

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ
АЛГОРИТМІВ КОНТРОЛЮ ВЕРСІЙ ДОКУМЕНТІВ
RESEARCH AND SOFTWARE IMPLEMENTATION OF
ALGORITHMS FOR DOCUMENT VERSION CONTROL

Виконав(ла): студент(ка) 4 курсу, групи КНТ-112

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

АРТАМОШИН Є.В.

(ПРИЗВИЩЕ та ініціали)

Керівник ГОФМАН Є.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

АРТАМОШИНА Єгора Володимировича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація алгоритмів контролю версій документів. Research and Software Implementation of Algorithms for Document Version Control

– керівник проєкту (роботи) к.т.н., доцент, ГОФМАН Євгеній Олександрович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “07” квітня 2026 року № 139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Розробка програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГОФМАН Є.О., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій	2 тиждень	Розділ 2
	розробки.		
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження	5 тиждень	Розділ 4
	програмного забезпечення.		
7	Оформлення пояснювальної записки та документів	6 тиждень	Додатки
	до неї.		
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Єгор АРТАМОШИН
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Євгеній ГОФМАН
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до випускної кваліфікаційної роботи бакалавра:
100 с., 7 табл., 38 рис., 3 дод., 10 джерел.

СИСТЕМА КОНТРОЛЮ ВЕРСІЙ, КОНТРОЛЬ ВЕРСІЙ
ДОКУМЕНТІВ, GIT, РІЗНИЦЯ ВЕРСІЙ, ХЕШУВАННЯ, NODE.JS, CLI-
ЗАСТОСУНОК, АЛГОРИТМ МАЙЕРСА.

Об'єкт дослідження – процес інженерії програмного забезпечення для контролю версій електронних документів, управління історією версій та відновлення попередніх станів даних.

Предмет дослідження – алгоритми, методи та програмні інструменти для реалізації систем контролю версій документів, механізми збереження об'єктів, визначення відмінностей між версіями файлів та відновлення станів документів.

Мета роботи – підвищення ефективності систем контролю версій, зниження мінімального рівня кваліфікаційних вимог до користувача та мінімізація кількості помилок при роботі GIT-подібних алгоритмів.

Матеріали, методи та технічні засоби: використано методи побудови модульної архітектури програм, алгоритми хешування та стиснення даних, механізми роботи з файловою системою та алгоритми порівняння текстових документів. Для реалізації системи використано мову програмування TypeScript та платформу Node.js. Для реалізації алгоритму різниці версій використано бібліотеку diff, яка базується на алгоритмі Майерса. Стиснення об'єктів реалізовано за допомогою бібліотеки zlib. Розробка відбувалась у середовищі WebStorm на персональному комп'ютері з операційною системою Windows 10.

Результати. Розроблено систему контролю версій «Kai», яка реалізує фундаментальні механізми Git-подібних систем контролю версій. Система

підтримує створення локального репозиторію, додавання файлів до відстеження, створення комітів, перегляд історії змін, визначення різниці між версіями документів та відновлення попередніх станів файлів. Реалізовано об'єктне сховище із використанням хешування SHA-256 та стиснення даних. Алгоритм різниці версій підтримує визначення змін як на рівні рядків, так і на рівні окремих слів із генерацією HTML-представлення результатів порівняння.

Висновки. В результаті виконання роботи було досліджено основні алгоритми сучасних систем контролю версій та реалізовано спрощену систему контролю версій документів. Розроблена система дозволяє досліджувати принципи роботи Git-подібних систем, демонструє механізми зберігання необхідних даних та метаданих, коміт-моделі та алгоритми визначення різниці версій та може використовуватись у навчальних цілях.

Галузь використання – навчання принципам роботи систем контролю версій, дослідження алгоритмів збереження та порівняння документів, демонстрація внутрішньої будови Git-подібних систем.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 100 pages, 7 tables, 38 figures, 3 appendixes, 10 sources.

VERSION CONTROL SYSTEM, DOCUMENT VERSION CONTROL, GIT, VERSION DIFFERENCE, HASHING, NODE.JS, CLI APPLICATION, MYERS ALGORITHM.

Research object – the software engineering process for version control of electronic documents, management of version history, and restoration of previous data states.

Research subject – algorithms, methods, and software tools for implementing document version control systems, mechanisms for object storage, determining differences between file versions, and restoring document states.

The purpose of the work – increasing the efficiency and reducing the minimum level of qualification requirements for users of version control systems, minimizing the number of errors when working with Git-like algorithms.

Materials, methods, and technical means: methods of modular software architecture design, data hashing and compression algorithms, file system interaction mechanisms, and text document comparison algorithms were used. The system was implemented using the TypeScript programming language and the Node.js platform. The diff library, based on the Myers algorithm, was used to implement the version difference algorithm. Object compression was implemented using the zlib library. Development was carried out in the WebStorm environment on a personal computer running the Windows 10 operating system.

Results. The “Kai” version control system was developed, implementing the fundamental mechanisms of Git-like version control systems. The system supports local repository creation, adding files for tracking, creating commits, viewing change history, determining differences between document versions, and restoring

previous file states. An object storage mechanism using SHA-256 hashing and data compression was implemented. The version difference algorithm supports detecting changes both at the line level and at the individual word level with generation of an HTML representation of comparison results.

Conclusions. As a result of the work, the main algorithms of modern version control systems were studied and a simplified document version control system was implemented. The developed system allows studying the principles of operation of Git-like systems, demonstrates mechanisms for storing required data and metadata, commit models, and version difference algorithms, and can be used for educational purposes.

Field of application – teaching the principles of version control systems, researching algorithms for document storage and comparison, and demonstrating the internal structure of Git-like systems.

ЗМІСТ

	С.
Вступ.....	11
1 Аналіз предметної області.....	14
1.1 Короткий опис досліджуваної предметної області	14
1.2 Огляд існуючих рішень проблеми.....	15
1.2.1 Система Git	15
1.2.2 Система Apache Subversion.....	16
1.2.3 Система Perforce.....	16
1.2.4 Сервіси на основі систем контролю версій	17
1.2.5 Порівняльна таблиця існуючих програмних продуктів.....	17
1.3 Опис основних вимог до програми	18
1.3.1 Визначення функціональних вимог	18
1.3.2 Нефункціональні вимоги.....	19
1.4 Постановка завдання роботи.....	20
1.5 Висновки за розділом 1	21
2 Розробка архітектури програми.....	23
2.1 Структура системи	23
2.1.1 Основні компоненти системи	23
2.1.2 Взаємодія компонентів системи	23
2.2 Функціонування системи	25
2.2.1 Основні процеси системи.....	25
2.2.2 Логіка роботи системи.....	26
2.3 Вибір мови програмування та інструментарію для розробки програми.....	27
2.3.1 Опис переваг та обмежень вибраної мови.....	28
2.3.2 Середовище розробки.....	29
2.3.2.1 Обґрунтування вибору та опис його можливостей	29
2.4 Вимоги до технічних і програмних засобів.....	30
2.4.1 Необхідні технічні засобів для розробки та експлуатації.....	30
2.4.2 Вимоги до апаратного та програмного забезпечення	31

2.5 Висновки за розділом 2	31
3 Розробка програми	33
3.1 Архітектура системи.....	33
3.1.1 Структурна схема системи.....	34
3.2 Реалізація алгоритмів системи.....	34
3.2.1 Алгоритм створення репозиторію	35
3.2.2 Алгоритм додавання файлів.....	36
3.2.3 Алгоритм створення коміту	37
3.2.4 Алгоритм відновлення версій	39
3.2.5 Алгоритм визначення змін.....	40
3.3 Структура зберігання даних.....	44
3.3.1 Структура службової директорії	44
3.3.2 Формат збереження об'єктів.....	45
3.3.3 Файл index.....	45
3.3.4 Формат збереження комітів	46
3.3.5 Файл HEAD.....	46
3.3.6 Порівняння структури зберігання Kai та Git.....	47
3.4 Обробка та валідація помилок	47
3.4.1 Централізована обробка помилок	48
3.4.2 Валідація вхідних даних.....	48
3.4.3 Оптимізація роботи з файлами	49
3.5 Висновки за розділом 3	49
4 Експлуатація, тестування та експериментальне дослідження програми	51
4.1 Опис застосування програми	51
4.1.1 Огляд планової цільової аудиторії	52
4.1.2 Розподіл сценаріїв використання програми.....	53
4.2 Інструкція по роботі з програмою	54
4.2.1 Встановлення програми.....	54
4.2.1.1 Встановлення Node.js	55
4.2.1.2 Встановлення системи Kai.....	55

	10
4.2.2 Ініціалізація репозиторію	55
4.2.3 Додавання файлів та створення комітів	56
4.2.4 Перегляд історії комітів	57
4.2.5 Порівняння версій файлів	58
4.2.6 Відновлення версій	58
4.3 Функціональне тестування системи.....	59
4.3.1 Тестування команд системи.....	59
4.3.2 Тестування обробки помилок	61
4.3.3 Аналіз результатів тестування.....	62
4.3.4 Аналіз продуктивності системи.....	63
4.4 Рекомендації щодо використання	64
4.5 Плани з розвитку	65
Висновки	67
Перелік джерел посилання	69
Додаток А Технічне завдання	70
Додаток Б Текст програми	77
Додаток В Слайди презентації.....	94

ВСТУП

Сучасний світ впевнено крокує в епоху тотальної домінації інформаційних технологій. Світовий стрибок популярності ІТ-сектору спричинений епідемією 2019 року та появою масового штучного інтелекту в 2022 році, додав ще більше популярності в, і без того, популярну та перспективну сферу. Зараз важко уявити будь-яку сферу діяльності, де б люди не використовували комп'ютер або інші обчислювальні машини. Їх використання передбачає створення певних застосунків, на розробку яких зазвичай витрачається багато часу. До того ж, в процесі розробки часто бувають моменти, коли програмісти припускаються помилок, і відсутність змоги швидко вирішити проблему, або просто повернутися до останньої робочої версії, може бути фатальною для продукту, компанії, самого програміста тощо. Тому невід'ємним помічником будь-якого розробника є система контролю версій.

Окрім можливості керувати історією змін, також дуже важлива складова таких систем – можливість працювати командою над одним проєктом, над одними файлами, бачити хто які зміни робить та об'єднувати їх за необхідності. Весь цей функціонал вже давно присутній в популярних сервісах, система дуже надійна і відлагоджена.

Парадокс в тому, що за всієї популярності та поширеності систем контролю версій, люди нерідко взагалі не знають як ними правильно користуватись, і дуже мала кількість людей, яка знає як вони працюють зсередини. В мережі інтернет насправді досить мало чіткої та послідовної інформації щодо принципу роботи алгоритмів цих систем. Часто мають місце лише інструкції практичного використання таких систем, хоча розуміння їхньої роботи має як теоретичне, так і практичне значення. По-перше, знання даних алгоритмів дозволяє використовувати існуючі системи, значно краще розуміючи їхню поведінку в складних ситуаціях. По-друге, дослідження таких невеликих, і в той же час складних і надійних систем, сприяє значному

розвитку навичок розробки складних та органічних архітектурних рішень, оскільки такі системи чудово балансують між оптимізацією та складними алгоритмічними рішеннями.

Об'єктом дослідження є процес контролю версій документів, що включає збереження змін, управління історією версій та відновлення попередніх станів даних.

Предметом дослідження є алгоритми контролю версій документів.

Мета роботи – дослідження алгоритмів контролю версій документів та їх програмна реалізація у вигляді спрощеної системи з використанням середовища Node.js.

Для досягнення поставленої мети в роботі вирішувалися такі завдання:

- проаналізувати сучасні методи та інструменти контролю версій документів;
- розробити структуру та принцип роботи спрощеної системи контролю версій (яка все ж виконує основні функції таких систем);
- програмно реалізувати функціонал збереження змін, керування історією та перегляд різниці між версіями;
- експериментально перевірити працездатність системи, його швидкодію, безпеку та зручність використання.

Методика дослідження включає аналіз наукових джерел та існуючих систем контролю версій, таких як Git, Apache Subversion та Perforce Helix Core, вивчення алгоритмів визначення різниці між версіями, зокрема підходів на основі пошуку спільної підпоследовності. У роботі використовується об'єктно-орієнтоване програмування та середовище виконання Node.js для реалізації спрощеної системи контролю версій. Тестування здійснюється шляхом перевірки коректності роботи алгоритмів, оцінки продуктивності та аналізу використання пам'яті на практичних прикладах.

Розроблена система може бути використана як навчальний інструмент для вивчення принципів роботи систем контролю версій, а також як базова модель для подальших досліджень і розширення функціональності.

Результати роботи мають практичну цінність, оскільки дозволяють глибше зрозуміти алгоритмічні основи збереження змін, оцінити ефективність різних підходів та застосовувати ці знання при розробленні програмних систем.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Короткий опис досліджуваної предметної області

Контроль версій документів – це процес управління змінами у файлах, який передбачає збереження історії редагувань, можливість відстеження змін та відновлення попередніх версій. Такий підхід найчастіше використовується в розробці програмного забезпечення, оскільки передбачає велику кількість змін в багатьох файлах, де помилка та неможливість повернутися до робочої версії може бути фатальною для продукту. Крім того контроль версій також важливий та зручний у системах спільної роботи, де декілька користувачів можуть одночасно вносити зміни до одного ресурсу. Найпростіший спосіб контролю версій реалізується шляхом створення копій файлів, але це часто призводить до дублювання даних, втрати історії змін або труднощів у відстеженні актуальної версії документа.

Основні поняття, пов'язані з контролем версій, включають:

- версія – зафіксований стан файлу в певний момент часу;
- коміт (англ. commit) – фіксація стану проєкту;
- різниця (англ. diff) – набір змін між двома версіями документа;
- гілка (англ. branch) – незалежна лінія розвитку файлу,
- злиття (англ. merge) – об'єднання змін з різних гілок.

Сучасні системи контролю версій використовують складні алгоритми обробки та зберігання змін, що дозволяє досить ефективно вести та контролювати проєкт. Зокрема, окрім базового фундаменту контролю версій такі системи використовують більш складні алгоритми пошуку підпоследовностей, які мінімізують витрати часу та обчислювальних потужностей для представлення змін; а також просунуті підходи до кешування даних, завдяки чому можна значно заощадити пам'ять накопичувача.

Серед сучасних тенденцій у цій сфері можна виділити оптимізацію алгоритмів порівняння змін, а також інтеграцію з хмарними сервісами для забезпечення спільної роботи. Крім того, популярність таких систем сильно

росте, особливо на фоні розвитку ІТ-сектору, зумовленому епідемією коронавірусу та розвитком ШІ. Це зумовлює актуальність дослідження алгоритмів контролю версій та їх програмної реалізації для кращого розуміння принципів їх роботи, що має значно полегшити практичне використання таких систем.

1.2 Огляд існуючих рішень проблеми

Розглянемо три системи контролю версій: Git (близько 94% всієї індустрії) Apache Subversion (близько 5%) та Perforce Helix Core (далі Perforce) (<1%). Не дивлячись на тотальну домінацію Git, який вважається еталоном індустрії, інші системи також цікаві з точки зору роботи їхніх алгоритмів, які демонструють зовсім інші підходи до контролю версій, а також, в окремих випадках, демонструють кращі за Git показники.

1.2.1 Система Git

Git – це розподілена система контролю версій, що зберігає повну історію змін на кожному пристрої користувача. Основною особливістю Git є збереження змін у вигляді знімків (англ. snapshots), де кожен коміт представляє стан усіх файлів у певний момент часу [1].

При такому підході важливо приділити достатньо уваги механізмам стиснення даних, що Git і робить, і це, на додачу до контролю відсутності змін файлів (тобто файли, в яких немає змін, не дублюються), робить його простим та надійним інструментом контролю версій, хоча через велику кількість функціоналу та консольне управління, поріг входження до нього є достатньо високим.

Переваги Git:

- висока продуктивність;
- підтримка гілкування та злиття;

- розподілена архітектура.

Недоліки:

- складність внутрішньої реалізації;
- високий поріг входження користувачів;
- падіння продуктивності за наявності великих обсягів даних.

1.2.2 Система Apache Subversion

Apache Subversion (SVN) – централізована система контролю версій, де всі дані зберігаються на сервері, до якого кожен користувач має отримувати доступ. На відміну від Git, SVN зберігає не повні змінені файли, а лише зміни, що заощаджує місце, але може ускладнити відновлення версії файлу [2].

Особливістю даної системи є необхідність підключення до інтернету, оскільки система централізована, звичайно можна використовувати її локально, але тоді втрачається можливість командної роботи, бо все одно для збереження змін необхідний зв'язок з сервером.

Основні переваги SVN:

- простіша модель роботи;
- централізоване управління.

Недоліки:

- залежність від сервера;
- обмежені можливості офлайн роботи;
- не дуже ефективна робота з гілками.

1.2.3 Система Perforce

Perforce – високопродуктивна централізована система контролю версій, яка використовується в великих комерційних проєктах, зазвичай в ігровій індустрії [3]. Головною її особливістю є оптимізація для роботи з великими обсягами даних та робота з бінарними файлами (що є дуже

актуальним при роботі з текстурами та іншими типами файлів притаманними для ігрової індустрії). На жаль, вона має всі недоліки, притаманні централізованій системі.

Основні переваги:

- висока продуктивність при роботі з великими репозиторіями;
- ефективна робота з бінарними файлами;
- централізоване управління.

Недоліки:

- залежність від сервера;
- складність налаштування та адміністрування;
- корпоративна ліцензія.
-

1.2.4 Сервіси на основі систем контролю версій

Для спрощення спільної роботи над проектами існують декілька сервісів таких як GitHub, GitLab та Bitbucket [4]. Вони не єдині, але найбільш популярні. Вони не реалізують ніякої власної логіки, а лише використовують готові рішення контролю версій (всі використовують Git, хоча Bitbucket до 2020 року також підтримував систему Mercurial).

Основні можливості таких сервісів:

- віддалене зберігання репозиторіїв;
- інтерфейс для перегляду змін;
- інструменти для командної роботи;
- інтеграція з іншими сервісами.

1.2.5 Порівняльна таблиця існуючих програмних продуктів

Оскільки головна мета даної роботи – це дослідження, тобто ціль зрозуміти алгоритми роботи просунутих рішень щодо контролю версій, порівняння з розроблюваним застосунком недоречно, бо його суть в тому, щоб

показати приклад реалізації, а не привнести щось нове. Таким чином логічно зіставити відмінності функціоналу та підходів у вже існуючих системах.

В таблиці 1.1 приведено порівняння вище описаних систем з розроблюваним програмним продуктом.

Таблиця 1.1 — Порівняння існуючих систем контролю версій та розроблюваного програмного продукту

Критерій	Git	SVN	Perforce	Kaі
Тип системи	Розподілена	Централізована	Централізована	Локальна
Спосіб збереження	Знімки	Зміни	Зміни	Знімки
Робота офлайн	Повна	Обмежена	Обмежена	Повна
Робота з бінарними файлами	Обмежена	Середня	Висока	Обмежена
Основна сфера використання	Розробка ПЗ	Корпоративні системи	Великі проєкти, ігрова індустрія	Навчання
Зручність модифікації	Низька	Низька	Низька	Висока
Прозорість структури репозиторію	Низька	Низька	Низька	Висока
Освітня придатність	Середня	Середня	Низька	Висока

1.3 Опис основних вимог до програми

1.3.1 Визначення функціональних вимог

Функціональні вимоги визначають, що саме повинна робити система. Для моєї системи контролю версій визначено такі основні функціональні вимоги:

- створення комітів – система повинна забезпечувати можливість зберігання стану файлу, як окремої версії з унікальним ідентифікатором;

- збереження історії версій – усі коміти повинні зберігатися у вигляді впорядкованої історії, дозволяючи чітко відстежувати етапи «розвитку» документа;
- обчислення різниці між версіями – система повинна мати алгоритм визначення відмінностей між двома версіями документу;
- перегляд історії змін – користувач повинен мати можливість продивитись всю історію комітів;
- відновлення стану – повинна бути можливість відновлення стану проєкту/документа до будь-якого збереженого «знімку»;
- збереження даних у файловій системі – всі версії та дані які відносяться до них та до проєкту повинні локально зберігатись без використання серверу;
- підтримка командного інтерфейсу – взаємодія з системою повинна відбуватись за допомогою командного рядку.

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги визначають характеристики та обмеження системи:

а) вимоги до безпеки:

- 1) забезпечення цілісності даних шляхом використання хешування версій за алгоритмом SHA-256 [5];
- 2) перевірка цілісності історії змін при зчитуванні;

б) вимоги до продуктивності:

- 1) час створення коміту - не більше 1 секунди для текстових файлів середнього обсягу;
- 2) час обчислення різниці між версіями - не більше 2 секунд;
- 3) швидке завантаження історії змін (до 100 комітів без затримок);

в) вимоги до користувацького інтерфейсу:

- 1) мінімальна кількість команд для виконання основних операцій (commit, log, diff, checkout);
 - 2) зрозуміле відображення змін між версіями;
 - 3) інформативні повідомлення про помилки;
- г) вимоги до надійності:
- 1) можливість відновлення системи після збою;
 - 2) відсутність пошкодження історії версій при некоректному завершенні роботи;
- д) вимоги до масштабованості:
- 1) підтримка збереження не менше 1000 версій одного документа;
 - 2) можливість роботи з файлами розміром до декількох мегабайт;
 - 3) збереження продуктивності при зростанні кількості комітів;
- е) вимоги до сумісності:
- 1) підтримка середовища виконання Node.js;
 - 2) коректна робота на основних операційних системах (Windows, Linux, macOS);
 - 3) незалежність від сторонніх сервісів та мережевого підключення.

1.4 Постановка завдання роботи

Дослідження та програмна реалізація алгоритмів контролю версій документів спрямовані на вивчення, ефективне використання та розуміння роботи збереження змін, обробки історії версій та визначення різниці між файлами.

Основні завдання роботи:

- аналіз предметної області. Дослідити принципи роботи існуючих комерційних систем контролю версій, визначити їхні переваги, недоліки, підходи до вирішення їхніх проблем та особливості реалізації алгоритмів;
- дослідження алгоритмів. Вивчити основні алгоритми контролю версій, зокрема алгоритми відстеження змін, їх обробку та доступ до історії;
- розробка архітектури системи. Спроекувати спрощену, демонстраційну структуру системи контролю версій на основі середовища Node.js;
- реалізація функціоналу. Програмно реалізувати базові можливості системи;
- тестування та оптимізація. Провести тестування функціональності, безпеки, швидкодії та зручності, оцінити зміну цих факторів в залежності від обсягу даних.

Цілі розробки:

- дослідити та проаналізувати алгоритми, що лежать в основі контролю версій;
- реалізувати спрощену систему контролю версій документів як експериментальну модель;
- продемонструвати принципи роботи алгоритмів збереження змін і визначення різниці між версіями;
- оцінити ефективність обраних підходів у контексті обробки текстових даних.

1.5 Висновки за розділом 1

У даному розділі було проведено аналіз предметної області та розглянуто основні підходи до контролю версій документів. Визначено, що дана технологія є дуже важливим інструментом у сучасних інформаційних системах, яка забезпечує зручне і надійне зберігання даних з доступом до всієї історії змін.

Було проаналізовано існуючі системи контролю версій, визначені їх переваги, недоліки, особливості роботи та обмеження. У результаті встановлено, що вони мають досить різні підходи до вирішення однієї проблеми та реалізують різні алгоритми відстеження змін та зберігання даних.

Сформульовано функціональні та нефункціональні вимоги до системи, що буде розроблена в ході цієї роботи, яка має дослідницький характер і спрямована на демонстрацію базових механізмів контролю версій. Особливу увагу приділено алгоритмам визначення змін між версіями та ефективному збереженню даних.

На основі проведеного аналізу визначено основні завдання роботи, що включають дослідження алгоритмів контролю версій та їх програмну реалізацію у вигляді спрощеної системи. Отримані результати створюють базу для подальшого розглядання алгоритмічних підходів та їх практичного застосування.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Структура системи

2.1.1 Основні компоненти системи

Архітектура розроблюваної системи контролю версій є спрощеною та орієнтованою на дослідження алгоритмів збереження змін і обробки історії версій. Система має декілька окремих частин, які мають бути відокремленими один від одного задля досягнення масштабованості продукту, тому побудована за модульним принципом [6]. Структура програми складається з таких основних компонентів:

- ядро системи – забезпечує створення комітів, управління історією та виклик алгоритмів обробки змін;
- модуль роботи з файловою системою – здійснює читання та запис файлів, контролює цілісність даних та обробку помилок при роботі зі сховищем;
- модуль збереження версій – реалізує механізми хешування та ідентифікацію версій, контролює ланцюг комітів, забезпечуючи правильне та послідовне сприйняття історії змін;
- модуль визначення змін – реалізує алгоритми визначення різниці між версіями, формує структуроване відображення змін;
- модуль взаємодії – забезпечує взаємодію користувача з системою через командний рядок.

2.1.2 Взаємодія компонентів системи

Для забезпечення чіткого розділення відповідальностей, взаємодія між компонентами системи відбувається за такою схемою:

Користувач – Модуль взаємодії:

- користувач вводить необхідні команди;
- отримані команди перевіряються на коректність та передаються в ядро системи для подальшого опрацювання.

Схема взаємодії модулів системи зображена на рисунку 2.1.

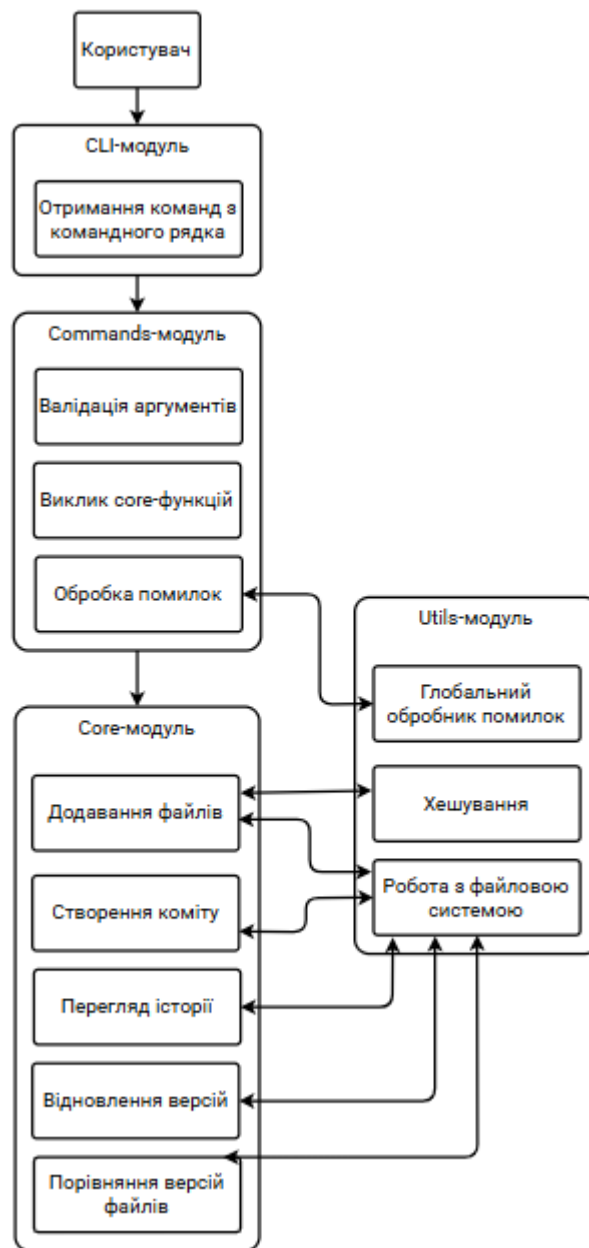


Рисунок 2.1 – Схема взаємодії модулів системи

Модуль взаємодії – Ядро системи:

- отримавши команди, ядро визначає необхідні операції;
- координує роботу інших модулів в залежності від команди.

Ядро – Модуль збереження:

- при створенні коміту необхідні дані передаються в модуль збереження;

- модуль формує необхідну структуру даних .

Ядро – Модуль визначення змін:

- при необхідності порівняння версій ядро звертається до модулю визначення змін, передаючи йому версії, які необхідно порівняти;
- модуль повертає результат до ядра.

Ядро – Модуль роботи з файловою системою:

- при будь-якій операції ядро отримує необхідні дані за допомогою модуля роботи з файловою системою;
- модуль повертає результат до ядра.

Модуль збереження – Модуль роботи з файловою системою:

- забезпечується доступ до історії версій та їх відновлення;
- усі необхідні файли та метадані зберігаються у файловій системі.

2.2 Функціонування системи

2.2.1 Основні процеси системи

Функціонування системи можна описати через основні процеси, які відбуваються під час використання програми.

Процес створення репозиторію:

- користувач ініціює створення нового репозиторію;
- система створює службову директорію;
- формується початкова структура каталогів для збереження версій та метаданих.

Процес створення коміту:

- користувач ініціює збереження поточного стану документа;
- обчислюється хеш для ідентифікації версії кожного зі змінених документів;
- формується структура коміту, що містить дані, метадані та посилання на попередню версію;
- дані зберігаються у файловій системі.

Процес перегляду історії репозиторію:

- користувач запитує список комітів;
- система зчитує збережені метадані;
- формується послідовний список версій;
- результат відображається користувачу.

Процес визначення різниці:

- користувач обирає з якою версією порівняти поточний стан документу;
- викликається модуль визначення змін;
- формується відповідь, яка відображає додані, змінені та видалені рядки.

Процес відновлення версії:

- користувач обирає необхідну версію;
- вміст файлу відновлюється на збережений стан.

2.2.2 Логіка роботи системи

Логіка роботи системи базується на таких принципах.

Принцип незмінності версій:

- будь який коміт незмінний після створення;
- зміни не перезаписують попередні дані, а створюють нові версії;

Принцип ідентифікації через хешування:

- кожна версія має унікальний ідентифікатор (хеш);
- хеш формується на основі вмісту документу;
- це дозволяє перевірити незмінність та цілісність даних.

Принцип лінійної історії:

- коміти організовані у вигляді послідовного ланцюга;
- кожен коміт містить посилання на попередній.

Принцип роботи з файловою системою:

- всі дані зберігаються локально у вигляді файлів і каталогів;
- не використовується окрема база даних;
- структура сховища забезпечує швидкий доступ до необхідних даних.

2.3 Вибір мови програмування та інструментарію для розробки програми

Для розробки системи контролю версій обрано мову TypeScript [7] (надбудова над JavaScript, що привносить до неї статичну типізацію) із використанням середовища виконання Node.js [8], яке дозволяє мові JavaScript існувати та працювати за межами браузеру при цьому не втрачаючи своїх переваг. Вибір такого стеку обумовлений такими факторами:

Статична типізація: TypeScript на відміну від свого «старшого брата» JavaScript має статичну типізацію, що дозволяє писати більш передбачуваний код, який легко читати та розуміти та мати можливість знайти помилку ще до тестування, що значно покращує продуктивність роботи. І хоча опис типів теж займає певний час, все ж відловлювати фантомні помилки набагато витратніше за часом та силами.

Наявність великої кількості бібліотек: TypeScript має велику підтримку розробників, які постійно завантажують нові або оновлюють вже існуючі бібліотеки для того, щоб спростити розробку своїм колегам. Це дозволяє не витрачати час на те, щоб зробити щось, що вже давно зроблене набагато якісніше і неодноразово протестовано.

Зрозумілий синтаксис: TypeScript має простий і чистий C-подібний синтаксис, що полегшує розробку, розуміння коду та його підтримку командою розробників. Це сприяє швидкому написанню коду та зменшенню кількості помилок.

Активна підтримка та екосистема: TypeScript (JavaScript) активно підтримується, а Node.js часто отримує нові оновлення, які дозволяють працювати простіше і ефективніше.

2.3.1 Опис переваг та обмежень вибраної мови

Переваги.

Завдяки простому синтаксису та відсутності необхідності в особливих інструментах для роботи, досягається дуже потужна продуктивність та швидкість роботи, зробити простий прототип є дуже простою задачею.

На будь якій платформі з необхідною версією Node.js код буде виконуватись абсолютно однаково, що гарантує кросплатформність системи.

Функція «Hot Reload» прискорює процес тестування та налагодження коду.

Розвинена екосистема включає різноманітні бібліотеки для будь яких задач, таких як: шифрування, робота з файлами, стиснення даних тощо.

Обмеження.

На відміну від компільованих мов TypeScript (JavaScript) має обмежену швидкодію, ця проблема може показати себе при роботі з дуже великими файлами, але для середньостатистичних обсягів даних (тим більше для експериментальної моделі) її швидкодії цілком достатньо. TypeScript має однопотокową модель, тому виконувати паралельні операції неможливо.

Порівняння мов програмування наведено в таблиці 2.1.

Таблиця 2.1 – Порівняння мов програмування

Критерій	TypeScript	JavaScript	C++
Типізація	Статична	Динамічна	Статична
Простота розробки	Висока	Висока	Низька
Швидкодія	Середня	Середня	Висока
Робота з файловою системою	Проста	Проста	Ускладнена
Екосистема	Розвинена	Розвинена	Розвинена
Поріг входження	Середній	Низький	Високий

TypeScript обрано через його статичну типізацію, кросплатформність і зручність розробки, що ідеально відповідає потребам даного проєкту.

2.3.2 Середовище розробки

Для розробки застосунку обрано WebStorm – інтегроване середовище розробки (IDE), яке є стандартом для роботи з TypeScript та JavaScript. Це рішення обґрунтоване наступними причинами.

Глибока підтримка TypeScript.

Середовище забезпечує повноцінну роботу з типами, автодоповненням та аналізом коду.

Потужні інструменти.

Середовище надає широкий набір функцій: редактор коду з інтелектуальним автодоповненням, підтримка Git, профілювання продуктивності та пам'яті.

Підтримка плагінів.

WebStorm підтримує плагіни для спрощення розробки та більшим контролем над проєктом.

Інтеграція з Git.

Вбудована підтримка систем контролю версій полегшує командну роботу та відстеження змін у коді.

2.3.2.1 Обґрунтування вибору та опис його можливостей

WebStorm обрано за його надійність, глибоку підтримку необхідних мов програмування та широкий набір інструментів, що значно спрощують розробку [9].

Редактор коду WebStorm має інтелектуальне автодоповнення, авторефакторинг, можливість створювати свої сніпети (декілька букв або слово яке трансформується в великий шматок універсального коду для певних

задач), а з встановленням різних плагінів може отримати ще більше функцій таких як демонстрація розмірів імпортованих бібліотек, або інформація щодо дати та автору редагування обраного рядка.

Крім того дане середовище підтримує просте налагодження: точки зупину, перегляд змінних, стек викликів.

Порівняння середовищ розробки наведено в таблиці 2.2.

Таблиця 2.2 – Порівняння середовищ розробки

Критерій	WebStorm	Visual Studio Code
Підтримка TypeScript	Повна	Часткова
Інструменти	Потужні	Середні
Інтеграція з Git	Вбудована	Вбудована
Плагіни	Багато	Багато
Налаштування	Мінімальне	Потребує обов'язкового налаштування
Розмір	~1.5 Гб	~200 Мб
Вартість	Платне	Безкоштовне

WebStorm обрано через його спеціалізацію на TypeScript і наявність усього необхідного для розробки, тестування та оптимізації, без особливих налаштувань.

2.4 Вимоги до технічних і програмних засобів

2.4.1 Необхідні технічні засобів для розробки та експлуатації

Для розробки та експлуатації системи контролю версій необхідні такі технічні засоби:

- комп'ютер для розробки: для написання коду, компіляції та тестування;

- інтернет-з'єднання на етапі розробки: для встановлення необхідних бібліотек та доступу до документації.

2.4.2 Вимоги до апаратного та програмного забезпечення

Апаратне забезпечення:

- процесор: Intel Core i3 або еквівалент (достатній для обробки файлів та виконання алгоритмів);
- оперативна пам'ять: 4 Гб (рекомендується 8 Гб для комфортної роботи з середовищем розробки);
- вільне місце на диску: 8 Гб (включаючи WebStorm, репозиторій і залежності).

Програмне забезпечення:

- операційна система: Windows 10/11, macOS 10.14+, або Linux (Ubuntu 20.04+);
- Node.js: версія 18+ для виконання серверної частини програми;
- TypeScript: версія 5.0+ для розробки з використанням статичної типізації;
- npm (Node Package Manager): для керування залежностями проєкту;
- WebStorm: середовище розробки для написання та відлагоджування коду.

2.5 Висновки за розділом 2

У другому розділі було описано архітектуру розроблюваної системи контролю версій, визначено основні компоненти системи та їхню взаємодію. Вибір мови програмування TypeScript та середовища виконання Node.js обумовлений їх простотою, ефективністю роботи, а також великою екосистемою. WebStorm обрано як середовище розробки завдяки його

інтеграції з TypeScript і потужним інструментам для написання, тестування та налагодження коду.

3 РОЗРОБКА ПРОГРАМИ

3.1 Архітектура системи

Архітектура розроблюваної системи контролю версій (далі «Kai») побудована за модульним принципом, метою якого є розділення відповідальності між компонентами системи, щоб покращити сприймання та розуміння коду, забезпечити окремий розвиток різного функціоналу (без ризику зламати старий модуль при розробці нового) та гарантувати безпеку роботи – кожен модуль має свою зону відповідальності, яка не може зачепити інший функціонал.

Система має консольний інтерфейс (CLI), за аналогом Git та має наступні модулі першого рівня:

- модуль обробки команд (cli) – приймання команд користувача, їх обробка, виявлення помилок в написанні;
- модуль команд (commands) – обробка вхідних даних, їх передача в ядро системи та відображення статусу виконання користувачеві;
- модуль ядра системи (core) – виконання логіки відповідних команд;
- модуль допоміжного функціоналу (utils) – реалізація окремих допоміжних дій, таких як: робота з файлами, хешування, загальна обробка помилок.

Схема взаємодії модулів зображена на рисунку 3.1.

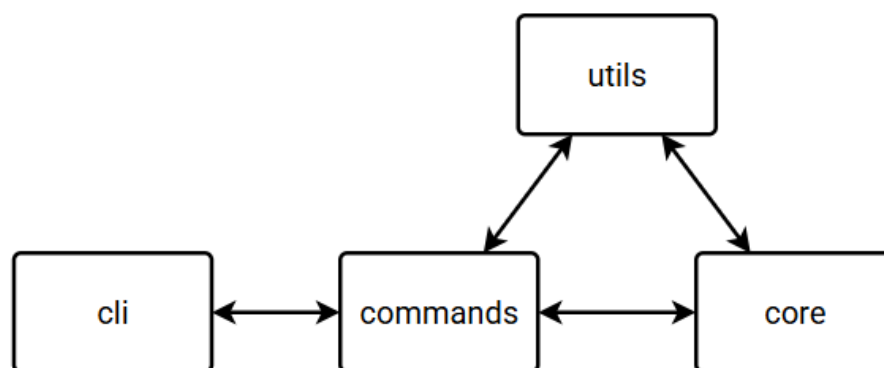


Рисунок 3.1 – Схема взаємодії модулів системи

3.1.1 Структурна схема системи

Структурно система складається з декількох рівнів, кожен з яких виконує свою окрему функцію, а модулі першого (або найвищого) рівня ще можуть ділитися на підмодулі.

Структурну схему системи зображено на рисунку 3.2.

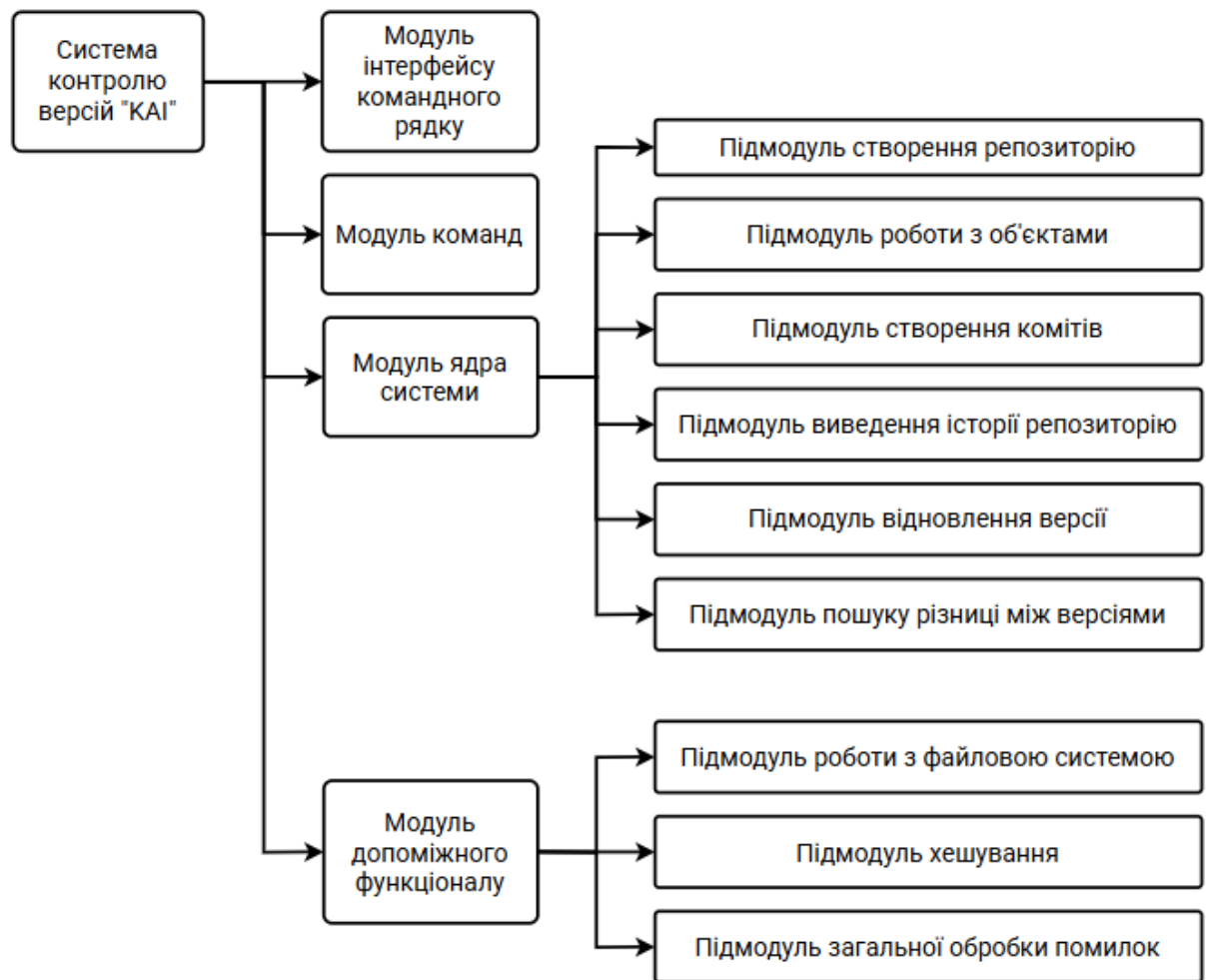


Рисунок 3.2 – Структурна схема системи

3.2 Реалізація алгоритмів системи

Оскільки основною ціллю розроблення системи «Kai» є дослідження та практична реалізація основних алгоритмів контролю версій, окрім опису роботи програми буде актуально більш детально розібрати їх роботу в

«ідеальному» випадку та дослідити їх експериментальну реалізацію (експериментальна реалізація передбачає спрощення тільки в окремих випадках, не перенавантажуючи проєкт дуже складними методами оптимізації, основний принцип повністю збережений).

3.2.1 Алгоритм створення репозиторію

Створення репозиторію є першим і фундаментальним етапом роботи будь-якої системи контролю версій. Під час ініціалізації репозиторію (виконання команди «kai init») створюється службова директорія «.kai» з необхідною структурою файлів, яка наведена на рисунку 3.3.

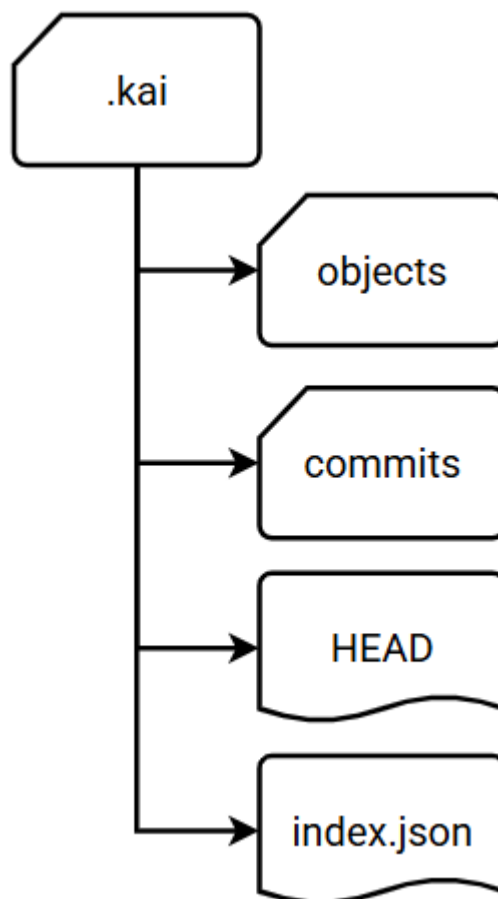


Рисунок 3.3 – Базова структура файлів службового репозиторію

На відміну від Kai, Git має трохи складнішу структуру. Це зумовлено по перше тим, що Git також має можливість зберігати дані віддалено і для цього необхідні додаткові метадані. Крім того Git вважає коміти та «знімки» файлів об'єктами та не розділяє їх окремо, маючи більш складний спосіб визначення типу об'єкту, де в кожному об'єкті окрім корисної інформації зберігаються також метадані про тип. Але це ускладнення базується на додатковій оптимізації, враховуючі, що обсяг репозиторію Git, як комерційної системи контролю версій, може бути достатньо великим; в даному ж випадку це є неактуально, а для наочності та розуміння алгоритму дій і зовсім непотрібно.

3.2.2 Алгоритм додавання файлів

Робота алгоритму продемонстрована на рисунку 3.4.



Рисунок 3.4 – Алгоритм додавання файлу

Щоб дати користувачу змогу контролювати, які файли потрібно зберігти, а не слідкувати за всіма, існує алгоритм додавання файлів. Його суть полягає в тому, що перед тим як зробити коміт, користувач повинен обрати файли, які необхідно зберігти. Тут по суті відбувається найголовніша частина створення коміту. При виконанні команди «add» система на основі вмісту файлу вираховує хеш (у випадку Kai це алгоритм SHA-256) і зберігає ці дані в

файлі index в форматі: відносний шлях до файлу – хеш. Також на даному етапі важливо зберігти знімок файлу в об'єкти, де назвою файлу буде хеш, а вмістом стиснений, за допомогою бібліотеки zlib та алгоритму «DEFLATE» (в системі Kai використовується точно такий алгоритм), оригінальний вміст файлу.

Щодо алгоритму хешування слід зазначити, що Git донедавна використовував алгоритм SHA-1, що генерує 40-символьний код, але через побоювання колізій починаючи з версії 2.29 ввів підтримку SHA-256 (64-символьний код), який і використовує система Kai.

3.2.3 Алгоритм створення коміту

По суті створення коміту не передбачає нічого окрім окремої фіксації змін, всі підготовчі етапи вже виконані під час додавання файлів. Але існує неочевидний момент, не можна просто взяти файли з index та зберегти їх окремо, оскільки коміт повинен не тільки зберігати змінені файли, а й загальний стан репозиторію, інакше буде не можливо правильно відновити версію. Є два підходи:

- рекурсивно пройти по всім директоріям і файлам репозиторію та зберігти всі файли (але це дуже довго, непродуктивно та може призвести до помилок);
- отримати файли з попереднього коміту, ті що змінились перезаписати, ті що більше не існують видалити та додати нові.

Звісно, що будь-яка нормальна система буде застосовувати саме другий варіант. Для роботи такої системи потрібен ланцюг комітів, функціонування якого досягається за допомогою правильної структури кожного коміту, а саме дані коміту повинні містити наступні поля:

- повідомлення – обов'язкове для нормального сприйняття людиною;
- дата та час створення коміту – для додаткової зручності;
- список файлів, за структурою index файлу;

– хеш попереднього коміту.

Саме хеш попереднього коміту дозволяє мати структуру ланцюга історії репозиторію, що дозволяє точно відслідковувати хронологічний порядок змін та отримати дані попереднього стану репозиторію. До речі вже ця система по суті дозволяє мати функціонал гілкування, але для його зручного використання необхідно мати назви гілок та їх представлення, зрозуміле користувачу (наразі ця система не імплементована в Kai).

В кінці цього алгоритму необхідно змінити HEAD на хеш тільки що створеного коміту та очистити index файл.

Повний принцип роботи алгоритму створення коміту зображений на рисунку 3.5.

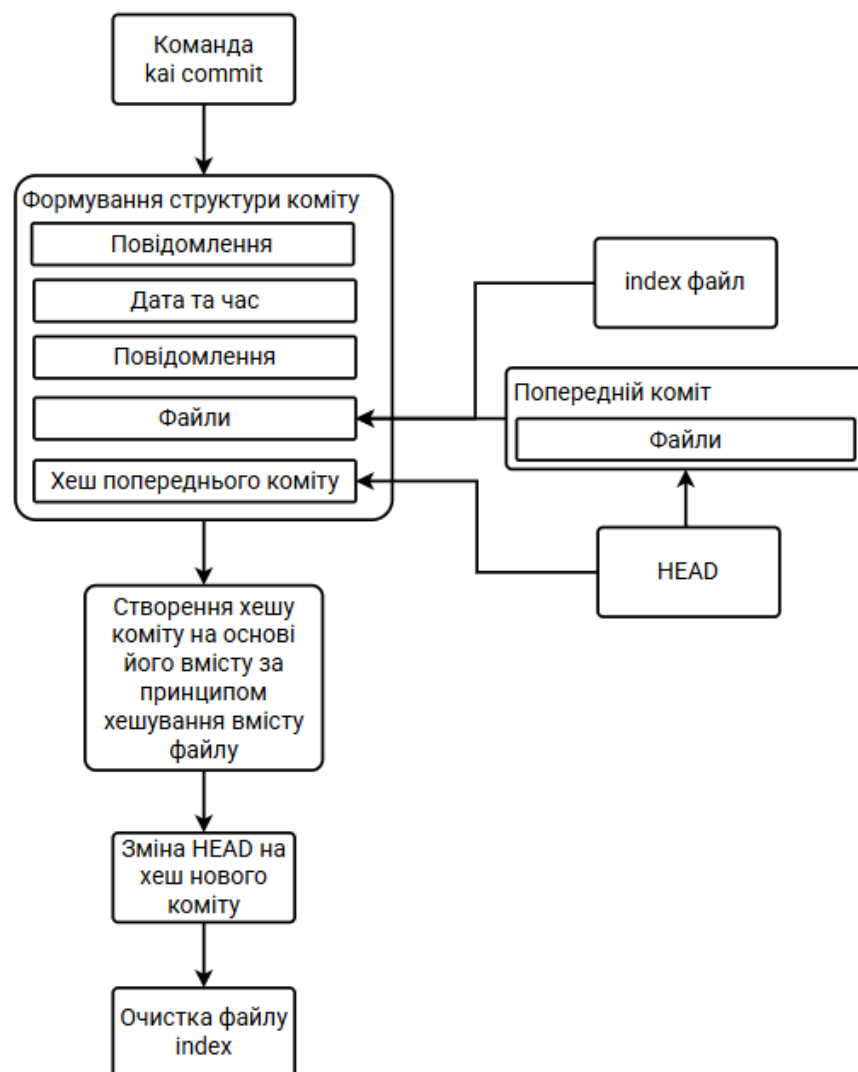


Рисунок 3.5 – Принцип роботи алгоритму створення коміту

3.2.4 Алгоритм відновлення версій

Відновлення версій є однією з найпростіших функцій в системі контролю версій, оскільки ніяка обробка даних не потрібна, все що для цього потрібно вже підготовлено, потрібно тільки перезаписати файли. Єдине, що необхідно зробити задля правильного відновлення версії – це видалити файли яких немає в цільовому коміті і які збережені в поточному, але для більш простого розуміння будемо вважати, що ми видаляємо всі файли за якими слідкуємо. В результаті маємо таку схему алгоритму (рисунок 3.6).

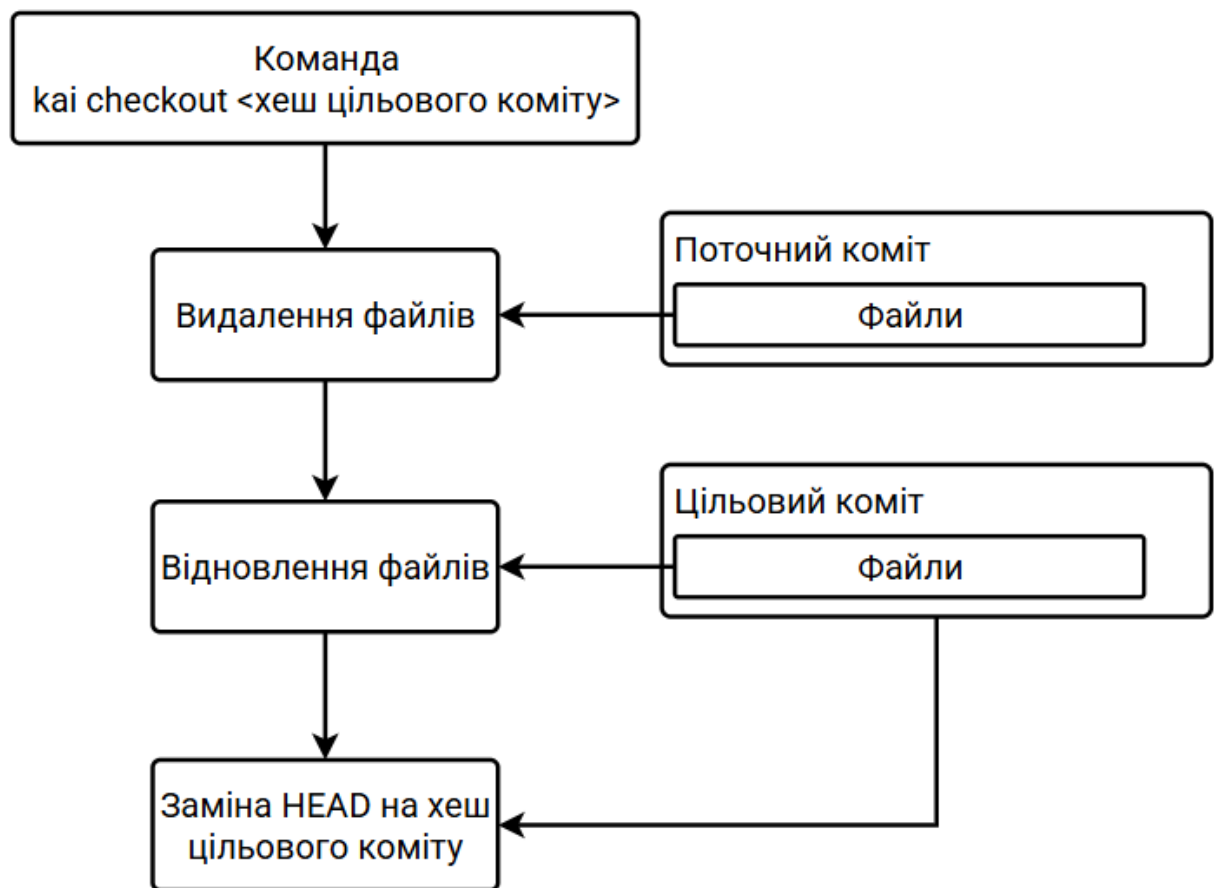


Рисунок 3.6 – Принцип роботи алгоритму відновлення версії

3.2.5 Алгоритм визначення змін

Визначення змін є, мабуть, найцікавішим та найскладнішим з алгоритмів, які були розглянуті раніше (якщо говорити про алгоритм Майерса, який використовує Git). Його робота буде розглянута окремо та буде представлена одним блоком в схемі алгоритму роботи порівняння саме в Kai.

Алгоритм Майерса є найпоширенішим та найточнішим алгоритмом побудови різниці документів. Його суть полягає в побудові графу змін та пошуку найкоротшого шляху «жадібним пошуком».

Для прикладу візьмемо два рядки, де будемо шукати зміни: перший рядок – «СВАВА», другий – «АСАВВ». Побудуємо таблицю, де вісь X – перший рядок, вісь Y – відповідно другий (рисунок 3.7).

	С	В	А	В	А
А					
С					
А					
В					
В					

Рисунок 3.7 – Початкова таблиця символів

Далі позначимо «спеціальні» квадрати – квадрати в яких по двом осям символи перетинаються (рисунок 3.8).

	С	В	А	В	А
А					
С					
А					
В					
В					

Рисунок 3.8 – Таблиця зі «спеціальними» квадратами

Далі суть алгоритму просто в тому, щоб знайти найкоротший шлях від лівого верхнього кута до правого нижнього за допомогою «жадібного пошуку», де «спеціальні» квадрати мають нульову вагу. В результаті отримуємо результат зображений на рисунку 3.9.

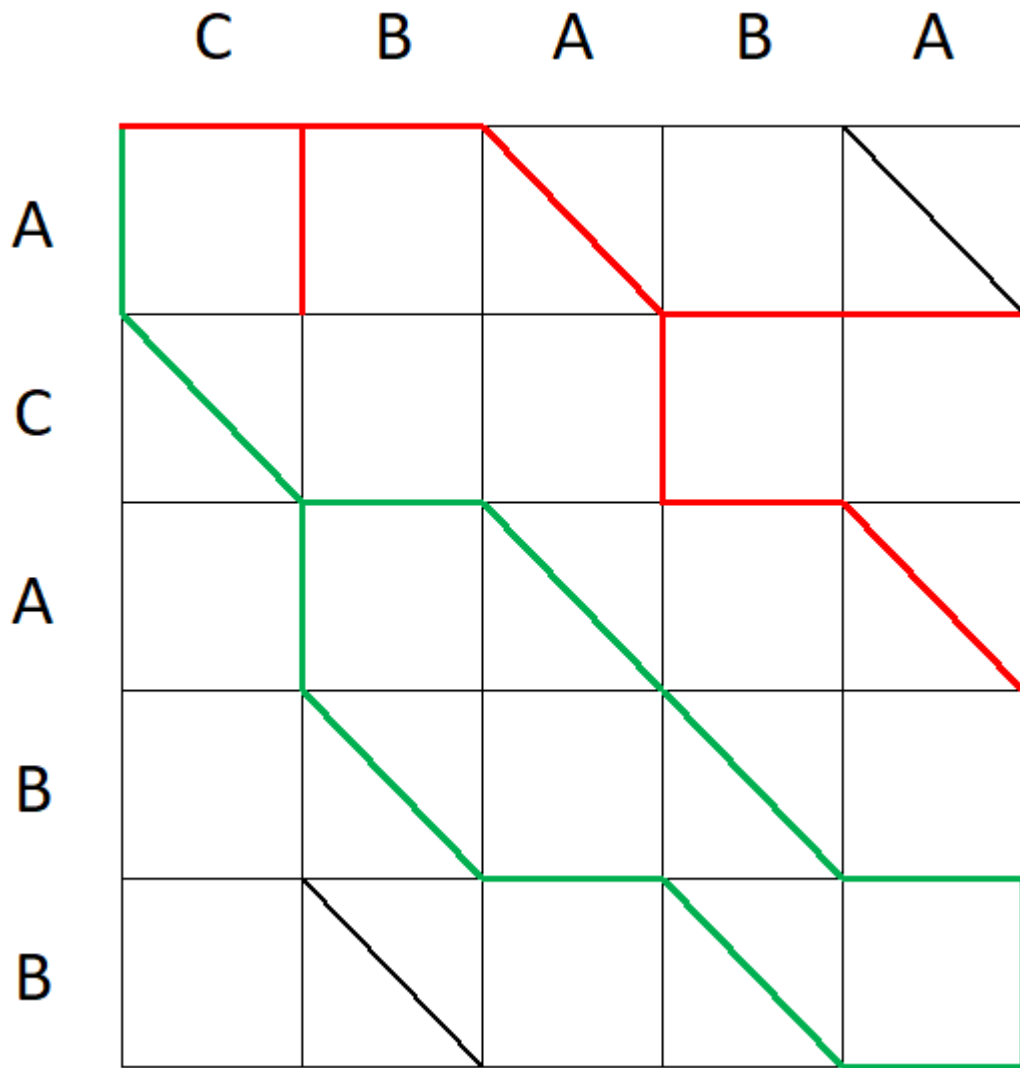


Рисунок 3.9 – Результат пошуку найкоротшого шляху

В даному випадку маємо два найкоротші шляхи, що в принципі неважливо, алгоритм обере перший до якого дійшов. Суть в тому, що рух по горизонталі означає видалення відповідного символу з осі X, а рух по вертикалі – додавання символу з осі Y, діагоналі означають відсутність змін. Далі в коді це можна представити як масив структур:

- тип: додавання, видалення, відсутність змін;
- контент: символ або строка про яку йде мова.

При цьому послідовність типу видалення-додавання, можна представити як «зміна». У випадку з порівнянням файлів на першому етапі варто шукати зміни по рядках, далі шукати послідовність видалення-

додавання та використати алгоритм локально на рядку за принципом наведеному у прикладі, щоб представити зміни більш точно.

В системі Kai це реалізовано за допомогою бібліотеки diff, яка вже надає простий інтерфейс для порівняння файлів за допомогою алгоритму Майерса. Візуалізація ж імплементується через побудову HTML файлу та графічне відображення змін, яке є дуже схожим на те, як це виглядає на GitHub та подібних сервісах (рисунок 3.10).

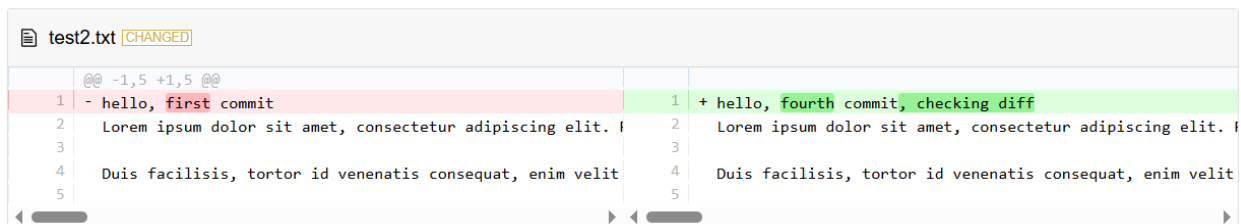


Рисунок 3.10 – Відображення змін в системі Kai

Схема виконання порівняння в системі представлена на рисунку 3.11.

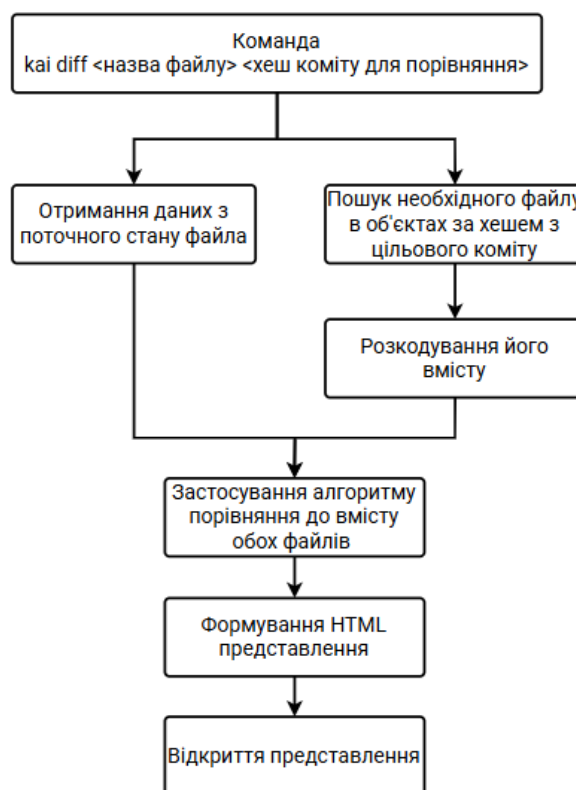


Рисунок 3.11 – Схема виконання порівняння в системі Kai

3.3 Структура зберігання даних

У системі контролю версій Kai зберігання даних організовано в локальній файловій системі та відсутня інтеграція з окремою базою даних. Це позбавляє потреби в додаткових залежностях, робить систему простішою та кросплатформенною (Git працює так само, але має підтримку віддалених репозиторіїв).

Всі службові дані та метадані системи зберігаються в директорії «.kai», яка створюється під час ініціалізації репозиторію.

3.3.1 Структура службової директорії

В розділі 3.2.1 вже було частково описано структуру службової директорії, але лише базовий вигляд при створенні репозиторію, повний перелік файлів та каталогів під час роботи такий:

- objects – містить стиснені копії файлів, збережені під іменами їхніх хешів, на основі вмісту, що забезпечує відсутність дублювань при відсутності змін;
- commits – містить інформацію про створені коміти;
- HEAD – файл, що містить хеш поточного коміту;
- index.json – файл, що містить список файлів, доданих до відстеження;
- diff.html – файл, що містить останнє HTML представлення різниці версій.

Дана структура є трохи спрощеною в порівнянні з Git, але зберігає всі його принципи роботи та дозволяє більш наочно бачити кроки роботи системи контролю версій.

3.3.2 Формат збереження об'єктів

Усі версії файлів в системі зберігаються в директорії «objects». Кожен об'єкт – окремий файл, ім'я якого – хеш сгенерований на основі вмісту оригінального файлу. Вміст об'єкта – стиснений бібліотекою zlib вміст оригінального файлу.

Така структура вирішує дві потенційні проблеми:

- можливість дублювання та збереження двох або більше однакових за вмістом файлів;
- нераціональне використання пам'яті пристрою – окрім того, що дані не дублюються, вони ще й стискаються. Але слід зазначити, що бібліотека zlib не дуже ефективно працює з невеликими об'ємами даних та для маленьких файлів, розмір об'єкту може бути навіть вищим, але різниця в декілька десятків байт не є суттєвою.

Git також використовує так звані «pack» файли, куди періодично додатково стискає і записує всі об'єкти, задля економії місця. Звісно, процес розшифрування даних звідти є набагато складнішим ніж з окремих файлів, але з метою економії пам'яті це досить логічне рішення. Каі в свою чергу не має такого функціоналу, але це дозволяє значно спростити реалізацію та розуміння системи.

3.3.3 Файл index

Файл index працює як спрощений аналог «staging area» у Git. В нього додаються файли які повинні бути збережені в наступному коміті. Його призначення полягає в:

- збереженні списку файлів, доданих через команду «add»;
- формуванні набору даних для наступного коміту.

Кожен запис має вигляд: ключ – відносний шлях до файлу (від кореню репозиторію), значення – хеш відповідного об'єкту.

3.3.4 Формат збереження комітів

Кожен коміт – це окремий файл в директорії «commits». Ці файли містять службову інформацію для можливості коректного відновлення стану репозиторію.

Назва файлу – хеш, на основі вмісту (за тою ж системою, що і хеш об'єктів). Поля коміту такі:

- повідомлення – текстове представлення вмісту коміту, яке задається користувачем;
- дата та час створення – спрощує розуміння історії репозиторію;
- хеш батьківського коміту – дозволяє будувати ланцюг історії версій;
- список файлів – структура така сама як в index файлі.

Такий формат містить всю базову необхідну інформацію, яка необхідна для керування версіями. У випадку з Git можна додати додаткові дані про автора коміту, посилання на об'єкт дерева (підтримка гілок) тощо.

3.3.5 Файл HEAD

Файл HEAD необхідний для збереження інформації про поточний коміт. Він просто зберігає хеш коміту і є початковою точкою історії репозиторію, оскільки кожен коміт зберігає хеш попереднього.

Вміст даного файлу змінюється при створенні нового коміту або при перемиканні версій. Наявність даної інформації в окремому файлі значно спрощує розуміння поточної «точки відліку» без необхідності аналізу всіх комітів в пошуках останнього або необхідного.

3.3.6 Порівняння структури зберігання Kai та Git

Порівняння структури зберігання даних Git та Kai наведено в таблиці 3.1.

Таблиця 3.1 – Порівняння структури зберігання даних Kai та Git

Критерій	Git	Kai
Службова директорія	.git	.kai
Типи об'єктів	blob, tree, commit	object, commit
HEAD	Так	Так
index	Так (складніша система)	Так
Pack файли	Так	Ні
Стиснення	Zlib	Zlib
Хешування	SHA-1, SHA-256	SHA-256
Формат зберігання	Сильно оптимізований	Оптимізований

Таким чином, структура зберігання Kai є спрощеним аналогом архітектури Git та добре забезпечує реалізацію базових механізмів контролю версій при значно меншій складності внутрішньої організації.

3.4 Обробка та валідація помилок

Під час розробки системи контролю версій Kai особлива увага приділялась забезпеченню стабільності роботи програми та обробці помилок зі зрозумілим фідбеком користувачеві, що пішло не по плану. Під час тестування виявлено, що зазвичай помилки виникають в основному в двох випадках: якщо користувач неправильно вводить команду або її параметри, або якщо файл неможливо прочитати через неправильний формат або інші чисто технічні проблеми. Але тим не менш система обладнана значною кількістю перевірок та централізованою системою обробки помилок.

3.4.1 Централізована обробка помилок

Kaі має окремий допоміжний модуль обробки помилок. Його задачею є виконання переданої функції з автоматичним перехопленням винятків.

Такий підхід значно спрощує код та позбавляє необхідності постійно писати конструкцію обробки помилок; функцію цього модулю можна застосовувати один раз на рівні команд, тим самим не звертаючи уваги на даний аспект всередині логіки програми. На практиці в системі це використовується на рівні команд та роботи з файлами, як на найвразливіших місцях програми.

Крім того такий підхід забезпечує однаковий формат повідомлень про помилки, що додає прозорості в користуванні системою.

3.4.2 Валідація вхідних даних

Щоб гарантувати правильність виконання алгоритмів та не дозволити їм виконати незворотні дії (наприклад видалення файлів), важливо перевірити цілісність системи на етапі обробки вхідних даних.

У системі реалізовані такі перевірки:

- існування репозиторію;
- існування файлів, що додаються;
- наявність комітів або відстежуваних файлів;
- існування об'єктів.

При виявленні помилок або відсутності важливих файлів система перериває виконання алгоритму, виводить інформативне повідомлення та не змінює стан репозиторію.

Як варіант для вдосконалення контролю цілісності даних, можна також реалізувати швидке створення резервної копії, на випадок якщо виконання певним чином перерветься або виникнуть додаткові обставини, що можуть спричинити порушення структури репозиторію або службового каталогу.

3.4.3 Оптимізація роботи з файлами

Оскільки система контролю версій постійно виконує операції читання, запису та порівняння файлів, важливим аспектом реалізації є оптимізація роботи з файловою системою.

В системі Kaі для зменшення обсягу збережених даних використовується стиснення за допомогою бібліотеки `zlib`. Перед записом «знімків» файлів їх вміст стискається, що дозволяє:

- зменшити використання пам'яті;
- прискорити роботу з великими файлами;
- прибрати дубльовані дані.

Кожен файл зберігається як окремий об'єкт з хешем в якості ім'я, що забезпечує швидкий доступ до необхідних файлів (це працює за принципом роботи хеш-таблиці).

У системі Kaі використовуються синхронні операції роботи з файловою системою. Такий підхід було обрано свідомо, оскільки система реалізована у вигляді консольного застосунку, де більшість команд виконуються послідовно та не потребують паралельної обробки даних.

На відміну від серверних або графічних застосунків, у CLI-системі відсутня необхідність забезпечення безперервної взаємодії з користувачем під час виконання операцій. Користувач очікує завершення команди перед переходом до наступної, тому використання асинхронних механізмів не дає суттєвих переваг у продуктивності або зручності роботи.

3.5 Висновки за розділом 3

У третьому розділі було розглянуто програмну реалізацію системи контролю версій Kaі та описано алгоритми, що забезпечують функціонування систем контролю версій та їх реалізація в межах Kaі. Було розроблено та описано модульну архітектуру системи, яка включає компоненти для обробки

команд, реалізації логіки контролю версій, роботи з файловою системою та допоміжних операцій.

У процесі реалізації було досліджено та програмно реалізовано базові механізми систем контролю версій, зокрема:

- створення репозиторію;
- додавання файлів до відстеження;
- збереження об'єктів;
- створення комітів;
- відновлення станів репозиторію;
- визначення різниці між версіями.

Окрему увагу було приділено алгоритму порівняння Майерса, як найскладнішому та найцікавішому з усіх представлених до цього. Це дозволяє мати набагато ширше уявлення про внутрішні, низькорівневі функції систем контролю версій.

Було описано структуру зберігання службових даних у репозиторії Kai та виявлено, що не дивлячись на спрощення, вона абсолютно точно демонструє принципи роботи гігантів на комерційному ринку. До того ж представлена структура ідеальна для дослідження та простого сприйняття.

Крім того, у розділі представлено механізми обробки помилок, які дозволяють з мінімальними зусиллями забезпечувати коректне перехоплення будь-яких помилок на всіх рівнях роботи програми.

Отримані результати підтверджують можливість реалізації повноцінного аналогу комерційної системи та дає розгорнуте розуміння її роботи, що на перший погляд виглядає дуже складно та неочевидно.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

У даному розділі розглянуто практичне використання розробленою системи контролю версій Kai, проведено тестування її функцій та виконано експериментальне дослідження реалізованих алгоритмів. Особливу увагу приділено аналізу алгоритмів збереження об'єктів, створення комітів, визначення змін між версіями файлів та відновлення їхніх станів.

Оскільки система є дослідницькою реалізацією принципів роботи систем контролю версій, основною ціллю дослідження є оцінка коректності роботи алгоритмів, аналіз їх поведінки при різних вхідних даних та порівняння окремих механізмів роботи з їхніми реалізаціями в Git.

У межах розділу розглянуто процес роботи користувача з Kai, виконано функціональне тестування команд, проаналізовано продуктивність та ефективність системи, та визначено напрями її можливого подальшого розвитку.

4.1 Опис застосування програми

Система контролю версій Kai призначена для дослідження та демонстрації базових алгоритмів контролю версій. Програма реалізована у вигляді консольного застосунку та дозволяє виконувати фундаментальні операції сучасних комерційних систем контролю версій.

Основними можливостями розробленої системи є:

- створення репозиторію;
- додавання файлів до відстежування;
- створення комітів;
- перегляд історії змін;
- відновлення версій документів;
- визначення різниці між версіями файлів.

На відміну від комерційних систем, Kai має спрощену архітектуру та не може конкурувати з гігантами, як Git, але, в той же час, така структура ідеальна для дослідження внутрішніх механізмів роботи систем контролю версій. Таким чином програма може використовуватись:

- для навчальних цілей;
- для дослідження алгоритмів (особливо різниці версій та методам зберігання «знімків»);
- для демонстрації основних принципів роботи систем контролю версій;
- як спрощена модель розподілених систем контролю версій.

Оскільки програма реалізована за допомогою CLI, взаємодія користувача відбувається через консоль, шляхом введення чітких команд, які можна побачити класично для таких систем після введення назви програми (в даному випадку «kai»). Приклад виведення наведено на рисунку 4.1.

```
Kai version control system

Options:
  -V, --version      output the version number
  -h, --help         display help for command

Commands:
  init              Initialize repository
  add <filename>    Add file to staging
  commit <message>  Commit staged files
  log               Log repository history
  checkout <targetCommit> Restore needed state
  diff <fileName> <targetCommit> Show difference between file versions
```

Рисунок 4.1 – Перелік команд та їх аргументів системи контролю версій Kai

4.1.1 Огляд планової цільової аудиторії

Оскільки Kai має дослідницький характер, її основною цільовою аудиторією є користувачі, зацікавлені у вивченні внутрішнього принципу роботи систем контролю версій та алгоритмів обробки змін.

Для аналізу потенційного розподілу цільової аудиторії було сформовано прогнозований розподіл користувачів та напрямів використання системи, що базується на аналізі сфер застосування Git-подібних систем, навчальних матеріалів та популярності систем контролю версій серед різних категорій користувачів, основними з яких є:

- студенти спеціальностей, споріднених до інженерії програмного забезпечення;
- розробники-початківці;
- користувачі, що вивчають Git, або які відчувають проблеми з його розумінням;
- викладачі та дослідники.

Прогнозований розподіл цільової аудиторії наведений на рисунку 4.2.

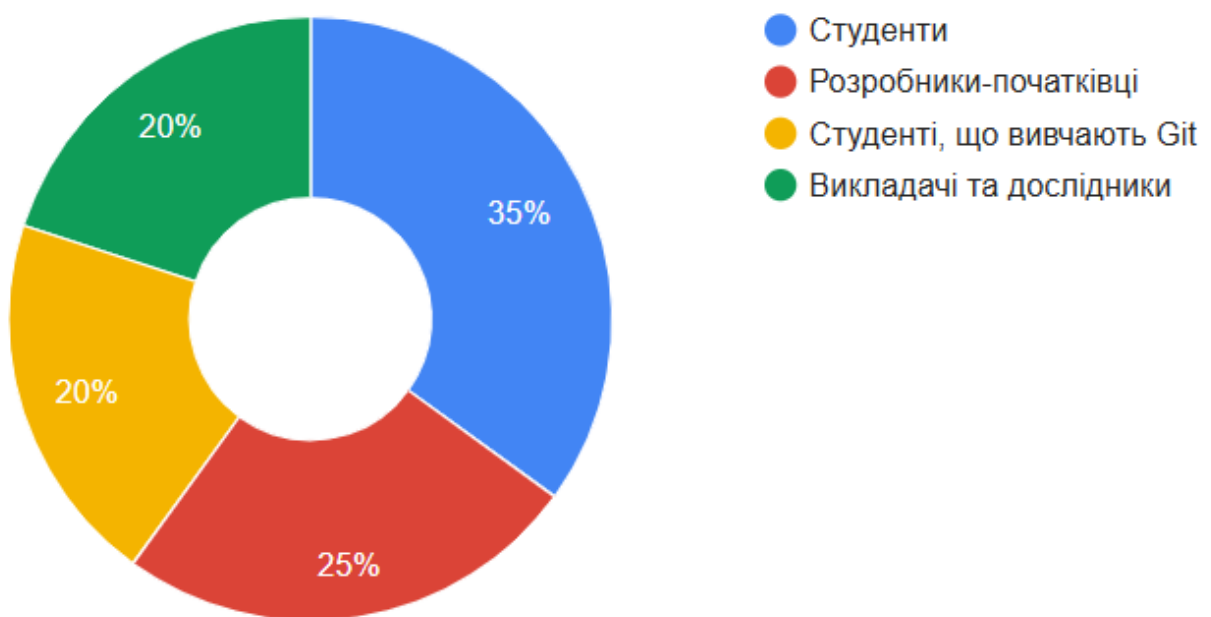


Рисунок 4.2 – Прогнозований розподіл цільової аудиторії системи Kai

4.1.2 Розподіл сценаріїв використання програми

На основі мети програми, сценарії її використання можуть бути наступними:

- навчання принципам роботи Git-подібних систем;

- дослідження алгоритмів різниці версій документів;
- аналіз збереження «знімків» файлів;
- тестування збереження стану репозиторію;
- навчальні демонстрації.

Прогнозований розподіл сценаріїв використання наведений на рисунку 4.3.

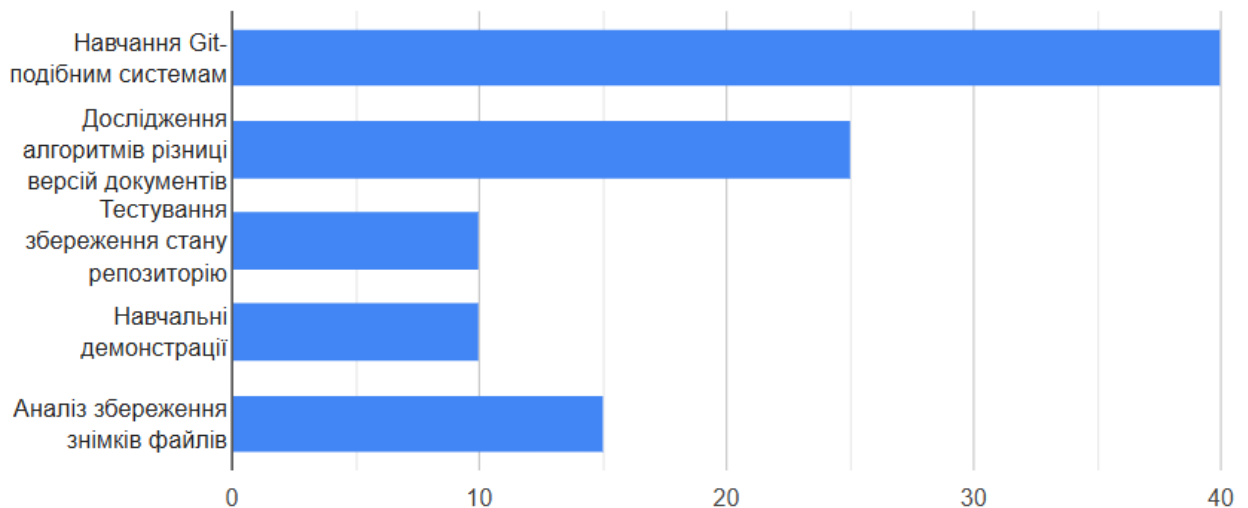


Рисунок 4.3 – Прогнозований розподіл сценаріїв використання системи Kai

4.2 Інструкція по роботі з програмою

Система Kai реалізована в вигляді консольного застосунку та запускається через командний рядок операційної системи. Взаємодія користувача з нею відбувається за допомогою команд, подібних до Git.

В даному розділі описано процес встановлення системи та середовища виконання, створення репозиторію, виконання основних команд та приклади їх роботи.

4.2.1 Встановлення програми

Перш за все, для роботи системи Kai необхідне середовище виконання Node.js, яке доступно на всіх основних операційних системах.

Для коректного встановлення необхідно близько 100 Мб вільного місця та Node.js версії 18+.

4.2.1.1 Встановлення Node.js

Для встановлення Node.js необхідно перейти на офіційний сайт [10], завантажити необхідну версію для своєї операційної системи, запустити програму-інсталятор та виконати стандартні кроки встановлення.

Після успішного завершення необхідно перевірити правильність встановлення командою «node -v» в командному рядку системи. Якщо не виникає помилок та коректно відображається обрана версія, то можна переходити до наступного кроку.

4.2.1.2 Встановлення системи Kai

Після встановлення Node.js, за допомогою пакетного менеджера npm можна встановити саму систему контролю версій за допомогою команди «npm install -g kai». Дана команда виконує:

- завантаження програми;
- встановлення необхідних залежностей;
- реєстрацію CLI-команди «kai» у системі.

Після цих дій робота з системою доступна з будь якої директорії в командному рядку.

4.2.2 Ініціалізація репозиторію

Після успішного встановлення можна починати повноцінну роботу з системою. Першим кроком для цього, в будь-якій системі контролю версій, є створення репозиторію. Ця дія відбувається за допомогою команди «kai init» з необхідного каталогу на комп'ютері користувача.

Дана команда створює службовий каталог з усіма необхідними піддиректоріями та файлами.

Приклад виклику команди та успішного створення репозиторію зображений на рисунку 4.3.

```
D:\test-repo>kai init
Initializing repository...
Repository created in D:\test-repo
D:\test-repo>
```

Рисунок 4.3 – Результат виконання команди «init»

Створена структура зображена на рисунку 4.4.

```
▼ test-repo
  ▼ .kai
    commits
    objects
```

Рисунок 4.4 – Структура репозиторію після виконання команди «init»

4.2.3 Додавання файлів та створення комітів

Приклад зображений на рисунку 4.5.

```
D:\test-repo>kai add test1.txt
File test1.txt is added for staging

D:\test-repo>kai add test2.txt
File test2.txt is added for staging

D:\test-repo>
```

Рисунок 4.5 – Результат додавання файлів

Після створення репозиторію можна починати роботу з файлами. Система Kai дозволяє роботу з усіма типами текстових файлів не зважаючи на розширення. Робота з зображеннями може бути не достатньо оптимізованою та продуктивною. Отже, після виконання необхідних для користувача дій з файлами, перед створенням коміту необхідно додати файли, які потрібно буде зафіксувати. Це реалізується за допомогою команди «`kai add <назва файлу>`». Успішне виконання операції буде підтверджено відповідним повідомленням.

Після виконання даної операції можна зафіксувати зміни за допомогою команди «`kai commit <повідомлення>`», приклад зображений на рисунку 4.6.

```
D:\test-repo>kai commit "first commit"
Committing files...
Committed 2 files

D:\test-repo>_
```

Рисунок 4.6 – Результат коміту

4.2.4 Перегляд історії комітів

Перед тим як використовувати команди, що потребують хеш коміту в якості аргументу, такі як команда відновлення версії або порівняння станів файлу, необхідно отримати необхідні хеші. Це можна зробити за допомогою команди «`kai log`». Приклад виконання команди зображений на рисунку 4.7.

```
D:\test-repo>kai log

24c8cb35de61cbc1f4cb11a6f495f46fe97ff0eb26875a5220dcb4a2b9571616
second commit
Mon May 11 2026

↑

136aeb62f0d94e2ffa5a882ca9c961c8b42d9bb78b66b19d35d53390a883609c
first commit
Mon May 11 2026

D:\test-repo>
```

Рисунок 4.7 – Вигляд історії комітів

4.2.5 Порівняння версій файлів

При бажанні порівняти версії файлу необхідно скористатись командою «`kai diff <назва файлу> <хеш коміту для порівняння>`». Дана команда при відсутності помилок (рисунок 4.8) в параметрах порівняє файли, створить HTML представлення результату та відкриє його в стандартній програмі для перегляду HTML файлів в системі. Представлення порівняння буде мати вигляд зображений на рисунку 4.9.

```
D:\test-repo>kai diff test1.txt 136aeb62f0d94e2ffa5a882ca9c961c8b42d9bb78b66b19d35d53390a883609c
Generating diff
Success
D:\test-repo>
```

Рисунок 4.8 – Результат успішного виконання команди порівняння версій

test1.txt [CHANGED]	
@@ -1,1 +1,2 @@	
1 - Перший коміт - перший файл	1 + Перший коміт - перший файл!!!!
	2 + Другий коміт - перший файл, зміна першого рядка ^

Рисунок 4.9 – Представлення порівняння версій файлу

4.2.6 Відновлення версій

При необхідності або бажанні відновити певну версію стану репозиторію необхідно скористатися командою «`kai checkout <хеш цільового коміту>`». Це відновить саме ту версію всіх файлів, збережених в необхідному коміті. Тож, при виконанні даної команди (рисунок 4.10), буде відновлено стан файлів до першого коміту (в даному прикладі). З рисунку 4.9 видно, що в другому коміті було додано другий рядок та змінено перший шляхом додавання знаків оклику, після виконання маємо стан файлу зображений на рисунку 4.11.

```
D:\test-repo>kai checkout 136aeb62f0d94e2ffa5a882ca9c961c8b42d9bb78b66b19d35d53390a883609c
Processing...
Restored 136aeb62f0d94e2ffa5a882ca9c961c8b42d9bb78b66b19d35d53390a883609c commit state
D:\test-repo>
```

Рисунок 4.10 – Виконання команди відновлення версії

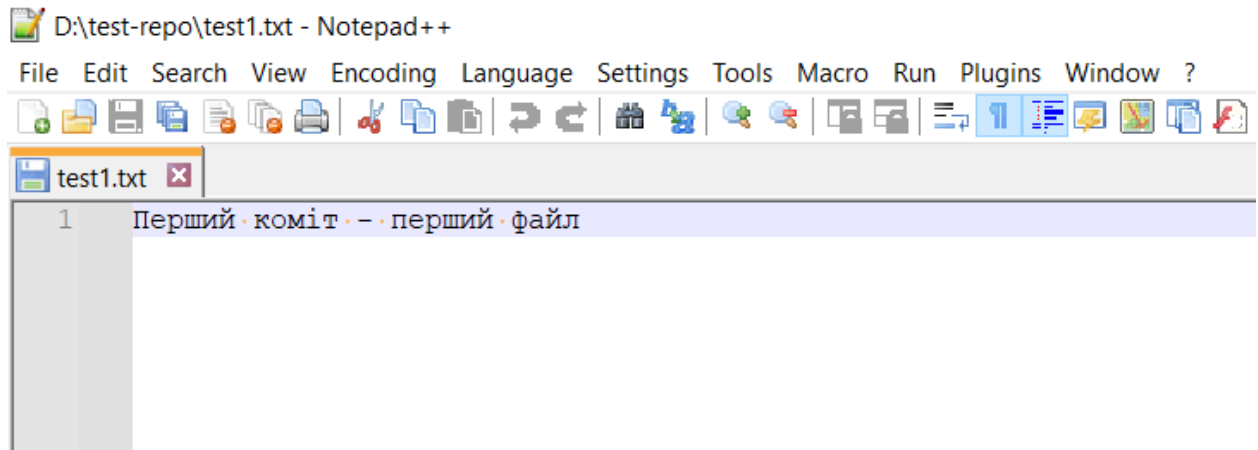


Рисунок 4.11 – Стан файлу після відновлення

4.3 Функціональне тестування системи

Функціональне тестування системи Kai проводилось з метою перевірки коректності роботи всіх команд та алгоритмів. В процесі тестування були виявлені помилки та вразливості, які в результаті були виправлені.

Тестування виконувалось у середовищі Windows 10, з використанням Node.js версії 24.10.0.

4.3.1 Тестування команд системи

Для перевірки працездатності системи було виконано тестування CLI-команд, результати тестування наведено в таблиці 4.1.

Таблиця 4.1 – Результати тестування команд системи Kai

Команда	Тестовий сценарій	Очкований результат	Результат
init	Створення нового репозиторію	Створення .kai з необхідним набором службових файлів та каталогів	Успішно
add	Додавання відповідного файлу до index	Файл додається	Успішно
commit	Створення коміту	Створення коміт-файлу	Успішно
log	Відображення історії	Виведення списку комітів ланцюгом	Успішно
diff	Порівняння версій	Генерація та відображення diff файлу	Частково успішно
checkout	Відновлення версії	Відновлення файлів відповідно до версії	Частково успішно

В результаті тестів було виявлено певні помилки. Вони не були критичними і, не дивлячись на них, команди виконували свою основну функцію. Але були проблеми з командами «diff» та «checkout». Проблеми були пов'язані суто з особливостями роботи Node.js та бібліотеки «open».

Так, команда «diff» створювала необхідний файл, але не відкривала його, коренем даної проблеми стало те, що в найновішій версії бібліотеки «open» потрібно передавати додатковий параметр в одноіменну функцію, але даний параметр не повинен мати такого впливу, тому було прийняте рішення встановити більш стабільну версію бібліотеки.

Команда «checkout» відновлювала версію цілком коректно, але не видаляла поточні відстежувані файли, що призводило до некоректного вигляду репозиторію в окремих випадках. Рішенням даної проблеми стало використання іншої, однаковою за суттю, функції видалення. Першою була використана новіша та більш універсальна, але, за якоїсь причини, не

виконувала свою основну функцію в середовищі, що було використано для розробки, тому в результаті було використано іншу, старішу, але більш низькорівневу та надійну функцію.

Загалом результати тестування виявились доволі успішними і продемонстрували майже ідеальну роботу системи без великої кількості правок.

4.3.2 Тестування обробки помилок

Окремо було проведено тестування обробки помилок, оскільки, беручи до уваги принцип дії програми, успішність виконання функцій багато в чому залежить від коректності вводу користувача, а, враховуючи відсутність миттєвих підказок, шанс людської похибки є достатньо значним.

Результати тестування наведено в таблиці 4.2.

Таблиця 4.2 – Результати тестування обробки помилок

Ситуація	Очікуване повідомлення	Результат
1	2	3
Виконання будь-якої команди за межами репозиторію	Not a repository	Успішно
Відновлення версії до неіснуючого коміту	There is no commit with <наданий хеш> hash	Успішно
Додавання неіснуючого файлу	File <назва файлу> does not exist	Успішно
Коміт без файлів	Nothing to commit	Успішно

Дане тестування продемонструвало повну коректність роботи системи обробки помилок.

4.3.3 Аналіз результатів тестування

За результатами тестування було виконано 10 основних та 8 більш детальних тестових сценаріїв (загалом 18).

Результати тестування такі:

- успішно пройдено: 16;
- виявлено помилок: 2;
- виправлено помилок: 2.

Як вже було зазначено раніше, всі виявлені проблеми були пов'язані з особливостями роботи Node.js та бібліотек, а причиною припущення цих помилок стала недосконалість знань розробника про особливість роботи з цими модулями. В кінцевому результаті всі виявлені помилки було швидко усунуто та забезпечено повністю коректну роботу системи.

Статистика тестування представлена на рисунку 4.12.

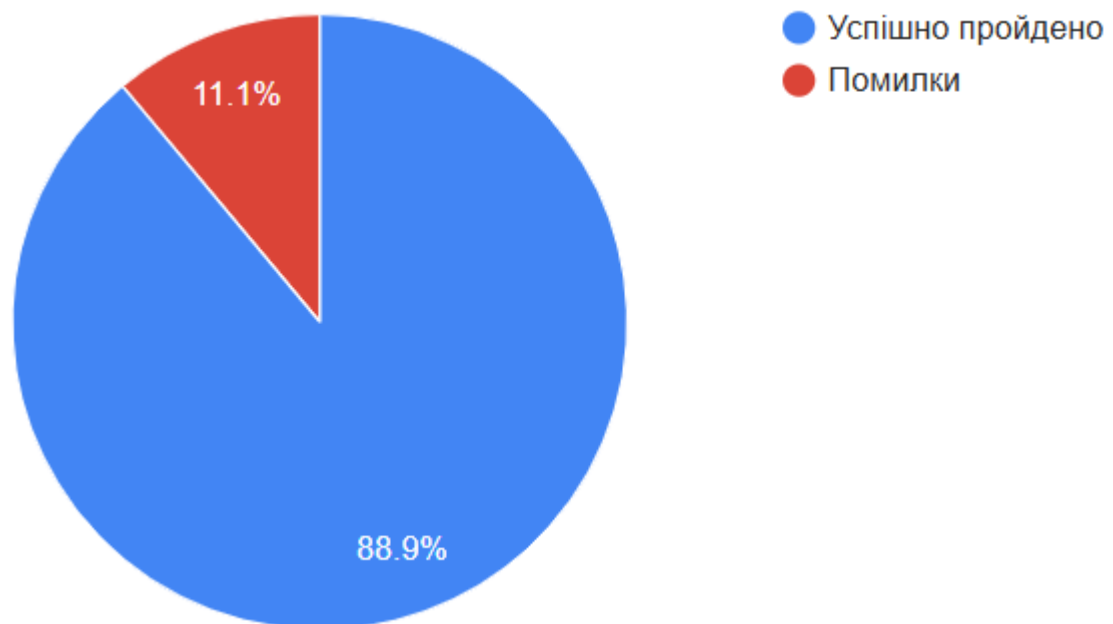


Рисунок 4.12 – Статистика результатів тестування

4.3.4 Аналіз продуктивності системи

Дуже важливою складовою систем контролю версій є їхня швидкодія. З вибору мови програмування для розробленої системи можна легко зробити висновок, що швидкість роботи алгоритмів буде значно нижчою в порівнянні з компільованими мовами типу C++, але, оскільки система просто не розрахована на роботу не з текстовими файлами, результати демонструють, що для людського ока різниця в швидкості при роботі з дуже маленькими файлами (до 1 Кб) та великими (5 Мб – для простого текстового документу це дуже багато) майже непомітна.

Тестування проводилось на ПК з такими парметрами:

- процесор: Intel Core i5 12 потоків, до 4,4 Гц
- оперативна пам'ять: 16 Гб;
- накопичувач: SSD M2.

Для більш слабких або потужніших комп'ютерів результати можуть відрізнятися, але в відношенні зі швидкодією інших процесів будуть однаковими.

Результати тестування швидкості коміту (add + commit) приведено у таблиці 4.3.

Таблиця 4.3 – Результати тестування швидкості коміту

Розмір файлу	Швидкість
1 Кб	~40 мс
100 Кб	~60 мс
1 Мб	~120 мс
5 Мб	~400 мс

Графік зміни швидкодії в залежності від розміру файлу представлений на рисунку 4.13.

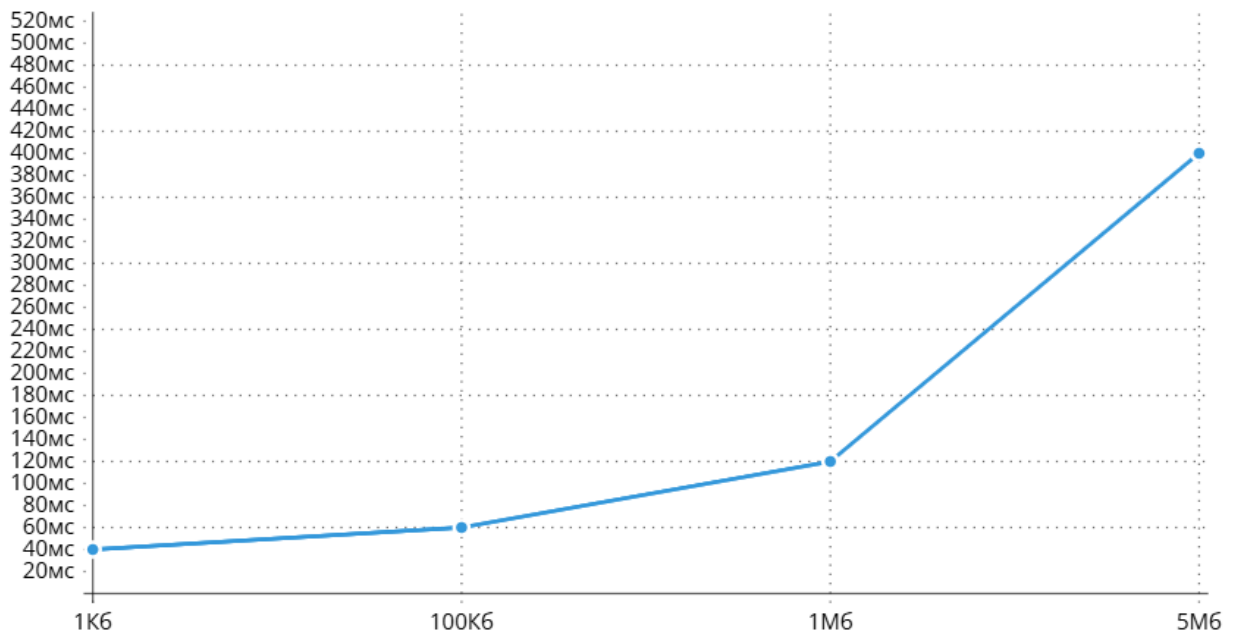


Рисунок 4.13 – Графік зміни швидкості коміту в залежності від об'єму файлів

4.4 Рекомендації щодо використання

Грунтуючись на проведеному тестуванні та результатах експериментального дослідження, було сформовано певні рекомендації щодо застосування розробленої системи контролю версій Kai.

Перш за все потрібно гарантувати повне дотримання програмних вимог використання системи. Використання версії Node.js нижче за 18, може призвести до неочікуваних результатів роботи та помилок. З точно перевірених даних відомо, що версія нижче за 13-ту гарантовано призведе до помилок при визначенні різниці між версіями файлів та при додаванні файлу до відстеження через відсутність однієї з необхідних функцій даних версій. Їхня підтримка нерелевантна (хоча і можлива), бо при покращенні системи та додаванні нового функціоналу це може стати великою проблемою або сильно вдарити по оптимізації через використання неефективних та старих підходів до роботи з системою.

Також, спроби ручної зміни службової директорії можуть призвести до незворотних наслідків, хоча в якості дослідження, при дотриманні всіх схем

роботи, наданих в даній роботі, правильні ручні зміни можуть якісно впливати на розуміння роботи систем контролю версій.

З практичних рекомендацій важливо відзначити, що робота з будь-якими нетекстовими файлами не підтримується (не була запланована та, відповідно, не тестувалася). Теоретично це цілком можливо, але заради цілісності даних категорично не рекомендується.

Задля правильності збереження станів, треба пам'ятати що Kai не відстежує зміни в файлах, в коміт буде додано саме той стан, який був збережений при додаванні файлу, тому додавання файлу слід виконувати безпосередньо перед комітом. Така поведінка особливо має сенс за наявності графічного інтерфейсу, такому як в середовищі розробки WebStorm – файли обираються в момент коміту, а не задовго до цього моменту, але при відсутності явного графічного представлення необхідно звикнути до цього.

4.5 Плани з розвитку

Оскільки система є експериментальною, вона має безліч шляхів розвитку. Її архітектура дозволяє просте створення нового та доповнення старого функціоналу. З найближчих «контрольних точок» є реактивне відстеження змін в файлах, інтеграція системи гілок, можливість додавання декількох файлів одразу, представлення повного вигляду репозиторію зі статусами файлів тощо.

Але варто пам'ятати, що кожна з цих змін зрештою призведе до втрати дослідницької мети проєкту. Все це буде наближати її до Git, а це взагалі не має жодного сенсу. Створення аналогу, або правильніше – копії, Git призведе тільки до посилення навичок програмування розробника, що не є метою цього проєкту. Звісно, можна покращити вже присутній функціонал, щоб, навіть без підтримки графічного інтерфейсу, наблизитись до його зручності, але результат виконаної роботи буде дуже далеким від первинної мети.

Реальний сенс створення нової системи контролю версій не задля експерименту та дослідження – це конкурентоспроможність з гігантом та монополістом в цій сфері – Git. Для реалізації цієї мети необхідно винайти кардинально інший підхід до контролю версій. В даному ж випадку, розроблена система відкриває завісу секретів роботи найпопулярнішої системи, яку багатьом важко зрозуміти.

Підбиваючи підсумок, вважаю розроблену систему Kai ідеальною для виконання тих цілей, які були перед нею поставлені.

ВИСНОВКИ

В ході роботи було вирішено актуальну задачу дослідження та реалізації базових алгоритмів систем контролю версій документів. В результаті для досягнення мети, було створено власну Git-подібну систему контролю версій «Kai». Розроблена система реалізує всі фундаментальні алгоритми таких систем та дозволяє досліджувати принципи їх роботи на практиці, без зайвих ускладнень, притаманних комерційним системам, які під шарами абстракцій та «синтаксичного цукру» інкапсулюють свої механізми та ускладнюють користувачам їх розуміння.

Проведено аналіз сучасних систем контролю версій, таких як: Git, Apache Subversion та Perforce. В результаті дослідження виявлено, що всі вони використовують складні алгоритми збереження, хешування та визначення змін між версіями, а деякі з них мають в корені дуже складну архітектуру системи, що робить їх використання незручним та саме через це вони не користуються такою популярністю як Git (який є безумовним лідером в своїй сфері, фактично не маючи конкурентів). Але, не дивлячись на всю цю популярність, відкриті джерела містять замало інформації саме про принципи роботи таких систем, що підтверджує актуальність дослідження.

Спроектовано та реалізовано архітектуру системи Kai за модульним принципом, що з самого початку дало фундамент для простої та зрозумілої розробки, з можливістю простого доповнення функціоналу. Крім того, така структура гарантує безпеку виконання та мінімізує вірогідність та критичність помилок. Kai створено мовою TypeScript (JavaScript) в середовищі Node.js. Як і у випадку Git, всі необхідні дані зберігаються локально на ПК користувача. Об'єкти хешуються алгоритмом SHA-256, якого цілком достатньо для гарантії відсутності дублювань при різних вхідних даних.

Проведено комплексне тестування розробленого застосунку, де були виявлені декілька некритичних помилок в роботі основних функцій, які було швидко та надійно виправлено. Крім функціонального тестування,

проводилось також тестування продуктивності системи, чи відповідає вона вимогам швидкодії, які б залишили користувача без зайвих питань щодо оптимізації. І, не дивлячись на використання не найшвидшої мови програмування, яка, на превеликий жаль, не має підтримки роботи з потоками та не є компільованою, швидкість виконання всіх операцій є цілком достатньою. Виконання операцій з великими файлами не займає більше пів секунди, що є чудовим результатом. Звісно, при роботі з великою кількістю об'ємних файлів одночасно, вже можна відчутти, що дії виконуються не миттєво. Але, жодна команда наразі не має підтримки роботи з декількома файлами одночасно, що просто вбиває проблему швидкодії.

На основі експериментального дослідження виявлено, що основною цільовою аудиторією розробленої системи можуть стати студенти (35%), розробники початківці (25%), користувачі, що вивчають Git (20%) та викладачі та дослідники (20%), тобто всі групи користувачів, яких цікавить саме дослідження та навчання, а не розробка стартапів та інші види комерційної діяльності.

Сформовано можливі шляхи для вдосконалення системи, та в той же час виявлено, що завелика кількість нового функціоналу, оптимізації та підходів до забезпечення цілісності та безпеки даних, віддаляє суть застосунку від його початкової мети.

Розроблений застосунок чітко та успішно виконує всі свої функції як дослідницької, так і системи контролю версій. В межах роботи детально розібрано принципи та алгоритми роботи систем контролю версій, що дає їх повне розуміння. Таким чином, мета роботи досягнена, а завдання успішно виконано.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Git. URL: <https://git-scm.com/> (date of access: 11.05.2026).
2. Apache Subversion. URL: <https://subversion.apache.org/> (date of access: 11.05.2026).
3. Perforce. URL: <https://www.perforce.com/p/vcs/free-version-control> (date of access: 11.05.2026).
4. GitHub vs. Bitbucket vs. GitLab. URL: <https://rewind.com/blog/github-vs-bitbucket-vs-gitlab-comparison/> (date of access: 12.05.2026).
5. SHA-2 – Wikipedia. URL: <https://en.wikipedia.org/wiki/SHA-2> (date of access: 08.05.2026).
6. Modular software architecture. URL: <https://pretius.com/blog/modular-software-architecture> (date of access: 16.05.2026).
7. TypeScript: JavaScript with syntax for types. URL: <https://www.typescriptlang.org/> (date of access: 16.05.2026).
8. Introduction to Node.js. URL: <https://nodejs.org/learn/getting-started/introduction-to-nodejs#introduction-to-nodejs> (date of access: 16.05.2026).
9. WebStorm: The JavaScript and TypeScript IDE. URL: <https://www.jetbrains.com/webstorm/promo/> (date of access: 16.05.2026).
10. Node.js – Run JavaScript Everywhere. URL: <https://www.jetbrains.com/webstorm/promo/> (date of access: 16.05.2026).

ДОДАТОК А
Технічне завдання

Вступ

Система контролю версій «Kai» призначена для дослідження роботи Git-подібних (розподілених) систем контролю версій. За рахунок своєї простоти та відсутності шифрування всіх службових даних, кожен користувач може наочно побачити всі внутрішні механізми роботи таких систем. Область застосування охоплює навчання та дослідження роботи систем контролю версій. За рахунок відсутності великої кількості надбудов вона має ідеально демонструвати як саме працює збереження, порівняння та відновлення версій файлів.

A.1 Підстави до розробки

Підставою для розробки є завдання на дипломну роботу, видане 07 квітня 2026 року. Найменування теми розробки – «Дослідження та програмна реалізація алгоритмів контролю версій документів».

A.2 Призначення розробки

Застосунок призначений для дослідження роботи систем контролю версій, але при цьому може успішно виконувати всі базові функції таких систем. Його архітектура та система зберігання повинна дозволяти користувачам наочно бачити всі кроки виконання функціоналу контролю версій, а також, з точки зору користувацького інтерфейсу, не мати суттєвих відмінностей від реальних аналогів (таких як Git), з метою реальної практичної демонстрації, позбавленої зайвих спрощень та умовностей.

A.3 Вимоги до програми чи програмного виробу

A.3.1 Вимоги до функціональних характеристик

Система контролю версій «Kai» повинна виконувати всі фундаментальні функції систем контролю версій, а саме:

- створення репозиторію;
- додавання файлів;
- створення комітів;
- перегляд історії;
- відновлення версій;
- демонстрація різниці версій.

Дані функції є базою та фундаментом контролю версій, всі інші надбудови корисні, але мають на меті розширення та покращення вищеперерахованого функціоналу, глобально нових функцій вони не реалізують.

Для контролю версій документів важливо слідкувати за станом всього репозиторію та окремих файлів, тому важливо дотримуватись безпечної роботи з документами та мати потужну систему роботи з помилками. Оскільки дані системи мають консольний інтерфейс, перш за все необхідно забезпечити коректну роботу модулю обробки команд, бо це єдина та найголовніша частина взаємодії з користувачем, який, не маючи багато досвіду, здатен припускатися великої кількості помилок. Таким чином, перед виконанням дій з файлами важливо впевнитись, що за відсутності форсмажорів, пов'язаних з діями, що мають апаратну залежність або інші фактори, незалежні від програмного коду, вся функція буде виконана повністю, тобто всі вхідні дані повинні бути перевірені з самого початку і забезпечити повну цілісність даних.

A.3.2 Вимоги до надійності

Для забезпечення надійного функціонування застосунку необхідно дотримуватися кількох умов. З точки зору програми, всі запобіжні заходи вже

прийняті, але система не інкапсулює всі внутрішні механізми за користувацьким інтерфейсом. Службові каталоги та файли, не дивлячись на свою прихованість, все ще доступні користувачам. Ручна зміна службових файлів без досконалого знання системи може призвести до незворотної втрати файлів.

Окрім того, спроби перервати процес під час виконання команд системи можуть призвести не тільки до неуспішного виконання поточної функції, але й до порушення цілісності системи як такої, що буде потребувати створення репозиторію з самого початку.

Тож, не дивлячись на відкритість системи, задля дослідницьких цілей, потрібно обережно ставитись до її внутрішньої будови.

A.3.3 Умови експлуатації

Застосунок повинен працювати стабільно при стандартних кліматичних умовах, притаманних пристрою користувача. Користувач має володіти трохи більше ніж базовими навичками роботи з ПК, оскільки програма має консольний інтерфейс та потребує базового розуміння роботи з програмами, що позбавлені класичних кнопок та інших графічних форм взаємодії. Обслуговування програми не потребує спеціального персоналу, лише періодичне оновлення через пакетний менеджер npm.

A.3.4 Вимоги до складу параметрів технічних засобів

Для роботи застосунку необхідний мобільний пристрій із такими мінімальними характеристиками:

- операційна система: Windows, MacOS або Linux (будь яка версія, що підтримує Node.js 18+);
- процесор: частота не нижче 1.1 МГц;
- оперативна пам'ять: мінімум 256 Мб;
- вільне місце на диску: не менше 100 Мб.

A.3.5 Вимоги до інформаційної й програмної сумісності

Система контролю версій «Kai» створена на базі середовища Node.js, яке є абсолютно кросплатформенним та сумісним з будь якою іншою програмою. Єдина умова сумісності – підтримка системою необхідної версії середовища (18+).

Ця версія поєднує в собі нові технології, цілком достатні для роботи такого застосунку, та відсутність застарілих рішень до роботи з файлами, хешування та інших важливих процесів роботи.

Робота з файлами підлаштовується під будь-яку операційну систему, не потребуючи зміну коду під кожну з них (не дивлячись на серйозні відмінності робити з файлами в Window та MacOS, Linux).

A.3.6 Вимоги до маркування й упакування

Застосунок розповсюджується через пакетний менеджер npm. Вимоги до фізичного маркування чи пакування відсутні, оскільки продукт є цифровим.

A.3.7 Вимоги до транспортування й збереження

Оскільки застосунок розповсюджується в цифровому вигляді, вимоги до транспортування та збереження не застосовуються.

A.4 Вимоги до програмної документації

Програмна документація до застосунку включає пояснювальну записку та технічне завдання. Ці документи містять вичерпну інформацію про розробку, функціонал і вимоги до програми.

A.5 Техніко–економічні показники

Розроблена система контролю версій «Kai» має навчально–дослідницьке призначення та орієнтована на демонстрацію принципів роботи Git-подібних систем. Використання програми дозволяє спростити процес вивчення алгоритмів контролю версій, механізмів збереження змін та визначення різниці між версіями документів. На відміну від теоретичного вивчення, система надає можливість практично дослідити роботу основних механізмів систем контролю версій, що підвищує ефективність навчального процесу.

A.6 Стадії й етапи розробки

Розробка включає наступні етапи:

- постановка завдання (1 тиждень);
- аналіз предметної області (1–2 тижні);
- розробка архітектури (1 тиждень);
- розробка програми (1–2 тижні);
- тестування (1 тиждень);
- оформлення документації (1 тиждень).

A.7 Порядок контролю та приймання

Процес перевірки працездатності застосунку передбачає кілька етапів тестування. Найголовніше – це функціональне тестування, яке перевіряє коректність роботи усіх команд при різних (в тому числі неправильних) вхідних даних.

Не менш важливим є тестування продуктивності, оскільки система контролю версій має зберігати статус допоміжного інструменту та не відволікати увагу користувача на особливості та швидкість своєї роботи.

Найважливіша частина тестування – тестування безпеки та надійності. Будь-яка людська похибка не повинна призводити до порушення цілісності даних та подібних випадків. Якщо виникає помилка, користувач повинен отримати детальний та зрозумілий опис того, що він робить не так, або де зламалась система. Абсолютно неприпустимо отримувати занадто технічні помилки, які свідчать про проблеми реалізації, оскільки таких проблем бути не повинно.

ДОДАТОК Б
Текст програми

Б.1 Файл cli/index.ts

Цей файл відповідає за обробку введених консольних команд:

```
import { Command } from 'commander';
import {
  addCommand,
  checkoutCommand,
  commitCommand,
  diffCommand,
  initCommand,
  logCommand,
} from '../commands';

const commander = new Command();

commander
  .name('kai')
  .description('Kai version control system')
  .version('1.0.0');

commander
  .command('init')
  .description('Initialize repository')
  .action(initCommand);

commander
  .command('add <filename>')
  .description('Add file to staging')
  .action(addCommand);

commander
  .command('commit <message>')
  .description('Commit staged files')
  .action(commitCommand);

commander
  .command('log')
  .description('Log repository history')
  .action(logCommand);

commander
  .command('checkout <targetCommit>')
  .description('Restore needed state')
  .action(checkoutCommand);

commander
  .command('diff <fileName> <targetCommit>')
  .description('Show difference between file versions')
  .action(diffCommand);

export { commander };
```

Б.2 Файл add.ts

Цей файл оброблює команду додавання файлів:

```
import { errorHandler } from '../utils';
import { addFile } from '../core/repository';

const addCommand = (fileName: string) => {
  errorHandler(() => {
    addFile(fileName);

    console.log(`File ${fileName} is added for
staging`);
  });
};

export { addCommand };
```

Б.3 Файл checkout.ts

Цей файл оброблює команду відновлення версії:

```
import { errorHandler } from '../utils';
import { checkoutToCommit } from '../core/repository';

const checkoutCommand = (targetCommit: string) => {
  errorHandler(() => {
    console.log('Processing...');

    checkoutToCommit(targetCommit);

    console.log(`Restored ${targetCommit} commit
state`);
  });
};

export { checkoutCommand };
```

Б.4 Файл commit.ts

Цей файл оброблює команду коміту:

```
import { errorHandler } from '../utils';
import { commitFiles } from '../core/repository';

const commitCommand = (message: string) => {
  errorHandler(() => {
    console.log('Committing files...');
```

```

        const filesNumber = commitFiles(message);

        console.log(`Committed ${filesNumber} files`);
    });
};

export { commitCommand };

```

Б.5 Файл `commands/diff.ts`

Цей файл оброблює команду визначення версії:

```

import { errorHandler } from '../utils';
import path from 'path';
import { showDiff } from '../core/repository';

const diffCommand = (fileName: string, targetCommitHash:
string) => {
    errorHandler(() => {
        console.log('Generating diff');

        const filePath = path.join(process.cwd(),
fileName);

        showDiff(filePath, targetCommitHash);
    });
};

export { diffCommand };

```

Б.6 Файл `init.ts`

Цей файл оброблює команду ініціалізації репозиторію:

```

import { initRepo } from '../core/repository';
import { errorHandler } from '../utils';

const initCommand = () => {
    errorHandler(() => {
        console.log('Initializing repository...');

        initRepo();

        console.log(`Repository created in
${process.cwd()}`);
    });
};

```

```
export { initCommand };
```

Б.7 Файл `log.ts`

Цей файл оброблює команду виведення історії версій:

```
import { errorHandler } from '../utils';
import { logHistory } from '../core/repository';

const logCommand = () => {
  errorHandler(() => {
    logHistory();
  });
};

export { logCommand };
```

Б.8 Файл `commands/index.ts`

Цей файл експортує обробники команд:

```
export * from './init';
export * from './add';
export * from './commit';
export * from './log';
export * from './checkout';
export * from './diff';
```

Б.9 Файл `core/diff.ts`

Цей файл створює HTML-представлення різниці версій документів:

```
import { createTwoFilesPatch } from 'diff';
import { html } from 'diff2html';
import { writeFile } from '../utils';

interface Props {
  diffFilePath: string;
  fileName: string;
  fileOldContent: string;
  fileNewContent: string;
}

const createDiffFile = ({
  diffFilePath,
  fileName,
```

```

        fileOldContent,
        fileNewContent,
    }: Props) => {
        try {
            const patch = createTwoFilesPatch(
                fileName,
                fileName,
                fileOldContent,
                fileNewContent
            );

            const htmlDiff = html(patch, {
                drawFileList: false,
                matching: 'lines',
                outputFormat: 'side-by-side',
            });

            writeFile(
                diffFilePath,
                `
                <html lang='en'>
                    <head>
                        <link
                            rel="stylesheet"
href="https://cdn.jsdelivrivr.net/npm/diff2html/bundles/css/di
ff2html.min.css"
                        />
                    </head>
                    <body>
                        ${htmlDiff}
                    </body>
                </html>
                `
            );

            return { success: true };
        } catch {
            return { success: false };
        }
    };

export { createDiffFile };

```

Б.9 Файл head-handler.ts

Даний файл містить функції для контролю файлу HEAD:

```

import path from 'path';
import { HEAD } from '../../consts';
import { isPathExists, readFile, writeFile } from

```

```

'../utils';

const getHead = (repoPath: string) => {
    const headPath = path.join(repoPath, HEAD);

    if (!isPathExists(headPath)) return null;

    return readFile(headPath);
};

const setHead = (repoPath: string, hash: string) => {
    const headPath = path.join(repoPath, HEAD);

    return writeFile(headPath, hash);
};

export { getHead, setHead };

```

Б.10 Файл index-handler.ts

Цей файл містить функції для контролю index-файлу:

```

import path from 'path';
import { readFile, writeFile } from '../utils';
import { INDEX_FILE } from '../../consts';

const readIndex = (repoPath: string): Record<string,
string> => {
    const indexPath = path.join(repoPath, INDEX_FILE);

    try {
        return JSON.parse(readFile(indexPath) || '');
    } catch {
        return {};
    }
};

const writeIndex = (repoPath: string, index: Record<string,
string>) => {
    const indexPath = path.join(repoPath, INDEX_FILE);

    writeFile(indexPath, JSON.stringify(index, null, 2));
};

export { readIndex, writeIndex };

```

Б.11 Файл object-handler.ts

Цей файл містить функції для роботи з об'єктами:

```
import path from 'path';
import zlib from 'zlib';
import { isPathExists, readFileAsBuffer, writeFile } from
'../utils';

const saveObject = (repoPath: string, hash: string,
content: string) => {
    const objectPath = path.join(repoPath, 'objects',
hash);

    if (!isPathExists(objectPath)) {
        const compressedContent =
zlib.deflateSync(content);
        writeFile(objectPath, compressedContent);
    }
};

const readObject = (repoPath: string, hash: string) => {
    const objectPath = path.join(repoPath, 'objects',
hash);

    if (!isPathExists(objectPath)) throw `Cannot read
object ${hash}`;

    const objectData = readFileAsBuffer(objectPath);

    if (!objectData) return '';

    return zlib.inflateSync(objectData).toString('utf-8');
};

export { saveObject, readObject };
```

Б.12 Файл repository.ts

Цей файл містить головні функції для безпосередньої роботи з репозиторієм:

```
import path from 'path';
import {
    createDirectory,
    deleteFile,
    hashContent,
    isPathExists,
    readFile,
```

```

        writeFile,
    } from '../utils';
import {
    COMMITS_DIR,
    DIFF_FILE,
    INDEX_FILE,
    OBJECTS_DIR,
    REPO_DIR,
} from '../../consts';
import { readObject, saveObject } from './object-handler';
import { readIndex, writeIndex } from './index-handler';
import { execSync } from 'node:child_process';
import { getHead, setHead } from './head-handler';
import { CommitData } from '../../types';
import { createDiffFile } from './diff';
import open from 'open';

const findRepoRoot = () => {
    let currentPath = process.cwd();

    while (true) {
        if (isPathExists(path.join(currentPath, REPO_DIR)))
        {
            return currentPath;
        }

        const parent = path.dirname(currentPath);

        if (parent === currentPath) return null;

        currentPath = parent;
    }
};

const getRepoPath = () => {
    const repoRoot = findRepoRoot();

    if (!repoRoot) {
        throw Error('Not a repository');
    }

    return path.join(repoRoot, REPO_DIR);
};

const initRepo = () => {
    if (findRepoRoot()) {
        throw Error('Repository already exists');
    }

    const repoPath = path.join(process.cwd(), REPO_DIR);
    const objectsPath = path.join(repoPath, OBJECTS_DIR);
    const commitsPath = path.join(repoPath, COMMITS_DIR);
    const indexPath = path.join(repoPath, INDEX_FILE);

```

```

    createDirectory(repoPath);
    createDirectory(objectsPath);
    createDirectory(commitsPath);

    writeFile(indexPath, JSON.stringify({}, null, 2));

    if (process.platform === 'win32') {
        execSync(`attrib +h "${repoPath}"`);
    }
};

const addFile = (fileName: string) => {
    const repoPath = getRepoPath();

    const filePath = path.join(process.cwd(), fileName);

    if (!isPathExists(filePath)) {
        throw Error(`File ${filePath} does not exist`);
    }

    const fileContent = readFile(filePath) || '';

    const hash = hashContent(fileContent);

    const indexContent = readIndex(repoPath);

    const relativePath = path
        .relative(path.dirname(repoPath), filePath)
        .replaceAll('\\', '/');

    if (indexContent[relativePath] &&
        indexContent[relativePath] === hash) {
        throw `File ${fileName} already staged`;
    }

    saveObject(repoPath, hash, fileContent);

    indexContent[relativePath] = hash;

    writeIndex(repoPath, indexContent);
};

const readCommit = (repoPath: string, commitHash: string)
=> {
    const commitPath = path.join(repoPath, COMMITS_DIR,
        commitHash);

    if (!isPathExists(commitPath))
        throw Error(`There is no commit with ${commitHash}
hash`);

    return JSON.parse(readFile(commitPath) || '') as

```

```

CommitData;
};

const commitFiles = (message: string) => {
  const repoPath = getRepoPath();

  const indexData = readIndex(repoPath);

  if (!indexData || Object.keys(indexData).length === 0)
  {
    throw Error('Nothing to commit');
  }

  const head = getHead(repoPath);

  let prevFiles: Record<string, string> = {};

  if (head) {
    prevFiles = readCommit(repoPath, head).files;
    prevFiles = Object.fromEntries(
      Object.entries(prevFiles).filter(([file]) => {
        return
isPathExists(path.join(path.dirname(repoPath), file));
      })
    );
  }

  const commitData: CommitData = {
    message: message,
    timestamp: Date.now(),
    parent: head,
    files: { ...prevFiles, ...indexData },
  };

  const commitId =
hashContent(JSON.stringify(commitData));

  const commitPath = path.join(repoPath, COMMITS_DIR,
commitId);

  writeFile(commitPath, JSON.stringify(commitData));
  writeIndex(repoPath, {});
  setHead(repoPath, commitId);

  return Object.keys(indexData).length;
};

const logHistory = () => {
  const repoPath = getRepoPath();
  let currentCommit = getHead(repoPath);

  if (!currentCommit) {
    console.log('History is empty');
  }

```

```

        return;
    }

    while (true) {
        const currentCommitData = readCommit(repoPath,
currentCommit);

        const dataToPrint = [
            currentCommit,
            currentCommitData.message,
            new
Date(currentCommitData.timestamp).toDateString(),
        ];
        const maxLength = Math.max(...dataToPrint.map((i)
=> i.length));

        // top line
        console.log(`┌${'─'.repeat(maxLength + 2)}┐`);

        dataToPrint.forEach((i) => {
            const padded = i.padEnd(maxLength, ' ');
            console.log(`| ${padded} |`);
        });

        // bottom line
        console.log(`└${'─'.repeat(maxLength + 2)}┘`);

        if (!currentCommitData.parent) return;

        console.log('↑'.padStart(Math.floor((maxLength + 4)
/ 2) + 1, ' '));

        currentCommit = currentCommitData.parent;
    }
};

const checkoutToCommit = (targetCommitHash: string) => {
    const repoPath = getRepoPath();
    const head = getHead(repoPath);
    if (!head) throw 'Repository is empty';

    const headData = readCommit(repoPath, head);
    const targetCommitData = readCommit(repoPath,
targetCommitHash);

    Object.keys(headData.files).forEach((filePath) => {
        deleteFile(path.join(path.dirname(repoPath),
filePath));
    });

    Object.entries(targetCommitData.files).forEach(([filePath,
fileHash]) => {

```

```

        const fullFilePath =
path.join(path.dirname(repoPath), filePath);

        createDirectory(path.dirname(fullFilePath));
        writeFile(fullFilePath, readObject(repoPath,
fileHash));
    });

    writeIndex(repoPath, {});
    setHead(repoPath, targetCommitHash);
};

const showDiff = (filePath: string, targetCommitHash:
string) => {
    const repoPath = getRepoPath();

    if (!isPathExists(filePath)) {
        throw Error(`File ${path.basename(filePath)} does
not exist`);
    }

    const relativeFilePath = path
        .relative(path.dirname(repoPath), filePath)
        .replaceAll('\\', '/');

    const targetCommitData = readCommit(repoPath,
targetCommitHash);
    const targetCommitFileHash =
        targetCommitData.files[relativeFilePath] ?? null;

    const newFileContent = readFile(filePath);

    if (!newFileContent) return;

    const oldFileContent = !targetCommitFileHash
        ? ''
        : readObject(repoPath, targetCommitFileHash);

    const diffFilePath = path.join(repoPath, DIFF_FILE);

    const diffResult = createDiffFile({
        diffFilePath: diffFilePath,
        fileName: path.basename(relativeFilePath),
        fileOldContent: oldFileContent,
        fileNewContent: newFileContent,
    });

    if (diffResult.success) {
        open(diffFilePath).then(() => {
            console.log('Success');
        });
    } else {
        throw 'Error while creating diff';
    }
}

```

```

    }
  };

export {
  findRepoRoot,
  initRepo,
  addFile,
  commitFiles,
  logHistory,
  checkoutToCommit,
  showDiff,
};

```

Б.13 Файл error-handler.ts

Цей файл містить функцію для глобальної обробки помилок:

```

const errorHandler = <T, Args extends any[]>(
  func: (...args: Args) => T,
  ...args: Args
) => {
  try {
    return func(...args);
  } catch (error: unknown) {
    if (typeof error === 'string') {
      console.log(error);
    } else {
      console.log(`Error occurred: ${error as
Error}.message`);
    }
    return null;
  }
};

export { errorHandler };

```

Б.14 Файл fs.ts

Цей файл містить функції для роботи з файловою системою:

```

import fs from 'fs';
import { errorHandler } from '../error-handler';

const isPathExists = (path: string) => {
  return fs.existsSync(path);
};

const createDirectory = (path: string) => {
  return errorHandler(() => fs.mkdirSync(path, {

```

```

recursive: true }));
};

const writeFile = (path: string, content: string |
NodeJS.ArrayBufferView) => {
    return errorHandler(() => fs.writeFileSync(path,
content, 'utf-8'));
};

const readFile = (path: string) => {
    return errorHandler(() => fs.readFileSync(path, 'utf-
8'));
};

const readFileAsBuffer = (path: string) => {
    return errorHandler(() => fs.readFileSync(path));
};

const deleteFile = (path: string) => {
    return errorHandler(() => fs.unlinkSync(path));
};

export {
    isPathExists,
    createDirectory,
    writeFile,
    readFile,
    readFileAsBuffer,
    deleteFile,
};

```

Б.15 Файл hash.ts

Цей файл містить функцію хешування:

```

import crypto from 'crypto';

const hashContent = (content: string) => {
    return
crypto.createHash('sha256').update(content).digest('hex');
};

export { hashContent };

```

Б.16 Файл `utils/index.ts`

Цей файл експортує підмодулі службового модулю:

```
export * from './fs';
export * from './error-handler';
export * from './hash';
```

Б.17 Файл `consts.ts`

Цей файл містить всі необхідні в проєкті константи:

```
const REPO_DIR = '.kai';
const OBJECTS_DIR = 'objects';
const COMMITS_DIR = 'commits';
const INDEX_FILE = 'index.json';
const HEAD = 'HEAD';
const DIFF_FILE = 'diff.html';

export { REPO_DIR, OBJECTS_DIR, COMMITS_DIR, INDEX_FILE,
  HEAD, DIFF_FILE };
```

Б.18 Файл `types.ts`

Цей файл містить всі необхідні в проєкті нестандартні типи:

```
export interface CommitData {
  message: string;
  timestamp: number;
  parent: string | null;
  files: Record<string, string>;
}
```

Б.19 Файл `index.ts`

Цей файл є точкою входу в програму та викликає обробник команд:

```
#!/usr/bin/env node

import { commander } from './modules/cli';

commander.parse();
```

Б.20 Файл package.json

Цей файл містить всі залежності проекту:

```
{
  "name": "kai-version-control",
  "version": "1.0.0",
  "description": "",
  "homepage": "https://github.com/Yegor2305/kai-version-control#readme",
  "bugs": {
    "url": "https://github.com/Yegor2305/kai-version-control/issues"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/Yegor2305/kai-version-control.git"
  },
  "license": "ISC",
  "author": "Yehor",
  "type": "commonjs",
  "main": "index.ts",
  "bin": {
    "kai": "./dist/index.js"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "tsc",
    "dev": "tsc -w"
  },
  "devDependencies": {
    "@types/node": "^25.6.1",
    "prettier": "^3.8.3",
    "ts-node": "^10.9.2",
    "typescript": "^6.0.3"
  },
  "dependencies": {
    "commander": "^14.0.3",
    "diff": "^9.0.0",
    "diff2html": "^3.4.56",
    "open": "^10.2.0"
  }
}
```

ДОДАТОК В
Слайди презентації

Національний університет «Запорізька політехніка»
Кафедра програмних засобів

Дослідження та програмна реалізація алгоритмів контролю версій документів

Виконав:
студент групи КНТ-112

Єгор АРТАМОШИН

Керівник:
к.т.н доцент

Євгеній ГОФМАН

2026

Рисунок В.1 – Слайд №1

Мета та завдання дослідження

○ **Об'єкт дослідження**

Процес інженерії програмного забезпечення для контролю версій електронних документів, управління історією версій та відновлення попередніх станів даних

○ **Предмет дослідження**

Алгоритми, методи та програмні інструменти для реалізації систем контролю версій документів, механізми збереження об'єктів, визначення відмінностей між версіями файлів та відновлення станів документів

○ **Мета роботи**

Підвищення ефективності систем контролю версій, зниження мінімального рівня кваліфікаційних вимог до користувача та мінімізація кількості помилок при роботі GIT-подібних алгоритмів

2

Рисунок В.2 – Слайд №2

Аналіз існуючих рішень

Порівняння систем Git, SVN, Perforce та системи Kai

Критерій	Git	SVN	Perforce	Kai
Тип системи	Розподілена	Централізована	Централізована	Локальна
Спосіб збереження	Знімки	Зміни	Зміни	Знімки
Робота офлайн	Повна	Обмежена	Обмежена	Повна
Робота з бінарними файлами	Обмежена	Середня	Висока	Обмежена
Основна сфера використання	Розробка ПЗ	Корпоративні системи	Великі проекти, ігрова індустрія	Навчання
Зручність модифікації	Низька	Низька	Низька	Висока
Прозорість структури репозиторію	Низька	Низька	Низька	Висока
Освітня придатність	Середня	Середня	Низька	Висока

3

Рисунок В.3 – Слайд №3

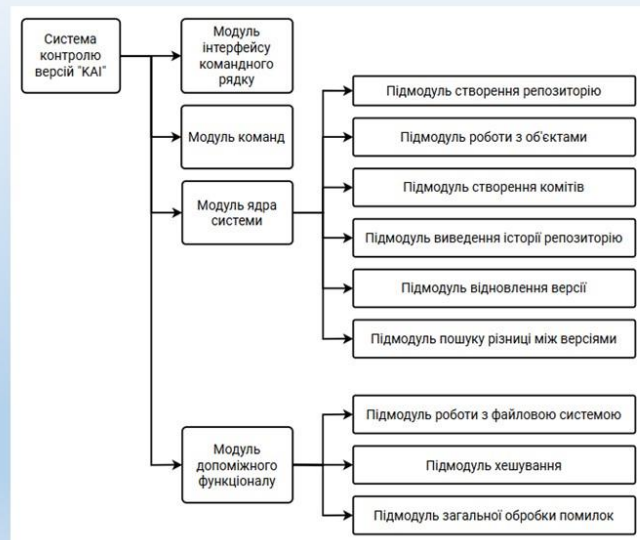
Порівняння мов програмування

Критерій	TypeScript	JavaScript	C++
Типізація	Статична	Динамічна	Статична
Простота розробки	Висока	Висока	Низька
Швидкодія	Середня	Середня	Висока
Робота з файловою системою	Проста	Проста	Ускладнена
Екосистема	Розвинена	Розвинена	Розвинена
Поріг входження	Середній	Низький	Високий

4

Рисунок В.4 – Слайд №4

Структурна схема програми

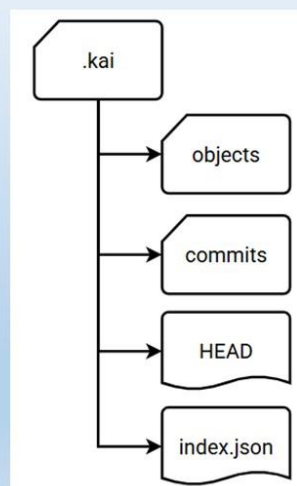


5

Рисунок В.5 – Слайд №5

Створення репозиторію

Структура файлів в службовому каталозі



6

Рисунок В.6 – Слайд №6

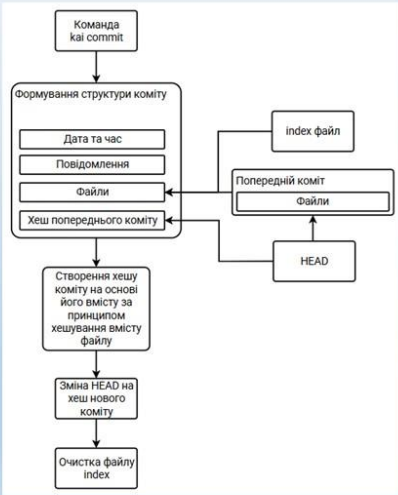
Алгоритм додавання файлів



7

Рисунок В.7 – Слайд №7

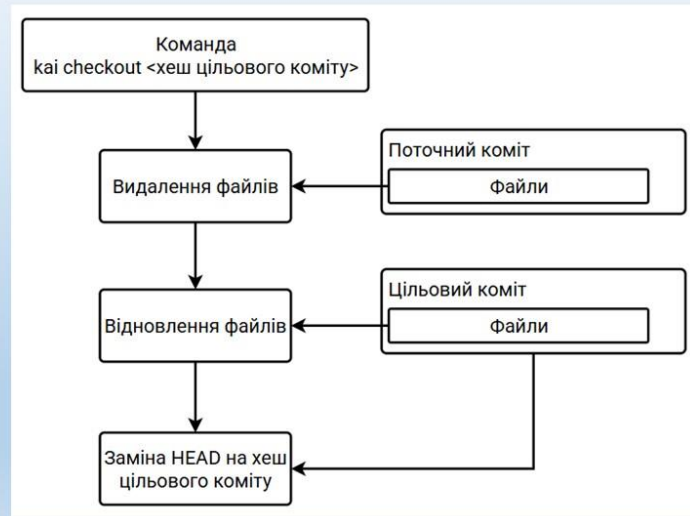
Алгоритм створення комітів



8

Рисунок В.8 – Слайд №8

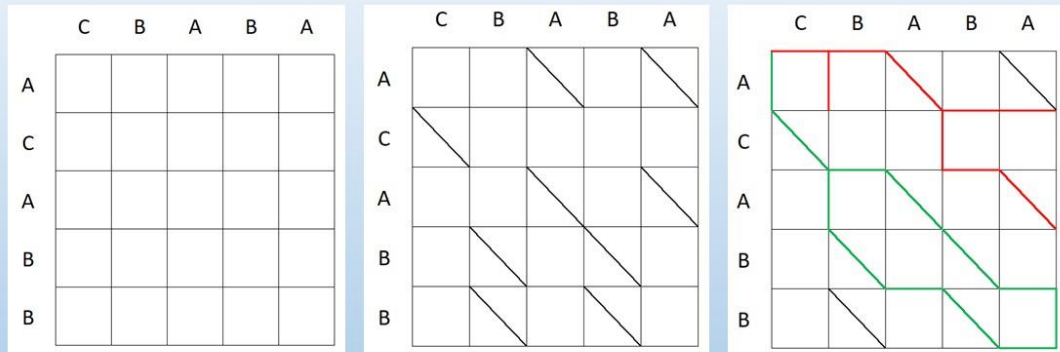
Алгоритм відновлення версій



9

Рисунок В.9 – Слайд №9

Алгоритм порівняння версій



10

Рисунок В.10 – Слайд №10

Схема виконання порівняння в системі Kai



11

Рисунок В.11 – Слайд №11

Висновки та перспективи

Результати

- Досліджено принципи та алгоритми систем контролю версій
- Розроблено власну систему контролю версій
- Спроектовано зручну модульну архітектуру програми

Перспективи розвитку

- Реактивне відстеження змін в файлах
- Система гілок
- Інтеграція з хмарними сервісами

12

Рисунок В.12 – Слайд №12