465	12%
FuturesTest.java	3985	3240	81%	280	7%	465	12%
Maps.java	4652	2768	60%	1380	30%	504	11%
EnumsBenchmark.java	2692	2658	99%	18	1%	16	1%
EnumsBenchmark.java	2692	2658	99%	18	1%	16	1%
LocalCacheTest.java	3162	2621	83%	122	4%	419	13%
Maps.java	4429	2596	59%	1382	31%	471	11%
LocalCacheTest.java	3110	2572	83%	122	4%	416	13%
CharMatcherBenchmark.java	2359	2281	97%	46	2%	32	1%
CharMatcherBenchmark.java	2359	2281	97%	46	2%	32	1%
MapMakerInternalMap.java	2967	2118	71%	437	15%	412	14%
MapMakerInternalMap.java	2967	2118	71%	437	15%	412	14%
CacheLoadingTest.java	2450	2033	83%	116	5%	301	12%
CacheLoadingTest.java	2450	2033	83%	116	5%	301	12%
Synchronized.java	2265	1867	83%	51	2%	327	14%
Synchronized.java	2108	1749	83%	51	2%	308	15%
TypeTokenTest.java	3028	1707	84%	46	2%	275	14%
TypeTokenTest.java	2028	1707	84%	46	2%	275	14%
AbstractClosingFutureTest.java	1833	1656	90%	61	3%	116	6%
AbstractClosingFutureTest.java	1833	1656	90%	61	3%	116	6%
MapsTest.java	1810	1531	85%	90	5%	189	10%
IteratorsTest.java	1703	1448	85%	48	3%	207	12%
IteratorsTest.java	1703	1448	85%	48	3%	207	12%
MapsTest.java	1720	1441	84%	90	5%	189	11%
MapInterfaceTest.java	1631	1374	84%	120	7%	137	8%
MapInterfaceTest.java	1631	1374	84%	120	7%	137	8%
AbstractMapBasedMultimap.java	1719	1281	73%	222	13%	236	14%
AbstractMapBasedMultimap.java	1674	1222	73%	222	13%	230	14%
NullPointerTesterTest.java	1532	1212	79%	67	4%	253	17%
Sets.java	2208	1186	54%	806	37%	204	9%
IterablesTest.java	1430	1154	81%	72	5%	204	14%
Multimaps.java	2292	1152	51%	888	39%	212	9%
Sets.java	2142	1151	54%	799	37%	192	9%
Multimaps.java	2237	1140	51%	868	40%	209	9%
AbstractFutureTest.java	1341	1131	84%	97	7%	113	8%
AbstractFutureTest.java	1341	1131	84%	97	7%	113	8%
SetsTest.java	1353	1130	84%	54	4%	169	12%
IterablesTest.java	1393	1122	81%	72	5%	199	14%
SetsTest.java	1320	1105	84%	52	4%	163	12%
Total:	288282	512582	65%	188598	24%	87110	11%

Рисунок 4.17 – Загальна статистика проекту Guava

Рядки вихідного коду становлять близько 65% від загальної кількості рядків. Це свідчить про те, що значний обсяг проекту зосереджено на функціональності, але також приділено увагу тестам та документації. Guava є більш масштабним проектом за кількістю рядків, і це логічно, враховуючи широту її утиліт для різних задач у Java.

Проведемо сканування проекту за допомогою інструментів статичного аналізу (рис. 4.18 — 4.25):

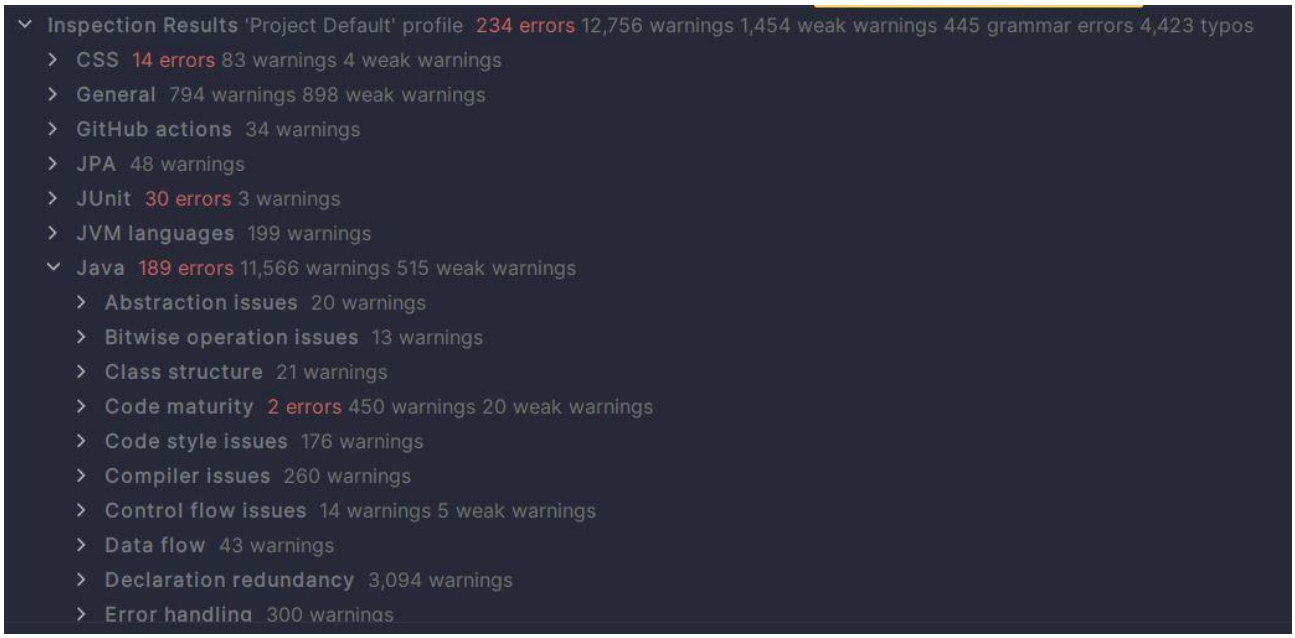


Рисунок 4.18 – Результати сканування проекту Guava з InteliJ

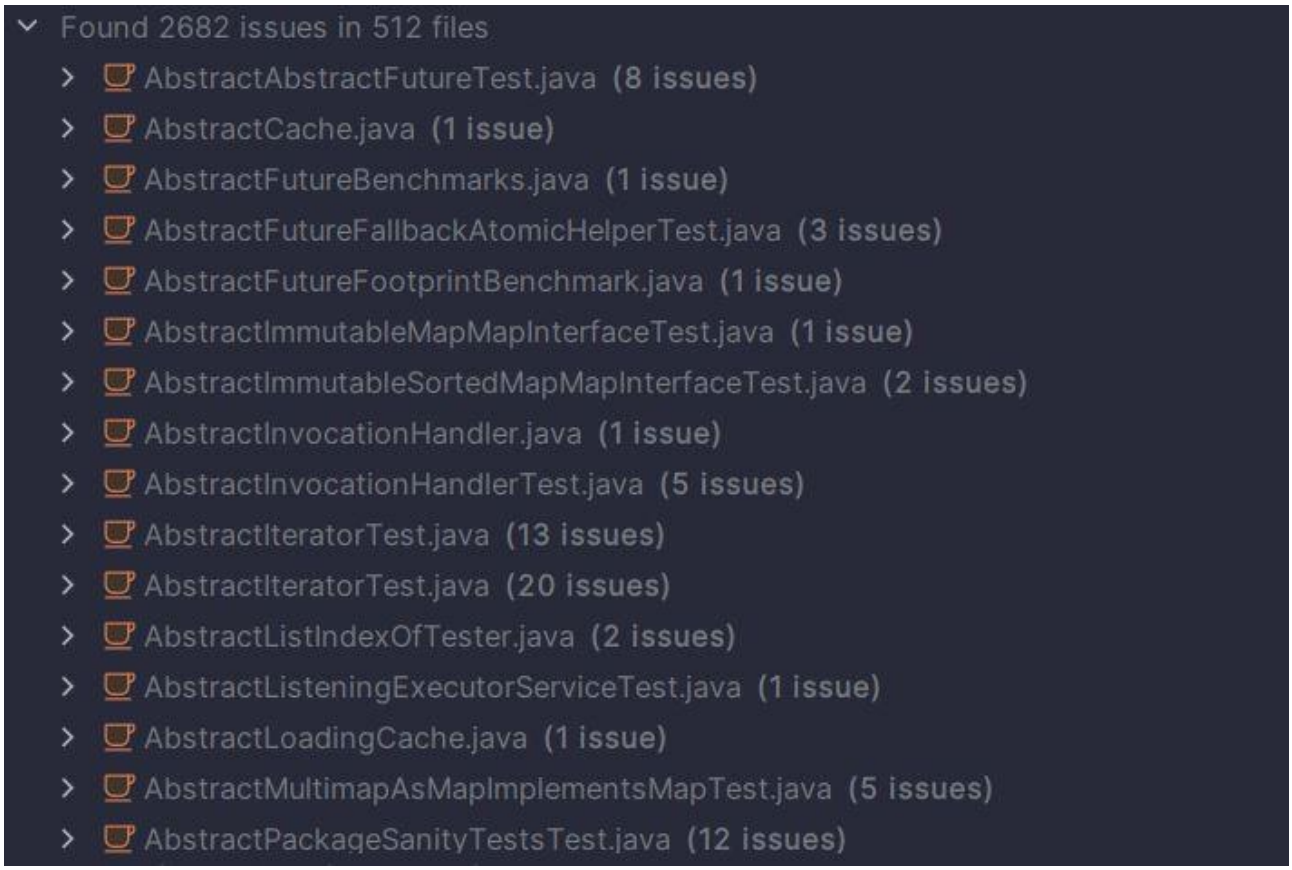


Рисунок 4.19 – Результати сканування проекту Guava з SonarLint

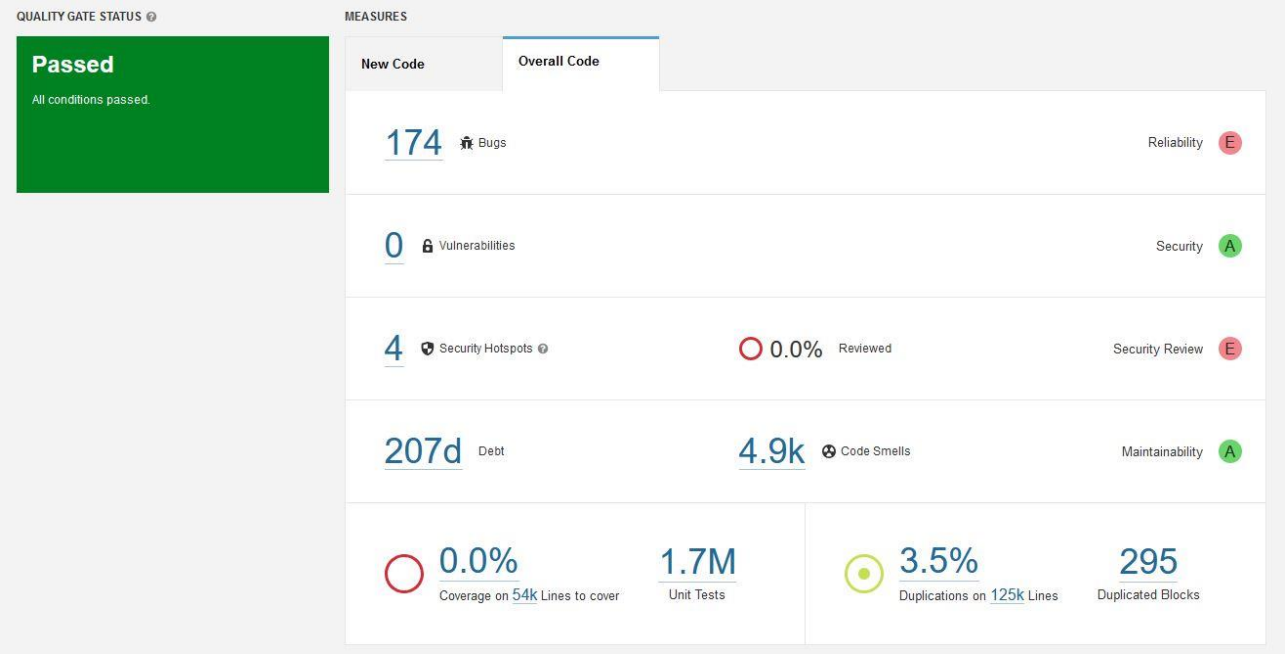


Рисунок 4.20 – Результати сканування проекту Guava з SonarQube

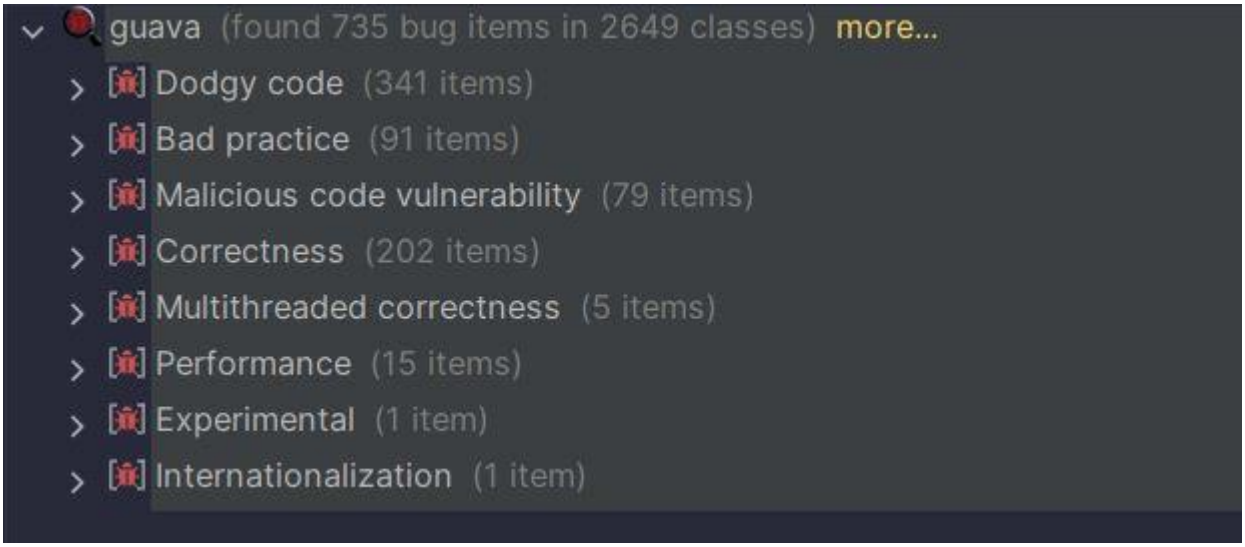


Рисунок 4.21 – Результати сканування проекту Guava з SpotBugs

Рисунок 4.22 – Результати сканування проекту Guava з Qodana

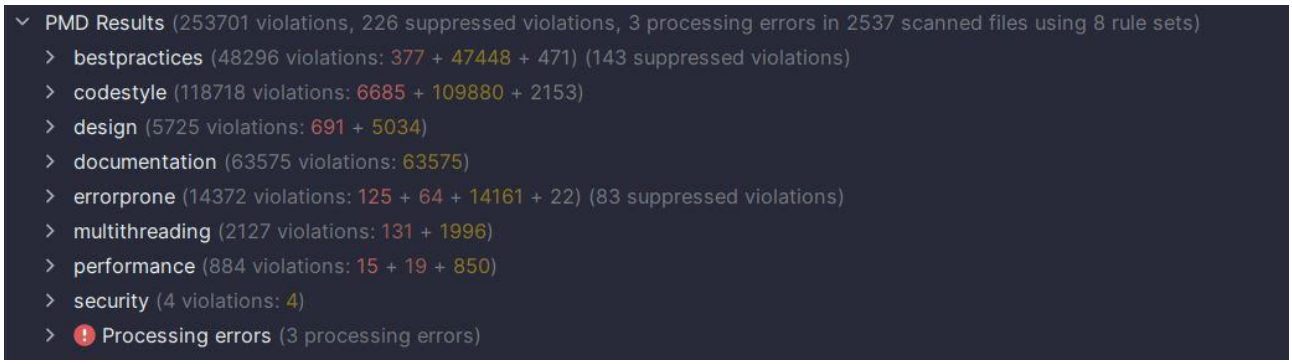


Рисунок 4.23 – Результати сканування проекту Guava з PMD

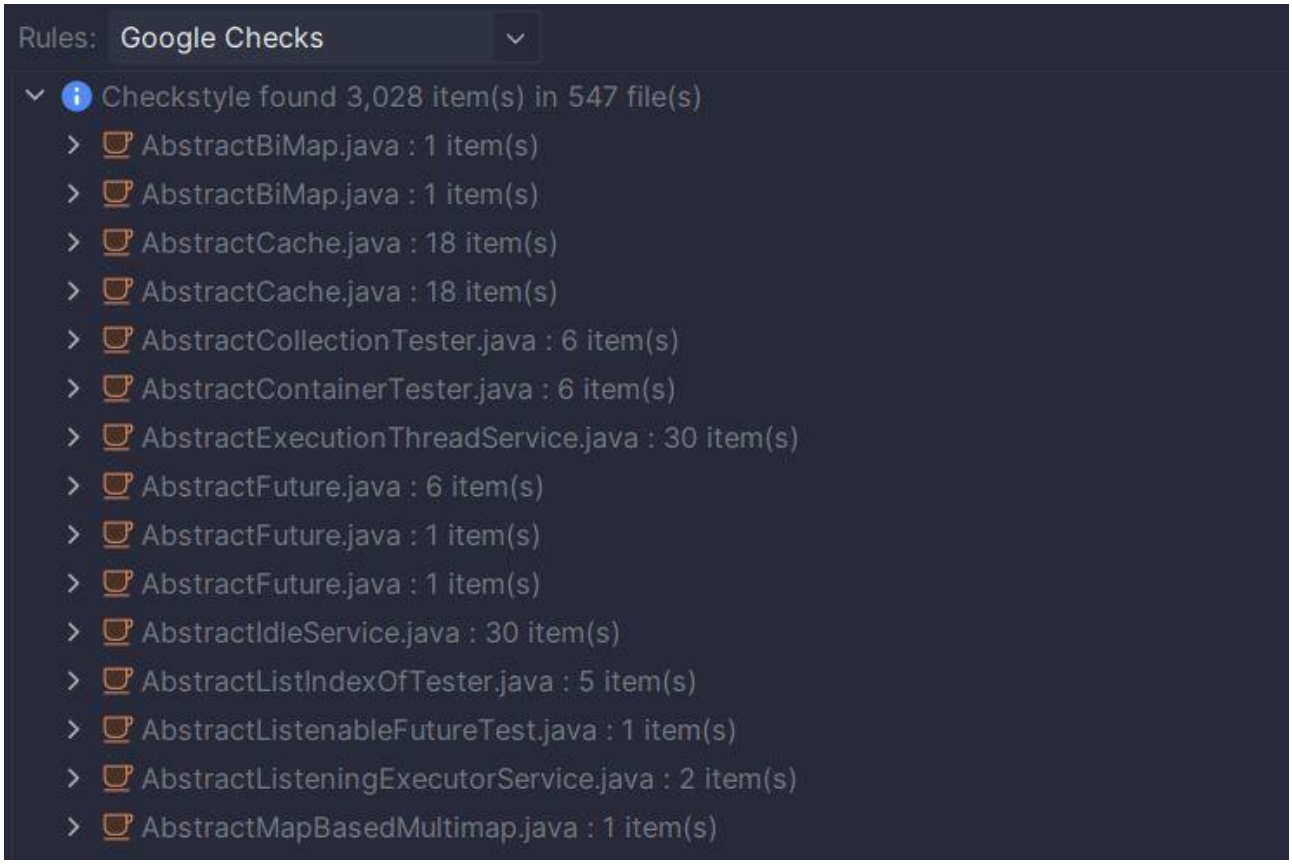


Рисунок 4.24 – Результати сканування проекту Guava з CheclStyle та посібником зі стилю від Sun

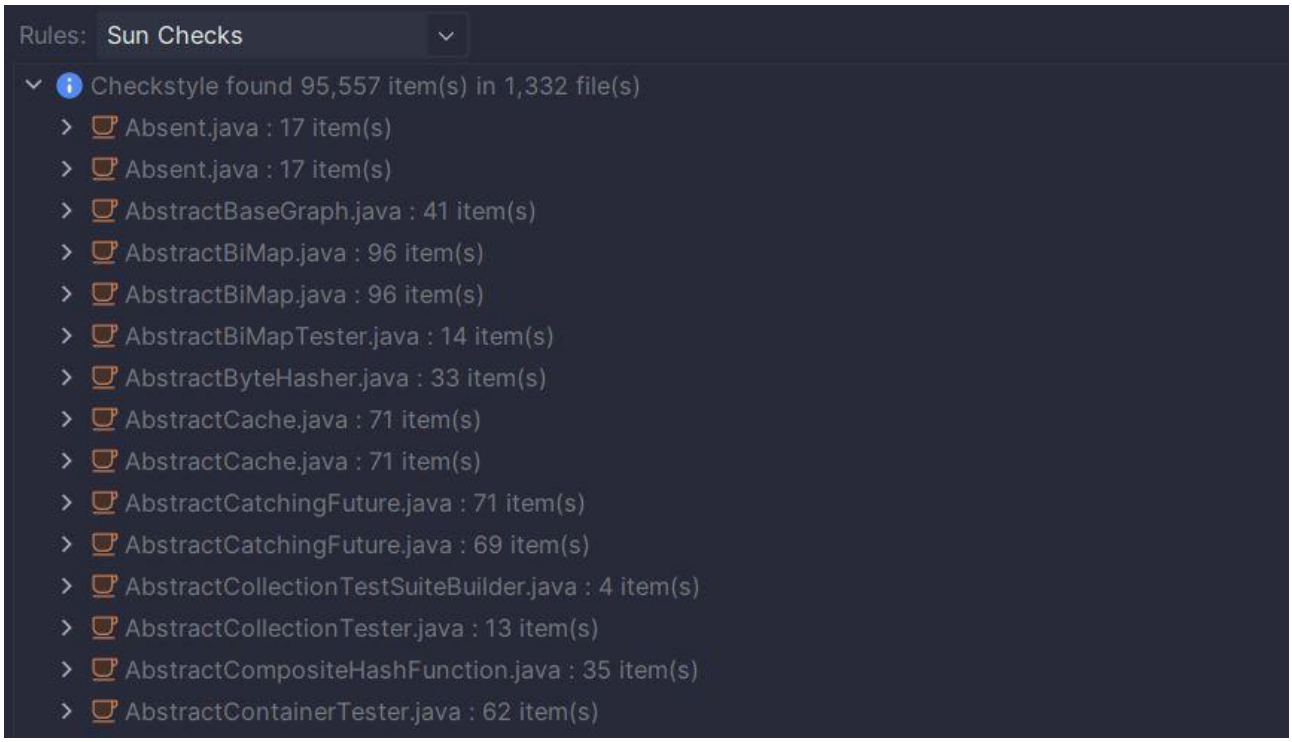


Рисунок 4.25 – Результати сканування проекту Guava з CheclStyle та посібником зі стилю від Google

Найвищу кількість помилок на рядок коду демонструє PMD, із

коефіцієнтом 0.49494715. Це свідчить про те, що цей аналізатор виявляє найбільше проблем серед усіх перевірених інструментів. Інші аналізатори, такі як Check Style Sun (0.186422855) і Check Style Google (0.005907348), мають суттєво менші показники, що свідчить про їх орієнтованість на перевірку стилістичних аспектів коду. IntelliJ IDEA, SonarLint, SonarQube, SpotBugs і Qodana виявляють помилки на рядок на значно нижчому рівні, з коефіцієнтами від 0.0014 до 0.028. Використання різних статичних аналізаторів дає змогу покрити різні аспекти якості коду, тому варто продовжити комбінований підхід.

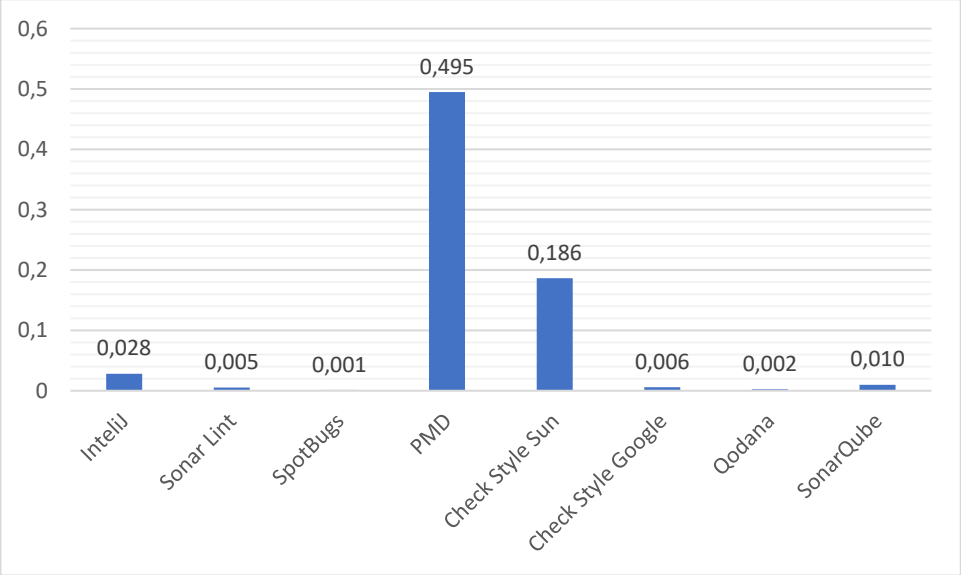


Рисунок 4.26 – Кількість помилок на рядок коду Guava

Базуючись на даних від Statistic виберемо десять найбільших файлів та проведемо сканування кожного з них(табл. 4.2) та побудуємо графік порівняння кількості помилок від різних статичних аналізаторів(рис. 4.26):

Таблиця 4.2 – Результати сканування десяти найбільших файлів проекту Guava

Назва файлу	Кількість рядків вихідного коду	InteliJ	Sonar Lint	Spot Bugs	PMD	Check Style Sun	Check Style Google
com.google.common.cache.LocalCache	3691	369	181	108	1710	1222	1
com.google.common.collect.Maps	2768	223	136	7	1420	1522	10

com.google.common.collect.MapMakerInternalMap	2118	172	82	38	1049	857	4
com.google.common.collect.Synchronized	1887	78	28	1	766	408	1
com.google.common.collect.AbstractMapBasedMultimap	1261	66	20	4	439	283	1
com.google.common.collect.Sets	1198	106	55	8	673	561	1
com.google.common.collect.Multimaps	1152	138	56	7	637	772	12
com.google.common.base.CharMatcher	1036	127	20	1	491	405	16
com.google.common.util.concurrent.ClosingFuture	987	95	53	6	796	890	15
com.google.common.util.concurrent.AbstractFuture	959	145	60	5	619	479	1

Найбільша кількість помилок виявлена у файлі com.google.common.cache.LocalCache, який має 3691 рядок коду і найбільший сумарний показник помилок серед усіх аналізаторів. Цей файл є найвразливішим з точки зору потенційних проблем і потребує додаткового уваги для рефакторингу або оптимізації. Другим за кількістю проблем є файл com.google.common.collect.Maps із 2768 рядками коду. У ньому також виявлено велику кількість помилок за результатами PMD та Check Style Sun. Файли з меншою кількістю рядків коду, такі як com.google.common.util.concurrent.AbstractFuture, мають менший показник помилок, але їхній вплив на загальну якість проекту може бути значним через критичність функціоналу.

На графіку (рис. 2.27) видно, що найбільше помилок виявляється в найбільших файлах, таких як LocalCache та Maps. Це свідчить про те, що складність і розмір файлу корелюють із кількістю знайдених проблем.

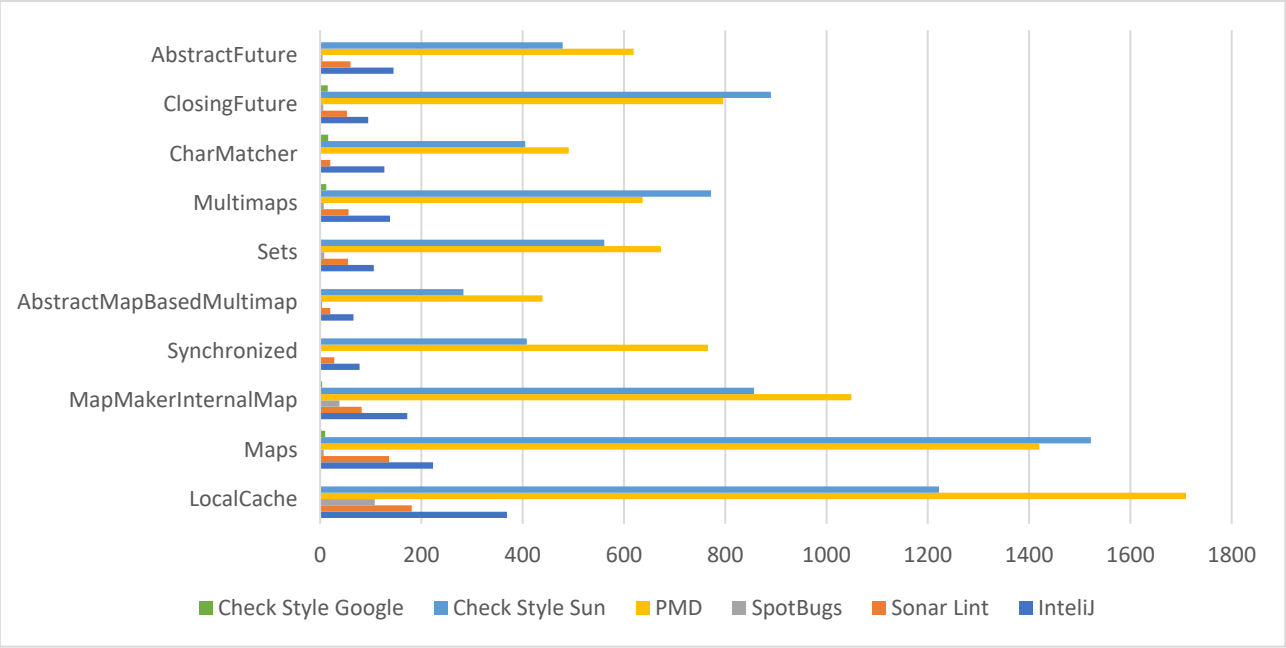


Рисунок 4.27 – Порівняння результатів сканування десяти найбільших файлів проекту Guava

Необхідно сфокусувати увагу на рефакторингу файлів із найбільшою кількістю проблем, таких як LocalCache і Maps. Менші файли також потребують перевірки, оскільки окремі критичні помилки можуть значно вплинути на стабільність і функціональність проекту.

Розглянемо Git статику:

Проект Guava демонструє довготривалу історію розвитку, що охоплює 5472 дні, з яких 2350 днів були активними (42.95%). У репозиторії зберігається 3314 файлів із загальною кількістю рядків коду 1,038,542, що включає 731,336 видалених рядків і 176,987 доданих. Це свідчить про значний обсяг роботи, спрямованої на оновлення та оптимізацію коду. Кількість комітів становить 6502, що в середньому дорівнює 2.8 комітів за активний день і 1.2 комітів за всі дні. Така частота оновлень свідчить про регулярність і сталість роботи над проектом. У Guava брали участь 474 автори, що в середньому складає 13.7 комітів на одного контриб'ютора, що вказує на значний внесок спільноти.

Графік(рис. 4.28) демонструє, що найвища інтенсивність комітів спостерігалася в період між 2010 та 2013 роками. У ці роки було досягнуто максимуму в частоті оновлень, що вказує на активний розвиток функціоналу та суттєві зміни в структурі проекту. Після 2013 року активність почала поступово

знижуватися, що може бути пов’язано із завершенням основної стадії розробки або переходом до підтримки стабільної версії.

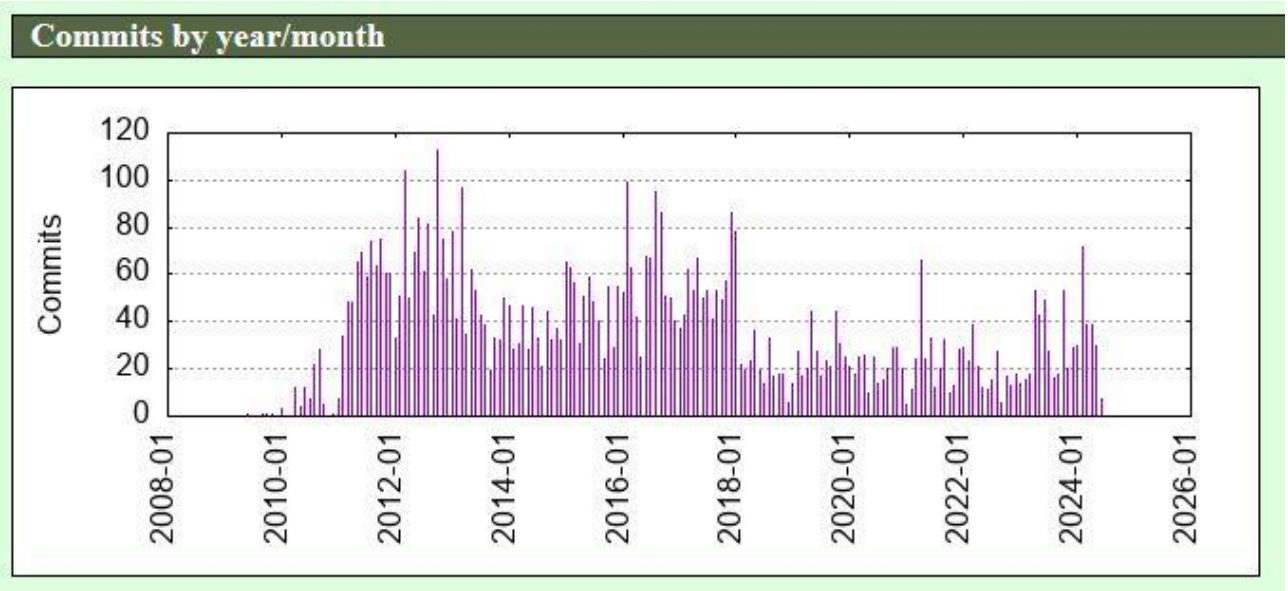
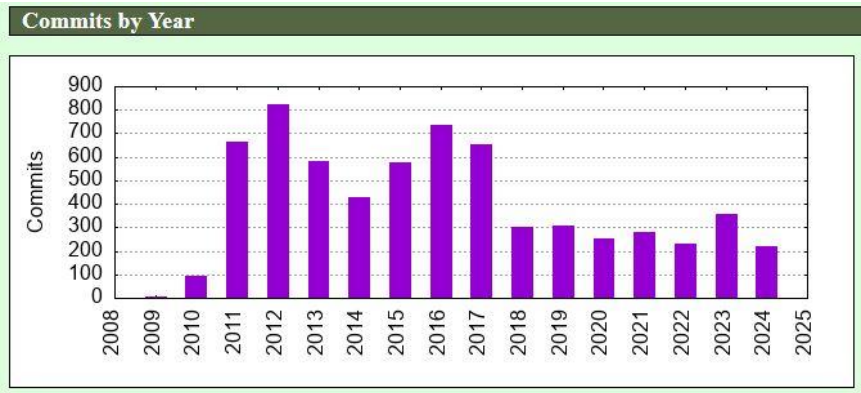


Рисунок 4.28 – Статистика коммітів за місяцями у проекті Guava

Графік (рис. 4.29) підтверджує цю тенденцію: у 2012 році було зроблено понад 822 коміта, що становить 13% від усіх комітів проєкту. У наступні роки кількість комітів стабільно знижувалася. Наприклад, у 2019–2024 роках комітів було в середньому менше 300 на рік, що свідчить про перехід від активної розробки до етапу стабілізації.



Year	Commits (% of all)	Lines added	Lines removed
2024	217 (3.34%)	10266	8285
2023	355 (5.46%)	28257	24812
2022	231 (3.55%)	27699	17910
2021	278 (4.28%)	35264	17153
2020	252 (3.88%)	28907	50583
2019	310 (4.77%)	21238	14281
2018	303 (4.66%)	27935	16659
2017	651 (10.01%)	604389	104225
2016	738 (11.35%)	75320	64084
2015	576 (8.86%)	72036	106162
2014	426 (6.55%)	26260	15555
2013	582 (8.95%)	127253	51853
2012	822 (12.64%)	259188	172607
2011	663 (10.20%)	317327	55129
2010	94 (1.45%)	94659	12020
2009	4 (0.06%)	13880	18

Рисунок 4.29 – Статистика коммітів за роками у проекті Guava

Проект Guava демонструє стабільний ріст кількості файлів та рядків коду. На графіку (рис. 4.30) помітно, що основний приріст відбувався в період з 2010 по 2018 рік. У цей час кількість файлів суттєво зросла, що свідчить про

активний розвиток і масштабування проекту. Після 2018 року ріст сповільнився, а кількість файлів стабілізувалася на рівні приблизно 3300, що вказує на завершення активної фази розробки та перехід до підтримки.

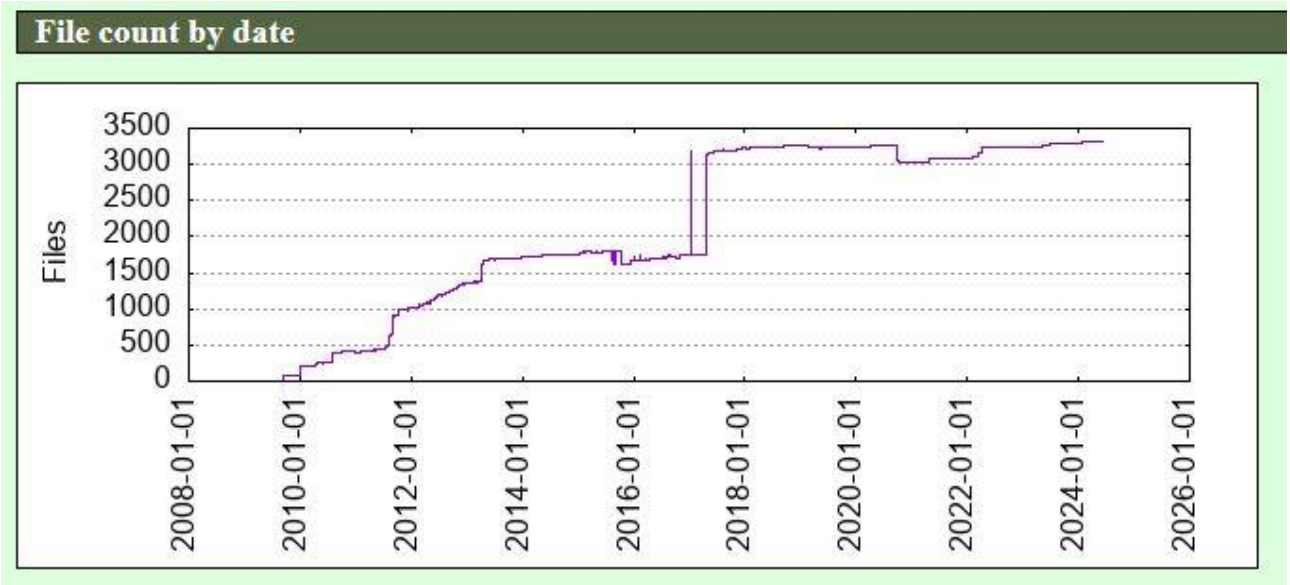


Рисунок 4.30 – Статистика кількості файлів у проекті Guava

На графіку (рис. 4.31) видно подібну динаміку. Обсяг коду інтенсивно зростав у період до 2018 року, після чого збільшення стало мінімальним, і проект стабілізувався на рівні понад 1.2 мільйона рядків коду. Це може бути наслідком оптимізації існуючого коду та зменшення додавання нових функцій.

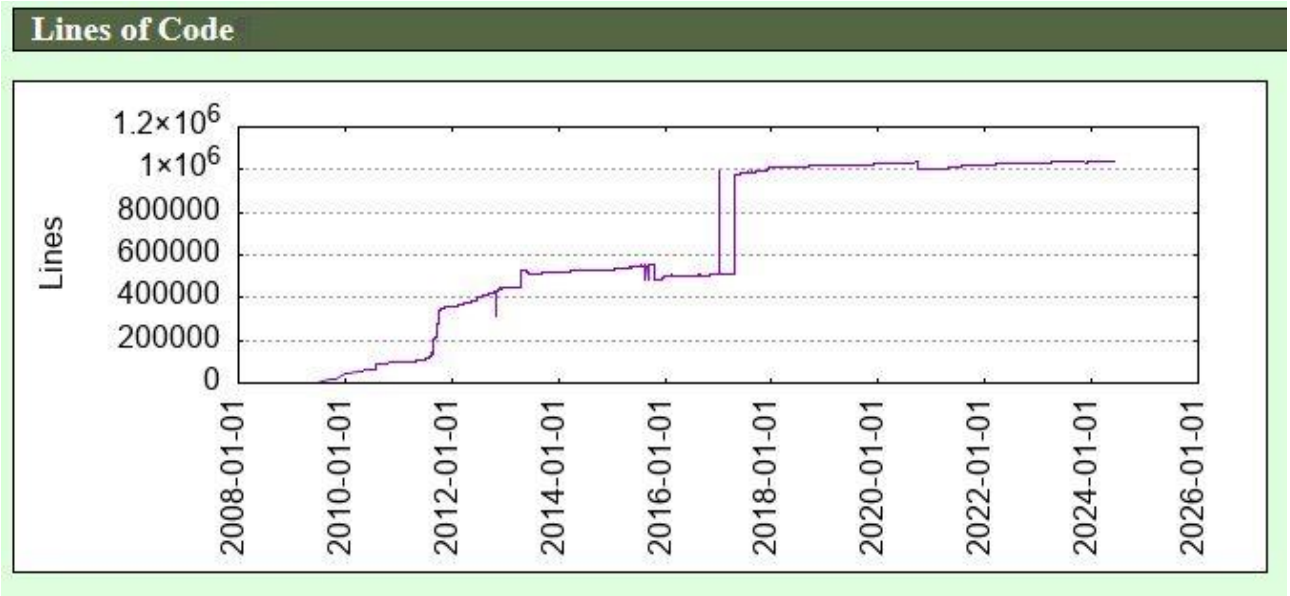


Рисунок 4.31 – Статистика росту рядків коду у проекті Guava

Таким чином, аналіз цих графіків показує, що проект Guava пройшов класичні етапи розвитку: активне розширення в перші роки, досягнення зрілості

та стабілізації після 2017 року.

Розглянемо загальну інформацію про репозиторій:

Репозиторій Guava демонструє помірну активність із загальною кількістю коммітів 6696, пул-реквестів 2538 і відкритих задач (issues) — 6128. Число учасників проекту становить 317, що є доволі значною спільнотою, яка сприяє розвитку проєкту. Високий рівень популярності репозиторію підтверджується 50306 зірками та 10918 форками. Значна кількість учасників (317) і популярність проекту (зірки та форки) свідчить про довготривалий інтерес та використання Guava у спільноті розробників. Це підтверджує значущість проекту для екосистеми Java.

Guava демонструє стабільну підтримку та розвиток із високим рівнем закритих задач та пулл-реквестів, що свідчить про ефективну роботу команди. Відносно мала кількість відкритих задач вказує на швидке вирішення проблем і стабільність проекту. Більшість змін є невеликими та середніми, із швидким процесом рев'ю, що свідчить про добре налагоджені робочі процеси. Великі та складні пулл-реквести трапляються рідко й потребують додаткової уваги, вказуючи на відповідальний підхід до інтеграції критичних оновлень. Додаткові дослідження активності проекту, відношення додавань та видалень у пулл-реквестах, відношення загальної кількості змін у коді до кількості коментарів від розробників, відношення часу, який пулл-реквест був відкритим до кількості коментарів від розробників в пулл-реквесті, та відношення часу, який пулл-реквест був відкритим до загальної кількості змін у коді в пулл-реквестах приведені нижче (див. Додаток В).

На графіку (рис. 4.32), що ілюструє співвідношення коммітів, пул-реквестів і задач у часовому вимірі, видно, що пік активності припадає на початкові фази розробки проекту. Найбільша кількість коммітів спостерігається у проміжку до 2015 року, після чого темп зменшився і стабілізувався. Це може свідчити про те, що більшість основного функціоналу було реалізовано на ранніх етапах, а подальший розвиток зосередився на підтримці й оптимізації. Також графік показує цікавий стрибок задач на початку 2015 року.

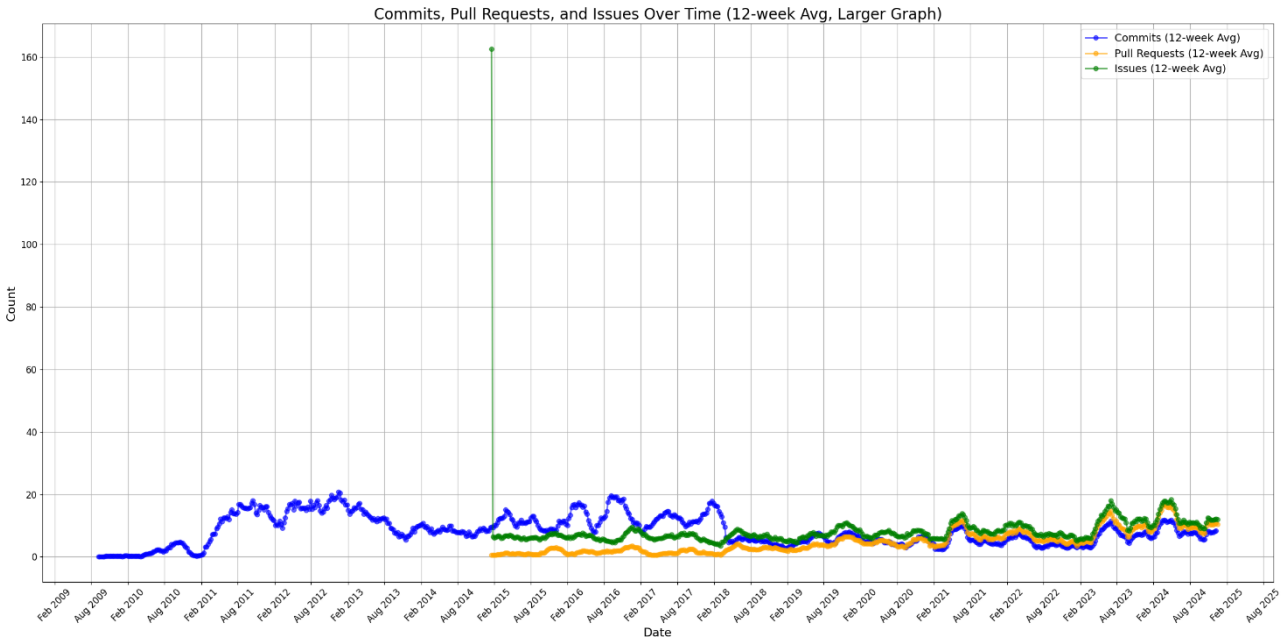


Рисунок 4.32 – Відношення коммітів/пулл-реквестів/задач репозиторію Guava

Guava є популярним і добре підтримуваним проектом із великим внеском на ранніх стадіях розвитку. Хоча активність трохи зменшилась з часом, проект все ще залишається стабільним і затребуваним серед розробників.

4.2.3 Junit5

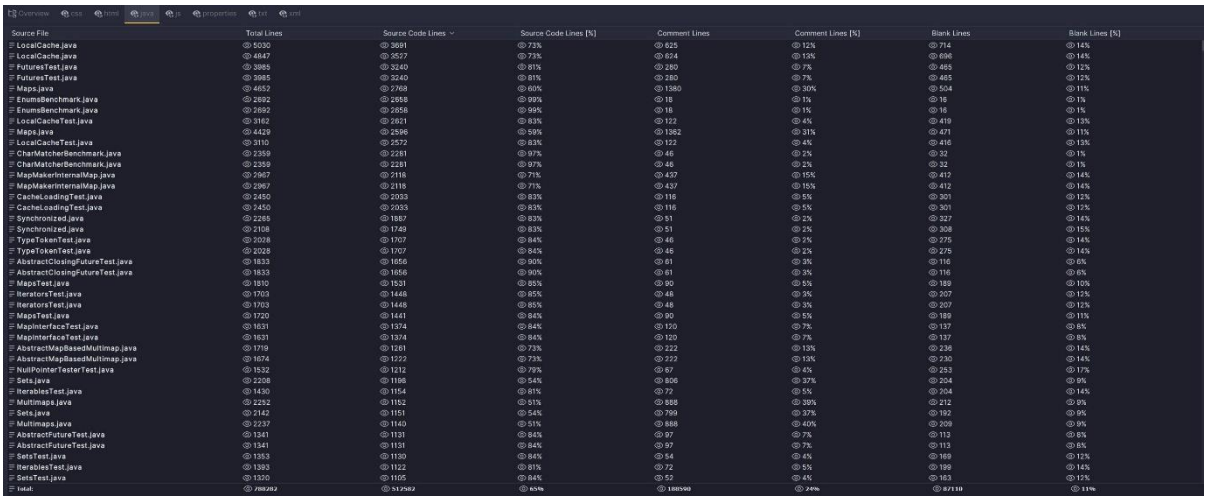
JUnit 5 — це сучасна версія популярного фреймворку для тестування в Java, створена як еволюція JUnit 4. Проект запущений у 2015 році для врахування нових можливостей Java 8 і пізніших версій, з акцентом на модульність, гнучкість і розширюваність. Завдяки покращеній архітектурі став стандартом для модульного тестування в Java.

Структурна характеристика проекту:

JUnit5 є дуже модульним проектом та має зручну та зрозумілу документацію. Всі вимоги до пулл-реквестів та додавання коду дуже добре описані. JUnit5 використовує додаткові засоби форматування коду, але не має явних вимог до стилю коду, хоча загальні положення з цієї теми описані. Код проекту має дуже високий рівень документації та має не має непотрібної документації, а рівень якості коду є дуже високим. Код дуже легко читати та розуміти. Але для розуміння структури модулів потрібні додаткові знання та розуміння принципів роботи движка. Загалом, JUnit5 є дуже гарним та добре

структурованим проектом з виском рівнем документації.

Роздивимось загальну статистику рядків коду у проекті(рис 4.33):



Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
LocalCache.java	5030	3691	73%	625	12%	714	14%
LocalCache.java	4847	3527	73%	624	13%	696	14%
FuturesTest.java	3965	3240	81%	280	7%	465	12%
FuturesTest.java	3985	3240	81%	280	7%	465	12%
Maps.java	4652	2768	60%	1380	30%	504	11%
EnumsBenchmark.java	2692	2658	99%	18	1%	16	1%
EnumsBenchmark.java	2692	2658	99%	18	1%	16	1%
LocalCacheTest.java	3162	2621	83%	122	4%	419	13%
Maps.java	4429	2596	59%	1362	31%	471	11%
LocalCacheTest.java	3110	2572	83%	122	4%	416	13%
CharMatcherBenchmark.java	2359	2281	97%	46	2%	32	1%
CharMatcherBenchmark.java	2359	2281	97%	46	2%	32	1%
MapMakerInternalMap.java	2967	2118	71%	437	15%	412	14%
MapMakerInternalMap.java	2967	2118	71%	437	15%	412	14%
CacheLoadingTest.java	2450	2033	83%	116	5%	301	12%
CacheLoadingTest.java	2450	2033	83%	116	5%	301	12%
Synchronized.java	2265	1867	83%	51	2%	327	14%
Synchronized.java	2108	1749	83%	51	2%	308	15%
TypeTokenTest.java	2028	1707	84%	46	2%	275	14%
TypeTokenTest.java	2028	1707	84%	46	2%	275	14%
AbstractClosingFutureTest.java	1833	1656	90%	61	3%	116	6%
AbstractClosingFutureTest.java	1833	1656	90%	61	3%	116	6%
MapsTest.java	1810	1531	85%	90	5%	189	10%
IteratorsTest.java	1703	1446	85%	48	3%	207	12%
IteratorsTest.java	1703	1448	85%	48	3%	207	12%
MapsTest.java	1720	1441	84%	90	5%	189	11%
MapInterfaceTest.java	1631	1374	84%	120	7%	137	8%
MapInterfaceTest.java	1631	1374	84%	120	7%	137	8%
AbstractMapBasedMultimap.java	1719	1261	73%	222	13%	236	14%
AbstractMapBasedMultimap.java	1674	1222	73%	222	13%	230	14%
NullPointerTester Test.java	1532	1212	79%	67	4%	253	17%
Sets.java	2208	1198	54%	806	37%	204	9%
IterablesTest.java	1430	1154	81%	72	5%	204	14%
Multimaps.java	2252	1162	51%	888	39%	212	9%
Sets.java	2142	1151	54%	769	37%	192	9%
Multimaps.java	2237	1140	51%	888	40%	209	9%
AbstractFutureTest.java	1341	1131	84%	97	7%	113	8%
AbstractFutureTest.java	1341	1131	84%	97	7%	113	8%
SetsTest.java	1353	1130	84%	54	4%	169	12%
IterablesTest.java	1393	1122	81%	72	5%	199	14%
SetsTest.java	1320	1105	84%	52	4%	163	12%
Total:	788282	512582	65%	188590	24%	87110	11%

Рисунок 4.33 – Загальна статистика проекту Junit5

Рядки вихідного коду становлять близько 58% від загальної кількості рядків. Це дещо нижче, ніж у Apache Dubbo (~64%) та Google Guava (~65%), що може пояснюватися великою кількістю тестів і документації, що притаманне проєктам, орієнтованим на тестування. JUnit 5 є найкомпактнішим проектом серед розглянутих раніше, однак його фокус на тестуванні пояснює вищий відсоток тестового та супровідного коду.

Проведемо сканування проекту за допомогою інструментів статичного аналізу (рис. 4.34 — 4.41):

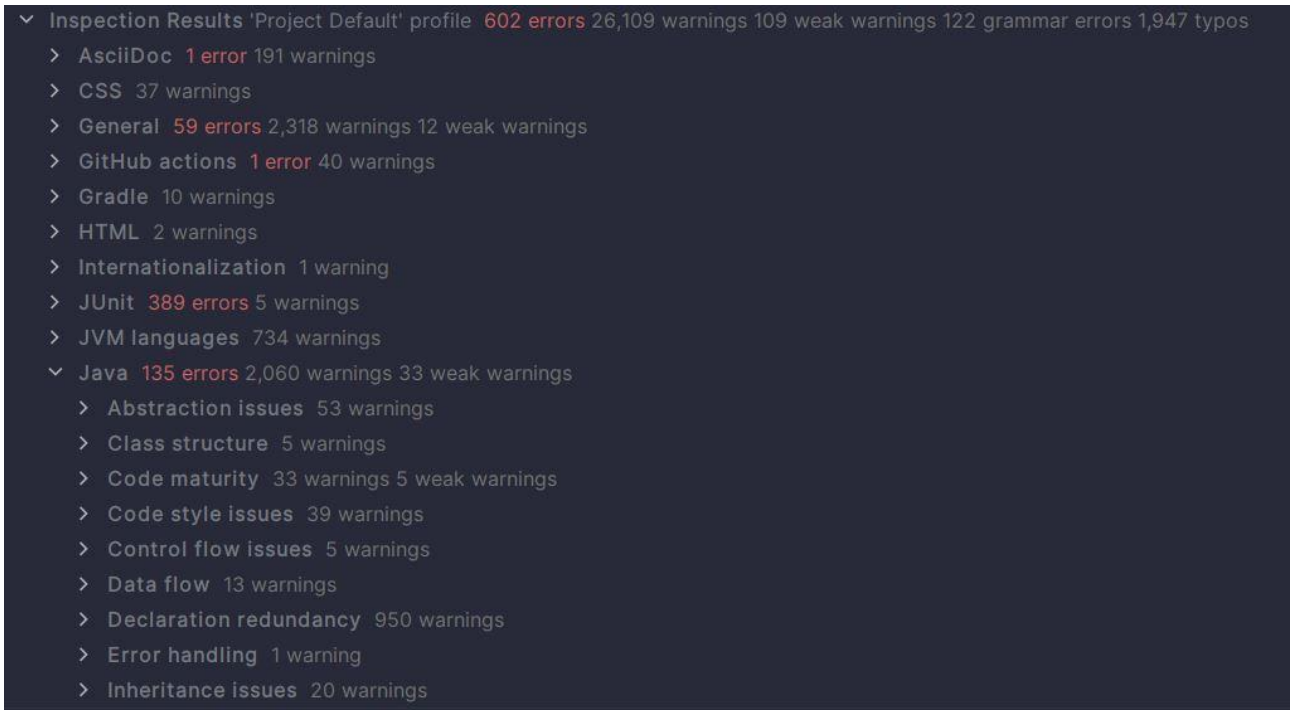


Рисунок 4.34 – Результати сканування проекту Junit5 з InteliJ

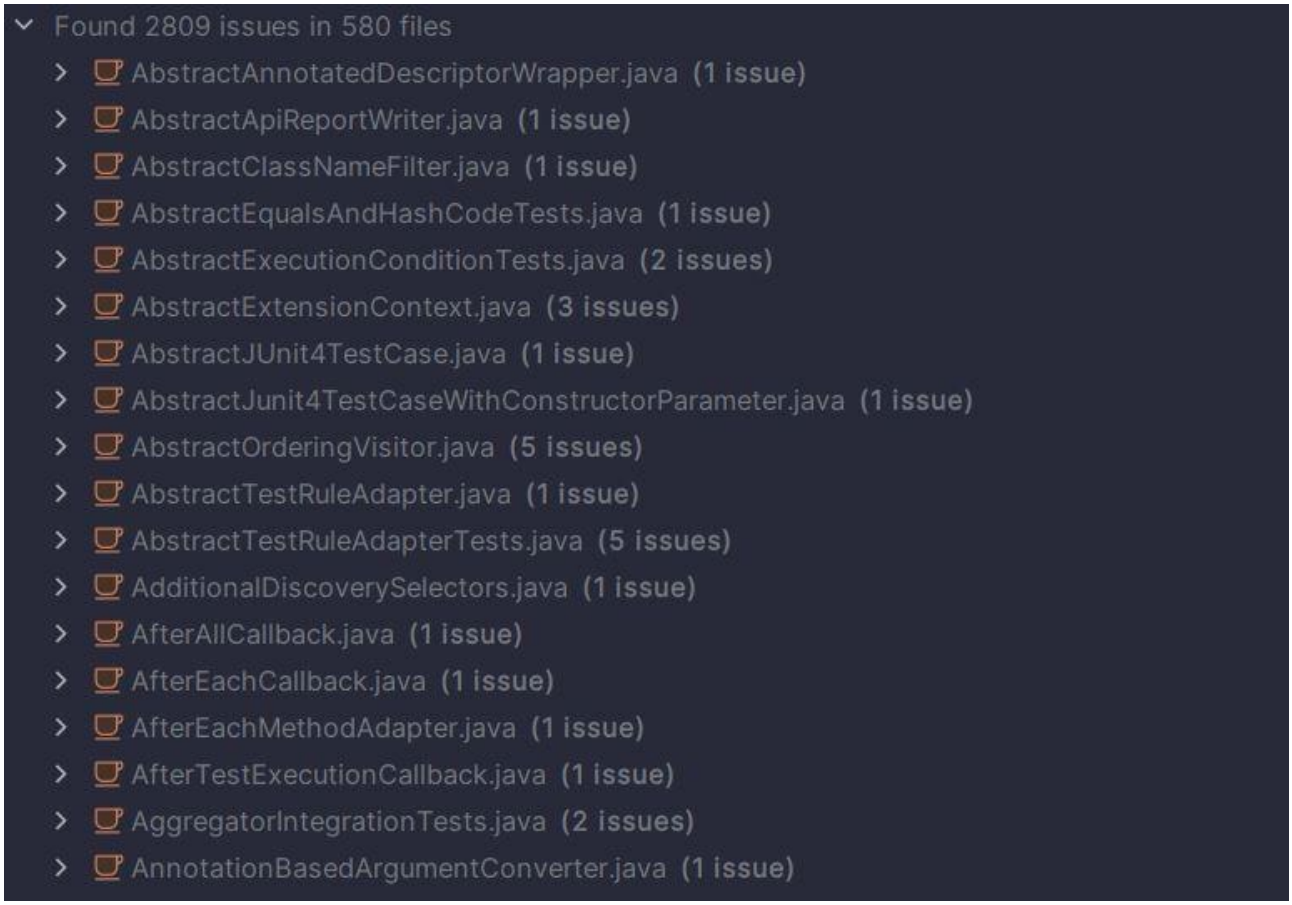


Рисунок 4.35 – Результати сканування проекту Junit5 з SonarLint

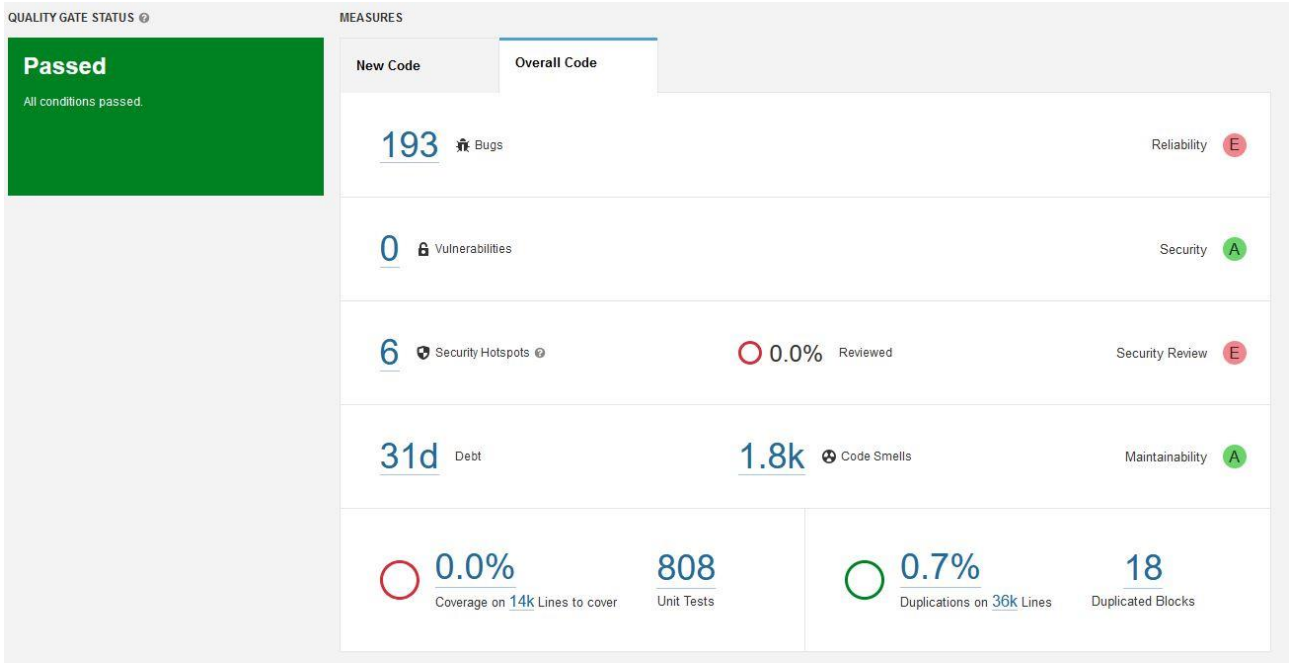


Рисунок 4.36 – Результати сканування проекту Junit5 з SonarQube

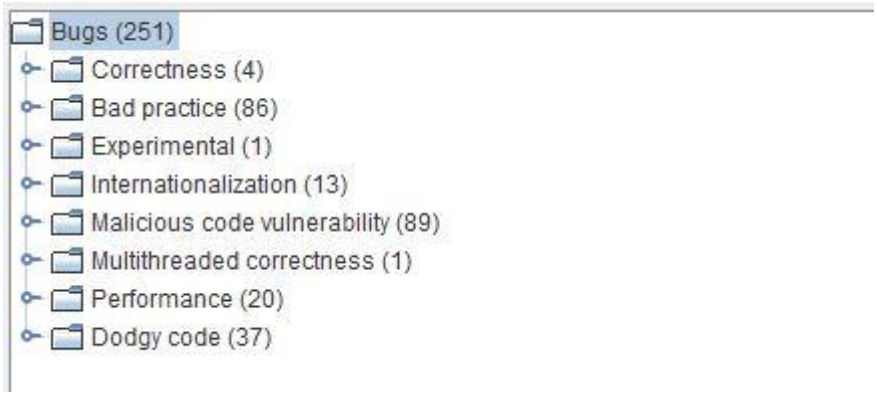


Рисунок 4.37 – Результати сканування проекту Junit5 з SpotBugs

Рисунок 4.38 – Результати сканування проекту Junit5 з Qodana

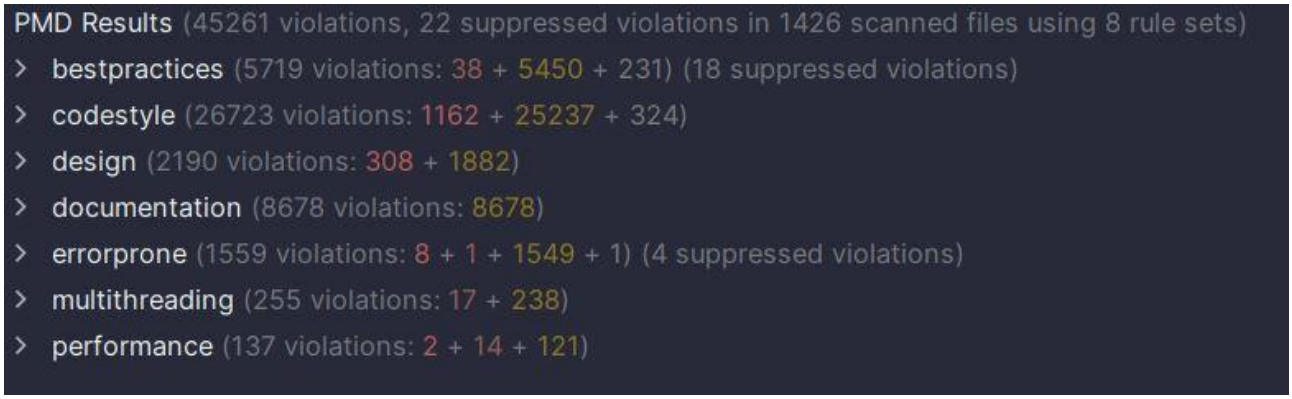


Рисунок 4.39 – Результати сканування проекту Junit5 з PMD

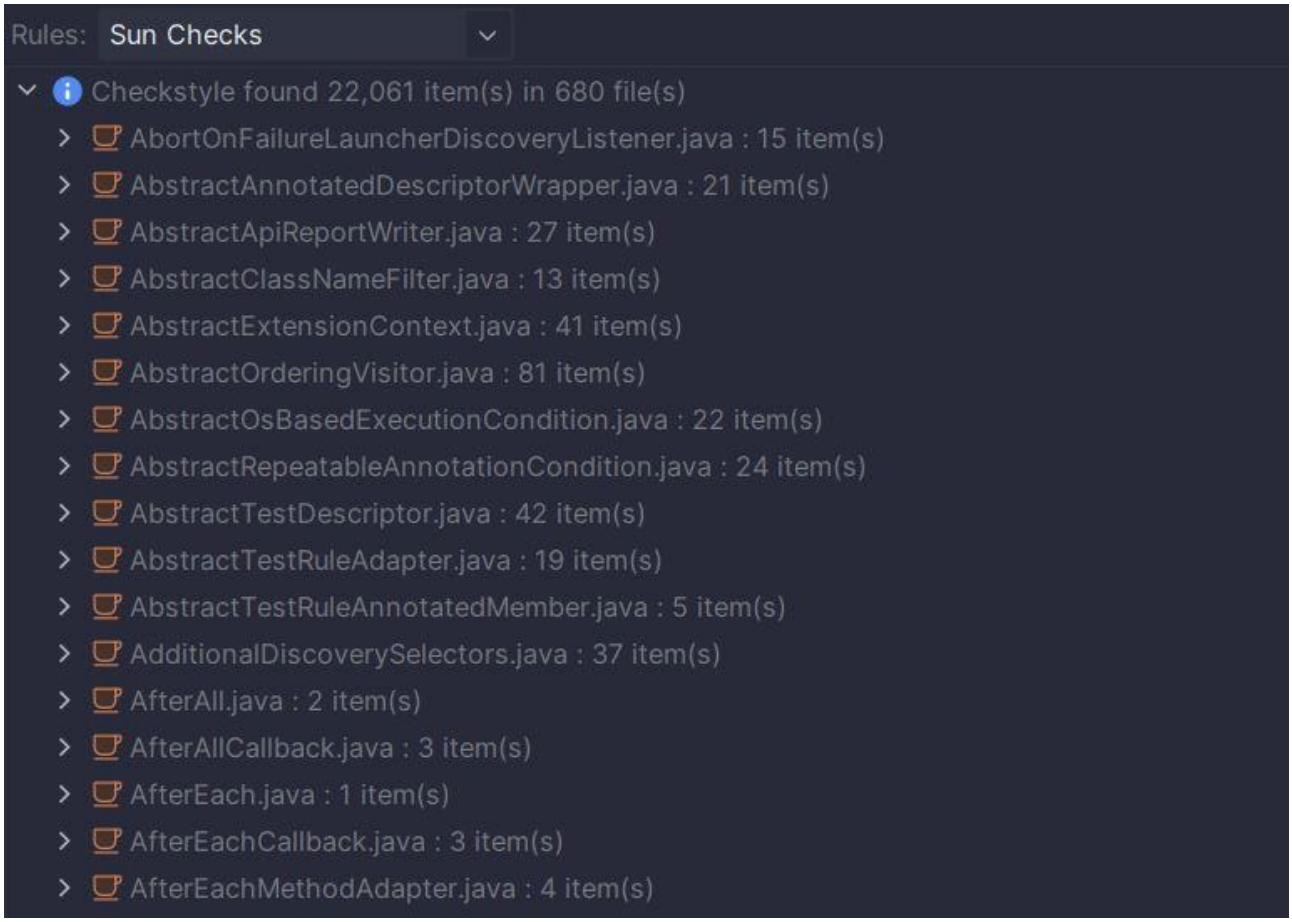


Рисунок 4.40 – Результати сканування проекту Junit5 з CheclStyle та посібником зі стилю від Sun

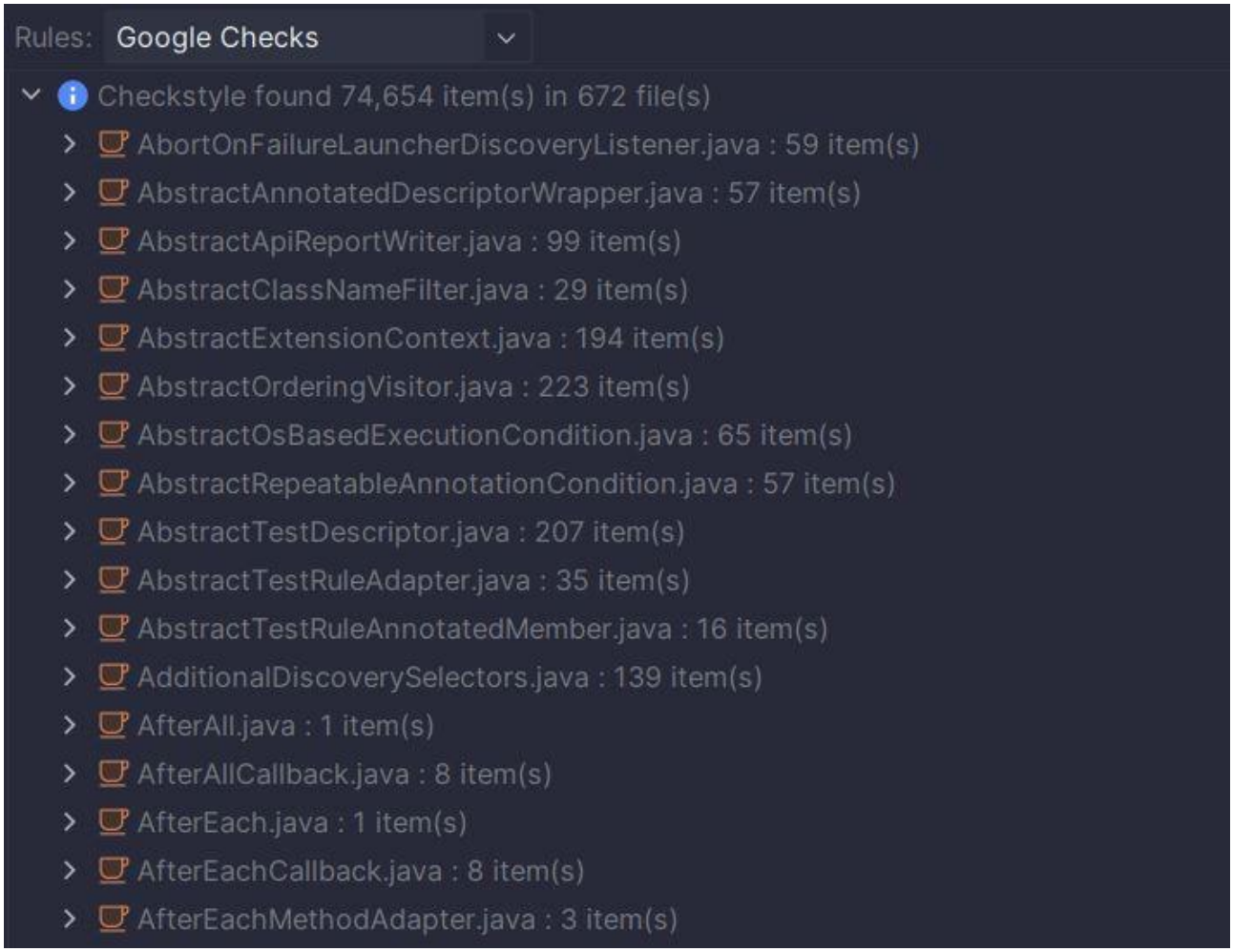


Рисунок 4.41 – Результати сканування проекту Junit5 з CheclStyle та посібником зі стилю від Google

Найвищий коефіцієнт помилок на рядок коду демонструє Check Style Google (0.737848149), що свідчить про його високі вимоги до стилістичних стандартів. PMD посідає другу позицію за суворістю, із коефіцієнтом 0.447340331, що вказує на його здатність виявляти широкий спектр помилок. Check Style Sun (0.218041471) також показує високу кількість виявлених помилок, але поступається PMD і Check Style Google. Найнижчий коефіцієнт помилок демонструє Qodana (0.001423234), що може бути результатом його зосередженості на специфічних аспектах перевірки. SpotBugs (0.002480776) і Sonar Lint (0.027762952) орієнтовані на вузьке коло помилок, таких як баги або вразливості, і мають значно меншу кількість помилок. IntelliJ IDEA має коефіцієнт 0.265077388, що є середнім серед представлених інструментів. Це демонструє її універсальність, оскільки вона покриває як стилістичні, так і

функціональні аспекти. SonarQube із показником 0.019421218 знаходиться між Qodana і IntelliJ, демонструючи більший фокус на якості коду, ніж Qodana, але меншу суворість порівняно з PMD чи Check Style.

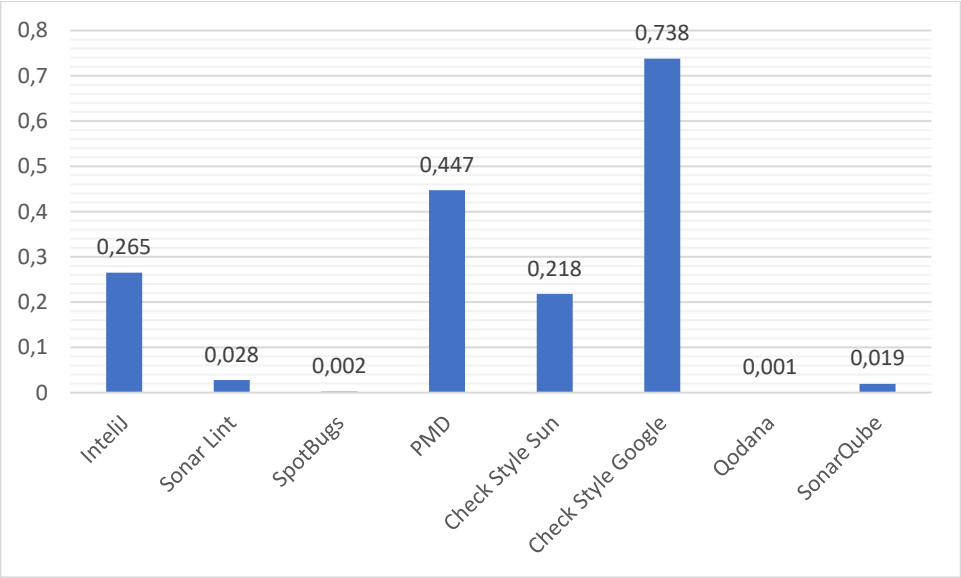


Рисунок 4.42 – Кількість помилок на рядок коду Junit5

Базуючись на даних від Statistic виберемо десять найбільших файлів та проведемо сканування кожного з них(табл. 4.3) та побудуємо графік порівняння кількості помилок від різних статичних аналізаторів(рис. 4.43):

Таблиця 4.3 – Результати сканування десяти найбільших файлів проекту Junit5

Назва файлу	Кількість рядків вихідного коду	InteliJ	Sonar Lint	Spot Bugs	PMD	Check Style Sun	Check Style Google
org.junit.jupiter.api.Assertions	1040	17	0	4	1009	2376	4538
org.junit.platform.commons.util.ReflectionUtils	1004	39	20	2	498	788	2688
org.junit.jupiter.engine.descriptor.ClassBasedTestDescriptor	386	8	4	0	181	233	681
org.junit.jupiter.engine.extension.TempDirectory	384	10	10	3	146	199	695
org.junit.jupiter.api.AssertArrayEquals	348	9	1	0	277	284	717
org.junit.platform.reporting.legacy.xml.XmlReportWriter	329	4	2	0	134	185	584

org.junit.platform.engine.discovery.DiscoverySelectors	299	16	13	0	226	428	1251
org.junit.platform.suite.commons.SuiteLauncherDiscoveryRequestBuilder	283	4	4	0	70	159	588
org.junit.platform.commons.util.AnnotationUtils	259	15	8	0	157	250	661
org.junit.platform.launcher.listeners.MutableTestExecutionSummary	239	3	1	1	81	99	411

Класи з найбільшою кількістю виявлених помилок:

- org.junit.jupiter.api.Assertions (найбільша кількість помилок, особливо в Check Style).
- org.junit.platform.commons.util.ReflectionUtils (другий за суворістю об’єкт перевірки).

Класи, які містять менше помилок:

- org.junit.platform.launcher.listeners.MutableTestExecutionSummary.
- org.junit.platform.commons.util.AnnotationUtils.

Для покращення якості коду треба пріоритизувати класи з великою кількістю помилок (Assertions, ReflectionUtils).

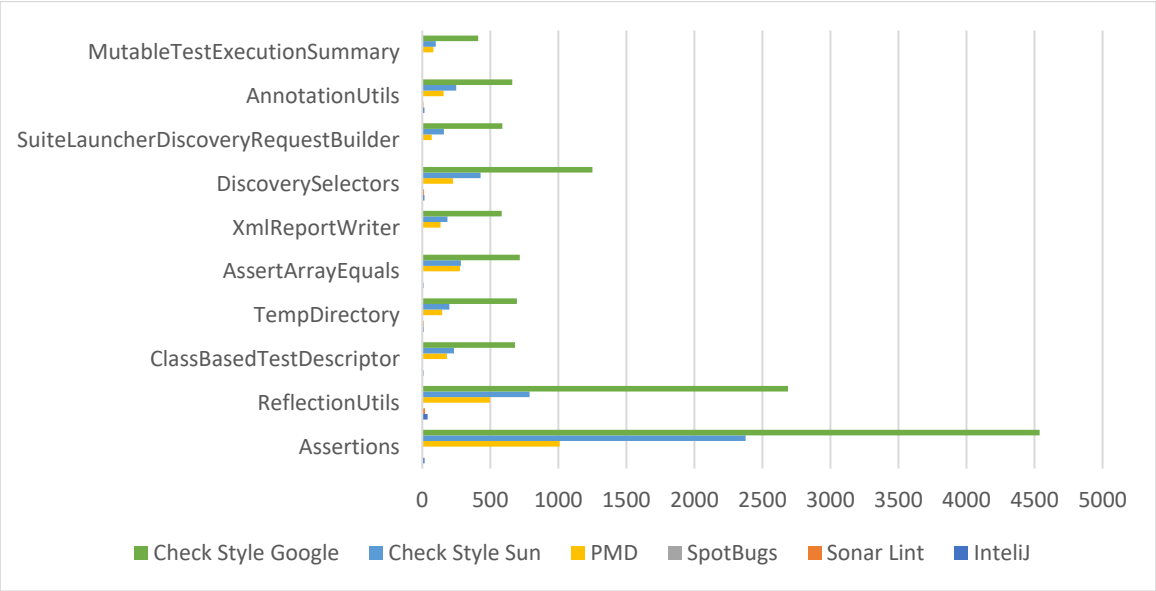


Рисунок 4.43 – Порівняння результатів сканування десяти найбільших файлів

проекту Junit5

Check Style Google і PMD найкраще підходять для виявлення потенційних проблем у стилі та структурі. Інструменти на кшталт SpotBugs, Qodana та SonarLint краще використовувати для перевірки багів і вразливостей. Використання кількох аналізаторів дає змогу охопити як стилістичні, так і логічні помилки.

Розглянемо Git статику:

Проект JUnit5 існує вже 3287 днів, з яких 1760 днів були активними (53.54%). Це свідчить про стабільну та тривалу розробку. Загалом було зроблено 8243 коміти, що в середньому становить 4.7 коміти на активний день і 2.5 коміти на всі дні існування проекту. Загальна кількість файлів: 1831. У коді проєкту додано 193291 рядків коду (384758 доданих і 191467 видалених), що свідчить про динамічний розвиток та рефакторинг. У розробці проекту взяли участь 245 авторів, із середнім внеском 33.6 комітів на автора. Це свідчить про колективний підхід до розробки.

Активність за місяцями (рис. 4.44), найвища активність спостерігалася у період 2016–2017 років, де кількість комітів значно перевищує середні показники. Після 2018 року активність знижується, що, ймовірно, пов’язано зі стабілізацією проекту та завершенням основної фази розробки.

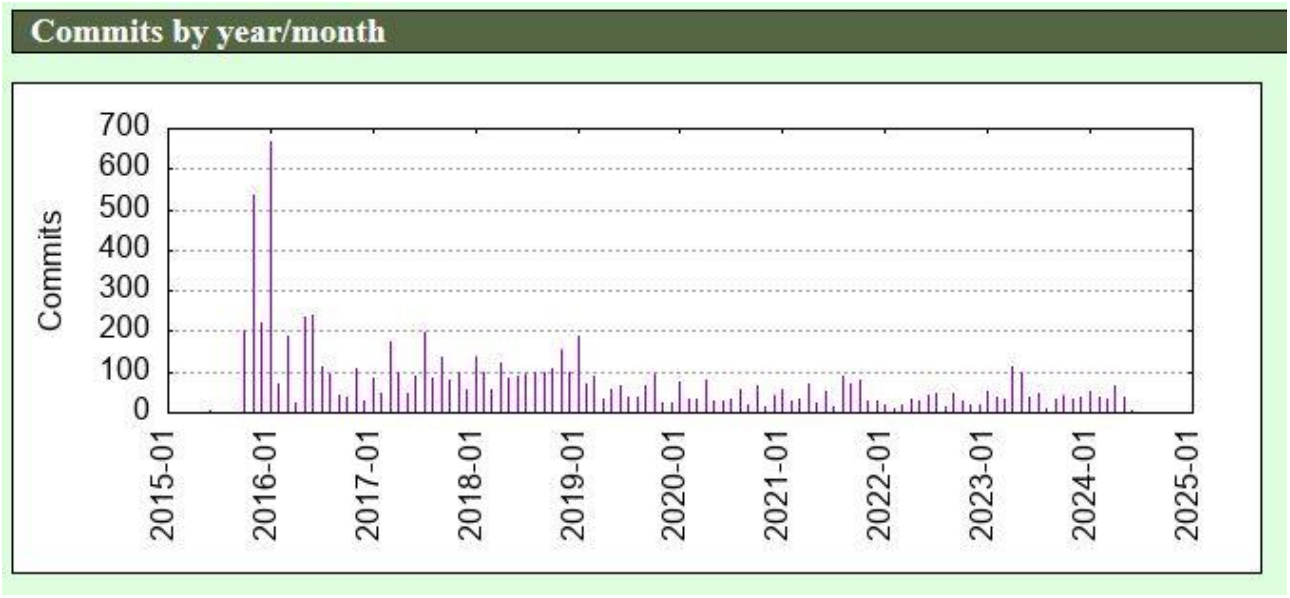


Рисунок 4.44 – Статистика коммітів за місяцями у проекті Junit5

Активність проєктуJUnit5 за роками(рис 4.45) демонструє поступовий розвиток і спад після основних фаз розробки. У 2015 році, коли проект розпочався, було виконано 965 коміти, що становить 12% від загального числа. У 2016 році активність досягла піку з 1847 комітами (22%), що свідчить про інтенсивну фазу розробки. У 2018 році високий рівень активності ще тривав із 1245 комітами (15%). У 2019 році кількість комітів знизилася до 784 (10%), і ця тенденція продовжувалася у 2020 році з 515 комітами (6%) та у 2021 році з 573 комітами (7%). У наступні роки активність проекту поступово зменшувалася, досягнувши 320 комітів у 2022 році (4%). У 2024 році було виконано лише 233 коміти, що свідчить про мінімальну активність, можливо, пов'язану з переходом проекту до фази підтримки або завершенням основних робіт.

Year	Commits (% of all)	Lines added	Lines removed
2024	233 (2.83%)	13043	5967
2023	576 (6.99%)	21051	11595
2022	320 (3.88%)	16370	7198
2021	573 (6.95%)	25883	11624
2020	515 (6.25%)	20559	10377
2019	784 (9.51%)	39247	18659
2018	1245 (15.10%)	69590	37915
2017	1185 (14.38%)	53950	22912
2016	1847 (22.41%)	86402	44247
2015	965 (11.71%)	38663	20973

Рисунок 4.45 – Статистика коммітів за роками у проекті Junit5

З 2015 року спостерігається стабільне зростання(рис. 4.46) кількості файлів у проекті. Це свідчить про активний розвиток функціональності та поступове розширення проекту. На початкових етапах (2015-2017 роки) ріст був помірним, але з часом темпи розробки прискорились, що може бути пов’язано з переходом до стабільної архітектури та підвищенням популярності JUnit5.

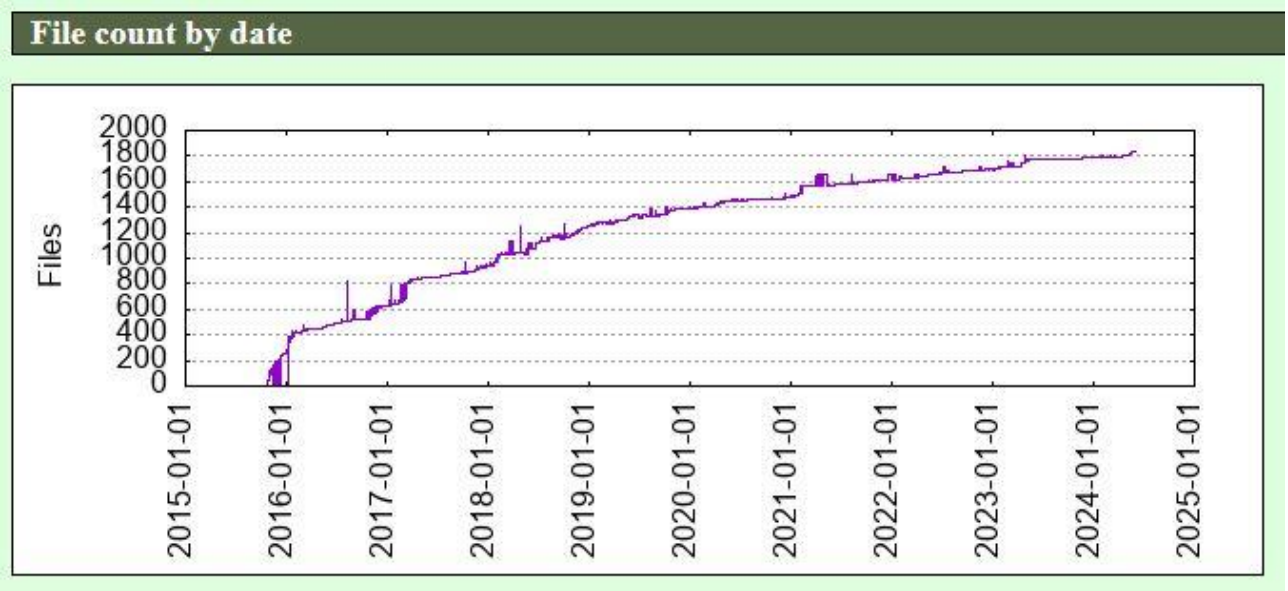


Рисунок 4.46 – Статистика кількості файлів у проекті Junit5

Графік(рис. 4.47) зростання рядків коду демонструє аналогічну тенденцію. Починаючи з перших років розробки, кількість рядків коду зросла до понад 180,000, що свідчить про значний обсяг внесених змін, додавання нових функцій і виправлення помилок.

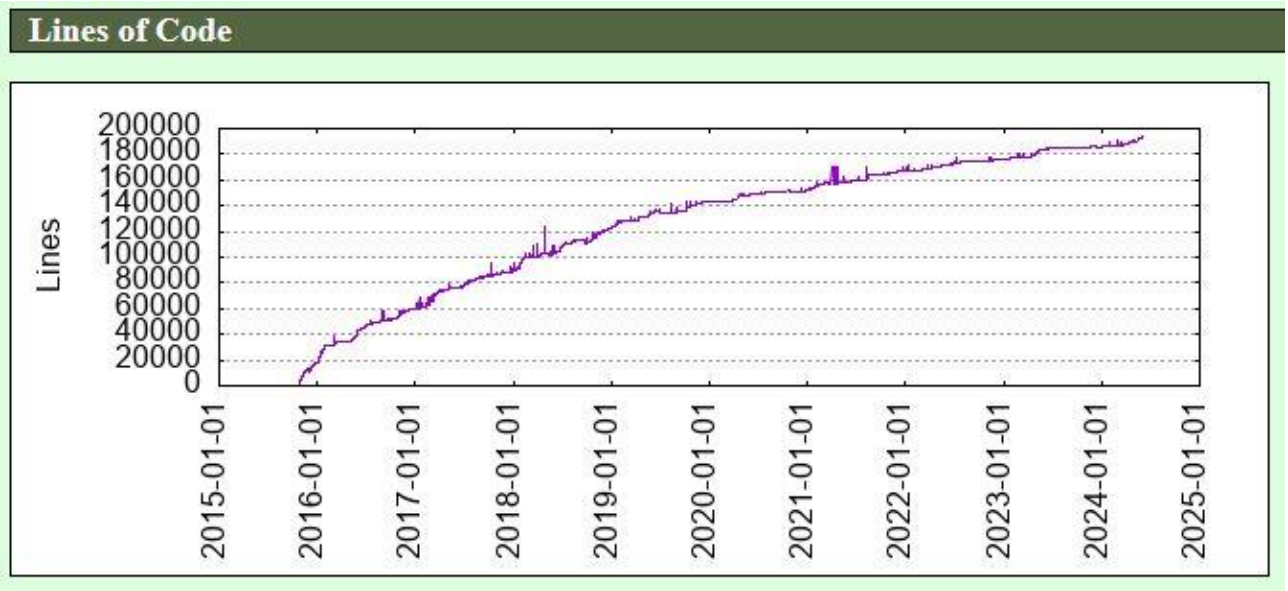


Рисунок 4.47 – Статистика росту рядків коду у проекті Junit5

Обидва графіки показують стабільність у розвитку проекту навіть після кількох років від запуску. Це підтверджує, що проект активно підтримується, вдосконалюється та залишається актуальним. Обидва графіки показують стабільність у розвитку проекту навіть після кількох років від запуску. Це підтверджує, що проект активно підтримується, вдосконалюється та залишається

актуальним. Зростання обсягу коду та файлів поступово стабілізується після 2022 року, що свідчить про досягнення певного рівня зрілості. Це може вказувати на те, що основна функціональність реалізована, а подальші зміни зосереджені на оптимізації, виправленні помилок та підтримці.

Зростання обсягу коду та файлів поступово стабілізується після 2022 року, що свідчить про досягнення певного рівня зрілості. Це може вказувати на те, що основна функціональність реалізована, а подальші зміни зосереджені на оптимізації, виправленні помилок та підтримці.

Розглянемо загальну інформацію про репозиторій:

Проект JUnit5 демонструє високу активність з моменту його створення, зокрема, загальна кількість комітів становить 8765, що свідчить про інтенсивний розвиток. Було створено 1753 пул-реквестів і зареєстровано 4072 задачі (issues), що відображає значну активність як з боку розробників, так і користувачів. Середньостатистична активність проекту також підкреслюється числом учасників (229), що свідчить про широку спільноту розробників. Популярність JUnit5 підтверджується кількістю зірок на GitHub (6445) та форків (1505), що вказує на визнання проєкту в індустрії та активне його використання.

JUnit 5 демонструє помірну, але стабільну активність із високим рівнем закритих пулл-реквестів та задач, що вказує на ефективну підтримку й розвиток. Більшість змін є невеликими та швидко інтегруються завдяки злагодженому процесу рев'ю. Складніші пулл-реквести з більшою кількістю змін потребують більше часу та обговорень, але це є винятком. Проект активно розвивається, додаючи новий функціонал і підтримуючи якість кодової бази. Додаткові дослідження активності проекту, відношення додавань та видалень у пулл-реквестах, відношення загальної кількості змін у коді до кількості коментарів від розробників, відношення часу, який пулл-реквест був відкритим до кількості коментарів від розробників в пулл-реквесті, та відношення часу, який пулл-реквест був відкритим до загальної кількості змін у коді в пулл-реквестах приведені нижче (див. Додаток В).

На графіку(рис. 4.48) видно, що максимальна активність спостерігалася

на початкових етапах, зокрема в 2016–2017 роках, коли кількість комітів, пул-реквестів і задач стрімко зростала. Пік активності, ймовірно, збігся з періодом активного розвитку основної функціональності проекту. Після цього кількість комітів і пул-реквестів поступово стабілізувалася, хоча зберігаються регулярні оновлення та підтримка.

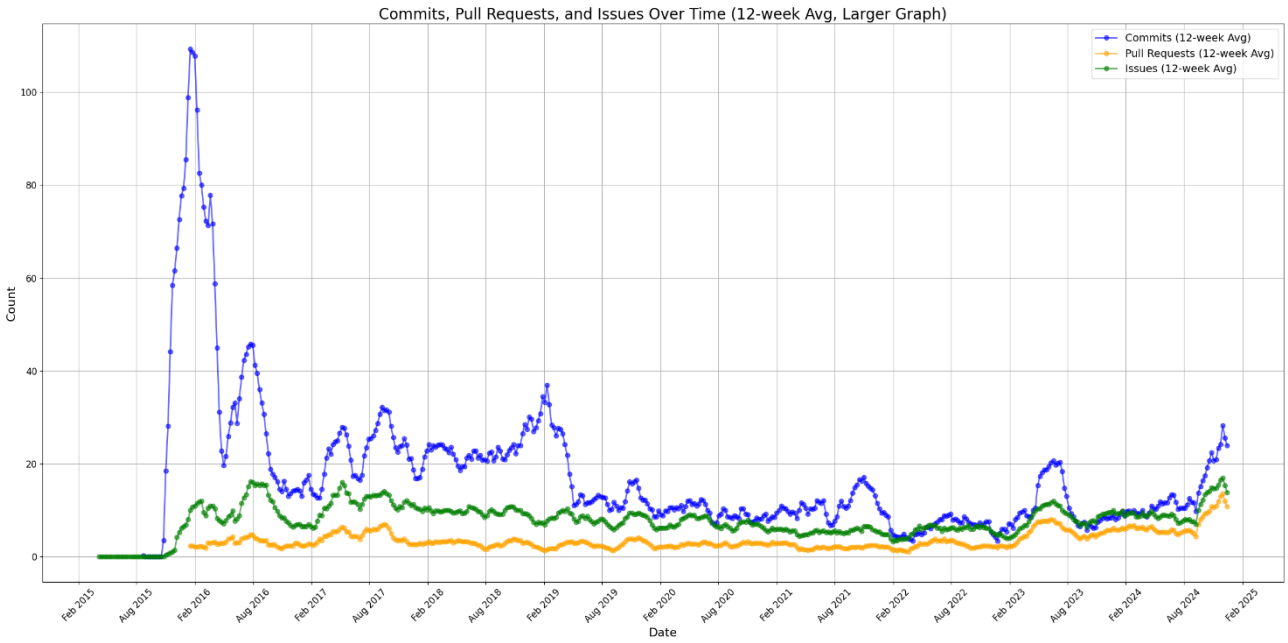


Рисунок 4.48 – Відношення коммітів/пулл-реквестів/задач репозиторію Junit5

На графіку(рис. 4.48) також видно певні сплески активності у більш пізні періоди, що може бути пов’язано з випуском нових функцій або оновленням проєкту відповідно до змін у екосистемі Java.

Отже, проєкт JUnit5 демонструє сталість та популярність у спільноті розробників, маючи значну історію комітів, активне обговорення задач і регулярне оновлення.

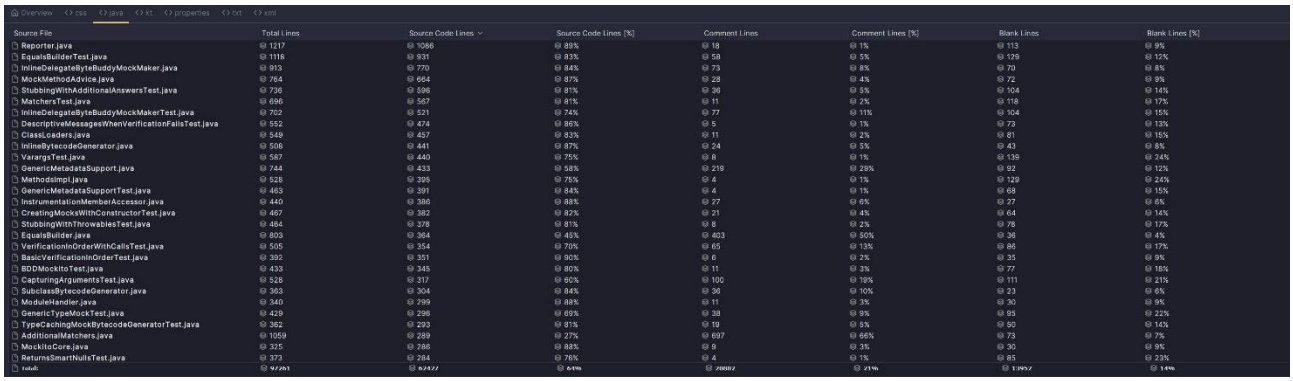
4.2.4 Mockito

Mockito — це популярний фреймворк для створення mock-об'єктів у Java, розроблений у 2008 році для спрощення модульного тестування. Проект зосереджений на читабельності тестів і мінімізації шаблонного коду, швидко ставши стандартом у тестуванні завдяки простоті використання та активній підтримці спільноти.

Структурна характеристика проекту:

Mockito має доволі просту логічну структуру. Проектна документація надає загальну інформацію, але такий проект міг би мати кращий рівень. Проект має набагато завеликих класів, код добре задокументований. Але присутні деякі старі класи. Mockito має високе покриття тестами. Загалом, проект добре оформлений, але має замало потрібної інформації.

Роздивимось загальну статистику рядків коду у проекті(рис 4.49):



Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Reporter.java	1217	1086	89%	18	1%	113	9%
EqualsBuilderTest.java	1118	931	83%	58	5%	129	12%
InlineDelegateByteBuddyMockMaker.java	913	770	84%	73	8%	70	8%
MockMethodAdvice.java	764	664	87%	28	4%	72	9%
StubbingWithAdditionalAnswersTest.java	736	596	81%	36	5%	104	14%
MatchersTest.java	696	567	81%	11	2%	118	17%
InlineDelegateByteBuddyMockMakerTest.java	702	521	74%	77	11%	104	15%
DescriptiveMessagesWhenVerificationFailsTest.java	552	474	86%	5	1%	73	13%
ClassLoaders.java	549	457	83%	11	2%	81	15%
InlineBytecodeGenerator.java	506	441	87%	24	5%	43	8%
VarargsTest.java	587	440	75%	8	1%	139	24%
GenericMetadataSupport.java	744	433	58%	219	29%	92	12%
MethodsImpl.java	528	395	75%	4	1%	129	24%
GenericMetadataSupportTest.java	463	391	84%	4	1%	68	15%
InstrumentationMemberAccessor.java	440	386	88%	27	6%	27	6%
CreatingMocksWithConstructorTest.java	467	382	82%	21	4%	64	14%
StubbingWithThrowablesTest.java	464	378	81%	8	2%	78	17%
EqualsBuilder.java	803	364	45%	403	50%	36	4%
VerificationInOrderWithCallsTest.java	505	354	70%	65	13%	86	17%
BasicVerificationInOrderTest.java	392	351	90%	6	2%	35	9%
BDDMockitoTest.java	433	345	80%	11	3%	77	18%
CapturingArgumentsTest.java	528	317	60%	100	19%	111	21%
SubclassBytecodeGenerator.java	363	304	84%	36	10%	23	6%
ModuleHandler.java	340	299	88%	11	3%	30	9%
GenericTypeMockTest.java	429	296	69%	38	9%	95	22%
TypeCachingMockBytecodeGeneratorTest.java	382	293	87%	19	5%	50	14%
AdditionalMatchers.java	1059	289	27%	697	66%	73	7%
MockitoCore.java	325	286	88%	9	3%	30	9%
ReturnsSmartNullsTest.java	373	284	76%	4	1%	85	23%
total	97261	62427	64%	28882	21%	13952	14%

Рядки вихідного коду становлять близько 64% від загальної кількості рядків. Це схоже на Apache Dubbo (~64%) і Google Guava (~65%), вказуючи на зосередженість на основній функціональності з достатнім рівнем супровідного коду. Mockito, хоч і менший за масштабами порівняно з іншими розглянутими проєктами, має подібну структуру до Dubbo і Guava. Його компактний розмір і високий відсоток коду вказують на зосередженість на основних функціях з підтримкою тестування та документації.

Проведемо сканування проекту за допомогою інструментів статичного аналізу (рис. 4.50 — 4.57):

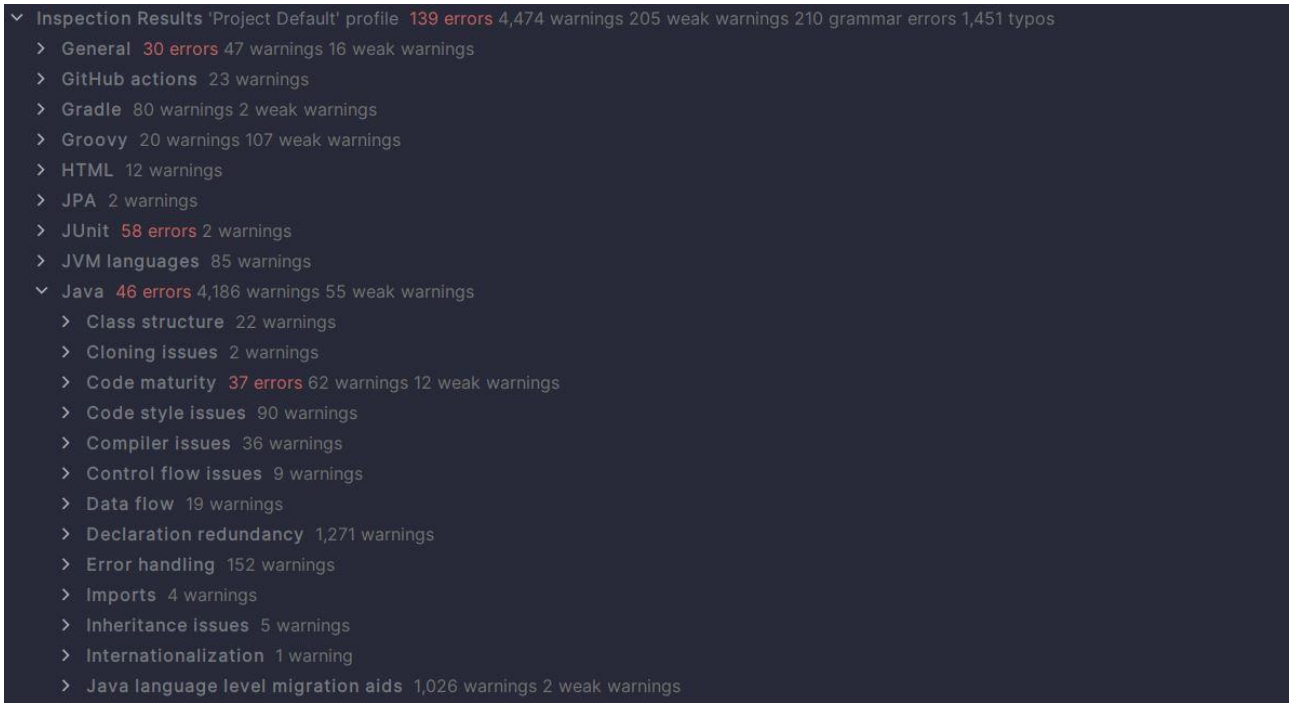


Рисунок 4.50 – Результати сканування проекту Mockito з InteliJ

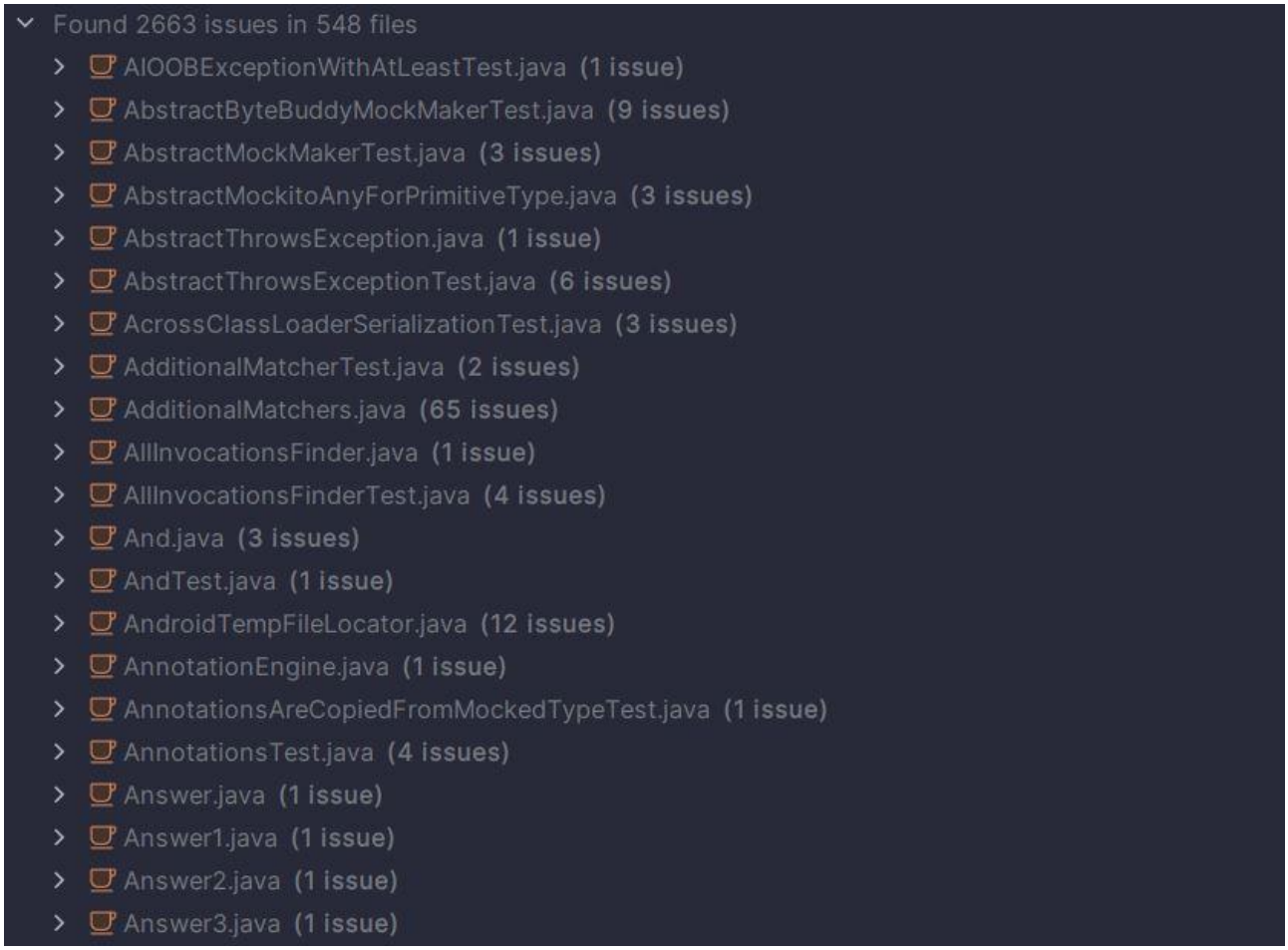


Рисунок 4.51 – Результати сканування проекту Mockito з SonarLint

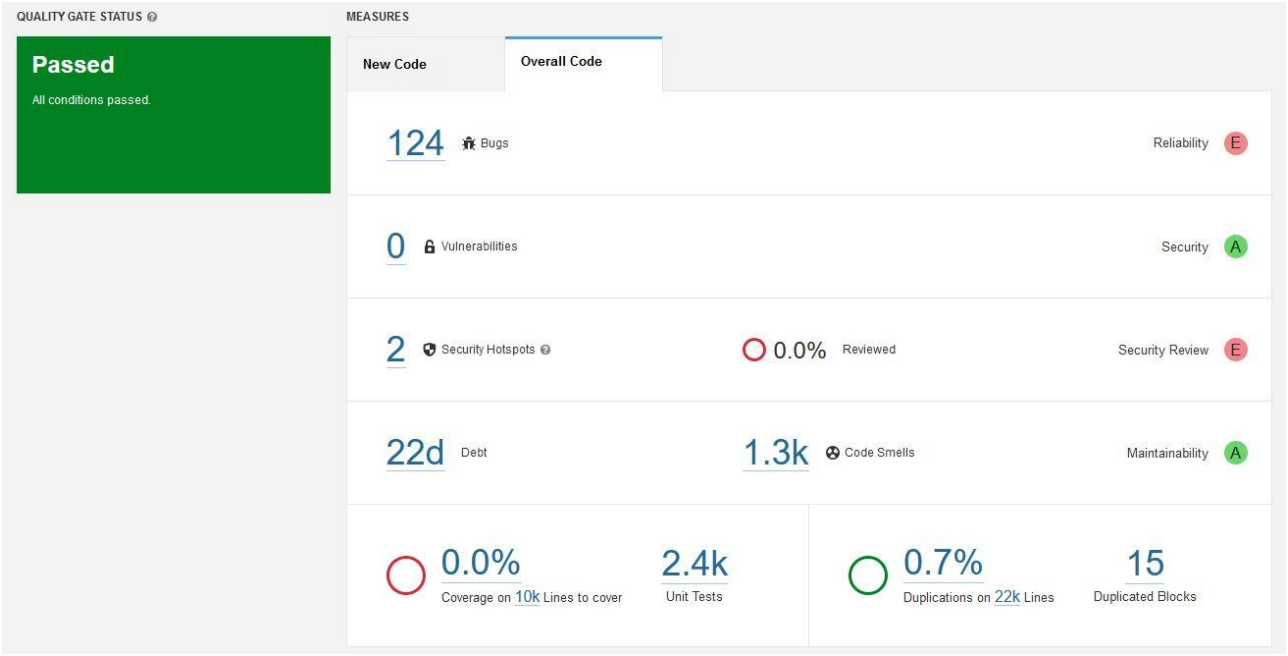


Рисунок 4.52 – Результати сканування проекту Mockito з SonarQube

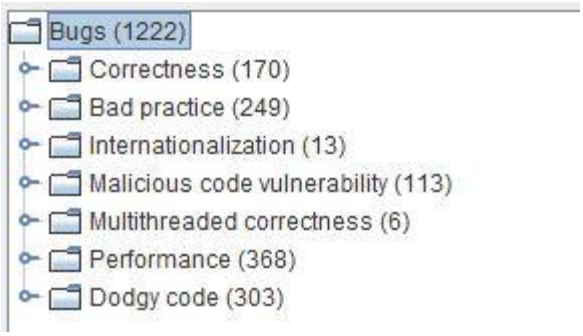


Рисунок 4.53 – Результати сканування проекту Mockito з SpotBugs

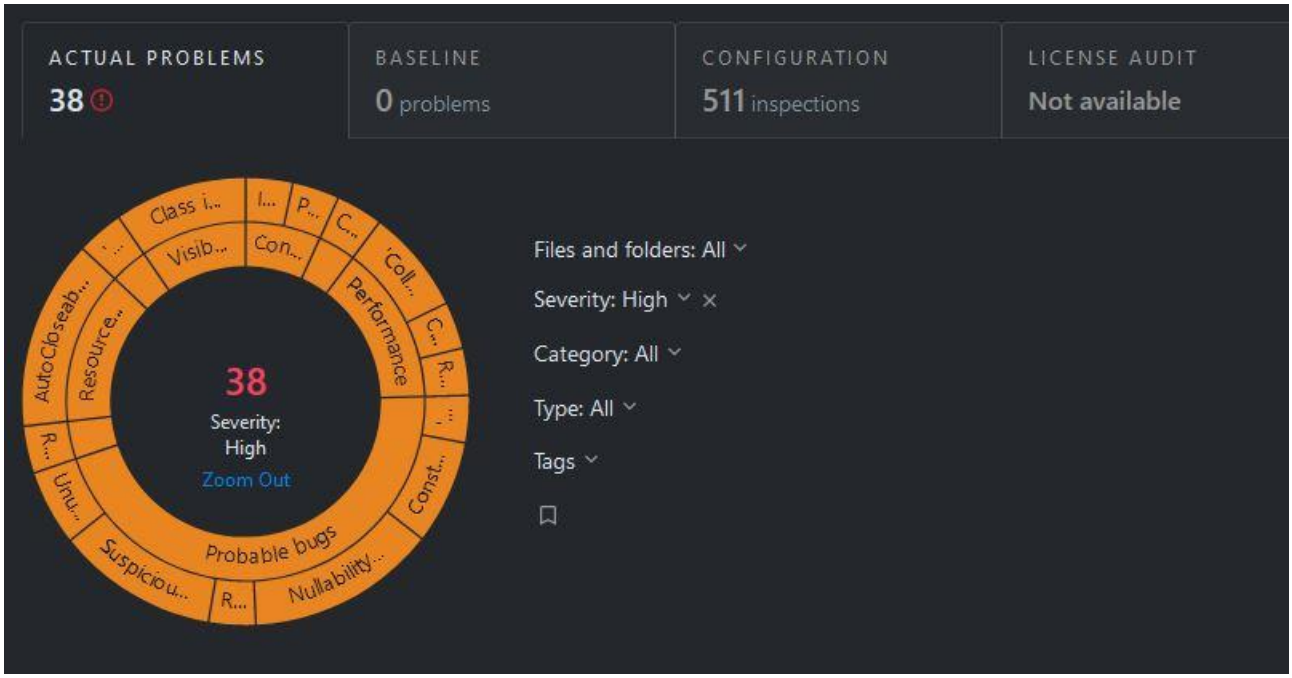


Рисунок 4.54 – Результати сканування проекту Mockito з Qodana

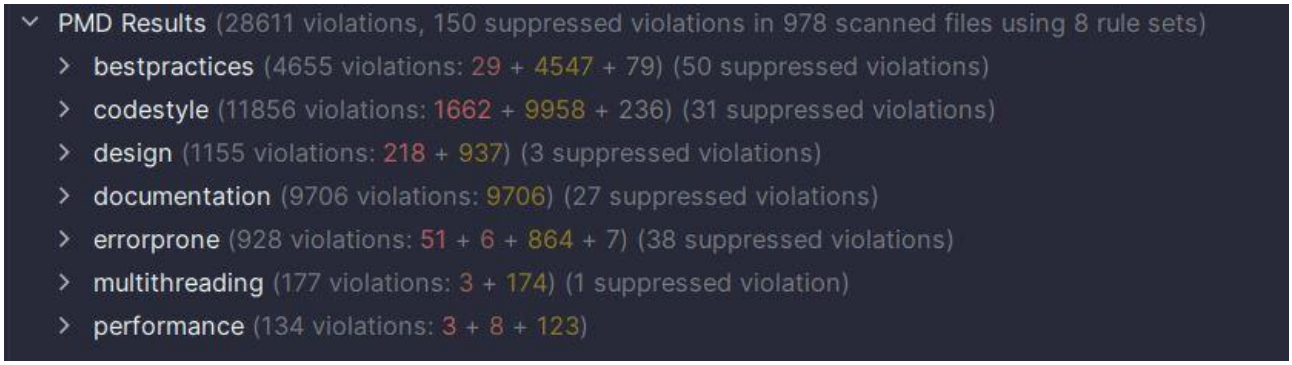


Рисунок 4.55 – Результати сканування проекту Mockito з PMD

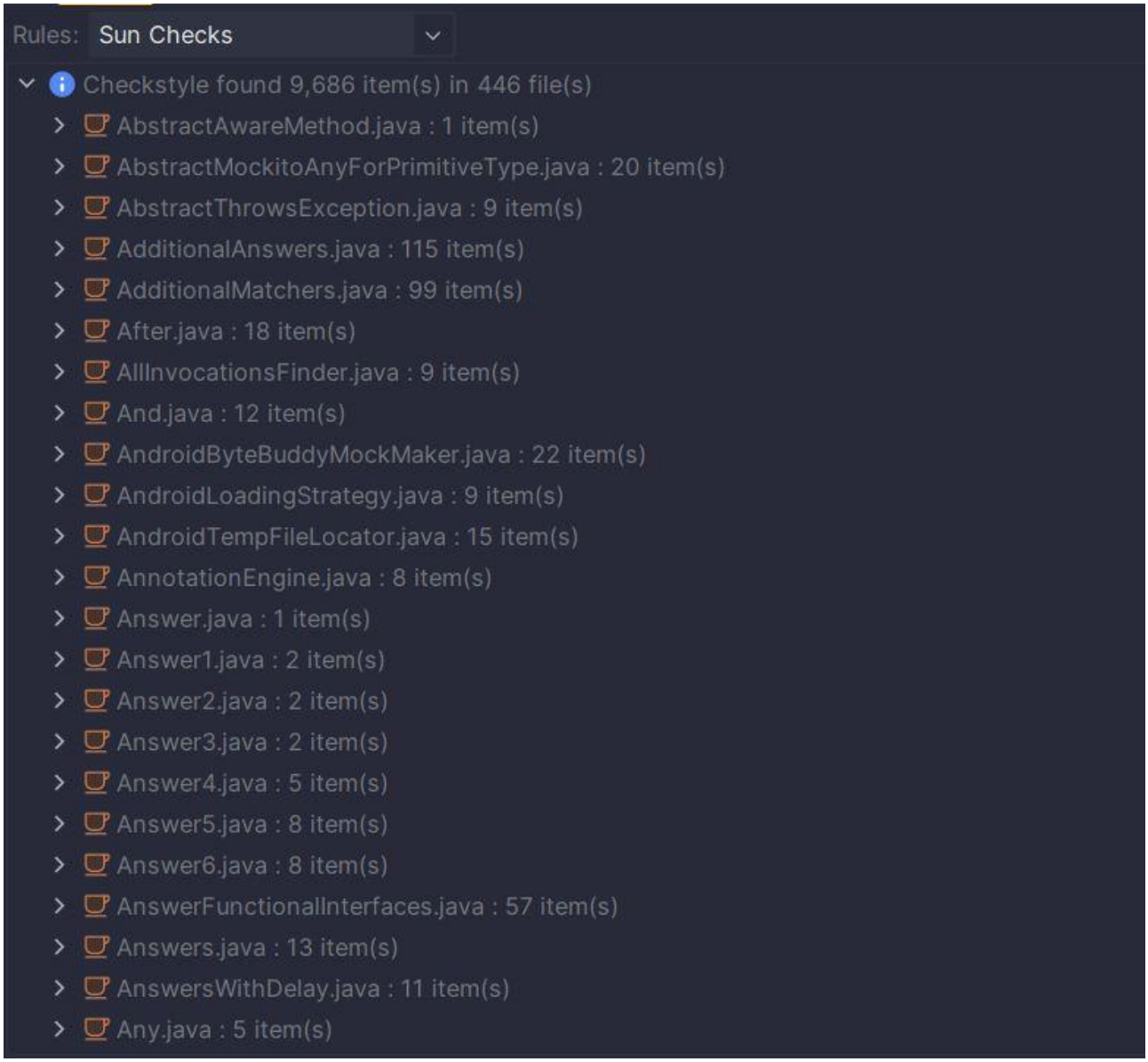


Рисунок 4.56 – Результати сканування проекту Mockito з CheclStyle та посібником зі стилю від Sun

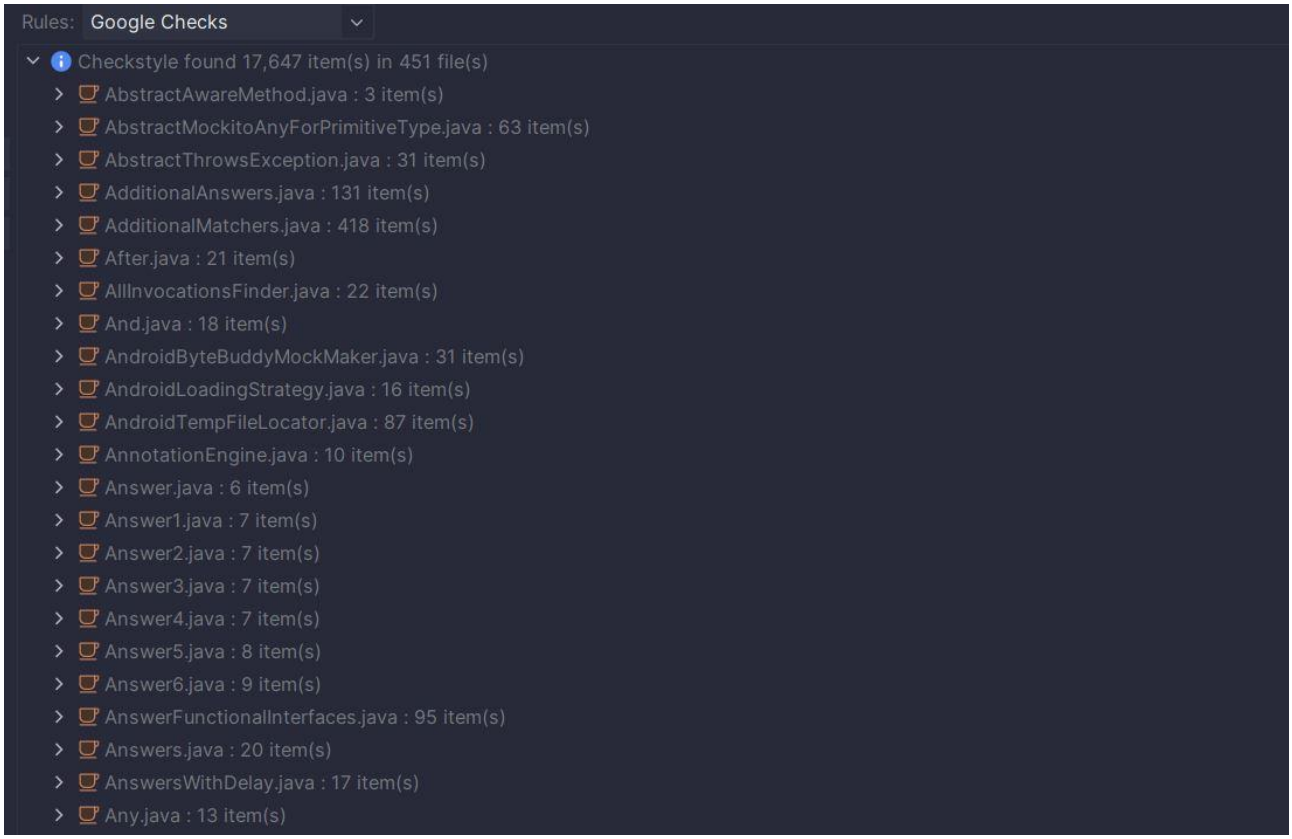


Рисунок 4.57 – Результати сканування проекту Mockito з CheclStyle та посібником зі стилю від Google

Базуючись на даних від Statistic виберемо десять найбільших файлів та проведемо сканування кожного з них(табл. 4.4) та побудуємо графік порівняння кількості помилок від різних статичних аналізаторів(рис. 4.58):

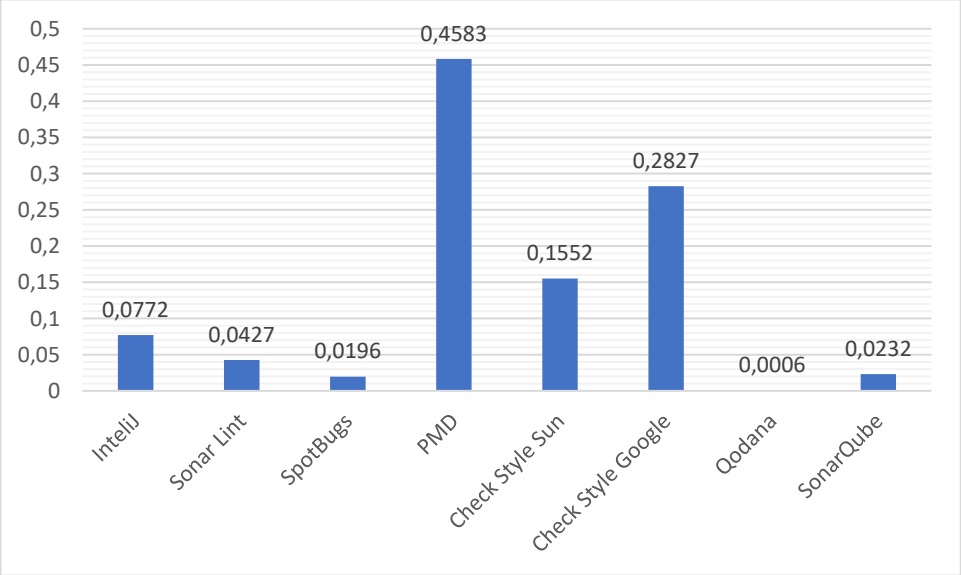


Рисунок 4.58 – Кількість помилок на рядок коду Mockito

Аналіз найбільших файлів проекту Mockito за допомогою різних

інструментів виявив значну варіативність у кількості знайдених проблем. Інструмент PMD виявляє найбільшу кількість потенційних помилок серед усіх сканованих файлів (середнє значення на рядок коду становить 0.458), що свідчить про його строгість у перевірці відповідності коду правилам написання. Високий рівень знайдених помилок також спостерігається у Check Style Google (0.283) та Check Style Sun (0.155), що підтверджує необхідність дотримання стилістичних стандартів у проекті. SonarQube (0.023) та SonarLint (0.043) мають більш помірний рівень виявлення помилок, зосереджуючись переважно на логічних та структурних проблемах у коді. SpotBugs, з найнижчим середнім значенням (0.020), демонструє специфічність у пошуку дефектів, зосереджених на багаторівневому аналізі коду. Інструмент InteliJ має значно меншу кількість виявлених проблем (0.077), що свідчить про його інтеграцію з процесом розробки і зосередженість на менш критичних аспектах коду. Qodana показує найнижче середнє значення (0.001), що може бути обумовлено його специфічністю у виявленні вузького спектра проблем.

Таблиця 4.4 – Результати сканування десяти найбільших файлів проекту Mockito

Назва файлу	Кількість рядків вихідного коду	InteliJ	Sonar Lint	Spot Bugs	PMD	Check Style Sun	Check Style Google
org.mockito.internal.exceptions.Reporter	1086	25	9	0	299	412	507
org.mockito.internal.creation.bytebuddy.InlineDelegateByteBuddyMockMaker	770	31	14	6	197	222	536
org.mockito.internal.creation.bytebuddy.MockMethodAdvice	664	40	23	25	247	218	348
org.mockito.internal.creation.bytebuddy.InlineBytecodeGenerator	441	40	20	4	144	141	262
org.mockito.internal.util.reflection.GenericMetadataSupport	433	35	13	8	205	185	412

org.mockito.internal.util.reflection.InstrumentationMemberAccessor	386	15	11	2	88	80	206
org.mockito.internal.matchers.apachecommons.EqualsBuilder	364	22	1	35	175	101	344
org.mockito.internal.creation.bytebuddy.SubclassBytecodeGenerator	304	10	3	0	111	91	172
org.mockito.internal.creation.bytebuddy.ModuleHandler	299	13	4	3	99	80	161
org.mockito.AdditionalMatchers	289	65	65	9	217	99	418

Найбільш проблемним файлом виявився org.mockito.internal.exceptions.Reporter, який має найбільшу кількість помилок за всіма інструментами, особливо у PMD та Check Style Google. Це вказує на необхідність додаткової уваги до цього файлу з метою підвищення його якості.

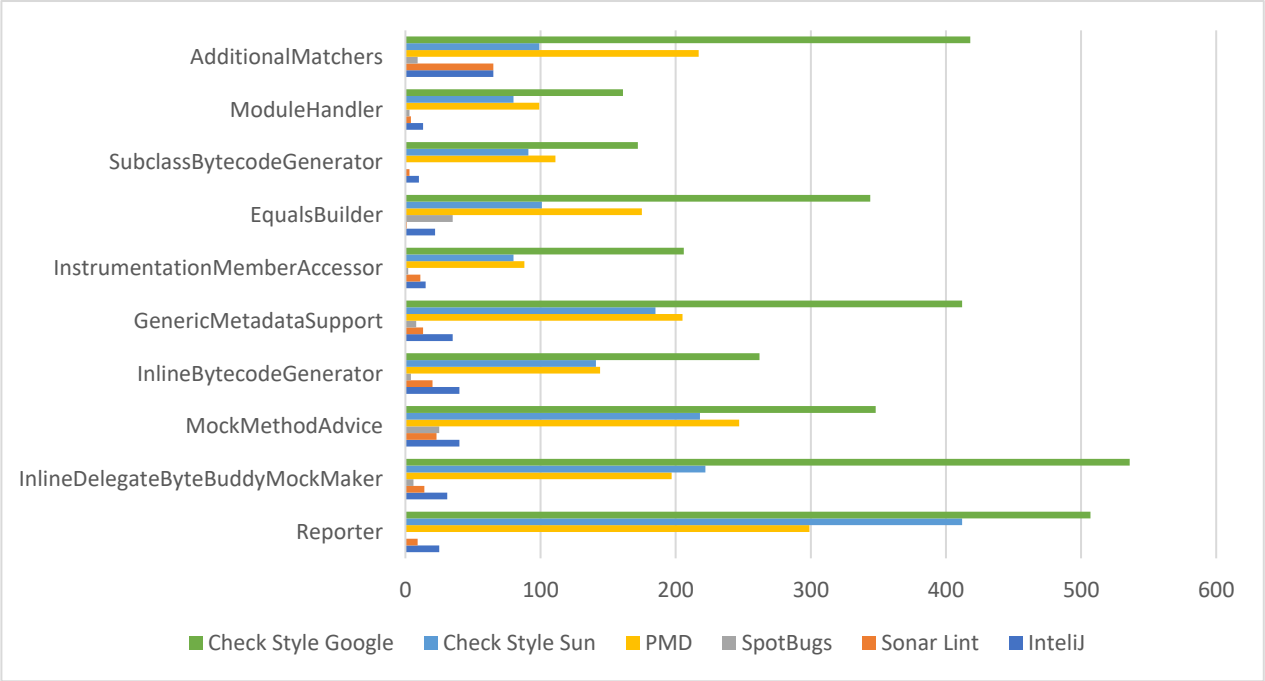


Рисунок 4.59 – Порівняння результатів сканування десяти найбільших файлів проекту Mockito

PMD виявляє найбільше помилок у більшості файлів, що свідчить про його строгі правила статичного аналізу. Check Style Google і Check Style Sun вказують на значну кількість стилістичних проблем, що може бути вирішено

через впровадження єдиних стандартів кодування. SpotBugs знаходить значно менше дефектів, зосереджуючись на специфічних логічних помилках. SonarLint та SonarQube дають меншу кількість помилок, але акцентують увагу на важливих структурних аспектах. Файли з великою кількістю рядків коду виявляються найбільш проблемними, що вказує на потребу у рефакторингу та розбитті великого коду на менші компоненти. Проблемні файли, такі як Reporter та InlineDelegateByteBuddyMockMaker, потребують першочергової уваги. Це допоможе знизити загальну кількість помилок у проекті Mockito та покращити якість коду.

Загалом, аналіз демонструє, що інструменти мають різну специфіку і можуть бути корисними в комплексі для забезпечення якісного коду в проекті Mockito.

Розглянемо Git статику:

Проекту 6046 днів (близько 16 років), з яких активними були 1735 днів (28.7%), що свідчить про нерівномірний розподіл роботи протягом усього часу. У проекті нараховується 1099 файлів, загальний обсяг коду складає 105953 рядків (428053 доданих і 322100 видалених), що свідчить про активну підтримку та рефакторинг. Загальна кількість комітів – 6090, що в середньому становить 3.5 коміти на активний день і лише 1 коміт на всі дні, вказуючи на зосередження активності у певні періоди. У проекті взяли участь 333 автори, що в середньому дає 18.3 комітів на автора.

Внесення змін відбувалося нерівномірно (рис. 4.60): періоди високої активності змінювалися місяцями майже повної відсутності змін. Найбільш насичені місяці припадають на час пікової активності 2015–2017 років, коли кількість комітів перевищувала 150 на місяць.

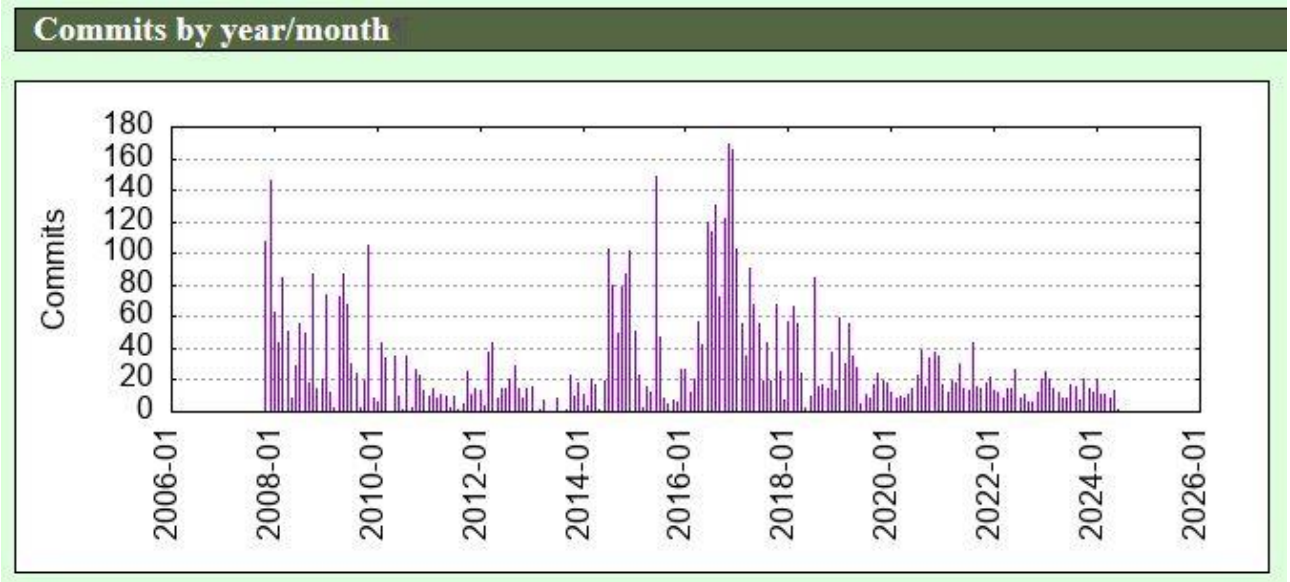
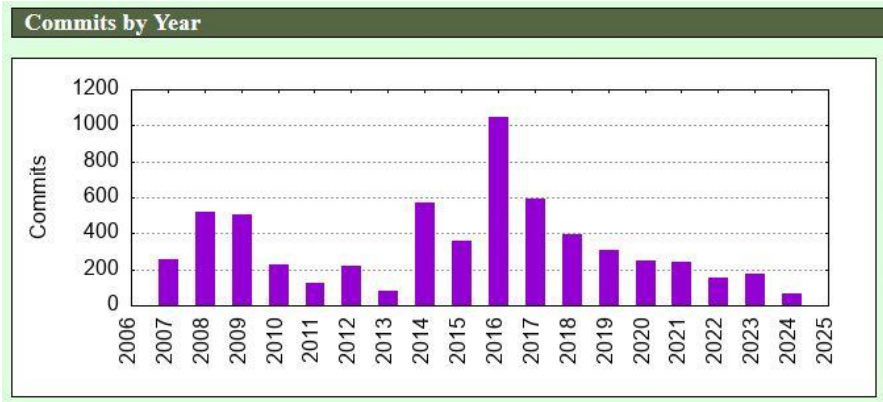


Рисунок 4.60 – Статистика коммітів за місяцями у проекті Mockito

Найвищий рівень активності спостерігався у 2014–2017 роках(рис. 4.61), коли кількість комітів досягала пікових значень (від 500 до 1040 комітів на рік). У цей період відбулося багато змін у коді, що підтверджує значну активність розробників. З 2017 року активність значно знижується, і кількість комітів не перевищує 400 на рік, що може свідчити про завершення основного етапу розробки і перехід до фази підтримки. У 2024 році активність становила 65 комітів, що свідчить про продовження роботи над проектом, але в малих масштабах.



Year	Commits (% of all)	Lines added	Lines removed
2024	65 (1.07%)	864	9203
2023	176 (2.89%)	6331	3183
2022	151 (2.48%)	6210	2079
2021	239 (3.92%)	10185	7271
2020	249 (4.09%)	20200	11039
2019	310 (5.09%)	10090	4423
2018	396 (6.50%)	7234	2696
2017	590 (9.69%)	11862	6612
2016	1048 (17.21%)	29971	20322
2015	355 (5.83%)	70294	103817
2014	571 (9.38%)	12369	7707
2013	84 (1.38%)	2927	987
2012	221 (3.63%)	19292	6841
2011	124 (2.04%)	7988	3031
2010	230 (3.78%)	10552	3699
2009	505 (8.29%)	64120	14434
2008	523 (8.59%)	98567	88981
2007	253 (4.15%)	38997	25775

Рисунок 4.61 – Статистика коммітів за роками у проекті Mockito

Проект Mockito пройшов кілька фаз розвитку: активна розробка (до 2017

року), поступове зниження активності (2017–2023 роки), і підтримки певної активності у 2024 році. Незважаючи на зменшення кількості комітів, підтримка проекту триває, що вказує на його важливість і затребуваність у спільноті розробників.

З моменту початку проекту у 2008 році(рис 4.62) кількість файлів зростала поступово, відображаючи збільшення обсягу функціоналу. Значне зростання кількості файлів спостерігається у період з 2009 по 2012 роки, після чого темпи додавання файлів стали стабільнішими. Після 2020 року кількість файлів залишається майже незмінною на рівні близько 1100, що вказує на фокусування команди розробників на підтримці й оптимізації існуючого функціоналу замість створення нових модулів.

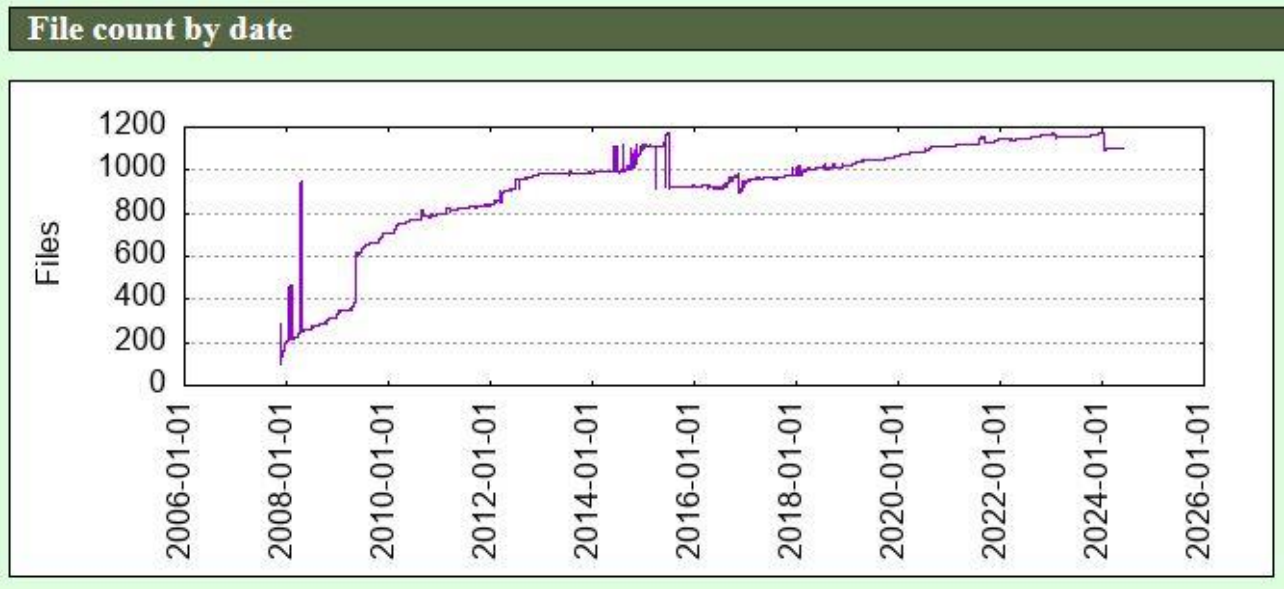


Рисунок 4.62 – Статистика кількості файлів у проекті Mockito

З моменту початку розробки загальна кількість рядків коду у проекті суттєво зросла(рис. 4.63), відображаючи інтенсивний розвиток у початкові роки. Великі скачки у кількості рядків коду у 2009 і 2014 роках можуть вказувати на інтеграцію великих змін або нових функцій. Після 2015 року темпи зростання кількості рядків коду сповільнилися, але зберігається стабільний рівень підтримки та невеликого розвитку. Максимальна кількість рядків коду досягла понад 110000, що свідчить про масштабність проекту.

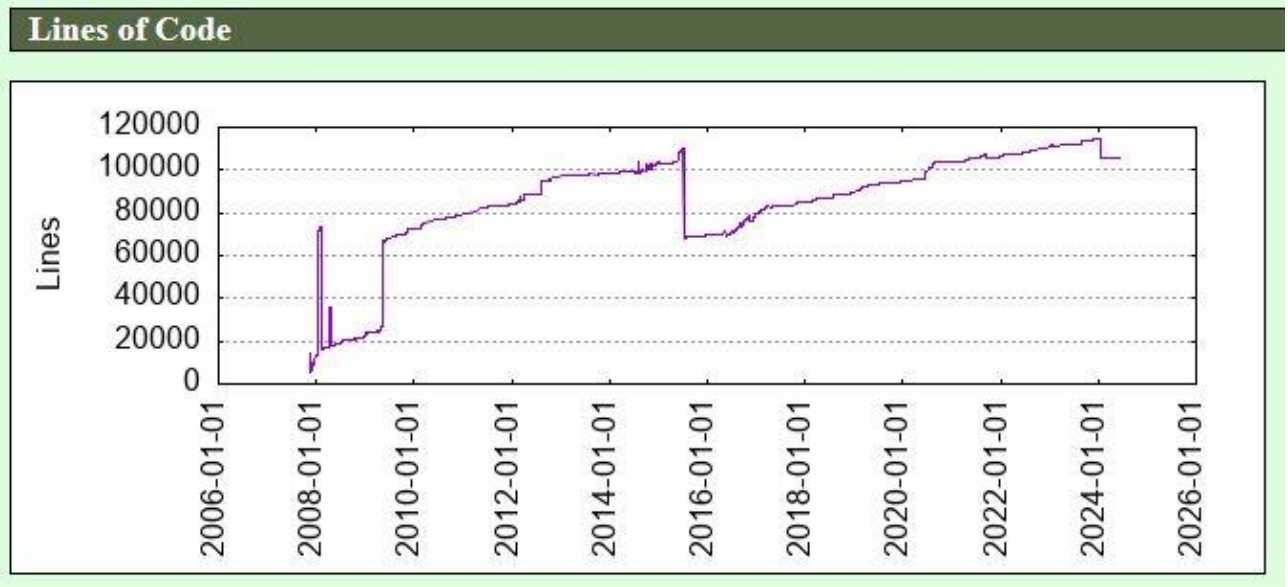


Рисунок 4.63 – Статистика росту рядків коду у проекті Mockito

Проект Mockito розвивався поступово, з піками активності у певні періоди. Наразі кількість файлів та рядків коду стабілізувалася, що вказує на перехід проекту до фази підтримки. Це свідчить про те, що основні функції вже реалізовані, а команда розробників зосереджена на оптимізації та підтримці стабільності.

Розглянемо загальну інформацію про репозиторій:

Репозиторій має 6213 комітів, що свідчить про інтенсивний розвиток проекту протягом усього часу його існування. 1822 пулл-реквестів і 3516 задач відображають активну взаємодію спільноти з кодовою базою та активне обговорення та вирішення проблем. У проекті взяли участь 289 контриб’юторів, що свідчить про відкритість репозиторію та популярність серед розробників. Mockito має 14936 зірок на GitHub, що є показником його високої популярності та широкого використання серед розробників. Репозиторій також має 2569 форків, що свідчить про значну кількість розробників, які адаптують його для власних потреб або беруть участь у його розробці.

Mockito демонструє середню активність із стабільною підтримкою проекту. Більшість пулл-реквестів містять невеликі зміни, які швидко інтегруються з мінімальними обговореннями, що свідчить про ефективні процеси рев’ю. Проект збалансовано розвивається: додається новий функціонал,

а старий код оптимізується, підтримуючи якість. Складні зміни трапляються рідко та обробляються структуровано, що робить Mockito надійним інструментом для тестування. Додаткові дослідження активності проекту, відношення додавань та видалень у пулл-реквестах, відношення загальної кількості змін у коді до кількості коментарів від розробників, відношення часу, який пулл-реквест був відкритим до кількості коментарів від розробників в пулл-реквесті, та відношення часу, який пулл-реквест був відкритим до загальної кількості змін у коді в пулл-реквестах приведені нижче (див. Додаток В).

Графік(рис. 4.64) відображає піки активності у певні періоди, зокрема значний ріст комітів у проміжок часу між 2013 і 2018 роками. Це може бути пов’язано з активним розвитком проекту, інтеграцією нових функцій або випуском важливих оновлень. Після піку спостерігається поступове зменшення інтенсивності активності, але вона залишається стабільною. Це свідчить про те, що проект досяг певної зрілості, і розробники більше фокусуються на підтримці, виправленні помилок і вдосконаленні існуючого функціоналу.

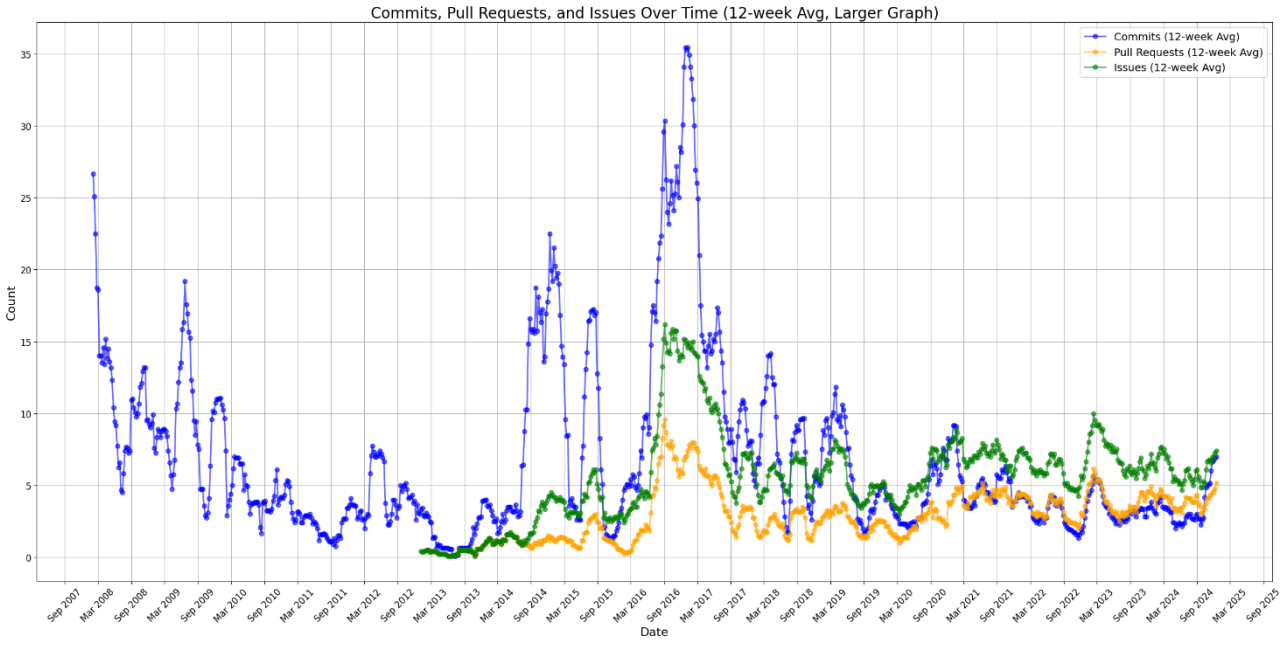


Рисунок 4.64 – Відношення коммітів/пулл-реквестів/задач репозиторію Mockito

Кількість пулл-реквестів порівняно з задачами вказує на те, що спільнота бере участь у внесенні змін до проекту, а не лише у звітуванні про проблеми, але кількість задач все одно залишається вищою. Графік показує, що коміти стабільно перевищують кількіст пулл-реквестів та задач, що свідчить про

активну розробку як з боку основних учасників, так і з боку спільноти.

Mockito є активним та популярним проектом із відкритим кодом, який користується попитом серед спільноти розробників. Висока кількість комітів і контриб'юторів демонструє постійний розвиток, а кількість зірок та форків свідчить про його значення для програмування. Хоча темпи активності зменшилися після пікових років, стабільність у кількості пулл-реквестів та задач вказує на надійний рівень підтримки.

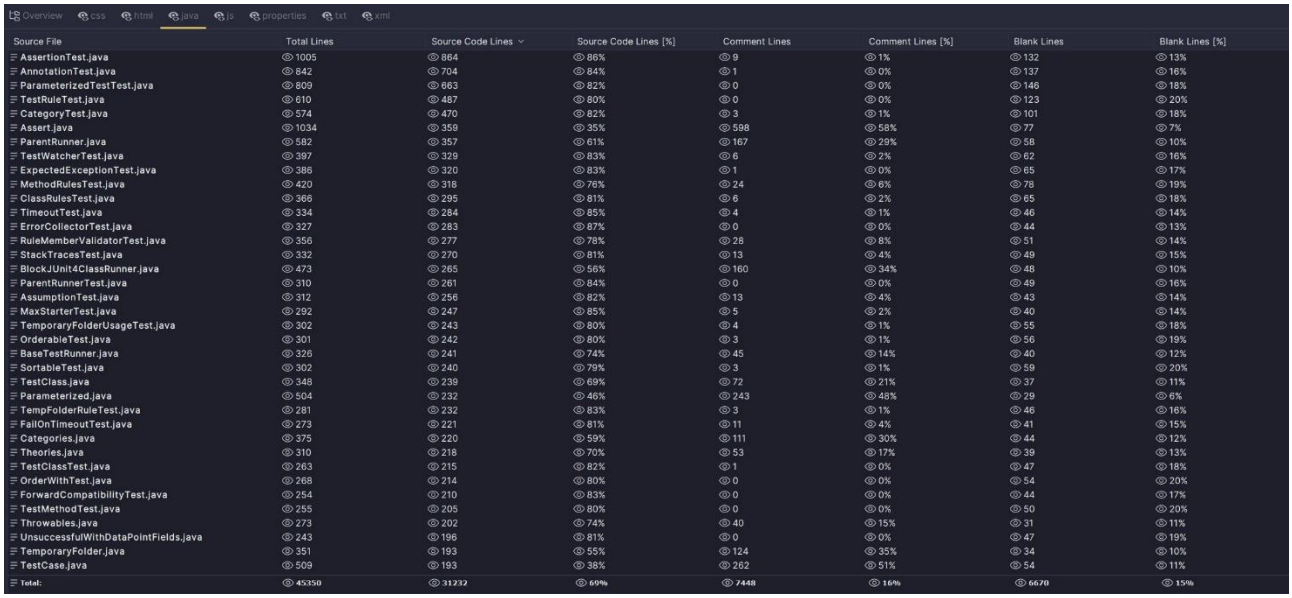
4.2.5 Junit4

JUnit 4 — це версія фреймворку для тестування в Java, випущена у 2006 році з підтримкою анотацій, що значно спростило написання тестів. Вона стала стандартом для модульного тестування у Java і використовувалася десятиліттями, перш ніж її поступово замінив JUnit 5.

Структурна характеристика проекту:

JUnit4 має класичну монолітну структуру та має зручну документацію, де описані основні положення. JUnit4 слідує «Google Java Style» та має хороший рівень документації коду. Кількість великих класів у проекті мінімальна, та в основному представляє собою тести. Проект дуже добре пропрацьован та має високий рівень покриття текстами. JUnit4 пройшов довгий процес еволюції та покращень, що дуже добре видно. Цей проект модна назвати самим суворим у дотримці принципам «Чистого коду». Загалом, JUnit4 це прекрасний проект з цікавою та довгою історією.

Роздивимось загальну статистику рядків коду у проекті(рис 4.65):



Overview | .css | .html | .java | .js | .properties | .txt | .xml

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
AssertionTest.java	1005	864	86%	9	1%	132	13%
AnnotationTest.java	842	704	84%	1	0%	137	16%
ParameterizedTestTest.java	809	663	82%	0	0%	146	18%
TestRuleTest.java	610	487	80%	0	0%	123	20%
CategoryTest.java	574	470	82%	3	1%	101	18%
Assert.java	1034	359	35%	598	58%	77	7%
ParentRunner.java	582	357	61%	167	29%	58	10%
TestWatcherTest.java	397	329	83%	6	2%	62	16%
ExpectedExceptionTest.java	386	320	83%	1	0%	65	17%
MethodRulesTest.java	420	318	76%	24	6%	78	19%
ClassRulesTest.java	366	295	81%	6	2%	65	18%
TimeoutTest.java	334	284	85%	4	1%	46	14%
ErrorCollectorTest.java	327	283	87%	0	0%	44	13%
RuleMemberValidatorTest.java	356	277	78%	28	8%	51	14%
StackTracesTest.java	332	270	81%	13	4%	49	15%
BlockJUnit4ClassRunner.java	473	265	56%	160	34%	48	10%
ParentRunnerTest.java	310	261	84%	0	0%	49	16%
AssumptionTest.java	312	256	82%	13	4%	43	14%
MaxStarterTest.java	292	247	85%	5	2%	40	14%
TemporaryFolderUsageTest.java	302	243	80%	4	1%	55	18%
OrderableTest.java	301	242	80%	3	1%	56	19%
BaseTestRunner.java	326	241	74%	45	14%	40	12%
SortableTest.java	302	240	79%	3	1%	59	20%
TestClass.java	348	239	69%	72	21%	37	11%
Parameterized.java	504	232	46%	243	48%	29	6%
TempFolderRuleTest.java	281	232	83%	3	1%	46	16%
FailOnTimeoutTest.java	273	221	81%	11	4%	41	15%
Categories.java	375	220	59%	111	30%	44	12%
Theories.java	310	218	70%	53	17%	39	13%
TestClassTest.java	263	215	82%	1	0%	47	18%
OrderWithTest.java	268	214	80%	0	0%	54	20%
ForwardCompatibilityTest.java	254	210	83%	0	0%	44	17%
TestMethodTest.java	255	205	80%	0	0%	50	20%
Throwables.java	273	202	74%	40	15%	31	11%
UnsuccessfulWithDataPointFields.java	243	196	81%	0	0%	47	19%
TemporaryFolder.java	351	193	55%	124	35%	34	10%
TestCase.java	509	193	38%	262	51%	54	11%
Total:	45350	31232	69%	7448	16%	6670	15%

Рисунок 4.65 – Загальна статистика проекту Junit

Рядки вихідного коду становлять близько 69% від загальної кількості рядків. Це один із найвищих відсотків серед розглянутих проектів, що може бути зумовлено більшою концентрацією на основній функціональності, оскільки JUnit 4 не мав настільки широкої модульної структури, як JUnit 5. JUnit 4 є найменшим за обсягом проектом серед аналізованих, із високою часткою вихідного коду (~69%). Це відображає його зосередженість на функціональності, без надлишкового супровідного коду. Така структура є типовою для більш старих і менш модульних проектів.

Проведемо сканування проекту за допомогою інструментів статичного аналізу (рис. 4.66 — 4.73):

Рисунок 4.66 – Результати сканування проекту Junit4 з InteliJ

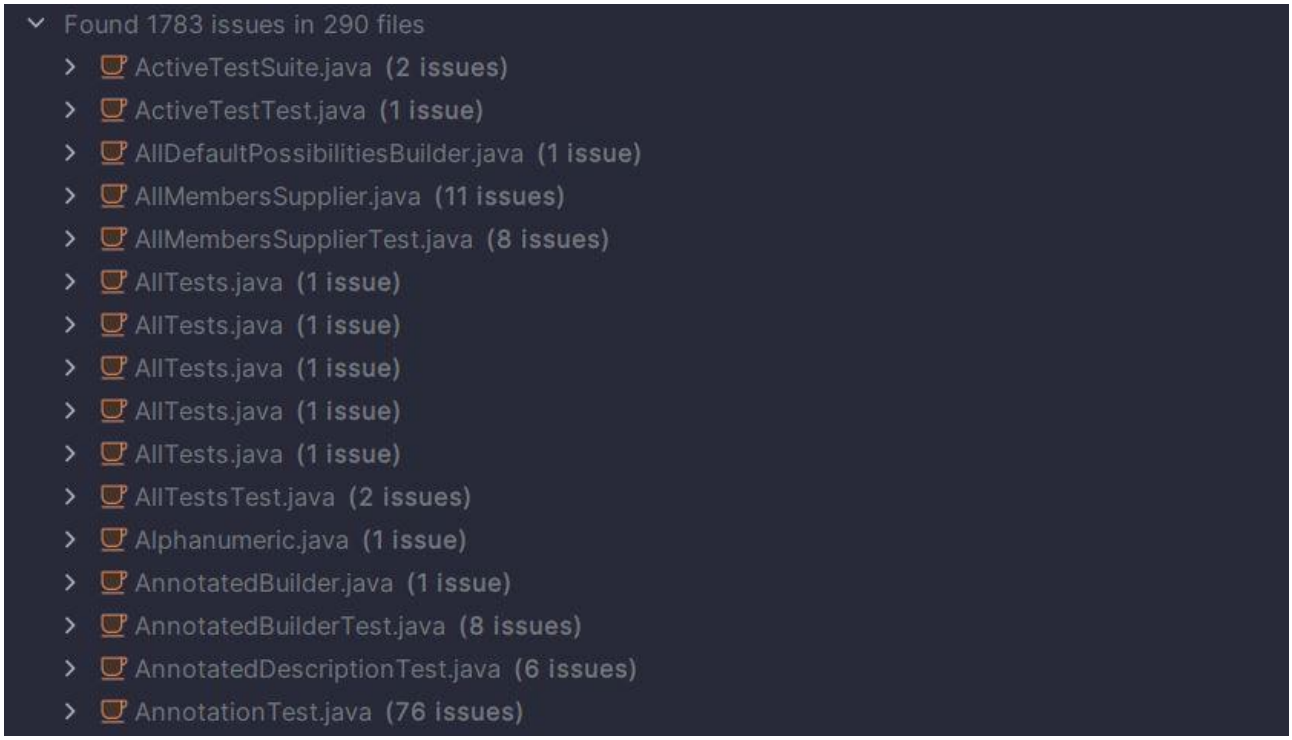


Рисунок 4.67 – Результати сканування проекту Junit4 з SonarLint

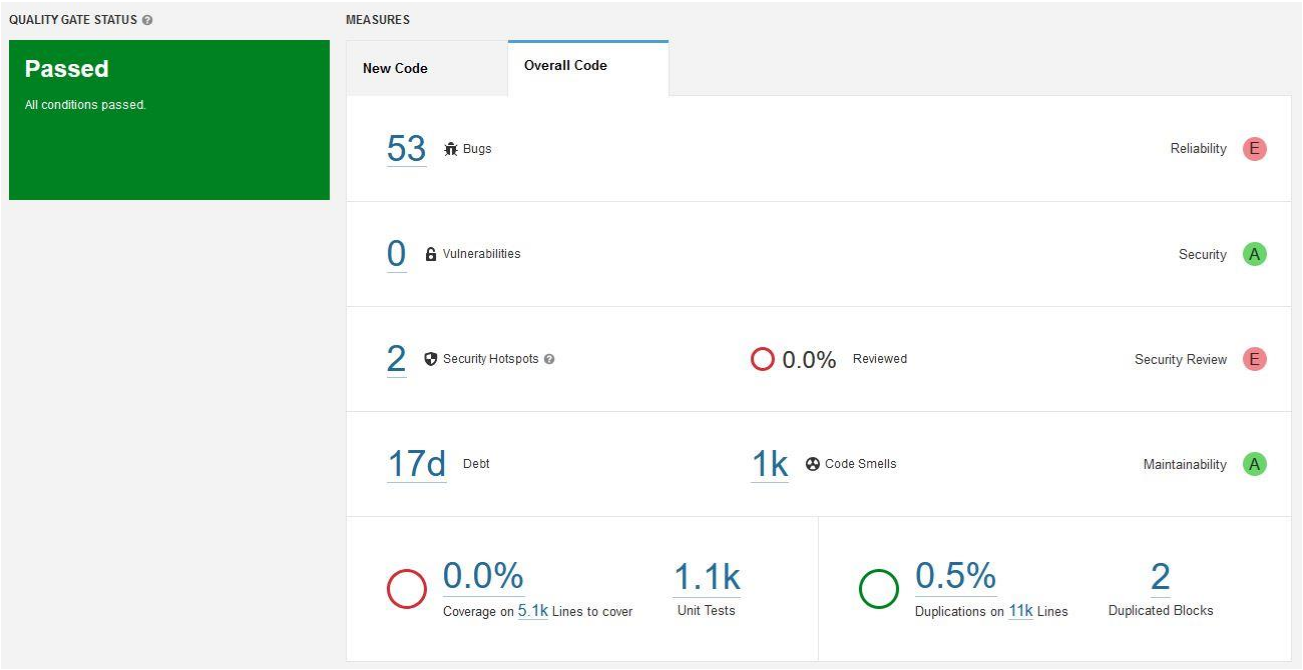


Рисунок 4.68 – Результати сканування проекту Junit4 з SonarQube

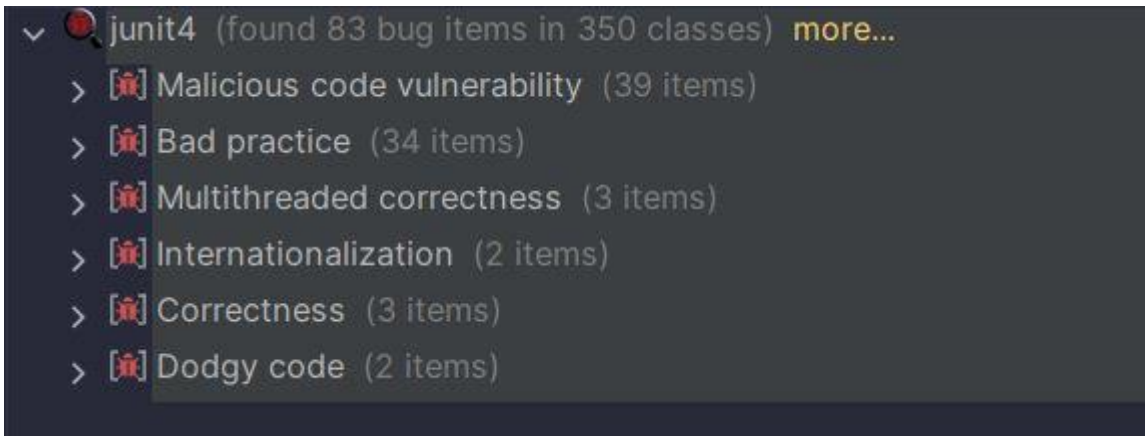


Рисунок 4.69 – Результати сканування проекту Junit4 з SpotBugs

Рисунок 4.70 – Результати сканування проекту Junit4 з Qodana

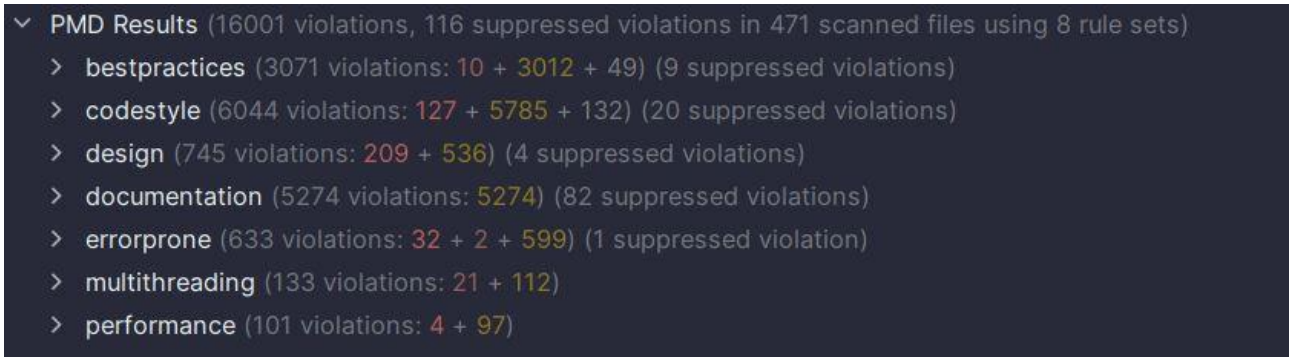


Рисунок 4.71 – Результати сканування проекту Junit4 з PMD

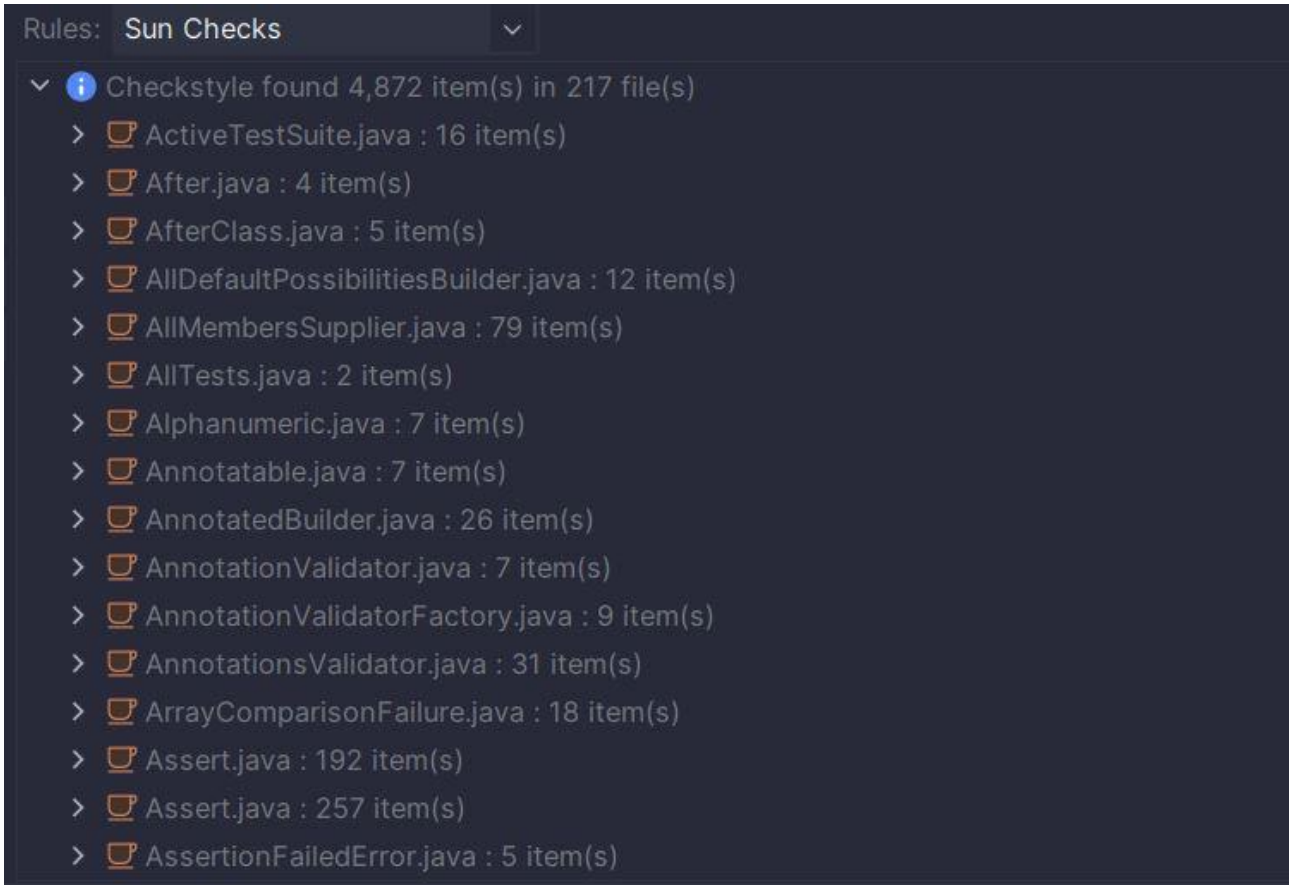


Рисунок 4.72 – Результати сканування проекту Junit4 з CheclStyle та посібником зі стилю від Sun

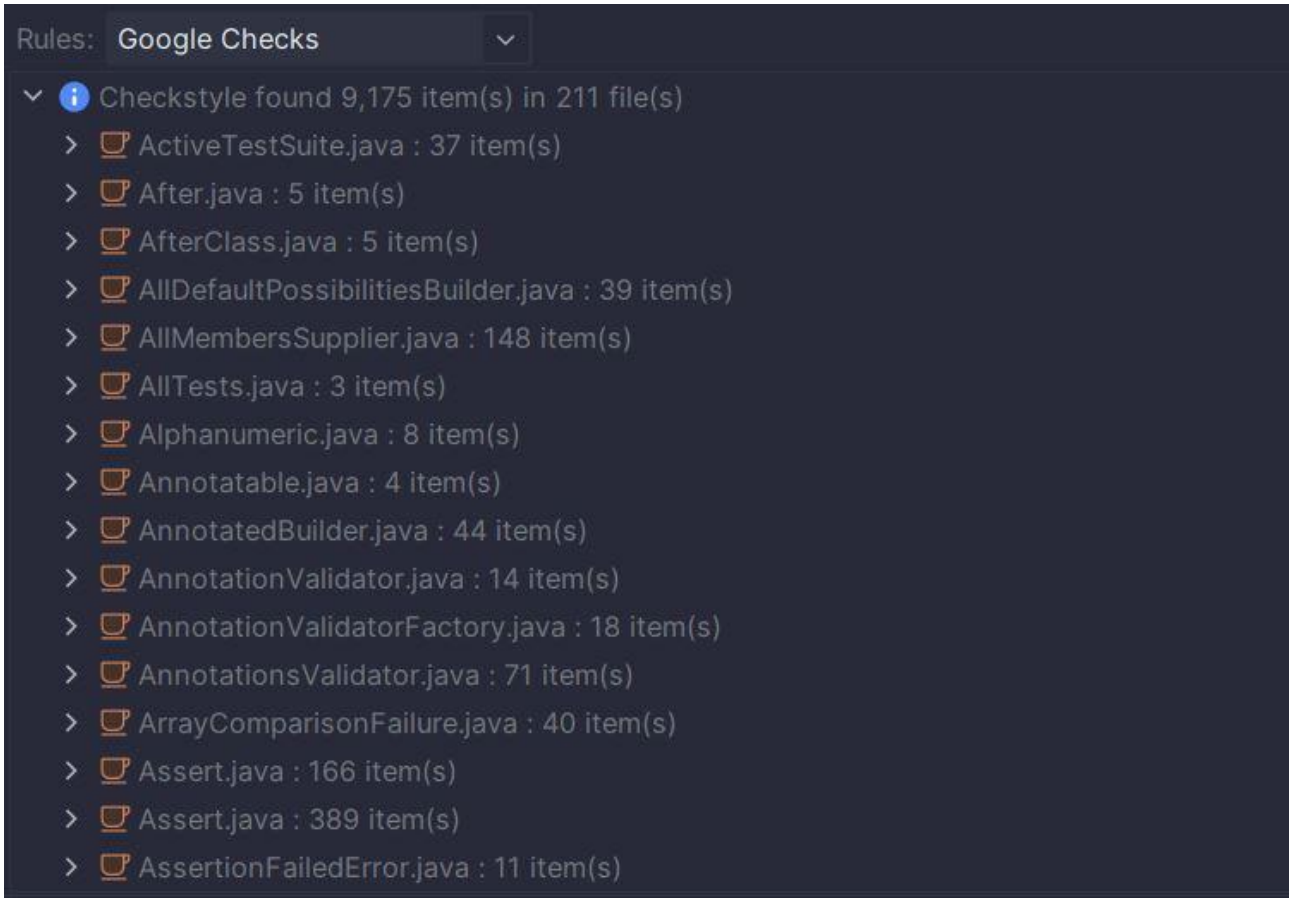


Рисунок 4.73 – Результати сканування проекту Junit4 з CheclStyle та посібником зі стилю від Google

Базуючись на даних від Statistic виберемо десять найбільших файлів та проведемо сканування кожного з них(табл. 4.5) та побудуємо графік порівняння кількості помилок від різних статичних аналізаторів(рис. 4.74):

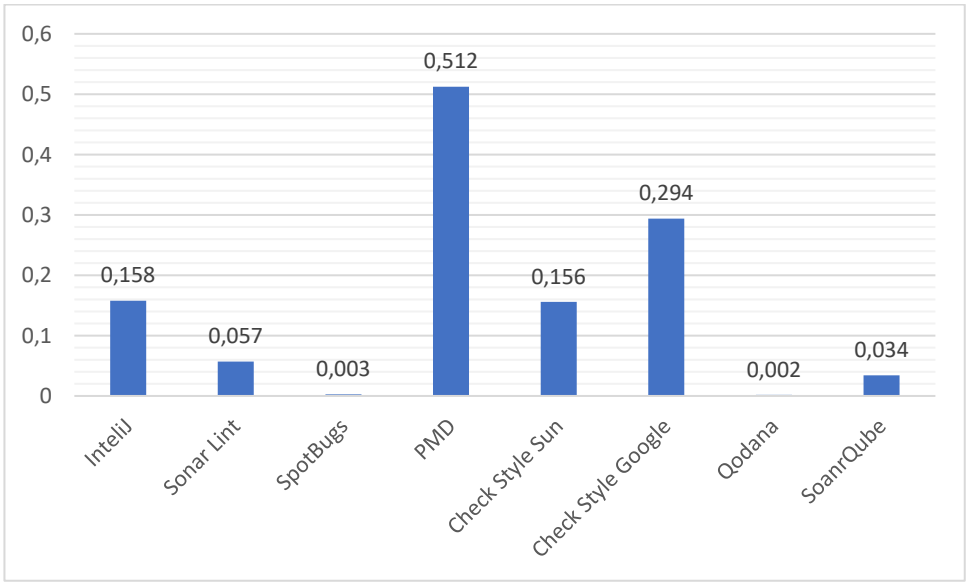


Рисунок 4.74 – Кількість помилок на рядок коду Junit4

У проекті JUnit4 щільність помилок (кількість виявлених помилок на один рядок коду) значно відрізняється між різними інструментами статичного аналізу, що свідчить про їхню спеціалізацію та ефективність у певних аспектах аналізу коду. PMD показав найвищу щільність помилок — 0.5123, демонструючи ефективність у виявленні як стилістичних, так і логічних проблем у коді. Check Style Google із показником 0.2938 виявляє чимало порушень, пов'язаних зі стилем оформлення коду, рекомендованим Google. IntelliJ також забезпечує високий рівень перевірки, виявляючи 0.1579 помилок на рядок, зосереджуючись на загальній якості та базових стилістичних проблемах. Check Style Sun має результат, близький до IntelliJ, але орієнтований на правила оформлення від Sun Microsystems. SonarLint більш вибірково виявляє потенційні помилки, зосереджуючись на проблемах, що впливають на продуктивність і безпеку. SonarQube(0.0342), SpotBugs (0.0027) і Qodana (0.0019) показали нижчі результати, що свідчить про їхню специфіку або менш суворий підхід до аналізу.

Найбільш суворим інструментом для виявлення помилок у проекті є PMD, тоді як Qodana і SpotBugs мають найменшу щільність, орієнтуючись на вузькоспеціалізовані аспекти аналізу.

Таблиця 4. – Результати сканування десяти найбільших файлів проекту Junit4

Назва файлу	Кількість рядків вихідного коду	InteliJ	Sonar Lint	Spot Bugs	PMD	Check Style Sun	Check Style Google
org.junit.Assert	359	137	8	0	301	257	389
org.junit.runners.ParentRunner	357	34	15	2	114	109	285
org.junit.runners.BlockJUnit4ClassRunner	265	32	16	1	93	114	216
junit.runner.BaseTestRunner	241	46	13	2	121	72	234
org.junit.runners.model.TestClass	239	32	16	3	118	122	205
org.junit.runners.Parameterized	232	23	12	3	101	80	207
org.junit.experimental.categories.Categories	220	15	8	1	122	118	248

org.junit.experimental.theories.Theories	218	9	5	1	87	98	191
org.junit.internal.Throwables	202	12	5	0	82	66	151
org.junit.rules.TemporaryFolder	193	31	10	1	73	67	186

Аналіз результатів інструментів статичного аналізу показав, що PMD є найефективнішим інструментом для виявлення проблем у коді, виявляючи 51.23% усіх порушень. Він найбільше відзначив проблеми у файлі org.junit.Assert, який є одним із найбільш проблемних. Інструменти Check Style Sun та Check Style Google виявили 15.59% та 29.37% порушень відповідно, причому Google вказує на більшу кількість проблем через більш суворі правила. Основні порушення стосуються файлів Assert, ParentRunner та BlockJUnit4ClassRunner. Інструмент IntelliJ виявив 15.79% порушень, зосереджених переважно у файлі org.junit.Assert, що свідчить про його ефективність у виявленні проблем якості коду. Sonar Lint і SonarQube виявили відповідно 5.71% і 3.41% порушень, що демонструє їхню корисність для ідентифікації специфічних проблем, хоча їх ефективність є нижчою порівняно з іншими інструментами. SpotBugs та Qodana показали низькі результати, виявивши лише 0.27% та 0.18% порушень відповідно, що свідчить про їхню вузьку спеціалізацію.

Отже, найбільша кількість проблем стосується файлів org.junit.Assert, ParentRunner та BlockJUnit4ClassRunner, які потребують першочергового рефакторингу. Основну увагу слід зосередити на стандартизації стилю оформлення коду, використовуючи PMD та Check Style як основні інструменти аналізу. Для виявлення специфічних багів варто оптимізувати налаштування SpotBugs та Qodana.

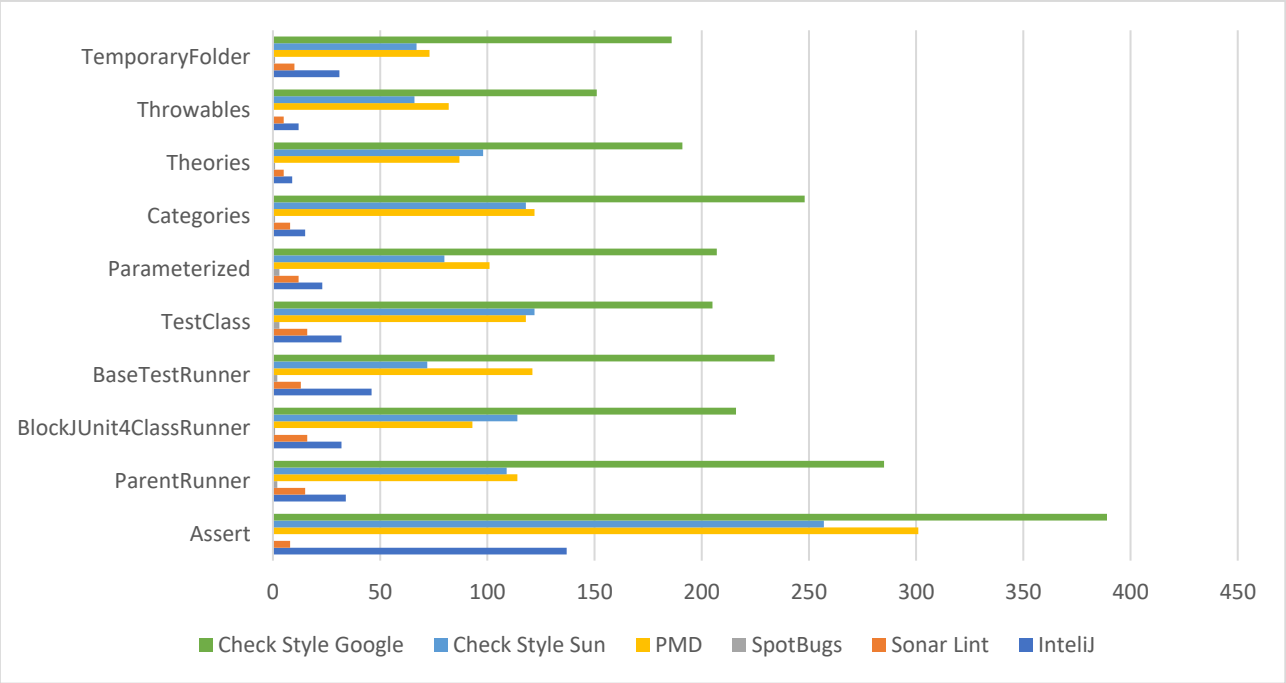


Рисунок 4.75 – Порівняння результатів сканування десяти найбільших файлів проекту Junit4

Загалом, проект JUnit4 демонструє помірний рівень якості коду з акцентом на стандарти оформлення.

Розглянемо Git статику:

Проект триває 8556 днів (понад 23 роки), але лише 1090 днів (12.74%) є активними. Це свідчить про періодичну роботу над проектом із чітко визначеними етапами активності. У репозиторії міститься 616 файлів, а загальна кількість рядків коду становить 55,597, з яких 150,024 рядки додано, а 94,427 рядків видалено. Така динаміка вказує на активне переписування та оптимізацію коду в процесі розробки. Загальна кількість комітів — 2513. У середньому, виконується 2.3 коміти за активний день або 0.3 коміти на всі дні проекту. Це може свідчити про стабільний, але не інтенсивний робочий процес. У проекті взяли участь 200 авторів, кожен із яких у середньому зробив 12.6 комітів. Це демонструє, що робота була рівномірно розподілена серед учасників.

Графік(рис. 4.76) підтверджує сплески активності в певні періоди, що, ймовірно, відповідає реалізації ключових оновлень або версій проекту.

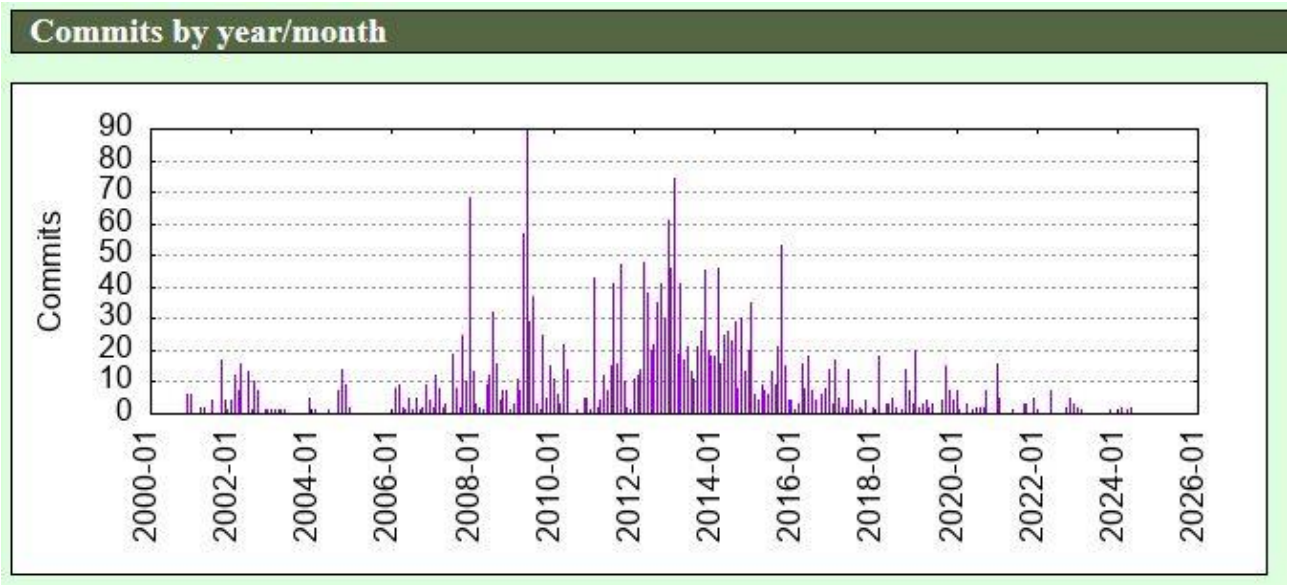
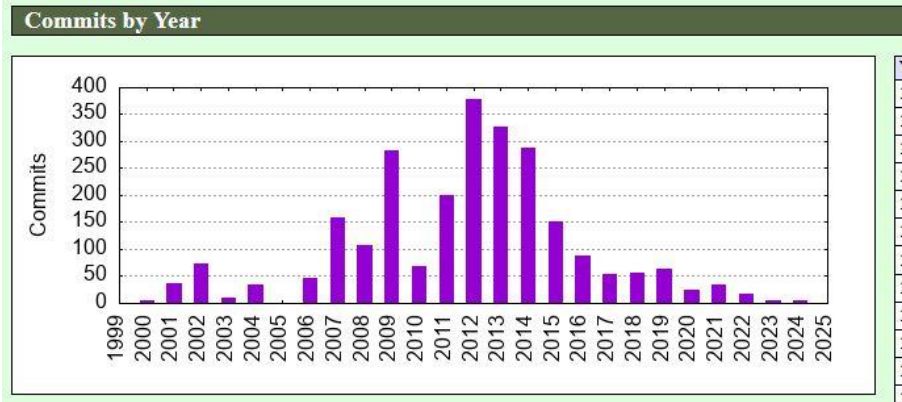


Рисунок 4.76 – Статистика коммітів за місяцями у проекті Junit4

Найвища активність спостерігається в період із 2006 до 2014 року, що відображено у статистиці комітів за роками(рис. 4.77). Після цього активність знижується, що свідчить про завершення основної стадії розробки та перехід до підтримки проекту.



Year	Commits (% of all)	Lines added	Lines removed
2024	5 (0.20%)	417	309
2023	4 (0.16%)	14	30
2022	18 (0.72%)	249	287
2021	33 (1.31%)	714	348
2020	25 (0.99%)	562	49
2019	64 (2.55%)	594	594
2018	56 (2.23%)	3650	1486
2017	54 (2.15%)	3456	489
2016	88 (3.50%)	2665	837
2015	151 (6.01%)	934	376
2014	289 (11.50%)	9339	8074
2013	326 (12.97%)	7088	2188
2012	378 (15.04%)	30264	24235
2011	200 (7.96%)	5038	1422
2010	68 (2.71%)	3525	2408
2009	283 (11.26%)	9000	3901
2008	107 (4.26%)	6130	2817
2007	159 (6.33%)	15035	9478
2006	47 (1.87%)	32322	29786
2004	34 (1.35%)	4957	1379
2003	10 (0.40%)	21	19
2002	72 (2.87%)	2279	1663
2001	36 (1.43%)	4165	2177
2000	6 (0.24%)	7606	75

Рисунок 4.77 – Статистика коммітів за роками у проекті Junit4

Проект JUnit4 мав активний період розробки в перші роки, після чого перейшов до етапу підтримки. Значна кількість учасників і регулярні зміни коду

свідчать про важливість і популярність проекту серед розробників.

Кількість файлів у проекті(рис. 4.78) поступово збільшувалася з моменту його створення. Основний приріст файлів припадає на період з 2000 по 2014 рік, після чого динаміка зростання сповільнилася. Окремі різкі спадання можуть вказувати на видалення або реорганізацію файлів у репозиторії.

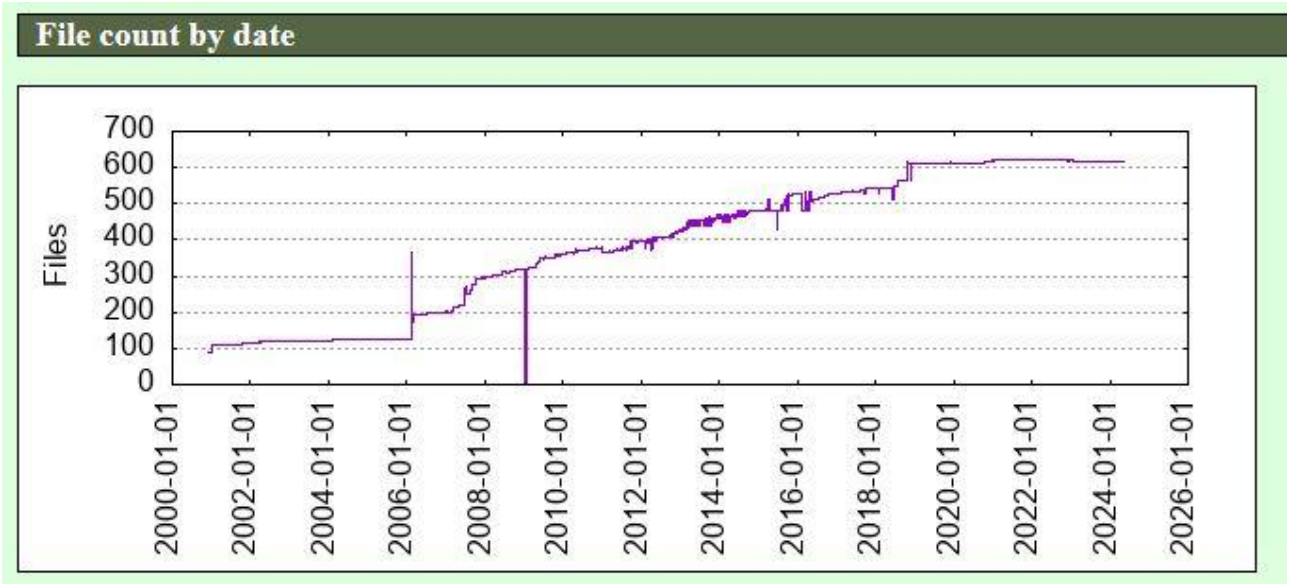


Рисунок 4.78 – Статистика кількості файлів у проекті Junit4

Кількість рядків коду(рис. 4.79) значно зросла в перші роки розробки, що характерно для активної фази розробки. Після 2014 року темпи зростання коду стали більш стабільними, що свідчить про завершення основної фази розробки та зосередження на підтримці і оптимізації проекту.

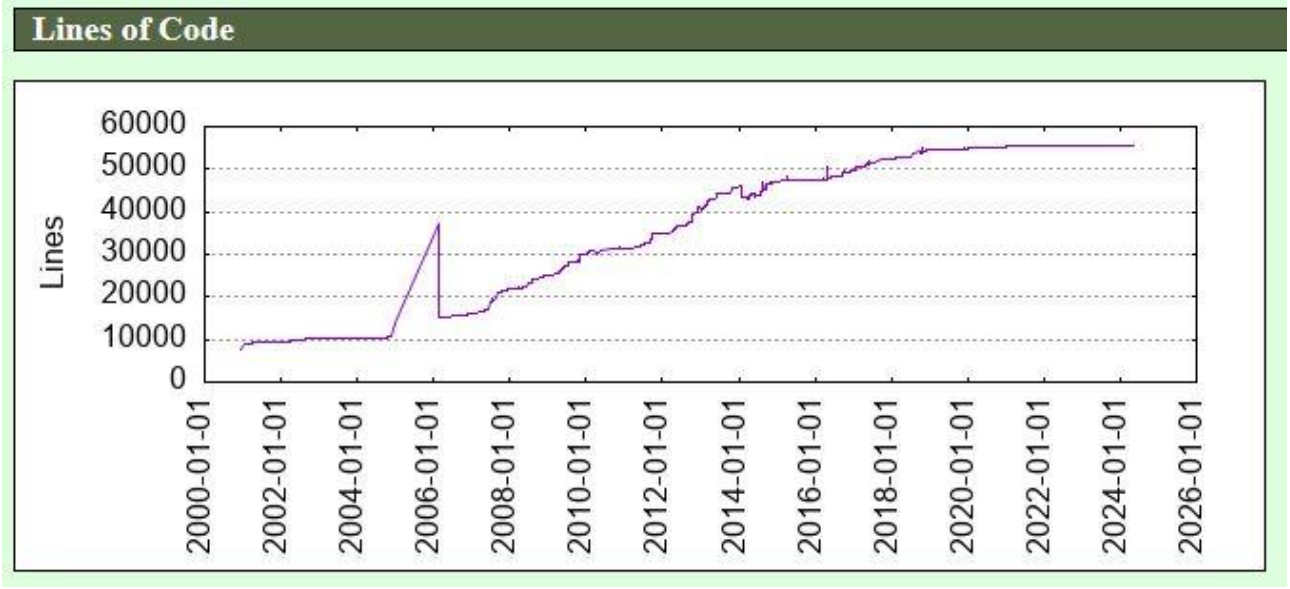


Рисунок 4.79 – Статистика росту рядків коду у проекті Junit4

Кількість рядків коду значно зросла в перші роки розробки, що характерно для активної фази розробки. Після 2014 року темпи зростання коду стали більш стабільними, що свідчить про завершення основної фази розробки та зосередження на підтримці і оптимізації проекту.

Проект JUnit4 розвивався активно в перші роки, з великим приростом файлів і рядків коду. У більш пізні періоди основні зусилля були спрямовані на підтримку, рефакторинг і оптимізацію існуючої кодової бази.

Розглянемо загальну інформацію про репозиторій:

Репозиторій має 2513 комітів, що свідчить про тривалий розвиток і значний обсяг роботи. Кількість пулл-реквестів становить 908, що вказує на залучення спільноти до внесення змін і покращень. Відкрито 1740 задач (issues), що свідчить про постійний процес обговорення, вдосконалення та виправлення. Загалом 146 контриб'юторів долучилися до роботи над проектом, що підтверджує активну участь відкритої спільноти. Проект має 8527 зірок (stars), що свідчить про популярність і визнання серед користувачів. Кількість форків — 3288, що вказує на активне використання проекту іншими розробниками та організаціями для власних потреб.

JUnit 4 має найнижчу активність серед порівнюваних репозиторіїв, що вказує на зниження інтересу через перехід спільноти до JUnit 5. Проект є зрілим і стабільним із мінімальними активними змінами, підтримуючи базовий рівень функціональності. Пулл-реквести затверджуються швидко з малою кількістю обговорень, що свідчить про впорядковані процеси. Довготривалі реквести трапляються рідко і зазвичай не потребують детальних обговорень, підтверджуючи завершений стан проекту. Додаткові дослідження активності проекту, відношення додавань та видалень у пулл-реквестах, відношення загальної кількості змін у коді до кількості коментарів від розробників, відношення часу, який пулл-реквест був відкритим до кількості коментарів від розробників в пулл-реквесті, та відношення часу, який пулл-реквест був відкритим до загальної кількості змін у коді в пулл-реквестах приведені нижче (див. Додаток В).

На графіку (рис. 4.80) видно періоди пікової активності, які, ймовірно, відповідають основним етапам розробки, випуску нових версій або внесення масштабних змін. Після 2015 року активність поступово зменшується, що характерно для зрілого проекту на стадії підтримки.

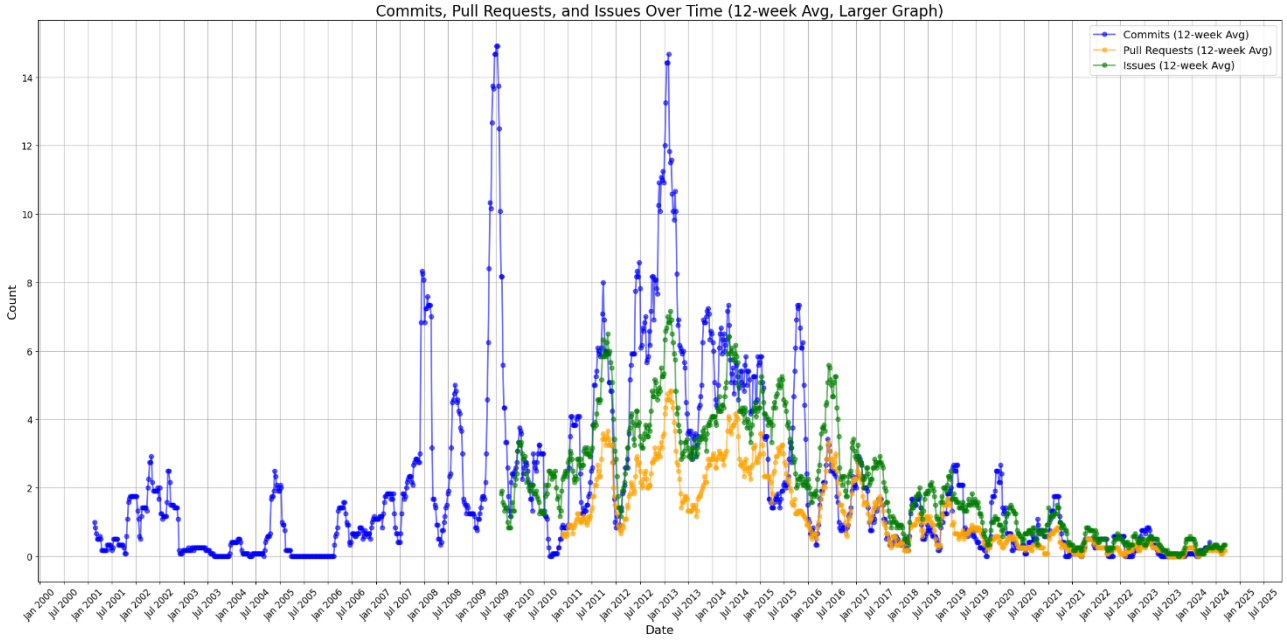


Рисунок 4.80 – Відношення коммітів/пулл-реквестів/задач репозиторію Junit4

Основна частина комітів і пулл-реквестів припадає на період активного розвитку, що характерно для таких проектів, як JUnit4. Зменшення активності у пізні періоди вказує на стабілізацію проекту. Висока кількість задач говорить про активний зворотний зв'язок від користувачів, що допомагало підтримувати якість і функціональність коду. Популярність проекту підтримується завдяки активній спільноті та стабільному внесенню змін у минулому. Враховуючи зменшення активності, проект, ймовірно, перебуває на стадії підтримки, де основний акцент зроблено на виправленні помилок і незначних покращеннях.

Репозиторій JUnit4 демонструє високу активність у минулому, значний внесок від спільноти та популярність у середовищі розробників. Проект пройшов етап активного розвитку і зараз, ймовірно, перебуває на стадії підтримки, забезпечуючи стабільність та якість для своїх користувачів.

4.3 Результати експерименту

Аналізуючи п’ять Java-проектів — JUnit 4, JUnit 5, Mockito, Apache Dubbo

та Google Guava, можна побачити, як їхній історичний контекст і цілі вплинули на розмір, структуру та пріоритети кожного з них.

JUnit 4: Легкість і основа тестування

- Проєкт компактний (31,232 рядки вихідного коду, 69% функціонального коду), оскільки фокусувався лише на базових можливостях тестування без додаткових модулів чи інструментів.
- Його структура ідеально підходить для епохи, коли модульність і розширюваність не були головними пріоритетами.

JUnit 5: Еволюція та модульність

- Проект значно масштабніший (101,178 рядків коду), але частка функціонального коду зменшилася до ~58%. Це пояснюється більшою увагою до тестування, документації та зворотної сумісності.
- JUnit 5 відображає сучасні потреби розробників, балансуючи між функціональністю, тестами й розширеннями.

Mockito: Орієнтація на тестування через моки

- Структура проекту (62,427 рядків вихідного коду, ~64%) збалансована між функціональністю та супровідним кодом, що є типовим для бібліотек, орієнтованих на тестування.
- Mockito продемонстрував ефективність вузькоспеціалізованих інструментів, забезпечуючи зручність розробників і підвищуючи якість тестування.

Apache Dubbo: Високопродуктивна платформа мікросервісів

- Проект великий (267,379 рядків вихідного коду, ~64%) з помітною часткою конфігурацій і тестів, оскільки Dubbo працює в складних розподілених середовищах.
- Як інструмент для високопродуктивної роботи в реальних умовах, Dubbo підкреслює важливість тестування і документації для забезпечення стабільності.

Google Guava: Інструменти для розробників

- Guava має найбільший розмір (512,582 рядки вихідного коду, ~65%), оскільки охоплює багато аспектів розробки. Це відображає її універсальний характер і широкий спектр використання.
- Guava є символом інженерного підходу Google, зосередженого на масштабованих і повторно використовуваних рішеннях.

Наведено таблицю(4.6) для більш детального порівняння:

Таблиця 4.6 – Порівняння об’єму коду між проектами

Метрика	JUnit 4	Mockito	JUnit 5	Apache Dubbo	Google Guava
Загальна кількість рядків	45,350	97,261	172,995	418,717	788,282
Рядки вихідного коду	31,232	62,427	101,178	267,379	512,582
Частка вихідного коду	~69%	~64%	~58%	~64%	~65%

Проекти Dubbo, Guava, JUnit5, Mockito та JUnit4 відіграють ключову роль у розвитку Java-екосистеми, кожен із яких має свою історію, вплив та унікальні аспекти застосування. Їхній історичний бекграунд та сучасний стан відображають еволюцію підходів до програмування, тестування, масштабування та організації коду.

Аналіз активності проектів демонструє їхню значущість: великі обсяги коммітів, активні контриб'ютори та стабільна кількість пулл-реквестів підтверджують попит. Статичний аналіз цих проектів показує високий рівень якості коду, що є наслідком активної участі спільноти та використання сучасних стандартів розробки.

Код та структура цих проектів є прикладом, що втілює найкращі практики програмування. Їхній впорядкований та підтримуваний код сприяє легкості внесення змін, забезпеченню стабільності й високої продуктивності.

Висновки до розділу 4

У цьому розділі було виконане детальне дослідження, під час якого використовувалися різноманітні інструменти для збору та обробки інформації, такі як аналіз статистичних даних, автоматизовані інструменти оцінки та порівняльний аналіз. Це дозволило отримати всебічне розуміння предметної області, виявити ключові закономірності та зібрати достатню кількість даних для подальшого аналізу й формулювання обґрунтованих висновків. Розглянуті проекти є яскравими прикладами успішних проектів з відкритим вихідним кодом. Вони сформували основи для сучасної розробки на Java, забезпечуючи інструменти для ефективного тестування, модульності, масштабування та організації коду. Їхній вплив поширюється на численні проекти, а їхня історія є свідченням важливості спільноти у створенні програмного забезпечення, яке адаптується до нових викликів та стандартів. Ці проекти ілюструють еволюцію Java-екосистеми, перехід від простоти до більш складних і модульних архітектур, а також зростання ролі тестування й супровідного коду. Кожен із них відповідає конкретним потребам свого часу і продовжує впливати на сучасну розробку.

ВИСНОВКИ ТА РЕКОМЕДАЦІЇ

З наукової точки зору, питання підтримуваності та масштабованості коду привертало дедалі більше уваги з початку розвитку комп'ютерної науки. Наукові дослідження в програмній інженерії, особливо у сфері аналізу програм та оцінювання їхніх метрик, акцентують увагу на тому, як структура і читабельність коду впливають на його легкість у підтримці. Книга Роберта Мартіна «Чистий код»[1], стала знаковою в історії програмної інженерії. Автор книги пропонує не тільки новаторські концепції, скільки консолідує найкращі практики, здобуті завдяки десятиліттям спроб і помилок у цій галузі. «Чистий код» містить уроки, які допомагають уникати типових помилок у розробці, супроводі та масштабуванні програмного забезпечення. Проблеми супроводжуваності та масштабованості залишаються фундаментальними викликами програмної інженерії, особливо з урахуванням постійного зростання складності й обсягу сучасних програмних систем.

У цій науковій роботі було розглянуто 5 різних за структурою, розміром та призначенням проектів: JUnit4, JUnit5, Mockito, Guava та Dubbo, вони є важливими складовими екосистеми Java, кожен з яких відіграє свою унікальну роль у розвитку сучасного програмного забезпечення. Ця робота була спрямована на дослідження якості коду різноманітних проектів для пошуків можливих кореляцій між явними та неявними метриками, які можуть впливати на його якість та у результаті, на його підтримуваність та масштабованість. Кожен з проектів зробив еволюцію своїй сфері діяльності. JUnit4 став основою для впровадження культури автоматизованого тестування, тоді як JUnit5, будучи його наступником, враховує сучасні вимоги, зокрема підтримку Java 9+ і розширену модульність. Mockito забезпечує гнучкість у тестуванні завдяки можливості імітації об'єктів, що робить його ключовим інструментом для створення юніт-тестів. Guava, бібліотека від Google, значно розширила можливості стандартної бібліотеки Java, пропонуючи високоякісний і стабільний код, який інтегрується у велику кількість проектів. Dubbo, орієнтований на мікросервісну архітектуру, став незамінним інструментом для масштабованих

корпоративних рішень, активно підтримується спільнотою та відповідає сучасним вимогам до комунікації між сервісами. Аналіз активності цих проектів свідчить про їхній розвиток і стабілізацію. Наприклад, JUnit5 перейняв роль JUnit4, продовжуючи його еволюцію, тоді як активність у Guava знизилася через досягнення зрілості. Dubbo демонструє значний інтерес серед корпоративних розробників завдяки актуальності мікросервісів. Результати статичного аналізу підтверджують високу якість коду в усіх проектах, а велика кількість контриб'юторів і активність пулл-реквестів свідчать про їх популярність і затребуваність.

Загалом, усі проекти є прикладом успішної реалізації відкритого програмного забезпечення. Вони вплинули на сучасну розробку Java, формуючи ефективні підходи до тестування, масштабування, та організації коду. Їхня стійкість і актуальність відображають важливість підтримки з боку спільноти розробників та готовність до адаптації під нові виклики. Ці проекти пройшли довгий шлях змін та розвитку у напрямку покращення якості коду і у результаті проведеного дослідження можна побачити як рівень якості коду підвищив рівень підтримки та масштабування. Що свідчить про можливо непряму, але кореляцію між цими явищами.

Дуже важко простежити прямий вплив принципів чистого коду на якість та простоту підтримки коду. Так, подібний зв’язок між якістю та підтримкою коду є доволі логічним, але має під собою дуже малу доказову та фактичну базу. До того ж, у подібному досліджені важко виокремити вплив "конфаундерів", гнучких змінних, не прямих кореляцій та інше від реальних статичних даних.

Але основна проблематика цього дослідження і полягає в тому, що таких досліджень загалом дуже мало і дуже багато суджень є суто суб’єктивними та ненадійними. І ця робота пропонує методи та засоби загального дослідження подібної теми, розвиток та деталізація яких, з плином часу, надати більш надійні та точні данні, які допоможуть розробникам створювати програмний код, який буде легко та зручно підтримувати та масштабувати.

Запропонована методологія є складається не тільки з пласких чисельних показників помилок у коді, але й враховує сторонні фактори, як рівень підтримки команд розробки, історичний контекст та сферу діяльності проектів. Нажаль, проекти не можливо порівнювати один до одного. Бо програмне забезпечення не має еталону, тому що це в першу чергу про людей, їх помилки, думки та ідеї. Однак дотримання певного набору правил і найкращих практик допоможе індустрії інформаційних технологій продовжувати свій розвиток, враховуючи помилки минулого, та рухатися вперед до нових технологічних досягнень.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Мартін, Роберт. Чистий код: створення, аналіз і рефакторинг / Роберт Мартін. — Київ: Діалектика, 2019. — 464 с.
2. Макконнелл, Стів. Code Complete: A Practical Handbook of Software Construction / Стів Макконнелл. — 2-е вид. — Редмонд: Microsoft Press, 2004. — 960 с.
3. Мартін, Роберт. Чиста архітектура: мистецтво дизайну програмного забезпечення / Роберт Мартін. — Київ: Діалектика, 2021. — 432 с.
4. Лутц, Марк. Python. Біблія користувача: підручник з Python / Марк Лутц. — Київ: Діалектика, 2011. — 1216 с.
5. Лутц, Марк. Програмування на Python: навчальний курс / Марк Лутц. — Київ: Вільямс, 2010. — 624 с.
6. Сандерс, Джон. Рефакторинг: поліпшення якості існуючого коду / Джон Сандерс. — Київ: Видавничий дім «СофтПрес», 2020. — 368 с.
7. Фаулер, Мартін. Архітектура корпоративних додатків / Мартін Фаулер. — Київ: Видавництво «Діалектика», 2018. — 552 с.
8. Beck K. Agile Software Development: Principles, Patterns, and Practices / K. Beck. – Upper Saddle River : Pearson Education, 2002. – 536 с.
9. Beck K. Implementation Patterns / K. Beck. – Boston : Addison-Wesley Professional, 2007. – 194 с.
10. Knuth D. E. Literate programming / D. E. Knuth. – Stanford : Center for the Study of Language and Information, 1992. – 368 с.
11. Dahl O. J. Structured Programming / O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. – London : Academic Press, 1972. – 297 с.
12. Hunt A. The Pragmatic Programmer / A. Hunt, D. Thomas. – Boston : Addison-Wesley, 2000. – 320 с.
13. Kernighan B. W. The Elements of Programming Style / B. W. Kernighan, P. J. Plauger. – 2-ге вид. – New York : McGraw-Hill, 1978. – 178 с.
14. Fowler M. Refactoring: Improving the Design of Existing Code / M. Fowler. – Boston : Addison-Wesley, 1999. – 464 с.

15. Li Z. A systematic mapping study on technical debt and its management / Z. Li, P. Liang, P. Avgeriou // Journal of Systems and Software. – 2018. – Т. 144, № 3. – С. 190–219.
16. Bavota, G., Russo, B. A large-scale empirical study on self-admitted technical debt // Empirical Software Engineering. – 2019. – Т. 24. – с. 1412-1456.
17. Besker, T., Martini, A., Bosch, J. Software developer productivity loss due to technical debt — A replication and extension study examining developers' development wor // Journal of Systems and Software. – 2019. – Т. 156. – с. 41 - 61.
18. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs // Cognitive Psychology. – 1987. – Т. 19, №. 3. – с. 295-341.
19. Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality [Електронний ресурс] / Justus Bogner [та ін.] // 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Гамбург, 25–26 берез. 2019 р. – [Б. м.], 2019. – С. 187–195. – Режим доступу: https://doi.ieeecomputersociety.org/10.1109/ICSA-C.2019.00041.
20. Taibi, D., Lenarduzzi, V., Pahl, C. Architectural Patterns for Microservices: A Systematic Mapping Study // IEEE Software. – 2018. – Т. 38, №. 3. – с. 52-60.
21. Cecchet, E., Marguerite, J., Zwaenepoel, W. Performance and scalability of EJB applications // SIGPLAN Not. – 2002. – Т. 37, № 11. – С. 246-261. – DOI: https://doi.org/10.1145/583854.582443.
22. Rios, N., Mendonça, M. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners // Information and Software Technology. – 2018. – Т. 102. – с. 117-145.
23. Khan, S. M. Measuring the Structure in Software Quality Management. – 2023. – 156 с.
24. Spinellis D. Code Quality: The Open Source Perspective. Boston : Addison-Wesley Professional, 2006. – 480 с.

25. Ogheneovo, E. E. On the Relationship between Software Complexity and Maintenance Costs // Journal of Computer and Communications. – 2014. – Т. 2, № 14. – С. 1-6. – DOI: 10.4236/jcc.2014.214001.
26. Newman, S. Building Microservices. – O'Reilly Media, Inc., 2021. – 616 с.
27. Брукс, Ф. П. Міфічний людино-місяць / Ф. П. Брукс. – 1975. – 322 с.
28. Сучасні інформаційні та комунікаційні технології на транспорті, в промисловості і освіті: Тези XVIII Міжнародної науково-практичної конференції (Дніпро, 12-13 грудня 2024 р.). – Д.: УДУНТ, 2024. – 191 с.
29. Наука і сталий розвиток транспорту 2024. Т.ІІ: зб. тез доп. Всеукр. наук.-техн. конф. студентів і молодих учених, Дніпро, 27 листоп. 2024р.-Дніпро: УДУНТ, 2024.- 204 с.
30. Монографія
31. Statistic Plugin [Електронний ресурс] – Режим доступу: https://plugins.jetbrains.com/plugin/4509-statistic
32. Mourya, S., Singh, P., Singh, V. An Insight into Code Smell Detection Tool // Lecture Notes in Computer Science. – 2024. – DOI: 10.1007/978-3-031-55048-5_17.
33. Charoenwet, W., Thongtanunam, P., Pham, V.-T., Treude, C. An Empirical Study of Static Analysis Tools for Secure Code Review. – 2024. – DOI: 10.48550/arXiv.2407.12241.
34. Maggi, K., Verdecchia, R., Scommegna, L., Vicario, E. Evolution of Code Technical Debt in Microservices Architectures // Journal of Systems and Software. – 2024. – Т. 222. – С. 112301. – DOI: 10.1016/j.jss.2024.112301.
35. Verdecchia, R., Maggi, K., Scommegna, L., Vicario, E. Technical Debt in Microservices: A Mixed-Method Case Study. – 2024.
36. Katin, A., Lenarduzzi, V., Taibi, D., Mandić, V. On the Technical Debt Prioritization and Cost Estimation with SonarQube Tool // Lecture Notes in Computer Science. – 2022. – DOI: 10.1007/978-3-030-97947-8_40.

37. Genfer, P., Zdun, U. Exploring Architectural Evolution in Microservice Systems Using Repository Mining Techniques and Static Code Analysis // Lecture Notes in Computer Science. – 2024. – DOI: 10.1007/978-3-031-70797-1_10
38. Tupsakhare, P. Monolithic vs. Microservices: Analyzing Architectural Paradigms in Modern Software Development // The Journal of Scientific and Engineering Research. – 2019. – С. 299-303. – DOI: 10.5281/zenodo.13918620.
39. Ozkan, N., Kolukısa, A. Investigating Causes of Scalability Challenges in Agile Software Development from a Design Perspective. – 2019. – DOI: 10.1109/UBMYK48245.2019.8965633.
40. Altman, E., Arnold, M., Bordawekar, R., Delmonico, R. M., Mitchell, N., Sweeney, P. F. Observations on tuning a Java enterprise application for performance and scalability // IBM Journal of Research and Development. – 2010. – Т. 54, № 5. – С. 2:1-2:12. – DOI: 10.1147/JRD.2010.2057090.
41. Sharma, T., Spinellis, D. A Survey on Software Smells // Journal of Systems and Software. – 2017. – Т. 138. – DOI: 10.1016/j.jss.2017.12.034.
42. Sharma, T., Spinellis, D. Do We Need Improved Code Quality Metrics? – 2020. – DOI: 10.48550/arXiv.2012.12324.
43. The Joel Test [Електронний ресурс] – Режим доступу: https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/
44. Antinyan, V., Staron, M., Sandberg, A. Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time [Електронний ресурс] // Empirical Software Engineering. – 2017. – Т. 22, № 6. – С. 3057–3087. – Режим доступу: https://doi.org/10.1007/s10664-017-9508-2.
45. Basutakara, B. S., Jayanthi, P. N. A Review of Static Code Analysis Methods for Detecting Security Flaws [Електронний ресурс] // Journal of University of Shanghai for Science and Technology. – 2021. – Т. 23, № 06. – С. 647–653. – Режим доступу: https://doi.org/10.51201/jusst/21/05320.

46. Code Standardization and Risk Mitigation in Software Development [Електронний ресурс] – Режим доступу: https://www.sonarsource.com/learn/code-standardization-and-risk-mitigation-in-software-development/
47. Why You Should Care About Pull Request Size + Best Practices [Електронний ресурс] – Режим доступу: https://brainhub.eu/library/pull-request-size-best-practices
48. It's probably time to stop recommending Clean Code [Електронний ресурс] – Режим доступу: https://qntm.org/clean
49. "Clean" Code, Horrible Performance[Електронний ресурс] – Режим доступу: https://www.computerenhance.com/p/clean-code-horrible-performance
50. When ‘Clean Code’ Hampers Application Performance [Електронний ресурс] – Режим доступу: https://thenewstack.io/when-clean-code-hampers-application-performance/
51. When clean code becomes harmful Performance [Електронний ресурс] – Режим доступу: https://dev.to/remojansen/clean-code-is-considered-harmful-3lh0
52. Clean Code Is Slow, but You Need It Anyway… [Електронний ресурс] – Режим доступу: https://betterprogramming.pub/clean-code-is-slow-but-you-still-need-it-anyway-ffcac6973c93
53. Chapin, N. Do we know what preventive maintenance is? // Proceedings 2000 International Conference on Software Maintenance. – San Jose, CA, USA, 2000. – с. 15–17. – DOI: 10.1109/ICSM.2000.882970.

ДОДАТКИ