БАКАЛАВРСЬКА РОБОТА

БР. ІП - 10.00.00.000 ПЗ

Група ІП-21-4

Макарук Арсеній

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Макарук Арсеній Васильович

(прізвище, ім'я, по батькові)

УДК 004.4

(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка методології графічного супроводу процесу розробки ПЗ на основі класифікованих ресурсів GIT

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього рівня Макарук А.В.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Корнута Володимир Андрійович, к.т.н., доцент

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

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.

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

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2025 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Макаруку Арсенію Васильовичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “**Розробка методології графічного супроводу процесу розробки ПЗ на основі класифікованих ресурсів GIT”**

керівник проекту (роботи) Корнута Володимир Андрійович

затверджені наказом закладу вищої освіти від **“** 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 11 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз проблематики графічної візуалізації розробки програмного забезпечення
2. Методологія та методи статистичного аналізу вихідного коду java
3. Проектування служби аналізу коду
4. Імплементація методології графічного супроводу розробки ПЗ на основі ресурсів GIT
5. Оцінка працездатності методології

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

1. Вигляд завдань в GitLab (рис. 1.1)
2. Огляд комунікації gRPC (рис. 1.2)
3. Архітектура ExplorViz (рис. 1.3)
4. Приклад візуалізації даних під час виконання в ExplorViz (рис. 1.4)
5. Модель даних з полями метрик (рис. 2.1)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **28 квітня 2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз проблематики графічної візуалізації розробки програмного забезпечення	04.05.2025	виконано
2	Методологія та методи статистичного аналізу вихідного коду java	15.05.2025	виконано
3	Проектування служби аналізу коду	25.05.2025	виконано
4	Імплементація методології графічного супроводу розробки ПЗ на основі ресурсів GIT	02.06.2025	виконано
5	Оцінка працездатності методології	07.06.2025	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	11.06.2025	виконано

Студент – дипломник _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 83 сторінки, 35 рисунків, список використаних джерел із 35 найменуваннями.

Метою роботи є розробка та імплементація методології графічного супроводу, яка поєднує статистичний аналіз Java-коду, обробку історії змін і візуалізацію результатів для підтримки ефективного управління програмними проєктами.

Об'єкт дослідження - процес розробки програмного забезпечення із використанням систем контролю версій, зокрема Git.

Предмет дослідження - методи та засоби графічного супроводу процесу розробки ПЗ на основі аналізу вихідного коду Java і метаданих репозиторіїв Git.

В першому розділі окреслено ключові вимоги до методології візуалізації та визначено інструментарій, що дозволяє реалізувати комплексну систему супроводу розробки ПЗ.

В другому розділі розроблено структуру обробки вихідного коду Java із використанням визначених метрик складності, а також забезпечено ефективне керування даними Git-репозиторіїв.

В третьому розділі успішно реалізовано архітектуру та компоненти системи, що забезпечують автоматизований аналіз, збір метрик та візуалізацію змін у вихідному коді з прикладними сценаріями тестування.

Висновок: розроблено методологію та реалізовано інструмент для графічного супроводу процесу розробки ПЗ з використанням структурованих даних і класифікованих ресурсів із репозиторіїв Git

КЛЮЧОВІ СЛОВА: ГРАФІЧНА ВІЗУАЛІЗАЦІЯ, ВИХІДНИЙ КОД, АНАЛІЗ КОДУ, МЕТРИКИ ЯКОСТІ, GIT, РЕПОЗИТОРІЙ, СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ, DOCKER, МЕТОДОЛОГІЯ РОЗРОБКИ, СТРУКТУРОВАНІ ДАНІ

ANNOTATION

The bachelor's thesis contains 83 pages, 35 figures, a list of used sources with 35 names.

The purpose of the work is to develop and implement a graphical support methodology that combines statistical analysis of Java code, processing of change history and visualization of results to support effective management of software projects.

The object of the study is the process of software development using version control systems, in particular Git.

The subject of the study is methods and tools for graphical support of the software development process based on the analysis of Java source code and Git repository metadata.

The first section outlines the key requirements for the visualization methodology and defines the tools that allow implementing a comprehensive software development support system.

The second section develops a structure for processing Java source code using defined complexity metrics, and also ensures effective data management of Git repositories.

In the third section, the architecture and components of the system that provide automated analysis, metrics collection and visualization of changes in the source code with applied testing scenarios have been successfully implemented.

Conclusion: a methodology has been developed and a tool has been implemented for graphical support of the software development process using structured data and classified resources from Git repositories

KEYWORDS: GRAPHICAL VISUALIZATION, SOURCE CODE, CODE ANALYSIS, QUALITY METRICS, GIT, REPOSITORY, VERSION CONTROL SYSTEMS, DOCKER, DEVELOPMENT METHODOLOGY, STRUCTURED DATA

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

SDP - Software Development Process

AST - Abstract Syntax Tree

VCS - Version Control System

Git - Розподілена система контролю версій

CI/CD - Continuous Integration / Continuous Delivery

gRPC - Google Remote Procedure Call

JVM - Java Virtual Machine – віртуальна машина Java

LOC - Lines of Code – кількість рядків коду

Docker - платформа для контейнеризації застосунків

ВСТУП

У сучасних умовах розробки програмного забезпечення важливу роль відіграє не лише створення функціонального коду, але й забезпечення його зрозумілості, підтримуваності та прозорості для всієї команди розробників. За умов швидких змін у вимогах і високої динаміки проєктів особливої актуальності набувають засоби візуалізації, які дають змогу представити складні технічні характеристики програмного продукту у доступній формі.

Традиційні підходи до моніторингу та аналізу коду часто не враховують історію змін, структуру репозиторіїв і внутрішню динаміку розвитку програмних систем. Це створює ризики для підтримки якості продукту, своєчасного виявлення технічного боргу та прийняття обґрунтованих інженерних рішень.

Актуальність роботи

У сучасному процесі розробки програмного забезпечення ефективна візуалізація коду та динаміки змін у репозиторіях є критично важливою для розуміння архітектури проєкту, виявлення технічного боргу та оптимізації командної взаємодії. Зі збільшенням складності ПЗ та кількості учасників у розробці виникає необхідність у створенні інструментів, які б забезпечували автоматичне отримання, обробку й графічне подання ключових характеристик коду та метрик з репозиторіїв. Особливої актуальності набувають методи глибокого аналізу Java-коду із застосуванням сучасного технологічного інструментарію (GitLab, JavaParser, gRPC, Docker), що дозволяють не лише проводити статичний аналіз, а й інтегрувати результати у процес неперервної розробки.

Зростання масштабів і складності сучасних програмних систем зумовлює потребу в ефективному візуальному супроводі процесів розробки, що забезпечує прозорість, контроль і якісну оцінку стану проєктів. Особливу актуальність має інтеграція засобів аналізу вихідного коду та систем

контролю версій у єдине візуальне представлення, що дозволяє виявляти закономірності, зміни та потенційні проблеми у програмному продукті на ранніх етапах.

Одним із підходів до реалізації такої задачі є побудова методології, яка ґрунтується на класифікації структурних та історичних даних із репозиторіїв програмного забезпечення. У цьому дослідженні проаналізовано сучасні інструменти та технології, визначено ключові метрики оцінки якості коду, запропоновано архітектуру та реалізовано систему для графічного супроводу процесу розробки ПЗ на основі аналізу репозиторіїв Git.

Метою роботи є розробка та імплементація методології графічного супроводу, яка поєднує статистичний аналіз Java-коду, обробку історії змін і візуалізацію результатів для підтримки ефективного управління програмними проєктами.

Завдання дослідження

Проаналізувати існуючі підходи до візуалізації програмного забезпечення.

Визначити набір метрик для аналізу якості вихідного коду Java.

Розробити архітектуру інструменту для інтеграції з Git-репозиторіями.

Реалізувати програмні модулі для збору, обробки та візуалізації структурованих даних.

Провести експериментальну оцінку працездатності запропонованої методології.

Об’єкт дослідження - процес розробки програмного забезпечення із використанням систем контролю версій, зокрема Git.

Предмет дослідження - методи та засоби графічного супроводу процесу розробки ПЗ на основі аналізу вихідного коду Java і метаданих репозиторіїв Git.

Методи дослідження

- Статичний аналіз вихідного коду Java;

- Методи програмної інженерії для виявлення структурних і функціональних характеристик програм;
- Аналіз метрик складності, зчеплення, глибини вкладеності;
- Використання бібліотек JGit, JavaParser, gRPC, Docker для реалізації сервісної архітектури;
- Емпіричне тестування на прикладних сценаріях.

Наукова новизна

Запропоновано методологію, що об’єднує аналіз вихідного коду Java із класифікацією змін у Git-репозиторіях, а також реалізовано архітектуру мікросервісної системи для автоматизованого збору та візуалізації метрик, що може бути інтегрована у CI/CD-процеси.

Практичне застосування

Розроблений інструмент може бути використаний у командах розробників для контролю якості коду, моніторингу динаміки змін у проєкті, полегшення рев’ю коду, підтримки документації та прийняття інженерних рішень.

Запропонована у роботі методологія графічного супроводу розробки ПЗ, що поєднує аналіз вихідного коду, статистичні метрики, історію змін та сучасні інструменти візуалізації, є актуальною відповіддю на ці виклики. Вона спрямована на підвищення ефективності командної розробки, підтримку процесів безперервної інтеграції та управління складністю програмних систем.

Бакалаврська робота містить 83 сторінки, 35 рисунків, 3 розділи список використаних джерел із 35 найменуваннями.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ ГРАФІЧНОЇ ВІЗУАЛІЗАЦІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Передумови розробки методології візуалізації

Рішення для моніторингу та аналізу програмного забезпечення набувають зростаючого значення протягом процесу розробки. Постійна перевірка проектів програмного забезпечення та огляд коду, що знаходиться у виробництві, є невід'ємною частиною підтримки кодової бази програмного забезпечення. Для виявлення потенційних слабких місць у проектуванні програмного забезпечення або надто складної структури вихідного коду часто застосовується статичний аналіз програм. Висока складність вихідного коду ускладнює його розуміння розробниками, що негативно впливає на супроводжуваність кодової бази. Відтак, аналіз програмного забезпечення з точки зору складності та взаємозв'язків є важливим для покращення його розуміння.

Оскільки вихідний код часто керується в репозиторіях Git, поєднання даних статичного аналізу з метриками історії Git сприяє виявленню потенційних недоліків проектування, ідентифікуючи файли з високою частотою звернень та змін.

У даній роботі представлена реалізація початкового етапу для графічної візуалізації коду робустними, розширюваними та легко конфігурованими можливостями статичного аналізу, здатними обчислювати та інтегрувати метрики Git. Аналітична служба розроблена з урахуванням легкої інтеграції в конвеєри неперервної інтеграції для забезпечення актуальних оглядів проектів, що ініціюються змінами в репозиторії Git. Буде продемонстровано принципи проектування та реалізації такого статичного аналізу, особливості обробки репозиторіїв Git та підходи до забезпечення розширюваності служби. Крім того, буде показано можливості служби

аналізувати довільні проекти Java як у локальному середовищі (контейнер Docker), так і в інтегрованих середовищах неперервної інтеграції (наприклад, на основі GitLab). Оскільки на момент написання даної дисертації супровідна візуалізація для згенерованих даних відсутня, корисність створених даних буде досліджено в майбутніх роботах. Git є однією з поширених систем керування версіями та широко використовується. Її структура, що відображає постійну еволюцію вихідного коду з можливістю відстеження минулих змін, може розкривати внутрішні процеси розробки, зокрема динаміку еволюції кодової бази.

Розробники витрачають значну частину часу (близько 50-60%) на задачі з обслуговування, зокрема на розуміння програмного коду. Це підкреслює актуальність розробки вдосконалених інструментів, які б сприяли підвищенню ефективності виконання цього завдання.

Зважаючи на те, що історичне версіонування, надане системою Git, може дати уявлення про еволюцію компонентів програмного забезпечення, виникає необхідність у вилученні структурних даних безпосередньо з вихідного коду. Інші статичні дані та метрики, зокрема показники складності класів, також можуть бути проаналізовані або отримані з вихідного коду. Для забезпечення максимальної корисності для користувача, необхідно ідентифікувати значущі та цінні дані для вилучення.

Пропонований продукт є інструментом для візуалізації живих трас виконання програмного забезпечення. Він досліджує потенціал візуалізації програмного забезпечення як засобу полегшення розуміння програм. Розроблювана система сфокусована на аналізі поведінки програмного забезпечення під час виконання (runtime) і є особливо придатним для моніторингу великих програмних ландшафтів, зосереджуючись на візуалізації динамічних даних. Його розширення шляхом інтеграції інструменту статичного аналізу підвищить його ефективність, дозволяючи

оперативно здійснювати огляд кодової бази проекту для виявлення певних архітектурних проблем.

Незважаючи на наявність численних інструментів статичного аналізу, їх застосування часто обмежується випадками легкої інтеграції в існуючі робочі процеси. Зацікавленість розробників у вихідному коді часто перевищує їхній інтерес до обчислених метрик, що пов'язано зі складністю доступу та генерації даних статичного аналізу. Відтак, надання результатів аналізу у вигляді легко інтегрованого рішення, що не вимагає значної конфігурації, забезпечить доступ розробника до цінних даних без додаткових зусиль. У зв'язку із зазначеними обмеженнями інших підходів, у даній дисертації розробляється легко інтегрований, готовий до інтеграції в CI (Continuous Integration) інструмент статичного аналізу для проектів Java та їхніх репозиторіїв Git. Метою є генерація даних для подальшої візуалізації.

1.2. Опис основних задач, що вирішуються в роботі

Основна мета даної роботи полягає в розробці служби аналізу вихідного коду. Зазначена служба буде здійснювати аналіз коду Java з репозиторіїв Git, збирати структурні дані та метрики коду, а також забезпечувати їх передачу до віддаленого пункту призначення даних.

Задачі, що вирішуються:

1. Визначення цінних даних для розуміння програмного забезпечення.

Великі програмні системи зі значною кількістю файлів та складною структурою каталогів можуть суттєво ускладнювати процес їх розуміння. Відповідно, задачі супроводу та розширення можуть бути ускладнені через час, необхідний інженерам програмного забезпечення або розробникам для ідентифікації задіяних компонентів системи. Метрики та дані, отримані з вихідного коду, можуть сприяти розумінню міжкласової взаємодії. З іншого боку, історичні дані з Git-комітів можуть інформувати про пакети, які

перебувають у стадії активної розробки. Метою є ідентифікація значущих даних з вихідного коду та репозиторію Git, а також підготовка цих даних для майбутньої візуалізації.

2. Реалізація аналізу вихідного коду

Наступним кроком є реалізація модуля аналізу вихідного коду. Кожен файл вихідного коду в проекті Java буде проаналізовано на предмет даних та метрик, визначених на етапі G1. Метою є розробка фундаментального статичного аналізу вихідного коду, який може бути застосований до будь-якого наданого проекту. Особлива увага приділяється коректному визначенню типів, зважаючи на їхнє значне контекстне значення у програмних системах.

3. Обробка даних репозиторіїв

Оскільки багато проектів використовують Git як систему керування версіями вихідного коду, а керовані репозиторії мають тенденцію до зростання складності та розміру, обчислювальні ресурси, необхідні для повного аналізу програмного забезпечення при кожному коміті, також зростатимуть. Підхід до аналізу, чутливий до змін (commit-sensitive), є рішенням для скорочення часу обчислень. Лише зміни між послідовними комітами будуть включені в процес аналізу. Реалізація, передбачена у G2, відповідає вимогам G3, якщо при додаванні нового коміту до моніторингового репозиторію аналізується лише цей новий коміт для оновлення повного представлення даних.

4. Забезпечення мережевої взаємодії

На даному етапі згенеровані дані є доступними лише локально і мають бути передані на віддалений пункт призначення. Пункт призначення функціонує як простий шлюз, який може бути інтегрований з ExplorViz. У випадку, якщо аналіз не може бути виконаний для певних комітів, дані аналізу можуть бути відсутніми. Безпосередньо після виклику, служба аналізу запитує стан даних пункту призначення (ідентифікатор останнього

переданого коміту) і може відправити відсутні дані, проаналізовані як пакетне завдання, до досягнення актуального стану.

5. Контейнеризація (Docker) для інтеграції в CI/CD

Фінальним етапом розробки є контейнеризація інструменту аналізу шляхом створення самодостатнього Docker-образу, що може бути інтегрований у конвеєр безперервної інтеграції/безперервного розгортання (CI/CD). Аналізатор буде ініціюватися новими комітами в репозиторії Git, забезпечуючи таким чином актуальність даних для візуалізації. Оскільки аналіз є безстановим (stateless), згенерований Docker-контейнер є легковичним і повністю видаляється після виконання, що спрощує розгортання та зменшує споживання апаратних ресурсів.

6. Оцінка

На фінальному етапі необхідно продемонструвати працездатність служби у конкретному застосуванні, перевірити її стабільність для великих проектів та підтвердити легкість інтеграції. Буде проведено оцінку ефективності контейнерного підходу та корисності згенерованих даних.

1.3. Підходи до розуміння програмного забезпечення та статичний аналіз даних

Розуміння програм є галуззю досліджень, що вивчає когнітивні процеси, пов'язані з осмисленням програмного забезпечення, зокрема його вихідного коду. Оскільки програмне забезпечення проходить етапи розробки, супроводу та розширення, зрозумілість кодової бази має постійно контролюватися, оскільки незрозумілий код не підлягає ефективному супроводу.

У цій галузі сформульовано дві класичні теоретичні моделі:

- Нисхідний підхід (Top-Down) - цей підхід значною мірою спирається на попередні знання розробника у відповідній програмній області. Під час

процесу розуміння програми висуваються гіпотези стосовно її функціонування. Зазначені гіпотези зазвичай є високорівневими припущеннями, які верифікуються або спростовуються шляхом детальнішого вивчення коду. Цей ітераційний процес триває до рівня найменших компонентів, забезпечуючи, таким чином, повне розуміння програми.

- Висхідний підхід (Bottom-Up) – розробник прагне зрозуміти програму шляхом послідовного аналізу її складових частин, починаючи з рівня найменших компонентів. За відсутності попередніх знань про систему, початковим етапом є розуміння базових компонентів. Шляхом поступового агрегування та формування абстракцій вищого рівня досягається розуміння всієї програми.

Зважаючи на те, що більшість розробників володіють певними знаннями з попередніх проектів, на практиці вони часто застосовують обидві стратегії розуміння одночасно [3].

З метою покращення загального розуміння програм, особливо на рівні високорівневих абстракцій, можуть застосовуватися візуальні засоби, зокрема візуалізація. Вони можуть полегшити процес осмислення програмного забезпечення для розробників. Оскільки дані аналізу покликані сприяти кращому розумінню програмного забезпечення розробниками, необхідним є визначення типів даних, що найбільшою мірою сприяють зрозумілості.

На відміну від динамічного аналізу, що вивчає код під час виконання, статичний аналіз передбачає дослідження програмного забезпечення без його запуску. Статичний аналіз може бути застосований безпосередньо до вихідного коду, а також до скомпільованих бінарних файлів для виявлення потенційних помилок. При застосуванні статичного аналізу до вихідного коду можуть бути побудовані графи викликів, що надають аналітикам або розробникам уявлення про потенційні виклики функцій та наявні залежності [6].

Незважаючи на можливість статичного аналізу практично будь-якого типу файлів, у контексті даної роботи доцільним є обмеження розгляду лише аналізом вихідного коду Java. Типові інструменти статичного аналізу для Java включають інтегровані середовища розробки (IDE), як-от Eclipse та IntelliJ IDEA, з їхніми вбудованими функціями підтримки розробки (наприклад, перевірка синтаксису, пропозиції коду), а також зовнішні інструменти, подібні до Checkstyle, призначені для моніторингу якості вихідного коду. Оскільки аналізуються репозиторії Git, історичні дані про зміни в коді, що фіксуються системою керування версіями, можуть бути включені як додатковий матеріал для аналізу, окрім самого вихідного коду. Виходячи з контексту даної роботи, дані репозиторію аналізуються статично.

1.4. Технологічний інструментарій для реалізації

1.4.1. Неперервна інтеграція

Неперервна інтеграція (Continuous Integration, CI) описує автоматизацію процесів збірки та перевірки програмних проектів. Її метою є виявлення помилок збірки на ранніх етапах розробки, що зумовлює її широке застосування в індустрії та сприяє прискоренню процесу розробки [5]. Безперервне розгортання (Continuous Delivery, CD) являє собою наступний етап автоматизації, сфокусований на створенні надійних версій програмного забезпечення з високою частотою. Добре спроектований конвеєр CI/CD може підвищити швидкість розробки та загальну продуктивність. Проте, неефективно спроектований конвеєр CI/CD може призвести до негативних наслідків.

Для ефективної підтримки розробників, інструменти мають надавати релевантну інформацію у найбільш потрібний момент процесу розробки. Оперативна доступність інформації забезпечується шляхом автоматизованої

попередньої підготовки даних (у контексті даної роботи – у момент їх фіксації в репозиторії Git).

1.4.2. GitLab

GitLab є провідною платформою для хостингу вихідного коду, що забезпечує керування версіями за допомогою Git та підтримку практик DevOps. З моменту першого релізу в 2011 році платформа позиціонується як відкрита альтернатива конкуренту GitHub, пропонуючи версію GitLab Community Edition (CE) під ліцензією MIT. GitLab може бути використаний як самостійне рішення або через хмарний сервіс (програмне забезпечення як послуга, SaaS) на gitlab.com. Платформа пропонує багатофункціональну екосистему конвеєрів CI/CD із вбудованими та легко конфігурованими ранерами.

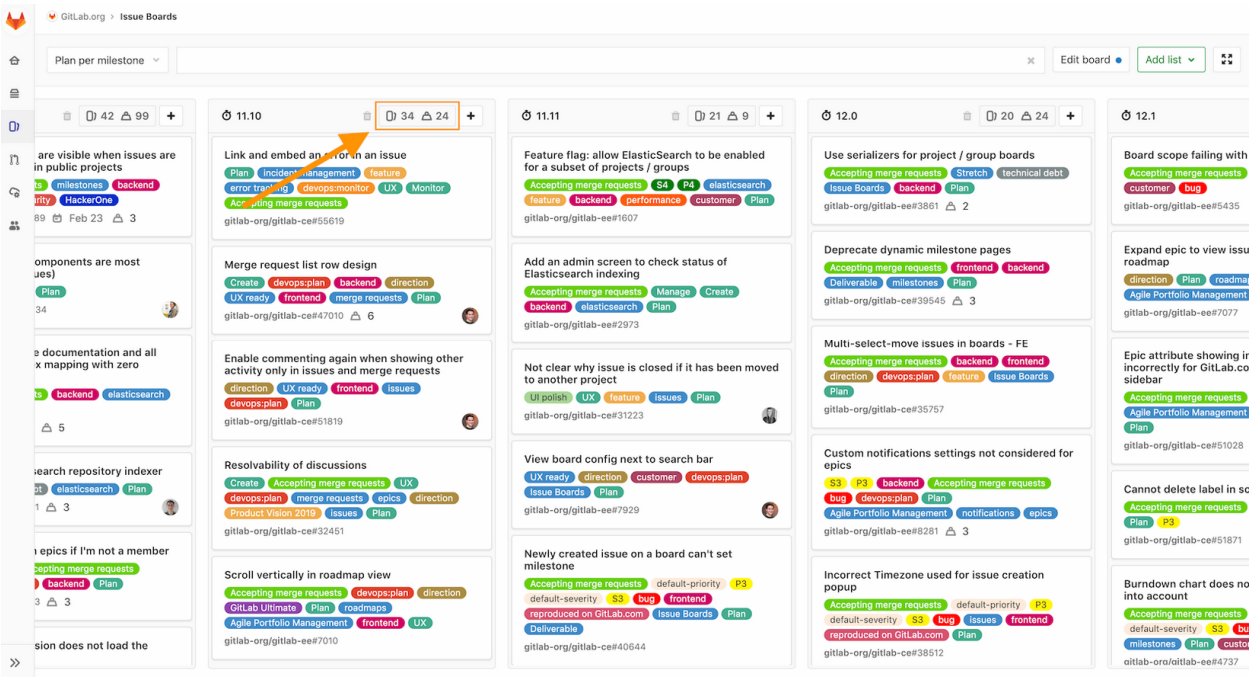


Рисунок 1.1 – Вигляд завдань в GitLab

Завдання, визначені в конвеєрі CI/CD, можуть використовувати попередньо зібрані Docker-контейнери. Це дозволяє включати функції

перевірки, верифікації або аналізу як етапи, що виконуються після збірки (post-build stages), для моніторингу поточної якості проекту.

Базовий план сервісу GitLab SaaS є безкоштовним, включаючи використання інтегрованого конвеєра CI/CD та його ранерів. Це робить його оптимальним вибором для академічних проектів, проектів з відкритим вихідним кодом або приватних проектів, що вимагають складних рішень для автоматизації розробки з функціоналом забезпечення якості. GitLab використовується як середовище виконання завдань CI (CI-runner), а контейнер адаптується спеціально для забезпечення легкої інтеграції з цим середовищем.

1.4.3. JGit

Eclipse JGit є реалізацією системи керування версіями Git з відкритим вихідним кодом у вигляді бібліотеки Java. Вона надає можливість програмного доступу до репозиторіїв Git. Хоча JGit пропонує обмежений інтерфейс командного рядка (CLI), подібний до стандартного Git CLI, проте менш функціональний, його основним призначенням не є повна заміна останнього. Деякі команди Git (так звані 'porcelain commands') можуть бути викликані через API для спрощеного доступу до високорівневих функцій без необхідності використання CLI. Однак, основна увага приділяється керуванню внутрішніми ідентифікаторами об'єктів Git. Для цього використовується механізм, відомий як RevWalk, що забезпечує обхід дерева комітів Git. JGit буде використаний для аналізу локального репозиторію Git та перевірки різних гілок з метою отримання необхідних даних з віддаленого репозиторію.

1.4.4. JavaParser

JavaParser є бібліотекою, написаною на Java, що дозволяє розробникам взаємодіяти з кодом Java як з об'єктним представленням. Код Java

перетворюється в представлення Абстрактного Синтаксичного Дерева (Abstract Syntax Tree, AST). AST складається з ієрархічно розташованих вузлів, кожен з яких містить інформацію про свої структурні властивості. Наприклад, атрибути, пов'язані з вузлом, що представляє метод у коді Java, можуть включати PublicAccessSpecifier, MethodDefinition та VoidDataType. Це надає всебічне уявлення про кожен структурний елемент аналізованої програми. JavaParser призначений не тільки для аналізу вихідного коду; він також надає функціональність для його модифікації. Незважаючи на потенційну складність великих AST для розуміння, JavaParser пропонує механізми, що дозволяють користувачам більш інтуїтивно здійснювати пошук та модифікацію AST. Таким чином, користувач може сконцентруватися на виявленні релевантних патернів. Розширенням JavaParser є пакет JavaSymbolSolver, який здійснює ідентифікацію та повертає інформацію про типи з AST. Він надає функціональність для виявлення як вбудованих типів Java, так і типів, специфічних для конкретного проекту.

JavaParser та його компонент JavaSymbolSolver використовуються для виконання статичного аналізу та визначення типів даних.

1.4.5. Бібліотека gRPC

Google розробила gRPC як відкриту кросплатформенну бібліотеку для віддаленого виклику процедур (Remote Procedure Call, RPC), призначену для взаємодії між розподіленими застосунками. gRPC підтримує одно- та двонаправлену, блокуючу та неблокуючу комунікацію між клієнтом і сервером, а також потокову передачу даних (streaming). Він використовує HTTP/2 як транспортний протокол. Структура даних у gRPC, на відміну від HTTP, де зазвичай застосовується JSON, є більш строго визначеною. gRPC використовує Protocol Buffers як фреймворк для серіалізації даних [12].

На Рисунку 1.2 ілюструється принцип роботи gRPC. Слід зазначити, що служба та клієнти можуть бути реалізовані різними мовами програмування. Реалізація інструменту аналізу як клієнта gRPC дозволяє розробити сервер gRPC будь-якою іншою мовою, забезпечуючи таким чином можливість інтеграції аналізу в проекти, що використовують мови програмування відмінні від Java.

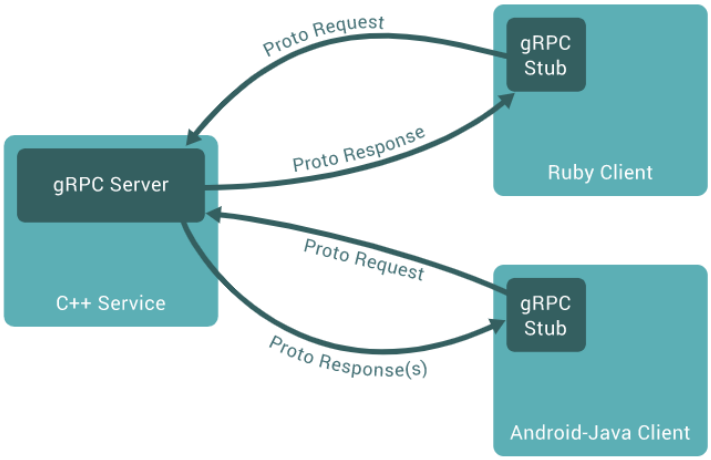


Рисунок 1.2 - Огляд комунікації gRPC

1.4.6. Protocol Buffers

Protocol Buffers (скорочено protobuf) — це формат даних та набір інструментів для серіалізації. Файли з розширенням .proto визначають структуру даних, подібну до полів у класах Java, де примітивні типи належать до об'єкта повідомлення. protoc (Protocol Buffer Compiler) — це компілятор та генератор коду protobuf. Він створює необхідні методи для роботи з даними, забезпечуючи простий та повністю детермінований процес отримання та встановлення значень, незалежно від використовуваної мови програмування чи системи. Повідомлення protobuf характеризуються зворотною та прямою сумісністю, компактністю завдяки бінарному представленню, проте не мають самоописуючих властивостей, притаманних, наприклад, формату JSON.

Protobuf використовується як формат даних для комунікації gRPC, а також для визначення внутрішнього формату даних, що застосовується у розробленій службі. В лістингу 1.1 представлено уривок визначення повідомлення MethodData, яке безпосередньо використовується як об'єкт для зберігання даних. Protobuf підтримує вкладені визначення повідомлень. Наприклад, на рядку 4 показано використання повідомлення ParameterData як типу вкладеного поля.

Лістинг 1.1. Фрагмент визначення повідомлення MethodData

```
message MethodData {
    string returnType = 1;
    repeated string modifier = 2;
    repeated ParameterData parameter = 3;
    repeated string outgoingMethodCalls = 4;
    bool isConstructor = 5;
    map<string, string> metric = 6;
}
message ParameterData {
    string name = 1;
    string type = 2;
    repeated string modifier = 3;
}
```

1.4.7. Docker

Docker є програмною платформою, що надає розробникам можливість пакувати та розгортати застосунки разом із їхніми залежностями в ізольованих контейнерах. Контейнери Docker є легковичними, портативними та самодостатніми середовищами виконання, які функціонують незалежно від особливостей апаратного та операційного оточення. Розробники можуть створювати контейнери, що інкапсулюють усі необхідні компоненти застосунку, та розгортати їх на будь-якій машині із встановленим Docker. Це суттєво спрощує процеси розробки та розгортання застосунків у різнорідних середовищах. Крім того, платформа надає можливість обміну та розповсюдження контейнерів через публічні або приватні репозиторії. Це полегшує надання готових до використання функціональних рішень.

1.4.8. Фреймворк Quarkus

У зв'язку зі зростанням популярності розгортання великомасштабних контейнеризованих програмних рішень, фреймворк Quarkus пропонує спрощений підхід до розгортання Java-застосунків у середовищі Kubernetes. Фреймворк дотримується концепції 'container-first', оптимізований для низького споживання пам'яті та швидкого часу запуску, що робить його особливо придатним для реактивних та розподілених застосунків.

Quarkus використовується як фреймворк для реалізації служби аналізу в рамках даної роботи.

1.5. Опис архітектури інструменту для аналізу поведінки Java-застосунків ExplorViz

ExplorViz є інструментом для аналізу поведінки Java-застосунків під час виконання (runtime). Він призначений для аналізу великих програмних ландшафтів, які набувають все більшого поширення в сучасній індустрії у зв'язку зі зростанням складності програмних систем. Надаючи інсайти стосовно програмного забезпечення, зокрема щодо взаємодії між компонентами та потоку управління в застосунках, інструмент покликаний сприяти кращому розумінню великомасштабних програмних систем інженерами програмного забезпечення. Вирішення архітектурних проблем полегшується при ідентифікації причин їх виникнення. Завдяки спеціалізованій візуалізації наданої інформації, навіть у сценаріях спільної роботи (що дозволяє кільком клієнтам одночасно спостерігати за візуалізованим програмним ландшафтом), ExplorViz сприяє прискоренню процесів вдосконалення програмного забезпечення. Масштабованість реалізована на рівні архітектури.

Короткий огляд архітектури ExplorViz представлено на рисунку 1.3. Розроблена служба буде інтегрована в середовище моніторингу (ліва частина

рисунка 1.3), відповідальне за збір даних, та виконуватиме функції, подібні до Adapter Service, представленого в середовищі аналізу. На подальших етапах аналіз та візуалізація будуть розширені для обробки новостворених структурних даних.

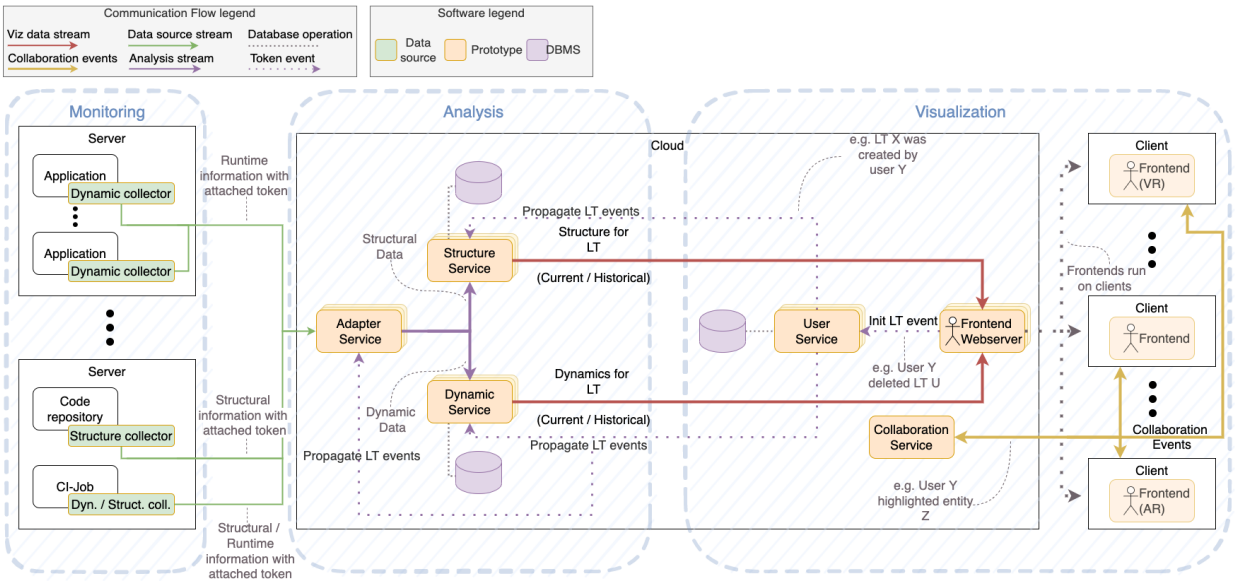


Рисунок 1.3 - Архітектура ExplorViz

На рисунку 1.3 представлена архітектура системи, яка складається з трьох основних блоків: моніторинг, аналіз та візуалізація.

Моніторинг відповідає за збір даних виконання (runtime) із запущених застосунків та структурних даних із репозиторіїв коду. Збір може ініціюватися через CI-завдання. Зібрані дані передаються до блоку Аналізу.

Аналіз отримує дані від Моніторингу через Adapter Service. Містить сервіси для обробки та зберігання структурних та динамічних даних (поточних та історичних), зокрема Dynamic Service та Structure Service. Також відповідає за поширення подій виконання (LT events).

Візуалізація отримує оброблені дані від блоку Аналізу. Включає Frontend/Webserver та User Service, які надають інтерфейс для користувача. Обробляє події співпраці та комунікаційні події з клієнтських застосунків (зокрема AR/VR) та надсилає/отримує потоки даних візуалізації.

Загальний потік даних йде від Моніторингу → Аналізу → Візуалізації, з додатковими потоками для взаємодії та подій між компонентами

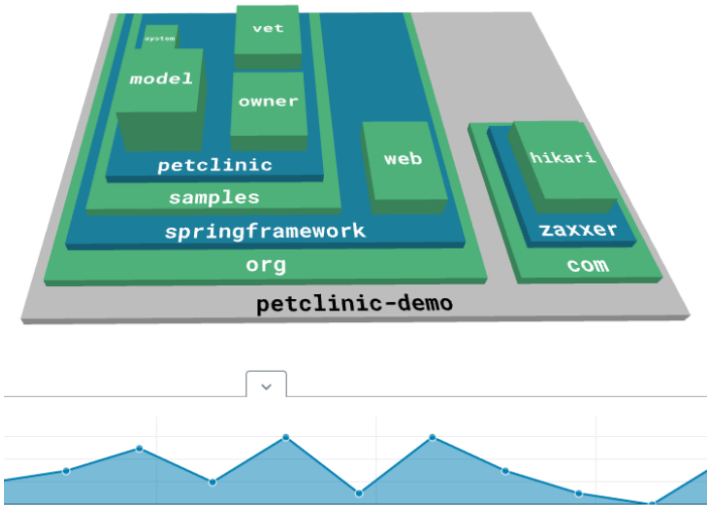


Рисунок 1.4 - Приклад візуалізації даних під час виконання в ExplorViz; використовується як зразок для майбутньої візуалізації структурних даних

Рисунок 1.4 демонструє візуалізацію поведінки програмного забезпечення під час виконання в ExplorViz. Незважаючи на динамічний характер візуалізованих даних, представлення поведінки під час виконання є корисним для розуміння бажаного вигляду візуалізації структурних даних, яка планується бути подібною. Оскільки інтерфейс ExplorViz надає можливість перегляду конкретних динамічних знімків даних, елементи користувацького інтерфейсу (наприклад, у нижній частині екрана) потенційно можуть бути використані для вибору коміту Git при візуалізації структурних даних, за умови відповідної реалізації.

Висновки до розділу

У першому розділі було розглянуто теоретичні та прикладні аспекти, пов’язані з проблематикою графічної візуалізації процесу розробки

програмного забезпечення. Встановлено, що зростання складності програмних систем та потреба в забезпеченні їхньої зрозумілості для різних учасників розробки зумовлюють актуальність побудови ефективних методів і засобів візуалізації. Сформульовано основні задачі дослідження, серед яких — визначення інформаційних моделей, що підлягають візуалізації, а також побудова відповідного технологічного стеку.

Проаналізовано сучасні підходи до розуміння структури програмного забезпечення, зокрема на основі статичного аналізу коду, що є ключовим для побудови адекватних візуальних представлень. У якості інструментальних засобів запропоновано використання таких технологій, як GitLab, JGit, JavaParser, gRPC, Protocol Buffers, Docker і Quarkus, які забезпечують як ефективний доступ до вихідного коду, так і гнучку інтеграцію в середовище розробки.

Окрему увагу приділено аналізу архітектури ExplorViz як прикладу інструменту для моніторингу та візуалізації виконання Java-застосунків, що дозволяє оцінити як переваги подібних рішень, так і існуючі обмеження. Це створює підґрунтя для подальшої розробки власного підходу до побудови візуалізацій, які відображають не лише структуру, а й поведінку програмних систем.

РОЗДІЛ 2. МЕТОДОЛОГІЯ ТА МЕТОДИ СТАТИСТИЧНОГО АНАЛІЗУ ВИХІДНОГО КОДУ JAVA

У цьому розділі обговорюються принципи виконання статичного аналізу вихідного коду Java та типи даних, які можуть бути зібрані. Крім того, буде розглянуто обробку проектів Java, що керуються в репозиторіях Git, принципи проектування аналізу як служби, а також обґрунтовано обмеження та причини вибору безстанової (stateless) архітектури у вигляді попередньо зібраного Docker-контейнера. Оскільки розробка розширення ExplorViz для візуалізації згенерованих даних аналізу не входить до обсягу даної дисертації, проектування здійснюється з орієнтацією на віртуальну, ще не реалізовану версію візуалізації, що базується на принципах візуалізації динамічних даних (див. рис. 1.4).

2.1 Аналіз вихідного коду

У розробці програмного забезпечення вихідний код слугує засобом вираження та фіксації інструкцій для виконання комп'ютером. Отже, для розуміння та аналізу програмного забезпечення точна синтаксична репрезентація вихідного коду не є необхідною і може ускладнювати процес аналізу. З метою ефективного аналізу кодової бази програми часто застосовується її перетворення в Абстрактне Синтаксичне Дерево (Abstract Syntax Tree, AST) перед початком аналізу, що спрощує наступний статичний аналіз. Компілятори часто використовують представлення AST для перевірки на наявність помилок перед генерацією виконуваного коду, оскільки AST будується на основі синтаксичного аналізу вихідного коду. Представлення коду програми у вигляді AST забезпечує частково незалежну від синтаксису репрезентацію, що дозволяє виконати певні синтаксичні спрощення. Це полегшує подальший аналіз AST, оскільки основна увага переміщується на

семантичне значення, а не на синтаксичну коректність. Створений AST містить ту саму семантичну інформацію, що й базовий вихідний код. Крім того, точна синтаксична репрезентація програми зазвичай не становить інтересу для розробника; вона слугує лише засобом передачі інформації, подібно до будь-якої природної мови. Відтак, основну увагу буде зосереджено на аналізі представлення програми у вигляді AST, уникаючи необхідності обробки різних синтаксичних репрезентацій з ідентичною семантикою.

Оскільки процес побудови та аналізу AST значною мірою залежить від конкретних мов програмування та парсерів, а ExplorViz сфокусований на Java, подальший розгляд обмежено проблематикою та рішеннями для мови програмування Java. Незважаючи на те, що певні концепції можуть бути застосовані до схожих мов програмування, існують специфічні для кожної мови особливості та проблеми.

Існує низка Java-проектів для генерації AST з коду Java. У даній роботі обрано проект JavaParser, оскільки він надає можливість використання JavaSymbolSolver для визначення типів на подальших етапах. Причина необхідності JavaSymbolSolver буде пояснена пізніше.

2.1.1 Структуровані дані

Після розгляду причин та переваг аналізу вихідного коду та побудови AST, можна перейти до визначення типів даних, які підлягають збору. Оскільки зібрані дані детермінують інформативність подальшої візуалізації, необхідно ретельно обирати дані для зберігання та ігнорування. Найпростішим підходом було б збереження всіх даних, отриманих зі створеного AST, оскільки вони відображають повний вміст Java-файлу. Проте, цей підхід суперечить основній меті служби аналізу – попередньому структуруванню та відбору даних для подальшої візуалізації.

Дані файлу

Репрезентація залежностей класів є можливою лише за умови, що всі використані класи перелічені та доступні для візуалізації. Мова програмування Java надає цю можливість, вимагаючи явного зазначення всіх використаних класів, статичних методів та переліків (enumerations) у файлі як імпортів. Отже, дані файлу включають список імпортів, пакет, до якого належить файл, та список усіх класів, що містяться в цьому файлі.

Дані класу

Основний вміст файлів Java становлять оголошення класів. Слід пам'ятати, що в Java існує чотири типи структур, подібних до класів: класи, абстрактні класи, інтерфейси та переліки (enumerations). Їх можна обробляти практично однаково, оскільки кожна з цих структур містить тіло, яке потенційно може включати методи та поля (у переліках поля називаються константами переліків). Класи можуть мати модифікатори доступу, такі як private, public або protected. Модифікатори мають бути включені у візуалізацію, оскільки модифікатори private та protected зазвичай позначають класи, які слід розглядати лише при аналізі внутрішньої реалізації. Кожен клас може бути ідентифікований за його повністю кваліфікованим ім'ям (Fully Qualified Name, FQN), яке формується шляхом об'єднання імені пакета та імені класу. Визначення FQN є простим для класів верхнього рівня. Проте, вкладені класи вимагають ієрархічного FQN, що включає FQN батьківського класу, об'єднаний з ім'ям вкладеного класу. Збереження FQN дозволяє візуалізації асоціювати кожен тип з відповідним класом. Дані класу включають список методів, полів, модифікаторів доступу, внутрішніх класів, реалізованих інтерфейсів та надкласу (super class).

Дані методу

Основна логіка програми та, відповідно, її складність, зосереджені всередині методів. Для розуміння завдання, яке виконує метод, його ім'я та сигнатура є ключовими джерелами інформації. Сигнатура включає типи

параметрів, які будуть детальніше розглянуті у підрозділі 4.1.2. Ім'я методу має бути достатньо описовим, аби вказувати на його призначення. Конструктори є специфічним типом методів, тому їх можна обробляти аналогічним чином. Окрім імені та параметрів, дані методу включають тип повернення та модифікатори. Ці записи, проте, здебільшого слугують контейнерами для інтеграції обчислених метрик. Ці метрики можуть узагальнювати певні характеристики внутрішньої реалізації методу.

Модель даних

Модель даних, представлену на рисунку 2.1, спроектовано на основі вищезазначених потреб. Модель було розширено за рахунок включення FieldData та ParameterData.

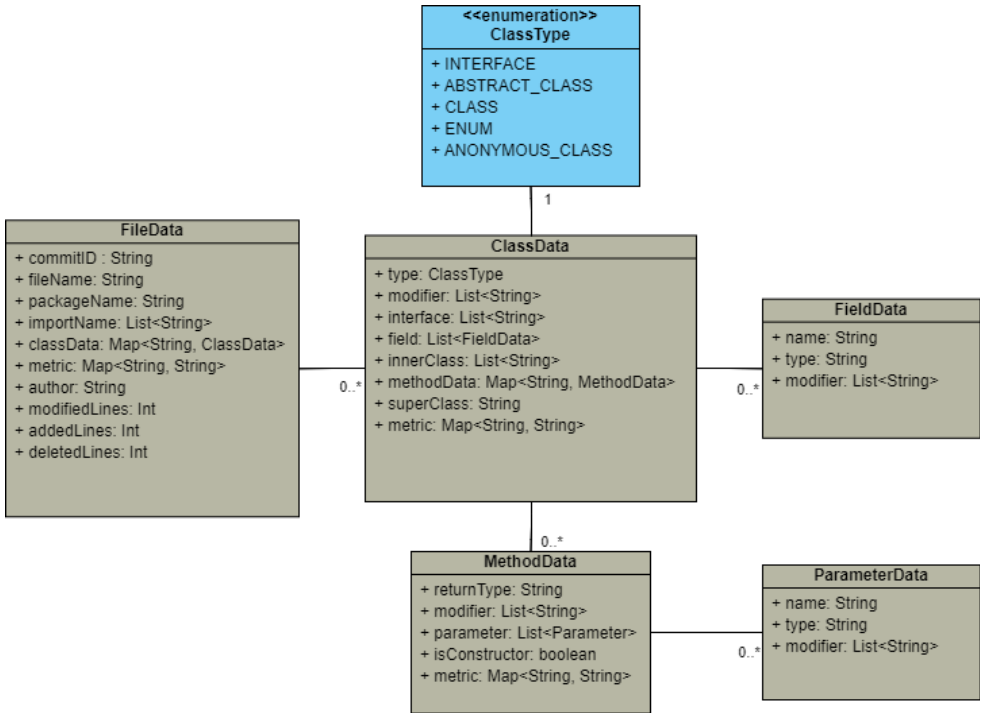


Рисунок 2.1 - Модель даних з полями метрик

Це зумовлено тим, що ці записи містять інформацію, яка визначає їх самих, а не атрибути батьківського класу чи методу, що вимагає їх виділення з ClassData та MethodData. Модель спроектована як 'плоска' (flat) на рівні класів. Відтак, усі класи зберігаються на одному рівні представлення. Це

забезпечує легке перелічення та обхід усіх доступних класів, незалежно від їхньої глибини ієрархії у вихідному коді. Точне ієрархічне положення представлено їхнім FQN.

2.1.2. Визначення типів

Як було зазначено вище, визначення типів є важливою складовою процесу збору даних. Інформація про наявність у методу параметрів з довільним ім'ям недостатня для розуміння програмістом завдання, яке виконує спостережувана частина програми. Програміст потребує інформації про використаний тип даних. Типи даних у Java можна класифікувати на п'ять різних категорій за складністю їх визначення: примітивні типи, вбудовані типи, пакетні типи, проектні типи та зовнішні типи.

Примітивні

Примітивні типи є найбільш базовими в мові Java. Вони інтегровані в мову і не є об'єктами. Примітивні типи є найбільш фундаментальними змінними типами, знайомими кожному розробнику Java. Відтак, їхня додаткова обробка для цілей розуміння не є необхідною. До примітивних типів належать boolean, byte, short, int, long, char, float та double. Окрім цих восьми примітивних типів, структуру даних масив, що позначається парою квадратних дужок після імені типу, може бути поєднано з будь-яким типом даних. Будь-який розробник Java здатний розпізнавати масиви та примітивні типи; відтак, їхня репрезентація не потребує модифікації.

Вбудовані

Вбудовані типи даних часто використовуються у багатьох проектах і є невід'ємною частиною мови Java. Ці типи визначені у пакеті java.lang і імпортуються неявно при використанні. Відтак, явний оператор імпорту не потрібен. До добре відомих вбудованих типів належать String, Class, Object, об'єктні оболонки примітивних типів (наприклад, Boolean, Integer), інтерфейси (Throwable, Iterable, Comparable) та анотації (Deprecated,

Override). Пакет java.lang також містить визначення помилок (StackOverflowError) та винятків (NullPointerException). Зазначені типи є лише частиною широко використовуваних або відомих типів, що містяться в пакеті. З метою уникнення неоднозначності для програміста, до вбудованих типів додається ім'я пакета. Таким чином, тип ThreadGroup представлено його повністю кваліфікованим ім'ям java.lang.ThreadGroup.

Типи в межах пакета

Однією з ключових концепцій дизайну мови Java є концепція пакета. У межах одного пакета класи з пакетним доступом можуть бути доступні класами, методами або полями з інших файлів цього ж пакета без необхідності явного імпорту. У файлі Java можуть міститися типи, які не належать до примітивних чи вбудованих, але при цьому не імпортовані явно. Для визначення таких типів необхідно здійснити аналіз кожного файлу в поточному пакеті за відсутності іншої інформації про місцезнаходження визначення типу. Якщо визначення типів для двох попередніх категорій може бути здійснено за допомогою простого пошуку, то визначення типів цієї категорії вимагає складнішого підходу. Відтак, для визначення таких типів буде використано JavaSymbolSolver, як згадувалося вище, разом із його інтегрованим компонентом JavaParserSymbolSolver. Розв'язувач типів надає FQN типу, що дозволяє ідентифікувати місцезнаходження його визначення.

Проектні типи

Проектні типи визначені в межах поточного проекту, але не в тому самому пакеті, де вони використовуються. Ці типи визначені у доступному вихідному коді проекту, що дозволяє зібрати відповідні дані. Оскільки компілятор Java також потребує ідентифікації цих типів, їх повністю кваліфіковані імена (FQN) зазначаються у списку імпортів. При необхідності отримання додаткових даних про ці типи, можна знайти відповідний вихідний код за шляхом, вказаним у FQN. Іноді типи можуть бути

відносними до імпортів. Проте FQN легко визначити, оскільки останній ідентифікатор в операторі імпорту відповідає першому ідентифікатору типу.

Зовнішні типи

Ці типи є найбільш складними для однозначного визначення. Зі зростанням розміру проекту та його залежностей від бібліотек використовуються зовнішні бібліотеки у форматі JAR-файлів. Ці JAR-файли зазвичай не зберігаються в репозиторії проекту, а завантажуються інструментом збірки за необхідності. Деякі проекти залежать від конкретних версій бібліотек, тоді як інші використовують лише останню доступну версію. Це може призводити до конфліктів або помилок у коді при використанні останньої версії бібліотеки разом із застарілим станом проекту. Оскільки виконується лише статичний аналіз і не здійснюється збірка проекту, це не обов'язково призведе до помилок аналізу, проте може спричинити наявність нерозв'язаних типів. Крім того, це потребуватиме отримання кожної використовуваної бібліотеки. Це може суттєво збільшити навантаження на систему через значну кількість використовуваних бібліотек та обсяг комітів, що підлягають аналізу. Ігнорування включення зовнішніх бібліотек у процес визначення типів (без необхідності аналізу самого JAR-файлу) не повинно суттєво обмежувати зрозумілість. Це пояснюється тим, що розробник, ймовірно, більше зацікавлений у коді аналізованого проекту, аніж у коді бібліотек. Отже, надається FQN для типу без детального аналізу його визначення. Оскільки зовнішні типи бібліотек зазвичай зазначаються в імпортах файлу, для ідентифікації достатньо зіставити імена імпорту зі знайденим ім'ям типу. Цей підхід має вищезазначені недоліки, пов'язані з невизначеними типами, проте являє собою компроміс між продуктивністю, простотою реалізації та повнотою даних.

Хоча майже всі типи повинні бути вирішені зараз, ми повинні визначити, що станеться, якщо визначення не вдається. Ми не можемо просто пропустити тип, оскільки це призведе до відсутності даних у даних

аналізу. Оскільки у нас немає резервного варіанту для цього випадку, щоб отримати визначений тип, ми можемо додати лише ім'я типу з вихідного коду, який ми знайшли, але не змогли визначити, і додати це ім'я типу як запис даних.

Подібні проблеми виникають, якщо використовуються імпорти з підстановкою. Ми могли б припустити, що якщо присутній один імпорт з підстановкою, то тип, про який йде мова, є частиною пакета, імпортованого за допомогою підстановки, якщо всі інші спроби визначення типів не вдалися, і тип-розв'язувач працював правильно. Тоді тип, про який йде мова, повинен бути визначений в межах пакета, імпортованого за допомогою підстановки. Це завжди не вдається, якщо присутні більше одного імпорту з підстановкою.

2.2. Опис метрик

Хоча базові структурні дані та визначені типи, описані вище, є важливими для загального розуміння взаємодії компонентів проекту, часто виникає необхідність отримати уявлення про "запах" коду (code smells). Це може бути корисно для оцінки підтримуваності окремих сегментів проекту або для оцінки ризику виникнення помилок, пов'язаного з високою складністю. Далі буде розглянуто деякі метрики, які підлягають реалізації. Певні метрики є більш інформативними для оцінки складності, ніж інші, проте всі вони надають додаткову інформацію до даних аналізу.

2.2.1. Кількість рядків коду

Як випливає з назви, ця метрика обчислюється на основі кількості рядків вихідного коду. Враховуються лише рядки власне коду; рядкові, блокові коментарі та коментарі JavaDoc ігноруються. Таким чином, загальна кількість рядків коду завжди менша або дорівнює загальній кількості рядків у

вихідному файлі. Кількість рядків коду обчислюється для файлу в цілому, включаючи класи та методи. Це дозволяє оцінити потенційну схильність файлу або класу до помилок. Хоча відсутні чіткі порогові значення для розмежування прийнятних та надмірно великих класів і файлів, ця метрика може надати загальне уявлення.

Файл, що містить 10 000 рядків коду, об'єктивно вважається менш бажаним порівняно з файлом обсягом 100 рядків. З іншого боку, файл обсягом 1000 рядків, рівномірно розподілених між кількома методами, може виглядати значним, проте потенційно є кращим за файл обсягом 500 рядків, що повністю містяться в одному методі.

2.2.2. Глибина вкладеності (Глибина вкладених блоків)

Базові елементи програми не обмежуються простими операторами; значна частина кодової бази складається з керуючих структур (наприклад, if, for, while). Якщо складність фрагмента коду, що містить лише прості оператори, може бути оцінена їх кількістю, то складність коду з керуючими структурами може бути менш очевидною навіть при однаковому обсязі в рядках. Метрика глибини вкладених блоків визначає максимальну глибину вкладеності керуючих структур і повертає це значення. Ця метрика покликана відобразити зростаюче когнітивне навантаження на розробника, пов'язане з необхідністю розуміння контексту виконання. Кожна керуюча структура створює новий контекст виконання для наступних операторів.

Розробник має враховувати всі застосовні умови для поточного контексту, що прямо пропорційно збільшує когнітивне навантаження зі зростанням глибини вкладеності. Високе значення глибини вкладених блоків може свідчити про надмірну складність методу, який потенційно може бути рефакторизований шляхом винесення частини коду в окремі функції програмного забезпечення.

2.2.3. Цикломатична складність

Метрика цикломатичної складності є ще одним засобом оцінки складності керуючих структур. На відміну від метрики глибини вкладеності, метрика цикломатичної складності підсумовує всі керуючі структури. Таким чином, складність зростає з кожною використаною керуючою структурою. Вкладені структури враховуються так само, як і структури верхнього рівня. Це робить дану метрику більш інформативною при використанні у поєднанні з метрикою глибини вкладеності. Комбінація низького значення глибини вкладеності та високого значення цикломатичної складності, ймовірно, свідчить про легшу зрозумілість коду, аніж зворотна ситуація. Ще одним відомим аспектом обчислення цієї метрики є її поведінка для конструкцій switch-case. Кожен вираз у блоці switch збільшує загальну складність на одиницю. Проблема проілюстрована у лістингу 2.1, де представлено приклад простого блоку switch-case. Результуюче значення метрики для даного прикладу становить 14, що свідчить про вищу складність порівняно з подвійно вкладеним міченим циклом for, для якого метрика дорівнює 5 (див. лістинг 2.2).

Лістинг 2.1. Значення складності 14

```
switch (month) {
    case 0: return "January";
    case 1: return "February";
    case 2: return "March";
    case 3: return "April";
    case 4: return "May";
    case 5: return "June";
    case 6: return "July";
    case 7: return "August";
    case 8: return "September";
    case 9: return "October";
    case 10: return "November";
    case 11: return "December";
    default: return "NONE";
}
```

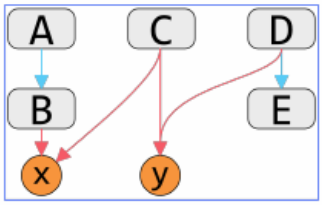
Лістинг 2.2. Значення складності 5

```
int sum = 0;
OUTER: for (int i = 0; i < limit; ++i) {
    if (i <= 2) {
        continue;
    }
    for (int j = 2; j < i; ++j) {
        if (i % j == 0) {
            continue OUTER;
        }
    }
    return sum;
}
```

2.2.4. Відсутність зчепленості в методах

Остання метрика, яку ми хочемо розглянути, — це метрика відсутності зчепленості в методах, точніше її варіант номер 4, скорочено LCOM4. Результати на основі цієї метрики вказують на те, чи можна розділити клас на кілька класів через відсутність зчепленості.

```
1 class A {
2     private int x;
3     private int y;
4     void methodA(){ methodB(); }
5     void methodB(){ x++; }
6     void methodC(){ y++; x++; }
7     void methodD(){ y++; methodE();}
8     void methodE(){}
9 }
```




```
1 class B {
2     private int x;
3     private int y;
4     void methodA(){ methodB(); }
5     void methodB(){ x++; }
6     void methodC(){ y++; }
7     void methodD(){ y++; methodE();}
8     void methodE(){}
9 }
```

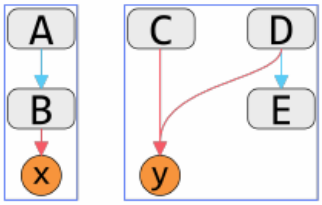


Рисунок 2.2 - Два схожих класи з їхніми відповідними LCOM-графами.

Лівий клас має LCOM=1 і вважається добре спроектованим класом. Правий має LCOM=2 і може бути розділений на два класи. Один новий клас

містить methodA, methodB та x, інший клас містить methodC, methodD, methodE та y.

У контексті метрики LCOM4 два методи пов'язані, якщо один з методів викликає інший або обидва мають доступ до одного і того ж поля. Ми аналізуємо всі методи в межах класу та шукаємо незв'язані методи. Хороший клас повинен давати результат LCOM4 значення 1, що означає, що всі містяні методи пов'язані. Значення 0 трапляється лише в тому випадку, якщо клас не містить жодних методів. Це може бути розглянуто як поганий клас, але це залежить від завдання класу. Ми повернемося до цього пізніше в третьому розділі, оскільки поведінка також залежить від реалізації. Крім того, класи, які можна розділити, дають результат LCOM4 значення 2 і вище, що означає, що ці класи можна винести в окремі класи, оскільки вони не є зчепленими. На рисунку 2.2 показано два майже ідентичних приклади реалізації класів.

2.3. Керування репозиторіями Git

Тепер, коли ми з'ясували, як аналізувати один стан проекту Java, включаючи всі його файли, ми повинні подумати про обробку знімків проектів Java. Це проекти Java в стані, який відповідає коміту Git. Аналіз кожного файлу для кожного коміту необхідний для спостереження за змінами протягом періоду розробки. Дані повинні бути структуровані так, щоб їх можна було легко та швидко отримати для наступної візуалізації.

2.3.1. Обробка змін

Спочатку ми повинні визначити, що таке зміна в нашому контексті. Ми можемо визначити зміну як будь-яку модифікацію файлу, яка змінила його вміст порівняно з попереднім комітом. Якщо файл не змінився, його не потрібно аналізувати знову. Оскільки це визначення в цілому збігається з

принципом, який використовує Git, ми повинні обробляти будь-який файл, який Git позначає як змінений.

Оскільки наш аналіз найменшого блоку даних — це метод, ми могли б перевірити, чи змінився метод. Якщо зміна виявлена, потрібно зберегти лише фрагмент даних проаналізованого методу. Це здається хорошою ідеєю в теорії, але має значні недоліки. Деякі метрики, згадані вище, а саме кількість рядків коду та LCOM4, обчислюються для всього класу або файлу. Оновлення лише даних методу призведе до неправильних значень для цих метрик і будь-яких метрик, які можуть бути додані в майбутньому і мають той самий клас або файловий масштаб. Отже, ми повинні оновити ці метрики також.

Крім того, ми повинні були б надсилати блок оновлення аналізу даних, який містить лише оновлення до останнього повного аналізу даних. Це призвело б до збільшення обчислювальних зусиль. Уявіть собі файл, який існує протягом 1000 комітів, змінений кожним комітом лише в одному методі. Це потребувало б від візуалізації накопичувати всі 999 оновлення та додати їх до початкового повного аналізу, щоб отримати його поточний стан. Хоча це було б хорошою ідеєю з точки зору розміру файлу, збільшення складності агрегації даних, виконуваної візуалізацією, скасовує перевагу. Крім того, аналіз повинен бути виконаний на всьому файлі через згадані вище метрики файлу, що робить час виконання майже ідентичним із невеликим зменшенням місця для зберігання, але збільшенням складності агрегації. Рішення полягає в тому, щоб надсилати весь пакет даних аналізу, якщо файл був змінений під час коміту. Результуючий пакет даних міститиме всі дійсні дані для проаналізованого коміту. Отже, візуалізація може бути простішою, оскільки найновіший пакет аналізу міститиме всі дані для файлу. Крім того, якщо файл не має змін, ми можемо ігнорувати його, оскільки старий стан все ще дійсний.

Що стосується змін, Git має різні типи змін для файлів. Вміст файлу може бути змінений, файл може бути перейменований, видалений або скопійований. Ми могли б зберегти ці різні типи змін, але ми цікавимося даними всередині файлу. Якщо файл більше не використовується в проекті, нас не цікавить те, що він видалений. Якщо файл перейменовано, нас не цікавить старе ім'я, оскільки нове ім'я має значення і використовується для аналізу. Те саме стосується і копіювання. Це спрощення разом із вищезгаданою стратегією повного файлу створює одну проблему: ми не можемо дізнатися, які файли є частиною коміту, оскільки ми не знаємо, чи файл було видалено, чи просто не змінено в цьому коміті. Щоб вирішити цю проблему, ми вводимо звіт про коміт, який містить список усіх файлів, які належать до поточного коміту. Накопичення будь-якого стану так же просте, як отримання його відповідного звіту про коміт і пошук усіх найновіших пакетів даних аналізу для кожного файлу, переліченого в списку.

2.3.2. Обробка гілок

Оскільки Git підтримує гілки, аналіз також повинен мати спосіб підтримки цієї функції. Можливо, користувач хоче аналізувати гілки main і develop одночасно для їх порівняння. На практиці гілки Git — це просто впорядковані списки з'єднаних комітів Git, отже, обробка гілок майже тривіальна. На рисунку 2.3 показано базову структуру репозиторію комітів, що містить дві гілки; значення вузлів міток символізують значення ідентифікаторів хешів комітів. Як ми бачимо, гілки ділять коміти 1 і 2, а дані аналізу однакові. Через обмеження проекту ми надішлемо ідентичні пакети даних аналізу двічі для комітів 1 і 2, один раз для гілки main і один раз для гілки develop. Це можна вирішити за допомогою пошуку при вставці даних до бази даних, оскільки комбінація ім'я файлу та ідентифікатора коміту є дійсним унікальним ключем і повинна бути вставлена лише один раз. Ми генеруємо додаткові дані аналізу для комітів з 3 по 6 лише один раз.

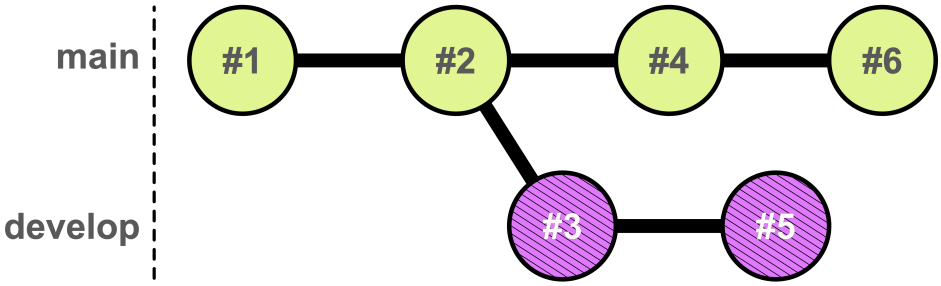


Рисунок 2.3 - Граф комітів репозиторію, що містить дві гілки: гілка develop відгалузилася від гілки main на коміті 2. Обидві гілки містять коміти 1 та 2.

2.3.3. Метрики Git

Дані аналізу вихідного коду та відповідні метрики є цінними, проте мають статичний характер. Однією з ключових переваг аналізу репозиторію проекту є можливість відстеження його еволюції з плином часу. Доступ до первинних даних Git дозволяє отримати інформацію про тип та частоту модифікацій файлів.

Підхід, що передбачає передачу лише даних аналізу при змінах, дозволив би візуалізації обчислювати частоту модифікацій, але не їхню кількість. Натомість, доступ до первинних даних Git дозволяє отримати більш детальну інформацію. Включення кількості змінених, доданих та видалених рядків у дані аналізу дозволяє здійснити кількісну оцінку коміту. Значна кількість змін у файлі протягом процесу розробки може слугувати індикатором потенційного "файлу-бога" (God Object), що має високу залежність від інших компонентів програмного забезпечення. Крім того, можливість обчислення дельти змін коду (code churn) на етапі візуалізації дозволяє отримати відповідну метрику [7].

Шляхом асоціювання імені автора з кожним об'єктом даних аналізу можна ідентифікувати файли, які часто змінювалися багатьма розробниками. Ці дані можуть бути отримані шляхом ідентифікації всіх версій файлу з однаковим ім'ям та формування множини авторів, які вносили зміни. Якщо файл змінюється значною кількістю різних авторів, це може свідчити про його загальне призначення (наприклад, утилітарні класи) або про нечітко

визначені зони відповідальності. Такі нечітко визначені робочі процеси важко виявити, використовуючи лише стандартний статичний аналіз коду. Проте, ця метрика є більш релевантною для аспектів управління людськими ресурсами або управління проектами, важливість яких зростає зі збільшенням розміру програмного проекту та його кодової бази.

2.4. Проектування служби аналізу коду

На рисунку 2.4 представлено архітектуру служби аналізу. Вона складається з Code Service, що функціонує як пункт призначення даних в екосистемі іншого проекту, та Code Agent, який відповідає за виконання аналізу вихідного коду та обробку репозиторію Git. Згідно з архітектурою ExplorViz, представленою на рисунку 1.3, Code Agent виконує роль Колектора Структури (Structure Collector) в середовищі моніторингу, забезпечуючи дані аналізу. Code Service виконує функції, подібні до Adapter Service, діючи як шлюз для передачі даних до бази даних.

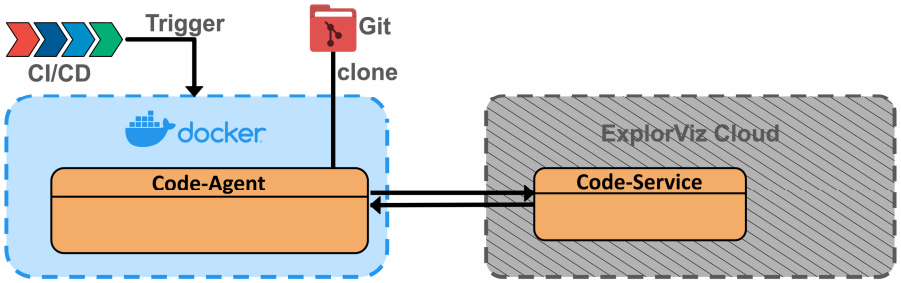


Рисунок 2.4 - Огляд проектування служби. Хмарне середовище ExplorViz використано як приклад і може бути замінене будь-яким іншим проектом

При реалізації Code Agent як безстанового компонента необхідно оцінити компроміс між перевагами такого підходу та потенційним збільшенням мережевого трафіку і складності, пов'язаними з цим типом проектування [4]. Недолік буде безумовно прийнятний, якщо повний стан системи необхідний постійно. Проте, оскільки повний стан необхідний лише

для накопичення метрик Git (що є завданням з невисокими вимогами до продуктивності), обрано більш самодостатній дизайн, який не вимагає зовнішнього управління станом. Таким чином, під час аналізу вдається утримувати навантаження на середовище функціонування Code Service на низькому рівні, оскільки його основним завданням є лише обробка та збереження вхідних даних.

Відтак, основний акцент зроблено на легкому, майже однонаправленому безстановому підході. В рамках цього підходу запит поточного віддаленого стану здійснюється один раз за виконання аналізу, незалежно від кількості комітів, що підлягають аналізу, або обсягу даних, вже збережених у базі даних. Це спрощує реалізацію Code Agent та Code Service і переносить обчислення метрик, що агрегують дані з кількох комітів, на етап візуалізації, яка має бути реалізована у майбутньому. Оскільки модуль візуалізації все одно потребуватиме агрегації необхідних даних, обчислювальні накладні витрати для цих метрик вважаються мінімальними.

Безстанова система легше інтегрується в конвеєр CI. Docker-контейнер може бути використаний як виконавче середовище. Він є повністю самодостатнім, не вимагає зовнішніх даних для ініціалізації і не зберігає стану між запусками. Процес оновлення спрощується завдяки капсуляції, що полегшує міграцію між конвеєрами або оновлення самого Code Agent.

Як було зазначено вище, обчислення метрик, що вимагають даних з попередніх комітів, обмежене через відсутність доступу до історичних даних безпосередньо у Code Agent. Якщо подальша експлуатація продемонструє, що перенесення відповідальності за обчислення агрегованих метрик на модуль візуалізації призводить до проблем на наступних етапах, розширення поточної архітектури не буде тривіальним. Це потребуватиме розширення функціональності Code Agent та Code Service, а також модифікації всього циклу аналізу.

З метою підготовки служби аналізу до інтеграції в конвеєр неперервної інтеграції, необхідно представити її як самостійний та легко вбудовуваний програмний пакет. Незважаючи на існування декількох варіантів реалізації, вирішення цього питання було досягнуто завдяки готовій підтримці попередньо зібраних Docker-контейнерів у Quarkus та GitLab CI. Параметри конфігурації можуть бути задані через змінні середовища. Можливість використання попередньо зібраного контейнера. Такий підхід полегшує інтеграцію даної служби як в існуючі, так і в нові проекти. Незважаючи на те, що проект та його попередньо зібраний контейнер спроектовані спеціально для середовища GitLab CI, інтеграція з іншими службами CI також є можливою, що буде продемонстровано у наступному розділі.

Висновки до розділу

У другому розділі розглянуто методологічні та практичні засади статистичного аналізу вихідного коду Java, що є ключовим етапом для подальшої візуалізації структурних і поведінкових характеристик програмного забезпечення. Показано, що ефективне опрацювання коду передбачає його попередню структуризацію та типізацію, що дозволяє отримати уніфіковане подання для подальшого аналізу.

Розкрито зміст основних метрик, що використовуються для кількісного оцінювання якості коду: кількість рядків, глибина вкладеності, цикломатична складність та показники зчепленості. Ці метрики дозволяють виявляти фрагменти коду з підвищеним рівнем складності та потенційною потребою в рефакторингу, що сприяє підвищенню підтримуваності та зрозумілості програмної системи.

Окрему увагу приділено обробці історії змін у системах контролю версій, зокрема Git. Описано методи фіксації змін, аналізу гілок та побудови

спеціалізованих метрик, що дозволяють оцінювати динаміку розвитку коду, активність розробників та складність процесу підтримки.

У результаті проведено проектування служби аналізу коду, яка поєднує функціональність збору, обробки та інтерпретації метрик із можливістю інтеграції у процес безперервної розробки. Це створює основу для побудови інструментів візуалізації, здатних представляти складну технічну інформацію в інтуїтивно зрозумілому вигляді.

РОЗДІЛ 3. ІМПЛЕМЕНТАЦІЯ МЕТОДОЛОГІЇ ГРАФІЧНОГО СУПРОВОДУ ПРОЦЕСУ РОЗРОБКИ ПЗ НА ОСНОВІ КЛАСИФІКОВАНИХ РЕСУРСІВ GIT

Після розгляду принципів проектування, подальша увага буде зосереджена на реалізації проекту. Буде розглянуто процеси клонування репозиторіїв, перемикання між гілками та обходу комітів. Буде описано методи ідентифікації змін у файлах Java та застосування файлових фільтрів. Далі буде розглянуто взаємодію з AST (Абстрактним Синтаксичним Деревом) за допомогою JavaParser та процес збору необхідних структурних даних з коректним визначенням типів. Буде висвітлено комунікацію між Code Agent та Code Service, а також збір та обчислення різних метрик. Нарешті, буде описано процес побудови служби у вигляді Docker-контейнера та принципи його конфігурації.

3.1. Структура проекту

Як обговорювалося в другому розділі, проект реалізовано у вигляді служби, що може бути інтегрована в нові або існуючі проекти. Незважаючи на те, що аналіз призначений для використання з ExplorViz, він є повністю функціональною самостійною службою. Оскільки комунікація між Code Service та Code Agent здійснюється за допомогою gRPC, сторонній пункт призначення, що реалізує власний gRPC-сервер, може бути використаний для інтеграції в будь-який інший проект з метою використання наданих аналітичних функцій.

3.1.1. Реалізація Code Service

Code Service виконує роль простого шлюзу для даних аналізу. Він призначений для інтеграції в ExplorViz і відповідає за обробку та збереження

вхідних пакетів даних. Code Service являє собою простий пункт призначення даних, що не містить складної бізнес-логіки. Він надає gRPC-інтерфейс та здійснює вивід даних на стандартний потік виведення. Потенційна інтеграція Code Service в ExplorViz проілюстрована на рисунку 3.1. Цю функціональність можна легко адаптувати для інтеграції в інші проекти, оскільки Code Service виступає лише як шлюз для вхідних даних.

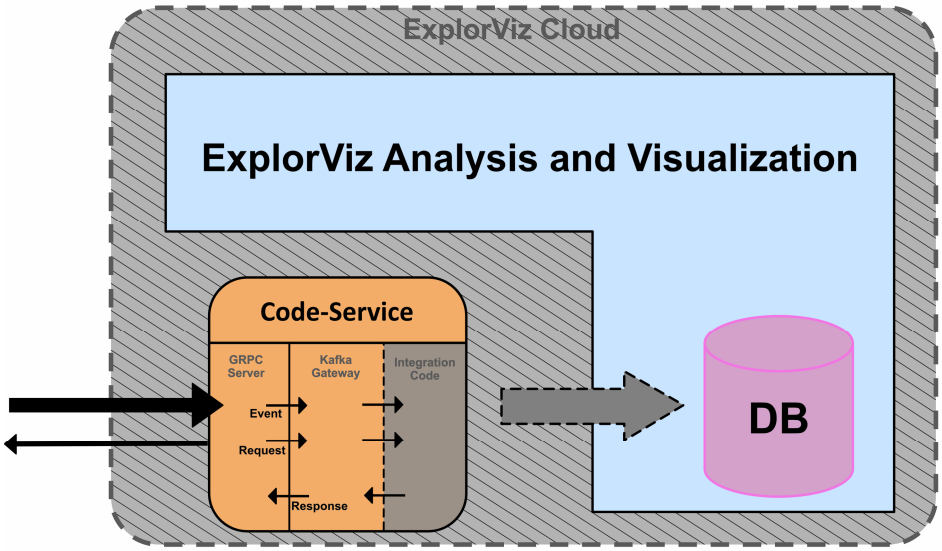


Рисунок 3.1 - Запропонована інтеграція Code Service в ExplorViz. Завдяки абстрактному дизайну, інтеграція в інші проекти є можливою аналогічним чином

Для інтеграції Code Service в ExplorViz або будь-яку іншу програмну систему, необхідно реалізувати відповідні методи в класі KafkaGateway. Метод processCommitReport призначений для обробки вхідних звітів про коміти. Кожен звіт надсилається після завершення аналізу відповідного коміту. processFileData викликається для кожного вхідного пакета FileData, що містить усі дані аналізу для окремого файлу. Нарешті, processStateData викликається у випадку запиту Code Agent'ом поточного стану даних аналізу, що містяться в базі даних. Детальні пояснення щодо реалізації представлені у коментованому оригінальному файлі в репозиторії Code Service.

Лістинг 3.1. Спрощений код KafkaGateway

```
public class KafkaGateway {
    public void processCommitReport(final CommitReportData commitReportData) {}
    public void processFileData(final FileData fileData) {}
    public String processStateData(final StateDataRequest stateDataRequest) {
        return "";
    }
}
```

3.1.2. Реалізація Code-Agent

Code-Agent — це виконавець аналізу. На відміну від код-сервісу, який можна реалізувати інакше як налаштований кінцевий пункт gRPC, код-агент надається та використовується як готовий до запуску контейнер Docker. Уся функціональність щодо статичного аналізу коду та керування репозиторієм реалізована в ньому, тому наступні розділи зосереджені саме на цій роботі.

3.2. Аналіз та взаємодія з репозиторієм Git

Щоб мати можливість аналізувати вміст репозиторіїв Git, ми повинні взаємодіяти з репозиторіями спочатку. Код-агент використовує JGit для керування репозиторієм, оскільки він надає абстракцію для легкого програмного взаємодії, схожого на Git CLI, і багатофункціональний Java API для точного керування репозиторієм. Хоча основна сфера застосування служби — це інтеграція в конвеєри CI, і тому вона сама клонує репозиторії, також повинна бути можливість запустити аналіз з локального репозиторію, наданого користувачем. Після завантаження репозиторію наступним кроком є вибір гілки, яку потрібно проаналізувати.

3.2.1. Доступ до репозиторію

Як показано в лістингу 3.2, процес вирішення, чи використовується локальний репозиторій, є досить простим. Змінна localRepositoryPath

заповнюється змінною середовища (див. Додаток) або залишається порожньою, якщо користувач не визначає змінну. Отже, наявність цього значення вирішує, чи клонувати репозиторії. З іншого боку, remoteRepositoryObject містить усі значення, необхідні для клонування репозиторію. Потім, за допомогою надання URL і назви гілки аналіз переходить у метод downloadGitRepository та починає клонування.

Лістинг 3.2. Рішення щодо клонування чи відкриття існуючого репозиторію

```
public Repository getGitRepository(final String localRepositoryPath,
    final RemoteRepositoryObject remoteRepositoryObject)
    throws IOException, GitAPIException {
    if (localRepositoryPath.isBlank()) {
        return this.downloadGitRepository(remoteRepositoryObject);
    } else {
        return this.openGitRepository(localRepositoryPath, remoteRepositoryObject.
            getBranchName());
    }
}
```

Лістинг 3.3. Клонування репозиторію за допомогою JGit. Спрощена версія методу downloadGitRepository

```
final Map.Entry<Boolean, String> checkedRepositoryUrl = convertSshToHttps(
    remoteRepositoryObject.getUrl());
String repoPath = remoteRepositoryObject.getStoragePath();
this.git = Git.cloneRepository()
    .setURI(checkedRepositoryUrl.getValue())
    .setCredentialsProvider(remoteRepositoryObject.getCredentialsProvider())
    .setDirectory(new File(repoPath))
    .setBranch(remoteRepositoryObject.getBranchNameOrNull())
    .call();
return this.git.getRepository();
```

Хоча клонування репозиторію в основному виконується JGit (лістинг 3.3), важливо виконати перевірку помилок і очищення вхідних даних користувача перед передачею даних до JGit. Невелика частина цього можна

побачити в виклику convertSshToHttps для забезпечення того, що користувач надав дійсний HTTP URL до репозиторію Git.

Цей код (лістинг 3.3) на Java демонструє процес клонування репозиторію за допомогою бібліотеки JGit. Він перевіряє URL репозиторію, перетворюючи SSH-URL на HTTPS, якщо це необхідно, і використовує ці дані для клонування репозиторію в задану директорію. Також можна вказати постачальника автентифікації для доступу до приватних репозиторіїв. Після виконання команди клонування, репозиторій стає доступним для подальшої роботи.

Крім цього, оскільки ми виконуємо мережеву комунікацію, ми також повинні враховувати її стандартні помилки. Як показано в згаданому листингу, також можливо надати постачальника автентифікації, що відкриває можливість клонувати приватні репозиторії. Після команди JGit ми маємо доступ до повністю клонованого репозиторію, перемикаємося на задану гілку, готові до роботи.

3.2.2. Проходження комітів

Хоча у нас є доступ до репозиторію для подальшої обробки, ми повинні отримати доступ до кожного коміту гілки та обробляти їх у порядку від найдавнішого до найновішого. Зручно, що JGit надає об'єкт RevWalk для проходження дерева комітів репозиторію. Оскільки ми цікавимося лише однією гілкою, ми обмежуємо її включенням комітів з гілки, яка нас цікавить. Потім ми повинні відсортувати RevWalk за часом коміту в порядку зростання. Тепер у нас є ітератор для проходження всіх комітів, включених до гілки, в порядку, який нам потрібен. Поки ми проходимо коміти, ми повинні пам'ятати, що у нас немає доступу до стану репозиторію на коміті, на який ми показуємо. Ми можемо взаємодіяти з файлами в потрібному стані лише після перемикання на конкретний коміт. Це створює значне навантаження на сховище, оскільки ми повинні відновити стан кожного

коміту в файловій системі. Щоб мінімізувати доступ до сховища найкращим чином, має сенс перевірити зміни перед перемиканням. Доступ до даних для аналізу потрібен лише в тому випадку, якщо моніторингові файли змінюються.

3.2.3. Виявлення змін

Хоча ми можемо отримати доступ до змін, створених комітом, ми не можемо запустити аналіз лише на основі цих даних. Однак ми можемо виявити, які коміти містять зміни в файлах, які нас цікавлять. Отже, можливо попередньо вибрати коміти, для яких потрібне перемикання. Крім того, ми можемо виявити файли, які містять зміни. Хоча краще було б розрізняти тип зміни, наприклад, чи це коментар або рядок коду, ми вирішили обробляти всі зміни однаково через простоту. Попередній аналіз був би потрібен для виявлення різниці, що в кінцевому рахунку призвів би до створення складного проекту самого по собі.

Для надання списку змінених файлів для аналізу реалізовано метод listDiff. Спрощена версія для детальнішого огляду показана в лістингу 3.4.

Лістинг 3.4. Метод listDiff, що генерує список файлових дескрипторів

```
public List<FileDescriptor> listDiff(final Repository repository,
    final RevCommit oldCommit,
    final RevCommit newCommit,
    final List<String> pathRestrictions) {
    final List<FileDescriptor> objectIdList = new ArrayList<>();
    final TreeFilter filter = getJavaFileTreeFilter(pathRestrictions);
    final List<DiffEntry> diffs = this.git.diff()
        .setOldTree(prepareTreeParser(repository, oldCommit.get().getTree()))
        .setNewTree(prepareTreeParser(repository, newCommit.getTree()));
            setPathFilter(filter)
        .call();
    for (final DiffEntry diff : diffs) {
        if (diff.getChangeType().equals(DiffEntry.ChangeType.DELETE)) {
            continue;
        }
        objectIdList.add(new FileDescriptor(diff.getNewId().toObjectId(), diff.
            getNewPath()));
    }
    return objectIdList;
}
```

Метод отримує посилання на репозиторій, що не додає значної цінності до нашого огляду, оскільки це потрібно лише для деякої внутрішньої обробки. Далі надаються oldCommit та newCommit для вказівки на коміти, між якими ми хочемо отримати різницю. Параметр pathRestrictions не має нас надалі цікавити, оскільки він потрібен лише для обмеження аналізу певними каталогами. У рядку 6 створюється фільтр для перевірки лише файлів Java, оскільки це єдині типи файлів, які буде обробляти аналіз. Починаючи з рядка 8, JGit генерує список змін між старим та новим комітом. Цикл for, що охоплює рядки з 12 по 18, створює FileDescriptor — простий об'єкт-контейнер, що містить дані для подальшого аналізу: дані для легкого пошуку змінених файлів, а також інформацію про модифікації. Випадок DELETE обробляється спеціально, оскільки — навіть якщо це зміна, при якій весь вміст видаляється з файлу, а отже, і сам файл — з порожнього файлу неможливо зібрати значущі дані. Тому цей файл можна пропустити при аналізі. Нарешті, повертається список змін. Шляхом перевірки, чи список порожній, можна виявити, чи можна пропустити checkout поточного коміту, зменшуючи таким чином непотрібні операції вводу/виводу. Після обробки репозиторію, служба може клонувати репозиторій, обійти всі доступні коміти гілки та надати список фактично змінених файлів для кожного коміту, які можуть бути проаналізовані.

3.3. Процес аналізу файлів

Перед будь-яким збором структурних даних або обчисленням метрик кожен змінений файл вихідного коду повинен бути перетворений в відповідне представлення AST. Завдяки JavaParser ми можемо покладатися на дуже здібний і зрілий проект парсингу. Передаючи вихідний код, який ми хочемо проаналізувати, JavaParser повертає AST для файлів, а також

можливість керувати його будівельними блоками за допомогою патерну відвідувача.

3.3.1. Збір структурних даних

Основне завдання збору структурних даних — заповнити модель даних, показану на рисунку 2.1. Всі записи, крім метрик даних, повинні бути заповнені, перш ніж ми зможемо обчислити будь-які з них, оскільки нам потрібна основна структура, заповнена іменами класів та методів. Це завдання здається простим, але ми повинні враховувати деякі особливості Java. Кожен файл Java містить один публічний клас (рахуючи енум і інтерфейс також), але може містити нескінченно багато непублічних класів. Отже, ми повинні стежити за класом, який ми в даний момент аналізуємо. Це необхідно, тому що відвідувач, наданий для AST, автоматично відвідує кожен вузол у правильному порядку, але не надає легкого доступу до батьківських вузлів. Тому невідомо, до якого класу належить метод. Обробка відстеження класів безпосередньо в відвідувачах не здається хорошою ідеєю дизайну, оскільки поточний клас є більш даними, специфічними для інформації, і не є релевантним для поточного відвідувача.

Хоча файл може містити кілька класів, він також може містити вкладені класи; рішення полягає в тому, щоб помістити ім'я класу в стек класів і створити FQN на основі ім'я пакета та всього стека класів. Складніше обробляти анонімні класи; приклад показано в лістингу 3.5. FQN інтерфейсу легко визначити, code.analysis.MyInterface, як і публічного класу, code.analysis.MyClass. А як же з анонімним класом, який утримує поле clazz? Використання вкладеного підходу класу та code.analysis.MyClass.MyInterface призведе до помилки при другому інстанціюванні анонімного класу MyInterface. Використання ім'я поля призведе до довільного FQN. Крім того, AST обробляє ім'я як батьківський об'єкт виразу створення об'єкта. Простий підхід до вирішення цієї проблеми — додати індекс до типу, який

збільшується при кожній зустрічі іншого анонімного класу. Результуючий FQN — code.analysis.MyClass.MyInterface#1.

Лістинг 3.5. Приклад класу, що містить анонімний клас

```
package code.analysis;
interface MyInterface {
    public void doSomething();
}
public class MyClass implements MyInterface{
    MyInterface clazz = new MyInterface() {
        public void doSomething() {}
    };
}
```

Сніпет також демонструє ще одну проблему, з якою ми маємо справу: перевантажені методи. Обидва методи класу MyClass наразі призводять до однакового FQN: code.analysis.MyClass.doSomething. Це можна було б вирішити аналогічно до того, як це було зроблено для анонімних класів. Однак, більш елегантним рішенням є хешування типів параметрів, оскільки вони мають бути унікальними. В іншому випадку код Java був би невалідним. Таким чином, безпараметровий метод отримує FQN code.analysis.MyClass.doSomething#1, а метод, що очікує int, отримує code.analysis.MyClass.doSomething#1980e. Після реалізації генерації FQN, включення решти атрибутів даних, таких як надклас, інтерфейси або модифікатори, зводиться лише до взаємодії з AST для отримання відповідних даних.

3.3.2. Визначення типів

Одночасно зі збором структурних даних ми повинні пам'ятати, що також потрібно отримати типи методів, параметрів або полів. Точніше, нам потрібен FQN типів, щоб показати, де вони визначені. Виявлення типу — це нетривіальне завдання, оскільки ми повинні шукати в кожному файлі, що

міститься в поточному пакеті, і в кожному файлі, вказаному в імпортах. В межах цих файлів потрібно шукати тип. Розв'язувач типів JavaSymbolSolver, який міститься в проекті JavaParser, може допомогти вирішити багато типів. ReflectionTypeSolver — один із доступних розв'язувачів. Він самостійно вирішує примітивні та вбудовані типи. Розв'язувач символів прикріплюється до JavaParser і, отже, доступний під час проходження дерева всередині відвідувача. Оскільки об'єкти вузлів параметрів, а також інші вузли, які містять типи, мають поле для отримання невизначеного об'єкта типу, цей тип можна легко визначити, викликавши метод resolve() об'єкта. Одразу після виклику розв'язувач типів спробує визначити тип у межах своїх обмежень. Оскільки ReflectionTypeSolver не може визначити ні пакетні, ні проектні типи, нам потрібен JavaParserTypeSolver. Для невеликих проектів може бути достатньо надати розв'язувачу всю папку з вихідним кодом. Для великих проектів можна використовувати змінну середовища ANALYSIS_SOURCE_DIRS, щоб обмежити визначення типів до певної папки.

Хоча багато типів можна визначити цим способом, використовуючи ці два розв'язувача типів, ми не можемо визначити типи, визначені в зовнішніх jar-файлах. Отже, ми можемо використовувати простий пошук імпорту як тимчасовий варіант. Він також слугує простим запасним варіантом, якщо станеться виняток під час визначення. Пошук ускладнюється тим, що ми повинні правильно обробляти масиви.

3.3.3. Обробка контексту

Як вже зазначалось, дані аналізу повинні бути надіслані до код-сервісу, що буде детальніше в наступному підрозділі. Хоча формат даних вже встановлено, оскільки ми хочемо використовувати protobuf, нам потрібно визначити, як дані, які ми хочемо надіслати, можуть бути визначені в форматі protobuf. Щоб уникнути проміжного об'єкта зберігання для

збереження структурних даних під час аналізу, надання обгорткових об'єктів для об'єктів повідомлень protobuf дозволяє використовувати їх безпосередньо як об'єкти зберігання, при цьому маючи можливість вбудовувати логіку в обгортку. Це створює абстракцію для додавання даних. Обгортки також надають зручні методи для легкого відстеження поточного контексту, що спрощує додавання нових структурних даних. Подібна, але більш абстрактна обгортка також надається для відвідувачів метрик.

3.4. Реалізація процесу взаємодії

Код-агент і код-сервіс можуть працювати на різних машинах і, отже, повинні мати спосіб обміну пакетами даних через мережу.

Оскільки ми спроектували об'єкти даних аналізу як повідомлення protobuf, розширення обох проектів Java для передачі та отримання вже визначених повідомлень — це проста справа додавання коду служби gRPC до відповідних файлів .proto, як показано в лістингу 3.6. Показаний код — це все, що нам потрібно для визначення комунікації, де клієнт надсилає пакет FileData серверу і не очікує нічого в відповідь.

Лістинг 3.6. Визначення gRPC-сервісу для повідомлення FileData

```
service FileDataService {
    rpc sendFileData (FileData) returns (google.protobuf.Empty) {}
}
```

Proto згенерує весь необхідний код для створення клієнта та серверного коду для нас. Нам потрібно отримати лише клієнт gRPC для сторони код-агента, який зручно надається за допомогою анотації Quarkus. Лістинг 3.7 показує весь код, необхідний для використання клієнта gRPC та надсилання повідомлення protobuf.

Лістинг 3.7. Реалізація клієнта gRPC у код-агенті

```
@GrpcClient(GRPC_CLIENT_NAME)
FileDataServiceGrpc.FileDataServiceBlockingStub fileDataGrpcClient;

public void sendFileData(final FileData fileData) {
    fileDataGrpcClient.sendFileData(fileData);
}
```

На стороні сервера (code-service), файл protobuf такий самий, як і визначення комунікації, не змінюється. Серверний код також простий, як показано в лістингу 3.8.

Лістинг 3.8. Реалізація сервера gRPC у код-сервісі

```
@GrpcService
public class FileDataServiceImpl implements FileDataService {
    @Override
    public Uni<Empty> sendFileData(final FileData request) {
        // робимо щось з даними, наприклад, передаємо їх до ExplorViz
        return Uni.createFrom().item(() -> Empty.newBuilder().build());
    }
}
```

На рисунку 3.2 показано результуючі шляхи комунікації. Пакети FileData та CommitReport надсилаються, коли вони готові на стороні клієнта, і не вимагають нічого в відповідь.

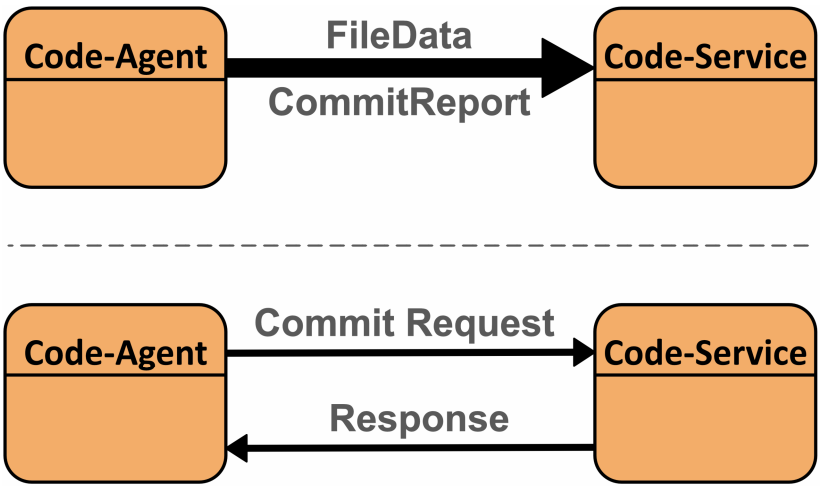


Рисунок 3.2 - Зв'язок між code-agent і code-service через gRPC

З іншого боку, StateData слідує типовому шаблону запит-відповідь. Код-агент надсилає запит, який містить поточну гілку для аналізу, а код-сервіс відповідає, надсилаючи найновіший ідентифікатор SHA коміту, який є в базі даних.

3.5. Реалізація метрик вихідного коду

Щоб забезпечити легко розширювану структуру проекту, всі метрики реалізовані як незалежні відвідувачі AST. Отже, обробка помилок може бути зведена до мінімуму, оскільки невдача обчислення метрики не призведе до повної втрати всіх наступних метрик. Всі відвідувачі слідують одній і тій же структурі. Вони використовують MetricAppender для утримання обробки FQN поза відвідувачем, водночас дозволяючи їм додавати метрики, коли завгодно. Отже, відвідувачі метрик легше підтримувати, оскільки вони містять мало неметричного коду. MetricAppender надає ті самі функції, що й FileDataHandler, використаний для побудови структурних даних. Однак оскільки структурні дані створюють записи для методів і класів, використання більш важке і залежить від знання внутрішнього представлення.

Лістинг 3.9. Методи для обробки вузлів MethodDeclaration та ClassOrInterfaceDeclaration.

```
public void visit(final ClassOrInterfaceDeclaration n, final Pair<MetricAppender,
    Object> data) {
    data.a.enterClass(n);
    data.a.putClassMetric("someClassMetric", "classMetricValue");
    super.visit(n, data);
    data.a.leaveClass();
}
public void visit(final MethodDeclaration n, final Pair<MetricAppender, Object>
    data) {
    data.a.enterMethod(n);
    data.a.putMethodMetric("someMethodMetric", "methodMetricValue");
    super.visit(n, data);
    data.a.leaveMethod();
}
```

Додавання більшої кількості метрик до аналізу в майбутньому більш ймовірне, ніж реструктуризація представлення даних; MetricAppender пропонує спрощений і легший інтерфейс.

Щоб відстеження поточного повного кваліфікованого імені (FQN) методу або класу виконувалося MetricAppender і не засмічувало відвідувача метрик (metric visitor). Оскільки MetricAppender є першим елементом Pair, доступ до нього здійснюється через data.a.

Метрика кількості рядків коду є єдиним винятком із вищезазначеного правила одного відвідувача на метрику, оскільки вона є частиною збору структурних даних. Це тому, що обчислення LOC є дешевим з точки зору продуктивності, оскільки JavaParser безпосередньо надає кількість рядків у файлі, класі або методі. Крім того, AST містить кількість коментарів, що дозволяє обчислити рядки вихідного коду на основі вже наявної інформації. Обчислення LOC також повинно бути досить стійким, оскільки єдиною помилкою, з якою воно може зіткнутися, є відсутність необхідної інформації від AST, що в кінцевому рахунку є результатом невдалого парсингу спочатку.

Лістинг 3.10. Реалізація розрахунку глибини для циклу for

```
public void visit(final ForStmt n, final Pair<MetricAppender, Object> arg) {
    currentDepth++;
    maxDepth = Math.max(maxDepth, currentDepth);
    super.visit(n, arg);
    currentDepth--;
}
```

Код метрики глибини вкладених блоків простий за структурою. Крім деякої складності щодо MetricAppender, він відвідує всі вузли керуючих структур (for, for-each, while, do-while, if, try, switch і case), методів і конструкторів, а також деякі спеціальні вузли (лямбда-вирази та синхронізовані блоки). Для кожного вхідного вузла лічильник збільшується, а для кожного вихідного — зменшується. Листинг 3.10 показує реалізацію

обробки вузла for. currentDepth — це лічильник, а maxDepth — змінна для збереження максимальної глибини, знайденої в контексті.

Ця метрика розраховується для кожного методу або конструктора у файлі та додається до даних аналізу.

Збір метрики цикломатичної складності працює аналогічно до вищеописаної глибини вкладених блоків. Основна різниця полягає в тому, що ми підраховуємо всі оператори керуючих структур незалежно від їхньої глибини. Цикломатична складність не є лише метрикою методу, але також використовується для класів і всього файлу. Крім того, потрібно також обчислити зважену цикломатичну складність. Оскільки збір даних більш складний, ніж раніше, ми використовуємо комбінацію здатності MetricAppender відстежувати поточний метод із простим HashMap для утримання відвідувача якомога чистим. Карта бере FQN методу як ключ і має ціле значення як значення, яке представляє кількість виникнень керуючих структур. Цикломатична складність не рахує лише ключові оператори потоку управління. Вона також враховує логічні та бінарні вирази &&, ||, & і |. Обчислення було реалізовано як можливо ближче до оригіналу.

Після повного аналізу всього файлу HashMap містить усі методи та їхні значення цикломатичної складності. Їх можна безпосередньо додати до FileData як записи метрик.

Метрики класу розраховуються шляхом підсумовування всіх значень метрик методів, що належать до класу. Те саме стосується метрики файлу. Зважена цикломатична складність — це середнє значення з усіх методів, що призводить до тривіального обчислення значення цикломатичної складності класу, поділеного на кількість методів, які містить клас.

Метрика LCOM4 є найбільш складною для обчислення порівняно з вищеописаними метриками. Як показано на рисунку 2.2, графове представлення вихідного коду є хорошим способом обчислення залежностей між методами та полями, що досягається додаванням вершини для кожного

методу та поля до неорієнтованого графу. Потім ми повинні проаналізувати тіло методу на предмет доступу до полів. Хоча JavaParser допомагав нам раніше, він не надає функціональності для виявлення контексту полів або методів. Отже, ClassA.valueA і this.valueA є доступами до полів, хоча перший може бути доступом до поля з іншого класу, а другий — точно до локального доступу до поля. Однак навіть перший є локальним доступом, якщо поточна назва класу — ClassA. Отже, ми повинні перевірити кожний доступ до поля, щоб побачити, чи збігається ім'я поля з одним із імен полів класу. Крім того, якщо це так, його також потрібно перевірити, чи контекст є this або ім'я поточного класу. Лише тоді поле доступне всередині методу, і ми можемо додати ребро для з'єднання обох вершин. Крім того, оскільки JavaParser не розрізняє доступ до локальної змінної та доступ до поля, ми також повинні враховувати приховування змінних. Ми розглянемо лістинг 3.11, щоб обговорити приховування полів.

Лістинг 3.11. Приклад затінення полів (field shadowing) та областей видимості (scopes) для викликів методів

```
class A {
    int x = 1;
    void methodA() {
        int x = 4;
        ClassB.methodB();
        x++;
    }
    void methodB() {
```

У прикладі змінна x у рядку 4 затіняє поле класу x, оскільки вони мають однакову назву. У рядку 6 відбувається доступ до локальної змінної x; таким чином, methodA не отримує доступу до жодних полів класу (що належать класу A). У рядку 9 жодна інша змінна не затіняє поле x; отже, methodB отримує доступ до поля. Інша схожа проблема виникає у рядку 5, де викликається methodB, але не той, що належить класу A. Так само, як і для полів, ми також повинні перевіряти контекст для викликів методів.

Особливо обробляються методи з порожніми тілами та успадковані методи, оскільки вони належать до суперкласу. Обидва пропускаються в процесі аналізу.

Після завершення ми можемо дослідити отриманий граф. Якщо всі вершини транзитивно пов'язані, значення LCOM дорівнює 1 і може бути додано як метрика класу через MetricAppender. В іншому випадку кількість підграфів визначає значення LCOM. Єдина причина, чому отримане значення дорівнює 0, полягає в тому, що це порожній клас. Це також може статися, якщо всі методи є успадкованими.

Вищеописані метрики реалізовані як стартова точка для побудови розширюваної робочої структури. Оскільки вже реалізовані метрики мають різну складність, вони повинні слугувати хорошими джерелами інформації для майбутніх метрик. MetricAppender спроектований для абстрагування внутрішньої структури даних та обробки. Оскільки ми бачили, що відвідувачі реалізовані з об'єктом Pair як акумулятором, лівий об'єкт зарезервований для MetricAppender, а правий об'єкт вільний для будь-якої реалізації метрики. Проект містить VisitorStub, який реалізує базову обробку класів і методів і може бути використаний як шаблон. Додавання нового відвідувача метрик до проекту вимагає його налаштування в методі calculateMetrics класу JavaParserService.

3.6. Git метрики та контейнер Docker

Останній тип метрик, який нам потрібно зібрати, — це метрики Git. Як вже зазначалося в попередньому підрозділі, ці метрики є корисними лише після того, як візуалізація їх агрегувала. Оскільки подальша обробка даних неможлива через обмеження легкої безстанової служби, ми можемо зібрати дані лише для майбутньої обробки.

Хоча ці дані потрібні зараз, їх збір відбувається в тих самих методах, які відповідальні за виявлення змін. Там ми маємо доступ до різниці між останньою ревізією файлу та його поточним вмістом. Отримані дані, які містять кількість змінених, доданих і видалених рядків, додаються до пакета FileData файлу.

Авторів можна ідентифікувати за їхніми самостійно обраними іменами користувачів або адресами електронної пошти для облікового запису Git. Оскільки ім'я користувача більш схильне до змін, ми вибрали адресу електронної пошти як ідентифікатор автора. Отже, autorIdent, прикріплений до кожного коміту, використовується для отримання ідентифікаційної адреси електронної пошти і додається до кожного файлу аналізу, створеного для коміту, про який йде мова.

Оскільки Quarkus слідує підходу "спочатку контейнер", створити контейнер Docker з проекту на основі Quarkus за допомогою jib 9 досить просто. Усі необхідні інструменти збірки вже інтегровані або доступні як розширення. Створений контейнер Docker був завантажений до DockerHub 2, щоб надати готову до використання службу аналізу.

Для керування функціональністю та налаштування часу виконання використовуються змінні середовища. Quarkus вже надає зручний спосіб інтеграції цих змінних.

Лістинг 3.12. Витяг з файлу application.properties проєкту

```
explorviz.gitanalysis.remote.url=${ANALYSIS.REMOTE_URL:${CI.REPOSITORY_URL:}}
explorviz.gitanalysis.branch=${ANALYSIS.BRANCH:${CI.COMMIT.BRANCH:}}
```

Лістинг 3.12 показує з'єднання змінних середовища з внутрішньо використаними властивостями. Змінна середовища ANALYSIS_REMOTE_URL може бути встановлена користувачем. Друга змінна, CI_REPOSITORY_URL, надається GitLab CI. Отже, якщо користувач не вкаже URL, але запустить контейнер у GitLab CI, URL автоматично

встановлюється на поточний репозиторій. Це заздалегідь встановлено для всіх обов'язкових змінних. Ми представимо інтеграцію в наступному підрозділі.

3.7. Оцінка працездатності методології

Основна мета оцінки – перевірити працездатність підходу з використанням Docker-контейнерів та надійність аналізу, а також отримати загальне уявлення про порядок величин продуктивності. Крім того, ми хочемо перевірити можливість переналаштування аналізу та доцільність його інтеграції з іншими CI-провайдерами, відмінними від запланованого. Ми оцінимо поточний стан аналізу в контексті допомоги в розумінні програми та визначимо значущість вихідних даних. Це включає запуск екземпляра в локальному та контрольованому середовищі, яке легко моніторити, для перевірки отриманих вихідних даних. З іншого боку, для підтвердження портативності ми інтегруємо сервіс у різні зовнішні середовища виконання без необхідності збору складних вихідних даних.

Основна мета оцінки — перевірити працездатність підходу з використанням контейнерів Docker і стійкість аналізу, а також загалом зрозуміти масштаб продуктивності. Крім того, ми хочемо перевірити можливість переналаштування аналізу та можливість інтеграції з іншими постачальниками CI, окрім передбаченого. Ми оцінимо працездатність контейнерного підходу та корисність створених даних. Це включає запуск екземпляра в легко контрольованому локальному середовищі для перевірки створеного виводу.

3.7.1. Прикладні сценарії

Щоб показати та оцінити можливості та використання служби аналізу, ми розглянемо 4 прикладних застосування служби. Спочатку ми побачимо,

як налаштувати службу як автономний додаток і запустити її на головній гілці проекту spring-petclinic. Переходячи до більш складного проекту plantUML, ми виконаємо аналіз зрілого проекту, щоб продемонструвати обробку помилок та стійкість аналізу, а також поточні обмеження.

Після цього, ми побачимо, як інтегрувати службу в GitLab CI, що працює на демонстраційному проекті. Завершимо базовою інтеграцією з GitHub Actions для перевірки портативності контейнерного дизайну.

Перш ніж переходити до більш складної інтеграції служби аналізу, ми запустимо контейнер Docker як автономну неінтегровану систему. Це може бути цікавим застосуванням для розробника, який хоче запустити аналіз локально на машині розробки.

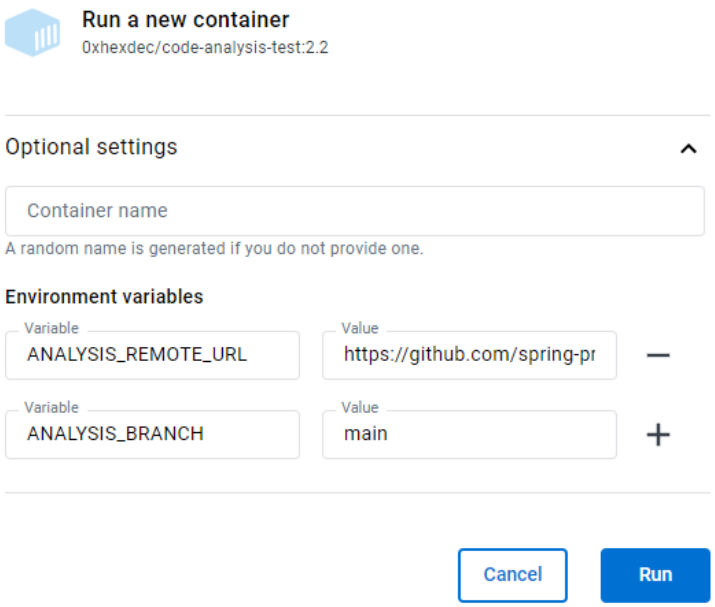


Рисунок 3.3 - Docker Desktop - конфігурація для автономного аналізу PetClinic

Щоб проаналізувати проект spring-petclinic, спочатку потрібно завантажити попередньо побудований Docker-образ. Потім ми вибираємо завантажений образ у Docker Desktop і натискаємо кнопку запуску. Конфігурація, готова до запуску, повинна виглядати схоже на ту, що

показана на рисунку 3.3, де вибрана головна гілка, а URL вказано від репозиторію, розміщеного на GitHub. Це вкаже аналізу клонувати репозиторій petclinic до тимчасової папки у файловій системі контейнера, а також запустити аналіз на всіх комітах головної гілки. Дані аналізу будуть збережені в контейнері як JSON-файли.

Після виконання вивід буде схожий на той, що показано на рисунку 3.4. Час виконання склав трохи більше 33 секунд для 845 комітів і створив 1848 JSON-файлів (5,5 Мбайт даних), збережених у папці /home/jboss/analysis-data. З 845 цих файлів — звіти про коміти, залишається 1003 файли FileData (іменовані за схемою: <ім'я_файлу>_<commitID>.json), які містять всі структурні дані та обчислені метрики. Ближчий огляд логу виводу показує два різних попередження під час виконання. Перше (1) — це попередження про тип, яке повідомляє, що тип Serializable не може бути визначений. Після розслідування цієї помилки виявилося, що це не помилка аналізу. Файл вихідного коду не містив необхідного імпорту; отже, файл мав би створити помилку компіляції. Друге попередження (2) повідомляє про наявність імпортів з підстановкою. Поведінка підстановки типів за замовчуванням відключена, і типи, які можуть бути визначені імпортами з підстановкою, призводять до попереджень. Ці попередження зникають після додавання змінної середовища ANALYSIS_RESOLVE_WILDCARDS=true.

Рисунок 3.4 - Вивід результатів аналізу petclinic

Хоча створені дані не призначені для читання людиною і повинні бути оброблені та візуалізовані для демонстрації своєї повної корисності, ми розглянемо невеликий приклад у лістингу 3.13, щоб зрозуміти, що ми отримали.

Лістинг 3.13. Фрагмент даних аналізу для AbstractTraceAspect.java на коміті #bcda93f


```
"commitID": "bcda93f280e9174f4e5c44fc830a864a963633a5",
"fileName": "AbstractTraceAspect.java",
"packageName": "org.springframework.samples.petclinic.aspects",
"importName": ["org.aspectj.lang.JoinPoint", ...],
"classData": {
    "org.springframework.samples.petclinic.aspects.AbstractTraceAspect": {
        "type": "ABSTRACT_CLASS", "modifier": ["public", "abstract"],
        "field": [{
            "name": "logger", "type": "org.slf4j.Logger",
            "modifier": ["private", "static", "final"]
        }],
        "methodData": {
            "org.springframework./.../.aspects.AbstractTraceAspect.trace#ac4c8c48": {
                "returnType": "void", "modifier": ["public"],
                "parameter": [{
                    "name": "jpsp","type": "org.aspectj.lang.JoinPoint.StaticPart"
                }],
                "metric": {
                    "loc": "6", ... ,"nestedBlockDepth": "2"
                }
            },
            "org.springframework./.../.aspects.AbstractTraceAspect.traced#1": {
                ... }
        }
    }
}
```


Як бачимо у витягу даних, файл містить один абстрактний клас (Abstract Class), який містить два методи під назвою trace. Один очікує на org.aspectj.lang.JoinPoint.StaticPart як параметр, а інший є безпараметричним. Метрики були створені для методів, класу та файлу. Дані Git додано у вигляді електронної адреси автора (змінено на довільну) та кількості змінених рядків, що вказує на незначну зміну. Фактично, було замінено лише бібліотеку логера з apache.commons на slf4j.

Щоб симулювати віддалений запуск для цього репозиторію plantUML, нам потрібен кінцевий пункт для відправки даних аналізу. Для цього застосування ми запустимо код-сервіс всередині IntelliJ IDEA на тій же машині, що й контейнер Docker. Це найпростіший варіант налаштування, оскільки він не вимагає жодного зовнішнього перенаправлення портів і може бути використаний як перший проект для ознайомлення з службою аналізу. Перш ніж щось інше, ми повинні запустити завдання quarkusDev gradle для запуску код-сервісу. Одразу після запуску служби ми можемо продовжити налаштування код-агента для аналізу репозиторію. Змінні середовища для налаштування служби для аналізу основної папки з вихідним кодом plantUML показані в лістингу 3.14.

Лістинг 3.14. Змінні середовища, використані для запуску аналізу plantUML

```
ANALYSIS_REMOTE_URL=https://github.com/plantuml/plantuml
ANALYSIS_BRANCH=master
ANALYSIS_SEND_TO_REMOTE=true
GRPC_HOST=host.docker.internal
ANALYSIS_SOURCE_DIRS=src
ANALYSIS_RESTRICT_DIRS=src/net/sourceforge/plantuml
```

Аналіз з обмеженням на папку з вихідним кодом тривав близько 43 хвилин і проаналізував 1130 комітів до завершення. Необмежений тестовий запуск у попередньому стані проекту з меншою кількістю метрик тривав більше 6 годин. Оскільки однією з основних цілей тесту з plantUML було перевірити більш розвинену кодову базу на предмет потенційних нестабільностей у процедурі аналізу, багатогодинне виконання могло не створити більш значущих даних, ніж ми вже отримали. Дані надсилалися до код-сервісу під час виконання і могли бути спостережені як консольний вивід.

Під час виконання відбулося кілька помилок, але жодна з них не завадила успішному завершенню. Ми хочемо зосередитися на двох типах помилок, які відбулися під час виконання. На рисунку 3.5 показано першу, більш серйозну помилку. JavaParser не міг створити AST з файлу вихідного коду через використання зарезервованого ключового слова. Аналіз виявив помилку та обійшов файл. Таким чином, ми втратили дані для всього файлу, але оскільки аналіз залежить від AST, ми не можемо надати простий обхідний шлях.

```
ERROR (main) Catched Javaparser exception, can't handle this, skipping file: CucaDiagramFileMakerJDot.java
ERROR (main) (line 382,col 50) '_' is a reserved keyword.
```

Рисунок 3.5 - Серйозна помилка: JavaParser не міг створити AST з файлу

Помилка на рисунку 3.6 спричинена CyclomaticComplexityVisitor і вказує на EmptyStackException. Виявилося, що відсутня реалізація для мовної функції спричинила помилку. Подібні помилки відбувалися багато разів під час розробки, оскільки Java пропонує багато мовних функцій, які можуть бути невідомими або рідко використовуваними. Причиною цієї помилки є припущення CyclomaticComplexityVisitor, що він працює в контексті методу при зустрічі керуючих структур. Це неправильне припущення, оскільки керуючі структури також дозволені в контексті статичних блоків всередині класів. Ця помилка, на щастя, призводить лише до відсутності запису метрики цикломатичної складності, але не заважає аналізу в іншому.

```
ERROR (main) Unable to create cyclomatic complexity metric for File: SecureCoder.java
ERROR (main) null: java.util.EmptyStackException
```

Рисунок 3.6 - Помилка обчислення метрики. Це призводить лише до відсутності запису метрики в даних аналізу

Інтеграція служби в GitLab CI — це найзручніший спосіб використання служби аналізу, оскільки вона була спроектована для роботи на ній. Обов'язкові змінні середовища вже заздалегідь встановлені на змінні, що надаються GitLab CI; отже, інтеграція служби в основному взаємодія з інтерфейсом користувача gitlab.com. Ми покажемо інтеграцію, використовуючи публічний репозиторій BusyDoingNothing 2. Репозиторій містить проект Java з кількома файлами Java. Хоча він виглядає як запуск, він не призначений для запуску. Більшість класів слугують для тестування конкретних аспектів аналізу, реалізації мовних функцій або спроектовані для перевірки обчислення метрик. Це робить його ідеальним кандидатом для використання проекту для тестування та перевірки інтеграції служби аналізу в різні ранери.

Інтеграція служби аналізу в проект, розміщений на GitLab, — це просто додавання файлу .gitlab-ci.yml у верхній папці проекту з вмістом, показаним у лістингу 3.15, щоб майже повністю налаштувати контейнер. Змінні середовища ANALYSIS_REMOTE_URL і ANALYSIS_BRANCH автоматично заповнюються GitLab CI, що означає, що за замовчуванням аналіз виконується проти поточного репозиторію. Оскільки поточне завдання CI надає ім'я гілки, на якій воно працює, усі гілки автоматично відстежуються та аналізуються за замовчуванням. Навіть якщо нова гілка буде відправлена, не потрібно змінювати конфігурацію.

Лістинг 3.15. Файл gitlab-ci.yml, який використовується для демонстрації коду, що міститься в репозиторії, до якого прив'язаний CI.

```
stages:
    - analysis
    analysis-job:
    stage: analysis
    image: 0xhexdec/code-analysis-test:2.2
    variables:
        ANALYSIS_FETCH_REMOTE: "true"
        ANALYSIS_SEND_TO_REMOTE: "true"
        ANALYSIS_CALCULATE_METRICS: "true"
        ANALYSIS_RESOLVE_WILDCARDS: "true"
```

Як можна було вже помітити, конвеєр (pipeline) налаштований таким чином, що використовується з'єднання з віддаленою кінцевою точкою (remote endpoint); отже, необхідне встановлення IP та порту. Хоча ці налаштування також можна визначити у файлі .gitlab-ci.yml, оскільки проєкт є відкритим (open source) та публічним, особисті дані, такі як IP та порт, були б видимі для всіх. За звичайних обставин це небажано. Щоб приховати приватні дані, рекомендується визначити GRPC_HOST та GRPC_PORT у GitLab як приватні змінні, які можна знайти в розділі Settings->CI/CD->Variables (Налаштування -> CI/CD -> Змінні). Змінні, визначені там, обробляються так само, як і змінні, визначені у файлі конфігурації, але видимі лише для учасників проєкту.

Як тільки нові коміти надсилаються до репозиторію, аналіз виконується, як показано на рисунку 3.7. Цей контейнер був налаштований на отримання статусу перед виконанням. Це призведе до прискорення часу виконання для наступних викликів.

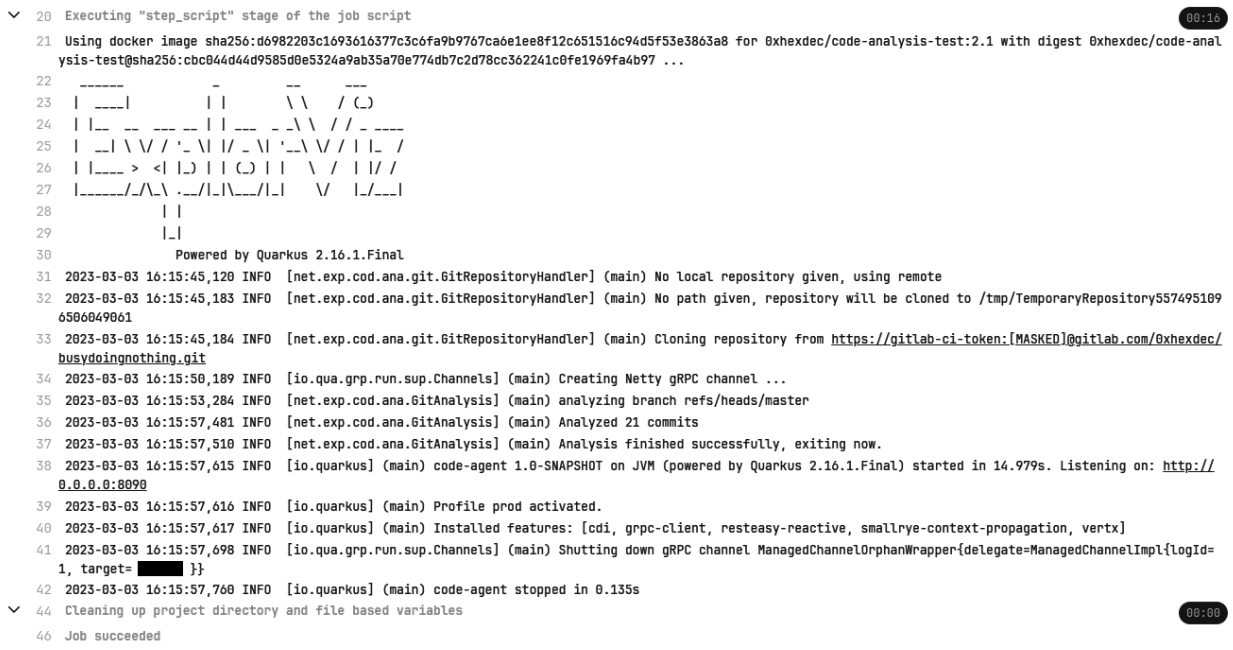


Рисунок 3.7 - Вивід успішного виконання контейнера за допомогою GitLab CI

Не всі проекти готові запускати CI як автономне рішення на локальній машині або мігрувати на GitLab. Тому ми запустимо аналіз з GitHub Actions, щоб показати портативність створеної служби аналізу. Інтеграція з GitHub не вбудована в службу аналізу і повинна бути налаштована вручну.

Подібно до описаного вище варіанту GitLab, спочатку потрібно створити проект Java. Проект потрібно відправити в репозиторій GitHub. У цьому випадку ми використовуємо публічний репозиторій. Наступним кроком є налаштування Action шляхом додавання main.yml, показаного в лістингу 3.16, до папки .github/workflows/ у проекті. Одразу після відправки в головну гілку служба аналізу почне працювати, але їй бракуватиме змінних середовища GRPC_HOST і GRPC_PORT. Ці змінні повинні бути визначені як безпечні змінні середовища проекту через інтерфейс користувача через налаштування -> Безпека -> Секрети та змінні -> Дії, вибрати Змінні та додати їх як змінні репозиторію.

Лістинг 3.16. Базовий файл main.yml, який використовується для GitHub Actions для запуску сервісу аналізу на поточному репозиторії

```yaml
name: Analysis
on:
    push:
        branches: ["main"]
jobs:
    analysis-job:
        runs-on: ubuntu-latest
        container:
            image: 0xhexdec/code-analysis-test:2.2
            env:
                GRPC_HOST: ${{vars.GRPC_HOST}}
                GRPC:PORT: ${{vars.GRPC_PORT}}
                ANALYSIS_REMOTE_URL: https://github.com/${{github.repository}}.git
                ANALYSIS_BRANCH: ${{ github.ref_name }}
                ANALYSIS_FETCH_REMOTE: "true"
                ANALYSIS_SEND_TO_REMOTE: "true"
                ANALYSIS_CALCULATE_METRICS: "true"
                ANALYSIS_RESOLVE_WILDCARDS: "true"
    steps:
        - name: Run analysis
        working-directory: /home/jboss
        run: java -jar quarkus-run.jar
```

Звернимо увагу на складнішу конфігурацію порівняно з файлом інтеграції GitLab: для запуску аналізу потрібно викликати quarkus-run.jar вручну, а також визначити робочий каталог.

Action тепер налаштований таким чином, що аналіз запускатиметься при кожному push у гілку main. Вивід GitHub Actions можна побачити на рисунку 3.8.

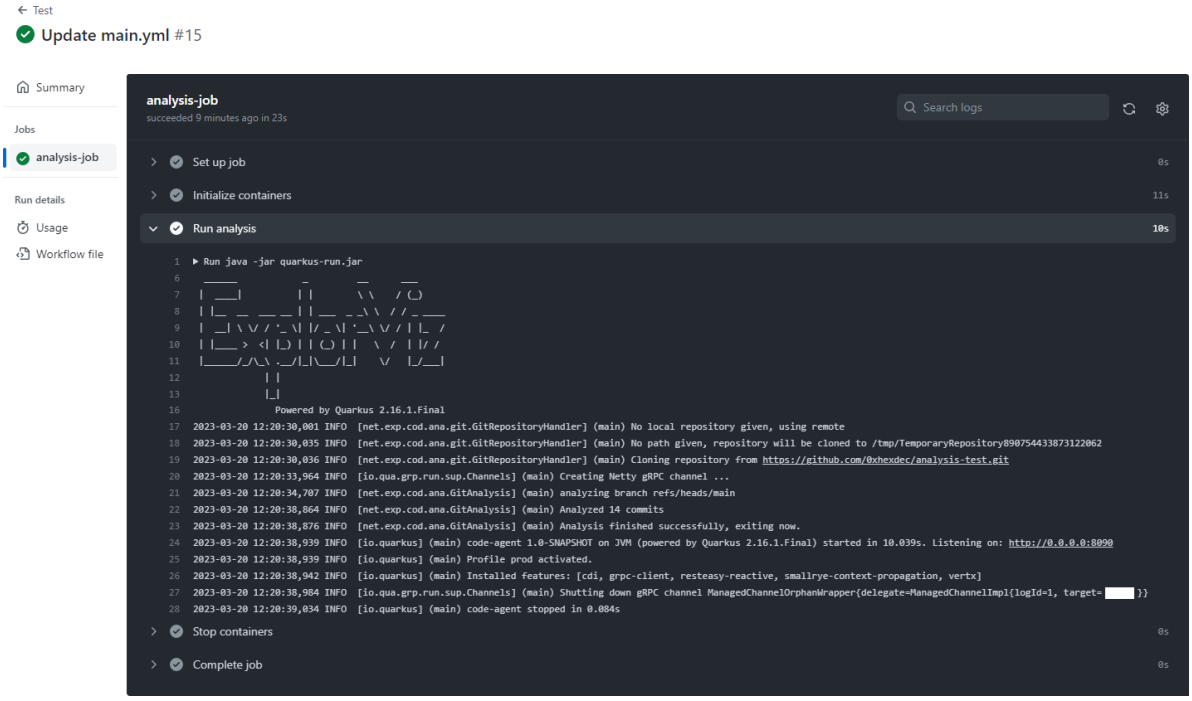


Рисунок 3.8 - Вивід успішного виконання контейнера за допомогою GitHub Actions

Як показано вище, ми успішно запустили аналіз на різних проектах без значних проблем. Ми не зіткнулися з жодним збоєм служби, навіть якщо ми зіткнулися з деякими серйозними помилками під час етапів аналізу. Аналіз завжди міг повністю відновитися. Крім того, ми показали, що служба, реалізована як контейнер Docker, дозволяє легко мігрувати до різних ранерів. Хоча спеціально підготовлена для роботи на GitLab CI, ми показали, що ранер можна легко налаштувати, змінивши деякі змінні середовища, щоб

мати можливість працювати з GitHub Actions. Інтеграція з іншими ранерами також можлива.

Крім того, було помітно, що структурний аналіз був більш стабільним і надійним, ніж обчислення метрик, навіть якщо йому подавали вихідний код, який призводив до помилок компіляції. Дизайн інкапсульованих відвідувачів для кожної метрики спрацював не лише з точки зору підтримки, але й загалом підвищив стійкість аналізу, оскільки помилка обчислення однієї метрики не впливала на результат іншої.

Випадково виявилося, що в обробці не передбачено переривання зв'язків. Хоча аналіз не запускається, якщо код-сервіс недоступний при запуску, якщо зв'язок переривається, аналіз продовжує надсилати свої дані до вказаного кінцевого пункту, поки не завершиться. Це може бути через рішення встановити відповідне повідомлення для FileData та StateData пакетів на empty.

Під час розробки та для деяких тестових виконань результати метрик випадково перевірялися за допомогою Checkstyle та PMD для порівняння (за умови, що метрика була обчислена інструментом, що не завжди було так для LCOM4). Деякі тести також реалізовані для метрик, але оскільки ці тестові випадки штучно створені для перевірки, вони можуть не бути хорошою заміною реального коду.

Хоча деякі метрики легше реалізувати або застосовувати до мови, інші — ні. Наприклад, LOC відносно легко зрозуміти, але її значення сильно залежить від реалізації. Хоча це може не бути проблемою для рядків коду, оскільки їх порядок величини більш виразистий, ніж саме значення, обчислення цикломатичної складності може дати різні результати в залежності від реалізації. Оскільки оригінальний творець обговорював метрику, використану для Fortran, розвинена мова, така як Java, складається з різних функцій, які можуть бути враховані при обчисленні. Оригінальна

реалізація не враховує мовні особливості, такі як тернарні оператори або foreach і map виклики на потоках.

Як зазначено вище, поточний стан як запуск і стабільна служба робить її простою для включення в існуючий конвеєр CI. Оскільки створені дані повинні бути оброблені та візуалізовані, щоб бути корисними, службу слід вважати технічно виконуваною, але не завершеною до такої міри, щоб зібрані дані були корисні для розробника в контексті розуміння програмного забезпечення. Її слід розглядати як основу для подальшого збору даних та для реалізації аналогічного інструменту в ExplorViz. Дані можуть бути використані для розробки візуалізації, а думки, пропозиції та висновки з цієї дисертації можуть бути використані як стартова точка для покращень, оскільки створено міцну, надійну та розширювану архітектуру служби.

Висновки до розділу

У третьому розділі детально описано практичну реалізацію методології графічного супроводу процесу розробки програмного забезпечення на основі аналізу структурованих даних із репозиторіїв Git. Сформовано архітектурну модель системи, що складається з компонентів Code Service і Code-Agent, які забезпечують автоматизоване збирання, обробку та передачу інформації про вихідний код.

Розглянуто механізми доступу до репозиторію Git, проходження комітів, виявлення змін і збирання релевантних структурних даних. Система підтримує визначення типів, контекстну обробку елементів коду, а також зберігання інформації у форматі, придатному для подальшої візуалізації та аналізу.

Реалізовано функціональність розрахунку метрик вихідного коду та інтеграцію Git-метрик у загальну модель. Для забезпечення масштабованості

та ізольованості середовища розгортання застосовано Docker-контейнери, що спрощує використання системи у різних середовищах розробки.

Проведено тестування працездатності реалізованого підходу на прикладних сценаріях, що підтвердило ефективність методології у виявленні змін, структурних характеристик і тенденцій розвитку проєктів. Отримані результати створюють підґрунтя для побудови гнучкої системи візуального супроводу процесу розробки, орієнтованої на покращення прозорості та керованості життєвого циклу програмного забезпечення.

ВИСНОВКИ

В дипломній роботі було досліджено, розроблено та імплементовано методологію графічного супроводу процесу розробки програмного забезпечення на основі аналізу вихідного коду та класифікованих даних із репозиторіїв Git.

На першому етапі проведено комплексний аналіз проблематики візуалізації програмного забезпечення. Визначено основні вимоги до інструментів, здатних відображати структуру та поведінку коду, а також обґрунтовано вибір технологічного стеку для реалізації подібних рішень. Проаналізовано наявні засоби та архітектури, зокрема інструмент ExplorViz, що дозволило виявити переваги та обмеження існуючих підходів.

У другому розділі було розроблено методологію статистичного аналізу Java-коду з урахуванням структурних характеристик і метрик якості. Сформовано набір показників, що дозволяють оцінювати складність, зчепленість та інші аспекти програмних артефактів. Також описано методи роботи з історією змін у Git-репозиторіях, що дозволяє враховувати динаміку розвитку проєкту при подальшому аналізі.

У третьому розділі реалізовано архітектуру системи, яка інтегрує аналіз вихідного коду та дані з Git у єдиний процес. Реалізовано компоненти для збирання структурних даних, обчислення метрик та підготовки інформації до візуалізації. Система розгортається в середовищі Docker, що забезпечує її гнучкість і портативність. Оцінка працездатності на прикладних сценаріях підтвердила ефективність розробленої методології у реальних умовах розробки ПЗ.

Таким чином, у межах дослідження запропоновано цілісний підхід до візуального супроводу розробки програмного забезпечення, який поєднує аналіз вихідного коду, метрик складності та історії змін, що забезпечує глибше розуміння технічного стану програмного.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Kim, Y., Kim, J., Jeon, H., Kim, Y.-H., Song, H., Kim, B., & Seo, J. (2020). Githru: Visual Analytics for Understanding Software Development History Through Git Metadata Analysis. arXiv preprint arXiv:2009.03115.
2. Nguyen, Q.H., & Eades, P. (2018). Towards faithful graph visualizations.
3. Nuzrath, S., Amarasinghe, N.H., Liyanage, K.T., Suriyawansa, K., Madanayake, D.P., & Kodagoda, N. (2019). gCodex: a tool to analyze software repositories over time (visualization). In 2019 International Conference on Advancements in Computing (ICAC) (pp. 174–179).
4. Ono, J.P., Freire, J., & Silva, C.T. (2021). Interactive data visualization in Jupyter notebooks. Comput. Sci. Eng., 23(02), 99–106.
5. Packer, H.S., Chapman, A., & Carr, L. (2019). GitHub2PROV: provenance for supporting software project management. In Proceedings of the 11th USENIX Conference on Theory and Practice of Provenance, TAPP 2019.
6. Susanna Ardigò et al. “M3tricity: visualizing evolving software & data cities”. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pages 130–133. doi: 10.1145/3510454.3516831. url: https://doi.org/10.1145/3510454.3516831.
7. Claude Bolduc. “Lessons learned: using a static analysis tool within a continuous integration system”. In: IEEE International Symposium on Software Reliability Engineering Workshops ISSREW (2016), pages 37–40. doi: 10.1109/ISSREW.2016.48.
8. Pimentel, J.F., Freire, J., Braganholo, V., & Murta, L. (2016). Tracking and analyzing the evolution of provenance from scripts. In IPAW 2016 (pp. 16–28).

9. Pinzger, M., Gall, H., Fischer, M., & Lanza, M. (2005). Visualizing multiple evolution metrics. In Proceedings of the 2005 ACM Symposium on Software Visualization, SoftVis 2005 (pp. 67–75).
10. Ragan, E.D., Endert, A., Sanyal, J., & Chen, J. (2016). Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. IEEE Trans. Visual Comput. Graph., 22(1), 31–40.
11. Rentz, N., & von Hanxleden, R. (2024). SPViz: A DSL-Driven Approach for Software Project Visualization Tooling. arXiv preprint arXiv:2401.17063.
12. Alshakhouri, M., Buchan, J., & MacDonell, S.G. (2021). Synchronised Visualisation of Software Process and Product Artefacts: Concept, Design and Prototype Implementation. arXiv preprint arXiv:2105.00705.
13. Ruven Brooks. “Towards a theory of the comprehension of computer programs”. In: International Journal of Man-Machine Studies 18.6 (1983), pages 543–554.
14. Moreno-Lumbreras, D., Gonzalez-Barahona, J.M., Robles, G., & Cosentino, V. (2024). Information Visualization for Agile Software Development. J SYST SOFTWARE.
15. Scheibel, W., Blum, J., Lauterbach, F., & Döllner, J. (2024). Integrated Visual Software Analytics on the GitHub Platform. Computers, 13(2), 33.
16. Silva, C.T., Freire, J., & Callahan, S.P. (2007). Provenance for visualizations: Reproducibility and beyond. Computing in Science & Engineering, 9(5), 82–89.
17. Buse, R.P.L., & Weimer, W. (2010). Learning a metric for code readability. IEEE Transactions on Software Engineering, 36(4), 546–558.
18. Begel, A., & Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In Proceedings of the 36th International Conference on Software Engineering (pp. 12–23).

19. Wojciech Cellary and Sergiusz Strykowski. “E-government based on cloud computing and service-oriented architecture”. In: Proceedings of the 3rd International Conference on Theory and Practice of Electronic Governance. ICEGOV ’09. Bogota, Colombia: Association for Computing Machinery, 2009, pages 5–10.
20. Paul M Duvall, Steve Matyas, and Andrew Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007.
21. Fritz, T., & Murphy, G.C. (2010). Using information fragments to answer the questions developers ask. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (pp. 175–184).
22. Bird, C., & Kalliamvakou, E. (2015). GitHub: A collaborative platform for software development. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (pp. 574–575).
23. Stevens, P. (2013). QWALKEKO: A declarative language for querying the history of versioned software projects. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (pp. 456–466).
24. Barik, T., et al. (2016). Commit Bubbles: Visualizing code commits for software maintenance tasks. In Proceedings of the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (pp. 1–9).
25. Rozenberg, G. (2014). RepoGram: A tool for visualizing software repositories. In Proceedings of the 2014 IEEE Working Conference on Software Visualization (pp. 1–10).
26. Manuel Egele et al. “A survey on automated dynamic malware-analysis techniques and tools”. In: ACM Comput. Surv. 44.2 (2008).
27. S. Elbaum and John Munson. “Code churn: a measure for estimating the impact of code change”. In: Conference on Software Maintenance (Sept. 2000).

28. Lee, S., et al. (2012). Visualizing the evolution of software systems using a 3D city metaphor. In Proceedings of the 2012 IEEE International Conference on Software Maintenance (pp. 1–10).
29. Biazzini, M., et al. (2014). Metagraph: A data structure for representing topologically relevant commits. In Proceedings of the 2014 ACM Symposium on Software Visualization (pp. 1–10).
30. Wilde, N. (2010). Linvis: A tool for visualizing the lineage of software artifacts. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (pp. 1–10).
31. Diehl, S. (2007). Software visualization: Visualizing the structure, behaviour, and evolution of software. Springer Science & Business Media.
32. Eick, S.G., Steffen, J.L., & Sumner, E.E. (1992). Seesoft—a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering, 18(11), 957–968.
33. Laune C. Harris and Barton P. Miller. “Practical analysis of stripped binary code”. In: SIGARCH Comput. Archit. News 33.5 (Dec. 2005), pages 63–68.
34. Wilhelm Hasselbring, Alexander Krause, and M. Bader. “Collaborative software visualization for program comprehension”. In: Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 22). IEEE. 2022.
35. Lanza, M., & Ducasse, S. (2003). Polymetric views—a lightweight visual approach to reverse engineering. IEEE Transactions on Software Engineering, 29(9), 782–795.

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: “ Розробка методології графічного супроводу процесу розробки ПЗ на основі класифікованих ресурсів GIT ”

Обсяг пояснювальної записки: 83 аркуші.

Дата закінчення роботи: 11 червня 2025 р.

Підпис студента _____