ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка інтелектуальної системи для моніторингу внеску розробників у програмні проєкти»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Владислав БЕРО
(підпис)

Виконав: здобувач вищої освіти групи ПД-41

Владислав БЕРО

Керівник: Володимир САДОВЕНКО
к.ф.-м.н., доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Беро Владиславу Федоровичу

1. Тема кваліфікаційної роботи: «Розробка інтелектуальної системи для моніторингу внеску розробників у програмні проекти»
керівник кваліфікаційної роботи д.т.н., професор Володимир САДОВЕНКО, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «27» лютого 2025 р. № 36.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки на мові Python, опис роботи систем моніторингу внеску розробників у спільні проекти, документація фреймворків PyGitHub та Git API.

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

1. Огляд системи аналітики для оцінки внеску розробників та аналіз існуючих аналогів, визначення вимог до програмного забезпечення.
2. Огляд та аналіз засобів реалізації системи аналітики для оцінки внеску розробників.
3. Проектування архітектури системи аналітики для оцінки внеску розробників.

4. Програмна реалізація системи для оцінки внеску розробників.

5. Перелік графічного матеріалу: *презентація*
    1. Аналіз аналогів.
    2. Вимоги до програмного забезпечення.
    3. Програмні засоби реалізації.
    4. Діаграма розгортання.
    5. Діаграма варіантів використання
    6. Діаграма класів
    7. Екранні форми.
    8. Демонстрація роботи застосунку
    9. Апробація результатів дослідження

6. Дата видачі завдання «28» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	28.02.-06.03.2025	
2	Аналіз та дослідження існуючих аналогів	07.03-12.03.2025	
3	Огляд та аналіз систем аналітики внеску розробників	13.03-15.03.2025	
4	Огляд та аналіз засобів реалізації систем аналітики внеску розробників	16.03-21.03.2025	
5	Проектування архітектури системи аналітики внеску розробників	22.03-03.04.2025	
6	Програмна реалізація та тестування системи аналітики внеску розробників	04.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-05.05.2025	
8	Розробка демонстраційних матеріалів	06.05-12.05.2025	
9	Попередній захист роботи	13.05-31.05.2025	

Здобувач вищої освіти _____ Владислав БЕРО
(підпис)

Керівник
кваліфікаційної роботи _____ Володимир САДОВЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 57 стор., 1 табл., 59 рис., 20 джерел.

Мета роботи – підвищення ефективності моніторингу та аналітики за внеском кожного розробника у розробку спільного проекту.

Об'єкт дослідження – процес контролю та організації внеску кожного розробника у розробку спільного проекту.

Предмет дослідження – процес збору та аналізу даних про активність розробників у системах контролю версій для оцінки їхнього індивідуального внеску в командну розробку програмного забезпечення.

Короткий зміст роботи: У роботі проаналізовано предметну галузь оцінки ефективності командної розробки програмного забезпечення, а також досліджено інструменти, що використовуються для моніторингу активності розробників у Git-системах. Проведено аналіз існуючих рішень, таких як GitHub Insights, GitLab Analytics, Gource тощо, виявлено їхні переваги та недоліки. На основі отриманих результатів розроблено десктопну аналітичну систему Kanter, яка підключається до репозиторіїв, здійснює збір статистики про внесок кожного розробника, будує графіки активності, визначає періоди неактивності та дозволяє формувати PDF-звіти. Програмну реалізацію виконано з використанням мови Python, бібліотек PyGitHub, Pandas, Matplotlib, Seaborn, а також графічного інтерфейсу на базі Tkinter. Сфера використання системи — технічне управління проєктами, оцінка ефективності команди з боку тім-лідів, проджект-менеджерів та аналітиків..

Сферою використання тім-лідами, проджект-менеджерами, технічними керівниками та аналітиками з метою оцінки активності розробників у спільних проєктах.

КЛЮЧОВІ СЛОВА: GIT, PYTHON, KANTER, API, СИСТЕМА

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 8
ВСТУП 9
1 АНАЛІЗ СИСТЕМ АНАЛІТИКИ ДЛЯ ОЦІНКИ ВНЕСКУ РОЗРОБНИКІВ 11
1.1 ПЗ для аналізу активності розробників 11
1.2 Специфіка оцінки ефективності командної розробки 14
1.3 Цільова аудиторія 14
1.4 Дослідження існуючих аналогів 16
1.5 Визначення вимог до застосунку 25
2 ЗАСОБИ РЕАЛІЗАЦІЇ 26
2.1 Середовище розробки 26
2.2 Мови програмування 26
2.3 Фреймворки 27
2.4 Системи контролю версій 29
2.5 Інструменти проектування інтерфейсу 30
3 ПРОЄКТУВАННЯ 32
3.1 Проектування архітектури застосунку 32
3.2 Моделювання варіантів використання 34
3.3 Проектування структури класів 35
3.4 Проектування модульної структури (архітектури пакетів) 35
3.5 Моделювання станів об’єктів 36
4 РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ 38
4.1 Архітектура інтерфейсу системи “Kanter” 39
4.2 Архітектура логіки системи “Kanter” 55
4.3 Головний модуль системи “Kanter” 67
ВИСНОВКИ 68
ПЕРЕЛІК ПОСИЛАНЬ 70
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 72
ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ 79

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

PDF – Portable Document Format

IDE – Integrated Development Environment

API – Application Programming Interface

Git – розподілена система керування версіями

ВСТУП

Актуальність: у сучасній розробці програмного забезпечення командна робота є ключовим фактором успішного впровадження ІТ-проєктів. Зі зростанням складності програмних систем та збільшенням кількості учасників розробки постає нагальна потреба в ефективному моніторингу індивідуального внеску кожного розробника. Це дозволяє не лише об'єктивно оцінювати ефективність працівників, а й своєчасно виявляти неактивних учасників, аналізувати динаміку змін у коді та приймати обґрунтовані управлінські рішення З розвитком інструментів контролю версій, з’явилась можливість автоматизовано відстежувати активність учасників проєкту. Проте більшість доступних засобів аналітики мають обмежену функціональність або орієнтовані лише на великі корпорації, тоді як малі команди та стартапи залишаються без простих і зручних рішень.

Саме тому актуальним є створення ПЗ, що дозволяє зручно та наочно візуалізувати активність розробників, порівнювати внесок за періодами, отримувати статистику з різних Git-сервісів, а також виявляти потенційні проблеми у взаємодії всередині команди. Така система сприяє підвищенню прозорості процесу розробки, ефективнішому управлінню командою та покращенню якості програмного продукту.

Об’єктом дослідження є процес контролю та організації внеску кожного розробника у розробку спільного програмного проекту.

Предметом дослідження є програмне забезпечення для збору, обробки та візуалізації статистики активності розробників на основі даних із систем контролю версій.

Метою роботи є підвищення ефективності моніторингу внеску розробників у командну розробку програмного забезпечення шляхом створення десктопного застосунку з використанням мови програмування Python та бібліотек Tkinter, PyGithub, Pandas, Matplotlib, Seaborn.

Методи дослідження: першим етапом було проведено аналіз існуючих програмних рішень для аналітики активності в командах розробників, таких як GitHub Insights, GitLab Analytics, Gource та інші, з метою виявлення їх переваг, недоліків і обмежень. На основі цього аналізу були сформульовані функціональні та нефункціональні вимоги до власної системи.

Наукова новизна роботи забезпечує: візуалізацію активності розробників, порівняння внеску за певний період, автоматичне виявлення неактивних учасників команди, відстеження змін у вихідному коді, інтеграцію з кількома.

Практична значущість результатів полягає в можливості використання реалізованої системи менеджерами проєктів, тім-лідами та технічними керівниками для оперативної оцінки ефективності розробників, прийняття управлінських рішень на основі фактичних даних, а також для підвищення прозорості та дисципліни в командній розробці.

1 АНАЛІЗ СИСТЕМ АНАЛІТИКИ ДЛЯ ОЦІНКИ ВНЕСКУ РОЗРОБНИКІВ

1.1 ПЗ для аналізу активності розробників

Програмне забезпечення для аналізу активності розробників — це спеціалізовані системи, які призначені для збору, обробки та візуалізації даних про внесок кожного учасника у командну розробку. Такі системи зазвичай інтегруються з платформами контролю версій (Git) і дозволяють проводити детальний аналіз активності на основі комітів, змін у коді, часу роботи, та інших параметрів. Головна мета таких систем — підвищити прозорість у роботі команди, спростити управління проєктом і надати керівникам об'єктивні дані для ухвалення рішень.

Основні компоненти системи аналітики активності розробників включають:

– Збір даних: отримання інформації про коміти, гілки, пул-реквести та інші дії розробників через API сервісів контролю версій.

– Обробка даних: агрегація та фільтрація зібраної інформації для подальшого аналізу.

– Візуалізація: побудова графіків, діаграм, хіт-мапів, що ілюструють внесок кожного учасника у розвиток проєкту.

– Експорт звітів: можливість збереження аналітичних даних у вигляді PDF-документів для презентацій чи внутрішнього використання.

– Фільтрація за періодом: вибір часових інтервалів для порівняння активності або оцінки прогресу проєкту.

Основні цілі ПЗ цього типу:

– оцінка ефективності окремих розробників;
– визначення найактивніших і найменш активних учасників;
– виявлення тенденцій у темпах розробки;
– створення прозорої системи контролю внеску;
– спрощення управлінських рішень;
– формування об’єктивної звітності.

Оцінимо переваги та недоліки системи.

Переваги:

– Прозорість процесу розробки: аналітичне ПЗ забезпечує чітке уявлення про внесок кожного члена команди у проект. Керівники можуть легко відслідковувати активність розробників, бачити обсяг змін у коді, частоту комітів, участь у злиттях гілок тощо. Це дозволяє виявити як лідерів команди, так і учасників, які можуть потребувати додаткової підтримки або мотивації.

– Прийняття управлінських рішень на основі об'єктивних даних: завдяки візуалізації активності у вигляді графіків і діаграм, керівники можуть оперативно реагувати на зниження темпів роботи, оптимізувати навантаження між учасниками, планувати дедлайни та обсяг задач більш ефективно.

– Автоматизація звітності: у традиційних умовах звітність про виконану роботу формується вручну, що забирає час і може бути суб’єктивною. Система аналітики дозволяє генерувати PDF-звіти автоматично на основі даних з репозиторіїв, що підвищує точність та знижує ризик маніпуляцій чи помилок.

– Мотиваційний ефект для розробників: усвідомлення того, що їх активність фіксується та відображається у статистиці, спонукає учасників команди до більш відповідального ставлення до задач і підвищує рівень самодисципліни.

– Гнучкість та масштабованість: система дозволяє фільтрувати дані за періодом, окремим розробником, що робить їх придатними як для невеликих команд, так і для великих проєктів з десятками учасників.

– Підтримка об’єктивної оцінки результатів: такі системи мінімізують суб’єктивність при оцінюванні ефективності команди, оскільки ґрунтуються на фактичних даних. Це може бути корисним під час проведення регулярних оглядів продуктивності, преміювання чи внутрішнього аудиту.

Недоліки:

– Обмеженість у вимірюванні якості: попри наявність великого обсягу статистичних даних, система не здатна повноцінно оцінити якість коду, складність реалізованих рішень чи архітектурні покращення.

– Високий ризик неправильної інтерпретації: без достатнього технічного розуміння аналітичних показників керівники можуть робити хибні висновки.

– Можливе створення тиску на команду: якщо аналітика використовується як головний критерій оцінки праці, це може викликати стрес, конкуренцію між колегами або спонукати до штучного завищення активності. Це негативно впливає на командну динаміку.

– Залежність від зовнішніх сервісів: система аналітики “Kanter” працюють за допомогою Git API, який має обмеження на кількість запитів, зміни в структурі даних. Це може ускладнити безперервну роботу ПЗ.

1.2 Специфіка оцінки ефективності командної розробки

Процес оцінки ефективності командної розробки програмного забезпечення має низку особливостей, які відрізняють його від оцінювання індивідуальної діяльності або виконання стандартних бізнес-процесів. Основна складність полягає в колективному характері роботи, великій кількості змінних та впливі людського фактору. Розглянуто основні специфічні риси.

Використання систем контролю версій:

– Командна розробка зазвичай ведеться з використанням систем контролю версій Git, які дозволяють відстежувати кожну зміну в коді.

– У Git-платформах зберігається вся історія комітів, авторство змін, повідомлення до комітів, що є джерелом цінної аналітичної інформації.

– Можливість аналізу даних дозволяє будувати об’єктивні метрики продуктивності.

Наявність ролей у команді:

– Команди розробників зазвичай мають розподіл ролей: бекенд, фронтенд, тестувальники, девопс, архітектори, тім-ліди тощо.

– Внесок учасників не завжди проявляється однаково — тестувальник може не комітити код, але відігравати важливу роль у якості продукту.

– Оцінка ефективності повинна враховувати рольову специфіку та розподіл відповідальностей.

Нерівномірність навантаження протягом спринтів:

– У проектах за Agile-методологіями навантаження може змінюватися від спринту до спринту.

– Деякі учасники більше залучені на старті (планування, архітектура), інші — під кінець (тестування, інтеграція). Це потрібно враховувати при інтерпретації показників активності, щоб уникнути хибних висновків.

Залежність від командної взаємодії:

– Ефективність розробника не завжди можна оцінити лише за кількістю комітів або строками задач.

– Якість коду, комунікація, участь у код-рев'ю та здатність працювати в команді є критичними, але менш формалізованими аспектами.

– Тому важливо поєднувати кількісні метрики з якісною оцінкою.

Наявність зовнішніх впливів:

– На ефективність команди можуть впливати фактори, що не фіксуються у системі контролю версій — відсутність інтернету, технічні збої, хвороби членів команди, конфлікти.

Необхідність регулярного моніторингу:

– Оцінка ефективності повинна бути динамічною — важливо не тільки оцінити підсумковий внесок, а й виявляти зниження або зростання активності у процесі.

– Системи аналітики дозволяють створювати дашборди з оновленням у реальному часі, що сприяє гнучкому управлінню проєктом.

1.3 Цільова аудиторія

Інтелектуальна система для моніторингу внеску розробників у програмні проєкти орієнтована на професійну аудиторію, яка займається організацією,

аналізом і покращенням командної роботи у сфері розробки програмного забезпечення.

До основних категорій користувачів належать:

– Тім-ліди (керівники команд розробників) – фахівці, які відповідають за ефективність роботи команди, координацію задач, розподіл навантаження та виявлення «вузьких місць». Їхня мета — отримання аналітики щодо активності кожного учасника команди для підвищення продуктивності та своєчасного прийняття управлінських рішень.

– Проджект-менеджери (керівники проєктів) – спеціалісти, що відповідають за виконання проєктів у визначені терміни з дотриманням бюджету та якості. Для них важливо бачити загальну динаміку роботи команди, прогрес у виконанні задач, визначення активних і пасивних періодів у розробці.

– Технічні директори та керівники IT-відділів – управлінці, що оцінюють ефективність команд на рівні організації. Їхня мета — порівняння результатів роботи різних команд, визначення найрезультативніших учасників, планування кадрових рішень і розподілу ресурсів.

– Аналітики продуктивності – фахівці, які займаються оцінкою ефективності праці, побудовою звітності, аналізом динаміки проєктів. Вони потребують достовірної та структурованої інформації з Git-систем для формування звітів, графіків і діаграм.

– HR-фахівці в IT-сфері – спеціалісти, що здійснюють моніторинг професійного розвитку співробітників, формують плани навчання, проводять оцінювання персоналу. Вони можуть використовувати систему для аналізу активності розробників у проєктах.

– Самостійні розробники та фрилансери – індивідуальні програмісти, які хочуть отримати об’єктивну картину власної активності, структурувати робочий час і виявити свої сильні та слабкі сторони для подальшого саморозвитку.

Система є універсальним інструментом для всіх, хто зацікавлений в об'єктивному аналізі внеску в розробку програмного забезпечення, незалежно від масштабу команди чи рівня відповідальності користувача.

1.4 Дослідження існуючих аналогів

У сучасних умовах інтенсивної командної розробки програмного забезпечення особливо важливою стає задача об’єктивної оцінки внеску окремих учасників у загальний результат. Для вирішення цього завдання існує низка спеціалізованих інструментів, орієнтованих на збір, візуалізацію та аналіз даних із систем контролю версій, таких як GitHub, GitLab та інші. У цьому розділі розглянуто найпоширеніші програмні засоби, які можуть бути використані для аналізу активності розробників, виявлення неактивних учасників команди, а також порівняння продуктивності у різні періоди часу.

Розгляд включає аналіз таких систем:

- GitHub Insights;
- GitLab Analytics;
- Gource;
- Open Source Contributions Dashboard.

1.4.1 GitHub Insights

GitHub Insights — це вбудований інструмент для візуалізації статистики активності розробників, орієнтований на команди, що працюють у GitHub, зокрема в межах GitHub Enterprise. Інструмент покликаний допомогти керівникам проєктів оцінювати залучення учасників команди на основі графіків активності, кількості комітів, пул-реквестів та інших показників.

Основні функції GitHub Insights включають:

– Графіки активності: візуалізація комітів за певний період.

– Порівняння періодів: можливість порівнювати активність за різні проміжки часу.

– Огляд по розробниках: статистика участі кожного члена команди в розробці.

– Відстеження продуктивності: аналіз змін у темпі роботи команди.

– Інтеграція з Git репозиторіями: робота з приватними репозиторіями

організацій.

– Автоматичне оновлення даних: зручна синхронізація з репозиторіями в реальному часі.

– Веб-інтерфейс: доступ через браузер без потреби у встановленні додаткових програм.

Переваги:

– Зручна візуалізація даних: зрозумілі графіки й діаграми дозволяють швидко оцінити активність команди.

– Підтримка командної аналітики: менеджери можуть аналізувати участь кожного розробника у спільному проєкті.

– Порівняння періодів: функціонал дозволяє виявити зміни в темпі розробки або ефективності роботи.

Недоліки:

– Відсутність аналізу на рівні рядків коду: немає можливості побачити кількість доданих або видалених рядків, що ускладнює оцінку обсягу роботи.

– Обмежена інформативність щодо неактивних розробників: не передбачено автоматичних сповіщень про зниження активності.

– Вузька інтеграція: працює виключно з GitHub, без підтримки інших популярних сервісів (GitLab, Bitbucket тощо).

– Веб-орієнтованість: відсутній десктопний застосунок, що обмежує офлайн-доступ.

GitHub Insights є корисним інструментом для базового аналізу активності в межах GitHub-екосистеми, однак його функціональність може бути недостатньою для глибокого оцінювання внеску розробників у проєктах з високими вимогами до аналітики. Приклад аналітики GitHub Insights наведено на рисунку 1.1.

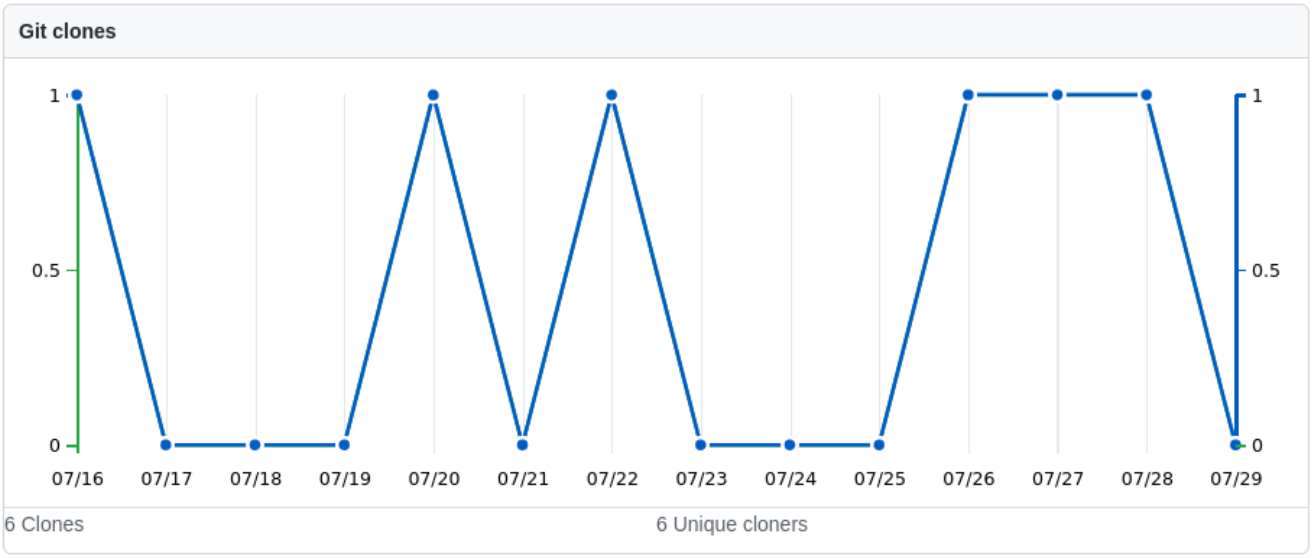


Рис. 1.1 Приклад графіку у GitHub Insights

1.4.2 GitLab Analytics

GitLab Analytics — це аналітична панель, інтегрована безпосередньо в платформу GitLab, яка призначена для моніторингу активності розробників і команд у процесі програмної розробки. Інструмент орієнтований на надання менеджерам та тім-лідам даних щодо ефективності роботи команди через аналіз метрик з репозиторіїв GitLab.

Основні функції GitLab Analytics включають:

– Моніторинг активності: перегляд кількості комітів, злиттів, змін у коді та відкритих/закритих задач.

– Візуалізація внеску: графічне представлення участі кожного учасника проєкту.

– Історія змін у файлах: відстеження змін у конкретних модулях або компонентах системи.

– Порівняння за періодами: аналіз активності в динаміці — по днях, тижнях чи спринтах.

– Метрики ефективності: оцінка участі в рев'ю, швидкості реагування, частоти комітів.

– Групова аналітика: агреговані дані за командами або відділами.

– Вбудованість у GitLab: без потреби сторонніх додатків або інтеграцій.

– Доступ через вебінтерфейс: усі функції доступні прямо через браузер.

– Інтеграція з CI/CD: можливість аналізувати зв’язок між активністю розробника та результатами автоматизованого тестування.

Переваги:

– Комплексна візуалізація активності: завдяки графікам та діаграмам користувач може побачити як кількісні, так і якісні зміни, зокрема роботу з критичними частинами коду.

– Глибокий аналіз участі в команді: система дозволяє аналізувати не лише коміти, а й участь у рев’ю, взаємодію з іншими членами команди.

– Оцінка динаміки роботи: можливість відстежувати зміни у продуктивності за обрані часові періоди.

Недоліки:

– Обмежена гнучкість налаштувань: інструмент пропонує фіксований набір метрик, без можливості створення власних показників або кастомних звітів.

– Відсутність автоматичних сповіщень: немає вбудованого механізму для повідомлень про зниження активності окремих розробників.

– Залежність від веб-інтерфейсу: GitLab Analytics доступний лише онлайн, не пропонує окремого настільного застосунку для автономної роботи.

GitLab Analytics є потужним засобом для базової оцінки командної ефективності в екосистемі GitLab, однак через обмежену гнучкість та відсутність автоматичних інтелектуальних функцій може не повністю відповідати потребам глибокої аналітики в середовищах із високою варіативністю проєктів.Приклад аналітики GitLab Analytics наведено на рисунку 1.2.

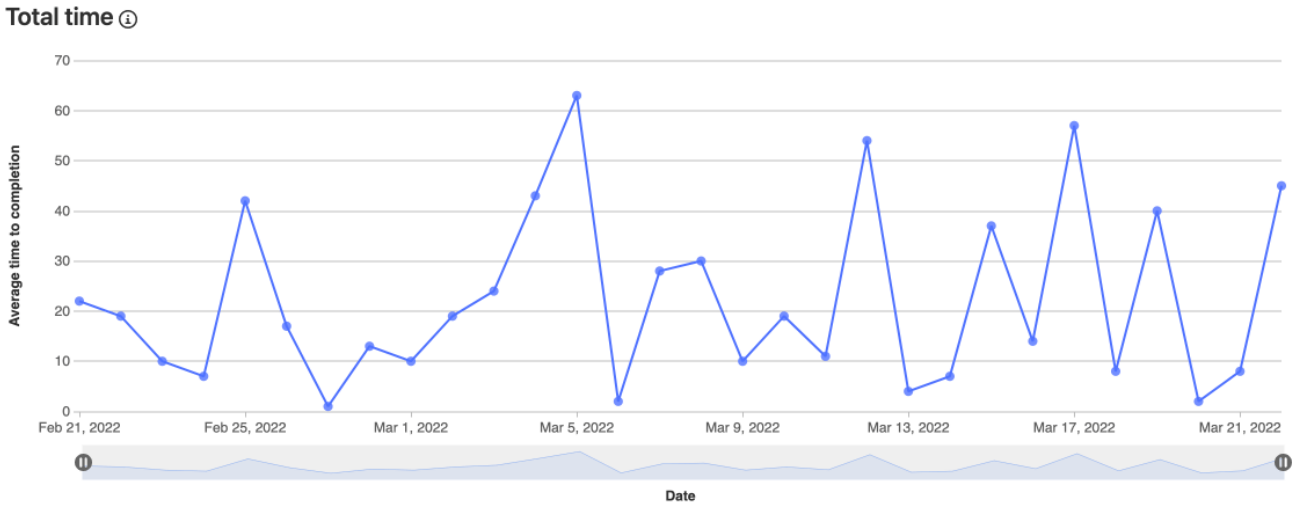


Рис. 1.2 Приклад графіку у GitLab Analytics

1.4.3 Gource

Gource — це програмний засіб для створення анімованої візуалізації історії змін у Git-репозиторіях. Інструмент перетворює дані про коміти, зміни у файлах та активність розробників на інтерактивну графіку у вигляді дерева, де кожен вузол відповідає окремому файлу або учаснику проєкту. Gource активно використовується для демонстрацій, презентацій та наочного представлення командної роботи у проєктах з відкритим кодом.

Основні функції Gource включають:

– Анімована візуалізація: графічне представлення структури репозиторію та змін у режимі реального часу.

– Відображення авторів: кожен розробник показаний у вигляді об'єкта, що взаємодіє з файлами, які він редагував.

– Хронологічне відтворення історії: можливість програмування змін за період існування проєкту.

– Підтримка різних VCS: Gource працює не лише з Git, а й з іншими системами контролю версій, такими як Mercurial чи Bazaar.

– Вивід у відеоформат: можливість зберігати візуалізацію у вигляді відео.

– Кросплатформеність: доступність для Windows, macOS та Linux.

Переваги:

– Наочність візуалізації: зручна графічна подача історії змін дозволяє швидко оцінити рівень активності у проєкті, що особливо корисно для великих команд.

– Простота використання: інтерфейс з мінімальними налаштуваннями дозволяє легко запускати візуалізацію навіть новачкам.

– Креативне застосування: часто використовується у промо матеріалах, дослідженнях та освітніх проєктах.

Недоліки:

– Відсутність чисельної аналітики: Gource не надає текстових звітів або метрик.

– Обмежений функціонал для аналітики: інструмент не дозволяє проводити глибокий аналіз внеску окремих учасників, періодів неактивності або змін на рівні коду.

– Вимога локального репозиторію: для роботи з GitHub чи GitLab необхідне попереднє клонування репозиторію, оскільки прямої інтеграції з API не передбачено.

– Нестача інтерактивності: візуалізація статична у сенсі користувацької взаємодії — неможливо фільтрувати дані або аналізувати конкретні фрагменти коду.

Інструмент є десктопним, але при цьому не інтегрується з GitHub чи GitLab API безпосередньо, вимагаючи попереднього локального клонування репозиторіїв. Приклад структури Gource наведено на рисунку 1.3.

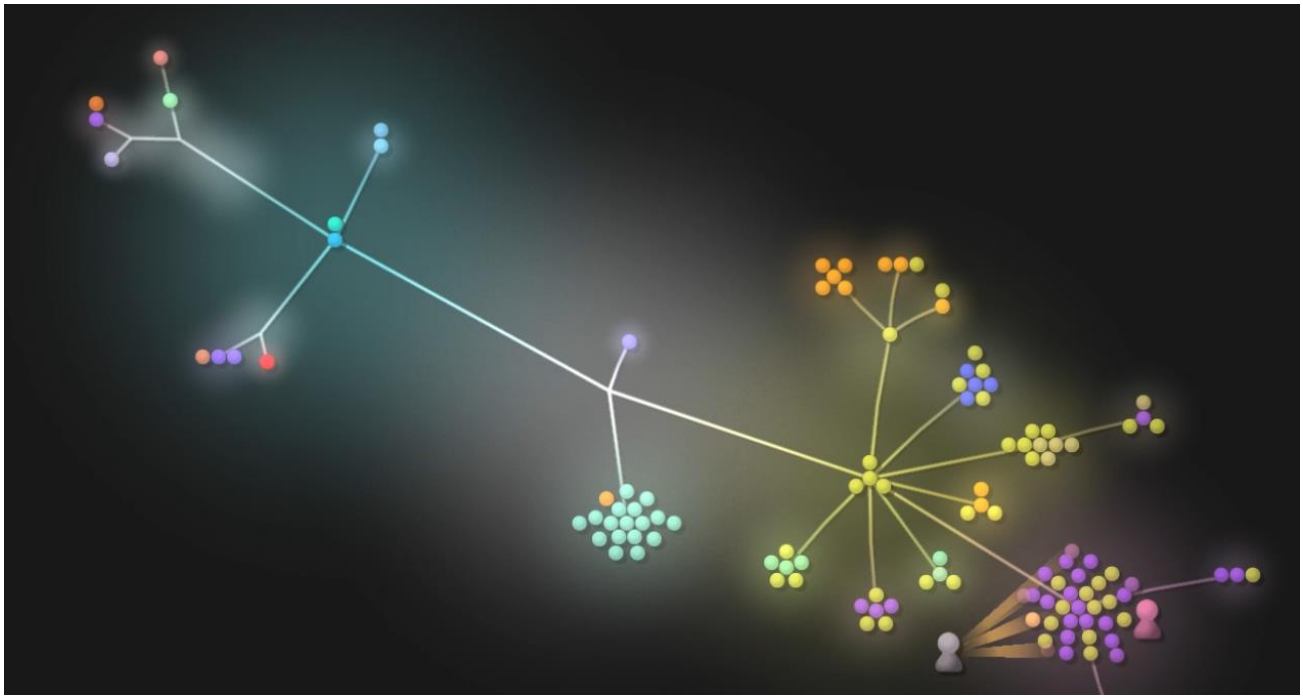


Рис. 1.3 Приклад структури проекту у Gource

1.4.4 Open Source Contributions Dashboard

Open Source Contributions Dashboard — це веб-орієнтований інструмент для збору та візуалізації активності розробників, що працюють над відкритими програмними проєктами. Застосунок вирішує проблему відсутності зручної аналітики внеску учасників у Git-репозиторії, надаючи керівникам команд або ментору спільнот зручні візуальні засоби для оцінки активності учасників.

Основні функції Open Source Contributions Dashboard включають:

– Перегляд статистики за репозиторіями: кількість комітів, пул-реквестів, активність за часом.

– Аналіз окремих учасників: відстеження індивідуального внеску в різні проєкти.

– Підтримка кількох платформ: збирання даних з різних Git-сервісів одночасно (GitHub, GitLab, Bitbucket).

– Інтерактивні графіки та діаграми: наочне представлення активності розробників.

– Фільтрація та сортування даних: зручна система вибору параметрів для перегляду.

– Можливість експорту звітів: формування підсумкової інформації у зручному форматі.

– Відкрита архітектура: можливість розширення функціональності згідно з потребами команди.

Переваги:

– Універсальність у підключенні сервісів: підтримка кількох Git-платформ одночасно дає змогу аналізувати розподілену діяльність у межах великої open source-екосистеми.

– Оцінка індивідуального внеску: система дозволяє легко порівнювати активність окремих учасників у різних проєктах.

– Корисність для менеджерів і координаторів: дає можливість об’єктивно оцінити залученість учасників у розробку, що важливо під час прийняття управлінських рішень.

Недоліки:

– Відсутність деталізації на рівні змін у файлах: система не підтримує аналіз змін на рівні конкретних рядків коду.

– Обмежена аналітика щодо динаміки внеску: немає можливості побудови детального порівняння активності учасника за різні періоди.

– Виключно веб-інтерфейс: відсутність офлайн-доступу може ускладнити використання системи в умовах обмеженого підключення до мережі.

– Немає системи сповіщень: користувачі не отримують повідомлень про зниження активності або інші критичні події, що знижує рівень інтерактивності.

Приклад аналітики Open Source Contributions Dashboard наведено на рисунку 1.4.

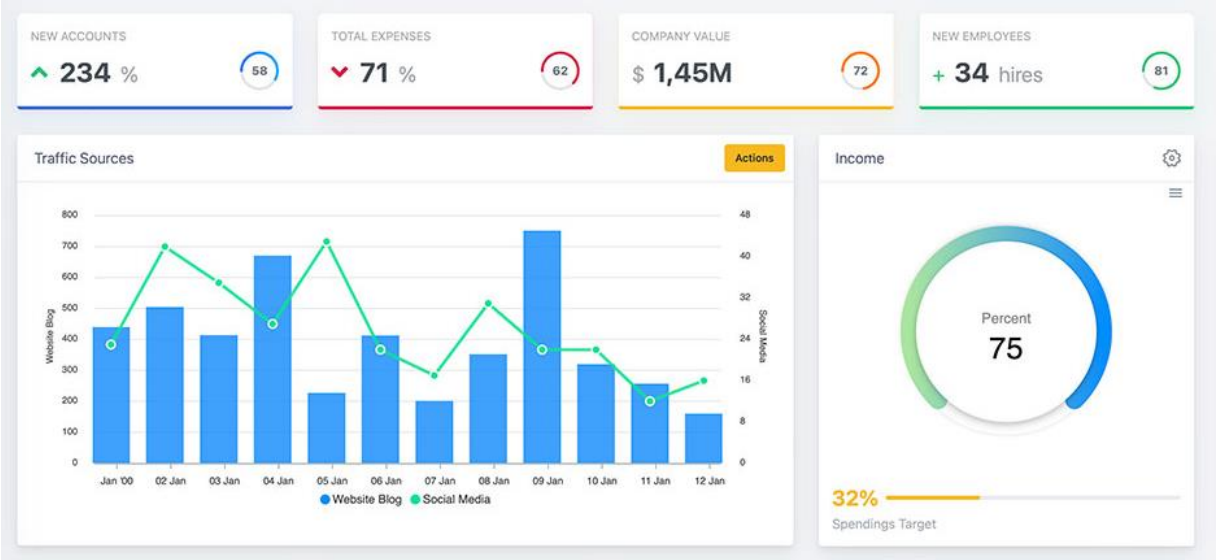


Рис. 1.4 Приклад аналітики Open Source Contributions Dashboard

Таблиця 1.1

Зведені результати аналізу характеристик систем моніторингу внеску розробників у проєкти

Показник	**GitHub Insights**	**GitLab Analytics**	**Gource**	**Open Source Contributions Dashboard**
Платформи	Web	Web	Windows, macOS, Linux	Web, Linux
Візуалізація активності розробників	+	+	+	+
Відстеження змін у файлах (рядки +/-)	+	+	-	+
Попередження про неактивних розробників	+	+	-	-
Порівняння активності за періодами	+	+	-	+

Продовження таблиці 1.1

Порівняння існуючих програм-аналогів

Показник	**GitHub Insights**	**GitLab Analytics**	**Gource**	**Open Source Contributions Dashboard**
Отримання даних з різних Git-сервісів	-	-	+	+
Десктопний додаток	-	-	+	-

1.5 Визначення вимог до застосунку

Аналіз особливостей процесу оцінки внеску в командну розробку дозволяє сформулювати вимоги до майбутньої програмної системи. Система повинна мати графічний інтерфейс, підтримувати підключення до репозиторіїв на Git-сервісах, автоматично збирати статистику по комітах, генерувати аналітичні звіти та надавати візуальні представлення даних.

Проаналізувавши специфіку розроблюваної системи Kanter, її цільову аудиторію (архітектори, проджект-менеджери, тім-ліди) та результати аналізу аналогічних програмних рішень, було сформульовано такі функціональні та нефункціональні вимоги до застосунку.

Функціональних вимог:

– Підключення до репозиторію через назву або токен, який вводить користувач (архітектор/проджект-менеджер).

– Отримання даних з репозиторію після натискання на кнопку “Оновити дані” та їх обробка з формуванням аналітики про діяльність кожного розробника.

– Автоматичне створення кнопок з доступом до графічних звітів по кожному учаснику команди.

– Збереження згенерованих звітів про активність проєкту на комп’ютер користувача за вказаним шляхом.

– Можливість вибору, чи зберігати дані про токен і репозиторії після завершення роботи системи.

Нефункціональні вимоги:

– Система працює на ОС Windows.

– Час створення аналітики не перевищує 15 секунд для команди розробників середнього розміру.

– Інтерфейс реагує на дії користувача менше ніж за 1 секунду.

– Підключення до Git-репозиторіїв реалізовано через інтеграцію з кількома Git-сервісами.

– Звіти зберігаються у форматі .pdf.

– Доступ до токенів здійснюється через захищене зберігання в .env файлі.

– У випадку помилок при запиті до репозиторію система повідомляє користувача з поясненням причин помилки.

– Інтерфейс підтримує українську мову.

– Система розроблена на мові програмування Python версії 3.11+.

– Інтерфейс побудований на основі бібліотеки Tkinter.

– Система надсилає повідомлення користувачеві, якщо виявлено розробників, які не виконували жодної роботи за останній тиждень, через вставне вікно.

Реалізація такої системи дозволяє автоматизувати та спростити процес оцінювання внеску кожного розробника, зменшити суб’єктивність при ухваленні управлінських рішень та підвищити прозорість командної роботи.

2 ЗАСОБИ РЕАЛІЗАЦІЇ

2.1 Середовище розробки

Інтегроване середовище розробки (IDE) – це комплексне програмне забезпечення, що надає програмістам різноманітні інструменти для розробки програмного коду. Інтегровані середовища розробки зазвичай включають текстовий редактор, інструменти для автоматизації збірки коду, дебагер та, часто, інтерфейс для системи контролю версій. Це допомагає розробникам ефективно писати, тестувати і відлагоджувати свої програми в одному середовищі.

2.1.1 PyCharm

Розробка системи здійснювалась у середовищі PyCharm — одній з найпопулярніших IDE для Python. інтегроване середовище розробки для мови програмування Python. PyCharm забезпечує розширену підтримку налагодження, автодоповнення коду, управління залежностями та зручний графічний інтерфейс для навігації проєктом. Система контролю версій Git використовувалась для керування кодовою базою та спільної роботи з репозиторіями. Git дозволяє зберігати історію змін, працювати з гілками, об’єднувати зміни та забезпечує репліковану модель зберігання коду, що робить його стандартом де-факто у сучасній розробці програмного забезпечення.

2.2 Мова програмування

Основу програмної реалізації становить мова Python та ряд допоміжних бібліотек і технологій, які забезпечують збір, обробку, візуалізацію та збереження аналітичних даних.

Python — інтерпретована об'єктно-орієнтована мова програмування високого рівня із суворою динамічною типізацією. Python має велику кількість бібліотек для

наукових, візуалізаційних та інженерних задач, що робить його зручним інструментом для швидкої побудови прототипів і стабільних програмних систем.

Python було обрано як основну мову програмування завдяки її універсальності, читабельності синтаксису та широкому набору бібліотек для обробки даних, побудови GUI, взаємодії з API та генерації візуалізацій. Завдяки активній спільноті та регулярним оновленням Python продовжує залишатися однією з найпопулярніших мов для розробки аналітичного програмного забезпечення. Використання Python 3.11+ забезпечило доступ до нових можливостей, покращеної продуктивності та сумісності з сучасними бібліотеками.

2.3 Фреймворки

У процесі розробки програмної системи “Kanter” для збору та аналізу внеску розробників у Git-проєкти було обґрунтовано вибір низки сучасних технологій і фреймворків. Підбір технологічного стеку здійснювався з урахуванням вимог до функціональності системи, зручності реалізації графічного інтерфейсу, обробки API-запитів та побудови візуальної аналітики.

2.3.1 PyGitHub

Бібліотека PyGitHub була використана для інтеграції з GitHub API. За допомогою цієї бібліотеки реалізовано ключову функціональність системи — збір даних про активність розробників у проєкті. PyGitHub підтримує автентифікацію через токен, дозволяючи працювати з приватними репозиторіями без порушення політики безпеки. Приклад використання PyGitHub наведено на рисунку 2.1.

```python
def connect(token, repository):
    connect_result = {'message': "З'єднання успішне!", 'success': False, 'repo': None}

    try:
        g = Github(f"{token}")
        repo = g.get_repo(f"{repository}")
        connect_result['success'] = True
        connect_result['repo'] = repo
    except Exception as e:
        connect_result['message'] = f"[ERROR] Невідома помилка: {str(e)}"

    return connect_result
```

Рис. 2.1 Приклад отримання даних за допомогою PyGitHub

2.3.2 .env

Конфіденційні дані, зокрема особисті токени доступу до GitHub API, зберігаються у захищеному вигляді за допомогою формату .env. Для цього використовується бібліотека python-dotenv, яка дозволяє витягувати змінні середовища з окремого файлу .env. Це підвищує безпеку зберігання приватних ключів, токенів і конфігурацій. Бібліотека python-dotenv дозволяє зберігати змінні середовища у спеціальному .env файлі. Такий підхід покращує безпеку системи та дозволяє зручно налаштовувати конфігурацію під час розгортання.

2.3.3 Seaborn

Бібліотека Seaborn використовується для побудови стильових статистичних графіків. Вона надає високорівневий API для створення привабливих і інформативних діаграм, таких як лінійні графіки, теплові карти, розподіли значень тощо. Завдяки вбудованим стилям і підтримці Pandas-структур, Seaborn дозволяє швидко створювати зрозумілі візуалізації аналітичних даних.

2.3.4 Matplotlib

Разом із Seaborn у системі використовується бібліотека Matplotlib, яка є потужним інструментом для побудови графіків, діаграм і візуальних елементів у середовищі GUI. Matplotlib дозволяє гнучко налаштовувати відображення осей, легенд, підписів і стилів, що є критично важливим при створенні PDF-звітів і

виведенні графіків у вікні програми. Приклад графіку створеного з використанням Matplotlib та Seaborn на рисунку 2.2.

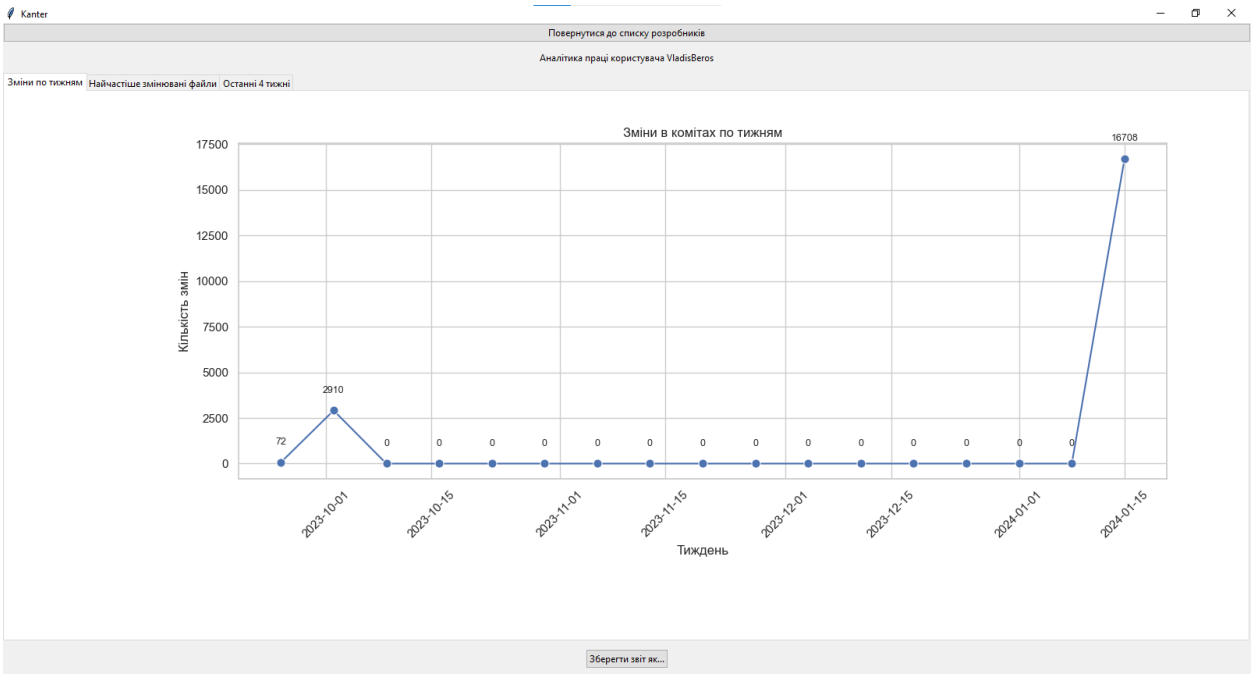


Рис. 2.2 Побудований графік за допомогою Matplotlib та Seaborn

2.3.5 Pandas

Pandas — бібліотека для обробки структурованих даних, яка надає високорівневі інструменти для фільтрації, сортування, агрегації та групування даних у форматі таблиць DataFrame. Pandas використовується для обробки табличних даних, зокрема збору, фільтрації, групування та агрегації інформації, отриманої з GitHub API. Завдяки DataFrame-структурам розробник може легко працювати з великими обсягами структурованих даних та готувати їх до візуалізації або експорту у звіти.

2.3.6 NumPy

NumPy — розширення мови Python, що додає підтримку великих багатовимірних масивів і матриць, разом з великою бібліотекою високорівневих математичних функцій для операцій з цими масивами. Вона забезпечує високу продуктивність при опрацюванні даних, що особливо важливо під час побудови

аналітичних графіків на основі великого обсягу інформації.

2.4 Системи контролю версій

GitHub — це веб-платформа для хостингу коду, яка базується на системі контролю версій Git. GitHub швидко став однією з найбільш впливових платформ у світі розробки програмного забезпечення, забезпечуючи інструменти для спільної роботи, контролю версій та коду розподіленого доступу.

Система контролю версій Git застосовувалася як під час розробки застосунку, так і в якості джерела даних для аналізу. Git дозволяє зберігати історію змін, відстежувати активність розробників, працювати з гілками та проводити злиття. Також система Kanter працює з віддаленими Git-сервісами, використовуючи Git API для збору статистики про внесок учасників проєкту.

2.5 Інструменти проектування інтерфейсу

За допомогою цієї бібліотеки реалізується автоматизоване збирання інформації про внесок окремих розробників у командні проєкти. Графічний інтерфейс програми створено засобами Tkinter — багатоплатформна графічна бібліотека інтерфейсів на основі засобів Tk, поширювана з відкритими вихідними текстами, написана Стіном Лумхольтом і Гвідо ван Россумом. Стандартна бібліотека Python для побудови GUI. Tkinter забезпечує базові інструменти для створення форм, кнопок, вкладок, полів введення та інших елементів інтерфейсу, що дозволяє користувачеві працювати з програмою у зручному віконному середовищі. Для кращої організації вікон було реалізовано вкладковий інтерфейс, що полегшує навігацію між модулями програми. Приклад екранних форм Tkinter наведено на рисунку 2.3.

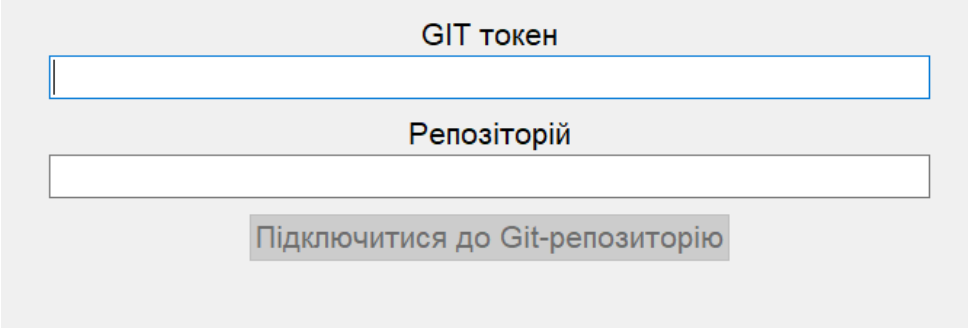


Рис. 2.3 Приклад віджету на Tkinter

3 ПРОЄКТУВАННЯ

3.1 Проєктування архітектури застосунку

Архітектура інтелектуальної системи “Kanter” реалізована за принципом автономного настільного клієнтського застосунку. Це дозволяє уникнути залежності від зовнішнього сервера та забезпечити повну обробку даних локально на машині користувача. Такий підхід є доцільним з огляду на потреби в інтерактивному аналізі, офлайн-доступності та підвищенні безпеки при обробці конфіденційних токенів доступу до Git API.

Застосунок “Kanter” реалізований мовою Python з використанням бібліотеки Tkinter для побудови графічного інтерфейсу користувача. Він підтримує роботу у віконному режимі з вкладками та інтерактивними елементами, що дозволяє користувачу здійснювати зручну навігацію між репозиторіями, переглядати статистику та експортувати аналітичні звіти.

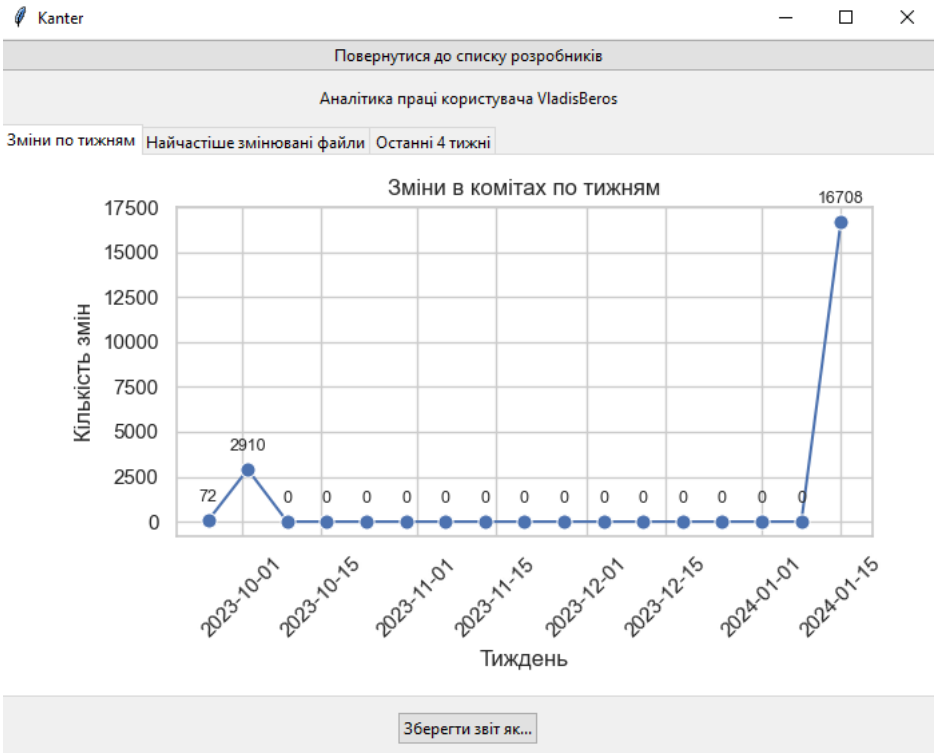


Рис. 3.1 Віджет Kanter

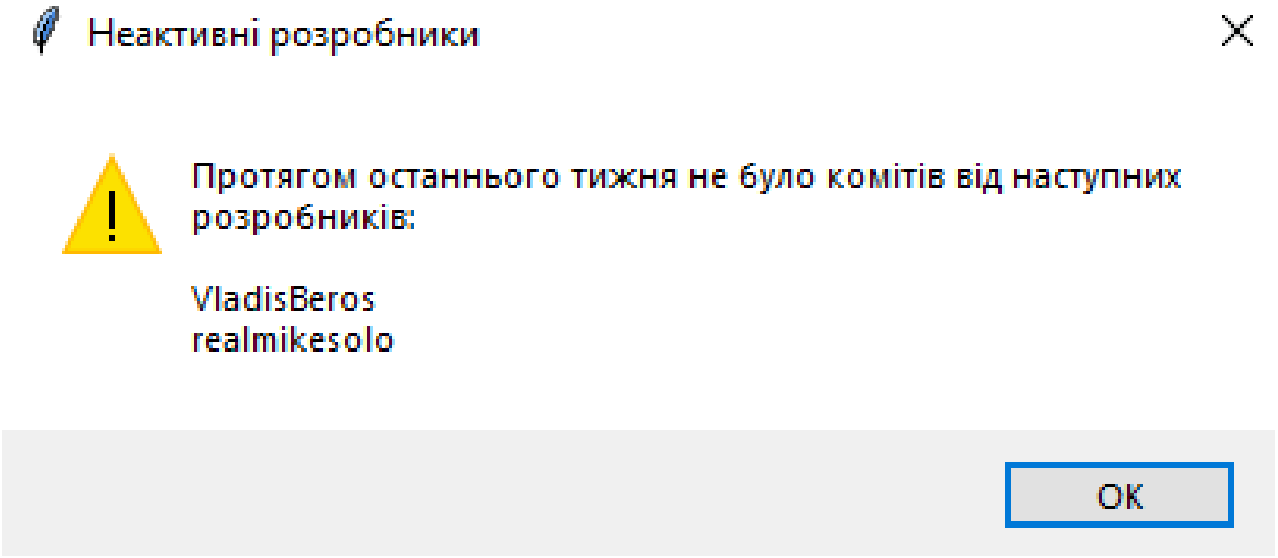


Рис. 3.2 Повідомлення у системі

Взаємодія з віддаленим сервером GitHub відбувається за допомогою офіційного API GitHub через бібліотеку PyGitHub. Користувач вводить особистий токен, який зберігається у .env файлі та завантажується при старті програми. Це забезпечує автентифікацію та доступ до як публічних, так і приватних репозиторіїв. Застосунок формує запити до GitHub API, отримує дані у форматі JSON, після чого виконується їхня обробка та фільтрація.

Аналітичний модуль програми використовує бібліотеки Pandas та NumPy для перетворення, фільтрації й агрегації даних. Наприклад, проводиться підрахунок кількості комітів по кожному розробнику, обсяг зміненого коду, частота активності в певні дні та інші метрики.

Для візуального представлення статистики використано бібліотеки Matplotlib та Seaborn, які дозволяють будувати графіки, діаграми, теплові карти та гістограми. Побудовані графіки інтегруються у GUI-додаток або експортуються до PDF-звітів для подальшого використання.

Таким чином, архітектура “Kanter” побудована як десктопний застосунок, який взаємодіє з віддаленими сервісами без потреби в окремому сервері чи веб-клієнті. Вся логіка обробки, візуалізації та управління даними виконується локально, що забезпечує високий рівень автономності, швидкодії та захисту даних користувача. Такий підхід ідеально підходить для індивідуального використання

тім-лідами або менеджерами, які бажають швидко оцінити ефективність роботи команди на основі активності в системах контролю версій.

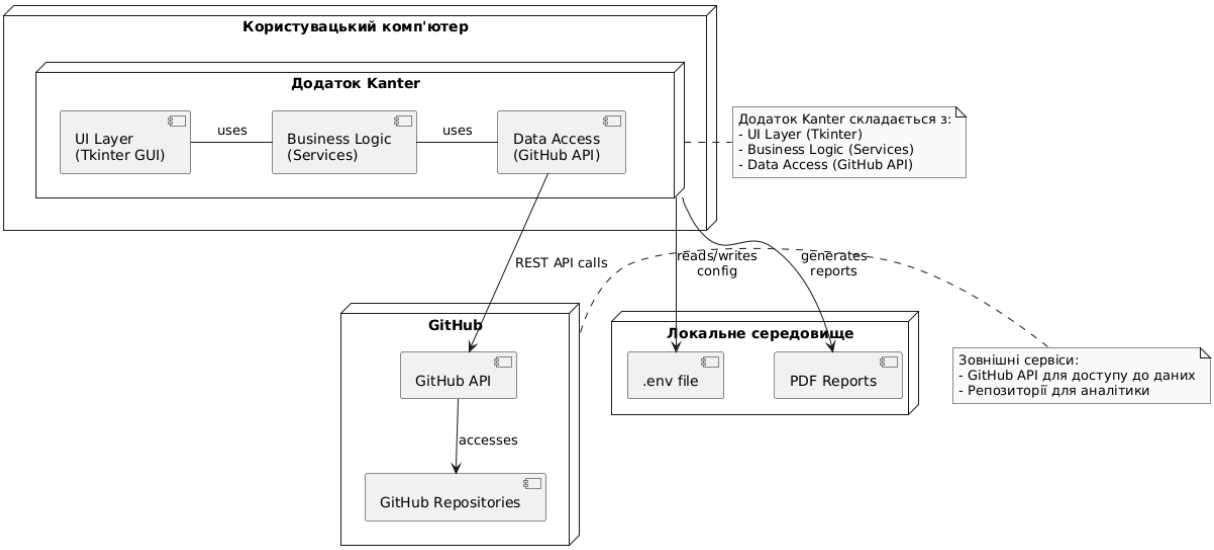


Рис. 3.3 Діаграма розгортання системи

3.2 Моделювання варіантів використання

У цьому підрозділі описується поведінка системи з точки зору взаємодії користувача з функціоналом. Супроводжується діаграмою варіантів використання, яка показує ролі і сценарії взаємодії.

Рис. 3.4 Діаграма варіантів використання системи

3.3 Проектування структури класів

Цей підрозділ присвячений об’єктно-орієнтованому поданню системи. Тут описується основна структура класів, зв’язки між ними, атрибути й методи. Ідеально для діаграми класів, що показує внутрішню реалізацію системи.

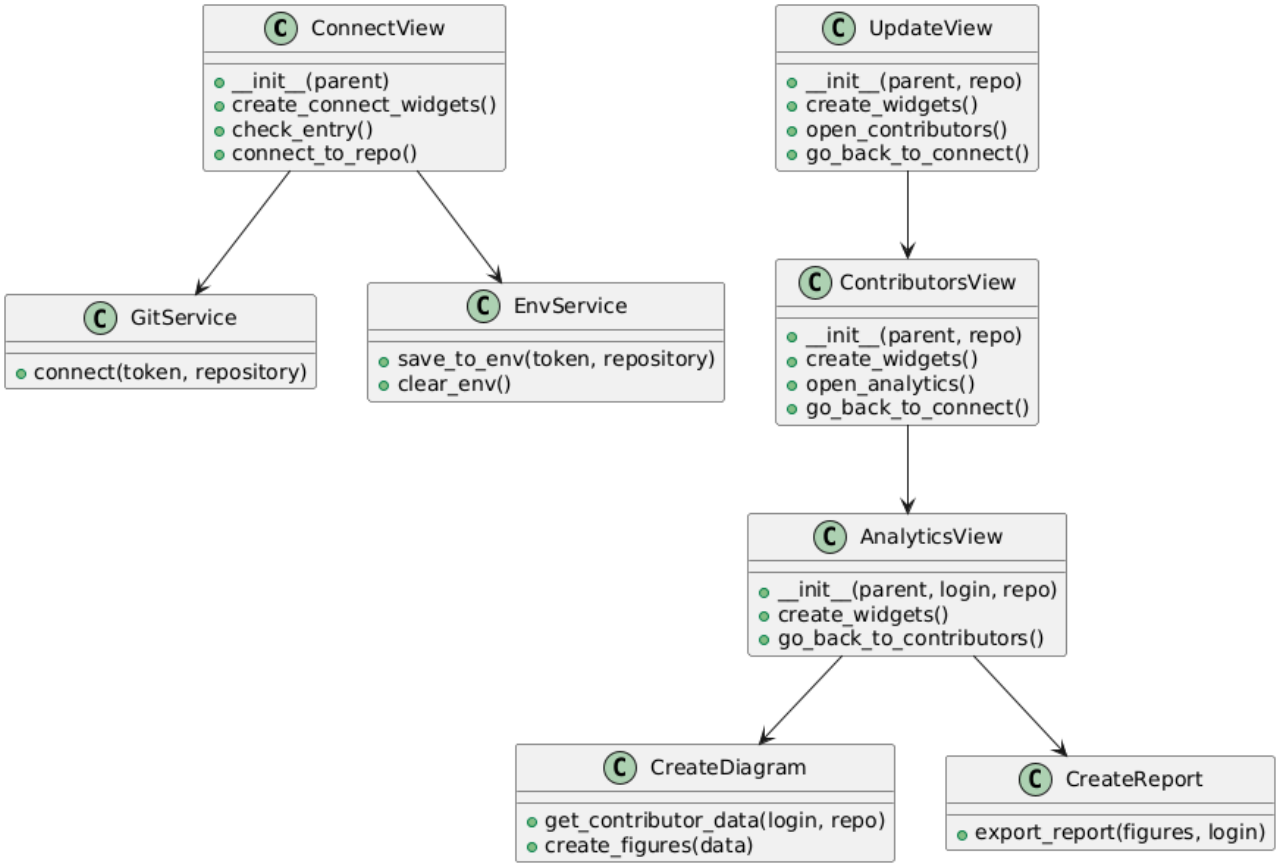


Рис. 3.5 Діаграма класів системи

3.4 Проектування модульної структури (архітектури пакетів)

Опис логічного розділення програми на окремі компоненти (пакети/модулі) згідно з принципами інкапсуляції, відповідальностей і повторного використання. Тут доречно додати діаграму пакетів, яка демонструє модулі, підсистеми та їхні залежності.

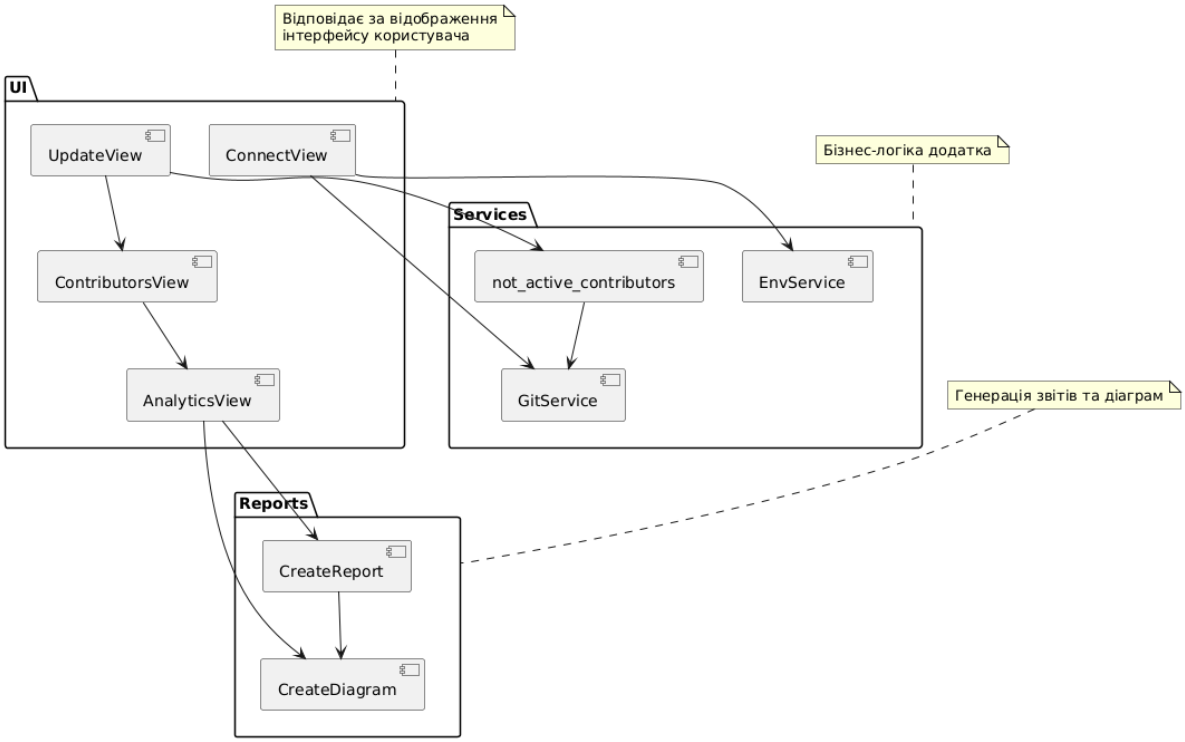


Рис. 3.6 Діаграма пакетів системи

3.5 Моделювання станів об’єктів

Опис зміни станів ключових об’єктів у процесі роботи системи.

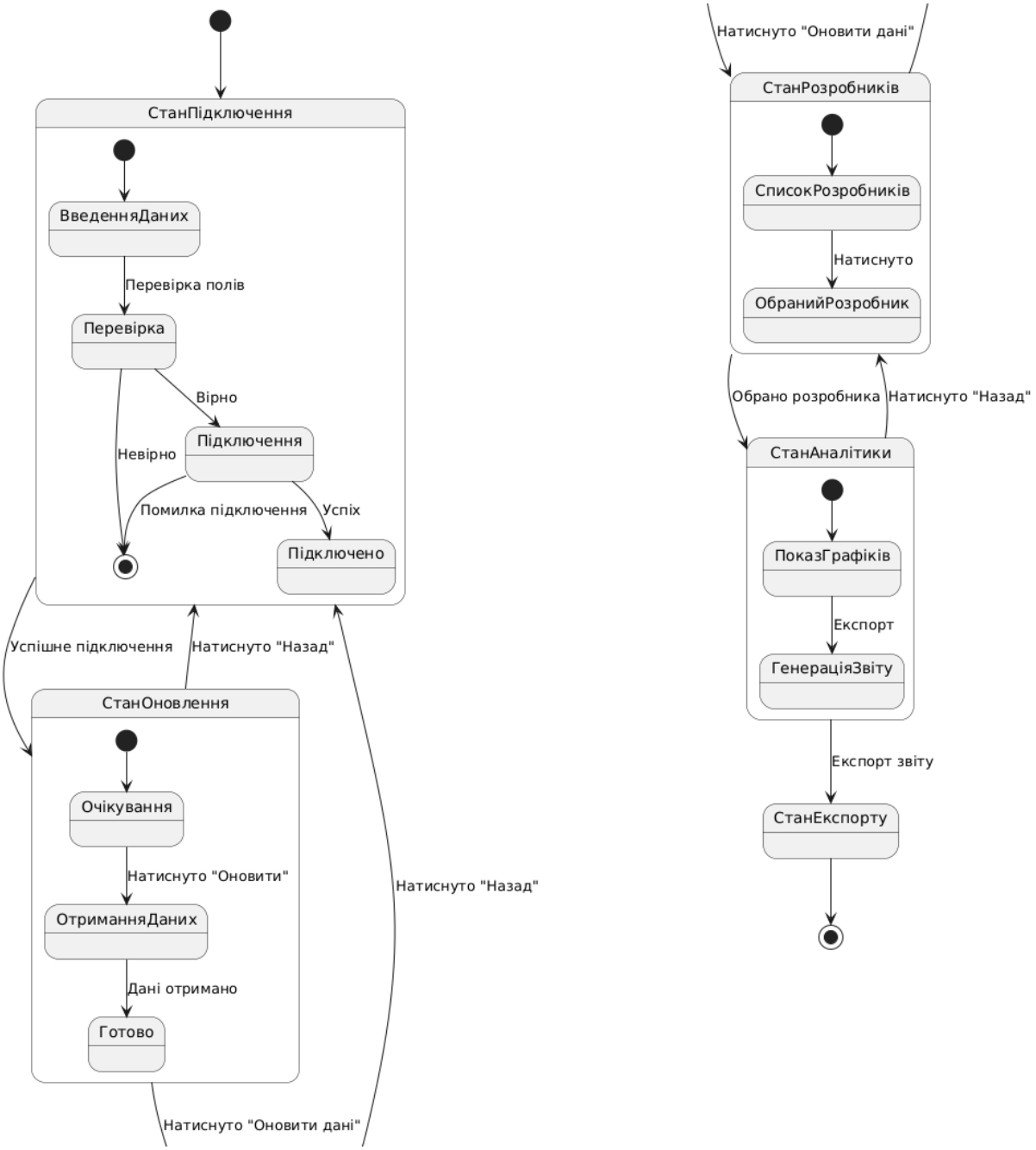


Рис. 3.7 Діаграма станів системи

4 РЕАЛІЗАЦІЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

Розроблена система Kanter реалізована мовою програмування Python з використанням графічної бібліотеки Tkinter. Архітектура побудована за принципами модульності, де кожен компонент відповідає за окрему функціональність: підключення до репозиторію, оновлення даних, перегляд активності розробників та аналітика з побудовою графіків. Що надає більшій читабельності коду для розробника у майбутньому або для інших розробників, які можут віправляти якісь функції, або додавати інші не ризикуючи зламати код програми.

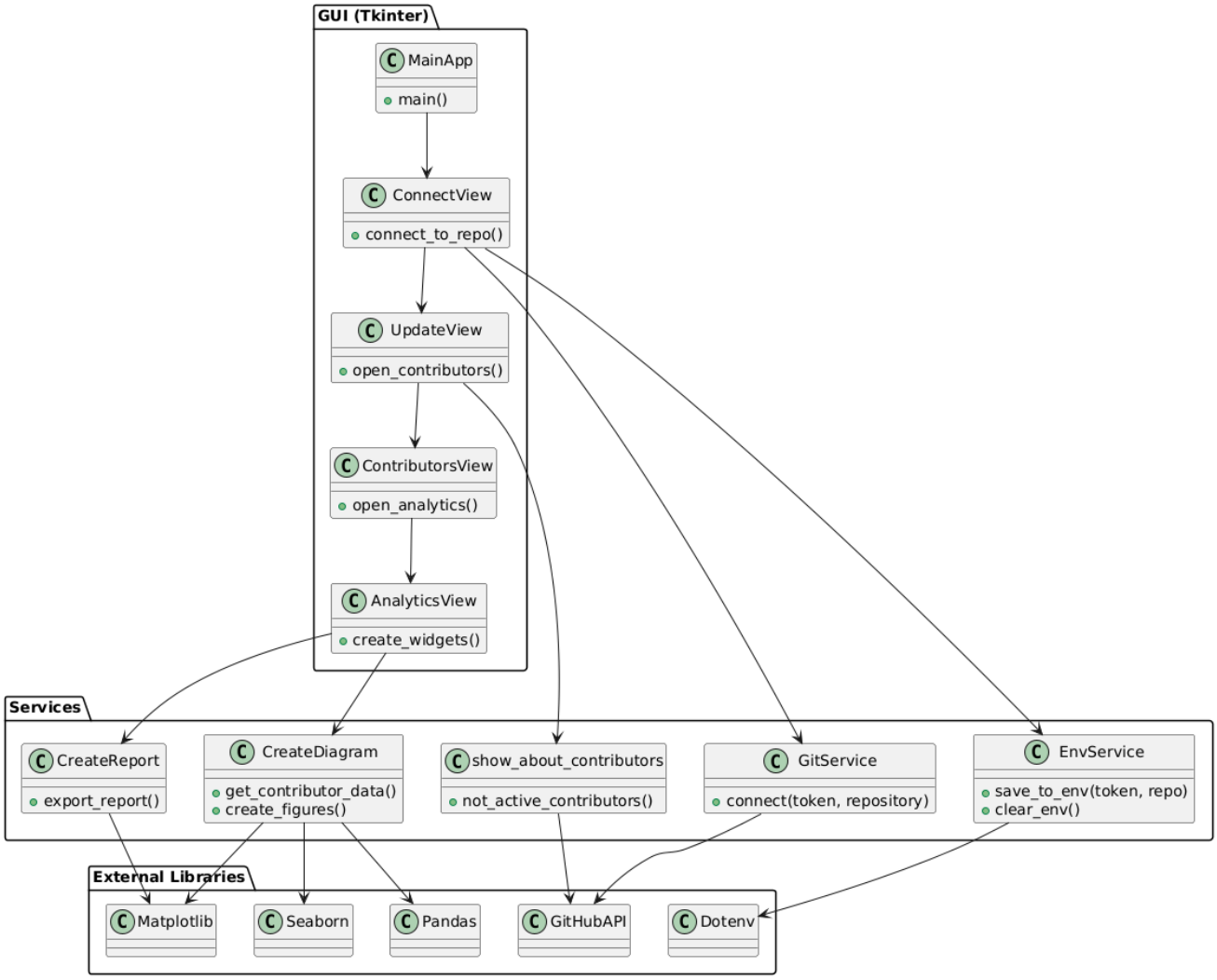


Рис. 4.1 Діаграма архітектури системи

4.1 Архітектура інтерфейсу системи “Kanter”

Інтерфейс реалізований у вигляді набору взаємозв’язаних вікон (вкладок), які відкриваються відповідно до дій користувача. Основні елементи:

– ConnectView — початкове вікно, яке дозволяє користувачеві ввести GIT-токен та посилання на репозиторій. Після успішного підключення зберігаються дані в .env файлі для зручності повторного використання..

– UpdateView — вікно з кнопками оновлення даних та повернення назад. При оновленні система перевіряє активність користувачів за останній тиждень.

– ContributorsView — динамічно згенерований список активних користувачів з репозиторію у вигляді кнопок. При натисканні на ім’я виконується перехід до аналітики конкретного розробника.

– AnalyticsView — відображає графічну аналітику діяльності розробника, а також дозволяє зберегти звіт у форматі PDF.

Компоненти пов’язані між собою через передачу посилання на parent (головне вікно) і repo — об’єкт підключеного репозиторію Git.

4.1.1 ConnectView

Інтерфейс підключення до Git-репозиторію є першим етапом взаємодії користувача з системою Kanter. Реалізовано модуль ConnectView, що відповідає за введення облікових даних: токену доступу та імені репозиторію. А також за ініціалізацію підключення до віддаленого репозиторію через Git API.

```
from tkinter import ttk, messagebox
from ui.update_view import UpdateView
from services.git_service import GitService
from services.env_service import EnvService
from dotenv import load_dotenv
import os
```

Рис. 4.2 Бібліотеки класу ConnectView у connect_view.py

Стандартна бібліотека Python для побудови GUI. Tkinter забезпечує базові інструменти для створення форм, кнопок, вкладок, полів введення та інших елементів інтерфейсу, що дозволило створити простий та зрозумілий інтерфейс системи “Kanter”. Для кращої організації вікон було реалізовано вкладковий інтерфейс, що полегшує навігацію між модулями програми.

Модуль ttk з бібліотеки Tkinter. ttk (themed tk) це розширення tcl/tk з набором віджетів. У ttk використовується двигун для створення віджетів. Цей двигун має підтримку тем і стилів оформлення. Завдяки цьому віджети ttk виглядають природніше в різних операційних системах.

У коді connect_view.py надає віджети. Він використовується для створення полів вводу (Entry), кнопок (Button), міток (Label) тощо.

messagebox — підмодуль, який дозволяє відображати вікна повідомлень користувачу як повідомлення про успішне або неуспішне підключення.

ui.update_view import UpdateView — імпортую UpdateView після успішного підключення до репозиторію.

from services.git_service import GitService — імпорт сервісу GitService для роботи з Git API.

from services.env_service import EnvService — імпортує EnvService відповідає за збереження токена та репозиторію в .env-файл для подальшого автоматичного завантаження при наступному запуску програми.

python-dotenv — це невеликий пакет, який зчитує пари ключ-значення з файлу. Метод з бібліотеки python-dotenv, який завантажує змінні середовища з .env-файлу, що є зручним способом конфігурації токенів, паролів тощо.

os — це модуль os надає безліч функцій для роботи з операційною системою, причому їхня поведінка, як правило, не залежить від ОС.

Після імпортування бібліотек та модулів з інших модулів програмного коду системи “Kanter” виконується сам клас ConnectView:

```
class ConnectView(ttk.Frame):  6 usages
    def __init__(self, parent):  Vlad
        super().__init__(parent)
        self.parent = parent
        self.create_connect_widgets()
```

Рис. 4.3 Клас ConnectView

Метод __init__ — конструктор класу ConnectView. Нічого не повертає та приймає self, parent:

– parent — це головне вікно або інший контейнер, в який вбудовується ConnectView.

– super().__init__(parent) викликає ініціалізацію батьківського класу ttk.Frame.

– create_connect_widgets() створює всі віджети для введення токена та репозиторію.

Метод create_connect_widgets() створює вікно у якому користувач системи “Kanter” може ввести Git токен та посилання на репозиторій та підключитися.

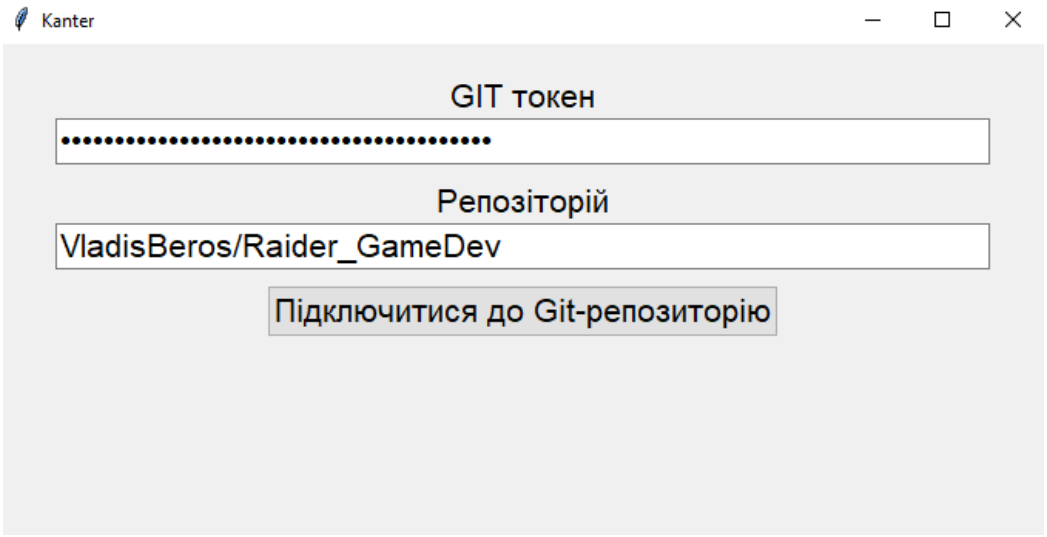


Рис. 4.4 Меню підключення до репозиторію

```
def create_connect_widgets(self):  1 usage  Vlad +1 *
    load_dotenv()

    style = ttk.Style()
    style.configure( style: "Large.TButton", font=("Arial", 16))

    label_token = ttk.Label(self, text="GIT токен", font=("Arial", 16))
    label_token.pack(pady=(20, 0))
    self.token_entry = ttk.Entry(self, width=50, font=("Arial", 16), show="•")
    self.token_entry.pack()
    self.token_entry.bind("<KeyRelease>", self.check_entry)

    label_repo = ttk.Label(self, text="Репозіторій", font=("Arial", 16))
    label_repo.pack(pady=(10, 0))
    self.repo_entry = ttk.Entry(self, width=50, font=("Arial", 16))
    self.repo_entry.pack()
    self.repo_entry.bind("<KeyRelease>", self.check_entry)

    repo_name = os.getenv("REPO_NAME", "")
    git_token = os.getenv("GIT_TOKEN", "")
    self.repo_entry.insert( index: 0, repo_name)
    self.token_entry.insert( index: 0, git_token)

    self.connect_button = ttk.Button(self,
                                     text="Підключитися до Git-репозиторію",
                                     style="Large.TButton",
                                     command=self.connect_to_repo)
    self.connect_button.pack(pady=(10, 0))
```

Рис. 4.5 Код методу create_connect_widgets()

Спочатку метод create_connect_widgets() виклиає метод load_dotenv() який завантажує змінні з .env, якщо такі існують.

Змінна slyle — створює об’єкт стилю для ttk віджетів, а також налаштовує стиль тексту: "Large.TButton" зі шрифтом Arial розміром 16.

lable_token та self.token_entry створює мітку "GIT токен" із зазначеним шрифтом Arial і створює поле введення для токена де ширина дорівнює 50 символів, шрифт Arial 16, символи відображаються як "•" для безпеки на випадок якщо інша людина побачить токен. Теж саме з label_repo, який створює поле вводу посилання на репозиторій з двох частин, які розділяє знак слеш (/): назва облікового запису творця репозиторію та сама назва репозиторію. І наприкінці прив'язує подію

натискання клавіші методу check_entry(), який змінює стан кнопки “Підключитися до репозиторію” з активної на неактивної, якщо у полях вводу щось введено. Потім при натисканні у параметру command виконується метод підключення до репозиторію connect_to_repo().

```
def check_entry(self, event):  2 usages  Vlad +1
    if self.token_entry.get() and self.repo_entry.get():
        self.connect_button["state"] = "normal"
    else:
        self.connect_button["state"] = "disabled"
```

Рис. 4.6 Метод check_entry() для перевірки стану кнопки

Метод connect_to_repo() спочатку витягує токен та репозиторій, а також викликає та отримує дані з полів та викликає GitService.connect(). За результатами якщо успішно — відображає UpdateView, або ні — показує повідомлення про помилку.

```
def connect_to_repo(self):  1 usage  Vlad +1
    token = self.token_entry.get()
    repository = self.repo_entry.get()

    connect_result = GitService.connect(token, repository)

    if connect_result['success']:
        messagebox.showinfo( title: "Успіх",  message: f"{connect_result['message']}")
        EnvService.save_to_env(token, repository)
        repo = connect_result['repo']

        self.destroy()
        update_view = UpdateView(self.parent, repo)
        update_view.pack(fill='both', expand=True)
    else:
        messagebox.showerror( title: "Помилка",  message: f"{connect_result['message']}")
```

Рис. 4.7 Метод connect_to_repo() для підключення до репозиторію

4.1.2 UpdateView

Після успішного підключення до Git-репозиторію користувач потрапляє на наступний етап роботи із системою "Kanter" — оновлення та обробка даних. За це відповідає модуль UpdateView, який надає графічний інтерфейс для виконання подальших дій: оновлення статистики контриб'юторів або повернення до етапу підключення.

```
from tkinter import ttk, messagebox
from ui.contributors_view import ContributorsView
from services.show_about_contributors import not_active_contributors
```

Рис. 4.8 Бібліотеки класу UpdateView у update_view.py

Імпортовані бібліотеки та модулі:

– from tkinter import ttk, messagebox — стандартна бібліотека Python tkinter забезпечує створення графічного інтерфейсу користувача. Підмодуль ttk використовується для створення сучасних віджетів з підтримкою тем, таких як Button, Frame, Style, які виглядають нативно в різних ОС. Підмодуль messagebox надає функції для показу діалогових вікон з повідомленнями користувачу, зокрема інформаційних (showinfo) чи помилкових.

– from ui.contributors_view import ContributorsView — імпортує вікно ContributorsView, яке відображається після успішного оновлення даних контриб’юторів з Git репозиторіїв.

– from services.show_about_contributors import not_active_contributors — імпорт функції, яка здійснює аналіз активності контриб’юторів у репозиторії, зокрема знаходить неактивних розробників. Вона також виконує оновлення даних перед переходом до наступного вікна.

Клас UpdateView наслідується від ttk.Frame, що дозволяє йому виступати в якості окремого фрейму інтерфейсу користувача, який можна вставити у головне вікно. Цей клас є частиною модульної структури додатку та реалізує екран з кнопками, які виконують ключові дії над підключеним репозиторієм. Метод

__init__(self, parent, repo) — це конструктор класу, який ініціалізує фрейм і викликає метод create_widgets() для створення необхідних елементів інтерфейсу, де parent — головне вікно або контейнер, у який вставляється фрейм, а repo — об’єкт репозиторію, отриманий після успішного з’єднання з Git через GitService.

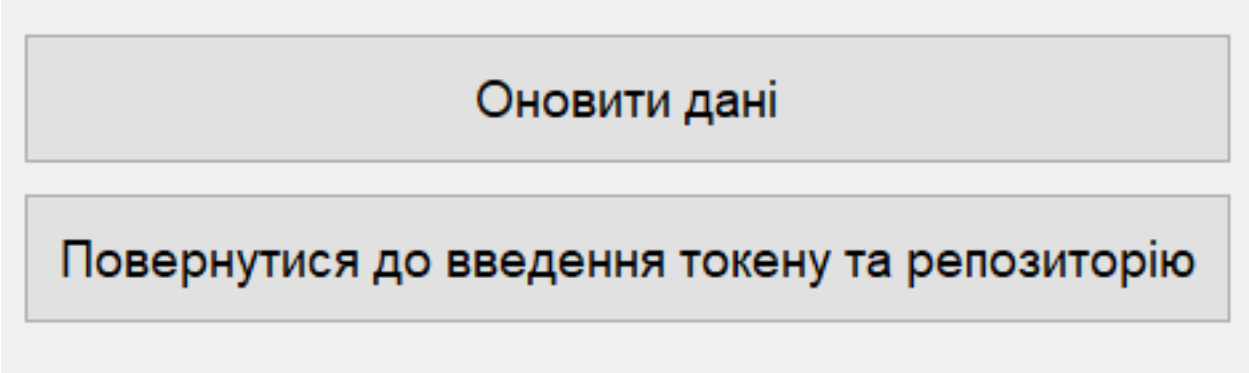


Рис. 4.9 Віджет UpdateView

```
class UpdateView(ttk.Frame):  2 usages
    def __init__(self, parent, repo):
        super().__init__(parent)
        self.parent = parent
        self.repo = repo
        self.create_widgets()
```

Рис. 4.9 Клас UpdateView

create_widgets() — метод створює основні графічні елементи на екрані: встановлює стиль для кнопок із шрифтом Arial розміром 14 та внутрішнім відступом 10 пікселів.

button_frame — розміщує кнопки та створює контейнер button_frame, який розміщується по центру вікна завдяки параметрам relx, rely і anchor у центрі.

update_data_button — кнопка "Оновити дані", яка при натисканні викликає метод open_contributors, що виконує оновлення інформації про учасників репозиторію.

back_button — кнопка "Повернутися до введення токену та репозиторію", яка дає можливість користувачу повернутись до першого екрану ConnectView.

У свою чергу конфігурація сітки використовується для правильного масштабування кнопок при зміні розміру вікна.

```
def create_widgets(self):  1 usage  Vlad *
    style = ttk.Style()
    style.configure( style: "Large.TButton", font=("Arial", 14), padding=10)
    button_frame = ttk.Frame(self)
    button_frame.place(relx=0.5, rely=0.5, anchor='center')

    update_data_button = ttk.Button(button_frame, text="Оновити дані",
                                    style="Large.TButton", command=self.open_contributors)
    update_data_button.grid(row=0, column=0, sticky='ew', pady=5)

    back_button = ttk.Button(button_frame, text="Повернутися до введення токену та репозиторію",
                             style="Large.TButton", command=self.go_back_to_connect)
    back_button.grid(row=1, column=0, sticky='ew', pady=5)

    button_frame.grid_rowconfigure( index: 0, weight=1)
    button_frame.grid_rowconfigure( index: 1, weight=1)
    button_frame.grid_columnconfigure( index: 0, weight=1)
```

Рис. 4.10 Метод create_widgets()

open_contributors(self) — це метод, що викликається при натисканні кнопки "Оновити дані". Завдяки messagebox.showinfo() виводиться інформаційне повідомлення: "Оновлення", "Дані оновлюються...". Наступне що робить метод: викликає функцію not_active_contributors(self.repo) для аналізу контриб’юторів та збору актуальних даних з Git API, далі знищує поточне вікно за допомогою методу destroy() та відкриває вікно ContributorsView, передаючи туди об'єкт repo.

```
def open_contributors(self):  1 usage  Vlad +1
    messagebox.showinfo( title: "Оновлення",  message: "Дані оновлюються...")
    not_active_contributors(self.repo)

    self.destroy()
    contributors_view = ContributorsView(self.parent, self.repo)
    contributors_view.pack(fill='both', expand=True)
```

Рис. 4.11 Метод open_contributors()

go_back_to_connect(self) — це метод, що викликається при натисканні кнопки "Повернутися до введення токену та репозиторію", який імпортує ConnectView (імпорт всередині методу використовується для уникнення циклічних імпортів), знищує поточне вікно та відображає ConnectView використовуючи лениве імпортування.

```
def go_back_to_connect(self):  1 usage  Vlad
    from ui.connect_view import ConnectView

    self.destroy()
    connect_view = ConnectView(self.parent)
    connect_view.pack(fill='both', expand=True)
```

Рис. 4.12 Метод go_back_to_connect()

Таким чином, клас UpdateView виконує роль навігаційного та функціонального вузла системи Kanter, дозволяючи користувачу виконати оновлення даних про розробників або повернутись на попередній етап. Завдяки використанню ttk, messagebox та чіткій модульній структурі, реалізація є зрозумілою, масштабованою та зручною для користувача.

4.1.3 ContributorsView

Після успішного підключення до GitHub-репозиторію, користувач системи “Kanter” потрапляє до інтерфейсу ContributorsView. Цей модуль відповідає за відображення списку всіх учасників репозиторію, отриманого через Git API. Кожен учасник представлений окремою кнопкою, натискання на яку відкриває аналітичне вікно з деталізацією його активності. Даний інтерфейс спрощує вибір розробника для подальшого аналізу.

```
import tkinter as tk
from tkinter import ttk, messagebox
from ui.programmer_analytics_view import AnalyticsView
```

Рис. 4.12 Бібліотеки класу ContributorsView у contributors_view.py

Імпортовані бібліотеки:

– tkinter — стандартна бібліотека Python для створення графічного інтерфейсу. Імпортується як tk, щоб мати доступ до низькорівневих віджетів. Canvas, необхідного для реалізації скролінгу.

– ttk — підмодуль tkinter, що надає сучасніші, стилізовані віджети. Використовується для створення Button, Frame, Scrollbar тощо.

– messagebox — модуль для створення діалогових вікон повідомлень, таких як повідомлення про завантаження аналітики або повернення до попереднього вікна.

– ui.programmer_analytics_view import AnalyticsView — імпорт представлення для відображення детальної аналітики конкретного програміста. Після натискання на кнопку з логіном користувача відкривається це вікно.

Конструктор __init__ — створює екземпляр ContributorsView. parent — батьківський елемент інтерфейсу, до якого додається це представлення. repo — об’єкт репозиторію, отриманий раніше через GitHub API. Він передається в представлення для отримання списку контриб’юторів. super().__init__(parent) —

викликає конструктор базового класу ttk.Frame. Метод create_widgets() створює всі елементи інтерфейсу для перегляду контриб’юторів.

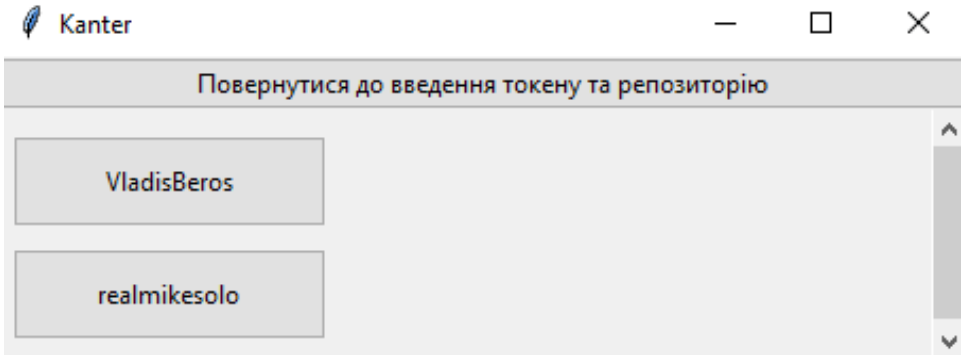


Рис. 4.13 Віджет ContributorsView

```
class ContributorsView(ttk.Frame):
    def __init__(self, parent, repo):
        super().__init__(parent)
        self.parent = parent
        self.repo = repo
        self.create_widgets()
```

Рис. 4.14 Клас ContributorsView

Реалізовано скролбар, де Canvas — полотно, яке містить усі віджети зі скролінгом. Воно забезпечує гнучкий механізм відображення великої кількості віджетів. Scrollbar — вертикальний скролбар, прив'язаний до полотна через метод yview. scrollable_frame — вкладена рамка, яка містить список кнопок з логінами. Вона створена всередині Canvas.

Кнопка повернення дозволяє користувачу повернутися на попередній етап до введення токену та назви репозиторію. Метод go_back_to_connect() відповідає за знищення поточного вікна та відкриття ConnectView.

self.scrollable_frame.bind("<Configure>", lambda e:

self.canvas.configure(scrollregion=self.canvas.bbox("all"))) забезпечує автоматичне оновлення області прокрутки при зміні розмірів вмісту scrollable_frame, де create_window() вставляє scrollable_frame всередину полотна та є прив'язка скролбару до вертикальної прокрутки полотна через pack() scrollable_frame та canvas. Розміщення скролбару праворуч та полотна ліворуч. Вони розширюються відповідно до вікна.

Створено генерацію кнопок методом get_contributors() GitHub API повертає список контриб'юторів репозиторію де імена (логіни) витягуються для кожного контриб'ютора і зберігаються в списку logins.

```
contributors = list(self.repo.get_contributors())
logins = [c.login for c in contributors]

for login in logins:
    btn = ttk.Button(
        self.scrollable_frame,
        text=login,
        command=lambda l=login: self.open_analytics(l),
        padding=(10, 10),
        width=20
    )
    btn.pack(pady=5, fill='x', expand=True)
```

Рис. 4.15 Створення кнопок з логінами розробників репозиторію

Для кожного логіна створюється окрема кнопка з текстом у вигляді імені користувача. При натисканні викликається метод open_analytics(), в який передається логін. Аргумент padding робить кнопку роздільної від інших елементів інтерфейсу. lambda l=login необхідний для правильного захоплення поточного значення змінної в циклі.

```
def create_widgets(self):  1 usage  Vlad
    self.canvas = tk.Canvas(self, borderwidth=5)
    self.scrollbar = ttk.Scrollbar(self, orient="vertical", command=self.canvas.yview)
    self.scrollable_frame = ttk.Frame(self.canvas)

    back_button = ttk.Button(self, text="Повернутися до введення токену та репозиторію", command=self.go_back_to_connect)
    back_button.pack(fill='x')

    self.scrollable_frame.bind("<Configure>", lambda e: self.canvas.configure(scrollregion=self.canvas.bbox("all")))
    self.canvas.create_window((0, 0), window=self.scrollable_frame, anchor="nw")
    self.canvas.configure(yscrollcommand=self.scrollbar.set)

    self.scrollbar.pack(side="right", fill="y")
    self.canvas.pack(side="left", fill="both", expand=True)

    contributors = list(self.repo.get_contributors())
    logins = [c.login for c in contributors]

    for login in logins:
        btn = ttk.Button(
            self.scrollable_frame,
            text=login,
            command=lambda l=login: self.open_analytics(l),
            padding=(10, 10),
            width=20
        )
        btn.pack(pady=5, fill='x', expand=True)
```

Рис. 4.16 Повний код методу create_widgets()

open_analytics() — це метод, що показує інформаційне повідомлення через messagebox.showinfo(), що аналітика готується. Далі знищує поточне вікно ContributorsView методом self.destroy(). Створює нове вікно аналітики AnalyticsView, передаючи батьківський елемент, логін користувача та об’єкт репозиторію. Аналітика відкривається у новій вкладці або в тому самому вікні.

```
def open_analytics(self, login):  1 usage  Vlad
    messagebox.showinfo( title: f"Обран {login}",  message: "Будується аналітика...")
    self.destroy()
    programmer_analytics_view = AnalyticsView(self.parent, login, self.repo)
    programmer_analytics_view.pack(fill='both', expand=True)
```

Рис. 4.17 Метод open_analytics(), що створює аналітику праці розробника

go_back_to_connect() — це метод лениво імпортує ConnectView безпосередньо всередині методу (уникнення циклічних імпортів) та знищує поточне вікно та відкриває вікно введення токену та репозиторію.

```
def go_back_to_connect(self):  1 usage  Vlad *
    from ui.connect_view import ConnectView
    self.destroy()
    connect_view = ConnectView(self.parent)
    connect_view.pack(fill='both', expand=True)
```

Рис. 4.18 Метод go_back_to_connect(), що повертає до меню підключення

4.1.4 AnalyticsView

Після вибору розробника з репозиторію користувач системи “Kanter” переходить до етапу аналітики внеску конкретного учасника команди. Для реалізації цієї частини інтерфейсу розроблено модуль AnalyticsView, який забезпечує побудову графіків активності розробника, а також функцію експорту аналітичного звіту у форматі PDF.

```
from tkinter import ttk
from services.create_diagram import CreateDiagram
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from services.create_pdf_report import CreateReport
```

Рис. 4.19 Бібліотеки класу AnalyticsView у programmer_analytics_view.py

Імпортовані бібліотеки та модулі:

– from tkinter import ttk — використовується для створення графічного інтерфейсу, аналогічно як у попередніх вікнах. Модуль ttk (themed tk) забезпечує сучасніші за виглядом віджети з підтримкою тем оформлення, які краще інтегруються в інтерфейс операційної системи.

– from services.create_diagram import CreateDiagram — імпортує сервісний клас CreateDiagram, який відповідає за обробку аналітичних даних і генерацію візуалізацій на основі активності розробника у Git-репозиторії. Метод get_contributor_data() отримує статистику комітів, а create_figures() — створює графіки.

– from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg — імпорт компоненту, що дозволяє вбудовувати графіки з бібліотеки matplotlib безпосередньо у вікна Tkinter. Це забезпечує інтерактивне відображення

побудованих графіків у вкладках інтерфейсу.

– from services.create_pdf_report import CreateReport — модуль для експорту створених графіків у PDF-файл. Клас CreateReport забезпечує метод export_report(), який генерує звіт для конкретного користувача на основі побудованих діаграм.

Клас AnalyticsView успадковується від ttk.Frame і є окремим вкладковим вікном, що відображає статистичну інформацію про одного з розробників. Метод __init__(), де parent — головне вікно або контейнер, у який вставляється цей фрейм, login — логін обраного користувача в GitHub, або інших. repo — об'єкт репозиторію, з якого буде витягнута інформація. Метод create_widgets() викликається одразу після ініціалізації, щоб створити весь інтерфейс.

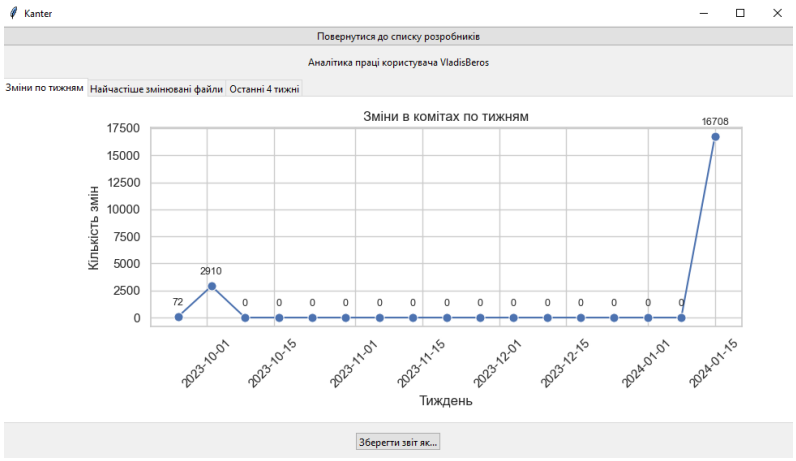


Рис. 4.20 Віджет з аналітикою AnalyticsView

```
class AnalyticsView(ttk.Frame):  2 usages
    def __init__(self, parent, login, repo):
        super().__init__(parent)
        self.parent = parent
        self.login = login
        self.repo = repo
        self.create_widgets()
```

Рис. 4.21 Клас AnalyticsView

create_widgets() це метод, що створює інтерфейс для перегляду аналітики, де

кнопка back_button — дає змогу повернутися назад до списку учасників репозиторію. Її обробник — метод go_back_to_contributors(), який знищує поточне вікно та завантажує попереднє. Мітка label — відображає заголовок вікна з іменем користувача, для якого проводиться аналітика. data = CreateDiagram.get_contributor_data(self.login, self.repo) — отримання статистики комітів користувача (кількість, розмір тощо). figures = CreateDiagram.create_figures(data) — побудова графіків на основі отриманих даних. Повертає словник із назвою графіка як ключем та об'єктом Figure як значенням.

notebook = ttk.Notebook(self) — створення вкладкового віджета. Кожен графік додається на окрему вкладку. В циклі for title, fig in figures.items() творюється вкладка frame, графік виводиться у цій вкладці через FigureCanvasTkAgg, використовується метод draw() для рендеру графіка, а get_tk_widget() вставляє його у графічне вікно.

bottom_frame = ttk.Frame(self) — контейнер для нижніх елементів інтерфейсу, де export_button — кнопка, що викликає метод CreateReport.export_report() з передачею побудованих графіків та логіну користувача. У результаті формується PDF-файл, зручний для збереження або презентації звіту керівництву.

```
def create_widgets(self):  1 usage  Vlad +1 *
    back_button = ttk.Button(self, text="Повернутися до списку розробників",
                             command=self.go_back_to_contributors)
    back_button.pack(fill='x')

    label = ttk.Label(self, text=f"Аналітика праці користувача {self.login}")
    label.pack(pady=10)

    data = CreateDiagram.get_contributor_data(self.login, self.repo)
    figures = CreateDiagram.create_figures(data)

    notebook = ttk.Notebook(self)
    notebook.pack(fill='both', expand=True)

    for title, fig in figures.items():
        frame = ttk.Frame(notebook)
        canvas = FigureCanvasTkAgg(fig, master=frame)
        canvas.draw()
        canvas.get_tk_widget().pack(fill='both', expand=True)
        notebook.add(frame, text=title)

    bottom_frame = ttk.Frame(self)
    bottom_frame.pack(fill='x', side='bottom', pady=10)
    export_button = ttk.Button(bottom_frame, text="Зберегти звіт як...",
                               command=lambda: CreateReport.export_report(figures, self.login))
    export_button.pack()
```

Рис. 4.22 Повний код методу create_widgets()

Метод go_back_to_contributors() імпортується модуль ContributorsView — попереднє вікно зі списком розробників та знищує поточне вікно AnalyticsView, та завантажує ContributorsView, передаючи у нього репозиторій.

```
def go_back_to_contributors(self):  1 usage  VladisBeros *
    from ui.contributors_view import ContributorsView
    self.destroy()
    contributors_view = ContributorsView(self.parent, self.repo)
    contributors_view.pack(fill='both', expand=True)
```

Рис. 4.23 Метод go_back_to_contributors()

4.2 Архітектура логіки системи “Kanter”

Логічна частина системи Kanter реалізована у вигляді набору взаємопов'язаних сервісів, які відповідають за роботу з Git API, обробку даних та генерацію звітів. Основні модулі:

– GitService — це центральний сервіс для взаємодії з Git API. Слугує для підключення до репозиторію за допомогою токена, кешування результатів підключення для оптимізації повторних запитів, надання об'єкта репозиторію для подальшої обробки.

– EnvService — відповідає за роботу з конфігурацією та безпеку даних: зберігання токена та назви репозиторію в .env файлі, очищення конфіденційних даних при виході з системи, автоматичне заповнення полів у ConnectView при повторному запуску.

– CreateDiagram — це сервіс для аналітики та візуалізації даних: збір статистики по розробнику, генерація графіків на основі matplotlib.

– CreateReport — це сервіс для експорту даних: збереження аналітики у PDF-форматі, автоматичне створення титульної сторінки звіту, інтеграція з графіками з CreateDiagram.

– not_active_contributors() — це допоміжна функція для перевірки активності розробників: визначає користувачів без комітів за останні 7 днів,

виводить список неактивних розробників через messagebox.

Система побудована так: кожен сервіс виконує одну чітку задачу, що спрощує масштабування та модифікацію коду.

4.2.1 GitService

GitService є основним сервісом системи Kanter, який відповідає за взаємодію з GitHub API через офіційну бібліотеку PyGithub. Він забезпечує автентифікацію за допомогою токену доступу, отримання даних репозиторію, кешування підключень для оптимізації продуктивності.

Бібліотеки та залежності:

– github (PyGithub) — офіційний Python-клієнт для Git API.

– functools.lru_cache — декоратор для кешування результатів підключення (зберігає до 10 останніх запитів).

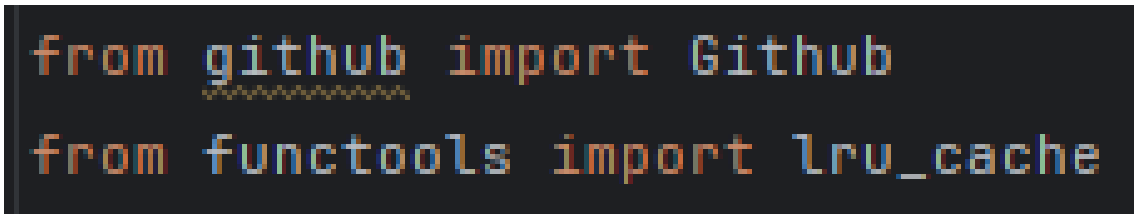


Рис. 4.24 Бібліотеки імпортовані у git_service.py

Метод connect() містить вхідні параметри: token — персональний токен доступу до Git, repository — повна назва репозиторію у форматі username/repo. Логіка роботи така: створює об’єкт Github з токеном, потім викликає get_repo() для отримання репозиторію і повертає словник з результатом:

– success — статус операції (True/False).

– message — повідомлення про помилку або успіх.

– repo — об’єкт Repository (при успіху).

Обробка помилок: перехоплює винятки та повертає їх у зрозумілому форматі. Кешування (@lru_cache). Зменшує кількість запитів до GitHub API при повторних підключеннях до того ж репозиторію. Інтеграція з іншими сервісами. Передає

об’єкт repo до UpdateView для перевірки активності розробників, до ContributorsView для отримання списку контриб’юторів, до CreateDiagram для побудови аналітики.

```python
@staticmethod  1 usage  Vlad
@lru_cache(maxsize=10)
def connect(token, repository):
    connect_result = {'message': "З'єднання успішне!", 'success': False, 'repo': None}

    try:
        g = Github(f"{token}")
        repo = g.get_repo(f"{repository}")
        connect_result['success'] = True
        connect_result['repo'] = repo
    except Exception as e:
        connect_result['message'] = f"[ERROR] Невідома помилка: {str(e)}"

    return connect_result
```

Рис. 4.25 Повний код connect() у GitService

4.2.2 EnvService

EnvService — це сервіс для роботи з конфігураційними даними системи “Kanter”. Він відповідає за збереження, завантаження та очищення конфіденційної інформації (GIT-токену та назви репозиторію) у файлі .env, що дозволяє системі зручно керувати налаштуваннями користувача між сеансами роботи.

EnvService надає функціонал збереження даних за допомогою save_to_env та перевіряє наявність файлу .env чи створює його, якщо не існує. Силаючись на .env за посиланням у ENV_PATH, записує токен (GIT_TOKEN) та назву репозиторію (REPO_NAME) у файл за допомогою set_key з бібліотеки python-dotenv та оновлює змінні середовища (os.environ) для негайного доступу під час поточного сеансу. Також очищення даних за допомогою clear_env, якщо користувач при закритті програми вказав, що хоче вийти з облікового запису та щоб конфіденційні дані не заповнювали поля при активації системи.

```python
ENV_PATH = ".env"

class EnvService:  4 usages  VladisBeros +1
    @staticmethod  1 usage  VladisBeros
    def save_to_env(token, repository):
        if not os.path.exists(ENV_PATH):
            with open(ENV_PATH, 'w'):
                pass

        set_key(ENV_PATH,  key_to_set: "GIT_TOKEN",  value_to_set: f'{token}')
        set_key(ENV_PATH,  key_to_set: "REPO_NAME",  value_to_set: f'{repository}')
        os.environ["GIT_TOKEN"] = token
        os.environ["REPO_NAME"] = repository

    @staticmethod  1 usage  VladisBeros
    def clear_env():
        if os.path.exists(ENV_PATH):
            unset_key(ENV_PATH,  key_to_unset: "GIT_TOKEN")
            unset_key(ENV_PATH,  key_to_unset: "REPO_NAME")
        os.environ.pop("GIT_TOKEN", None)
        os.environ.pop("REPO_NAME", None)
```

Рис. 4.26 Код сервісу EnvService

Використані бібліотеки:

– python-dotenv — для роботи з .env-файлами з додавання/оновлення значень за допомгою set_key та видалення ключа за допомогою unset_key.

– os — для роботи з операційною системою для перевірки наявності файлу за допомогою os.path.exists та керування змінними середовища за допомогою os.environ.

–

```python
from dotenv import set_key, unset_key
import os
```

Рис. 4.27 Імпортовані бібліотеки для EnvService у env_service.py

Переваги такого способу пов'язані у тому, що це безпечне зберігання токену (не в коді, а в окремому файлі), автоматизація роботи з налаштуваннями та відповідає принципу *Separation of Concerns* — відокремлює логіку роботи з конфігурацією від інтерфейсу та Git-логіки.

4.2.3 CreateDiagram

Клас CreateDiagram є ключовим сервісом системи Kanter, який відповідає за аналітику та візуалізацію даних про активність розробників у Git-репозиторії. Він реалізує два основні методи: збір статистики за допомогою get_contributor_data() та генерацію графіків за допомогою create_figures().

Використані бібліотеки:

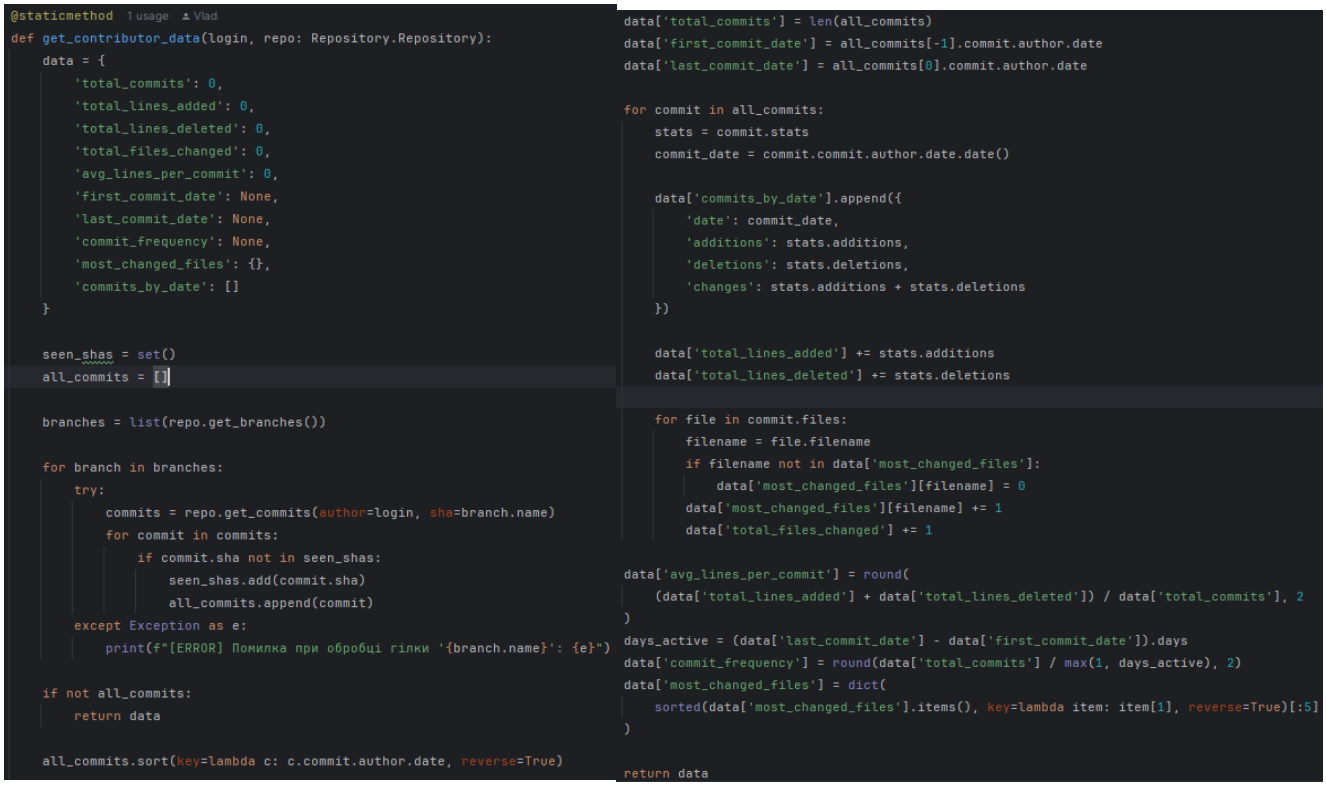
– github (PyGithub) – для роботи з Git API через об'єкт Repository.
– pandas – для обробки та агрегації даних у форматі DataFrame.
– seaborn та matplotlib – для створення професійних графіків.
– numpy – для роботи з числовими масивами.

```
from github import Repository
import pandas as pd
import seaborn as sns
import matplotlib.dates as mdates
from matplotlib.figure import Figure
import numpy as np
```

Рис. 4.28 Імпортовані бібліотеки для CreateDiagram

Метод get_contributor_data() відповідає за збір статистики по конкретному розробнику за login. Логіка роботи така:

– Перевіряє всі гілки репозиторію, щоб уникнути втрати комітів.
– Використовує seen_shas для уникнення повторного аналізу комітів.
– Підрахунок рядків коду (доданих/видалених), визначення найпопулярніших файлів (most_changed_files), розрахунок середньої активності (avg_lines_per_commit, commit_frequency).
– Обробка помилки при зверненні до окремих гілок, якщо коміти відсутні.



```python
@staticmethod  1 usage  Vlad
def get_contributor_data(login, repo: Repository.Repository):
    data = {
        'total_commits': 0,
        'total_lines_added': 0,
        'total_lines_deleted': 0,
        'total_files_changed': 0,
        'avg_lines_per_commit': 0,
        'first_commit_date': None,
        'last_commit_date': None,
        'commit_frequency': None,
        'most_changed_files': {},
        'commits_by_date': []
    }

    seen_shas = set()
    all_commits = []

    branches = list(repo.get_branches())

    for branch in branches:
        try:
            commits = repo.get_commits(author=login, sha=branch.name)
            for commit in commits:
                if commit.sha not in seen_shas:
                    seen_shas.add(commit.sha)
                    all_commits.append(commit)
        except Exception as e:
            print(f"[ERROR] Помилка при обробці гілки '{branch.name}': {e}")

    if not all_commits:
        return data

    all_commits.sort(key=lambda c: c.commit.author.date, reverse=True)
```

```python
    data['total_commits'] = len(all_commits)
    data['first_commit_date'] = all_commits[-1].commit.author.date
    data['last_commit_date'] = all_commits[0].commit.author.date

    for commit in all_commits:
        stats = commit.stats
        commit_date = commit.commit.author.date.date()

        data['commits_by_date'].append({
            'date': commit_date,
            'additions': stats.additions,
            'deletions': stats.deletions,
            'changes': stats.additions + stats.deletions
        })

        data['total_lines_added'] += stats.additions
        data['total_lines_deleted'] += stats.deletions

        for file in commit.files:
            filename = file.filename
            if filename not in data['most_changed_files']:
                data['most_changed_files'][filename] = 0
            data['most_changed_files'][filename] += 1
            data['total_files_changed'] += 1

    data['avg_lines_per_commit'] = round(
        (data['total_lines_added'] + data['total_lines_deleted']) / data['total_commits'], 2
    )
    days_active = (data['last_commit_date'] - data['first_commit_date']).days
    data['commit_frequency'] = round(data['total_commits'] / max(1, days_active), 2)
    data['most_changed_files'] = dict(
        sorted(data['most_changed_files'].items(), key=lambda item: item[1], reverse=True)[:5]
    )

    return data
```

Рис. 4.29 та 4.30 Метод get_contributor_data()

Після отримання та обробки даних метод create_figures() перетворює зібрані дані у набір графіків для подальшого відображення в AnalyticsView. Метод create_figures() приймає дані (data) як словник та повертає словник з трьома графіками:

– Зміни по тижням – лінійний графік з маркерами (Seaborn lineplot).

– Найчастіше змінювані файли – горизонтальні стовпчики (barh).

– Останні 4 тижні – порівняння доданих та видалених рядків (стовпчаста діаграма).

Використовується тема Seaborn whitegrid для чистого вигляду. Підписи осей та заголовки локалізовані українською мовою.

```python
sns.set_theme(style="whitegrid")
```

Рис. 4.31 Стилізація за допомогою Seaborn

```python
if not data.get("commits_by_date"):
    return figures

df = pd.DataFrame(data['commits_by_date'])
df['date'] = pd.to_datetime(df['date'])

df_weekly = df.groupby(pd.Grouper(key='date', freq='W-MON')).agg({
    'additions': 'sum',
    'deletions': 'sum',
    'changes': 'sum'
}).reset_index()
```

Рис. 4.32 Структуризація даних з data

```python
first_diagram = Figure(figsize=(5, 4))
first_axis = first_diagram.subplots()

sns.lineplot(data=df_weekly, x='date', y='changes', marker='o', markersize=8, ax=first_axis)

for i, row in df_weekly.iterrows():
    first_axis.text(row['date'], row['changes'] + max(df_weekly['changes']) * 0.05,
        s: f"{int(row['changes'])}", ha='center', va='bottom', fontsize=9
    )

first_axis.set_title('Зміни в комітах по тижням')
first_axis.set_xlabel('Тиждень')
first_axis.set_ylabel('Кількість змін')
first_axis.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))
first_axis.tick_params(axis='x', rotation=45)
first_diagram.tight_layout()
figures['Зміни по тижням'] = first_diagram
```

Рис. 4.33 Алгоритм побудови лінійного графіку

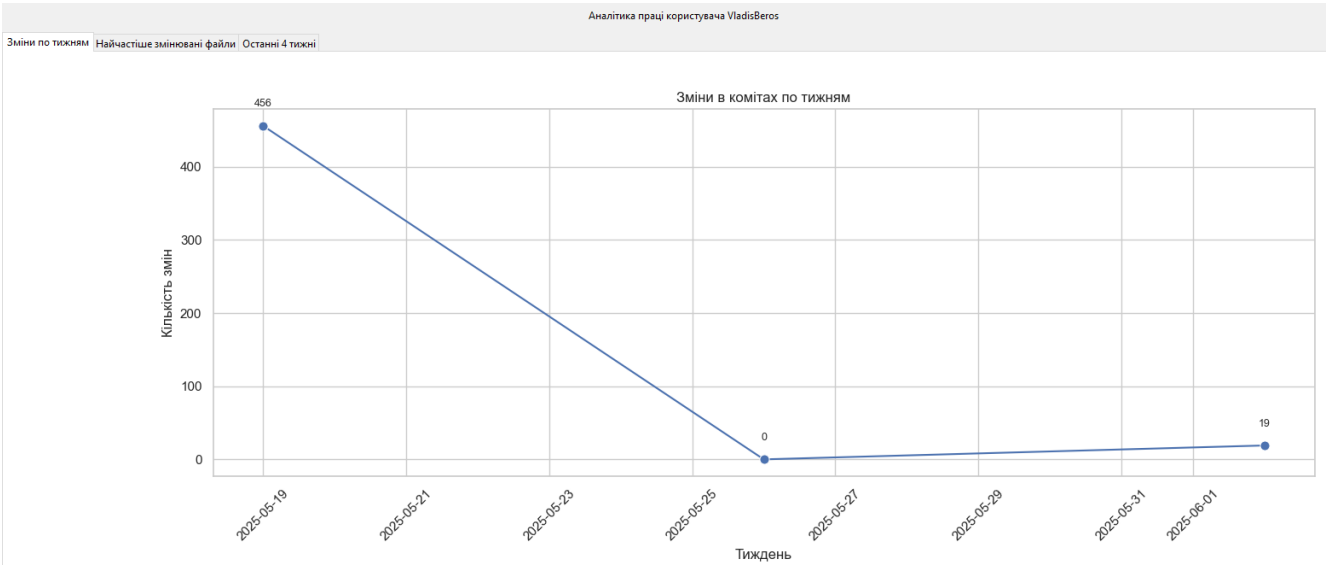


Рис. 4.34 Лінійний графік

```python
if data['most_changed_files']:
    second_diagram = Figure(figsize=(5, 4))
    second_axis = second_diagram.subplots()

    files = list(data['most_changed_files'].keys())
    changes = list(data['most_changed_files'].values())

    bars = second_axis.barh(files, changes, color=sns.color_palette("Blues_d"))
    second_axis.set_title('Найчастіше змінювані файли')
    second_axis.set_xlabel('Кількість змін')

    for bar in bars:
        width = bar.get_width()
        second_axis.text(width + max(changes) * 0.01, bar.get_y() + bar.get_height() / 2,
                 s: f'{int(width)}', va='center')

    second_diagram.tight_layout()
    figures['Найчастіше змінювані файли'] = second_diagram
else:
    second_diagram = Figure(figsize=(5, 4))
    second_axis = second_diagram.subplots()
    second_axis.text( x: 0.5,  y: 0.5,  s: 'Немає даних про зміни файлів', ha='center', va='center')
    figures['Найчастіше змінювані файли'] = second_diagram
```

Рис. 4.35 Алгоритм створення діаграми найчастіше змінюваних файлів

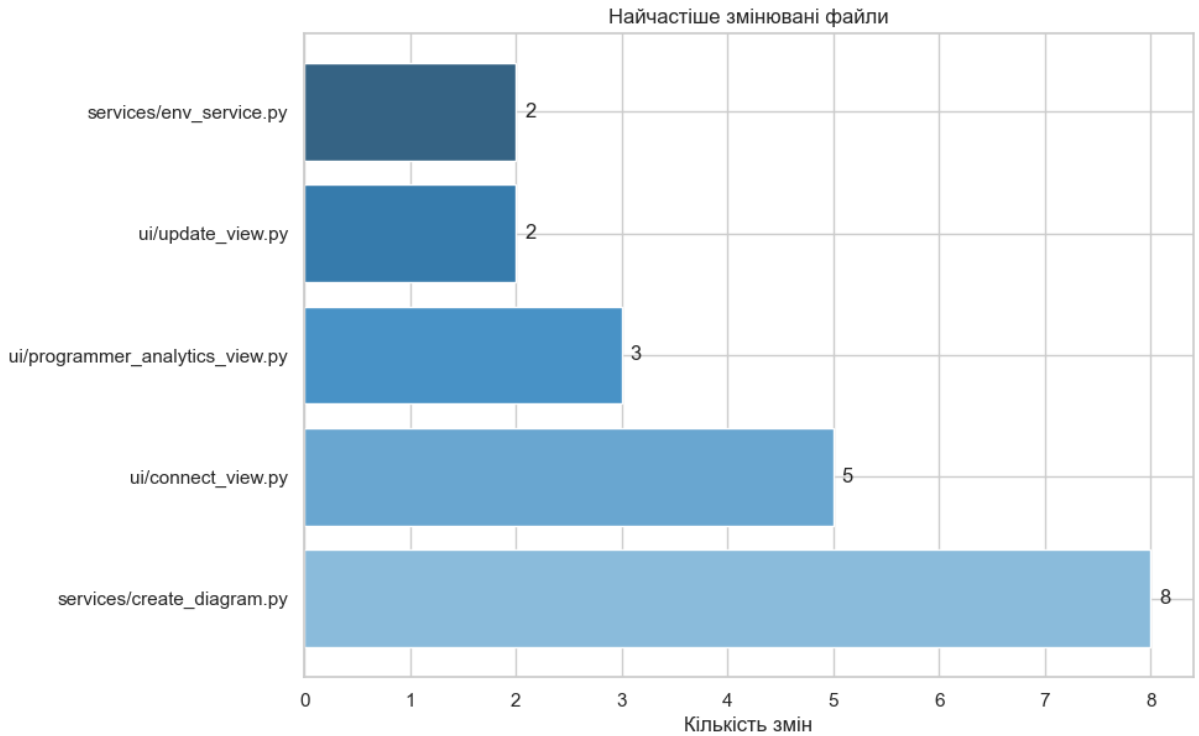


Рис. 4.36 Приклад найчастіше змінюваних файлів

```
third_diagram = Figure(figsize=(5, 4))
third_axis = third_diagram.subplots()

last_4_weeks = df[df['date'] >= (pd.to_datetime('today') - pd.Timedelta(weeks=4))]

if not last_4_weeks.empty:
    weekly_data = last_4_weeks.groupby(
        pd.Grouper(key='date', freq='W-MON')
    ).agg({
        'additions': 'sum',
        'deletions': 'sum'
    }).reset_index()

    weekly_data['week_str'] = weekly_data['date'].dt.strftime('%Y-%m-%d')

    bar_width = 0.35
    x = np.arange(len(weekly_data))

    third_axis.bar(x - bar_width / 2, weekly_data['additions'], width=bar_width, color='green', label='Додані рядки')
    third_axis.bar(x + bar_width / 2, weekly_data['deletions'], width=bar_width, color='red', label='Видалені рядки')

    third_axis.set_title('Зміни за останні 4 тижні')
    third_axis.set_xticks(x)
    third_axis.set_xticklabels(weekly_data['week_str'], rotation=45)
    third_axis.legend()
else:
    third_axis.text( x: 0.5,  y: 0.5,  s: 'Немає даних за останні 4 тижні', ha='center', va='center')

third_diagram.tight_layout()
figures['Останні 4 тижні'] = third_diagram

return figures
```

Рис. 4.37 Алгоритм створення діаграми змін за останні 4 тіждні

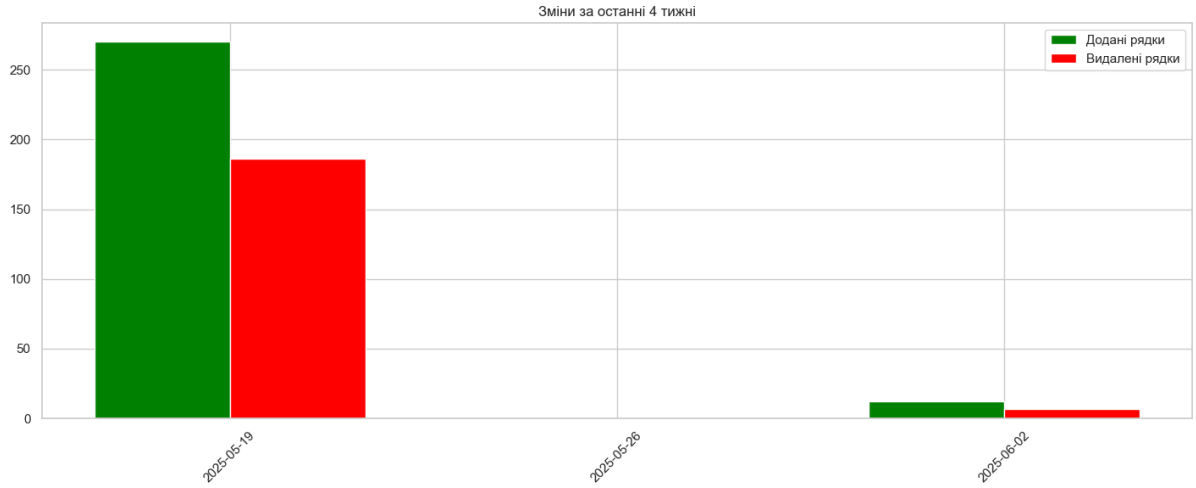


Рис. 4.38 Приклад аналітики змін за останні 4 тіжні

4.2.4 CreateReport

Модуль CreateReport відповідає за генерацію та експорт аналітичних звітів у форматі PDF. Цей компонент системи "Kanter" реалізує функціонал збереження графічної аналітики діяльності розробників у зручному для перегляду та роздруківки вигляді.

Бібліотеки та модулі:

– tkinter.filedialog - надає діалогові вікна для вибору місця збереження файлів.

– matplotlib.backends.backend_pdf - забезпечує функціонал для створення PDF-документів з графіками.

– matplotlib.pyplot - використовується для створення титульної сторінки звіту.

Клас CreateReport містить один ключовий статичний метод export_report(). Цей метод відкриває діалогове вікно для вибору місця збереження PDF-файлу за допомогою filedialog.asksaveasfilename, створює новий PDF-документ за допомогою PdfPages, генерує титульну сторінку звіту з назвою "Аналіз роботи розробника {login}", додає всі графіки зі словника figures до PDF-документа, виводить повідомлення про успішне збереження звіту через messagebox.showinfo.

Титульна сторінка створюється з використанням стандартного розміру A4. Всі графіки зберігаються у тому порядку, в якому вони представлені у словнику figures. Отримує готові графіки з модуля CreateDiagram через параметр figures. Використовує ім'я розробника (login) для персоналізації звіту. Інтегрується з інтерфейсом через діалогові вікна та повідомлення.

```
class CreateReport(ttk.Frame):  2 usages  VladisBeros
    @staticmethod  1 usage  VladisBeros
    def export_report(figures, login):
        file_path = filedialog.asksaveasfilename(defaultextension=".pdf", filetypes=[("PDF files", "*.pdf")], title="Зберегти звіт як PDF")
        if not file_path:
            return

        with PdfPages(file_path) as pdf:
            from matplotlib import pyplot as plt

            fig, ax = plt.subplots(figsize=(8.27, 11.69))
            ax.axis('off')
            ax.text( x: 0.5,  y: 0.9,  s: f"Аналіз роботи розробника {login}", fontsize=16, ha='center', va='top', weight='bold')
            pdf.savefig(fig)
            plt.close(fig)

            for fig in figures.values():
                pdf.savefig(fig)

            messagebox.showinfo( title: "Успіх",  message: f"Звіт про {login} успішно збережено.")
```

Рис. 4.39 CreateReport та статичний метод export_report()

Цей метод викликається з інтерфейсу AnalyticsView при натисканні кнопки "Зберегти звіт як..." і завершує цикл аналізу діяльності розробника, надаючи користувачу можливість зберегти результати для подальшого використання.

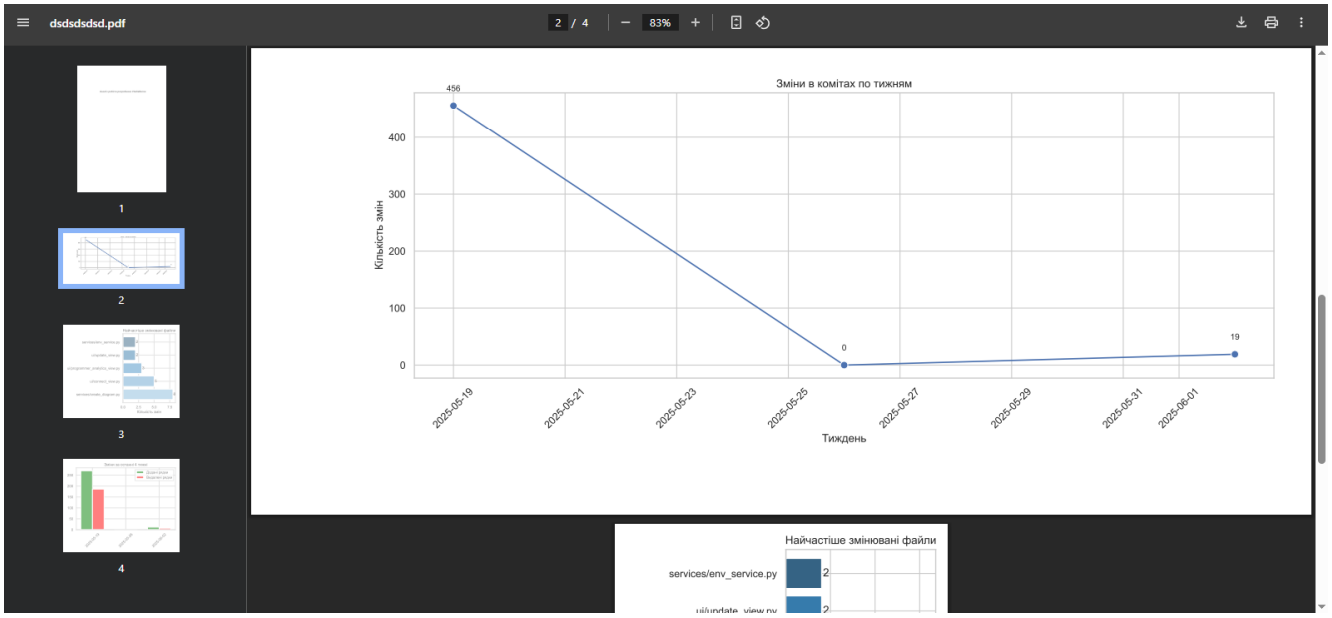


Рис. 4.40 Приклад збереженого PDF звіта

4.2.5 not_active_contributors()

Функція not_active_contributors є ключовим інструментом моніторингу активності розробників у системі Kanter. Вона реалізована у вигляді окремого модуля та виконує такі завдання:

– Аналізує активність контриб'юторів Git-репозиторію за останні 7 днів.

– Ідентифікує розробників, які не робили комітів протягом цього періоду.

– Візуалізує результат у вигляді pop-up сповіщення зі списком неактивних користувачів.

Функція приймає об'єкт репозиторію (repo) типу Repository.Repository з бібліотеки PyGithub. Далі, алгоритм роботи: отримує список усіх контриб'юторів через repo.get_contributors(), визначає часову межу (one_week_ago) для фільтрації комітів, для кожного розробника виконує запит до GitHub API за комітами автора за останній тиждень (repo.get_commits) та додає логін до списку inactive_logins, якщо коміти відсутні.

Візуалізація результатів виконується завдяки messagebox.showwarning для відображення списку неактивних розробників.

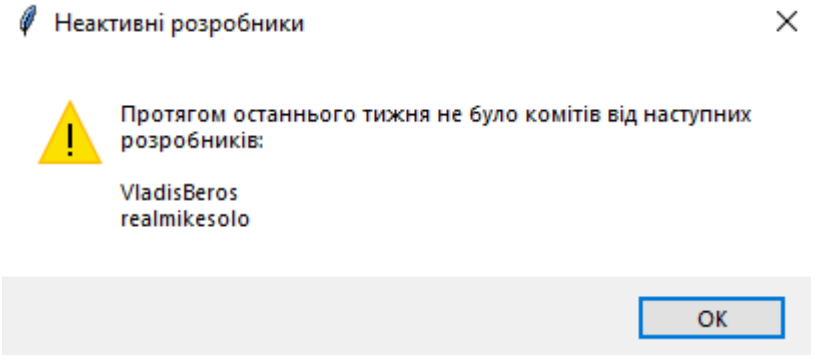


Рис. 4.41 Повідомлення про розробників, які давно не робили змін

Обробка помилок перехоплює винятки під час запитів до Git API, логує помилки у консоль із зазначенням логіну проблемного користувача та продовжує роботу навіть при збоях у окремих запитах. Сама функція викликається з модуля UpdateView при натисканні кнопки "Оновити дані", а результати використовуються для подальшого аналізу в ContributorsView.

```python
import datetime
from tkinter import messagebox

def not_active_contributors(repo):  2 usages  VladisBeros
    contributors = list(repo.get_contributors())
    inactive_logins = []
    one_week_ago = datetime.datetime.utcnow() - datetime.timedelta(days=7)

    for contributor in contributors:
        try:
            commits = list(repo.get_commits(author=contributor.login, since=one_week_ago))
            if not commits:
                inactive_logins.append(contributor.login)
        except Exception as e:
            print(f"[ERROR] Помилка при обробці користувача {contributor.login}: {e}")

    if inactive_logins:
        inactive_list = "\n".join(inactive_logins)
        messagebox.showwarning(
            title: "Неактивні розробники",
            message: f"Протягом останнього тижня не було комітів від наступних розробників:\n\n{inactive_list}"
        )
```

Рис. 4.42 Код функції not_active_contributors()

4.3 Головний модуль системи “Kanter”

Головний модуль програми Kanter є вхідною точкою системи та відповідає за ініціалізацію графічного інтерфейсу та керування життєвим циклом додатка. Реалізований у файлі main.py, він виконує імпортує бібліотеки та модулі: tkinter, messagebox, ConnectView, EnvService.

Функція on_closing() це обробник закриття програми. При виході пропонує очистити облікові дані через EnvService.clear_env() і якщо користувач погодився - виводиться інформаційне повідомлення про успішний вихід та знищує головне вікно завдяки destroy().

Функція main ініціалізує головне вікно Tkinter, встановлює заголовок, розгортає вікно на весь екран (state('zoomed')), налаштовує обробник закриття вікна (WM_DELETE_WINDOW), також запускає головний цикл подій (mainloop()). Точка входу виконується за умовою if __name__ == "__main__" гарантує запуск main() тільки при прямому виконанні файлу.

```
def on_closing(root):  1 usage  VladisBeros
    if messagebox.askyesno( title: "Вихід",  message: "Бажаєте вийти з облікового запису?"):
        EnvService.clear_env()
        messagebox.showinfo( title: "Вихід",  message: "Облікові дані очищено.")
    root.destroy()

def main():  Vlad +1
    root = tk.Tk()
    root.title("Kanter")
    root.state('zoomed')

    app = ConnectView(root)
    app.pack(fill='both', expand=True)

    root.protocol( name: "WM_DELETE_WINDOW", lambda: on_closing(root))
    root.mainloop()

if __name__ == "__main__":
    main()
```

Рис. 4.43 Головний модуль входу у систему “Kanter”

ВИСНОВКИ

У результаті виконання бакалаврської кваліфікаційної роботи було реалізовано повноцінну аналітичну систему Kanter для моніторингу активності розробників у програмних проєктах, що використовують систему контролю версій Git та сервіс GitHub. Система дозволяє автоматизовано збирати статистику про внесок кожного члена команди, візуалізувати ці дані у вигляді графіків та зберігати їх у форматі звіту PDF. Завдяки цьому вона може стати корисним інструментом для тім-лідів, проєктних менеджерів та інших спеціалістів, відповідальних за оцінку ефективності командної роботи.

В ході розробки було використано мову програмування Python, стандартну бібліотеку Tkinter для побудови графічного інтерфейсу, а також ряд зовнішніх бібліотек, зокрема matplotlib, seaborn, pandas, PyGithub та python-dotenv. Особливу увагу було приділено зручності користування — інтерфейс реалізовано у вигляді вкладок, що забезпечує просту навігацію між модулями системи. Користувач може швидко підключитися до репозиторію GitHub, обрати розробника зі списку, переглянути аналітику по його комітах, а також експортувати звіт для подальшого аналізу.

Система Kanter вирішує актуальну проблему прозорої оцінки внеску окремих учасників у командну розробку. Вона надає можливість не лише фіксувати активність у репозиторії, а й бачити динаміку роботи, інтенсивність комітів, обсяги зміненого коду, а також надає візуальні підказки щодо ефективності. Це дозволяє своєчасно виявляти як потенційних лідерів, так і менш активних учасників проєкту, тим самим сприяючи прийняттю обґрунтованих управлінських рішень.

У цілому, реалізований програмний продукт є готовим до використання у реальних проєктах і може бути розширений за рахунок підтримки інших Git-сервісів (GitLab, Bitbucket), інтеграції з базами даних для збереження історії, додавання більш складних метрик та інтерактивної аналітики.

Робота пройшла апробацію. За її результатами було опубліковано наступні

тези доповідей:

1. Беро В.Ф. Забезпечення безпеки інформації в інтелектуальної системи моніторингу внеску розробників у програмні проекти// Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 24.04.24, ДУІКТ, Київ, Україна, с. 445–447

2. Беро В.Ф. Хешування пароля // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в ІКТ», 24.04.24, ДУІКТ, Київ, Україна, с. 441

ПЕРЕЛІК ПОСИЛАНЬ

1. Grayson, J. (2012). Python and Tkinter Programming. Manning Publications.
2. Shipman, J. (2021). Tkinter 8.5 reference: a GUI for Python. Python Software Foundation.
3. Python Software Foundation. Python Standard Library: os module. https://docs.python.org/3/library/os.html
4. Sweigart, A. (2020). Automate the Boring Stuff with Python (2nd Edition).
5. Summerfield, M. (2010). Programming in Python 3: A Complete Introduction to the Python Language. Addison-Wesley.
6. Lutz, M. (2013). Learning Python (5th Edition). O'Reilly Media.
7. Harrison, K. (2021). Python for Data Analysis (2nd Edition). O'Reilly Media.
8. Hunter, J. D., & Droettboom, M. (2022). Matplotlib Documentation. https://matplotlib.org
9. Waskom, M. (2022). Seaborn: Statistical Data Visualization. https://seaborn.pydata.org
10. Rosebrock, A. (2018). GUI Programming with Tkinter. PyImageSearch.
11. Van Rossum, G. (2023). The Python Tutorial – Tkinter section. Python Software Foundation. https://docs.python.org/3/library/tkinter.html
12. Lundh, F. (2001). An Introduction to Tkinter. PythonWare. http://effbot.org/tkinterbook/
13. Chacon, S., & Straub, B. (2014). Pro Git (2nd Edition). Apress. https://git-scm.com/book/en/v2
14. GitHub, Inc. (2023). GitHub REST API v3 Documentation. https://docs.github.com/en/rest
15. GitHub Docs. Authenticating with personal access tokens. https://docs.github.com/en/authentication

16. The Python Packaging Authority. (2023). python-dotenv Documentation. https://pypi.org/project/python-dotenv/

17. Hunter, J. D. et al. (2023). Matplotlib.backends.backend_pdf — Save figures to a multipage PDF file. https://matplotlib.org/stable/users/prev_whats_new/whats_new_1.2.0.html#pdf-backend

18. Python Software Foundation. Built-in Functions - open(). https://docs.python.org/3/library/functions.html#open

19. ReportLab. (2022). Creating PDFs with Python. https://www.reportlab.com/docs/reportlab-userguide.pdf

20. Richardson, L. & Ruby, S. (2007). RESTful Web Services. O'Reilly Media.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка інтелектуальної системи для моніторингу внеску розробників у програмні проєкти

Виконав студент 4 курсу

групи П-41
Беро Владислав Федорович
Керівник роботи

К.ф.-м.н, доц, професор кафедри ІПЗ Садовенко Володимир Сергійович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** підвищення ефективності моніторингу та аналітики за внеском кожного розробника у розробку спільного проекту.
- **Об’єкт дослідження** процес контролю та організації внеску кожного розробника у розробку спільного проекту**.**
- **Предмет дослідження** процес збору та аналізу даних про активність розробників у системах контролю версій для оцінки їхнього індивідуального внеску в командну розробку програмного забезпечення.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Підбір та аналіз науково-технічної літератури.
2. Аналіз та дослідження існуючих аналогів.
3. Огляд та аналіз систем аналітики внеску розробників.
4. Огляд та аналіз засобів реалізації систем аналітики внеску розробників.
5. Проектування архітектури системи аналітики внеску розробників
6. Програмна реалізація та тестування системи аналітики внеску розробників.
7. Оформлення роботи: вступ, висновки, реферат.
8. Розробка демонстраційних матеріалів.
9. Попередній захист роботи.

АНАЛІЗ АНАЛОГІВ

Характеристики/ додаток	Система Kanter	GitHub Insights	GitLab Analytics	Gource	Open Source Contributions Dashboard
Візуалізація активності розробників	+	+	+	+	+
Відстеження змін у файлах (рядки +/-)	+	+	+	-	+
Попередження про неактивних розробників	+	+	+	-	-
Порівняння активності за періодами	+	+	+	-	+
Отримання даних з різних Git-сервісів	+	-	-	+	+
Десктопний додаток	+	-	-	+	-

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

- Користувач (Архітектор/Проджект-менеджер) може підключатися до репозиторія через назву та токен;
- Користувач (Архітектор/Проджект-менеджер) натискаючи на кнопку "Оновити дані", витягує дані з репозиторію та оновлює дані про роботу програмістів та створює список кнопок зі створенням аналітики про працю кожного розробника;
- Користувач (Архітектор/Проджект-менеджер) може сберігати звіти про активність команди проекту на комп'ютер по обраному шляху;
- Користувач (Архітектор/Проджект-менеджер) може вирішити, при закритті системи, зберігати дані про токен та репозиторій чи ні.

Нефункціональні вимоги:

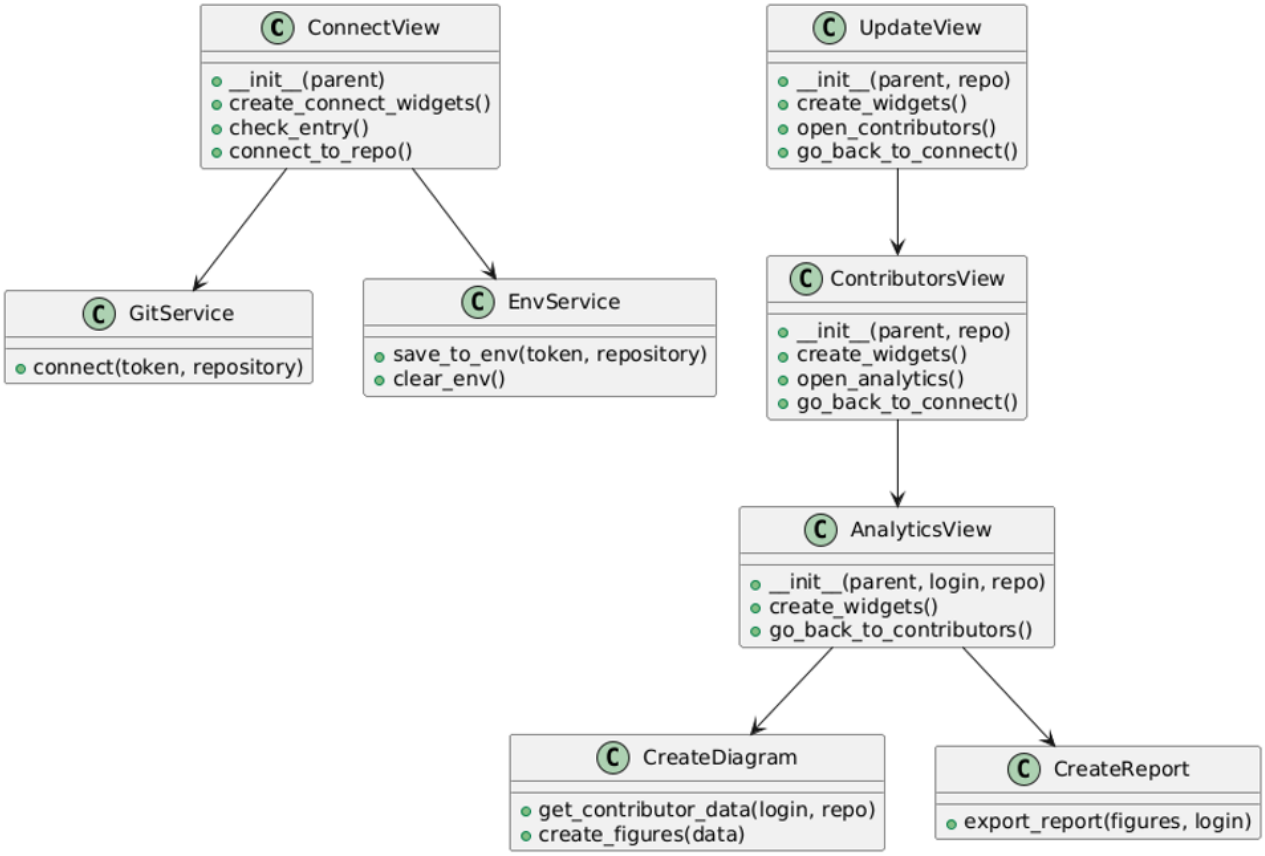
- Система працює на Windows;
- Створення аналітики займає не більше 15 секунд, для кожного розробника;
- Інтерфейс реагує на дії користувача менше ніж за 1 секунду.
- Підключення до Git-репозиторію виконується за допомогою додавання Git токену декількох Git-сервісів: Github, Gitlab тощо;
- Звіти зберігаються у файлі формату .pdf;
- Доступ до репозиторіїв здійснюється через захищене зберігання токенів, які мають зберігатися у .env файлі;
- У разі помилки під час запиту до Git-репозиторію, система висвічує повідомлення про помилку з її описом;
- Інтерфейс системи підтримує українську мову;
- Система написана версії Python 3.11+;
- Система використовує GUI Tkinter;
- Система повідомляє Користувача (Архітектора/Проджект-менеджера) за допомогою вставного вікна про розробників, які не зробили жодної роботи за останній тиждень.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

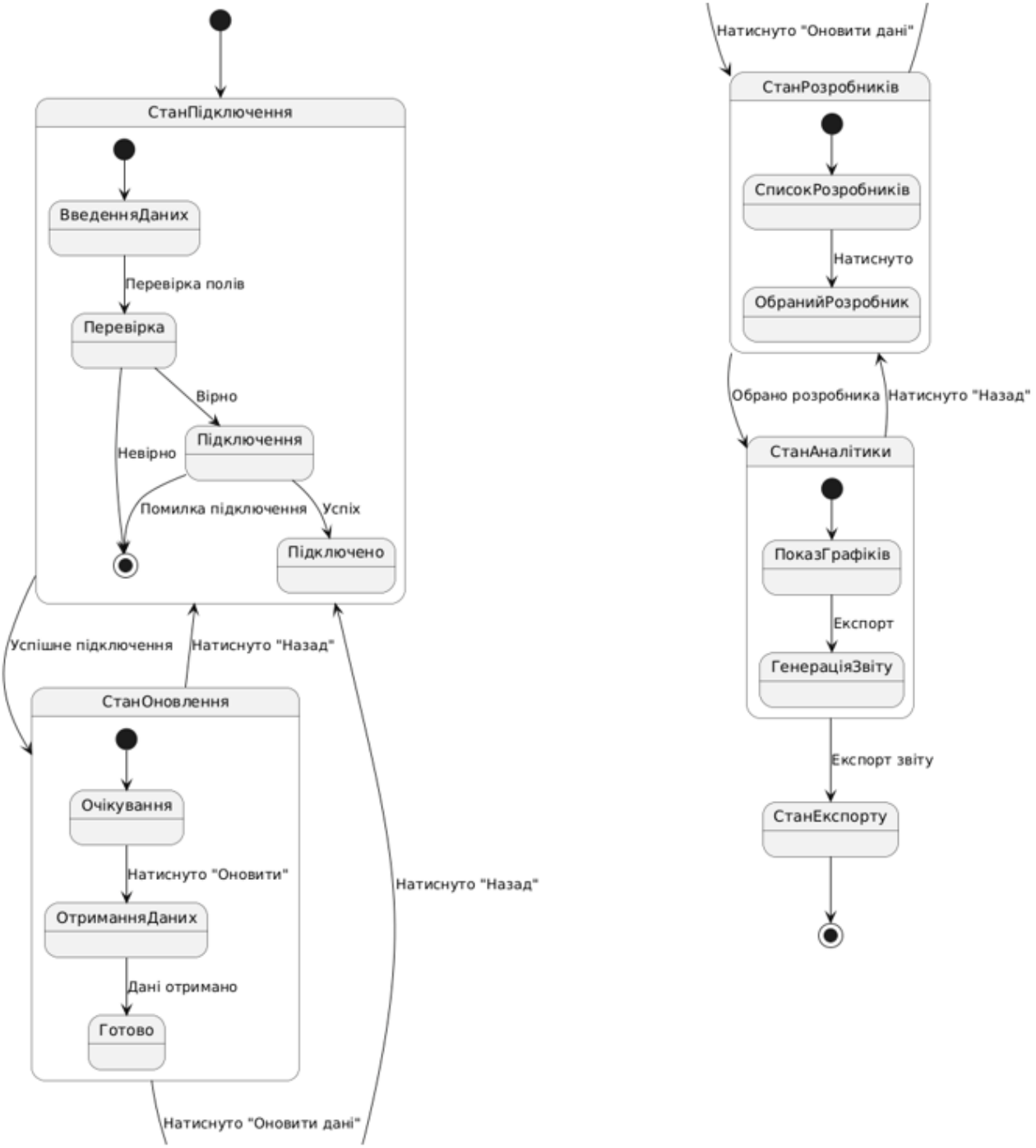
Python: мова програмування
PyGitHub: Бібліотека для отримання даних з GitHub API
Tkinter: GUI
.ENV: Зберігання конфіденційних даних
PyCharm: IDE
Seaborn: Стилізовані графіки для статистичної візуалізації
NumPy: Опрацювання числових даних, масивів і обчислень
Pandas: Збір, фільтрація й обробка табличних даних
Git: Система контролю версій та дані
Matplotlib: Бібліотека для побудови графіків у GUI

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ

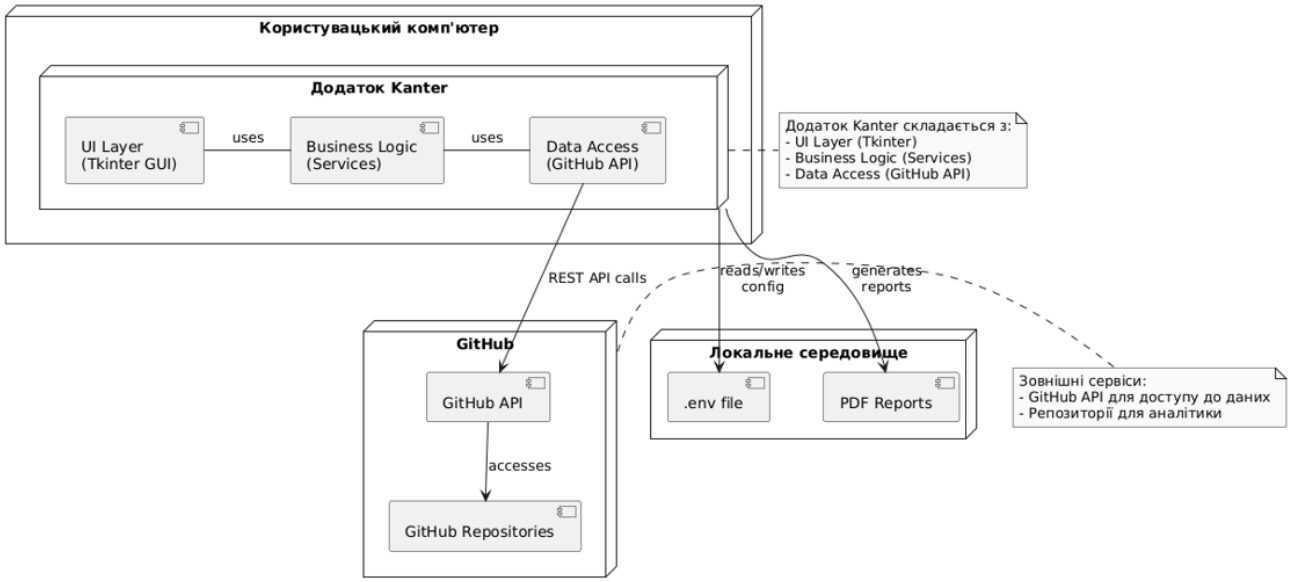
ДІАГРАМА КЛАСІВ



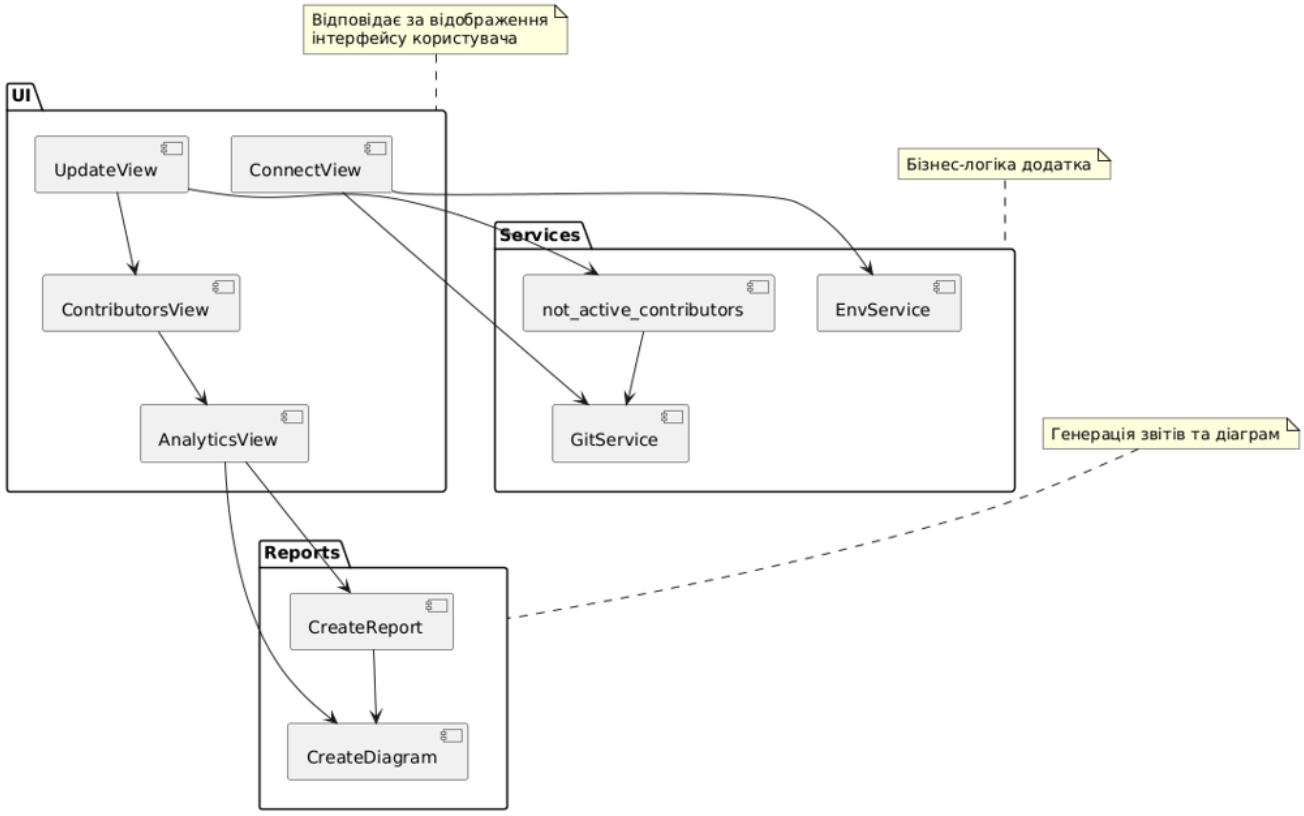
ДІАГРАМА СТАНІВ



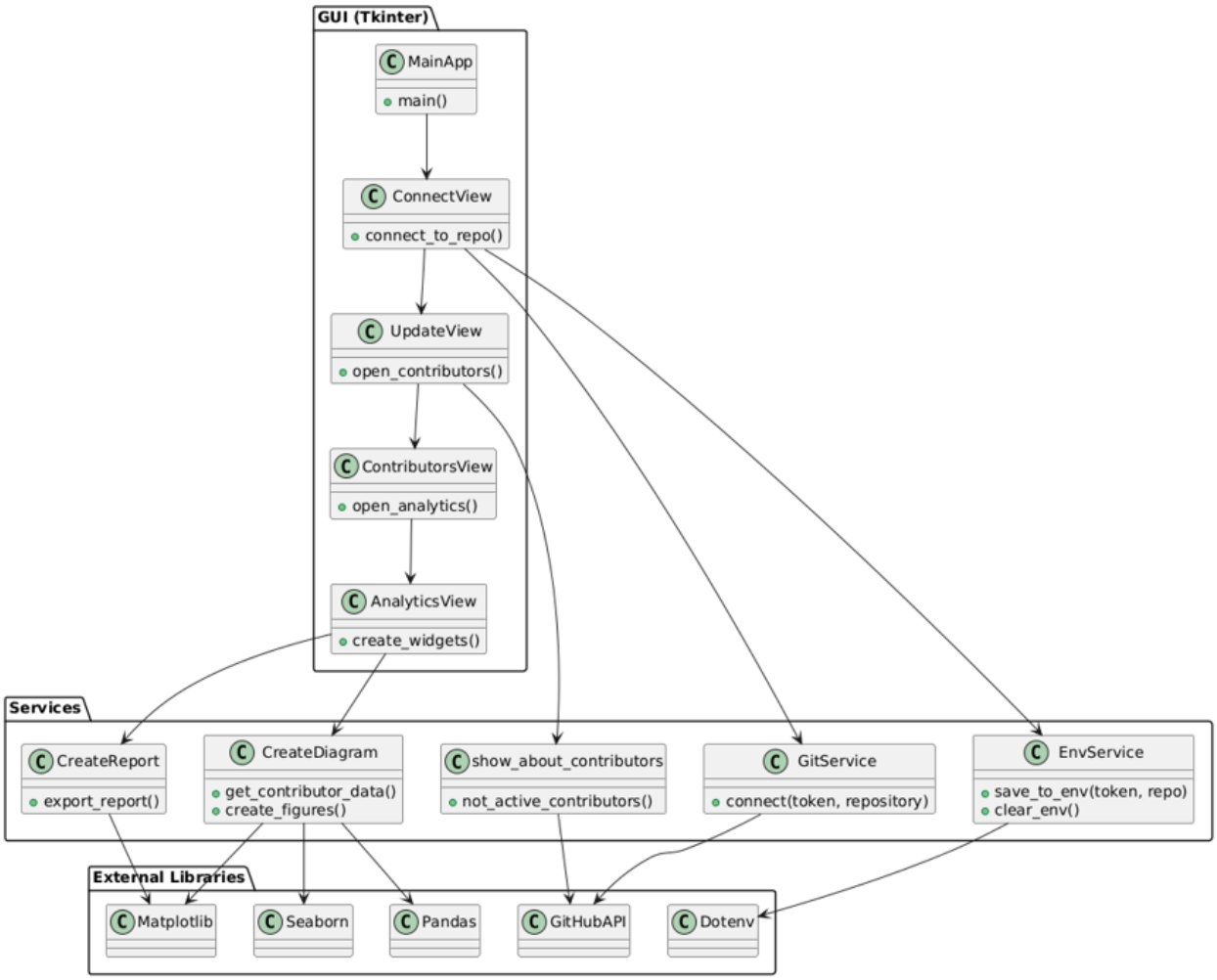
ДІАГРАМА РОЗГОРТАННЯ СИСТЕМИ



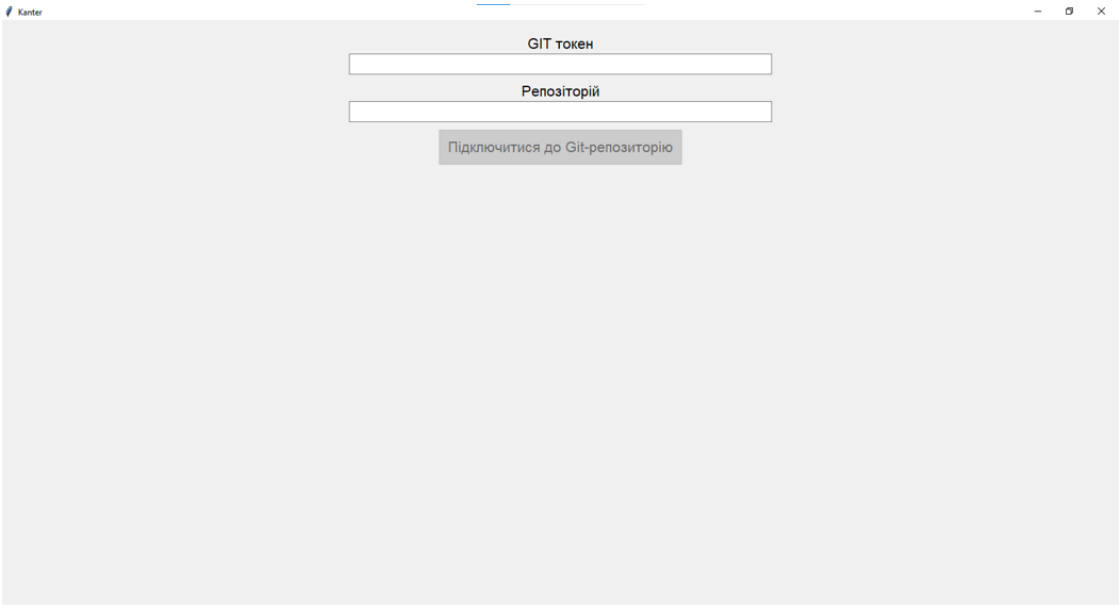
ДІАГРАМА ПАКЕТІВ СИСТЕМИ



ДІАГРАМА АРХІТЕКТУРИ СИСТЕМИ

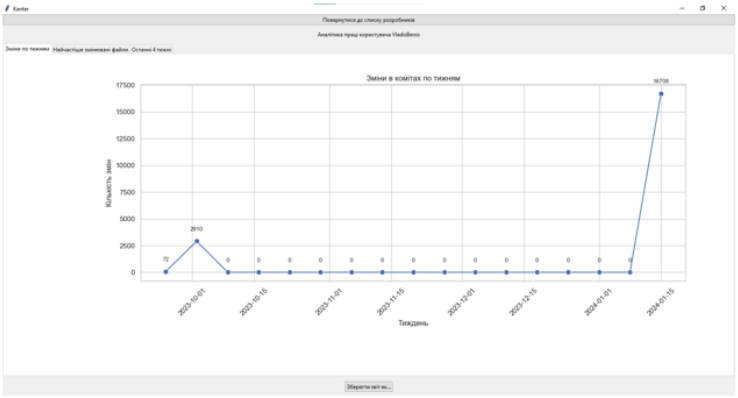


ЕКРАННІ ФОРМИ

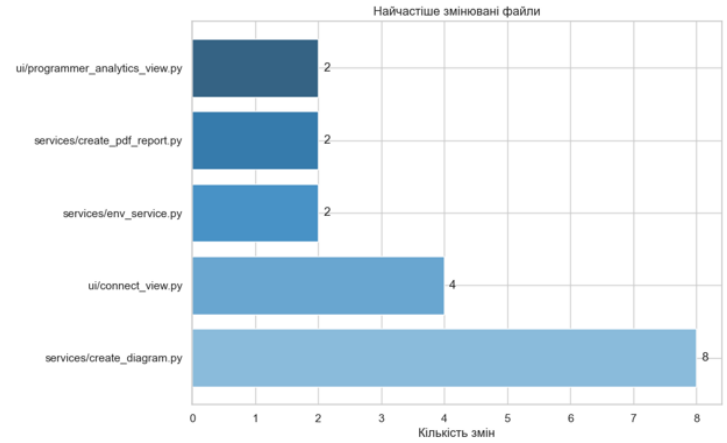


«Головний екран системи з полями для введення назви репозиторію та токену»

ЕКРАННІ ФОРМИ

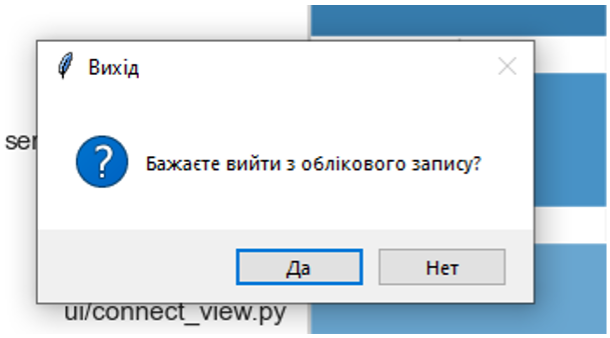


«Екран зі вкладками з аналітикою праці одного з розробників»

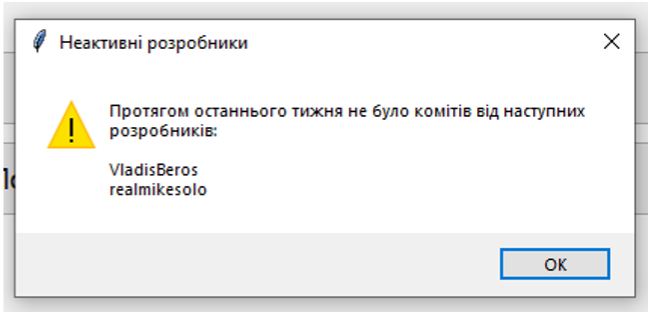


«Аналітика найчастіше змінюваних файлів»

ЕКРАННІ ФОРМИ



«Система питає: виходити з облікового запису?»



«Система повідомляє про неактивних розробників»

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
import tkinter as tk
from tkinter import messagebox
from ui.connect_view import ConnectView
from services.env_service import EnvService

def on_closing(root):
    if messagebox.askyesno("Вихід", "Бажаєте вийти з
облікового запису?"):
        EnvService.clear_env()
        messagebox.showinfo("Вихід", "Облікові дані
очищено.")
    root.destroy()

def main():
    root = tk.Tk()
    root.title("Kanter")
    root.state('zoomed')

    app = ConnectView(root)
    app.pack(fill='both', expand=True)

    root.protocol("WM_DELETE_WINDOW", lambda:
on_closing(root))
    root.mainloop()

if __name__ == "__main__":
 main()

from tkinter import ttk, messagebox
from ui.update_view import UpdateView
from services.git_service import GitService
from services.env_service import EnvService
from dotenv import load_dotenv
import os

class ConnectView(ttk.Frame):
    def __init__(self, parent):
        super().__init__(parent)
        self.parent = parent
        self.create_connect_widgets()

    def create_connect_widgets(self):
        load_dotenv()

        style = ttk.Style()
        style.configure("Large.TButton", font=("Arial",
16))

        label_token = ttk.Label(self, text="GIT токен",
font=("Arial", 16))
        label_token.pack(pady=(20, 0))
        self.token_entry = ttk.Entry(self, width=50,
font=("Arial", 16), show="•")
        self.token_entry.pack()
        self.token_entry.bind("<KeyRelease>",
self.check_entry)

        label_repo = ttk.Label(self, text="Репозіторій",
font=("Arial", 16))
        label_repo.pack(pady=(10, 0))
        self.repo_entry = ttk.Entry(self, width=50,
font=("Arial", 16))
        self.repo_entry.pack()
        self.repo_entry.bind("<KeyRelease>",
self.check_entry)

        repo_name = os.getenv("REPO_NAME", "")
        git_token = os.getenv("GIT_TOKEN", "")
        self.repo_entry.insert(0, repo_name)
        self.token_entry.insert(0, git_token)

        self.connect_button = ttk.Button(self,
                                         text="Підключитися до Git-
репозиторію",
```

```
                                style="Large.TButton",

  command=self.connect_to_repo)
        self.connect_button.pack(pady=(10, 0))

    def check_entry(self, event):
        if self.token_entry.get() and self.repo_entry.get():
            self.connect_button["state"] = "normal"
        else:
            self.connect_button["state"] = "disabled"

    def connect_to_repo(self):
        token = self.token_entry.get()
        repository = self.repo_entry.get()

        connect_result = GitService.connect(token,
  repository)

        if connect_result['success']:
            messagebox.showinfo("Успіх",
  f"{connect_result['message']}")
            EnvService.save_to_env(token, repository)
            repo = connect_result['repo']

            self.destroy()
            update_view = UpdateView(self.parent, repo)
            update_view.pack(fill='both', expand=True)
        else:
         messagebox.showerror("Помилка",
f"{connect_result['message']}")

  import tkinter as tk
  from tkinter import ttk, messagebox
  from ui.programmer_analytics_view import
  AnalyticsView

  class ContributorsView(ttk.Frame):
    def __init__(self, parent, repo):
        super().__init__(parent)
        self.parent = parent
        self.repo = repo
        self.create_widgets()

    def create_widgets(self):
        self.canvas = tk.Canvas(self, borderwidth=5)
        self.scrollbar = ttk.Scrollbar(self,
orient="vertical", command=self.canvas.yview)
        self.scrollable_frame = ttk.Frame(self.canvas)

        back_button = ttk.Button(self,
text="Повернутися до введення токену та
репозиторію", command=self.go_back_to_connect)
        back_button.pack(fill='x')

        self.scrollable_frame.bind("<Configure>", lambda
e:
self.canvas.configure(scrollregion=self.canvas.bbox("al
l")))
        self.canvas.create_window((0, 0),
window=self.scrollable_frame, anchor="nw")

self.canvas.configure(yscrollcommand=self.scrollbar.s
et)

        self.scrollbar.pack(side="right", fill="y")
        self.canvas.pack(side="left", fill="both",
expand=True)

        contributors = list(self.repo.get_contributors())
        logins = [c.login for c in contributors]

        for login in logins:
            btn = ttk.Button(
                self.scrollable_frame,
                text=login,
                command=lambda l=login:
self.open_analytics(l),
                padding=(10, 10),
                width=20
            )
            btn.pack(pady=5, fill='x', expand=True)

    def open_analytics(self, login):
        messagebox.showinfo(f"Обран {login}",
```

```
"Будується аналітика...")
        self.destroy()
        programmer_analytics_view =
AnalyticsView(self.parent, login, self.repo)
        programmer_analytics_view.pack(fill='both',
expand=True)

    def go_back_to_connect(self):
        from ui.connect_view import ConnectView
        self.destroy()
        connect_view = ConnectView(self.parent)
      connect_view.pack(fill='both', expand=True)

from tkinter import ttk
from services.create_diagram import CreateDiagram
from matplotlib.backends.backend_tkagg import
FigureCanvasTkAgg
from services.create_pdf_report import CreateReport

class AnalyticsView(ttk.Frame):
    def __init__(self, parent, login, repo):
        super().__init__(parent)
        self.parent = parent
        self.login = login
        self.repo = repo
        self.create_widgets()

    def create_widgets(self):
        back_button = ttk.Button(self,
text="Повернутися до списку розробників",

command=self.go_back_to_contributors)
        back_button.pack(fill='x')

        label = ttk.Label(self, text=f"Аналітика праці
користувача {self.login}")
        label.pack(pady=10)

        data =
CreateDiagram.get_contributor_data(self.login,
self.repo)
        figures = CreateDiagram.create_figures(data)

        notebook = ttk.Notebook(self)
        notebook.pack(fill='both', expand=True)

        for title, fig in figures.items():
            frame = ttk.Frame(notebook)
            canvas = FigureCanvasTkAgg(fig,
master=frame)
            canvas.draw()
            canvas.get_tk_widget().pack(fill='both',
expand=True)
            notebook.add(frame, text=title)

        bottom_frame = ttk.Frame(self)
        bottom_frame.pack(fill='x', side='bottom',
pady=10)
        export_button = ttk.Button(bottom_frame,
text="Зберегти звіт як...",
                                   command=lambda:
CreateReport.export_report(figures, self.login))
        export_button.pack()

    def go_back_to_contributors(self):
        from ui.contributors_view import
ContributorsView
        self.destroy()
        contributors_view =
ContributorsView(self.parent, self.repo)
      contributors_view.pack(fill='both', expand=True)

from tkinter import ttk, messagebox
from ui.contributors_view import ContributorsView
from services.show_about_contributors import
not_active_contributors

class UpdateView(ttk.Frame):
    def __init__(self, parent, repo):
        super().__init__(parent)
        self.parent = parent
        self.repo = repo
        self.create_widgets()
```

```
    def create_widgets(self):
        style = ttk.Style()
        style.configure("Large.TButton", font=("Arial",
14), padding=10)
        button_frame = ttk.Frame(self)
        button_frame.place(relx=0.5, rely=0.5,
anchor='center')

        update_data_button = ttk.Button(button_frame,
text="Оновити дані",
                                        style="Large.TButton",
command=self.open_contributors)
        update_data_button.grid(row=0, column=0,
sticky='ew', pady=5)

        back_button = ttk.Button(button_frame,
text="Повернутися до введення токену та
репозиторію",
                                 style="Large.TButton",
command=self.go_back_to_connect)
        back_button.grid(row=1, column=0, sticky='ew',
pady=5)

        button_frame.grid_rowconfigure(0, weight=1)
        button_frame.grid_rowconfigure(1, weight=1)
        button_frame.grid_columnconfigure(0, weight=1)

    def open_contributors(self):
        messagebox.showinfo("Оновлення", "Дані
оновлюються...")
        not_active_contributors(self.repo)

        self.destroy()
        contributors_view =
ContributorsView(self.parent, self.repo)
        contributors_view.pack(fill='both', expand=True)

    def go_back_to_connect(self):
        from ui.connect_view import ConnectView

        self.destroy()
        connect_view = ConnectView(self.parent)
        connect_view.pack(fill='both', expand=True)

from github import Repository
import pandas as pd
import seaborn as sns
import matplotlib.dates as mdates
from matplotlib.figure import Figure
import numpy as np

class CreateDiagram:
    @staticmethod
    def get_contributor_data(login, repo:
Repository.Repository):
        data = {
            'total_commits': 0,
            'total_lines_added': 0,
            'total_lines_deleted': 0,
            'total_files_changed': 0,
            'avg_lines_per_commit': 0,
            'first_commit_date': None,
            'last_commit_date': None,
            'commit_frequency': None,
            'most_changed_files': {},
            'commits_by_date': []
        }

        seen_shas = set()
        all_commits = []

        branches = list(repo.get_branches())

        for branch in branches:
            try:
                commits = repo.get_commits(author=login,
sha=branch.name)
                for commit in commits:
                    if commit.sha not in seen_shas:
                        seen_shas.add(commit.sha)
                        all_commits.append(commit)
            except Exception as e:
                print(f"[ERROR] Помилка при обробці
гілки '{branch.name}': {e}")
```

```
        if not all_commits:
            return data

        all_commits.sort(key=lambda c:
c.commit.author.date, reverse=True)

        data['total_commits'] = len(all_commits)
        data['first_commit_date'] = all_commits[-
1].commit.author.date
        data['last_commit_date'] =
all_commits[0].commit.author.date

        for commit in all_commits:
            stats = commit.stats
            commit_date =
commit.commit.author.date.date()

            data['commits_by_date'].append({
                'date': commit_date,
                'additions': stats.additions,
                'deletions': stats.deletions,
                'changes': stats.additions + stats.deletions
            })

            data['total_lines_added'] += stats.additions
            data['total_lines_deleted'] += stats.deletions

            for file in commit.files:
                filename = file.filename
                if filename not in data['most_changed_files']:
                    data['most_changed_files'][filename] = 0
                data['most_changed_files'][filename] += 1
                data['total_files_changed'] += 1

        data['avg_lines_per_commit'] = round(
            (data['total_lines_added'] +
data['total_lines_deleted']) / data['total_commits'], 2
        )

        days_active = (data['last_commit_date'] -
data['first_commit_date']).days
        data['commit_frequency'] =
round(data['total_commits'] / max(1, days_active), 2)

        data['most_changed_files'] = dict(
            sorted(data['most_changed_files'].items(),
key=lambda item: item[1], reverse=True)[:5]
        )

        return data

    @staticmethod
    def create_figures(data):
        sns.set_theme(style="whitegrid")
        figures = {}

        if not data.get("commits_by_date"):
            return figures

        df = pd.DataFrame(data['commits_by_date'])
        df['date'] = pd.to_datetime(df['date'])

        df_weekly = df.groupby(pd.Grouper(key='date',
freq='W-MON')).agg({
            'additions': 'sum',
            'deletions': 'sum',
            'changes': 'sum'
        }).reset_index()

        first_diagram = Figure(figsize=(5, 4))
        first_axis = first_diagram.subplots()

        sns.lineplot(data=df_weekly, x='date',
y='changes', marker='o', markersize=8, ax=first_axis)

        for i, row in df_weekly.iterrows():
            first_axis.text(row['date'], row['changes'] +
max(df_weekly['changes']) * 0.05,
                f"{int(row['changes'])}", ha='center',
va='bottom', fontsize=9
            )

        first_axis.set_title('Зміни в комітах по тижням')
```

```
        first_axis.set_xlabel('Тиждень')
        first_axis.set_ylabel('Кількість змін')

first_axis.xaxis.set_major_formatter(mdates.DateForm
atter('%Y-%m-%d'))
        first_axis.tick_params(axis='x', rotation=45)
        first_diagram.tight_layout()
        figures['Зміни по тижням'] = first_diagram

        if data['most_changed_files']:
            second_diagram = Figure(figsize=(5, 4))
            second_axis = second_diagram.subplots()

            files = list(data['most_changed_files'].keys())
            changes =
list(data['most_changed_files'].values())

            bars = second_axis.barh(files, changes,
color=sns.color_palette("Blues_d"))
            second_axis.set_title('Найчастіше змінювані
файли')
            second_axis.set_xlabel('Кількість змін')

            for bar in bars:
                width = bar.get_width()
                second_axis.text(width + max(changes) *
0.01, bar.get_y() + bar.get_height() / 2,
                         f'{int(width)}', va='center')

            second_diagram.tight_layout()
            figures['Найчастіше змінювані файли'] =
second_diagram
        else:
            second_diagram = Figure(figsize=(5, 4))
            second_axis = second_diagram.subplots()
            second_axis.text(0.5, 0.5, 'Немає даних про
зміни файлів', ha='center', va='center')
            figures['Найчастіше змінювані файли'] =
second_diagram

        third_diagram = Figure(figsize=(5, 4))
        third_axis = third_diagram.subplots()

        last_4_weeks = df[df['date'] >=
(pd.to_datetime('today') - pd.Timedelta(weeks=4))]

        if not last_4_weeks.empty:
            weekly_data = last_4_weeks.groupby(
                pd.Grouper(key='date', freq='W-MON')
            ).agg({
                'additions': 'sum',
                'deletions': 'sum'
            }).reset_index()

            weekly_data['week_str'] =
weekly_data['date'].dt.strftime('%Y-%m-%d')

            bar_width = 0.35
            x = np.arange(len(weekly_data))

            third_axis.bar(x - bar_width / 2,
weekly_data['additions'], width=bar_width,
color='green', label='Додані рядки')
            third_axis.bar(x + bar_width / 2,
weekly_data['deletions'], width=bar_width, color='red',
label='Видалені рядки')

            third_axis.set_title('Зміни за останні 4 тижні')
            third_axis.set_xticks(x)

third_axis.set_xticklabels(weekly_data['week_str'],
rotation=45)
            third_axis.legend()
        else:
            third_axis.text(0.5, 0.5, 'Немає даних за
останні 4 тижні', ha='center', va='center')

        third_diagram.tight_layout()
        figures['Останні 4 тижні'] = third_diagram

    return figures

from tkinter import ttk, filedialog, messagebox
from matplotlib.backends.backend_pdf import
```

```
PdfPages

class CreateReport(ttk.Frame):
    @staticmethod
    def export_report(figures, login):
        file_path =
filedialog.asksaveasfilename(defaultextension=".pdf",
filetypes=[("PDF files", "*.pdf")], title="Зберегти звіт
як PDF")
        if not file_path:
            return

        with PdfPages(file_path) as pdf:
            from matplotlib import pyplot as plt

            fig, ax = plt.subplots(figsize=(8.27, 11.69))
            ax.axis('off')
            ax.text(0.5, 0.9, f"Аналіз роботи розробника
{login}", fontsize=16, ha='center', va='top',
weight='bold')
            pdf.savefig(fig)
            plt.close(fig)

            for fig in figures.values():
                pdf.savefig(fig)

        messagebox.showinfo("Успіх", f"Звіт про
{login} успішно збережено.")

import datetime
from tkinter import messagebox

def not_active_contributors(repo):
    contributors = list(repo.get_contributors())
    inactive_logins = []
    one_week_ago = datetime.datetime.utcnow() -
datetime.timedelta(days=7)

    for contributor in contributors:
        try:
            commits =
list(repo.get_commits(author=contributor.login,
since=one_week_ago))
            if not commits:
                inactive_logins.append(contributor.login)
        except Exception as e:
            print(f"[ERROR] Помилка при обробці
користувача {contributor.login}: {e}")

    if inactive_logins:
        inactive_list = "\n".join(inactive_logins)
        messagebox.showwarning(
            "Неактивні розробники",
            f"Протягом останнього тижня не було
комітів від наступних
розробників:\n\n{inactive_list}"
        )

from github import Github
from functools import lru_cache

class GitService:
    @staticmethod
    @lru_cache(maxsize=10)
    def connect(token, repository):
        connect_result = {'message': "З'єднання
успішне!", 'success': False, 'repo': None}

        try:
            g = Github(f"{token}")
            repo = g.get_repo(f"{repository}")
            connect_result['success'] = True
            connect_result['repo'] = repo
        except Exception as e:
            connect_result['message'] = f"[ERROR]
Невідома помилка: {str(e)}"

        return connect_result

from dotenv import set_key, unset_key
import os

ENV_PATH = ".env"
```

```
class EnvService:
    @staticmethod
    def save_to_env(token, repository):
        if not os.path.exists(ENV_PATH):
            with open(ENV_PATH, 'w'):
                pass

        set_key(ENV_PATH, "GIT_TOKEN", f'{token}')
        set_key(ENV_PATH, "REPO_NAME",
f'{repository}')
        os.environ["GIT_TOKEN"] = token
        os.environ["REPO_NAME"] = repository

    @staticmethod
    def clear_env():
        if os.path.exists(ENV_PATH):
            unset_key(ENV_PATH, "GIT_TOKEN")
            unset_key(ENV_PATH, "REPO_NAME")
        os.environ.pop("GIT_TOKEN", None)
     os.environ.pop("REPO_NAME", None)
```