МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 11.00.00.000 ПЗ

Група ІПм-23-1

Брилковський Тарас

2025

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Брилковський Тарас Володимирович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Застосування методів та моделей Data Mining для аналізу репозиторіїв програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Брилковський Т.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Вовк Роман Богданович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**

доц. В.В. Бандура

“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Брилковському Тарасу Володимировичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи “ Застосування методів та моделей Data Mining для аналізу репозиторіїв програмного забезпечення ”

керівник проекту (роботи) Вовк Роман Богданович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

2. Строк подання студентом проекту (роботи) 24 січня 2025 р.

3. Вихідні дані до проекту (роботи) Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних технологій Data Mining

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз предметної області використання майнінгу для сховищ програмного забезпечення
2. Моделі та методи функціонування репозиторіїв вихідного коду для процесів майнінгу
3. Дослідження та опис показників якості програмного забезпечення та KPI
4. Імплементація методів data mining для аналізу якості репозиторіїв програмного забезпечення

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Локальна система контролю версій (рис. 1.1)
2. Централізована система контролю версій (рис. 1.2)
3. Розподілена система контролю версій (рис. 1.3)
4. Головна сторінка GitHub (рис. 1.4)
5. Менеджер сховищ GitLab (рис. 1.5)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Аналіз предметної області використання майнінгу для сховищ програмного забезпечення	15.10.2024	виконано
4	Моделі та методи функціонування репозиторіїв вихідного коду для процесів майнінгу	08.11.2024	виконано
5	Дослідження та опис показників якості програмного забезпечення та KPI	20.11.2024	виконано
6	Імплементація методів data mining для аналізу якості репозиторіїв програмного забезпечення	20.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	27.01.2025	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 86 с., 23 рис., 4 табл., 50 джерел.

Тема: Застосування методів та моделей Data Mining для аналізу репозиторіїв програмного забезпечення

Об'єкт дослідження: процеси функціонування репозиторіїв програмного забезпечення.

Мета роботи: розробити методику та інструментальні засоби для аналізу якості репозиторіїв програмного забезпечення на основі методів Data Mining.

Предмет дослідження: методи та моделі Data Mining, що застосовуються для аналізу даних репозиторіїв програмного забезпечення з метою оцінки їхньої якості.

Результати дослідження

Розроблено методику аналізу якості репозиторіїв програмного забезпечення з використанням сучасних алгоритмів Data Mining та запропоновано архітектуру програмного інструменту, яка враховує специфіку роботи з даними репозиторіїв.

Висновок

Отримані результати демонструють можливість ефективного використання методів Data Mining для покращення процесів аналізу якості програмного забезпечення, забезпечуючи тим самим підвищення продуктивності розробки та її кінцевої якості.

РЕПОЗИТОРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ, DATA MINING, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МЕТРИКИ ЕФЕКТИВНОСТІ, АНАЛІЗ ДАНИХ, АВТОМАТИЗАЦІЯ РОЗРОБКИ.

ABSTRACT

Master Thesis: 86 pp., 23 fig., 4 tab., 50 sources.

Thesis Subject: Application of Data Mining methods and models for analyzing software repositories

Object of research: processes of functioning of software repositories.

Purpose of work: to develop a methodology and tools for analyzing the quality of software repositories based on Data Mining methods.

Subject of research: Data Mining methods and models used for analyzing software data repositories with assessing their quality.

Research results

A methodology for analyzing the quality of software repositories using modern Data Mining algorithms has been developed and a structure of a software tool has been proposed that takes into account the specifics of working with repository data.

Conclusion

The results obtained demonstrate the possibility of effective use of Data Mining methods to improve software quality analysis processes, thereby ensuring increased development productivity and its final quality.

SOFTWARE REPOSITORIES, VERSION CONTROL SYSTEMS, DATA MINING, SOFTWARE QUALITY, PERFORMANCE METRICS, DATA ANALYSIS, DEVELOPMENT AUTOMATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MSR – Mining software repositories.

KPI – Key performance indicator.

VCS – Version control system.

SCM – Source code management.

CVS – Concurrent Versions System.

SVN – Apache Subversion.

ВСТУП

Актуальність теми.

Сучасна індустрія програмного забезпечення характеризується стрімким зростанням обсягів даних, збережених у репозиторіях програмного забезпечення (Software Repositories). Ці сховища містять не лише вихідний код, але й інформацію про історію змін, команди розробників, журнали помилок, запити на нові функції та інші важливі аспекти розробки програмного забезпечення. Така різнорідність і багатство даних створюють унікальні можливості для застосування методів Data Mining, які можуть ефективно обробляти великі обсяги інформації та виявляти приховані закономірності.

Ключовими викликами є забезпечення якості програмного забезпечення, що є критичним фактором для успішної комерціалізації та підтримки продукту. Традиційні методи оцінки якості, засновані на ручному аналізі або використанні окремих метрик, часто не забезпечують повного розуміння стану проєктів, особливо в умовах масштабної розробки.

Актуальність дослідження також підтверджується сучасними тенденціями в автоматизації розробки програмного забезпечення, що включає інтеграцію з системами контролю версій (VCS), DevOps-підходи, а також використання штучного інтелекту для прогнозування та моніторингу якості.

Крім того, велика кількість відкритих репозиторіїв на таких платформах, як GitHub та GitLab, створює можливості для масштабних досліджень і впровадження автоматизованих інструментів оцінки. Такий підхід дозволяє компаніям-розробникам отримувати конкурентні переваги, знижувати ризики, пов’язані з помилками, і підвищувати ефективність командної роботи.

Ураховуючи важливість та складність завдань, пов’язаних із забезпеченням якості програмного забезпечення, дослідження методів і

моделей Data Mining для аналізу даних репозиторіїв має не лише теоретичне, але й значне практичне значення для розвитку сучасної індустрії програмування.

Мета дослідження - розробити методику та інструментальні засоби для аналізу якості репозиторіїв програмного забезпечення на основі методів Data Mining.

Об'єкт дослідження - процеси функціонування репозиторіїв програмного забезпечення.

Предмет дослідження - методи та моделі Data Mining, що застосовуються для аналізу даних репозиторіїв програмного забезпечення з метою оцінки їхньої якості.

Завдання дослідження:

1. Провести аналіз предметної області використання методів Data Mining для роботи з репозиторіями програмного забезпечення.

2. Описати показники ефективності для вимірювання якості репозиторіїв.

3. Дослідити існуючі моделі та методи функціонування репозиторіїв і систем контролю версій.

4. Розробити архітектуру інструменту для аналізу якості репозиторіїв програмного забезпечення.

5. Реалізувати програмний інструмент із використанням сучасних технологій.

Методи дослідження

1. Аналіз і синтез наукової літератури для визначення сучасних методів роботи з репозиторіями.

2. Моделювання для створення архітектури інструменту.

3. Методи Data Mining для збору, обробки та аналізу даних із репозиторіїв.

4. Емпіричне тестування розробленого інструменту на даних реальних проєктів.

Наукова новизна отриманих результатів

Розроблено методику аналізу якості репозиторіїв програмного забезпечення з використанням сучасних алгоритмів Data Mining та запропоновано архітектуру програмного інструменту, яка враховує специфіку роботи з даними репозиторіїв.

Практичне значення результатів

Розроблений інструмент може використовуватися для оцінки якості програмного забезпечення, моніторингу процесів розробки, ідентифікації дефектів та оптимізації командної роботи. Це дозволяє компаніям-розробникам програмного забезпечення підвищувати якість кінцевих продуктів, знижувати ризики і витрати на усунення помилок.

Структура магістерської роботи. Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 86 сторінок, і містить 23 рисунки, 4 таблиць, список використаних джерел із 50 найменувань.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ВИКОРИСТАННЯ МАЙНІНГУ ДЛЯ СХОВИЩ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Опис процесу аналізу якості програмного забезпечення за допомогою сховищ даних

Метою цього проєкту є багатоверсійний аналіз якості програмного забезпечення шляхом аналізу репозиторіїв програмного забезпечення. Дві відповідні області – це аналіз репозиторіїв програмного забезпечення (MSR) та багатоверсійний аналіз якості програмного забезпечення. Основна мета цієї дисертації – визначити, як характеристики репозиторію програмного забезпечення впливають на якість програмного забезпечення під час його еволюції.

Галузь MSR аналізує багаті дані з репозиторіїв програмного забезпечення для отримання цікавої та корисної інформації про програмні системи, проєкти та розробку програмного забезпечення [1]. MSR – це широкий клас досліджень, що стосуються вивчення репозиторіїв програмного забезпечення [2].

Репозиторії програмного забезпечення – це артефакти, які створюються та змінюються під час еволюції програмного забезпечення. Такі репозиторії включають різноманітні джерела, а саме дані та метадані, що зберігаються в репозиторіях вихідного коду, інформацію з систем відстеження вимог/помилок, архіви комунікацій тощо [2].

MSR стосується еволюції версій програмного забезпечення, змін, даних (вихідний код, звіти про помилки тощо), метаданих (хто, коли, де, чому). Аналіз репозиторіїв програмного забезпечення включає аналіз даних, вимірювання даних, інтелектуальний аналіз даних, відновлення даних, статичний аналіз вихідного коду, статистичний аналіз (кореляція) [2]. У цій роботі MSR буде використано для збору даних про процеси розробки під час еволюції програмного забезпечення.

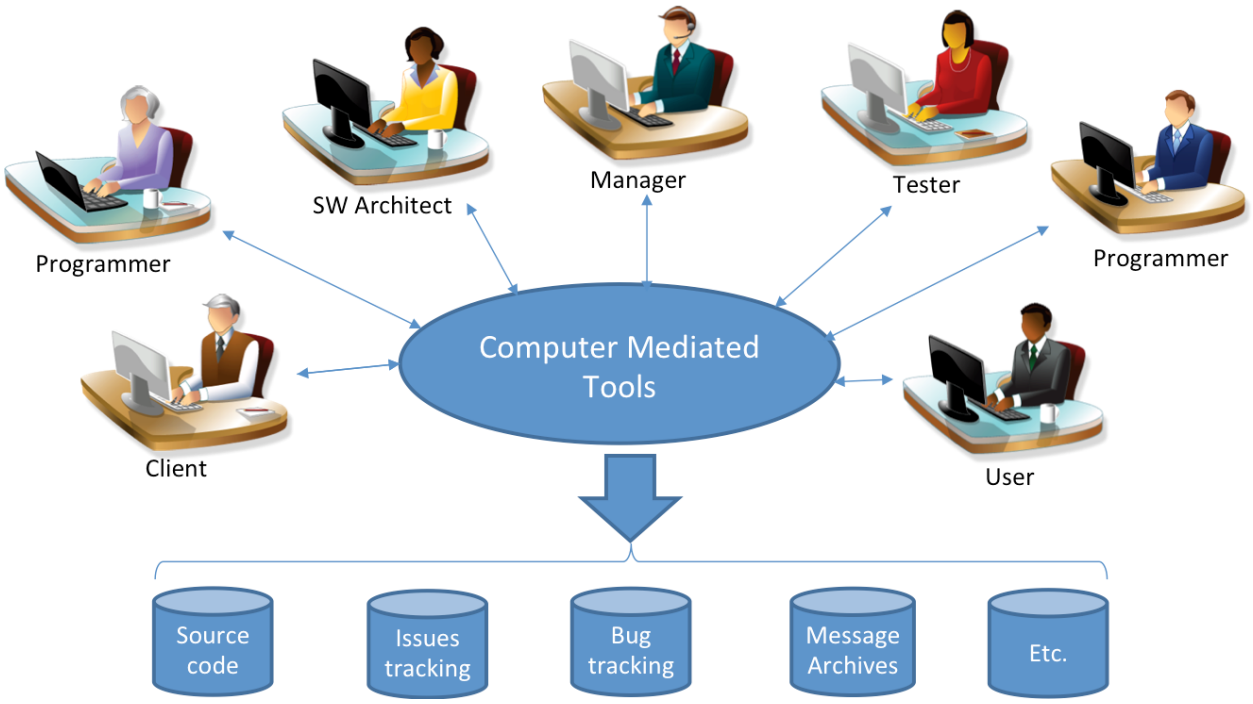


Рис. 1.1. Репозиторії програмного забезпечення

Існують різні типи сховищ програмного забезпечення (рис. 1.1). Найпопулярніші з них перераховані нижче [3]:

• Репозиторії вихідного коду, керовані системами контролю версій (VCS), такими як Git, Subversion, Mercurial.

• Системи відстеження проблем, такі як JIRA від Atlassian, Team Foundation Server, Redmine, YouTrack.

• Системи відстеження помилок, такі як Bugzilla та FogBugz.

• Архіви повідомлень і комунікацій.

Через велику кількість типів репозиторіїв дана робота зосереджується лише на репозиторіях вихідного коду. На рисунку 1.2 нижче [3] показано процес майнінгу для такого типу сховищ.

Кожен VCS має особливості. Функції являють собою цікаву інформацію, яку можна витягнути та проаналізувати для різних цілей. Наприклад, нижче наведено деякі можливі функції існуючих сховищ вихідного коду. Під час експериментів буде розглянуто декілька функцій із цього списку:

- Версії програмного забезпечення (мінорні та основні),
- Зміни коду (додані, відредаговані, видалені рядки),
- Кількість змінених файлів,
- Закріпити повідомлення,
- Метадані:
  - o Частота фіксації,
  - o Дата фіксації,
  - o Автор коміту.

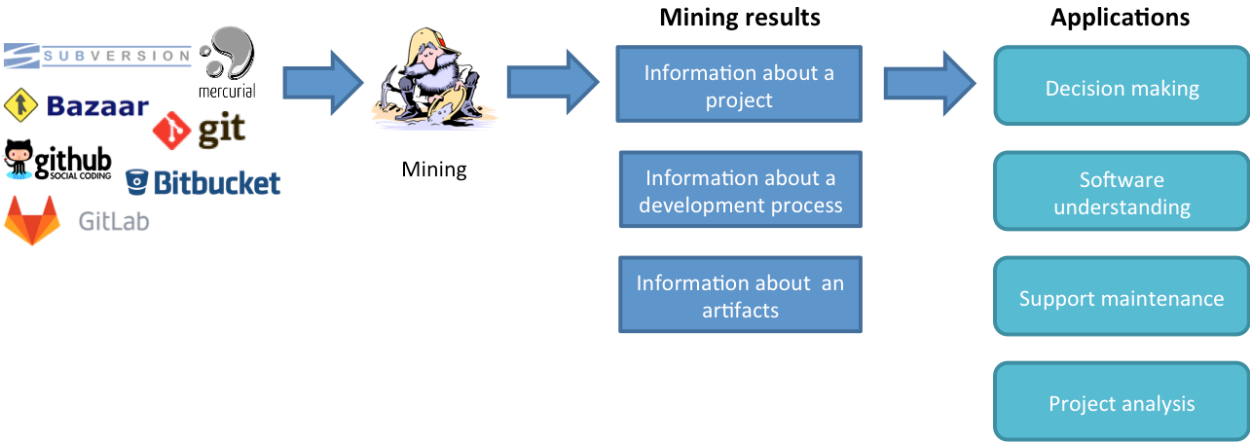


Рис. 1.2. Процес MSR

Для досягнення мети дослідження необхідно також оцінити якість програмного забезпечення, зафіксованого в репозиторіях. Він буде вимірюватися для кожної вибраної функції. Наприклад, у разі використання версій програмного забезпечення якість програмного забезпечення повинна вимірюватися для кожної версії проекту.

1.2. Опис показників ефективності для вимірювання якості репозитріїв програмного забезпечення

У цій роботі для вимірювання якості програмного забезпечення буде використано 8 ключових показників ефективності (KPI) з 21 метрикою якості. Вони представлені на рисунку нижче.

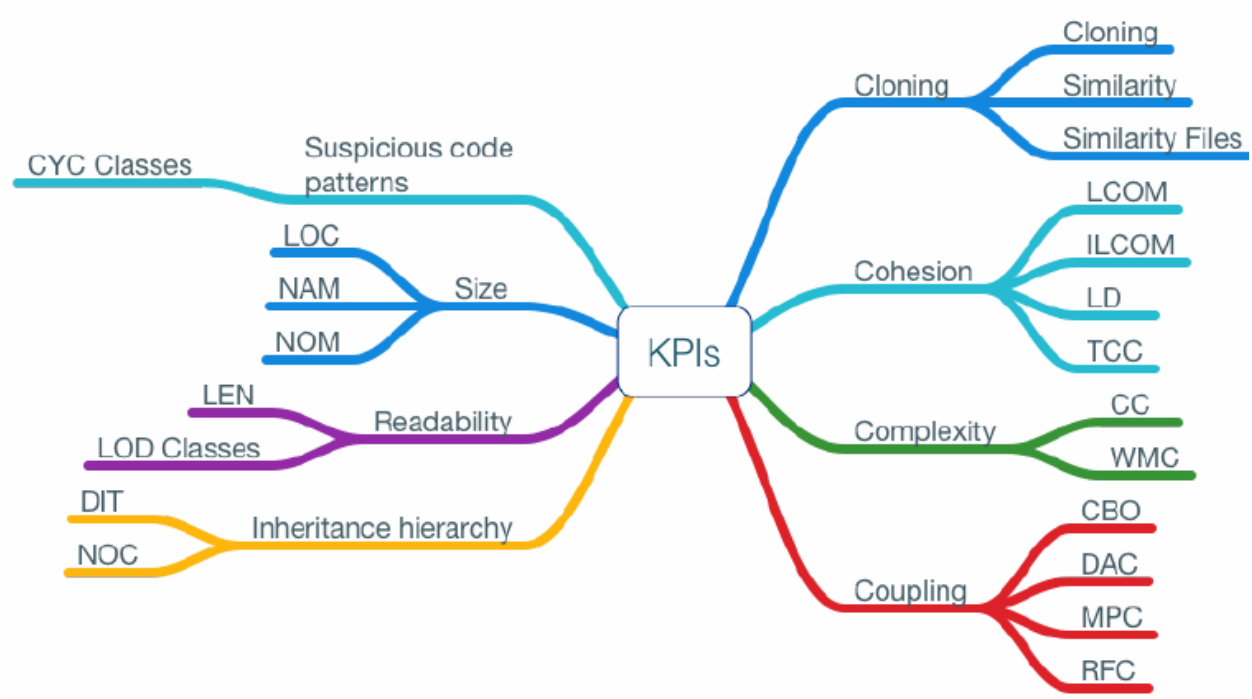


Рис. 1.3. KPI та метрики якості програмного забезпечення

Вимірювання будуть проводитися для Java-проектів. Нижче наведено опис показників якості. Усі ці KPI та показники вимірюватимуться за допомогою VizzAnalyzer™.

1. Клонування:
- Клонування - клонована частина класу/інтерфейсу,
- Подібність - схожість з іншими класами/інтерфейсами,
- Подібні файли – кількість подібних класів/інтерфейсів.
2. Згуртованість:
- LCOM - відсутність узгодженості методів,
- ILCOM - Покращення недостатньої згуртованості методів,
- LD - Локальність даних,
- TCC - жорстка згуртованість класу.
3. Складність:
- CC - цикломатична складність Mc Cabe,
- WMC - Weighted Method Count.
4. Зчеплення:
- CBO - зв'язок між об'єктами,

- DAC - зв'язок абстракції даних,
- MPC - Message Passing Coupling,
- RFC - відповідь для класу.

5. Ієрархія успадкування:

- DIT - глибина дерева успадкування,
- NOC - кількість дітей.

6. Читабельність:

- LEN - довжина назв,
- Класи LOD - відсутність документації в класах.

7. розмір:

- LOC - лінії коду,
- NAM - кількість атрибутів і методів,
- NOM - кількість локальних методів.

Підозрілі шаблони коду:

- Класи CYC - циклічні залежності між класами.

Якість репозиторію програмного забезпечення є важливим фактором для успішної розробки та підтримки програмного забезпечення. Існує багато різних показників ефективності та метрик, які можна використовувати для вимірювання якості репозиторіїв.

Загальні категорії метрик

1. Метрики розміру та складності:

- Кількість рядків коду (LOC): загальна кількість рядків у кодовій базі.

- Кількість файлів: загальна кількість файлів у репозиторії.

- Кількість комітів: загальна кількість змін, зафіксованих у репозиторії.

- Цикломатична складність: міра складності коду, заснована на кількості шляхів виконання.

2. Метрики активності та участі:

- Кількість авторів: кількість різних розробників, які зробили внесок у репозиторій.

- Кількість комітів на автора: середнє число комітів на одного автора.

- Час між комітами: середній час між комітами.

- Кількість коментарів у коді: кількість коментарів, доданих до коду.

3. Метрики якості коду:

- Щільність дефектів: кількість дефектів на тисячу рядків коду (KLOC).

- Кількість порушень кодування: кількість випадків, коли код не відповідає стандартам кодування.

- Покриття коду тестами: відсоток коду, який охоплений тестами.

- Кількість статичних аналізів коду: кількість попереджень, згенерованих інструментами статичного аналізу коду.

4. Метрики управління змінами:

- Час вирішення помилок: середній час, необхідний для вирішення помилки.

- Кількість відкритих помилок: кількість помилок, які ще не вирішені.

- Кількість відхилених змін: кількість змін, які були відхилені під час рецензування коду.

5. Метрики документації:

- Наявність файлу README: наявність файлу README, що описує проєкт.

- Повнота документації: рівень деталізації та повноти документації проєкту.

- Актуальність документації: як часто оновлюється документація.

Наведемо конкретні приклади метрик та їх опис.

а) Churn (Плинність):

- Вимірює кількість рядків коду, доданих, змінених або видалених протягом певного періоду.

- Висока плинність може свідчити про активну розробку або проблеми зі стабільністю коду.

б) Code Coverage (Покриття коду):

- Вимірює відсоток коду, який виконується під час тестів.

- Високе покриття коду може підвищити впевненість у якості коду.

в) Cyclomatic Complexity (Цикломатична складність):

- Вимірює складність програми, базуючись на кількості лінійно незалежних шляхів через код.

- Висока цикломатична складність може ускладнити розуміння та тестування коду.

г) Technical Debt (Технічний борг):

- Відображає накопичені проблеми в коді, які можуть призвести до проблем у майбутньому.

- Високий технічний борг може ускладнити підтримку та розвиток проєкту.

Важливо розуміти, що жодна метрика не є ідеальною сама по собі. Для отримання повної картини якості репозиторію рекомендується використовувати комбінацію різних метрик. Також важливо враховувати контекст проєкту та його цілі при інтерпретації метрик.

Метрики можна використовувати для:

- Моніторингу прогресу: відстеження змін у якості репозиторію з часом.

- Виявлення проблем: ідентифікація потенційних проблем у коді або процесі розробки.

- Прийняття рішень: прийняття обґрунтованих рішень щодо покращення якості репозиторію.

Для автоматизації збору та аналізу метрик можна використовувати різні інструменти, такі як SonarQube, Code Climate, GitStats та інші.

1.3. Постановка проблеми магістерського дослідження

Є кілька причин працювати над цією темою. По-перше, ми припускаємо, що функції сховища програмного забезпечення (наприклад, версії програмного забезпечення, зміни вихідного коду, частота комітів) впливають на якість коду і, таким чином, можуть впливати на кількість помилок у програмних продуктах. Це призведе до системних збоїв і витрат на їх усунення.

Існуюча робота в цій галузі не відповідає аналізу якості програмного забезпечення функціям сховища програмного забезпечення. У результаті втрачається цінна інформація, яку можна було б використати для запобігання помилкам, прийняття рішень, оптимізації процесів розробки.

Друга причина полягає в тому, що здатність вимірювати якість коду та аналізувати вплив функцій сховища програмного забезпечення на якість програмного забезпечення дозволяє нам краще розуміти історію проекту, стан якості проекту, процеси розробки та проводити майбутній аналіз проекту.

Основна мета представленого дослідження полягає в тому, щоб визначити, як функції сховища програмного забезпечення впливають на якість програмного забезпечення під час еволюції програмного забезпечення. Для досягнення цієї мети необхідно провести вимірювання якості програмного забезпечення. Якість буде вимірюватися для кожної вибраної функції. Наприклад, якість програмного забезпечення можна виміряти для кожної версії проекту, якщо використовується функція версій програмного забезпечення. Що стосується змін коду та змінених файлів, це можна виміряти для кожного коміту у VCS.

Коли всі KPI будуть отримані, їх слід об’єднати в одне значення якості [4] для кожної версії проекту (у разі використання функції сховища версій проекту). Після завершення всіх вимірювань можна розрахувати кореляцію між характеристиками сховища та значенням якості програмного забезпечення. Для отримання більш надійних результатів необхідно провести вимірювання якості програмного забезпечення та розрахунок кореляції для кількох проектів. Аналіз буде проведено для Java-проектів.

У результаті наших досліджень ми прагнемо автоматизувати процес аналізу якості, знайти взаємозв’язок між функціями репозиторію програмного забезпечення та значенням якості програмного забезпечення. Після цього ми прагнемо визначити найвпливовіші функції сховища.

Враховуючи все вищесказане, у цьому дослідницькому проекті є кілька дослідницьких питань:

1. Як функції сховища програмного забезпечення впливають на якість програмного забезпечення під час еволюції програмного забезпечення? Іншими словами, чи існує зв’язок між функціями сховища програмного забезпечення та якістю програмного забезпечення?

2. Які функції сховища мають найбільший вплив на якість програмного забезпечення?

3. Чи можна автоматизувати процес багатоверсійного аналізу якості програмного забезпечення за допомогою MSR, який містить KPI якості та вимірювання метрик, а також вилучення функцій репозиторію програмного коду ?

Очікується, що основним внеском представленого дослідження буде процедура аналізу якості програмного забезпечення за допомогою MSR та судження про вплив функцій репозиторію програмного забезпечення на якість програмного забезпечення. Існуючі роботи в цій галузі, не дозволяють зробити такий аналіз і винести таке судження.

Також однією з головних цінностей цієї роботи буде розроблений алгоритм інструменту, який може автоматизувати всю процедуру аналізу

якості програмного забезпечення через MSR та створювати зручні для користувача звіти XLSX. Цей інструмент надасть можливість іншим майбутнім дослідникам проводити власні більш детальні, комплексні дослідження та аналіз. Під час дослідження кілька проектів будуть проаналізовані за допомогою розробленого інструменту. Потім, на основі результатів і додаткового аналізу даних, можна буде відповісти на запитання дослідження та зробити судження про вплив функцій репозиторію програмного забезпечення.

У результаті нашого дослідження буде отримано три види результатів:

1. Процедура аналізу якості програмного забезпечення через MSR під час еволюції програмного забезпечення. Ця процедура описує та визначає весь процес аналізу, починаючи від клонування сховища до отримання результатів та етапу аналізу. Це доведе, що такий тип аналізу можливий, і дозволить майбутнім дослідникам легко використовувати таку процедуру для власного дослідження та створення ще більш складного аналізу.

2. Відповідь про:

a. Кореляція між якістю програмного забезпечення та функціями сховища програмного забезпечення. Це дозволить нам зрозуміти, як ці функції впливають на якість програмного забезпечення. Чи варто розробникам програмного забезпечення звертати на них увагу чи вони настільки малі, що ними можна знехтувати? Результати проведеного дослідження дадуть розгорнуту відповідь та обґрунтування щодо цього.

b. Найвпливовіші функції сховища. Також дуже важливо розуміти, які функції найбільше впливають на якість програмного забезпечення. Це дозволить розробникам запобігти майбутнім проблемам із нижчою якістю програмного забезпечення та звернути увагу на найважливіші функції.

3. Реалізований інструмент аналізу якості багатоверсійного ПЗ через MSR. Це один із найцінніших внесків цієї дисертації. Це дозволить майбутнім дослідникам і нам використовувати визначену процедуру для аналізу якості програмного забезпечення через MSR під час еволюції

програмного забезпечення. Це одна з найскладніших частин дисертації, враховуючи, що вона вимагатиме реалізації всього процесу, від клонування репозиторію Git, вилучення функцій репозиторію програмного забезпечення, виконання аналізу якості програмного забезпечення до збору всіх даних і створення звітів XLSX. Зрештою, цей інструмент також можна використовувати для:

а. Прийняття рішень керівниками проектів в IT-організаціях.

б. Аналіз проектів і процесів розробки.

в. Оптимізація витрат.

1.4. Висновки до розділу

Аналіз процесу використання сховищ програмного забезпечення показав, що сучасні методи майнінгу даних є ключовими для підвищення ефективності процесу аналізу якості програмного забезпечення. Сховища програмного забезпечення містять багатий набір історичних даних, таких як коміти, звіти про помилки, інформація про релізи, які можуть бути використані для виявлення закономірностей, прогнозування дефектів і підтримки прийняття рішень у процесі розробки. Застосування методів майнінгу даних дозволяє автоматизувати аналіз, зменшити суб'єктивність і підвищити точність оцінок.

Показники ефективності для вимірювання якості репозиторіїв відіграють вирішальну роль у забезпеченні об'єктивності оцінювання. До основних метрик належать:

- Метрики коду (наприклад, складність, кількість рядків коду, метрики повторного використання);

- Метрики процесу (частота комітів, тривалість циклів розробки, кількість авторів змін);

- Дані про дефекти (поширеність та частота появи помилок, час на їх виправлення).

- Використання цих показників забезпечує ефективний моніторинг стану репозиторіїв та дозволяє оцінювати їхній вплив на якість розроблюваного продукту.

Постановка проблеми дослідження дала змогу визначити основні виклики у використанні методів майнінгу даних для аналізу сховищ програмного забезпечення. Виявлено, що актуальними є питання:

- Підвищення точності прогнозування дефектів за допомогою комбінованих моделей;

- Адаптація майнінгу до специфіки великих і різнорідних репозиторіїв;

- Створення інструментів для автоматизації аналізу з урахуванням вимог до масштабованості.

Основна проблема дослідження полягає у розробці моделі аналізу репозиторіїв, яка дозволяє враховувати особливості різних типів проєктів та забезпечувати точність прогнозів.

Таким чином, у першому розділі розкрито значущість використання методів майнінгу даних у контексті аналізу сховищ програмного забезпечення. Встановлено ключові метрики ефективності та визначено напрямки подальшого дослідження для вдосконалення інструментів оцінки якості репозиторіїв.

РОЗДІЛ 2. МОДЕЛІ ТА МЕТОДИ ФУНКЦІОНУВАННЯ РЕПОЗИТОРІЇВ ТА СХОВИЩ ВИХІДНОГО КОДУ ДЛЯ ПРОЦЕСІВ МАЙНІНГУ

Цей розділ охоплює загальні та необхідні визначення та теоретичні знання про репозиторії програмного забезпечення, системи контролю версій, репозиторії програмного забезпечення для майнінгу та показники та інструменти якості програмного забезпечення.

2.1. Особливості репозиторіїв програмного забезпечення

В інформаційних технологіях та розробці програмного забезпечення репозиторій є центральним місцем, де організовано зберігаються та підтримуються дані. Термін репозиторій походить від латинського. Це означає посудину або камеру, в яку можна помістити речі. Інше значення — місце, де збирають речі.

Репозиторій може бути безпосередньо доступним для користувачів або може бути місцем для конкретних баз даних, файлів або документів, які можна отримати для подальшого переміщення або поширення в мережі [6]. У розробці програмного забезпечення репозиторій є центральним місцем для зберігання файлів. Він керується VCS для збереження кількох версій файлів. Зазвичай репозиторії зберігаються на сервері, де до них можуть отримати доступ кілька користувачів, але репозиторій також можна налаштувати на локальній машині для одного користувача [6].

Кожне сховище має три основні елементи - стовбур, гілки та теги. Поточна версія програмного проекту зберігається в транку. Це може бути кілька файлів вихідного коду, необхідні ресурси тощо. Гілки використовуються для розробки нових функцій і, отже, містять нові версії програми. Розробники створюють нову гілку кожного разу, коли вони роблять серйозні зміни в програмі. Після внесення всіх необхідних змін гілку

можна об’єднати в стовбур як останню версію. В іншому випадку він може бути припинений, якщо він містить небажані зміни [6]. Теги не пов'язані з активною розробкою і зазвичай використовуються для збереження версій проекту. Наприклад, щоразу, коли нова версія програмного забезпечення готова до випуску, розробники можуть створити для неї тег.

Репозиторій надає розробникам структурований і організований спосіб зберігання файлів вихідного коду. Це може бути корисним для проектів будь-якого розміру, але це особливо важливо для великих [6]. Будь-які зміни в проекті вносяться через фіксацію. Якщо останні оновлення викликають помилки чи інші проблеми, також можна швидко повернутися до попереднього стану проекту.

Багато VCS також підтримують порівняння різних версій файлів. Це може бути корисним під час дослідження вихідного коду. Крім того, якщо репозиторій зберігається на сервері, можна зробити перевірку всіх файлів для редагування і таким чином запобігти їх редагування більш ніж одному користувачеві одночасно [6].

2.2. Дослідження моделей функціонування систем контролю версій

Система контролю версій або VCS — це програмний інструмент, який відстежує будь-які зміни у файлі або наборі файлів у базі даних спеціального типу. Це допомагає керувати змінами з часом і надає доступ до певної версії на вимогу [7]. Контроль версій є компонентом керування конфігурацією програмного забезпечення. Він також відомий як контроль джерел або контроль версій. Це відповідає управлінню модифікаціями вихідного коду, документації, веб-сайтів тощо. Зазвичай зміни ідентифікуються «рівнем версії», «номером версії» або просто «версією» [8].

Якщо ви розробник програмного забезпечення, який пише код і хоче мати кожну версію проекту або вибраних файлів, тоді VCS є найбільш корисним інструментом у цьому випадку. [9] описує, що це дозволяє:

- Повернути зміни в певних файлах до одного з попередніх станів,
- Повернути весь проект,
- Порівняти зміни у файлах різних версій,
- Подивіться, хто змінив певний фрагмент коду.

VCS також дозволяє відновлювати файли, якщо вони з якоїсь причини були зіпсовані або втрачені.

Кожна система контролю версій має особливості. Функції — це деяка цікава інформація, яку можна отримати та проаналізувати для різних цілей. Наприклад, нижче наведено деякі можливі функції сховищ вихідного коду:

- Версії програмного забезпечення (мінорні та основні),
- Зміни коду (додані, відредаговані, видалені рядки),
- Кількість змінених файлів,
- Закріпити повідомлення,
- Метадані:
- Частота комітів,
- Дата скоєння,
- Автор коміту.

2.2.1. Типи систем керування версіями

Існує три типи систем керування версіями - локальні системи керування версіями, централізовані системи керування версіями та розподілені системи керування версіями. Давайте розглянемо їх усі.

1) Локальні системи керування версіями. Одним із поширених підходів, якщо вам потрібен простий контроль версій, є просто скопіювати файли в новий каталог із міткою часу. Якби це було так, у майбутньому може виникнути велика кількість проблем, а саме велика ймовірність того, що потрібний каталог буде забутий або втрачений, а деякі файли будуть пошкоджені неправильними записами.

Для вирішення таких проблем розробники створили локальні системи контролю версій, які дозволяють тримати всі зміни під контролем версій і

зберігати їх у базі даних [9]. На рисунку 2.1 представлена локальна система контролю версій.

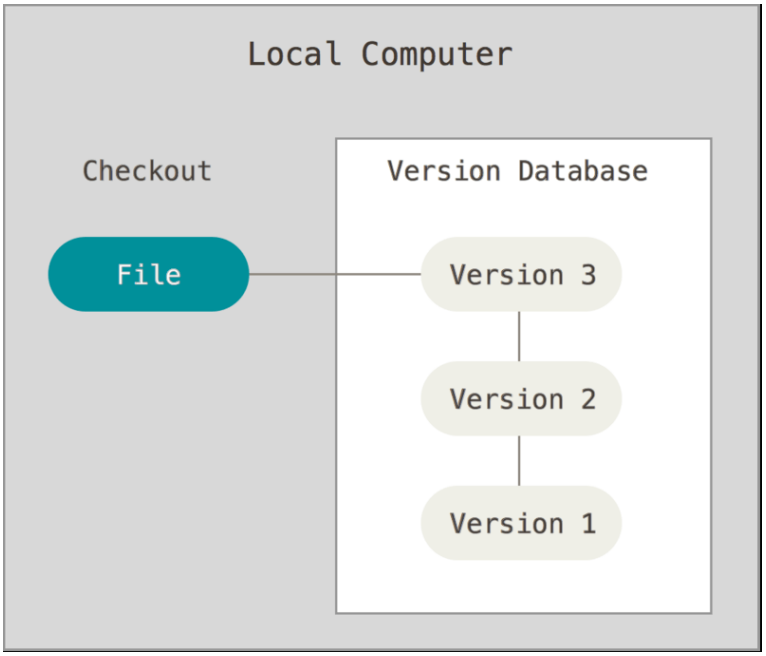


Рис. 2.1. Локальна система контролю версій

Одним із найпопулярніших інструментів для цього була система під назвою RCS. Навіть зараз він все ще поширюється на багатьох комп'ютерах. Підхід RCS базується на наборах латок. Кожен із таких наборів представляє відмінності між файлами. RCS зберігає їх на диску в спеціальному форматі. На вимогу він може легко відтворити стан будь-якого файлу в певний час у минулому, застосовуючи всі патчі [9].

2) Централізовані системи контролю версій. Наступним кроком у розвитку VCS є централізовані системи контролю версій (CVCS). Вони були створені для вирішення проблем співпраці, коли розробники хочуть разом працювати над одним проектом. Прикладом таких систем можуть бути CVS (Concurrent Versions System), Subversion (SVN), Perforce тощо [9]. На рисунку 2.2 представлена централізована система контролю версій.

Основна концепція CVCS полягає в тому, що всі вони мають один сервер і (можуть мати) багато клієнтів. Сервер зберігає всі файли з версіями, і клієнти

можуть перевірити їх. Централізовані системи контролю версій були стандартом контролю версій протягом багатьох років.

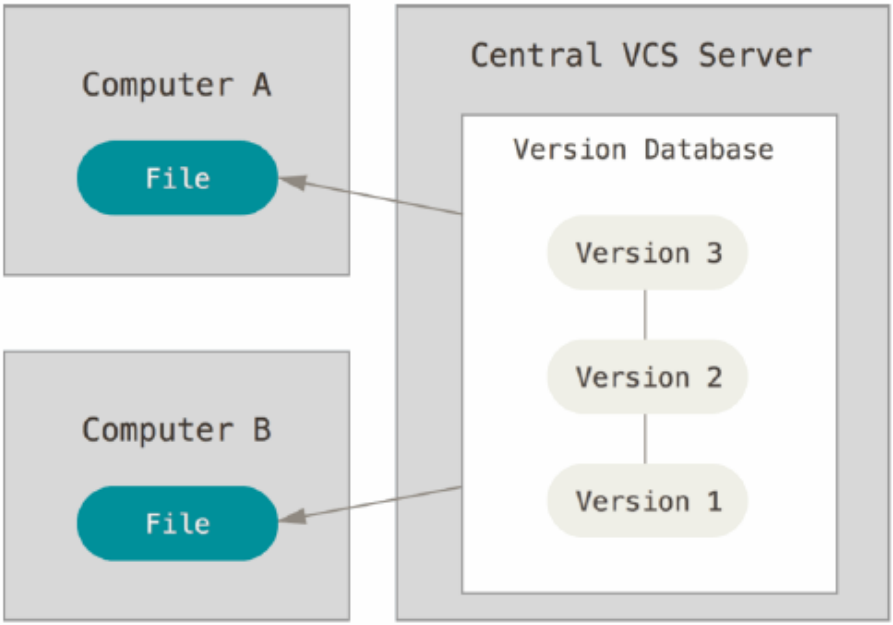


Рис. 2.2. Централізована система контролю версій

У порівнянні з локальними VCS, цей спосіб управління версіями пропонує багато переваг. Наприклад, можна перевірити, над чим працює кожен із розробників. Адміністратори можуть керувати доступом і дозволами. Загалом адмініструвати такі системи набагато легше, ніж працювати з локальними базами даних на кожному клієнті.

Однак централізовані VCS також мають кілька проблем. Одним і, мабуть, найбільш очевидним є те, що всі дані зберігаються лише на централізованому сервері. Це означає, що якщо сервер вимикається на деякий період часу, то ніхто не зможе зберегти свої зміни та співпрацювати взагалі. Крім того, якщо жорсткий диск центрального сервера буде пошкоджено, то всі дані будуть втрачені, крім локальних знімків, які можуть бути у розробників. Така ж ситуація з локальними VCS – «якщо у вас є вся історія проекту в одному місці, ви ризикуєте втратити все» [9].

3) Розподілені системи керування версіями (DVCS), такі як Git, Mercurial або Bazaar, працюють іншим способом. Розробники повністю

віддзеркалюють репозиторій, включаючи повну історію, замість того, щоб перевіряти останній знімок файлів. Тому, враховуючи, що кожен клон є повною копією всіх даних, немає небезпеки, якщо будь-який із центральних серверів загине. Будь-яке з клієнтських репозиторіїв можна відправити на сервер для його відновлення [9]. На рисунку нижче показано розподілену систему керування версіями.

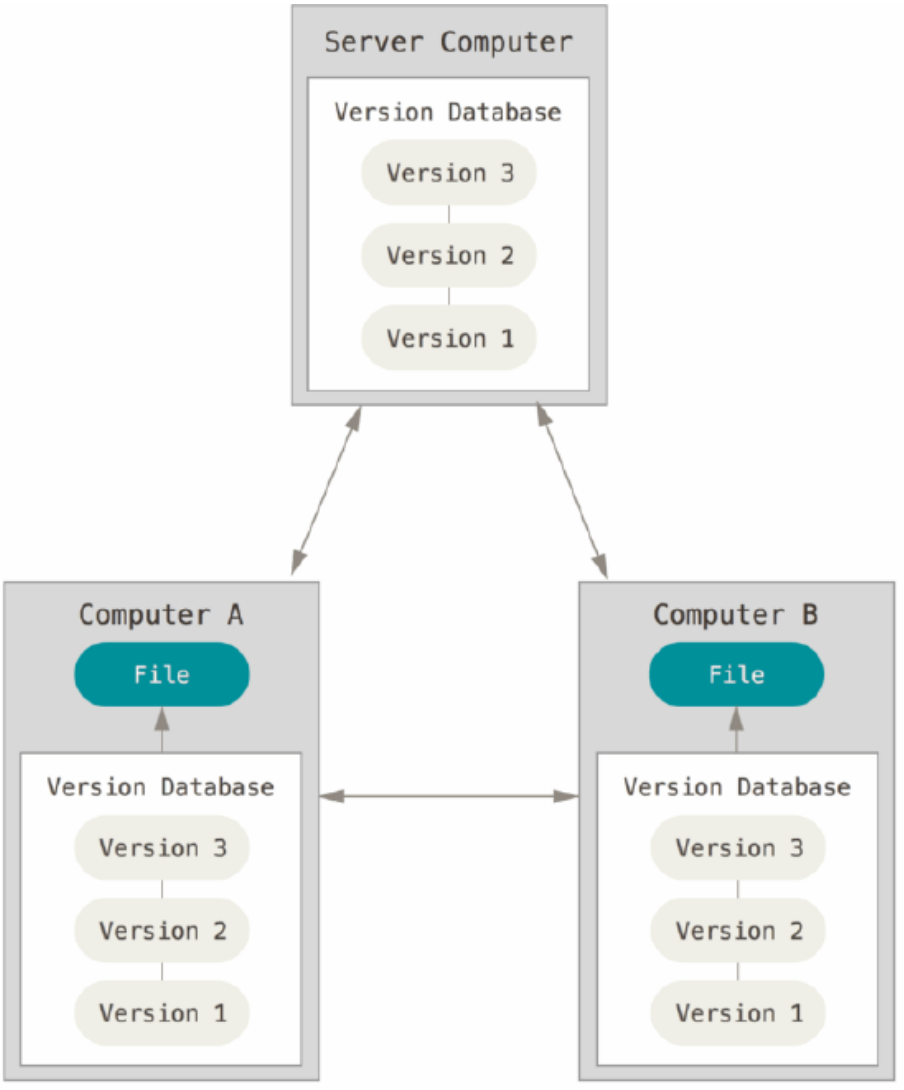


Рис. 2.3. Розподілена система контролю версій

Крім того, розподілені системи контролю версій також можуть досить добре працювати з кількома віддаленими сховищами. Це дозволяє розробникам одночасно різними способами співпрацювати з різними групами людей над одним проектом. Також можна налаштувати декілька

типів робочих процесів, наприклад ієрархічні моделі, що неможливо у VCS [9].

2.2.2. Програмні засоби керування версіями

Давайте розглянемо основні та найпопулярніші програмні засоби контролю версій, які зараз використовуються. Всі вони дуже корисні в повсякденній роботі і необхідні для організації мультирозробних проектів [8].

1) Git. Однією з найпопулярніших і де-факто стандартних систем контролю версій у світі на даний момент є Git [10]. Git — це VCS для відстеження змін у різних типах файлів і синхронізації роботи кількох людей над цими файлами. Він в основному використовується в розробці програмного забезпечення для керування вихідним кодом, але також може використовуватися в будь-якому наборі файлів для відстеження змін [11]. Git був створений у 2005 році Лінусом Торвальдсом для спрощення розробки ядра Linux.

Git має розподілену архітектуру, тому це хороший приклад розподіленої системи керування версіями. На відміну від більшості клієнт-серверних систем, таких як CVS або Subversion, де повна історія версій зберігається лише в одному місці (центральний сервер), у Git кожен розробник може мати на своєму комп’ютері повноцінне сховище з повною історією всіх змін. Функція відстеження всіх версій доступна навіть без доступу до мережі чи центрального сервера [11].

Плюси: повне дерево історії доступне офлайн; розподілена однорангова модель; дешева робота відділень; збільшення швидкості роботи [12].

Мінуси: навчання потрібне для тих, хто використовує Subversion; не є оптимальним для окремих розробників [12].

Існує багато різних веб-сервісів, які використовують Git і надають можливість використовувати Git як сервіс. Найбільш популярні з них GitHub, GitLab і Bitbucket. Розглянемо ці послуги докладніше.

GitHub - це веб-сервіс для контролю версій, який використовує Git. В основному використовується розробниками програмного забезпечення для програмування коду. GitHub пропонує керування вихідним кодом (SCM) і розподілену функцію керування версіями Git [13]. Крім того, він має багато додаткових функцій, які були розроблені над основною функціональністю Git. Сьогодні GitHub є одним із найпотужніших веб-сервісів Git. Він забезпечує:

- Контроль доступу,
- Функції співпраці, як-от коментарі,
- Запити на витягування,
- Відстеження помилок і запити функцій,
- Управління завданнями,
- Вікі.

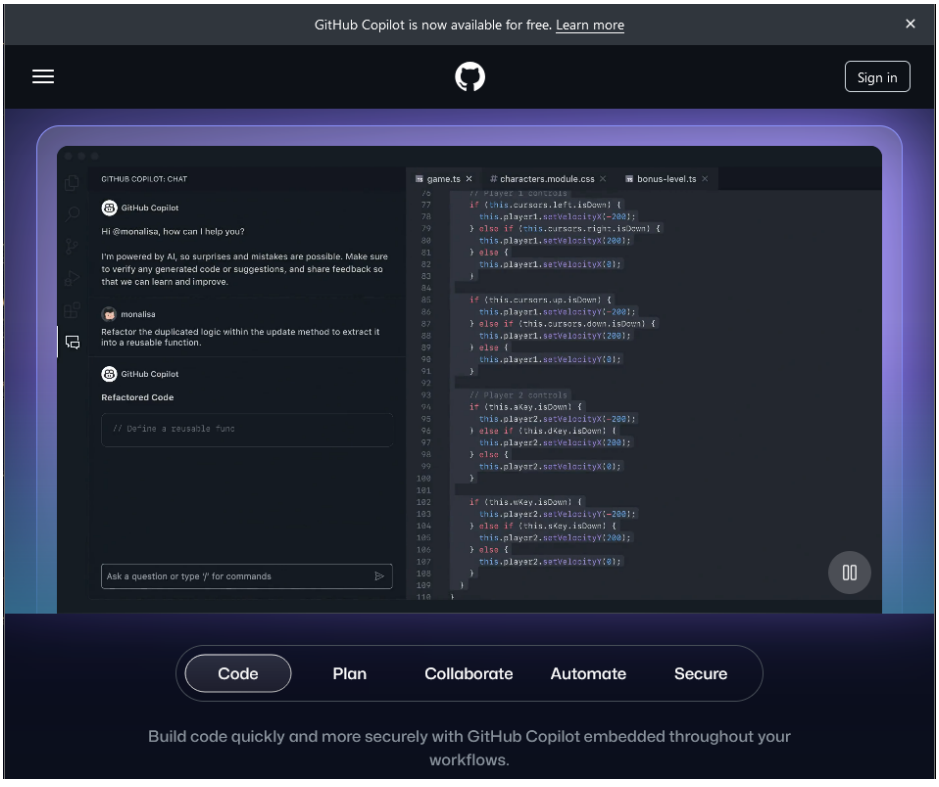


Рис. 2.4. Головна сторінка GitHub

Люди зазвичай використовують GitHub для проектів програмного забезпечення з відкритим кодом. Рік тому було близько 57 мільйонів сховищ,

створених понад 20 мільйонами користувачів. Це робить GitHub найбільшим хостингом вихідного коду в світі [13].

GitLab - це веб-менеджер сховищ Git з відкритим вихідним кодом і один з головних конкурентів GitHub. Зараз у нього більше 1400 учасників з відкритим кодом. Основна частина коду написана на Ruby і Go [14].

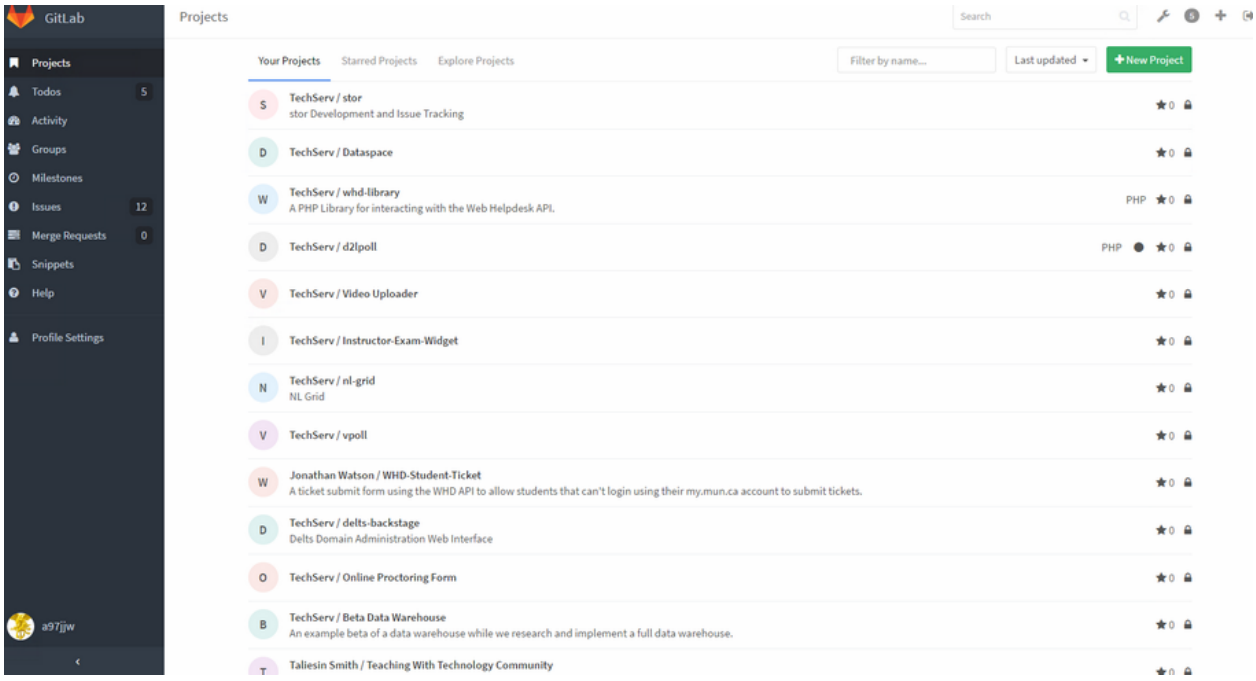


Рис. 2.5. Менеджер сховищ GitLab

GitLab розробили українці Дмитро Запорожець і Валерій Сізов. Основною метою було створити безкоштовний сервіс з відкритим вихідним кодом, який би міг конкурувати з GitHub, забезпечував високий рівень зручності та був безкоштовним. Окрім основних функцій, він має вікі, відстеження проблем і майже всі функції GitHub, згадані вище, а також кілька власних розроблених власноруч.

Bitbucket - це веб-сервіс для сховища контролю версій для вихідного коду та проектів розробки, що належить Atlassian (було придбано в 2010 році). Раніше він використовував Mercurial як систему контролю версій. Зараз він використовує Git. Однією з переваг Bitbucket є його глибока

інтеграція з іншим програмним забезпеченням Atlassian, таким як Jira, Confluence, Bamboo тощо [15].

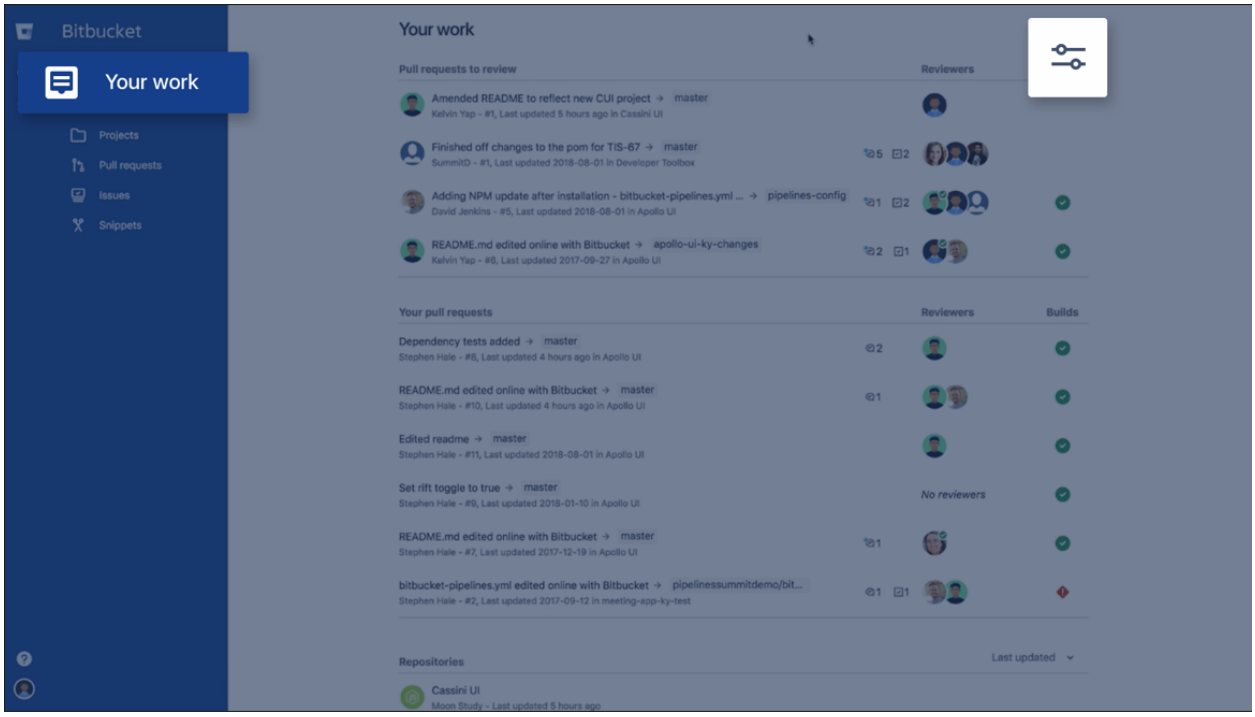


Рис. 2.6. Веб-сервіс для сховища контролю версій Bitbucket

Все рішення дуже схоже на GitHub і GitLab. Він також пропонує як безкоштовні облікові записи, так і комерційні плани [15]. Основна цільова аудиторія Bitbucket — професійні розробники програмного забезпечення з приватним пропрієтарним програмним кодом. У вересні 2016 року в Bitbucket було понад 5 мільйонів розробників і 900 000 команд.

2) Apache Subversion (SVN) був створений як альтернатива CVS і використовує найкращі його функції. Як і CVS, SVN також є безкоштовним рішенням із відкритим кодом. Щоб запобігти пошкодженню бази даних сховища, SVN використовує концепцію атомарних операцій. Це означає, що або застосовано всі зміни, внесені до джерела, або не застосовано жодної, і тому жодні часткові зміни не порушать вихідне джерело [12].

SVN вирішив одну з найбільших проблем CVS – високу вартість роботи філій. Він легко підтримує великі розгалужені проекти з багатьма напрямками [12]. У той же час у SVN також є кілька проблем - нижча

порівняльна швидкість і відсутність розподіленого контролю версій. Він використовує однорангову модель, а не використовує централізований сервер для зберігання оновлень коду. Крім того, якщо сервер не працює, жоден клієнт не зможе отримати доступ до коду. Це головний недолік підходу до виділеного сервера. Нижче наведено списки плюсів і мінусів SVN.

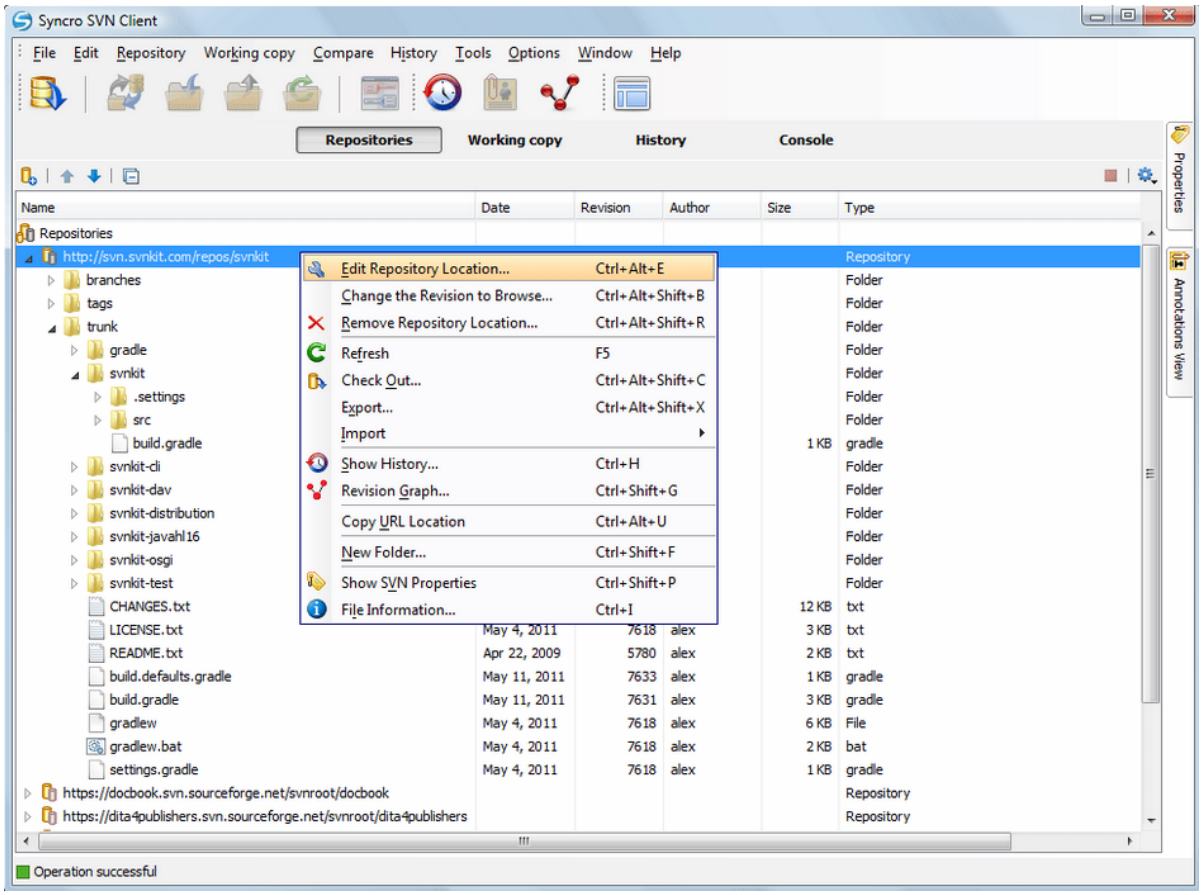


Рис. 2.7. Клієнтський додаток Apache Subversion (SVN)

Переваги:

- Включає атомарні операції,
- Дешевша робота відділень,
- Багато різних плагінів для IDE,
- На основі CVS.

Недоліки:

- Недостатня кількість команд управління репозиторієм,

- Містить помилки, пов'язані з перейменуванням файлів і каталогів,
- Повільна порівняльна швидкість.

3) Mercurial — це «розподілений інструмент контролю версій для розробників програмного забезпечення. Він підтримується в Microsoft Windows і Unix-подібних системах, таких як FreeBSD, macOS і Linux [16]». У [16] ми можемо знайти короткий опис переваг дизайну та взаємодії з Mercurial:

«Основні цілі дизайну Mercurial включають високу продуктивність і масштабованість, децентралізовану, повністю розподілену спільну розробку, надійну обробку як звичайного тексту, так і двійкових файлів, а також розширені можливості розгалуження та злиття, залишаючись концептуально простими. Він містить вбудований веб-інтерфейс.

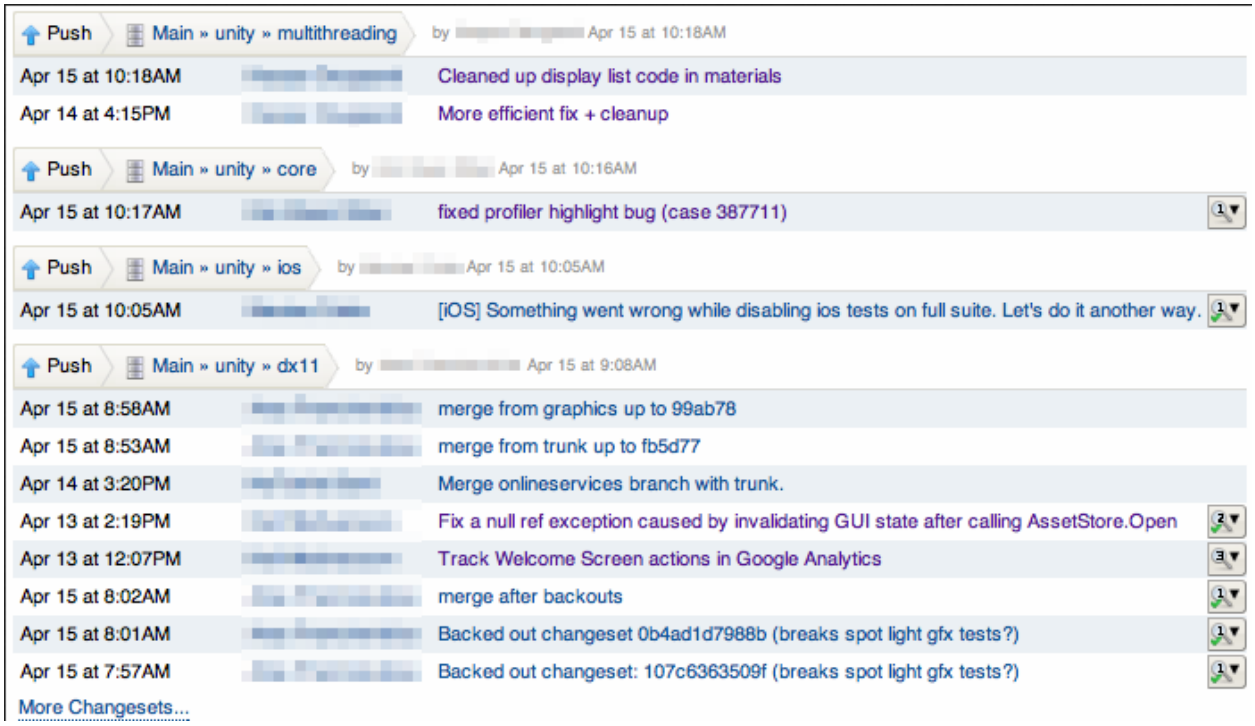


Рис. 2.8. Приклад контролю версіями засобами Mercurial

Mercurial також вжив заходів, щоб полегшити перехід для користувачів інших систем контролю версій, зокрема Subversion. Mercurial — це в основному програма, керована командним рядком, але доступні розширення

графічного інтерфейсу користувача, наприклад TortoiseHg, а кілька IDE пропонують підтримку контролю версій за допомогою Mercurial. Усі операції Mercurial викликаються як аргументи його програми-драйвера hg (посилання на Hg — хімічний символ елемента ртуть)».

4) GNU Bazaar — це розподілена клієнт-серверна система контролю версій. Bazaar може використовуватись командами, які співпрацюють у мережі, або одним розробником, який працює над кількома гілками локального контенту [17]. У Bazaar є команди, дуже схожі на команди CVS або Subversion. Це полегшує вивчення та розуміння.

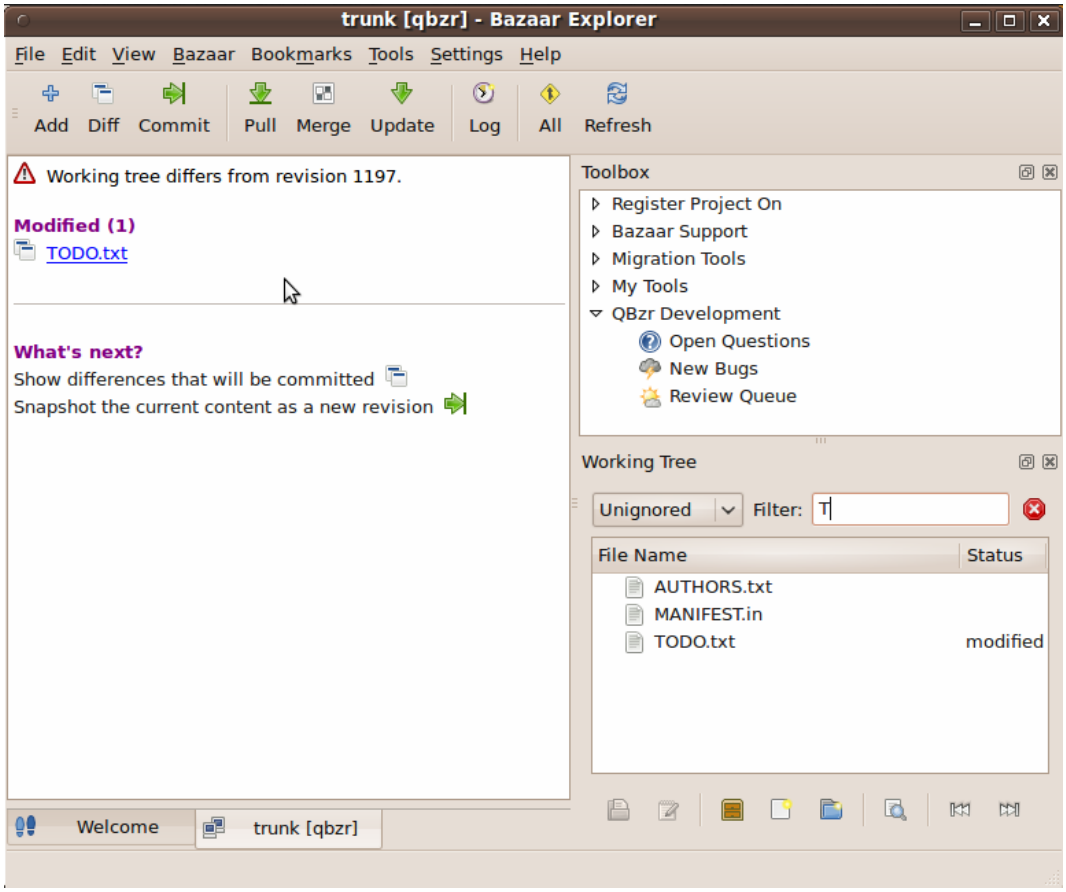


Рис. 2.9. Розподілена клієнт-серверна система контролю версій GNU Bazaar

За допомогою Bazaar можна створювати та запускати новий проект навіть без віддаленого сервера-сховища. Усе, що потрібно, це просто викликати команду init у каталозі, який має бути під контролем версій [17].

Bazaar може працювати як з центральним сервером, так і без нього, замість звичайної розподіленої VCS, яка не підтримує центральні сервери. Можна навіть використовувати обидва методи в одному проекті одночасно [17].

Bazaar також дозволяє працювати з іншими системами контролю версій, такими як Subversion, Mercurial, Git. Розробники можуть отримати нову гілку з іншої системи, внести зміни локально та закріпити їх у гілці Bazaar. Пізніше об'єднайте їх назад у основну (вихідну) систему. Ця можливість доступна для Subversion. Git і Mercurial мають доступ лише для читання [17].

2.3. Репозиторії програмного забезпечення для майнінгу (MSR)

Репозиторії програмного забезпечення для майнінгу (MSR) — це галузь розробки програмного забезпечення, де дослідники програмного забезпечення, застосовуючи методи інтелектуального аналізу даних, аналізують багаті дані в сховищах програмного забезпечення, щоб отримати корисну та дієву інформацію про програмні системи, проекти та програмну інженерію, створену розробниками в процесі розробки [1, 18]. MSR – це широкий клас досліджень з перевірки сховищ програмного забезпечення [2].

MSR стосується змін версій програмного забезпечення, змін, даних (вихідний код, звіти про помилки тощо), метаданих (хто, коли, де, чому). Під час створення сховищ програмного забезпечення для майнінгу отримана інформація, згідно з [18], може бути використана для «виявлення прихованих закономірностей і тенденцій, підтримки діяльності з розробки, підтримки існуючих систем або для покращення прийняття рішень для майбутньої розробки та еволюції програмного забезпечення. Зазвичай ці дані використовуються для кращого управління програмним забезпеченням і створення якісніших програмних систем шляхом аналізу минулих проектів розробки програмного забезпечення».

Репозиторії програмного забезпечення складаються з різних типів, таких як сховища вихідного коду, сховища помилок, списки розсилки, вікі-сторінки тощо. Сховища програмного забезпечення для майнінгу включають аналіз даних, вимірювання даних, інтелектуальний аналіз даних, відновлення даних, статичний аналіз джерела, статистичний аналіз (кореляція) [2]. На малюнку нижче представлено загальний потік сховищ програмного забезпечення для майнінгу.

Рис. 2.10. Процес MSR

Перш за все, щоб почати шукати інформацію, нам потрібні вихідні необроблені дані, за якими можна здійснювати пошук. Найцікавішими є історичні дані. Його можна знайти в різних системах, залежно від типу сховища програмного забезпечення:

- Для систем VCS дані можна отримати з CVS, SVN, Git, Mercurial.
- Для багтрекерів - від Bugzilla, JIRA, YouTrack.
- Для комунікаційних систем - з електронної пошти, журналів чатів, вікі-сторінок.

Наступним кроком є процес інтелектуального аналізу даних, на якому за допомогою спеціальних інструментів вся необхідна інформація, така як кількість комітів, кількість змін, автор коміту тощо (у випадку сховищ вихідного коду), буде вилучено та організовано у зрозумілий спосіб.

Нарешті, на останньому етапі вся отримана інформація буде проаналізована вручну або автоматично за допомогою спеціальних

інструментів візуалізації та аналізу даних. В обох випадках дослідники матимуть можливість знайти нову корисну інформацію та приховані ідеї. Результати аналізу можуть бути використані в широкому діапазоні різних сфер. Найбільш популярні з них:

- Гарантія якості,
- аналіз архітектури,
- передбачення помилок,
- Відгуки розробників.

2.4. Дослідження та опис показників якості програмного забезпечення та KPI

Для досягнення мети дослідження необхідно провести вимірювання якості програмного забезпечення. Для цього було використано 21 показник якості. Найважливіші з них представлені на малюнку нижче.

Метрики класифікуються як такі, що підтримують в основному твердження щодо складності програмного забезпечення, архітектури та структури програмного забезпечення, а також дизайну та кодування програмного забезпечення [19]. Нижче наведено список і малюнок показників якості.

1. Складність:

- розмір:

o Рядки коду (LOC)

- Складність інтерфейсу:

o Кількість атрибутів і методів (SIZE) o Кількість локальних методів (NOM)

- Структурна складність:

o Цикломатична складність Маккейба (CC)

o Зважена кількість методів (WMC) o Відповідь для класу (RFC)

2. Архітектура та структура:

- Спадок:

о Глибина дерева успадкування (DIT) о Кількість дітей (NOC)

- Зчеплення:

о Зв'язок між об'єктами (CBO) о Зв'язок абстракції даних (DAC) о Зв'язок передавання повідомлень (MPC)

- Згуртованість:

о Відсутність згуртованості в методах (LCOM) о Покращення LCOM (ILCOM) о Тісна згуртованість класу (TCC)

3. Інструкції з проектування та кодові угоди:

- Документація:

о Відсутність документації (LOD)

- Кодові угоди

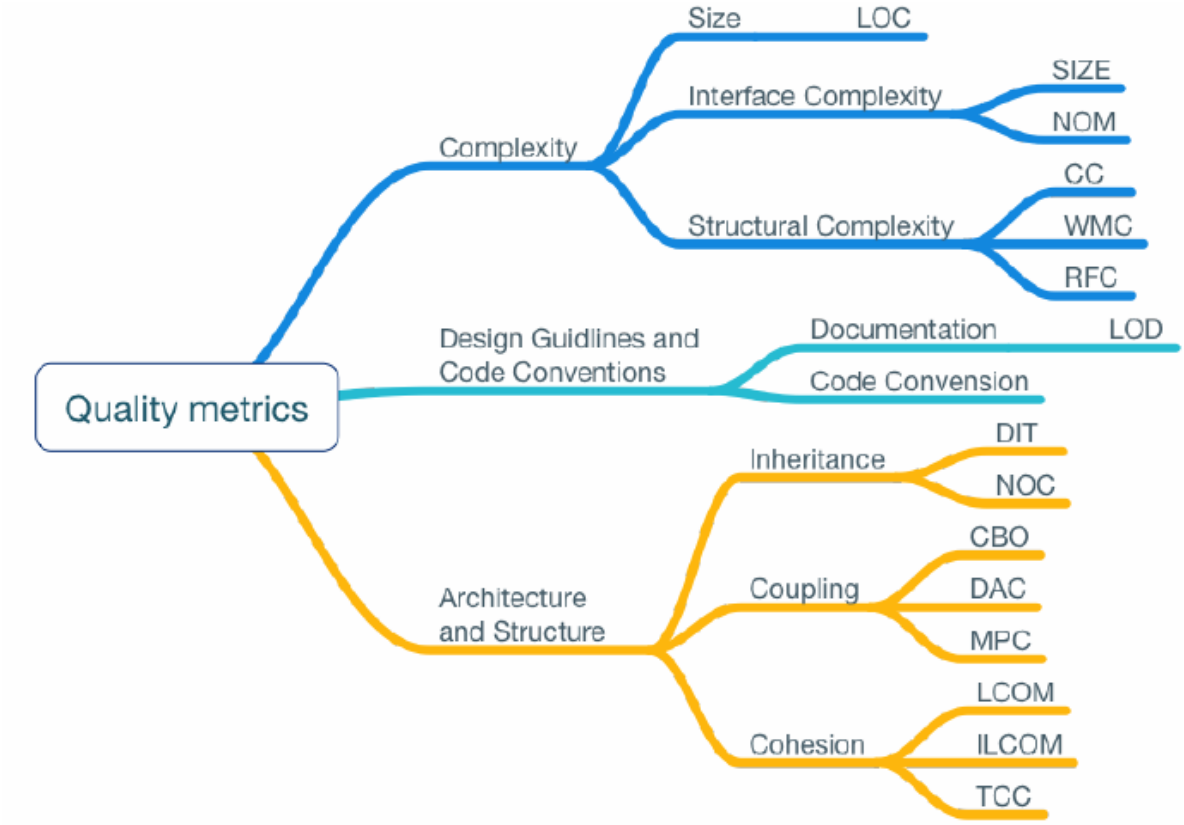


Рис. 2.11. Метрики якості ПЗ

Розглянемо детальніше кожну з метрик якості [19], які будуть використовуватися на етапі аналізу. Рядки коду просто підраховують рядки

вихідного коду (символи розриву рядків) певної сутності програмного забезпечення. Це простий, але потужний показник для оцінки складності програмних об’єктів. Оскільки це залежить від угод і формату коду, дуже важливо використовувати його в згенерованих кодах, оскільки в ньому можуть бути відсутні розриви рядків. Крім того, його можна виміряти лише у самому вихідному коді з інтерфейсу, і тому він є метрикою з боку інтерфейсу.

Кількість атрибутів і методів просто підраховує кількість атрибутів і методів класу. Це об’єктно-орієнтована метрика, яку можна застосувати до модульних мов, враховуючи кількість змінних (глобально видимих у модулі) і кількість його функцій і процедур.

Кількість локальних методів вимірює кількість методів, локально оголошених у класі. Успадковані методи не розглядаються. Це розмір інтерфейсу класу і дозволяє зробити висновки про його складність.

Цикломатична складність Маккейба є мірою складності структури керування програмним забезпеченням. Це кількість лінійно незалежних шляхів і, отже, мінімальна кількість незалежних шляхів під час виконання програмного забезпечення.

Зважена сума методів, реалізованих у класі. Він параметризований способом обчислення ваги кожного методу. Можливі показники ваги:

- Цикломатична складність Маккейба,
- Рядки коду,
- 1 (незважений WMC).

Цей варіант WMC використовує метрику цикломатичної складності McCabe для розрахунку ваги для кожного методу. Спочатку визначений як об’єктно-орієнтована метрика, її можна легко адаптувати до необ’єктно-орієнтованих систем, обчислюючи зважену суму функцій, реалізованих у модулі чи файлі. Відповідь для класу — це кількість (відкритих) методів у класі та методів, які безпосередньо викликаються ними. RFC застосовний лише до об'єктно-орієнтованих систем.

Глибина дерева успадкування (DIT) — це максимальна довжина шляху від класу до кореневого класу в структурі успадкування системи. DIT вимірює, скільки суперкласів може впливати на клас. DIT застосовний тільки до об'єктно-орієнтованих систем.

Кількість дітей (NOC) — це кількість безпосередніх підкласів (нащадків), підпорядкованих класу (батькові) в ієрархії класів. NOC вимірює, скільки класів безпосередньо успадковують методи або поля від суперкласу. NOC застосовний лише для об'єктно-орієнтованих систем.

Зв’язок між об’єктами (CBO) — це кількість інших класів, з якими зв’язаний клас. CBO застосовний тільки для об'єктно-орієнтованих систем. Data Abstraction Coupling (DAC) вимірює складність зв’язку, спричинену абстрактними типами даних (ADT). Ця метрика пов’язана зі зв’язком між класами, що представляє головний аспект об’єктно-орієнтованого дизайну, оскільки рівень зв’язку між класами вирішальним чином впливає на ступінь повторного використання, зусилля з обслуговування та тестування для класу. В основному те саме, що й DAC, але зв’язок обмежений посиланнями на типи.

Message Passing Coupling (MPC) вимірює кількість викликів методів, визначених у методах класу, до методів інших класів, а отже, залежність локальних методів від методів, реалізованих іншими класами. Це дозволяє робити висновки про передачу повідомлень (виклики методів) між об’єктами залучених класів. Це дозволяє зробити висновки щодо можливості повторного використання, технічного обслуговування та зусиль щодо тестування.

Показник «Відсутність згуртованості в методах» є показником кількості незв’язаних пар методів у класі, що представляють незалежні частини, які не мають зв’язку. Він представляє різницю між кількістю пар методів, які не мають спільних змінних екземплярів, і кількістю пар методів, які мають спільні змінні екземплярів.

Метрика покращення LCOM є мірою кількості підключених компонентів у класі. Компоненти — це методи класу, які спільно використовують (підключаються) змінні екземпляра класу. Чим менше окремих компонентів, тим вище зв’язаність методів у класі.

Показник Tight Class Cohesion вимірює зв’язок між загальнодоступними методами класу. Це відносна кількість прямо підключених відкритих методів у класі. Класи з низькою зв'язністю вказують на помилки в дизайні.

Lack Of Documentation показує, скільки коментарів бракує в класі, вважаючи один коментар класу та коментар на метод як оптимальні. Структура та зміст коментарів ігноруються.

2.5. Аналіз існуючих досліджень по темі

Сфера аналізу якості програмного забезпечення досить велика, і в даний час ця сфера популярна серед дослідників і розробників. Це зумовлює наявність великої кількості робіт і вже проведених досліджень у цій галузі. На жаль, не надто багато робіт, які використовують процес MSR. Найцікавіші та пов'язані дослідження представлені нижче.

У [22] автори описали вплив імен ідентифікаторів на якість коду. Вони припускають, що імена ідентифікаторів низької якості є причиною низькоякісного вихідного коду, який перетворюється на програмне забезпечення низької якості. Для оцінки якості вихідного коду використовуються цикломатична метрика складності та індекс ремонтопридатності. Також метрика читабельності застосовується для оцінки читабельності методів.

Автори використовують кілька методів дослідження, які включають збір даних і статистичний аналіз. Також розроблено інструмент для автоматичного вилучення та аналізу ідентифікаторів із вихідного коду Java. Використовуючи статистичний аналіз, вони виявили, що помилкові

ідентифікатори в класах Java пов’язані з низькою якістю вихідного коду. Вони показали, що асоціація також представляє вищий рівень деталізації методів Java і використовує методи, які використовуються в медицині для оцінки діагностичних тестів, щоб показати, які дефекти іменування ідентифікаторів можна використовувати для діагностики проблемного вихідного коду Java.

У наведеному вище дослідженні автори використовують лише кілька показників якості програмного забезпечення. У цьому випадку загальне значення якості не буде повноцінним. Це призведе до спотворення результатів. У моїй роботі цей недолік буде усунено шляхом використання 8 ключових показників ефективності з 21 метрикою якості.

У [23] автор аналізує відмінності двох наступних версій програмного забезпечення у вихідному коді та під час виконання, демонструючи вплив змін вихідного коду на виконання програмного забезпечення. Цей вплив допомагає розробникам зрозуміти, на що вплинуть програми під час виконання через їхні зміни у вихідному коді. Підхід, який дозволяє передбачити, що зазнає впливу в майбутньому, і відповідний інструмент під назвою IMPEX були створені та представлені в цій статті.

В дослідженні [23] присутній такий же недолік, як і попередній. Для визначення впливу автор використовує певні метрики, але в цьому випадку користувач цієї системи не знатиме реальний стан проекту після змін, оскільки він отримає неповну інформацію про якість програмного забезпечення.

Аналіз структурних змін програмного забезпечення між версіями (протягом еволюції програмного забезпечення) описано в [24]. Автори спостерігають за змінами, внесеними близько до випусків програмного забезпечення, і використовують метод дослідження збору даних, зокрема спостереження та вимірювання, щоб виміряти відтік коду та структурні зміни вихідного коду.

Під час свого дослідження автори виявили базові зміни (наприклад, зміни поля, властивості та методу) та складні зміни (наприклад, зміни за межами класу та зміни переміщення), які вказують на процеси рефакторингу. Також автори виявили, що ближче до дат релізу кількість структурних змін вихідного коду значно зростає.

У цій роботі відсутній аналіз якості програмного забезпечення та розрахунок кореляції між структурними змінами вихідного коду та якістю програмного забезпечення (наприклад, значення KPI, кількість помилок тощо). Однак це вказано в майбутніх цілях автора, а також використання алгоритмів машинного навчання для прогнозування якості програмного забезпечення.

На даний момент все вищезазначене через відсутність можна вважати недоліками. У цій роботі вони будуть усунені шляхом обчислення кореляції між якістю програмного забезпечення та характеристиками репозиторію програмного забезпечення та створення прогнозної моделі на основі спостережених даних про функції репозиторію програмного забезпечення та якість програмного забезпечення.

У [25] автори описують вплив рефакторингу на внутрішню якість програми. Метою авторів є розробка інструментів і методів для розробників програмного забезпечення, які допоможуть покращити якість програми на етапі рефакторингу. У цій статті автори пропонують формалізм для опису впливу рефакторингу структури програми. Основний внесок цієї роботи полягає в тому, щоб показати дрейф якості, викликаний рефакторингом коду. Нарешті автор описує подібну тему в [26], але зосереджується на довгостроковому впливі змін на якість програмного забезпечення.

Було проаналізовано існуючі роботи в області якості програмного забезпечення та виявлено, що всі вони описують аналіз якості програмного забезпечення без прив’язки до функцій репозиторію програмного забезпечення. Таким чином, вони втрачають важливу інформацію, яку можна

використовувати для запобігання помилкам, прийняття рішень та оптимізації процесів розробки.

2.6. Архітектура інструментів аналізу якості програмного забезпечення

У цьому розділі описано один із механізмів вимірювання якості – VizzAnalyzer™, який використовується під час аналізу якості програмного забезпечення.

Відповідно до [20] VizzAnalyzer™ — це «фреймворк, призначений для підтримки обслуговування та реінжинірингу програмного забезпечення. Дозволяє інтегрувати довільні інструменти зворотного проектування, тобто інструменти для програмного аналізу та/або візуалізації. Потім VizzAnalyzer TM може інтерактивно та ітеративно отримувати інформацію про програму, фокусувати її, аналізувати та візуалізувати».

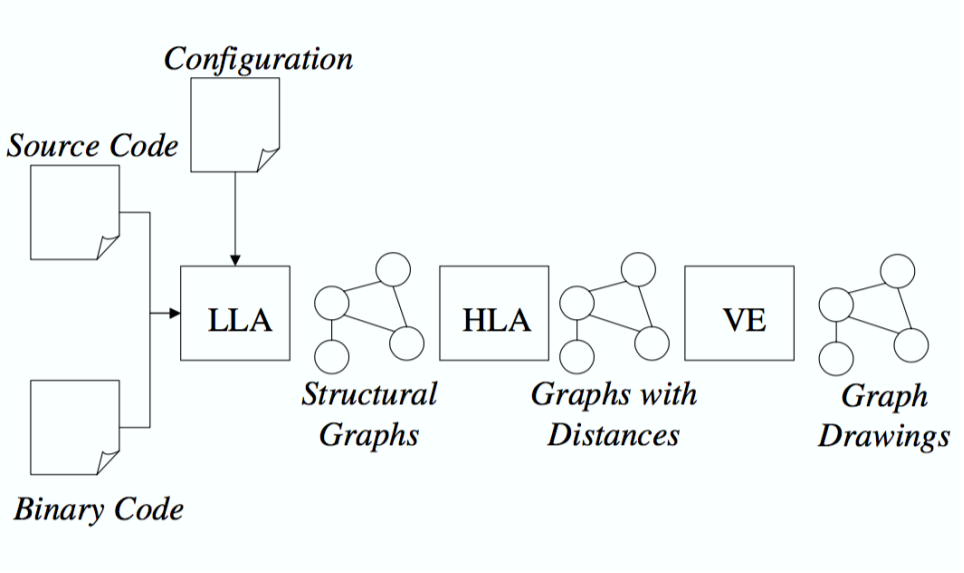


Рис. 2.12. Загальна архітектура фреймворку VizzAnalyzer

Програмна архітектура фреймворку складається з трьох основних компонентів, кожен з яких реалізований як точка розширення, куди можна додавати визначені користувачем алгоритми [21] (див. рис. 2.12):

- Механізм аналізу низького/бінарного рівня (LLA).
- Високорівневий механізм аналізу та показників (HLA).
- Механізм візуалізації (VE).

Відповідно до [21], давайте зрозуміємо, що робить кожен із компонентів. «LLA бере вихідний і двійковий код і створює модель досліджуваної програми. Модель програми фіксується у формі одного або кількох структурних графів. Ці графіки можна конфігурувати відповідно до потреб HLA».

«HLA працює на структурних графах. Він додає додаткову інформацію, зафіксовану або в наступних графіках, або в анотаціях існуючих. Спеціальні алгоритми HLA обчислюють анотації, які можна інтерпретувати як показники відстані між вузлами графіків. Щоб намалювати графік, потрібно мати принаймні один обчислений показник відстані». «VE обчислює макет графіка, що містить інформацію про відстань, і зображує графік. Він забезпечує інтерфейс користувача, який дозволяє масштабувати, обертати та агрегувати зображені графіки».

Ключові особливості та можливості VizzAnalyzer:

- Інтеграція інструментів: VizzAnalyzer є фреймворком, який дозволяє об'єднувати різні інструменти аналізу та візуалізації, створюючи комплексне середовище для розуміння та модифікації програмного забезпечення.

- Ітеративний аналіз: Фреймворк підтримує ітеративний процес аналізу, дозволяючи користувачам поступово заглиблюватися в деталі програми та отримувати повніше уявлення про її структуру та поведінку.

- Візуалізація: VizzAnalyzer надає різні засоби візуалізації, які допомагають представити інформацію про програму в наочній формі. Це полегшує розуміння складних взаємозв'язків та залежностей у коді.

- Аналіз: Фреймворк надає інструменти для аналізу програмного забезпечення, включаючи агрегацію, фільтрацію та об'єднання інформації. Це дозволяє користувачам зосередитися на найважливіших аспектах програми.

VizzAnalyzer може бути використаний у різних сценаріях, пов'язаних з аналізом та реінжинірингом програмного забезпечення, таких як:

- Розуміння існуючого коду: VizzAnalyzer допомагає розробникам зрозуміти структуру та поведінку незнайомого коду.

- Виявлення проблем у коді: Фреймворк може бути використаний для виявлення помилок, вразливостей та інших проблем у коді.

- Реінжиніринг програмного забезпечення: VizzAnalyzer допомагає в процесі реструктуризації та покращення існуючого коду.

- Візуалізація архітектури: Фреймворк надає засоби для візуалізації архітектури програмного забезпечення, що полегшує розуміння та документування складних систем.

Переваги VizzAnalyzer:

- Гнучкість: VizzAnalyzer дозволяє інтегрувати різні інструменти, що робить його гнучким та адаптованим до різних завдань.

- Потужні можливості аналізу: Фреймворк надає потужні інструменти для аналізу програмного забезпечення, які допомагають виявляти складні проблеми.

- Наочна візуалізація: VizzAnalyzer надає різні засоби візуалізації, які полегшують розуміння складних даних.

Загалом, VizzAnalyzer є потужним та гнучким фреймворком для аналізу та реінжинірингу програмного забезпечення, який може бути корисним розробникам та інженерам, що займаються підтримкою та модифікацією існуючих систем.

2.7. Висновки до розділу

В даному розділі проведено систематичний аналіз моделей і методів функціонування репозиторіїв та інструментів керування версіями. Підкреслено значущість використання репозиторіїв для аналізу процесів розробки, визначено ключові показники якості та описано сучасні

архітектури інструментів. Отримані результати можуть слугувати основою для побудови нових моделей та вдосконалення існуючих підходів до аналізу сховищ програмного забезпечення.

Особливості репозиторіїв програмного забезпечення полягають у тому, що вони виступають ключовим джерелом даних для аналізу процесів розробки. Репозиторії накопичують інформацію про зміни коду, авторів, історію розробки та інші дані, що дозволяють використовувати методи майнінгу для виявлення закономірностей, оцінки якості та оптимізації розробки. Дослідження моделей функціонування систем контролю версій показало, що системи керування версіями (СКВ) є фундаментальними для організації роботи з репозиторіями.

Репозиторії для майнінгу програмного забезпечення (MSR) є важливою дослідницькою областю, яка передбачає використання даних з репозиторіїв для підтримки рішень у процесах розробки. Основна увага приділяється аналізу змін, ідентифікації дефектів та покращенню ефективності команд розробників.

Аналіз існуючих досліджень (2.5) показав, що майнінг репозиторіїв активно застосовується для прогнозування дефектів, оптимізації командної роботи та автоматизації рутинних завдань. Виявлено, що інтеграція методів машинного навчання та штучного інтелекту значно підвищує точність аналізу.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДІВ DATA MINING ДЛЯ АНАЛІЗУ ЯКОСТІ РЕПОЗИТОРІЇВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Представлення методу дослідження

Щоб відповісти на визначені питання дослідження, буде використано кількісну стратегію та перевірку гіпотез. Перевірка гіпотез - це науковий метод, який складається з трьох компонентів:

• Збір даних шляхом спостереження та експериментування,

• Формулювання гіпотез,

• Перевірка гіпотез.

Цей метод використовує статистичні дані для визначення ймовірності того, що дана гіпотеза вірна [27]. Це стосується статистичного аналізу вибіркових даних з метою створення припущень щодо сукупності. На сьогодні існує три підходи до тестування, які коротко описані нижче [27].

1. P-значення. P-значення оцінюють, наскільки добре вибіркові дані підтверджують аргумент про істинність нульової гіпотези. Він вимірює, наскільки дані сумісні з нульовою гіпотезою.

2. Статистика оцінки. Оцінка – це процес, який дозволяє робити висновки про популяцію на основі інформації, отриманої з вибіркових даних [28]. Оцінка параметра сукупності може бути виражена двома способами: точковою оцінкою та інтервальною оцінкою.

3. Фактор Байєса. У статистиці використання факторів Байєса є байєсівською альтернативою класичній перевірці гіпотез [29][30]. Порівняння байєсівських моделей — метод відбору моделей на основі факторів Байєса.

Розглянемо алгоритм підходу P-values [27].

1. Сформулюйте гіпотезу:

- Нульова гіпотеза, позначена H0 , — це гіпотеза про те, що вибіркові спостереження є результатом чистої випадковості;

- Альтернативна або експериментальна гіпотеза, позначена Hі або Ha, – це гіпотеза про те, що на вибіркові спостереження впливає якась невипадкова причина (спостереження є результатом реального ефекту).

2. Визначте тестову статистику, використовуючи вибіркові дані та припускаючи, що H 0 є істинним.

3. Обчисліть P-значення, яке є ймовірністю того, що тестова статистика може бути отримана принаймні такою ж екстремальною, як спостережувана, припускаючи правдивість H0. Чим менше P-значення , тим сильніші докази проти H0.

Порівняйте P-значення з a (прийнятне значення значущості). Якщо P < a, тоді H 0 відхиляється, H 1 дійсний, і спостережуваний ефект є статистично значущим. В іншому випадку H0 не виключається.

Щоб мати можливість перевірити гіпотези, необхідно зібрати необхідну кількість даних. Для цього буде використано метод контрольованого експерименту . Контрольований експеримент є «високо сфокусованим способом збору даних і особливо корисним для визначення причинно-наслідкових зв’язків» [31]. У такому типі експериментів порівнюються результати, отримані від експериментальної та контрольної груп. Зазвичай вони однакові, за винятком одного аспекту в контрольній групі - незалежної змінної, вплив якої перевіряється [31].

Для проведення контрольованого експерименту необхідні експериментальна та контрольна групи. Експериментальна група – це «група осіб, які піддаються впливу досліджуваного фактора» [31]. Контрольна група не піддається впливу фактора. Всі інші зовнішні впливи повинні бути постійними, тобто кожен інший фактор або вплив між експериментальною групою та контрольною групою повинен залишатися незмінним. Єдине, що може відрізнятися між двома групами, це досліджуваний фактор [31].

Важливою частиною експериментів є змінні. Змінна — це будь-який фактор, яким можна керувати, змінювати чи вимірювати під час експерименту. У наукових експериментах існує кілька типів змінних. Найбільш часто використовуваними є незалежні та залежні змінні, зазвичай нанесені на діаграму чи графік [31]. У науковому експерименті незалежна змінна є єдиною змінною, яка змінюється, щоб перевірити, як вона впливає на залежну змінну. Залежна змінна - це змінна, яку слід спостерігати та вимірювати. Він є залежним, оскільки «це фактор, який залежить від стану незалежної змінної» [31].

У контрольованому експерименті все, крім незалежної змінної, залишається постійним. Набір даних використовується як контрольна група, яка «зазвичай є нормальним або звичайним станом, а одна або кілька інших груп досліджуються, де всі умови ідентичні контрольній групі, за винятком однієї змінної» [31].

Одна з переваг контрольованого експерименту полягає в тому, що можна усунути значну частину невизначеності в результатах. Контрольований експеримент забезпечує високий ступінь впевненості в результаті, але у випадку, якщо неможливо контролювати кожну змінну, плутаний результат неминучий [31].

Контрольовані експерименти мають як сильні, так і слабкі сторони. Однією з сильних сторін є те, що «результати можуть встановити причинний зв’язок, тобто вони можуть визначити причинно-наслідковий зв’язок між змінними» [31]. З іншого боку, контрольовані експерименти можуть бути штучними, тобто «здебільшого вони проводяться у штучних лабораторних умовах і, отже, прагнуть усунути багато ефектів реального життя. У результаті аналіз контрольованого експерименту повинен включати судження про те, наскільки штучні умови вплинули на результати». [31]

Вище було визначено два методи дослідження: перевірка гіпотези та контрольований експеримент для проведення дослідження та відповідей на запитання дослідження. З огляду на те, що ми зробили кілька припущень на

початку, найпопулярнішим підходом до того, як їх довести або спростувати, є перевірка гіпотези. У той же час одним із найбільш підходящих способів збору даних для цього є контрольований експеримент. У цьому розділі описано, як методи дослідження застосовуються в цьому проекті.

Згідно з визначеними дослідницькими питаннями, основною метою дослідження є визначення того, чи впливають особливості сховища програмного забезпечення на якість програмного забезпечення під час еволюції програмного забезпечення. Іншими словами, знайдіть кореляцію між функціями сховища програмного забезпечення та якістю програмного забезпечення. І якщо така кореляція існує, то визначити, які з функцій репозиторію найбільше впливають на якість програмного забезпечення.

Для проведення контрольованих експериментів необхідно визначити незалежні та залежні змінні. У випадку, коли ми хочемо виміряти якість програмного забезпечення залежно від функцій сховища програмного забезпечення, якість програмного забезпечення є залежною змінною, а функції VCS є незалежними змінними. Функції — це деяка цікава інформація, яку можна отримати та проаналізувати для різних цілей. У цьому дослідженні буде використано кілька функцій сховища вихідного коду:

• Зміни вихідного коду (додані, змінені, видалені рядки коду),

• Загальна кількість змін вихідного коду в одному коміті,

• Зміни файлів (додані, змінені, видалені файли),

• Загальна кількість змінених файлів в одному коміті,

• Довжина повідомлення коміту,

• Метадані:

- Дата коміту,

- Автор коміту.

Для досягнення мети дослідження необхідно провести вимірювання якості програмного забезпечення. Він буде вимірюватися для кожної вибраної функції. Наприклад, у разі використання змін файлів якість програмного забезпечення слід вимірювати для кожного коміту в проекті.

Для вимірювання якості програмного забезпечення буде використано 7 ключових показників ефективності з 21 метрикою якості (див. розділ 2). Нижче наведено список найцікавіших KPI:

• клонування,

• згуртованість,

• складність,

• зчеплення,

• ієрархія успадкування,

• читабельність,

• розмір.

Щоб отримати узагальнене значення якості, усі KPI будуть об’єднані в одне середнє значення всіх ключових показників ефективності. Це значення буде використовуватися як залежна змінна в контрольованих експериментах.

Усі необхідні дані будуть отримані за допомогою спеціально створеного інструменту. Інструмент буде виконувати три основні функції:

• Вимірювання всіх показників якості та KPI. Для цього буде використано VizzAnalyzer. Це бібліотека Java, яка дозволяє проводити аналіз якості програмного забезпечення.

• Вилучення функцій сховища вихідного коду. Це буде проведено за допомогою спеціально створеного модуля в інструменті.

• Генерація звітів XLSX, які містять всю інформацію про аналіз якості та функції сховища вихідного коду.

Коли вся необхідна інформація буде отримана та зведена в один документ, можна починати власне дослідження. Для побудови кореляції між якістю програмного забезпечення та функціями сховища вихідного коду в цьому дослідженні буде використано Microsoft Power BI.

Вимірювання та експерименти будуть проведені для кількох проектів на основі Java, Maven. Java є однією з ТОП-3 найпопулярніших мов програмування на сьогоднішній день, і в той же час Maven є провідною системою побудови та керування залежностями для проектів Java.

3.2. Загальний огляд та вимоги до системи перевірки якості репозиторіїв програмного забезпечення

Як було описано раніше, для проведення дослідження необхідно зібрати багато даних про якість програмного забезпечення та особливості сховища вихідного коду. Враховуючи те, що отримання такої кількості інформації вручну є важким і трудомістким завданням, було вирішено запровадити інструмент, який може виконувати всю цю роботу автоматично та максимально простим способом, щоб майбутні дослідники могли легко повторно використовувати або навіть розширювати функціональність цього інструменту для різних цілей.

Є кілька ключових функцій, які реалізовані в цьому інструменті перевірки якості ПЗ:

1. Клонування сховища Git із віддаленої URL-адреси на локальну машину.
2. Видобуток список усіх комітів у вказаній гілці.
3. Автоматичне створення проекту із залежностями за допомогою Maven.
4. Виконання аналізу якості програмного забезпечення для конкретного коміту.
5. Витягування функції сховища для певного коміту.
6. Агрегування отриманої інформації та створення звіту XLSX.

Інструмент доступний у двох різних варіантах: як виконуваний файл JAR, тож дослідники без знань програмування можуть легко використовувати його через командний рядок, і як бібліотека Java, яку можна імпортувати в існуючий проект Java і використовувати.

Щоб розширити наявні можливості інструменту, розробники можуть розгалужувати репозиторій Git на GitHub або надсилати запити на підключення до існуючого.

3.2.1. Випадки використання

Для кінцевого користувача частина основних функцій, описаних вище, прихована. Нижче наведено список дій (випадків використання), які доступні для кінцевого користувача. Діаграма варіантів використання представлена на рисунку 3.1.

1. Введіть вхідні дані для аналізу якості: віддалений URL, ім’я та ім’я власника сховища Git.
2. Укажіть одну з гілок репозиторію, коміти з якої будуть аналізуватися.
3. Вкажіть кількість комітів, які потрібно проаналізувати.
4. Вкажіть зміщення комітів. У разі необхідності, наприклад, аналіз можна почати з 10-го коміту зверху.
5. Отримайте згенерований звіт XLSX з результатами аналізу.

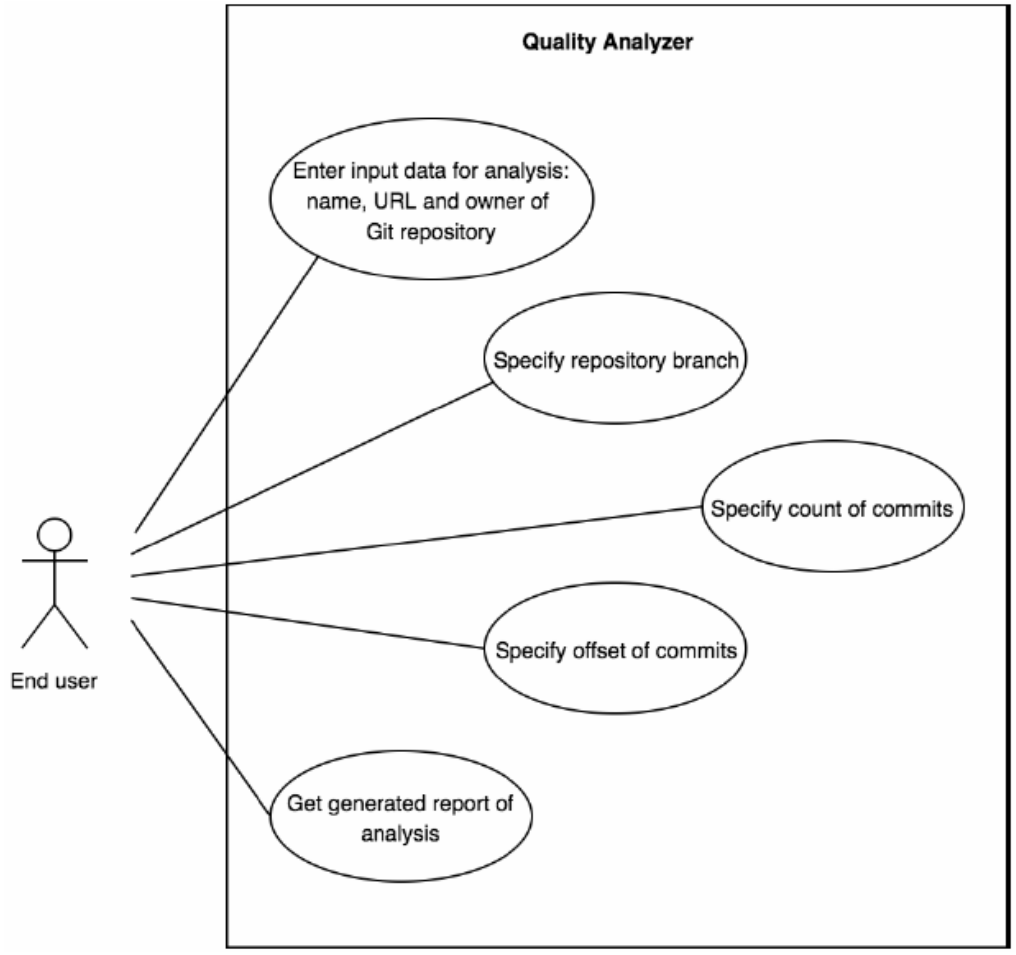


Рис. 3.1. Use case діаграма

3.2.2. Функціональні та нефункціональні вимоги

Нижче представлені функціональні (таблиця 3.1) і нефункціональні (таблиця 3.2) вимоги до програмного забезпечення.

Таблиця 3.1.

Функціональні вимоги

Код	**опис**
FR1	Інструмент має клонувати репозиторій Git із віддаленої URL-адреси на локальну машину.
FR2	Під час аналізу репозиторій слід зберігати у тимчасовій папці.
FR3	Репозиторій слід видалити після аналізу.
FR4	Інструмент повинен отримати список усіх комітів у вказаній гілці.
FR5	Інструмент повинен автоматично створювати проект із залежностями за допомогою Maven.
FR6	Інструмент повинен запускати аналіз якості програмного забезпечення для конкретного коміту.
FR7	Інструмент має видобувати функції сховища для певного коміту.
FR8	Інструмент має агрегувати отриману інформацію та генерувати звіт XLSX.
FR9	Програмне забезпечення має розповсюджуватися як виконуваний файл JAR.
FR10	Програмне забезпечення також має бути доступним у вигляді бібліотеки Java.

Таблиця 3.2.

Нефункціональні вимоги

Код	**опис**
NR1	Люди, які не мають жодних знань програмування, можуть легко почати використовувати інструмент.
NR2	Інструмент має бути доступний у двох версіях: виконуваний JAR-файл, який може використовуватися з CLI різними дослідниками; і як бібліотека Java, які можна імпортувати в інший проект.
NR3	Інструмент має бути розроблений таким чином, щоб можна було легко розширити функціональні можливості (наприклад, додати конектор до іншого VCS або замінити VizzAnalyzer™ іншою структурою).
NR4	Алгоритм аналізу якості повинен бути виконаний в розумні терміни.
NR5	Вихідний код має бути добре прокоментований.

3.3. Розробка архітектури програмного інструменту

У цьому розділі представлено загальний огляд програмної архітектури розробленого інструменту аналізу якості. Він містить описи основних компонентів, діаграму високого рівня та діаграму класів, щоб отримати більш детальне та технічне розуміння системи.

3.3.1. Компоненти

Програмне забезпечення складається з чотирьох основних компонентів (сервісів), які охоплюють весь функціонал, представлений вище. Сервіси: Git Service, Maven Service, Analysis Service і Report Service. Загальний огляд розробленої архітектури програмного забезпечення та його компонентів можна представити на рисунку нижче.

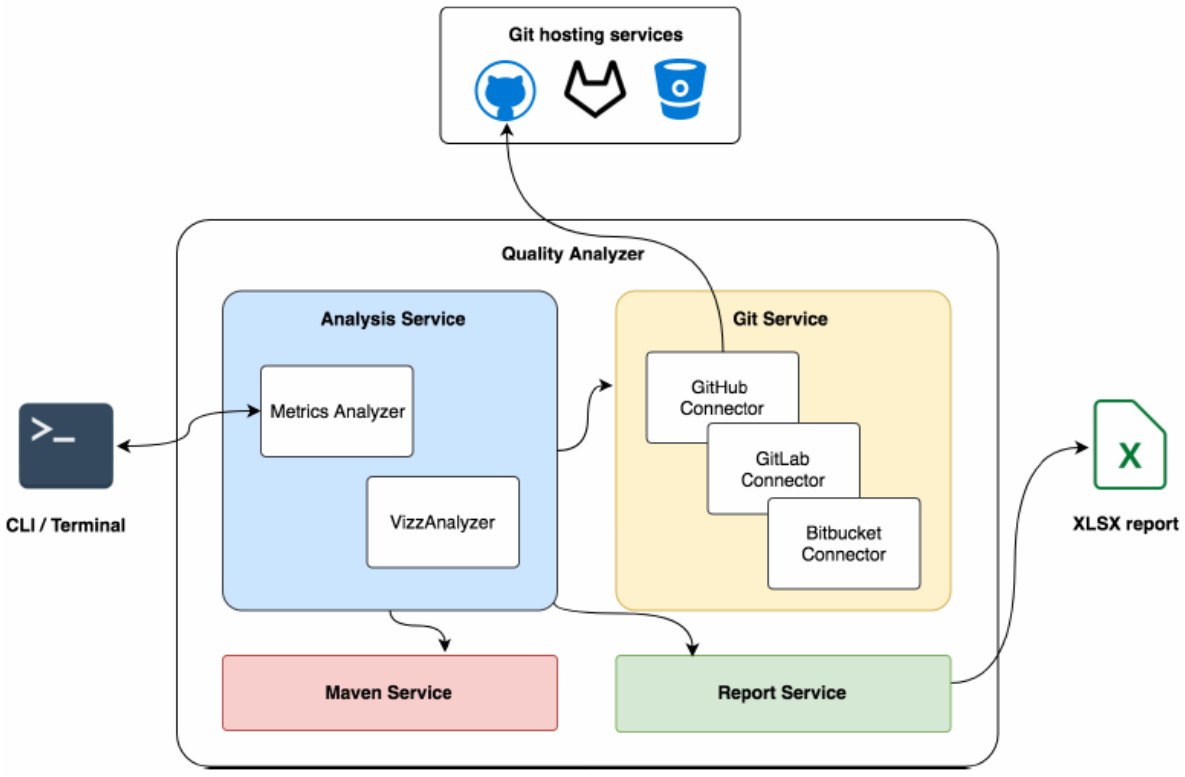


Рис. 3.2. Архітектура програмного інструменту

1) Служба Git. Він обробляє основні операції Git (наприклад, клонування репозиторію, перевірка певної гілки/коміту тощо), які необхідні

для маніпулювання сховищами Git, щоб отримати інформацію про функції сховища вихідного коду та провести аналіз у цілому.

Крім того, Git Service працює разом із конекторами для веб-сервісів Git-хостингу, таких як GitHub, GitLab і Bitbucket. Такі конектори потрібні для отримання детальної інформації про кожну фіксацію в репозиторії:

- Скільки рядків коду було додано/змінено/видалено,
- Скільки файлів було додано/змінено або видалено,
- Хто зробив зміни,
- Коли відбулися зміни,
- Як довго триває повідомлення коміту,
- Скільки комітів інші розробники залишили для цього коміту в GitHub/GitLab/Bitbucket,
- Загальна кількість усіх змінених файлів і рядків коду.

Реалізація заснована на Eclipse JGit. Це безкоштовна бібліотека для Java, яка надає базові функції для роботи зі сховищами Git. Це дослідження та програмне забезпечення охоплюють лише конектор GitHub. Інші конектори можуть бути легко реалізовані з метою розширення програмного забезпечення.

2) Служба Maven відповідає за потік збірки вихідного проекту, який слід проаналізувати. Щоб провести якісний аналіз, кожен вихідний проект має бути створений із залежностями заздалегідь. У цьому випадку значення метрик якості та KPI будуть більш точними.

Ця служба автоматично збирає Maven безпосередньо після того, як служба Git перевірить необхідний комміт. Реалізація заснована на безкоштовній бібліотеці Java від Apache - Maven Invoker. Він забезпечує функціональність Maven через Java.

3) Служба аналізу. Це основний сервіс, який проводить аналіз якості програмного забезпечення та обчислює всі метрики якості та KPI. Він заснований на бібліотеці VizzAnalyzer Java, яка описана в розділі 2.

4) Служба звітів. Після завершення аналізу настав час узагальнити всі результати та представити їх досліднику. Щоб зробити це максимально легким для них, було вирішено створити документ XLSX наприкінці аналізу та представити цей документ кінцевим користувачам. Тоді аналізувати та працювати з вихідними даними буде набагато легше.

Документ XLSX має визначену структуру. Загалом він містить два основних розділи: результати аналізу якості програмного забезпечення та результати вилучення функцій репозиторію вихідного коду. Нижче наведено список усіх даних у вихідному звіті XLSX:

1. Аналіз якості ПЗ:

- згуртованість,
- розмір,
- складність,
- ієрархія,
- зчеплення,
- клонування,
- читабельність,
- повна якість.

2. Вилучення функцій репозиторію:

- хеш коміту,
- дата коміту,
- електронна адреса автора коміту,
- ім’я автора коміту,
- довжина повідомлення коміту,
- кількість коментарів до коміту на GitHub,
- кількість доповнень,
- кількість видалення,
- загальна кількість змін,
- додані файли,
- змінені файли,

- видалені файли,

- загальна кількість змінених файлів.

Щоб мати більш детальний технічний огляд системи, нижче наведено діаграму класів розробленого програмного забезпечення.

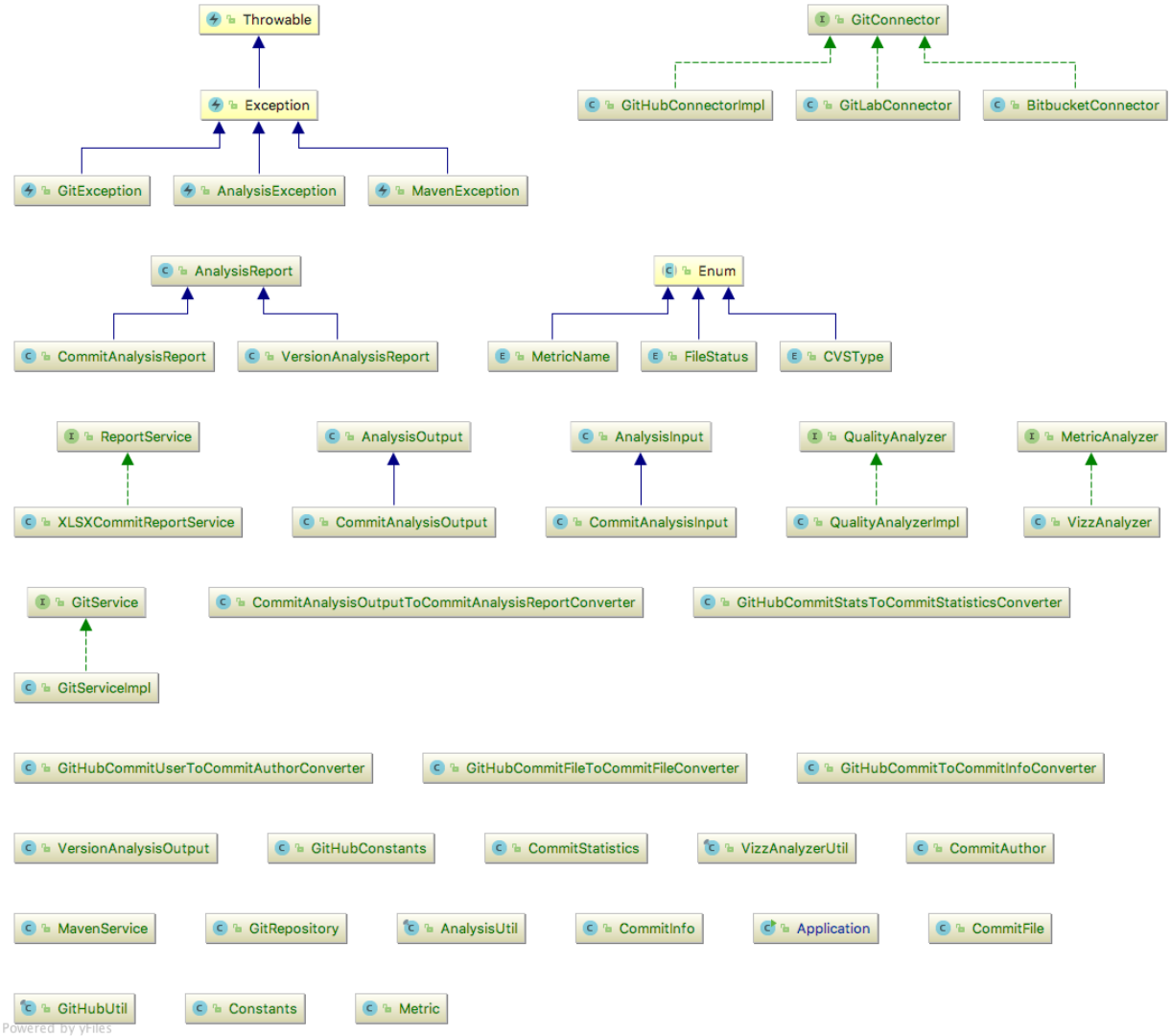


Рис. 3.3. Діаграма класів

3.4. Дослідження та вибір інструментів програмування

У цьому розділі описано основні інструменти та мови програмування, які використовувалися для розробки програмного забезпечення для аналізу якості.

3.4.1. Мова Java

Однією з найпопулярніших мов програмування, за даними GitHub [33], є Java. Java також дуже популярна в університетах. Студенти досить часто використовують його для навчальних і дипломних робіт. Цей інструмент також було розроблено з використанням Java.

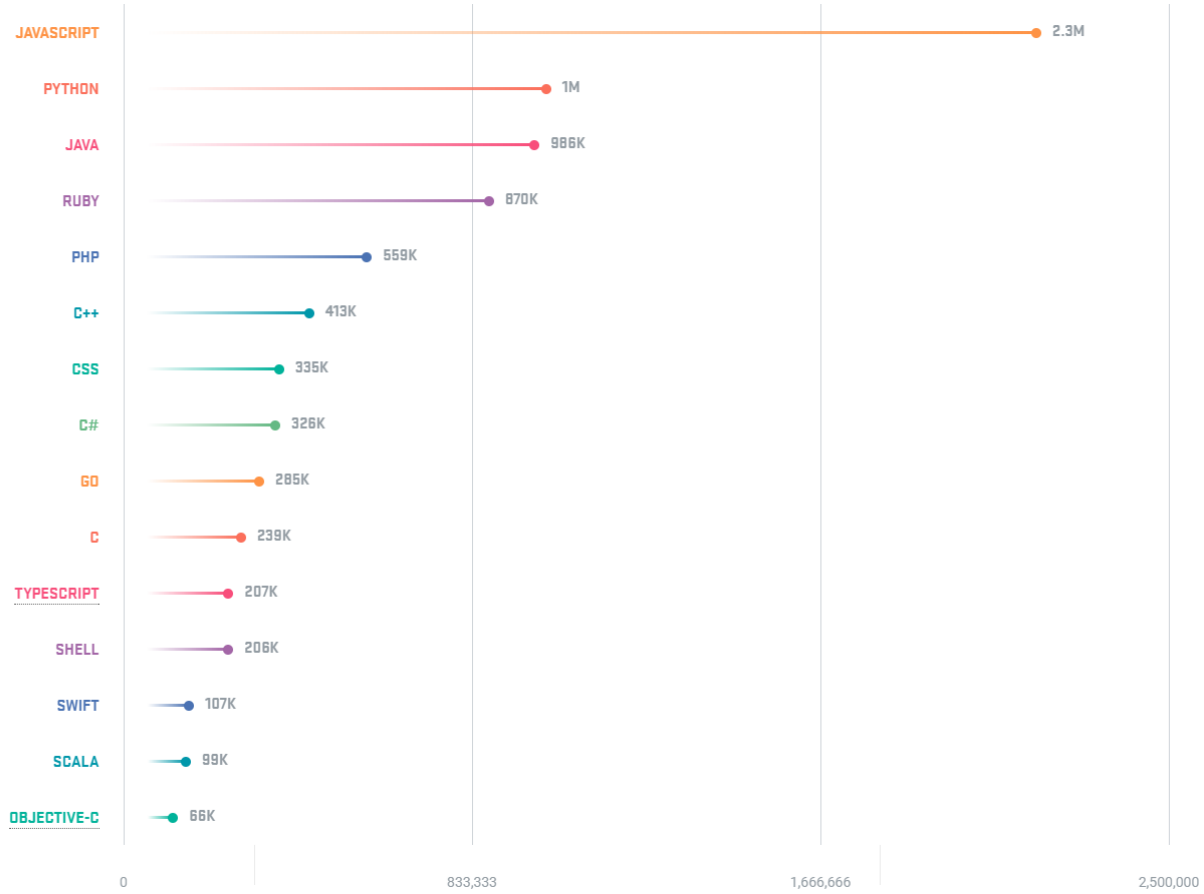


Рис. 3.4. Кількість запитів на отримання, створених на GitHub протягом 2024 року

Виходячи з наступних причин, було вирішено реалізувати програмне забезпечення на мові програмування Java:

1. У Java є багато вільнодоступних бібліотек сторонніх розробників, які можуть значно розширити функціональність базової мови, інкапсулювати багато складних завдань (наприклад, створення веб-додатків на основі Java EE у порівнянні з Spring Boot тощо) і допомогти розробникам багато робити

тривіальні речі швидше і зосереджуватися лише на найважливіших завданнях.

2. Бібліотека третьої сторони для вимірювання якості програмного забезпечення - VizzAnalyzzer™ була написана на Java. Це означає, що його можна імпортувати та використовувати лише в проектах на основі Java.

3. У центрі уваги аналізу якості програмного забезпечення в цьому дипломному проекті – аналіз проектів на основі Java.

3.4.2. Apache Maven

Для автоматизації збірок і керування залежностями під час розробки було вирішено використовувати Apache Maven. Відповідно до Apache, Maven — це «інструмент управління програмним проектом і розуміння. Базуючись на концепції об'єктної моделі проекту (POM), Maven може керувати збіркою проекту, звітністю та документацією з центральної частини інформації». [34]. Maven має багато переваг порівняно з іншими системами побудови та керування залежностями. Деякі з них [35, 36] перераховані нижче:

- Maven полегшує процес створення,

  Забезпечує єдину систему побудови,

  Має попередньо визначені цілі для виконання кількох певних завдань (наприклад, компіляції, пакування тощо),

  Дозволити завантаження залежностей (зовнішніх файлів JAR) через мережу,

  Має життєвий цикл із кількома попередньо визначеними фазами, такими як компіляція, упаковка, встановлення, розгортання тощо.

Як і Java, Maven також дуже популярний і простий у використанні. З огляду на те, що VizzAnalyzzer™ було написано на Java і можна було отримати доступ через репозиторій Softwerk AB Maven, було вирішено використати Maven як інструмент побудови та систему керування залежностями для цього дипломного проекту. У такому випадку вести розробку набагато легше.

3.4.3. Git

Для забезпечення безпеки вихідного коду проекту, усіх файлів і змін було вирішено використовувати систему контролю версій. Найбільш популярним і простим у використанні VCS на сьогодні є Git. Крім Git, вихідний код проекту зберігається в GitHub [32] (див. розділ 2). Це запобігає втраті інформації та забезпечує постійний доступ до коду.

3.5. Аналіз та візуалізація даних

Є багато різних варіантів візуалізації, які можна використовувати. У цьому проекті було вирішено не впроваджувати власний інструмент візуалізації, а використати один із існуючих. На сьогоднішній день є кілька найпопулярніших інструментів візуалізації та аналізу даних, таких як:

- Продукти Qlick: QlickView, QlickSence, QlickSence Cloud,
- Програмне забезпечення Tableu,
- Microsoft Power BI,
- SAS Visual Analytics,
- Microsoft Excel,
- D3js.
- RapidMiner,
- R (мова програмування).

Для візуалізації вихідних даних та проведення аналізу було вирішено використовувати Microsoft Power BI. Він повністю покриває всі потреби для цього дослідження і в той же час простий у запуску та не вимагає жодних спеціальних знань.

Microsoft Power BI — це «служба бізнес-аналітики, що надається Microsoft. Він забезпечує інтерактивну візуалізацію з можливостями бізнес-аналітики самообслуговування, де кінцеві користувачі можуть створювати звіти та інформаційні панелі самостійно, не залежачи від персоналу інформаційних технологій або адміністраторів баз даних» [37].

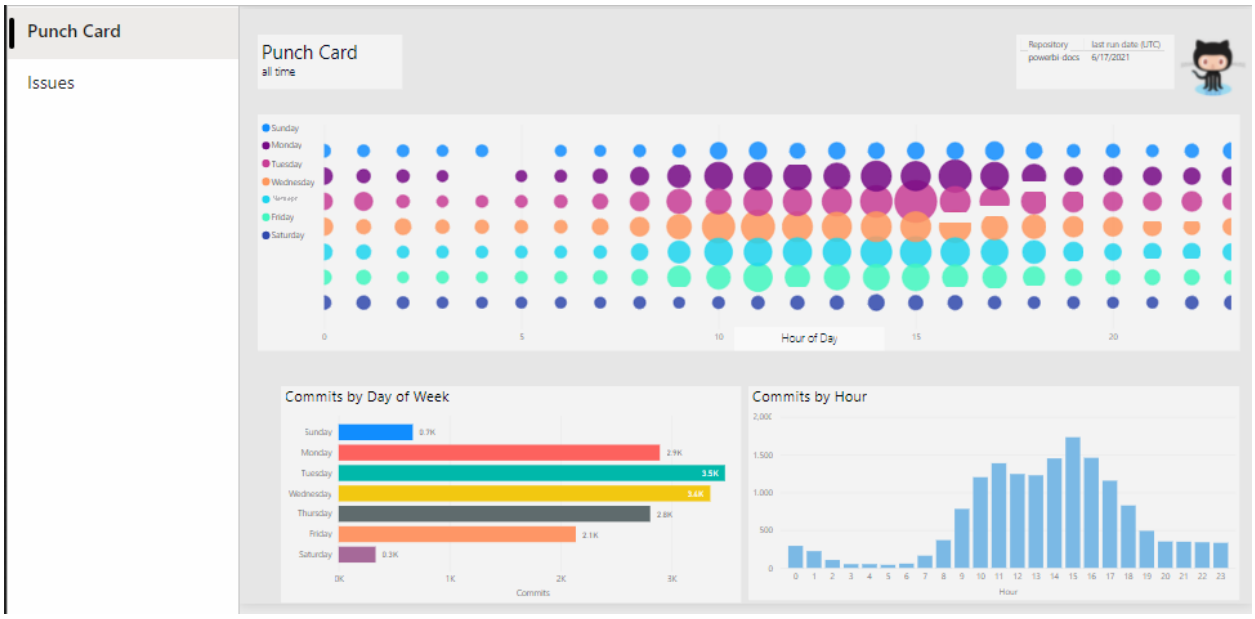


Рис. 3.5. Power BI підключений до GitHub

Power BI надає «хмарні служби BI, відомі як «Служби Power BI», а також інтерфейс на основі настільного комп'ютера під назвою «Power BI Desktop». Він пропонує можливості сховища даних, включаючи підготовку даних, виявлення даних та інтерактивні інформаційні панелі» [37].

Цей розділ також містить коротку інструкцію щодо використання інструменту аналізу якості в іншому проекті Java. По-перше, вам потрібно мати JAR-файл - quality-analyzer-l.0.jar. Потім його можна імпортувати в проект, наприклад, для проектів на основі Maven він має бути в папці ресурсів:

```
<dependency>
    <groupId>se.lnu.qualityanalyzer</groupId>
    <artifactId>quality-analyzer</artifactId>
    <version>1.0</version>
    <systemPath>${project.basedir}/src/main/resources/quality-analyzer-
1.0.jar</systemPath>
</dependency>
```

Для проведення аналізу необхідно підготувати вихідні дані. Існує 4 обов'язкових параметра:

1. Назва сховища вихідного коду,

2. Віддалена URL-адреса сховища в GitHub, GitLab або Bitbucket (наразі інструмент підтримує лише GitHub),

3. Власник сховища (ім’я користувача в службі хостингу Git),

4. Назва гілки, з якої будуть аналізуватися коміти,

5. Тип послуги хостингу Git: GitHub, GitLab або Bitbucket (наразі інструмент підтримує лише GitHub).

і два на вибір:

1. Кількість комітів, які необхідно проаналізувати,

2. Зміщення комітів, якщо аналіз потрібно починати не з верхнього коміту.

Нижче наведено зразок коду вхідних даних:

```
String repositoryName = "commons-beanutils";
String repositoryUrl = "https://github.com/apache/commons-beanutils.git";
String repositoryOwner = "apache";
String branchName = "trunk";
int numberOfCommits = 10;
int offsetOfCommits = 0;
```

Для виконання аналізу необхідно створити певний тип вхідних даних, створити об’єкт класу QualityAnalyzerImpl і запустити аналіз. Результатом буде список вихідних даних. Кожен з елементів списку є даними про один комміт у вказаній гілці, які включають KPI якості програмного забезпечення та функції сховища. Зразок коду наведено нижче:

```
CommitAnalysisInput input = new CommitAnalysisInput(
    VCSType.GIT_HUB,
    repositoryName,
    repositoryUrl,
    repositoryOwner,
    branchName,
    numberOfCommits,
    offsetOfCommits);
QualityAnalyzer analyzer = new QualityAnalyzerImpl();
List<CommitAnalysisOutput> output = analyzer.analyzeCommits(input);
```

Для створення звіту XLSX слід використовувати клас XLSXCommitReportService . Ця служба автоматично перетворить список вихідних даних у файл XLSX і збереже його на локальній машині. Зразок коду наведено нижче.

```
ReportService reportService = new XLSXCommitReportService();
reportService.create(input, output);
```

3.6. Оцінка застосування інструменту перевірки репозиторіїв програмного забезпечення

У цьому розділі описано застосування розробленого інструменту для проведення низки експериментів, основна мета яких полягає в тому, щоб зрозуміти, як функції сховища програмного забезпечення впливають на якість програмного забезпечення під час еволюції програмного забезпечення, і показано кореляцію між функціями сховища програмного забезпечення та якістю програмного забезпечення. Також з’ясуйте, які функції репозиторію найбільше впливають на якість програмного забезпечення.

Щоб відповісти на запитання дослідження, необхідно мати певну кількість даних, які можна проаналізувати. Щоб отримати ці дані, було вирішено провести кілька експериментів. У випадку цього дипломного проекту експеримент — це вимірювання якості програмного забезпечення та вилучення функцій репозиторію вихідного коду з подальшим встановленням кореляції між цими функціями та загальною якістю програмного забезпечення. Потім, коли всі дані отримані, можна візуалізувати та представити кореляцію на діаграмах розсіювання.

3.6.1. Підхід до вимірювання

Щоб виміряти якість вихідного коду та зібрати всі необхідні функції сховища вихідного коду, скористайтеся розробленим інструментом.

Розрахунок необхідно проводити для кожного проекту окремо. Тобто загалом це буде 3 різні аналізи. Для кожного з проектів необхідно проаналізувати близько 100-150 комітів. Це дасть нам можливість отримати дані за певний період часу і від різних розробників. В кінці кожного вимірювання буде створено звіт XLSX.

У цій роботі ми хочемо проаналізувати вплив функцій репозиторію в цілому, не зосереджуючись на конкретних проектах. Це дає нам можливість нейтралізувати вплив негативних факторів у конкретному проекті та отримати результати, які представляють різні проекти одночасно, включаючи вплив досвіду та навичок розробника. Для цього на фінальному етапі результати всіх експериментів будуть агреговані за середніми значеннями, візуалізовані, а потім можуть бути проаналізовані на кореляцію між функціями сховища програмного забезпечення та якістю програмного забезпечення.

Враховуючи, що розрахунок KPI якості займає досить багато часу, на цьому етапі складно аналізувати велику кількість комітів. Розділ 6 описує можливі варіанти того, як скоротити час обчислень і підвищити загальну продуктивність аналізу. У той же час вибрана кількість комітів все ще дозволяє нам зрозуміти, як функції репозиторію програмного забезпечення впливають на якість програмного забезпечення.

3.6.2. Вихідні проекти для вимірювань

Для проведення вимірювань використаємо три різні проекти на основі Java, Maven: Tillit, Apache Common Beanutils і JUnit4. Нижче наведено короткий опис кожного з них.

1) Tillit - веб-додаток для зберігання та перегляду інструкцій до будь-якої побутової техніки. Tillit має серверну та клієнтську частини, які розділені. Взаємодія між двома частинами базується на запитах HTTP. Beck-end — це веб-сервіси RESTful, засновані на Java та реалізовані за допомогою S pring Boot, MySQL і Liquibase. Front-end – це мобільна веб-програма з

відповідальним інтерфейсом користувача, написана з використанням JQuery, HTML5 і CSS3.

2) Apache Common BeanUtils. «Мова Java надає API Reflection і Introspection (пакети java.lang.reflect і java.beans у JDK Javadocs). Однак ці API можуть бути досить складними для розуміння та використання. [38]» Відповідно до [38], Apache Commons BeanUtils «надає просту у використанні та гнучку оболонку для рефлексії та самоаналізу». Він має 22 активних учасників і 87 розгалужень на GitHub.

3) JUnit4. Автори JUnit в [39] описують свій продукт як «просту структуру для написання повторюваних тестів». Вони також надають нам більш детальний опис фреймворку: «JUnit — це фреймворк модульного тестування для мови програмування Java. JUnit відіграв важливу роль у розвитку розробки, керованої тестуванням, і є одним із сімейства фреймворків модульного тестування, які спільно відомі як xUnit і походять від SUnit».

Дослідження, проведене серед 10 000 Java-проектів, розміщених на GitHub, показало, що JUnit (у порівнянні з slf4j-api) була найчастіше включеною зовнішньою бібліотекою. Кожну бібліотеку використовували 30,7% проектів». JUnit4 має майже 7000 зірок на GitHub, 2605 форків і 139 учасників.

Вибрані проекти реалізуються різними групами розробників з різними навичками та досвідом, і завдяки цьому під час експериментів ми дамо різні результати. Tillit створюється студентами і в основному представляє навички молодших розробників програмного забезпечення. JUnit — велика і дуже популярна бібліотека, якою користуються багато розробників у світі. Apache Common BeanUtils менша за JUnit4, але все ще досить популярна бібліотека. Разом ці 3 проекти представляють різні домени, різних розробників і різні рівні використання. Це робить їх гарним вибором для аналізу впливу функцій сховища програмного забезпечення на якість програмного забезпечення.

3.6.3. Аналіз та візуалізація даних

На наступному кроці необхідно розрахувати зміни якості у %. Цей спосіб дозволяє нам оперувати відносним значенням якості замість абсолютного, і, враховуючи це, усі отримані дані для трьох проектів можуть бути узагальнені та використані одночасно. Відповідно було вирішено використовувати Microsoft Power BI для візуалізації результатів та аналізу даних.

Давайте проаналізуємо вплив функцій репозиторію на якість програмного забезпечення: Загальна кількість змінених файлів у коміті, Додані файли, Змінені файли, Видалені файли, Загальна кількість змін у коміті, Доповнення, Видалення , Довжина повідомлення коміту.

На рисунку нижче показано точкову діаграму із загальною кількістю файлів і змінами якості у % за комітом.

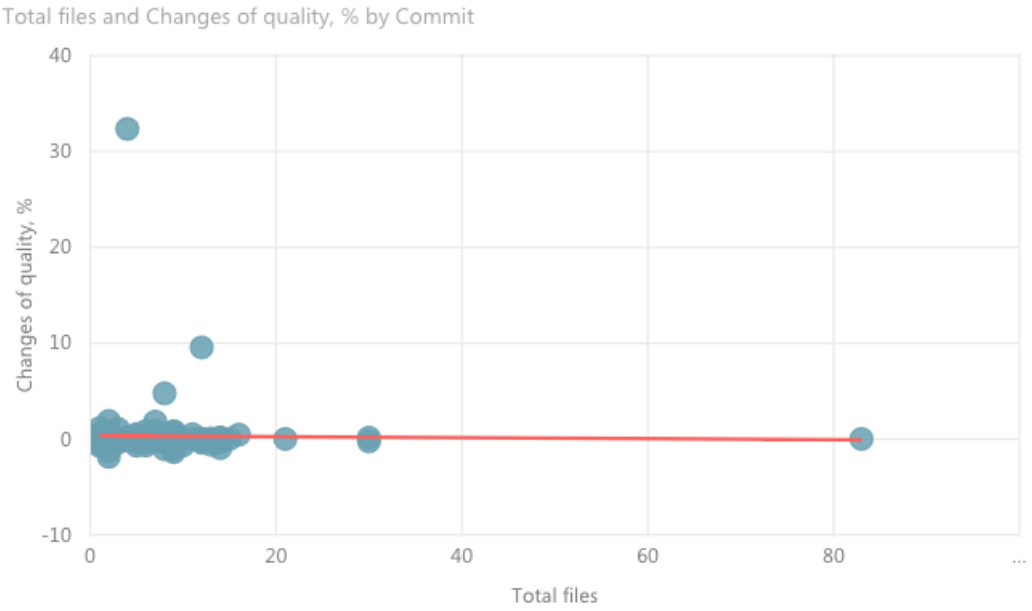


Рис. 3.6. Кореляція між загальною кількістю змінених файлів та змінами якості у %

Тут діапазон змін якості від -10% (зниження якості) до 32% (підвищення якості). Розглянемо більш детальний перегляд, фільтруючи значення по осях: Загальна кількість файлів < 40 і Зміни якості, % < 5.

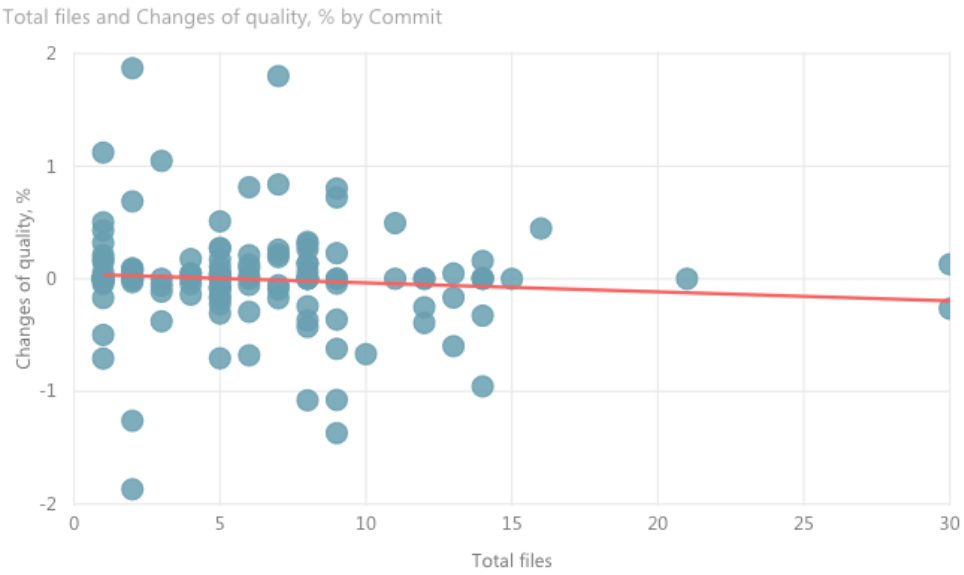
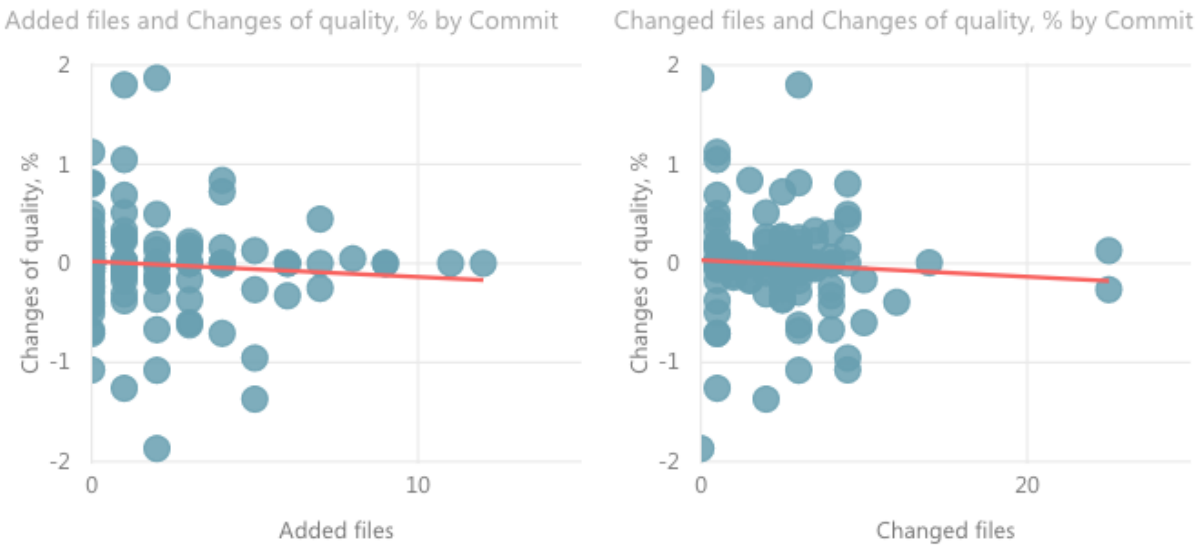


Рис. 3.7. Кореляція між загальною кількістю змінених файлів та змінами якості у %

Як видно з графіка, протягом еволюції програмного забезпечення якість вихідного коду змінювалася в межах 4%. Червона лінія на графіках відображає тенденцію кореляції. На рисунку вище майже немає жодної кореляції між кількістю змінених файлів і якістю програмного забезпечення. Конкретне числове значення кореляції можна знайти в кінці цього розділу.



а) б)

Рис. 3.8. Співвідношення між а – доданими файлами, б – зміненими файлами та змінами якості у %

Давайте детально розглянемо та створимо діаграми розсіювання для доданих, змінених і видалених файлів окремо. Щоб отримати тільки цінні значення, фільтри для осей застосовуються з самого початку.

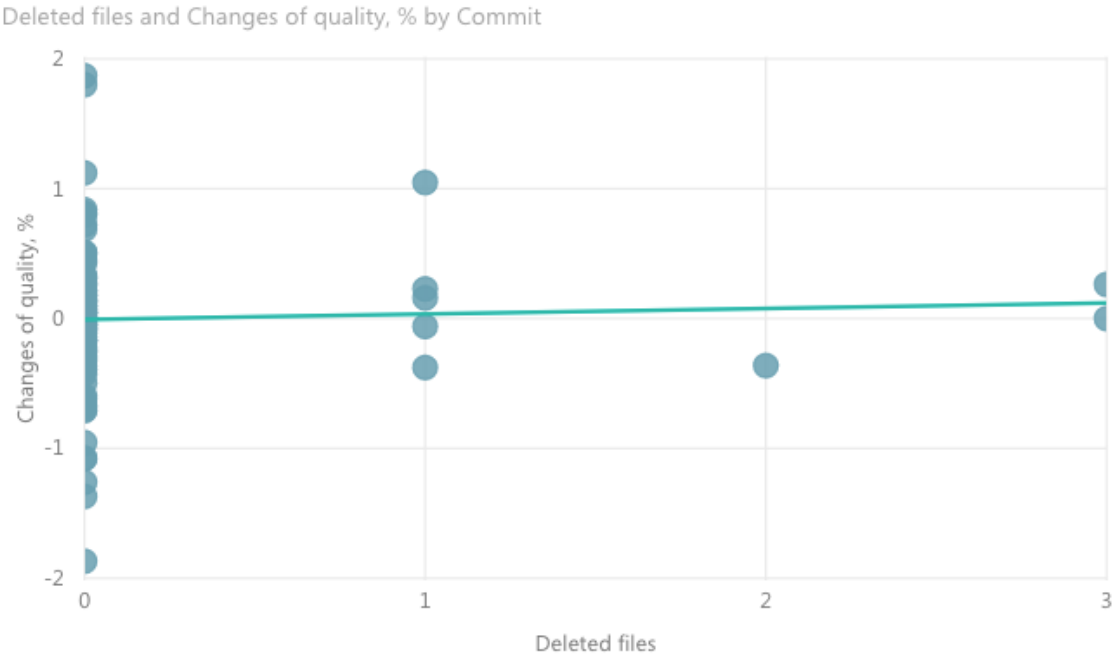


Рис. 3.9. Кореляція між видаленими файлами та змінами якості у %

Як бачимо, кореляція між видаленими файлами та зміною якості має позитивну тенденцію. Це означає, що коли розробник видаляє деякі файли вихідного коду, якість програмного забезпечення підвищується. Це можна інтерпретувати таким чином - кожен із файлів вихідного коду містить проблеми з якістю. Скільки файлів містить проект, стільки проблем з якістю . Таким чином, видалення файлів зменшує кількість проблем і підвищує загальну якість програмного забезпечення.

Тепер візуалізуємо кількість змін у файлах (додані, змінені, видалені рядки коду) і як вони впливають на якість програмного забезпечення.

Тут кореляція майже відсутня. Більш детальний графік на малюнку нижче показує нам, що зміни коду впливають на якість, але цей вплив занадто малий.

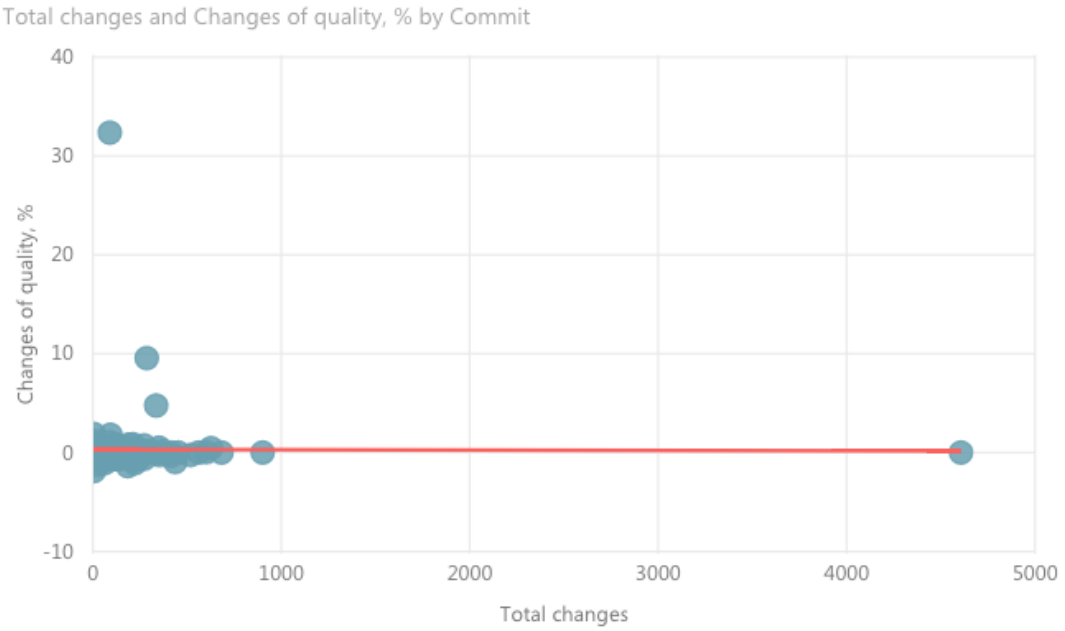


Рис. 3.10. Кореляція між загальними змінами та змінами якості у %

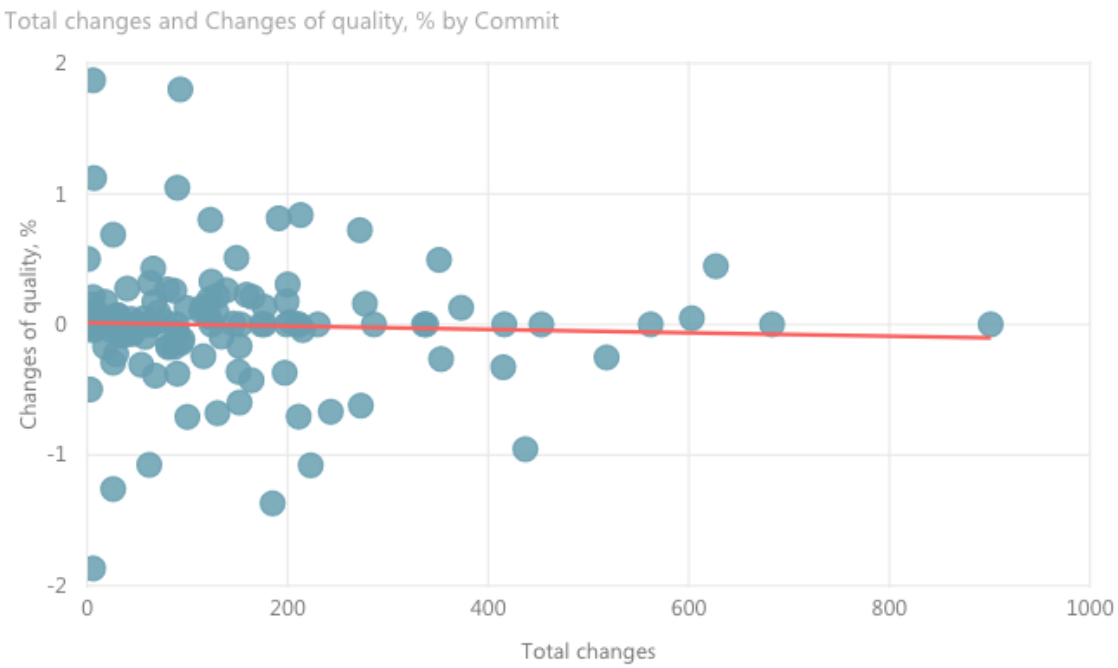


Рис. 3.11. Кореляція між загальною кількістю змін та змінами якості у %

Також нижче наведено детальні графіки для кожного типу змін (додавання, видалення).

Обидва наведені вище сюжети представляють ту саму ситуацію, що й майже всі інші сюжети – кореляція існує, але вона занадто мала, щоб говорити про реальний вплив.

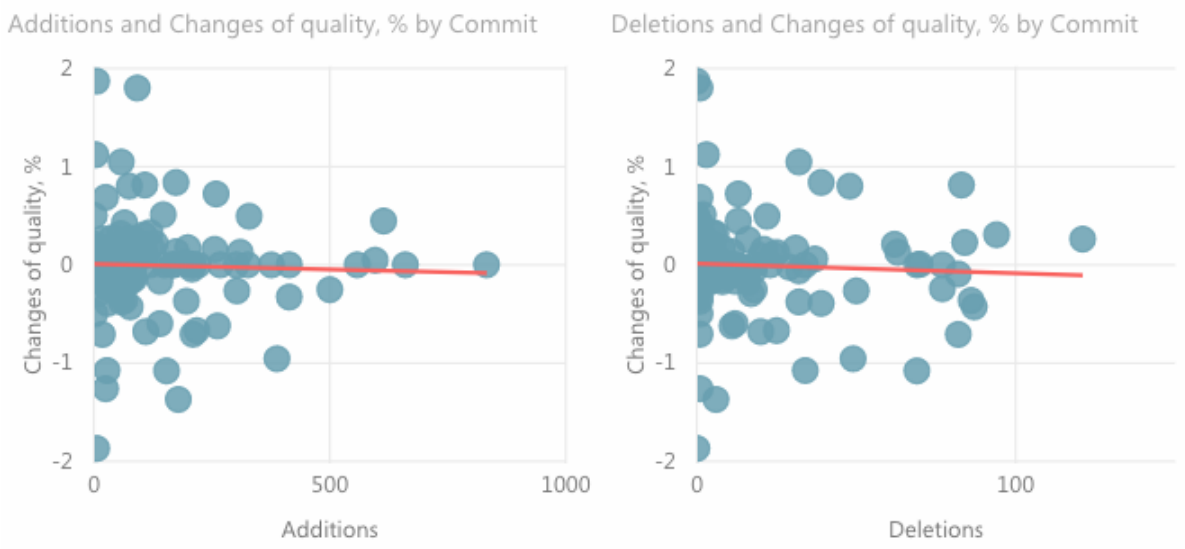


Рис. 3.12. Кореляція між а – додаваннями, б – видаленнями та змінами якості виражена у %

Функція останнього сховища - довжина повідомлення коміту. Цікаво, що чим довше повідомлення, тим краща якість коду. Причиною такої поведінки може бути те, що люди, які зазвичай витрачають час на написання описового повідомлення коміту, також піклуються про якість коду та створюють менше проблем із якістю або навіть виправляють існуючі.

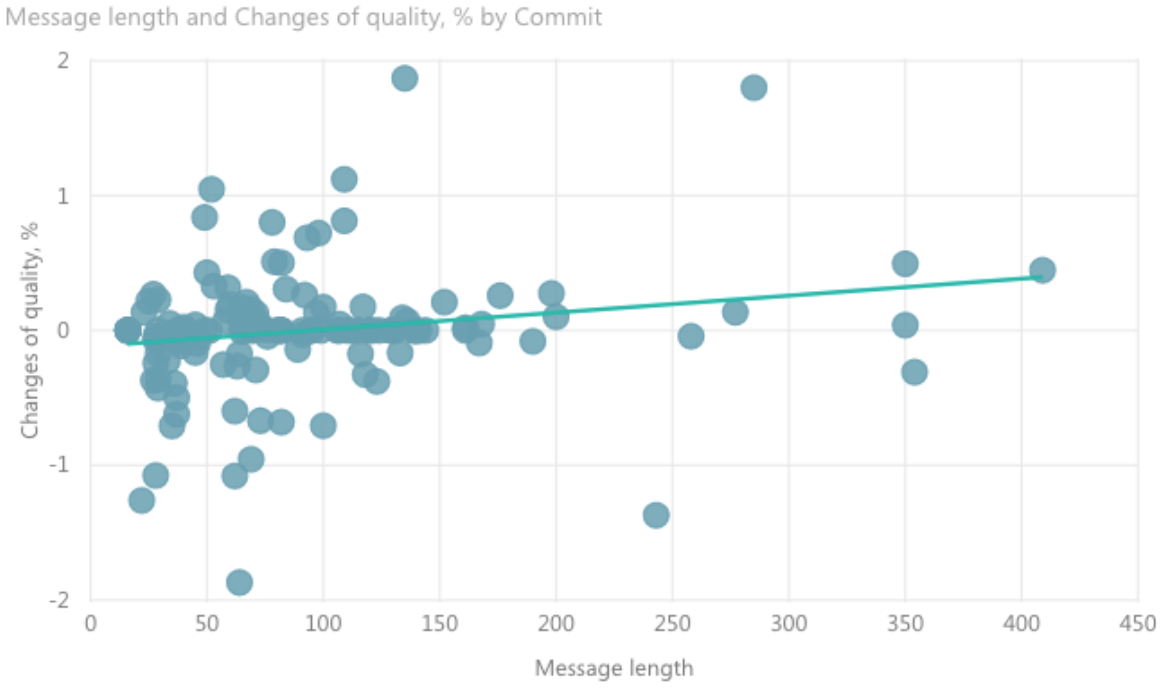


Рис. 3.13. Кореляція між довжиною повідомлення та змінами якості у %

Тепер розрахуємо кореляції між функціями сховища та загальним значенням якості для кожного з трьох вихідних проектів. Результати розрахунків (для перших 4 характеристик сховища) представлені в таблиці нижче.

Таблиця 3.3.

Кореляція між функціями сховища та загальним значенням якості

Project	Total files	Added files	Changed files	Deleted files
Tillit	-0.062	-0.094	-0.002	0.031
Comm. BeanUtils	-0.030	-0.132	-0.029	0.046
JUnit4	-0.035	0.088	-0.105	0.000
Average	-0.042	-0.046	-0.045	0.025

Результати для решти функцій сховища представлені в таблиці 3.4.

Таблиця 3.4.

Співвідношення між характеристиками сховища та загальним значенням якості

Project	Total changes	Additions	Deletions	Message length
Tillit	-0.031	-0.030	-0.023	0.010
Comm. BeanUtils	0.030	-0.068	0.116	0.084
JUnit4	-0.068	-0.058	-0.111	-0.034
Average	-0.023	-0.052	-0.006	0.020

Як правило, кореляція між функціями та якістю програмного забезпечення становить від -0,1 до 0.1. Це занадто мало, щоб сказати, що функції репозиторію якимось чином впливають на якість програмного забезпечення. Крім того, середні значення кореляції майже однакові для всіх ознак. Це означає, що жодна з цих функцій не має більшого впливу на якість програмного забезпечення, ніж інша.

На основі проведених експериментів і результатів аналізу даних можна зробити кілька основних і найважливіших висновків цього дослідження:

1. У процесі розвитку програмного забезпечення якість вихідного коду змінюється від -10% (зниження якості) до 30% (підвищення якості) на довгостроковому періоді та в межах 4% на короткостроковому.

2. Кореляції між якістю програмного забезпечення та кількістю доданих, змінених і видалених файлів майже немає.

3. Кореляція між видаленими файлами та зміною якості має позитивну тенденцію. Це означає, що коли розробник видаляє деякі файли вихідного коду, якість програмного забезпечення підвищується. Це можна інтерпретувати таким чином - кожен із файлів вихідного коду містить проблеми з якістю. Скільки файлів містить проект, стільки проблем з якістю. Таким чином, видалення файлів зменшує кількість проблем і підвищує загальну якість програмного забезпечення.

4. Зміни коду (додані та видалені рядки коду) впливають на якість, але цей вплив занадто малий і кореляція практично відсутня.

5. Чим довше повідомлення коміту, тим краща якість коду. Причиною такої поведінки можуть бути люди, які зазвичай витрачають час на те, щоб писати описове повідомлення коміту, також подбайте про якість коду та створюйте менше проблем із якістю або навіть виправляйте існуючі.

Крім того, всі поставлені раніше дослідницькі питання можна вважати відповідями. Кореляція між функціями та якістю ПЗ становить від -0,1 до 0,1. Це занадто мало, щоб сказати, що функції репозиторію якимось чином впливають на якість програмного забезпечення. Середні значення кореляції майже однакові за всіма ознаками. Це означає, що жодна з цих функцій не має більшого впливу на якість програмного забезпечення, ніж інша. Інструмент аналізу було впроваджено та використано для вимірювання всіх ключових показників ефективності та показників якості, а також вилучення функцій репозиторію програмного коду. Це повністю доводить можливість автоматизації процесу аналізу якості багатоверсійного програмного забезпечення через MSR.

Порівнюючи цю роботу з існуючими в області аналізу якості програмного забезпечення, можна сказати, що автори цих робіт не використовували підхід MSR. Усі роботи не мають зв'язку з функціями репозиторію програмного забезпечення, і, отже, весь раніше проведений аналіз описує якість програмного забезпечення лише з однієї, не MSR, перспективи, що можна інтерпретувати як слабкість або неповноту результатів, оскільки вони втрачати важливу, приховану інформацію, яку можна було б отримати зі сховищ програмного забезпечення та використати в майбутньому для запобігання помилкам, прийняття рішень та оптимізації процесів розробки.

Крім того, навіть ігноруючи відсутність MSR в існуючих роботах, вони також мають більш вузьке охоплення процесу аналізу якості. Наприклад, у [22] і [23] автори використовують лише кілька показників якості програмного забезпечення. У цьому випадку загальне значення якості не буде повноцінним. Це призведе до спотворення результатів. Наша робота усуває цей недолік, використовуючи 7 ключових показників ефективності з 21 метрикою якості. У [24] взагалі відсутній аналіз якості програмного забезпечення, а також розрахунок кореляції між структурними змінами вихідного коду та якістю програмного забезпечення. [25] і [26] описують вплив рефакторингу структури на внутрішню якість програми, що досить схоже на те, що ми досліджували в цій роботі, але відмінність полягає в тому, що вони зосереджені лише на процесі рефакторингу, тоді як ця теза має охоплено всі випадки, коли вихідний код може бути змінений, включаючи рефакторинг. Враховуючи це, внесок цієї роботи дозволить майбутнім дослідникам проводити більш детальні та комплексні дослідження та аналіз.

3.7. Висновки до розділу

Отже, в цьому розділі представлено комплексний підхід до імплементації методів Data Mining для аналізу репозиторіїв програмного

забезпечення. Розроблено архітектуру інструменту, обрано програмні засоби та протестовано систему на практичних прикладах. Отримані результати підтверджують практичну значущість і доцільність використання розробленого інструменту для забезпечення якості програмного забезпечення.

Метод дослідження базується на використанні інтегрованих підходів до аналізу даних репозиторіїв, включаючи попередню обробку, застосування алгоритмів Data Mining, а також оцінку результатів. Запропонований підхід дозволяє отримати структуровану інформацію про якість репозиторіїв на основі історичних даних

ВИСНОВКИ

В магістерській роботі розглянуто застосування методів та моделей Data Mining для аналізу репозиторіїв програмного забезпечення.

У першому розділі проведено аналіз предметної області застосування майнінгу даних для сховищ програмного забезпечення. Встановлено, що сховища програмного забезпечення є цінним джерелом даних для аналізу процесів розробки, оцінки якості програмного забезпечення та прогнозування дефектів. Описано ключові показники ефективності, які дозволяють об'єктивно вимірювати якість репозиторіїв, серед яких метрики коду, процесу та команди. Визначено основні проблеми дослідження, пов'язані з підвищенням точності аналізу, адаптацією методів майнінгу до різнорідних даних та створенням інструментів для автоматизації процесів.

У другому розділі розглянуто моделі та методи функціонування репозиторіїв і сховищ вихідного коду для процесів майнінгу. Описано особливості функціонування систем контролю версій (СКВ), які виступають базовим інструментом для організації роботи з репозиторіями. Досліджено різні типи СКВ (централізовані та розподілені) і програмні засоби для керування версіями, зокрема Git. Визначено архітектуру інструментів для аналізу якості програмного забезпечення, яка включає рівні збору, обробки та візуалізації даних. Проведено огляд існуючих досліджень у галузі майнінгу сховищ програмного забезпечення, що підтверджує актуальність і ефективність цього підходу.

У третьому розділі представлено практичну реалізацію методів Data Mining для аналізу репозиторіїв програмного забезпечення. Запропоновано метод дослідження, який включає збір, обробку та аналіз даних із використанням сучасних алгоритмів та інструментів програмування.

Розроблено архітектуру програмного інструменту, яка дозволяє проводити оцінку якості репозиторіїв за допомогою обраних метрик та візуалізувати результати.

Обрано оптимальні інструменти для реалізації системи, включаючи мову Java, платформу Apache Maven та систему контролю версій Git. Проведено тестування розробленого інструменту на реальних проєктах із відкритим кодом. Результати підтвердили його здатність ефективно ідентифікувати дефекти та оцінювати якість розробки.

Отже, магістерська робота охоплює комплексний аналіз предметної області, дослідження сучасних методів і моделей функціонування репозиторіїв, а також практичну реалізацію інструменту для майнінгу даних. Отримані результати демонструють можливість ефективного використання методів Data Mining для покращення процесів аналізу якості програмного забезпечення, забезпечуючи тим самим підвищення продуктивності розробки та її кінцевої якості.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance and Evolution: Research and Practice, 2007, 19:77-131.
2. M. A. Gerosa. Mining Sociotechnical Information From Software Repositories. Informatics Seminar, University of California, Irvine, 2014.
3. H. Barkmann, R. Lincke, and W. Lowe. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. Software Technology Group, School of Mathematics and Systems Engineering, Vaxjo University, Sweden, 2009.
4. M. Shaw. What Makes Good Research in Software Engineering? International Journal of Software Tools for Technology Transfer, 2002, vol. 4, no. 1, pp. 1-7.
5. Version Management, 2025. [Online]. Available: https://its.unl.edu/bestpractices/version-management.
6. What is version control, 2025. [Online]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control.
7. Version control, 2012. [Online]. Available: https://www2.le.ac.uk/services/research-data/organise-data/version-control.
8. Getting Started – About Version Control, 2018. [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control.
9. What is Git, 2018. [Online]. Available: https://www.atlassian.com/git/tutorials/what-is-git.
10. GitHub, [Online]. Available: https://github.com.
11. GitLab, [Online]. Available: https://about.gitlab.com.
12. Bitbucket, [Online]. Available: https://bitbucket.org.
13. Sunday O. Olatunji, Syed U. Idrees, Yasser S. Al-Ghamdi, Jarallah Saleh Ali Al-Ghamdi. Mining Software Repositories – A Comparative Analysis.

IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.8, August 2010.

14. Software Quality Metrics, 2009. [Online]. Available: http://www.arisa.se/compendium/node88.html.
15. W. Löwe. VizzAnalyzer – A Reverse Engineering Framework. Software Technology Group, MSI, University of Växjö, Sweden, 2004.
16. W. Löwe, M. Ericsson, J. Lundberg, T. Panas, N. Petersson. VizzAnalyzer – A Software Comprehension Framework. Software Technology Group, MSI, University of Växjö, Sweden, 2004.
17. S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the Influence of Identifier Names on Code Quality: An empirical study. 14th European Conference on Software Maintenance and Reengineering, Madrid, Spain, 2010, pp. 156–165.
18. D. Nemer. IMPEX: An Approach to Analyze Source Code Changes on Software Run Behavior. Journal of Software Engineering and Applications, 2013, vol. 6, no. 4, pp. 157-167.
19. Č. Gerlec, M. Heričko. Analyzing Structural Software Changes: A Case Study. BCI-LOCAL 2012, Novi Sad, Serbia, 2012, pp. 117-120.
20. B. D. Bois, T. Mens. Describing the impact of refactoring on internal program quality. ELISA workshop, Amsterdam, The Netherlands, 2003.
21. K. S. Herzig. Capturing the long-term impact of changes. 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 2010, pp. 393-396.
22. P. Good. Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses, 2nd ed. New York: Springer-Verlag, 2000.
23. G. Cumming. Understanding The New Statistics: Effect Sizes, Confidence Intervals, and Meta-Analysis. New York: Routledge, 2012.
24. D. G. T. Denison, C. C. Holmes, B. K. Mallick, A. F. M. Smith. Bayesian Methods for Nonlinear Classification and Regression. John Wiley, 2002.

25. S. L. Pfleeger. Experimental design and analysis in software engineering. Annals of Software Engineering 1. London: Centre for Software Reliability, City University, 1995.
26. Apache Maven Project, [Online]. Available: https://maven.apache.org.
27. Apache Maven Project – Introduction, [Online]. Available: https://maven.apache.org/what-is-maven.html.
28. Hassan, A. E. (2008). Mining software repositories: The quest for data. Proceedings of the 2008 Frontiers of Software Maintenance, 5–10. IEEE. DOI: 10.1109/FOSM.2008.4659253
29. Menzies, T., & Cukic, B. (2006). The PROMISE repository of software engineering databases: Past, present, and future. Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR), 1–6. ACM. DOI: 10.1145/1137983.1137994
30. Bird, C., Gourley, A., et al. (2006). Mining software repositories for social network metrics. Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR), 137–143. ACM. DOI: 10.1145/1137983.1138016
31. Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering, 33(1), 2–13. DOI: 10.1109/TSE.2007.256941
32. Zimmermann, T., Weißgerber, P., Diehl, S., & Zeller, A. (2005). Mining version histories to guide software changes. IEEE Transactions on Software Engineering, 31(6), 429–445. DOI: 10.1109/TSE.2005.72
33. Kim, S., & Whitehead Jr., E. J. (2006). Change classification approach to software maintenance effort prediction. Proceedings of the 2006 International Conference on Software Engineering (ICSE), 771–780. ACM. DOI: 10.1145/1134285.1134388
34. Nagappan, N., & Ball, T. (2006). Mining bug databases for unidentified software problems. Proceedings of the 21st IEEE/ACM International

Conference on Automated Software Engineering (ASE), 403–406. IEEE. DOI: 10.1109/ASE.2006.53

35. Graves, T., & Karr, A. (2005). Predicting faults from cached history. Proceedings of the 27th International Conference on Software Engineering (ICSE), 123–131. ACM. DOI: 10.1145/1062455.1062495
36. Nagappan, N., & Ball, T. (2006). Mining metrics to predict component failures. Proceedings of the 28th International Conference on Software Engineering (ICSE), 452–461. ACM. DOI: 10.1145/1134285.1134354
37. Catal, C., & Diri, B. (2009). A survey on software defect prediction using data mining techniques. Proceedings of the IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE), 274–279. DOI: 10.1109/SEKE.2009.123
38. Zheng, A., Zhang, L., & Kim, S. (2012). Mining software repositories for comprehensive fault localization. Proceedings of the International Conference on Software Engineering (ICSE), 234–243. IEEE. DOI: 10.1109/ICSE.2012.6227153
39. Khoshgoftaar, T. M., & Seliya, N. (2005). Data mining techniques for software engineering: An overview. Advances in Computers, 65, 143–185. DOI: 10.1016/S0065-2458(05)65003-2
40. Kagdi, H., Collard, M. L., & Maletic, J. I. (2007). Mining software repositories for improving software maintenance. Data & Knowledge Engineering, 61(3), 398–426. DOI: 10.1016/j.datak.2006.06.007
41. Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing, 27, 504–518. DOI: 10.1016/j.asoc.2014.11.020
42. Lo, D., & Khoo, S. C. (2009). SMArTIC: Towards building an accurate, robust and scalable specification miner. Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 265–275. ACM. DOI: 10.1145/1181775.1181796

43. Hindle, A., German, D. M., & Holt, R. (2008). What do large commits tell us?: A taxonomical study of large commits. Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR), 99–108. ACM. DOI: 10.1145/1370750.1370772
44. Hindle, A., Barr, E. T., Gabel, M., Su, Z., & Devanbu, P. (2012). On the naturalness of software. Proceedings of the 34th International Conference on Software Engineering (ICSE), 837–847. ACM. DOI: 10.1109/ICSE.2012.6227135
45. Mockus, A., & Weiss, D. (2000). Predicting risk of software changes. Bell Labs Technical Journal, 5(2), 169–180. DOI: 10.1002/bltj.2232
46. Hassan, A. E., & Holt, R. C. (2004). Studying the evolution of software systems using evolutionary code extractors. Proceedings of the 2004 International Working Conference on Reverse Engineering (WCRE), 76–85. IEEE. DOI: 10.1109/WCRE.2004.18
47. German, D. M. (2004). Mining CVS repositories: The softChange experience. Proceedings of the 1st International Workshop on Mining Software Repositories (MSR), 17–21. IEEE. DOI: 10.1109/MSR.2004.1280706
48. Gradle vs Maven Comparison, 2016. [Online]. Available: https://gradle.org /maven-vs-gradle.
49. Commons BeanUtils, 2016. [Online]. Available: https://commons.apache. org/proper/commons-beanutils.
50. P. D. Ellis. The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results, 1st ed. Cambridge, 2010.