МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ
Циклова комісія комп’ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему

ІНЖЕНЕРІЯ СИСТЕМ КОНТРОЛЮ ВЕРСІЙ

Виконав: студент групи 2П-22
Спеціальності
121 Інженерія програмного забезпечення
Владислав БАЗЮК
Керівник:
Станіслав МАРЧЕНКО

Черкаси 2026

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ФАХОВИЙ БІЗНЕС-КОЛЕДЖ

Циклова комісія комп'ютерної інженерії та інформаційних технологій

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувачка циклової комісії КІ та ІТ

_____ Наталя ФАЛЬЧЕНКО
(підпис)

«_____» _____2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Базюку Владиславу Руслановичу

1. Тема кваліфікаційної роботи Інженерія систем контролю версій

Керівник роботи Марченко Станіслав Віталійович, викладач першої категорії затверджені наказом закладу фахової передвищої освіти від «06» жовтня 2025 року № 84/1-у.

2. Строк подання студентом кваліфікаційної роботи 02.06.2026

3. Вихідні дані до кваліфікаційної роботи:_ мова програмування Python, стандартні бібліотеки Python hashlib, zlib, pathlib, os, shutil, json, datetime, argparse, tempfile, subprocess, unittest, система керування версіями Git, вебсервіс GitHub для розміщення репозиторію з первинним кодом, мова опису діаграм UML, інструмент diagrams.net для побудови структурних і поведінкових діаграм, текстовий формат JSON для збереження метаданих репозиторію, командний інтерфейс операційної системи для взаємодії з програмним прототипом.

4. Зміст кваліфікаційної роботи: аналіз предметної області (роль систем контролю версій в інженерії програмного забезпечення; централізовані та розподілені системи контролю версій; аналіз наявних систем контролю версій; постановка задачі на розроблення); проєктування та реалізація програмного забезпечення (аналіз вимог до програмного забезпечення; архітектура програмного прототипу системи контролю версій; алгоритми основних операцій системи контролю версій; тестування програмного продукту (вибір методів тестування; формування тест-плану; організація комп'ютерних експериментів).

5. Дата видачі завдання 15.09.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	14.10.2025	
2	Розділ 1. Аналіз предметної області	09.12.2025	
3	Розділ 2. Проєктування та реалізація програмного забезпечення	10.03.2026	
4	Розділ 3. Тестування програмного продукту	28.04.2026	
5	Висновки	12.05.2026	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	26.05.2026	
7	Перевірка кваліфікаційної роботи на наявність ознак плагіату (за 10 днів до захисту)	02.06.2026	
8	Подання кваліфікаційної роботи на затвердження завідувачці циклової комісії (за 7 днів до захисту)	10.06.2026	

Студент _____ Владислав БАЗЮК
(підпис)

Керівник роботи _____ Станіслав МАРЧЕНКО
(підпис)

АНОТАЦІЯ

Кваліфікаційна робота присвячена аналізу інженерних принципів функціонування систем контролю версій та розробленню програмного прототипу локальної розподіленої системи контролю версій. У роботі проаналізовано централізовану та розподілену моделі систем контролю версій, розглянуто характерні особливості Git, Subversion і Mercurial, а також визначено базові механізми, на яких ґрунтується робота таких систем: репозиторій, робоча копія, індекс змін, коміт, гілка, покажчик HEAD, хешування файлів, об’єктне сховище, злиття змін і віддалена синхронізація.

На основі проведеного аналізу сформульовано функціональні та нефункціональні вимоги до програмного прототипу. Розроблено архітектуру локальної системи контролю версій, визначено структуру службового каталогу .vcs, формат збереження метаданих, принципи організації об’єктного сховища та модель локальної віддаленої синхронізації. Реалізовано командний інтерфейс, що підтримує ініціалізацію репозиторію, перевірку стану файлів, додавання змін до індексу, створення комітів, перегляд історії, роботу з гілками, checkout, merge, обробку конфліктів, clone, push і pull між локальними репозиторіями.

Програмний прототип реалізовано мовою Python із використанням стандартних бібліотек для роботи з файловою системою, SHA-1 хешуванням, zlib-стисненням, JSON-серіалізацією, обробкою аргументів командного рядка та автоматизованим тестуванням. Для перевірки працездатності системи сформовано 12 груп тестових сценаріїв TC001–TC012 і виконано 27 автоматизованих перевірок засобами unittest. Результати роботи можуть бути використані як навчальний інструмент для демонстрації внутрішніх механізмів систем контролю версій та як основа для подальшого розширення, зокрема підтримки мережевих протоколів HTTP/SSH, авторизації користувачів, графічного інтерфейсу та оптимізації роботи з великими репозиторіями.

Ключові слова: СИСТЕМА КОНТРОЛЮ ВЕРСІЙ, РЕПОЗИТОРІЙ, ХЕШУВАННЯ, PYTHON, ІСТОРІЯ ЗМІН, КОМАНДНИЙ ІНТЕРФЕЙС.

ABSTRACT

The qualification thesis is devoted to the analysis of the engineering principles underlying version control systems and to the development of a software prototype of a local distributed version control system. The thesis analyzes centralized and distributed models of version control systems, examines the characteristic features of Git, Subversion, and Mercurial, and identifies the basic mechanisms on which such systems are based: repository, working copy, change index, commit, branch, HEAD pointer, file hashing, object storage, change merging, and remote synchronization.

Based on the conducted analysis, functional and non-functional requirements for the software prototype were formulated. The architecture of the local version control system was developed, the structure of the .vcs service directory was defined, and the format for storing metadata, the principles of organizing the object storage, and the model of local remote synchronization were specified. A command-line interface was implemented, supporting repository initialization, file status checking, adding changes to the index, creating commits, viewing history, working with branches, checkout, merge, conflict handling, clone, push, and pull operations between local repositories.

The software prototype was implemented in Python using standard libraries for file system operations, SHA-1 hashing, zlib compression, JSON serialization, command-line argument processing, and automated testing. To verify the functionality of the system, 12 groups of test scenarios, TC001–TC012, were developed, and 27 automated checks were performed using the unittest framework. The results of the work can be used as an educational tool for demonstrating the internal mechanisms of version control systems and as a basis for further extension, in particular for supporting HTTP/SSH network protocols, user authorization, a graphical user interface, and optimization for working with large repositories.

Keywords: VERSION CONTROL SYSTEM, REPOSITORY, HASHING, PYTHON, COMMAND-LINE INTERFACE.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ

СКВ	Система контролю версій, Version Control System (VCS)
CVCS	Central VCS, централізована система контролю версій
DVCS	Distributed VCS, розподілена система контролю версій
SCM	Source Code Management
SVN	Subversion
Mercurial	Розподілена система контролю версій
Merge	Операція злиття змін
Checkout	Операція переходу до визначеного стану репозиторію
HEAD	Посилання на поточний стан або останній коміт у репозиторії
SHA	Secure Hash Algorithm
SRS	Software Requirements Specification, специфікація вимог
FR	Функціональна вимога
NFR	Нефункціональна вимога
CI/CD	Continuous Integration / Continuous Delivery

ВСТУП

У ході розроблення програмного забезпечення постійно виникає потреба у фіксації змін, збереженні історії роботи над проєктом, відновленні попередніх станів програмного коду та організації паралельної роботи кількох розробників. Без спеціалізованих засобів керування змінами програмний проєкт швидко втрачає керованість: складно встановити, ким і коли було внесено зміну, яка версія файлу є актуальною, які правки спричинили помилку та яким чином повернути систему до стабільного стану. Саме ці завдання розв’язують системи контролю версій, які є одним із базових інструментів сучасної інженерії програмного забезпечення.

Система контролю версій забезпечує формалізоване збереження станів програмного проєкту, фіксацію змін у вигляді комітів, ведення журналу історії, порівняння різних версій файлів, підтримку гілкування та злиття змін. У сучасній практиці розроблення найбільш поширеними є такі системи, як Git, Subversion і Mercurial [1; 2; 3]. Вони реалізують різні підходи до організації репозиторіїв, збереження історії та синхронізації змін між учасниками розроблення. Зокрема, централізовані системи контролю версій ґрунтуються на використанні спільного центрального репозиторію, тоді як розподілені системи надають кожному розробнику повну локальну копію історії проєкту.

Попри широке використання готових систем контролю версій, їхні внутрішні механізми часто залишаються недостатньо зрозумілими для користувачів. У практичній роботі розробник зазвичай використовує набір команд, не аналізуючи принципи, за якими система визначає зміни у файлах, формує знімки стану проєкту, обчислює ідентифікатори об’єктів, зберігає метадані комітів і відновлює попередні версії. Це створює розрив між прикладним використанням інструменту та розумінням його інженерної побудови. Для фахівця з інженерії програмного забезпечення важливим є не лише вміння застосовувати готову систему контролю версій, а й розуміння архітектурних, алгоритмічних і файлових механізмів, які забезпечують її функціонування [4; 5; 6].

Актуальність теми зумовлена тим, що системи контролю версій є невіддільною частиною життєвого циклу програмного забезпечення. Вони використовуються на етапах розроблення, тестування, інтеграції, супроводу та розгортання програмних продуктів. Крім того, системи контролю версій безпосередньо пов’язані з практиками командної розробки, CI/CD, управлінням релізами та забезпеченням відтворюваності програмного продукту. Тому дослідження принципів їх функціонування та реалізація власного програмного прототипу мають як навчальне, так і прикладне значення.

Ступінь дослідження теми визначається наявністю значної кількості технічної документації, навчальних матеріалів і практичних реалізацій систем контролю версій. Найбільш дослідженими є повнофункціональні промислові рішення, зокрема Git, Subversion і Mercurial. Водночас для навчальних та інженерно-прикладних цілей доцільним є створення спрощеного програмного прототипу, який дозволяє ізолювати ключові механізми систем контролю версій і показати їх роботу без надлишкової складності промислових інструментів. Такий підхід дає змогу дослідити базові принципи збереження історії змін, хешування файлів, формування комітів і відновлення попередніх станів проєкту на рівні програмної реалізації.

Об’єктом дослідження є процеси збереження, відстеження та керування змінами у програмних проєктах.

Предметом дослідження виступають архітектурні принципи, моделі даних, алгоритми та базові механізми функціонування систем контролю версій.

Метою кваліфікаційної роботи є аналіз принципів функціонування систем контролю версій та розроблення програмного прототипу локальної системи контролю версій з базовою функціональністю. Для досягнення поставленої мети необхідно виконати такі завдання:

– проаналізувати основні поняття та моделі функціонування систем контролю версій, базові механізми роботи систем контролю версій, порівняти централізований і розподілений підходи до організації систем контролю версій;

– сформулювати функціональні та нефункціональні вимоги до програмного прототипу [7];

– спроєктувати архітектуру локальної системи контролю версій, визначити модель збереження службових даних, об’єктів і метаданих репозиторію [8];

– реалізувати основні команди системи контролю версій;

– провести функціональне тестування реалізованого програмного прототипу, проаналізувати результати реалізації, обмеження системи та напрями її подальшого розвитку.

Практична цінність кваліфікаційної роботи полягає у створенні програмного прототипу локальної системи контролю версій, який може бути використаний як навчальний інструмент для демонстрації внутрішніх механізмів VCS. Розроблене рішення дозволятиме простежити повний цикл роботи базової системи контролю версій: від створення службового каталогу репозиторію до фіксації змін і відновлення попередніх станів файлів.

Очікуваним результатом роботи є працездатний програмний прототип, що підтримує ініціалізацію репозиторію, перевірку стану файлів, додавання змін до індексу, створення комітів, перегляд історії змін, гілкування, відновлення попередніх версій, злиття гілок, обробку merge-конфліктів, клонування та локальну синхронізацію між репозиторіями. Отримані результати можуть бути використані як основа для подальшого розширення системи мережевими протоколами, графічним інтерфейсом і засобами керування доступом.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Роль систем контролю версій в інженерії програмного забезпечення

Система контролю версій – це програмний засіб, призначений для фіксації, збереження та відстеження змін у файлах програмного проєкту протягом його життєвого циклу. У загальному вигляді контроль версій розглядається як механізм запису змін до одного або кількох файлів у часі з можливістю подальшого повернення до конкретного попереднього стану [1]. У контексті інженерії програмного забезпечення такі системи застосовуються насамперед для керування змінами у первинному коді, конфігураційних файлах, документації, тестових наборах та інших артефактах розроблення.

Необхідність використання систем контролю версій зумовлена природою процесу створення програмного забезпечення. Програмний продукт постійно змінюється: додаються нові функції, виправляються дефекти, змінюється структура коду, оновлюються залежності, уточнюються вимоги та коригується технічна документація. Без формалізованого механізму обліку цих змін складно встановити, яка саме версія файлу є актуальною, які зміни були внесені в конкретний момент часу, хто їх виконав і з якою метою. У разі виникнення помилки відсутність історії змін ускладнює пошук причини дефекту та повернення системи до стабільного стану.

У найпростішому випадку контроль версій може використовуватись одним розробником для збереження послідовних станів власного проєкту. Однак основне значення систем контролю версій проявляється у командному розробленні, де над одним програмним продуктом одночасно працюють декілька учасників. У такому середовищі система контролю версій забезпечує узгоджене внесення змін, ізоляцію робочих копій, інтеграцію результатів роботи різних розробників і збереження повної історії розвитку проєкту. Завдяки цьому знижується ризик втрати даних, випадкового перезапису чужих змін або неконтрольованого розходження версій програмного коду.

З позиції життєвого циклу програмного забезпечення система контролю версій не обмежується етапом безпосереднього написання коду. Вона використовується під час аналізу вимог, проєктування, реалізації, тестування, інтеграції, випуску релізів і супроводу програмного продукту. У репозиторії можуть зберігатися не лише файли реалізації, а й специфікації вимог, UML-діаграми, сценарії тестування, інструкції з розгортання, конфігурації середовищ і супровідна технічна документація. Отже, система контролю версій виконує роль централізованого або розподіленого сховища інженерних артефактів проєкту.

Основними об’єктами взаємодії в системі контролю версій є репозиторій, робоча копія, коміт та історія змін. Репозиторій зберігає службові дані системи, метадані та зафіксовані стани проєкту. Робоча копія є поточним набором файлів, з якими безпосередньо працює розробник. Коміт фіксує певний стан проєкту або набір змін, що має смислову завершеність. Історія змін утворюється як послідовність комітів, пов’язаних між собою часовими та логічними відношеннями. У промислових системах, таких як Git, коміт фактично відображає зафіксований знімок стану проєкту, а не лише текстовий опис різниці між файлами [1; 5].

Типовий процес роботи з системою контролю версій передбачає створення або отримання репозиторію, внесення змін у робочу копію, перевірку стану файлів, підготовку змін до фіксації, створення коміту та, за потреби, синхронізацію з віддаленим сховищем. Для розподілених систем додатково характерні операції отримання змін з інших репозиторіїв, передавання власних змін і злиття незалежних гілок розроблення. У централізованих системах ключова взаємодія відбувається з єдиним центральним репозиторієм, тоді як у розподілених системах кожен учасник має власну повну копію репозиторію та може виконувати більшість операцій локально [1-3].

Узагальнену схему переміщення змін між основними областями системи контролю версій подано на рис. 1.1. Вона відображає послідовність переходу

змін від робочої області до індексу змін, подальше створення коміту в локальному репозиторії та синхронізацію з віддаленим репозиторієм.

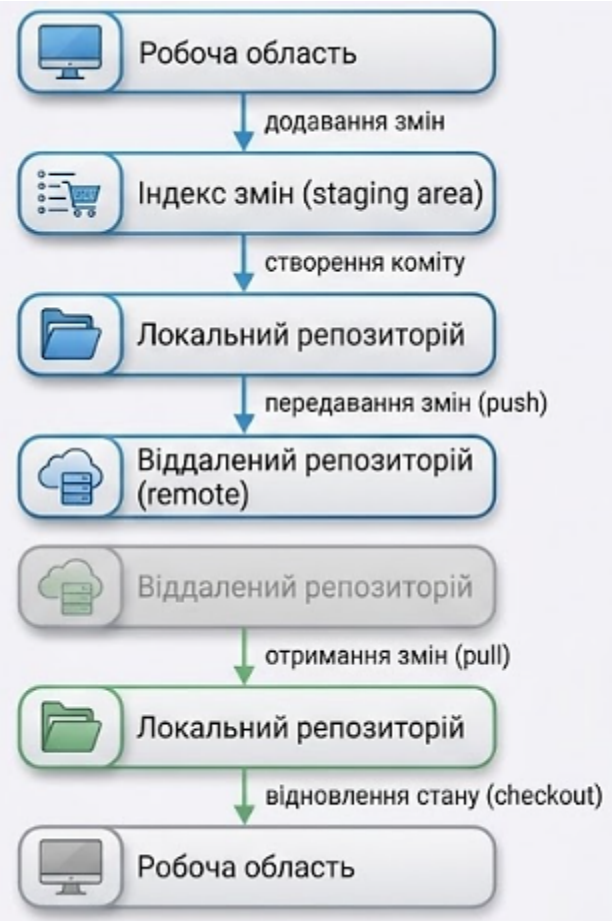

Рисунок 1.1 – Узагальнена схема переміщення змін у системі контролю версій

Як показано на рис. 1.1, робоча область містить поточні файли проєкту, з якими безпосередньо працює розробник. Після внесення змін окремі файли додаються до індексу змін, що виконує функцію проміжної області підготовки коміту. Створення коміту фіксує підготовлений стан у локальному репозиторії, де він зберігається як елемент історії проєкту. За наявності віддаленого репозиторію локальні зміни можуть бути передані до спільного сховища, а зміни інших учасників можуть бути отримані для оновлення локального стану проєкту.

Опис такого циклу можна подати через послідовність основних дій. Спочатку розробник створює локальний репозиторій або отримує копію вже наявного проєкту. Після цього він вносить зміни до файлів робочої копії: додає

новий код, редагує наявні модулі, змінює конфігураційні файли або документацію. Наступним кроком система визначає стан файлів: нові, змінені, видалені або незмінені. На основі цієї інформації розробник приймає рішення, які зміни мають бути зафіксовані в історії проєкту. Після створення коміту система зберігає відповідні дані у репозиторії та оновлює покажчик поточного стану.

У випадку командної розробки після локальної фіксації змін може виконуватися синхронізація з віддаленим репозиторієм. Якщо інші учасники проєкту також внесли зміни, система контролю версій повинна забезпечити їх інтеграцію. За відсутності суперечностей злиття може бути виконане автоматично. Якщо ж різні учасники змінили один і той самий фрагмент файлу несумісним способом, виникає конфлікт, який потребує ручного вирішення. Таким чином, система контролю версій не лише зберігає історію змін, а й підтримує механізми координації паралельної роботи.

Важливою властивістю систем контролю версій є забезпечення відтворюваності станів програмного продукту. Кожен зафіксований стан може бути використаний для аналізу, тестування, порівняння або відновлення. Це має безпосереднє значення для супроводу програмного забезпечення: у разі виявлення дефекту можна встановити, у якій версії він з’явився, які зміни йому передували та який коміт відповідає стабільному стану системи. Такий підхід спрощує технічне обслуговування, аудит змін і підготовку релізів.

З інженерної точки зору система контролю версій є не лише допоміжним інструментом, а окремим класом програмних систем, що має власну архітектуру, модель даних і алгоритми обробки змін. Внутрішня побудова такої системи охоплює механізми хешування вмісту файлів, збереження об’єктів, формування метаданих комітів, побудови історії, порівняння версій, відновлення попередніх станів і підтримки посилань на поточну версію. Саме ці механізми становлять основу для подальшого проєктування власного програмного прототипу локальної системи контролю версій.

Отже, системи контролю версій відіграють ключову роль у процесі інженерії програмного забезпечення, оскільки забезпечують керованість змін, збереження історії проєкту, підтримку командної розробки та відновлення попередніх станів програмного продукту. Для цієї кваліфікаційної роботи особливе значення має не лише аналіз готових рішень, а й дослідження базових принципів їх побудови, що дозволяє перейти від прикладного використання VCS до розуміння її внутрішньої інженерної структури.

Репозиторій є основним сховищем даних системи контролю версій. У ньому зберігаються зафіксовані стани проєкту, службові файли, метадані комітів, посилання між версіями та інша інформація, необхідна для відновлення історії змін. У централізованих системах репозиторій зазвичай розміщується на сервері й виступає єдиним джерелом актуального стану проєкту. У розподілених системах кожен учасник розроблення має власний локальний репозиторій, що містить повну або достатню для автономної роботи історію проєкту.

Робоча копія – це поточний набір файлів проєкту, з якими безпосередньо працює розробник. Саме в ній виконуються редагування первинного коду, створення нових файлів, видалення застарілих компонентів, зміна конфігураційних параметрів і оновлення документації. Робоча копія відрізняється від репозиторію тим, що вона відображає поточний стан файлової системи, який може містити ще не зафіксовані зміни.

У багатьох системах контролю версій між робочою копією та репозиторієм існує проміжний рівень підготовки змін. У Git цей рівень відомий як індекс змін або staging area. Його призначення полягає у вибірковому формуванні майбутнього коміту. Це означає, що розробник може змінити багато файлів, але до наступної фіксації додати лише частину з них. Такий підхід підвищує керованість історії, оскільки дозволяє створювати логічно завершені коміти, а не фіксувати всі зміни одночасно [5].

Коміт є зафіксованим станом проєкту або набором змін, що має самостійне змістове значення. З технічної точки зору коміт містить службові метадані: унікальний ідентифікатор, дату створення, автора, повідомлення, посилання на

попередній коміт і опис зафіксованих файлів. У промислових системах контролю версій ідентифікатор коміту часто формується на основі хешування його вмісту та метаданих. Це дає змогу однозначно ідентифікувати стан проєкту та виявляти зміну вмісту об’єктів [5; 6].

Історія змін формується як послідовність пов’язаних між собою комітів. Кожен новий коміт посилається на попередній, утворюючи ланцюг станів проєкту. Завдяки цьому система контролю версій може відтворити еволюцію програмного продукту, визначити момент появи певної зміни, порівняти різні версії файлів і повернути робочу копію до попереднього стану. Для розробника історія змін виконує не лише технічну, а й аналітичну функцію: вона дозволяє простежити логіку розвитку проєкту.

Гілка є окремою лінією розроблення, що дозволяє ізолювати зміни від основної історії проєкту. Гілкування використовується для розроблення нових функцій, виправлення дефектів, підготовки релізів або проведення експериментальних змін без порушення стабільного стану основної версії. Після завершення роботи зміни з окремої гілки можуть бути інтегровані до іншої гілки за допомогою операції злиття. Якщо зміни не суперечать одна одній, злиття виконується автоматично. Якщо ж різні гілки містять несумісні зміни в одному фрагменті файлу, виникає конфлікт, який потребує ручного вирішення [5].

Поточний стан репозиторію зазвичай визначається спеціальним покажчиком. У Git таким покажчиком є HEAD. Він вказує на поточний коміт або поточну гілку, з якою працює розробник. Завдяки цьому система може визначити, відносно якого стану потрібно порівнювати робочу копію, куди буде додано наступний коміт і який стан слід вважати активним. У власному програмному прототипі аналогічний механізм може бути реалізований через службовий файл, що зберігає ідентифікатор останнього коміту [5].

Віддалений репозиторій використовується для обміну змінами між учасниками розроблення. У розподілених системах він не обов’язково є єдиним джерелом істини, однак виконує роль спільної точки синхронізації. Розробник може передавати локальні зміни до віддаленого репозиторію, отримувати зміни

інших учасників і об’єднувати їх зі своїм локальним станом. У централізованих системах віддалений сервер є обов’язковим елементом роботи, тоді як у розподілених системах значна частина операцій може виконуватися локально.

Узагальнену модель взаємозв’язку базових об’єктів системи контролю версій подано на рис. 1.2. Схема відображає послідовність переходу змін від робочої копії до індексу змін, подальше формування коміту, включення його до історії змін і збереження відповідних даних у локальному та віддаленому репозиторіях.

Рисунок 1.2 – Концептуальна модель основних об’єктів системи контролю версій

Як показано на рис. 1.2, робоча копія є поточним станом файлів проєкту, з якими безпосередньо працює розробник. Індекс змін виконує функцію проміжної області, у якій визначається набір файлів для наступної фіксації. Коміт зберігає зафіксований стан змін і містить службові метадані: ідентифікатор, автора, дату, повідомлення та посилання на попередній коміт.

Послідовність комітів формує історію змін, а покажчик HEAD визначає поточний активний стан репозиторію.

У межах цієї моделі робоча копія відображає поточний стан файлів проєкту, індекс змін визначає набір файлів, підготовлених до фіксації, коміт зберігає зафіксований стан або зміну, а репозиторій забезпечує довготривале збереження історії. Гілки дозволяють підтримувати кілька незалежних напрямів розроблення, а покажчик HEAD визначає активний стан, з яким виконується подальша робота. Віддалений репозиторій забезпечує синхронізацію між різними локальними копіями проєкту.

Для подальшого проєктування власного програмного прототипу доцільно зосередитися на сутностях, які безпосередньо забезпечують локальну модель DVCS: робочій копії, службовому каталозі .vcs, індексі змін, об’єктному сховищі, комітах, гілках, покажчику HEAD, операціях злиття та локальній remote-синхронізації. Такий обсяг функціональності дозволяє продемонструвати ключові інженерні механізми систем контролю версій без відтворення всієї складності промислових інструментів [4; 5; 6].

1.2 Централізовані та розподілені системи контролю версій

Для аналізу систем контролю версій доцільно розділити їх на два основні класи: централізовані та розподілені. Такий поділ ґрунтується на способі організації репозиторію, збереження історії змін і взаємодії між учасниками розроблення. До централізованих систем належать, зокрема, Subversion, а до розподілених – Git і Mercurial. Усі ці системи призначені для керування змінами у програмних проєктах, однак відрізняються архітектурою, моделлю роботи з історією та способом синхронізації змін.

Централізована система контролю версій передбачає наявність єдиного центрального репозиторію, з яким взаємодіють усі учасники розроблення. Кожен розробник має локальну робочу копію проєкту, але повна історія змін зберігається переважно на центральному сервері. У такій моделі основні операції, пов’язані з отриманням актуального стану проєкту та фіксацією змін,

виконуються через звернення до центрального репозиторію. В офіційній документації Subversion робоча копія визначається як локальний приватний простір користувача, який використовується для взаємодії з центральним репозиторієм [2].

Перевагою централізованої моделі є відносна простота організації доступу та адміністрування. Оскільки основний репозиторій розміщено в одному місці, легше контролювати права користувачів, резервне копіювання, політики доступу та структуру гілок. Такий підхід є зручним для організацій, де необхідно забезпечити жорсткий контроль над джерелом змін і централізоване управління проєктом. Водночас ця модель має істотне обмеження: залежність від доступності центрального сервера. Якщо сервер недоступний, виконання частини операцій, зокрема повноцінної фіксації змін у спільній історії, ускладнюється або стає неможливим.

Розподілена система контролю версій використовує іншу архітектурну модель. У такій системі кожен учасник має локальний репозиторій, який містить історію змін і дозволяє виконувати основні операції без постійного підключення до віддаленого сервера. Розробник може створювати коміти, переглядати історію, створювати гілки та порівнювати версії локально. Віддалений репозиторій у цьому випадку виконує роль точки синхронізації між локальними репозиторіями різних учасників.

Git є найпоширенішим прикладом розподіленої системи контролю версій. Його модель роботи ґрунтується на використанні робочого дерева, індексу змін і локального репозиторію. Згідно з документацією Pro Git, типовий цикл роботи включає зміну файлів у робочому дереві, вибіркове додавання змін до staging area та створення коміту, який зберігає підготовлений знімок стану в Git-директорії [1; 4].

Mercurial також належить до розподілених систем керування вихідним кодом. Офіційний сайт Mercurial визначає його як безкоштовний розподілений інструмент керування вихідним кодом, призначений для роботи з проєктами різного розміру. На відміну від Git, Mercurial історично орієнтувався на простішу

модель використання та більш уніфікований командний інтерфейс. Обидві системи підтримують локальну історію, автономне створення комітів, гілкування, злиття та синхронізацію з віддаленими репозиторіями [3].

Узагальнене порівняння централізованої та розподіленої моделей контролю версій подано на рис. 1.3. Схема відображає відмінність між підходом, у якому всі учасники працюють із єдиним центральним репозиторієм, та підходом, у якому кожен розробник має власний локальний репозиторій і синхронізує його з віддаленим сховищем.

Рисунок 1.3 – Порівняння централізованої та розподіленої моделей контролю версій

Як показано на рис. 1.3, у централізованій моделі всі робочі копії взаємодіють з одним центральним репозиторієм. Такий підхід спрощує адміністрування доступу та контроль єдиного джерела змін, однак створює залежність від доступності центрального сервера. У розподіленій моделі кожен

учасник має власний локальний репозиторій, що дає змогу виконувати основні операції з історією змін автономно. Віддалений репозиторій у такій архітектурі використовується переважно як точка синхронізації між локальними репозиторіями учасників розроблення.

Ключова відмінність між централізованою та розподіленою моделлю полягає у розміщенні історії змін. У централізованій системі повна історія зосереджена на сервері, а локальна робоча копія виконує переважно функцію середовища редагування файлів. У розподіленій системі локальний репозиторій є повноцінним сховищем історії, тому користувач може виконувати значну частину операцій без мережевого доступу. Це підвищує автономність роботи, але водночас ускладнює процес синхронізації змін між кількома незалежними копіями репозиторію. Для узагальнення відмінностей між централізованими та розподіленими системами контролю версій доцільно подати порівняльну таблицю 1.1 [1; 2; 3].

Таблиця 1.1 – Порівняння централізованих і розподілених систем контролю версій

Критерій	Централізована система	Розподілена система
Організація репозиторію	Один центральний репозиторій	Кожен учасник має локальний репозиторій
Збереження історії	Основна історія зберігається на сервері	Історія зберігається локально у кожного учасника
Залежність від мережі	Висока для ключових операцій	Нижча, більшість операцій виконується локально
Типові представники	Subversion	Git, Mercurial
Командна робота	Через центральний сервер	Через синхронізацію локальних репозиторіїв
Адміністрування доступу	Простіше через єдиний сервер	Гнучкіше, але потребує узгодження правил синхронізації
Придатність до автономної роботи	Обмежена	Висока
Складність моделі	Нижча для початкового використання	Вища через наявність локальної та віддаленої історії

Як видно з таблиці 1.1, централізована модель є простішою з погляду адміністрування, але менш гнучкою в умовах автономної або розподіленої командної роботи. Розподілена модель забезпечує більшу незалежність розробників, локальну роботу з історією та зручні механізми паралельного розроблення, однак потребує чіткого розуміння операцій синхронізації, гілкування та злиття.

У межах цієї кваліфікаційної роботи розроблюваний програмний прототип доцільно будувати як локальну навчальну модель розподіленої системи контролю версій. Такий підхід дозволяє зосередитися на базових механізмах VCS: створенні службового каталогу, виявленні змін у робочій копії, хешуванні вмісту файлів, формуванні комітів, підтримці гілок, виконанні злиття, обробці конфліктів і спрощеній локальній синхронізації між репозиторіями без реалізації мережевих протоколів HTTP або SSH.

1.3 Аналіз наявних систем контролю версій

Для аналізу наявних систем контролю версій було обрано Git, Apache Subversion та Mercurial. Такий вибір зумовлений тим, що ці системи представляють основні підходи до організації контролю версій: Git і Mercurial належать до розподілених систем, тоді як Subversion є централізованою системою контролю версій. Порівняння цих рішень дозволяє визначити спільні механізми, характерні для систем цього класу, а також виявити функціональні обмеження, які необхідно врахувати під час проєктування власного програмного прототипу.

Git є розподіленою системою контролю версій, орієнтованою на швидку локальну роботу з історією змін. Основною особливістю Git є те, що кожен розробник має локальний репозиторій, у якому зберігається історія проєкту. Це дає змогу виконувати більшість операцій без постійного підключення до мережі: створювати коміти, переглядати історію, створювати гілки, виконувати злиття та порівнювати різні стани файлів. Віддалений репозиторій у Git використовується

переважно як засіб синхронізації між локальними копіями учасників розроблення [1; 4].

У внутрішній моделі Git важливе місце займає об’єктне сховище. Файли, дерева каталогів і коміти зберігаються як окремі об’єкти, ідентифіковані за допомогою хеш-значень. Завдяки цьому Git може ефективно визначати зміни у вмісті файлів, зберігати зв’язки між комітами та забезпечувати цілісність історії. Також Git використовує проміжну область підготовки змін — індекс або staging area. Цей механізм дозволяє розробнику самостійно визначити, які саме зміни будуть включені до наступного коміту [5; 6].

Сильною стороною Git є гнучка підтримка гілкування та злиття. Гілки у Git є легкими покажчиками на коміти, тому їх створення та перемикання між ними не потребує значних ресурсів. Це робить Git зручним інструментом для командної розробки, роботи з окремими функціями, виправлення дефектів і підготовки релізів. Водночас така гнучкість збільшує складність використання системи для початківців, оскільки вимагає розуміння локальної історії, віддалених гілок, індексу змін, операцій rebase, merge, fetch, pull і push [4; 5].

Apache Subversion є централізованою системою контролю версій. Її модель передбачає наявність єдиного центрального репозиторію, з яким взаємодіють усі робочі копії користувачів. На відміну від Git, локальна робоча копія Subversion не є повноцінним репозиторієм з повною історією проєкту. Вона містить файли, з якими працює користувач, і службові дані, необхідні для взаємодії з центральним сервером [2].

Перевагою Subversion є простіша модель роботи для організацій, у яких потрібне централізоване керування доступом і контроль єдиного джерела змін. Оскільки всі основні операції пов’язані з центральним репозиторієм, адміністрування прав користувачів, структура каталогів і політика доступу можуть бути організовані більш передбачувано. Subversion також підтримує версіювання каталогів, перейменування, копіювання та переміщення файлів у межах репозиторію [2].

Основним обмеженням Subversion є залежність від центрального сервера. Для повноцінної фіксації змін користувач повинен мати доступ до репозиторію. У разі відсутності мережевого з’єднання можливості роботи з історією суттєво обмежуються. Крім того, централізована модель менш гнучка для розподілених команд, оскільки всі зміни повинні проходити через єдину точку збереження.

Mercurial, як і Git, належить до розподілених систем контролю версій. Його архітектура також передбачає наявність локального репозиторію в кожного користувача, можливість автономного створення комітів, перегляду історії та синхронізації з віддаленими сховищами. Mercurial орієнтований на простішу та більш уніфіковану модель командного інтерфейсу. Завдяки цьому система є зрозумілішою для користувачів, які потребують розподіленої моделі контролю версій без надмірної складності команд і сценаріїв роботи [3].

Функціонально Mercurial підтримує базові механізми, необхідні для повноцінного контролю версій: створення комітів, перегляд історії, гілкування, злиття, обмін змінами між репозиторіями та відновлення попередніх станів. Водночас Mercurial має менше поширення у сучасних інструментальних екосистемах порівняно з Git. Це зменшує кількість інтеграцій, навчальних матеріалів і прикладів практичного використання [3].

Для порівняння розглянутих систем доцільно виділити набір критеріїв, які мають безпосереднє значення для розроблення власного програмного прототипу: тип архітектури, спосіб збереження історії, підтримка локальної роботи, механізм фіксації змін, підтримка гілкування, злиття, відновлення версій і складність використання.

Як видно з таблиці 1.2, усі розглянуті системи реалізують спільний набір базових функцій: збереження історії змін, фіксацію станів проєкту, роботу з попередніми версіями та підтримку командної взаємодії. Відмінності між ними полягають не в самому факті наявності контролю версій, а в архітектурній моделі, способі збереження історії та рівні автономності користувача.

Для цієї кваліфікаційної роботи найбільш важливими є не всі можливості промислових систем, а їхні базові механізми, які можна реалізувати у

навчальному програмному прототипі. До таких механізмів належать створення локального репозиторію, визначення змінених файлів, обчислення хешів, формування коміту, збереження метаданих, перегляд історії та відновлення стану файлів. Саме ці функції утворюють мінімальну основу системи контролю версій.

Таблиця 1.2 – Порівняння наявних систем контролю версій

Критерій	Git	Subversion	Mercurial
Тип системи	Розподілена	Централізована	Розподілена
Основне сховище історії	Локальний і віддалений репозиторії	Центральний репозиторій	Локальний і віддалений репозиторії
Автономна робота	Підтримується	Обмежена	Підтримується
Фіксація змін	Локальний коміт	Коміт до центрального репозиторію	Локальний коміт
Ідентифікація станів	Хеш-ідентифікатори об’єктів і комітів	Ревізії у центральному репозиторії	Ідентифікатори змін
Підтримка гілкування	Розвинена	Підтримується, але менш гнучка	Підтримується
Підтримка злиття	Розвинена	Підтримується	Підтримується
Віддалена синхронізація	Push, pull, fetch	Update, commit	Push, pull
Поріг входження	Вищий	Нижчий	Середній
Типове застосування	Командна та розподілена розробка	Централізована корпоративна розробка	Розподілена розробка з простішою моделлю команд

Разом із тим повноцінні промислові системи контролю версій містять значно ширший набір можливостей, ніж передбачено в межах навчального програмного прототипу. До таких можливостей належать підтримка мережевих протоколів синхронізації, автентифікація користувачів, керування правами доступу, оптимізація сховища, робота з великими репозиторіями та розширені алгоритми автоматичного злиття. У межах цієї кваліфікаційної роботи частина базових механізмів, зокрема робота з гілками, злиття, обробка конфліктів і

локальна синхронізація між репозиторіями, реалізується у спрощеному локальному вигляді. Це дозволяє продемонструвати принципи розподіленої системи контролю версій без створення повноцінної серверної інфраструктури [1; 4; 5].

Порівняння Git, Subversion і Mercurial показує, що для розроблення власної системи контролю версій доцільно використати локальну модель роботи, близьку до базових принципів розподілених VCS. Такий підхід дозволяє виконувати основні операції без серверної інфраструктури та зосередитися на внутрішній логіці репозиторію. Водночас прототип не повинен відтворювати всю складність Git або Mercurial, оскільки його основне призначення полягає у демонстрації фундаментальних механізмів контролю версій [1; 3; 4].

Отже, аналіз наявних систем контролю версій дозволяє визначити інженерну основу майбутнього програмного продукту. У власному прототипі необхідно реалізувати не повну промислову VCS, а обмежену локальну систему, яка відтворює ключові процеси: ініціалізацію репозиторію, виявлення змін, створення комітів, збереження історії та відновлення попередніх станів файлів. Саме ці функції є достатніми для демонстрації принципів роботи систем контролю версій і підготовки до подальшого проєктування архітектури програмного забезпечення.

1.4 Постановка задачі на розроблення

Проведений аналіз систем контролю версій показав, що промислові рішення цього класу мають розвинену функціональність, яка охоплює збереження історії змін, гілкування, злиття, синхронізацію з віддаленими репозиторіями, обробку конфліктів і підтримку командної роботи. У межах кваліфікаційної роботи реалізовано навчально-прикладний прототип, який відтворює ці механізми у спрощеному локальному вигляді, без повноцінної мережевої інфраструктури, автентифікації та оптимізацій промислового рівня.

Задача кваліфікаційної роботи полягає у розробленні програмного прототипу локальної розподіленої системи контролю версій, призначеної для

фіксації змін у файлах проєкту, збереження історії станів, роботи з гілками, злиття змін, обробки конфліктів і синхронізації між локальними репозиторіями. Розроблювана система має працювати з локальною файловою системою користувача та не вимагати мережевого сервера для демонстрації основних операцій.

Основним об’єктом роботи програмного прототипу є каталог програмного проєкту. Після ініціалізації системи в цьому каталозі має створюватися службовий каталог локального репозиторію, у якому зберігатимуться об’єкти, метадані комітів, індекс поточного стану та покажчик останньої зафіксованої версії. Службові дані повинні бути відокремлені від робочих файлів проєкту, щоб система могла виконувати аналіз змін і відновлення попередніх станів без порушення структури основного каталогу.

Програмний прототип повинен підтримувати набір команд, достатній для демонстрації повного циклу локальної DVCS. До таких команд належать ініціалізація репозиторію, перевірка стану робочої копії, додавання файлів до індексу, створення коміту, перегляд історії змін, створення та переключення гілок, злиття гілок, обробка конфліктів, додавання remote-шляху, клонування, передавання та отримання змін між локальними репозиторіями.

Для реалізації програмного прототипу обрано мову програмування Python. Вибір цієї мови зумовлений наявністю стандартних бібліотек для роботи з файловою системою, обчислення хеш-значень, стиснення даних, збереження структурованих даних, реалізації командного інтерфейсу та автоматизованого тестування [9]. Для роботи з файловими шляхами використано pathlib та os [10; 11], для хешування файлів – hashlib і стандарт SHA-1 [12; 13], для стиснення об’єктів – zlib [14; 15], для збереження метаданих – json і формат JSON [16; 17], для обробки аргументів командного рядка – argparse [18], для тестування – unittest [19], tempfile [20] і subprocess [21].

Попередні функціональні вимоги до програмного прототипу подано в таблиці 1.3. Окрім них, для програмного прототипу необхідно визначити

попередні нефункціональні вимоги, які впливають на якість реалізації, зручність тестування та подальше розширення системи. Вони подані в таблиці 1.4 [7].

Таблиця 1.3 – Попередні функціональні вимоги до програмного прототипу

ID	Вимога
FR-01	Система повинна підтримувати ініціалізацію локального репозиторію в каталозі проєкту.
FR-02	Система повинна створювати службову структуру для збереження об’єктів, комітів, індексу та покажчика поточного стану.
FR-03	Система повинна визначати нові, змінені та видалені файли у робочій копії.
FR-04	Система повинна обчислювати хеш-значення вмісту файлів для ідентифікації змін.
FR-05	Система повинна створювати коміт як зафіксований стан проєкту.
FR-06	Система повинна зберігати метадані коміту: ідентифікатор, дату створення, повідомлення, перелік файлів і посилання на попередній коміт.
FR-07	Система повинна забезпечувати перегляд історії створених комітів.
FR-08	Система повинна підтримувати відновлення файлів до стану, зафіксованого у вибраному коміті.
FR-09	Система повинна виводити користувачу повідомлення про результат виконання кожної команди.

Таблиця 1.4 – Нефункціональні вимоги до програмного прототипу

ID	Вимога
NFR-01	Система повинна мати модульну структуру програмного коду.
NFR-02	Службові дані репозиторію повинні зберігатися у відтворюваному форматі JSON або у стисненому об’єктному сховищі.
NFR-03	Основні команди системи повинні виконуватися локально без підключення до мережі.
NFR-04	Структура репозиторію повинна бути зрозумілою для аналізу, тестування та демонстрації на захисті.
NFR-05	Реалізація повинна дозволяти перевірку основних сценаріїв роботи за допомогою автоматизованих тестів unittest.
NFR-06	Система повинна обробляти типові помилки користувача, зокрема звернення до неіснуючої гілки, повторне створення гілки та конфлікт під час злиття.
NFR-07	Система повинна працювати в середовищі Windows і підтримувати UTF-8 для текстових файлів та повідомлень командного інтерфейсу.

Архітектура програмного прототипу передбачає поділ на окремі компоненти: модуль командного інтерфейсу, модуль ініціалізації репозиторію, модуль аналізу стану робочої копії, модуль хешування, модуль збереження об’єктів, модуль формування комітів, модуль роботи з гілками, модуль checkout, модуль merge, модуль локальної віддаленої синхронізації та модуль тестування. Такий розподіл забезпечує ізоляцію відповідальності між частинами системи та спрощує перевірку реалізованих функцій [8].

У межах кваліфікаційної роботи встановлюються такі обмеження розроблення. Прототип реалізує локальну модель DVCS і підтримує синхронізацію між репозиторіями через шлях до іншої папки на диску, але не реалізує мережеві протоколи HTTP або SSH. Операції злиття виконуються в спрощеній формі, а конфлікти обробляються через маркери у файлах і службовий файл MERGE_HEAD. Автентифікація користувачів, керування правами доступу, повноцінне рядкове автоматичне злиття та графічний інтерфейс залишаються напрямами подальшого розвитку системи.

Загалом, у першому розділі було розглянуто предметну область систем контролю версій та визначено їхнє місце в інженерії програмного забезпечення. Встановлено, що системи контролю версій забезпечують кероване збереження історії змін, фіксацію станів програмного проєкту, відновлення попередніх версій файлів і підтримку паралельної роботи кількох розробників над спільним кодом. Було проаналізовано базові поняття, на яких ґрунтується функціонування систем контролю версій: робоча копія, індекс змін, коміт, репозиторій, історія змін, гілка, покажчик HEAD і віддалений репозиторій. Визначено, що ці сутності формують логічну модель VCS і безпосередньо впливають на подальше проєктування структури локального репозиторію, формату метаданих та алгоритмів основних команд.

У ході аналізу наявних систем контролю версій було розглянуто Git, Apache Subversion і Mercurial. Визначено, що ці системи мають спільний набір базових функцій: фіксацію змін, збереження історії, перегляд попередніх станів, відновлення версій і підтримку командної роботи. Водночас, вони відрізняються

архітектурною моделлю, способом збереження історії, рівнем автономності користувача та складністю виконання окремих операцій.

На основі проведеного аналізу сформульовано задачу на розроблення програмного прототипу локальної системи контролю версій. Визначено, що прототип повинен підтримувати ініціалізацію репозиторію, аналіз стану робочої копії, хешування файлів, створення комітів, збереження метаданих, перегляд історії змін, гілкування, checkout, merge, обробку конфліктів, clone, push і pull між локальними репозиторіями. Подальший розвиток системи пов’язано не з базовими механізмами VCS, а з мережевими протоколами, авторизацією, графічним інтерфейсом та оптимізацією роботи з великими репозиторіями.

РОЗДІЛ 2 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Аналіз вимог до програмного забезпечення

На основі аналізу предметної області, виконаного в попередньому розділі, програмний продукт доцільно розглядати як навчально-прикладний прототип локальної розподіленої системи контролю версій. Його основне призначення полягає у демонстрації внутрішньої логіки роботи VCS: створення службового репозиторію, виявлення змін у робочій копії, хешування вмісту файлів, збереження об’єктів, формування комітів, підтримка гілок, виконання злиття та локальна синхронізація між репозиторіями [1; 4-6].

Програмний продукт не розглядається як повна заміна Git, Mercurial або Subversion. Його функціональність обмежується тими механізмами, які необхідні для відтворення базової моделі DVCS на рівні файлової системи. Такий підхід дозволяє зосередитися не на зовнішній складності промислових інструментів, а на їхніх ключових інженерних принципах: об’єктному сховищі, службових покажчиках, індексі змін, історії комітів і взаємозв’язку між локальним та віддаленим репозиторіями [4- 6].

Для узагальнення взаємодії користувача з програмним прототипом доцільно подати діаграму прецедентів використання. Вона показує, які основні операції виконує розробник під час роботи з локальною системою контролю версій.

На рис. 2.1 розробник виступає єдиним актором системи, оскільки прототип не містить окремих ролей адміністратора, сервера або системи керування доступом. Усі операції виконуються локально через командний інтерфейс. Така модель відповідає поставленій задачі: реалізувати не повноцінну багатокористувацьку платформу, а локальний прототип, що демонструє базові механізми DVCS.

Користувачем системи є розробник, який працює з локальним каталогом програмного проєкту через командний інтерфейс. Взаємодія з прототипом

здійснюється за допомогою набору команд, які імітують типовий цикл роботи із системою контролю версій: ініціалізація репозиторію, додавання файлів до індексу, створення коміту, перегляд стану, робота з гілками, злиття змін, клонування та синхронізація між локальними репозиторіями. Основні сценарії використання програмного прототипу наведено в таблиці 2.1 та охоплюють повний цикл роботи з локальною системою контролю версій: від створення репозиторію до виконання злиття та синхронізації. Важливо, що кожен сценарій має перевірюваний результат, який може бути підтверджений автоматизованими тестами або демонстрацією роботи командного інтерфейсу.

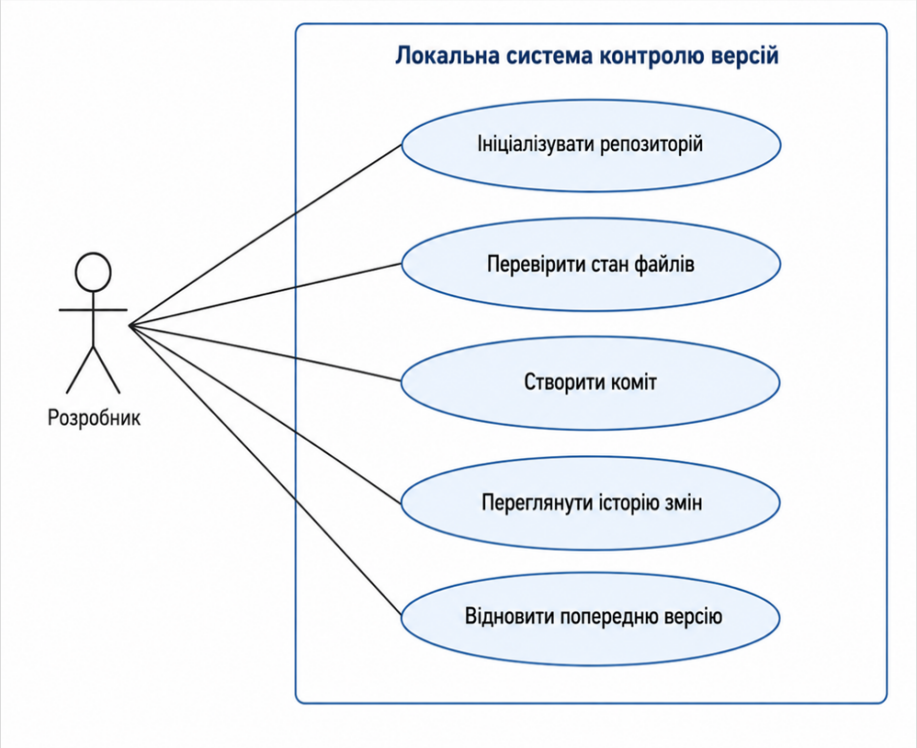

Рисунок 2.1 – Діаграма прецедентів використання локальної системи контролю версій

Функціональні вимоги визначають, які операції повинна виконувати система. Вони безпосередньо пов’язані з командами програмного прототипу та службовими структурами репозиторію. Основні функціональні вимоги наведено в таблиці 2.2.

Таблиця 2.1 – Основні сценарії використання програмного прототипу

ID	Сценарій використання	Вхідні умови	Очікуваний результат
UC-01	Ініціалізація репозиторію	Поточний каталог не містить службового каталогу .vcs	Створено службову структуру репозиторію, файл HEAD, каталог objects і каталог refs
UC-02	Додавання файлу до індексу	Репозиторій ініціалізовано, файл існує в робочій копії	Вміст файлу хешується, об’єкт зберігається у сховищі, запис додається до index
UC-03	Створення коміту	В індексі є підготовлені зміни	Створено об’єкт коміту, оновлено покажчик поточної гілки, індекс очищено
UC-04	Перегляд історії	У репозиторії існує хоча б один коміт	Виведено послідовність комітів поточної гілки з метаданими
UC-05	Створення нової гілки	У репозиторії існує поточний коміт	У каталозі refs/heads створено новий покажчик гілки
UC-06	Перемикання між гілками	Вказана гілка існує	Оновлено HEAD, робочу копію приведено до стану обраної гілки
UC-07	Перегляд стану репозиторію	Репозиторій ініціалізовано	Відображено поточну гілку, змінені, видалені, підготовлені та невідстежувані файли
UC-08	Злиття гілок	Існує поточна гілка та гілка-джерело злиття	Виконано швидке або звичайне злиття; у разі конфлікту створено конфліктні маркери
UC-09	Додавання віддаленого репозиторію	Репозиторій ініціалізовано, шлях до віддаленого каталогу відомий	У config.json записано адресу віддаленого репозиторію
UC-10	Клонування репозиторію	Існує вихідний локальний репозиторій	Створено копію репозиторію з робочою копією та службовими даними
UC-11	Передавання змін	Налаштовано віддалений репозиторій	Локальні об’єкти та покажчики гілок передано до віддаленого репозиторію
UC-12	Отримання змін	Налаштовано віддалений репозиторій	Зміни з віддаленого репозиторію отримано до локального стану

Функціональні вимоги з високим пріоритетом формують мінімально необхідну основу програмного прототипу. До них належать ініціалізація репозиторію, хешування, об’єктне збереження даних, створення комітів, перевірка стану, робота з гілками та злиття. Вимоги середнього пріоритету розширюють систему й демонструють модель розподіленої взаємодії між локальними репозиторіями.

Нефункціональні вимоги визначають якісні характеристики програмного продукту. Для розроблюваного прототипу найбільше значення мають коректність, надійність, відтворюваність результатів, переносимість і

тестованість. Нефункціональні вимоги наведено в таблиці 2.3. Вони дозволяють оцінювати не лише наявність команд, а й якість їх реалізації. Для системи контролю версій особливо важливою є цілісність даних, оскільки пошкодження службового сховища або неправильне оновлення покажчиків може призвести до втрати історії змін [7].

Таблиця 2.2 – Функціональні вимоги до програмного прототипу

ID	Вимога	Зміст вимоги	Пріоритет
FR-01	Ініціалізація репозиторію	Створення службового каталогу .vcs з базовою структурою	Високий
FR-02	Хешування файлів	Обчислення SHA-1-хешу для ідентифікації вмісту файлу	Високий
FR-03	Збереження об’єктів	Запис об’єктів у каталог objects у стисненому вигляді	Високий
FR-04	Індексування змін	Збереження підготовлених до коміту файлів у index	Високий
FR-05	Створення коміту	Формування коміту з метаданими, файлами та посиланням на попередній стан	Високий
FR-06	Перегляд історії	Виведення журналу комітів поточної гілки	Середній
FR-07	Перевірка стану	Визначення змінених, видалених, підготовлених і невідстежуваних файлів	Високий
FR-08	Робота з гілками	Створення та збереження гілок у refs/heads	Високий
FR-09	Перемикання гілок	Перехід на існуючу гілку та оновлення робочої копії	Високий
FR-10	Злиття змін	Виконання швидкого або звичайного злиття з обробкою конфліктів	Високий
FR-11	Віддалена синхронізація	Підтримка remote add, push і pull між локальними репозиторіями	Середній
FR-12	Клонування	Створення копії локального репозиторію з об’єктами та покажчиками	Середній

Технологічні вимоги визначають засоби, за допомогою яких реалізується програмний прототип. Для цієї роботи обрано мову Python та стандартні бібліотеки, оскільки вони забезпечують роботу з файловою системою, хешуванням, стисненням, JSON-метаданими, аргументами командного рядка та автоматизованим тестуванням без залучення зовнішніх фреймворків. Технологічні засоби реалізації наведено в таблиці 2.4.

Таблиця 2.3 – Нефункціональні вимоги до програмного прототипу

Код	Категорія	Вимога	Характеристика
NFR-01	Коректність	Команди повинні виконуватися відповідно до логіки VCS	Перевіряється тестами
NFR-02	Цілісність даних	Збережені об’єкти не повинні пошкоджуватися під час роботи	Контроль через хеші
NFR-03	Надійність	Помилкові команди не повинні змінювати стан репозиторію	Безпечна обробка помилок
NFR-04	Продуктивність	Основні команди мають виконуватися без помітних затримок	Для малих і середніх проєктів
NFR-05	Переносимість	Прототип має працювати у стандартному середовищі Python	Без зовнішніх залежностей
NFR-06	Зручність	Командний інтерфейс має бути зрозумілим для користувача	Однозначні команди
NFR-07	Підтримуваність	Код має бути придатним до розширення	Поділ логіки за функціями
NFR-08	Тестованість	Основні операції мають перевірятися автоматизовано	Використання unittest
NFR-09	Відтворюваність	Тести мають давати однаковий результат у тих самих умовах	Тимчасові тестові каталоги

Таблиця 2.4 – Технологічні засоби реалізації програмного прототипу

Компонент	Засіб реалізації	Призначення
Мова програмування	Python 3	Реалізація логіки системи та командного інтерфейсу
Хешування	hashlib	Обчислення SHA-1-хешів для файлів і службових об’єктів
Стиснення	zlib	Стиснення та відновлення об’єктів репозиторію
Робота з файлами	pathlib, os, shutil	Створення каталогів, копіювання та оновлення файлів
Метадані	json	Збереження index, config.json і даних комітів
Дата і час	datetime	Формування часових міток комітів
CLI	argparse	Обробка команд і параметрів командного рядка
Тестове середовище	tempfile	Створення ізольованих тимчасових каталогів
Запуск команд	subprocess	Перевірка роботи програми через командний рядок
Тестування	unittest	Модульне та інтеграційне тестування
Діаграми	diagrams.net	Побудова структурних і поведінкових схем

Обраний технологічний набір відповідає навчально-прикладному характеру роботи. Використання стандартних бібліотек Python дає змогу зосередитися на власній логіці системи контролю версій, не приховуючи ключові механізми за готовими сторонніми фреймворками. Це важливо для дипломної роботи, оскільки метою є не лише створення працездатного інструменту, а й

демонстрація принципів його внутрішньої реалізації.

Для реалізації програмного прототипу обрано мову програмування Python. Такий вибір зумовлений тим, що Python забезпечує достатній рівень виразності для швидкої розробки інженерного прототипу, має розвинені стандартні бібліотеки для роботи з файловою системою, хешуванням, стисненням і серіалізацією даних. Крім того, Python є придатним для створення командних утиліт, що відповідає характеру розроблюваної системи контролю версій [9].

У межах роботи принципово не використовуються готові бібліотеки для реалізації VCS. Це дозволяє самостійно відтворити внутрішні механізми систем контролю версій: обчислення хешів, збереження об’єктів, формування комітів, оновлення службових покажчиків і відновлення робочої копії. Такий підхід відповідає інженерній меті кваліфікаційної роботи, оскільки демонструє не використання готового інструменту, а проєктування власної моделі збереження історії змін.

Для ідентифікації вмісту файлів застосовується алгоритм SHA-1, реалізований засобами стандартної бібліотеки hashlib. У прототипі SHA-1 використовується не як криптографічний механізм захисту, а як спосіб отримання стабільного ідентифікатора вмісту. Якщо вміст файлу не змінюється, його хеш залишається однаковим; якщо файл змінено, формується інший ідентифікатор. Це дозволяє порівнювати стани файлів і визначати, які об’єкти вже збережені у сховищі [12; 13].

Для зменшення обсягу службових даних використовується zlib-стиснення. Перед записом у службовий каталог об’єкт стискається, а під час читання відновлюється у початковий вигляд. Такий підхід наближує прототип до принципів об’єктного сховища промислових систем контролю версій, де службові об’єкти зберігаються не як звичайні відкриті текстові файли, а як внутрішні структури репозиторію [14; 15].

Для збереження метаданих використовується формат JSON. Він застосовується для опису індексу, конфігурації віддаленого репозиторію та службової інформації комітів. Перевагою JSON є простота читання, можливість

прямого перетворення між структурами даних Python і текстовим представленням, а також зручність перевірки під час тестування [16; 17].

Робота з файловою системою реалізується за допомогою pathlib, os і shutil. Ці засоби відповідають за створення службових каталогів, перевірку наявності файлів, копіювання даних, видалення та оновлення робочої копії. Оскільки система контролю версій безпосередньо працює з файлами та каталогами, коректне використання файлових операцій є одним із ключових елементів реалізації [10; 11; 22].

Командний інтерфейс реалізується з використанням argparse. Він забезпечує обробку команд користувача та параметрів командного рядка. Для розроблюваного прототипу CLI є доцільнішим за графічний інтерфейс, оскільки основні системи контролю версій також активно використовують командну взаємодію. Крім того, CLI спрощує автоматизоване тестування, оскільки команди можна запускати у контрольованому середовищі [18].

Для перевірки працездатності застосовується unittest. Тестування охоплює як окремі службові функції, так і сценарії взаємодії з програмою через командний рядок. Для створення ізольованих тестових репозиторіїв використовується tempfile, а для запуску команд у режимі, наближеному до дій користувача, застосовується subprocess [19- 21].

Отже, обраний технологічний стек є достатнім для реалізації локального прототипу системи контролю версій. Він не перевантажує роботу сторонніми залежностями, забезпечує прозорість внутрішньої логіки та дозволяє показати ключові механізми VCS на рівні програмного коду.

2.2 Архітектура програмного прототипу системи контролю версій

Архітектура програмного прототипу побудована як набір функціональних модулів, що взаємодіють через файлову систему та службовий каталог .vcs. Центральним елементом системи є командний інтерфейс, через який користувач викликає операції init, add, status, commit, log, branch, checkout, merge, remote add, push, pull і clone. Кожна команда ініціює відповідний алгоритм обробки

службових даних репозиторію [8].

На логічному рівні систему можна поділити на такі складові: модуль командного інтерфейсу, модуль керування репозиторієм, модуль об’єктного сховища, модуль індексу змін, модуль комітів, модуль роботи з гілками, модуль злиття та модуль віддаленої синхронізації. Такий поділ дозволяє описати відповідальність кожної частини системи та уникнути змішування різних операцій в одному рівні реалізації.

Модуль командного інтерфейсу приймає команди користувача, перевіряє їхні аргументи та передає керування відповідним функціям. Він не зберігає дані репозиторію самостійно, а виконує роль точки входу до системи. Така організація спрощує використання прототипу та дозволяє запускати однакові команди як вручну, так і в автоматизованих тестах.

Модуль керування репозиторієм відповідає за перевірку наявності службового каталогу, створення початкової структури .vcs, читання та оновлення службових файлів. До його функцій належить перевірка того, чи перебуває користувач у межах ініціалізованого репозиторію, а також забезпечення доступу до службових шляхів objects, refs/heads, HEAD, index, config.json і MERGE_HEAD.

Модуль об’єктного сховища реалізує запис і читання об’єктів. Перед збереженням дані хешуються, після чого стискаються та записуються у каталог objects. Ідентифікатор об’єкта визначає його розміщення у службовому каталозі. Такий підхід дозволяє уникнути дублювання однакових об’єктів: якщо вміст файлу не змінився, його хеш залишається тим самим [6; 12; 14].

Модуль індексу змін відповідає за staging area. Він зберігає інформацію про файли, підготовлені до наступного коміту. Під час виконання команди add файл зчитується, його вміст зберігається як об’єкт, а до index додається відповідний запис. Після успішного створення коміту індекс очищується, що відображає завершення операції фіксації змін.

Модуль комітів формує метадані коміту. До них належать ідентифікатор, дата створення, автор, повідомлення, список зафіксованих файлів і посилання на

попередній коміт. Якщо коміт створюється внаслідок злиття, у ньому може бути зафіксовано кілька батьківських комітів. Це дозволяє відобразити факт об’єднання двох незалежних ліній розроблення.

Модуль гілок працює з каталогом refs/heads. Кожна гілка зберігається як окремий файл-покажчик, що містить ідентифікатор останнього коміту цієї гілки. Файл HEAD визначає, яка гілка є поточною. Така модель є спрощеним відтворенням принципу, що використовується у розподілених системах контролю версій [5].

Модуль злиття відповідає за інтеграцію змін з однієї гілки в іншу. Якщо поточна гілка не має власних змін відносно гілки-джерела, може бути виконано швидке злиття. Якщо зміни відбувалися паралельно, створюється новий коміт злиття. У разі суперечливих змін в одному файлі система формує конфліктні маркери та службовий файл MERGE_HEAD.

Модуль віддаленої синхронізації реалізує спрощену модель роботи з remote. На відміну від промислових систем, прототип не використовує мережеві протоколи HTTP або SSH. Віддалений репозиторій подається як інший локальний каталог. Це дозволяє продемонструвати принципи push, pull і clone без реалізації серверної інфраструктури [4].

Організаційну програмного прототипу подано на рис. 2.2. Користувач взаємодіє з прототипом через CLI-модуль. Далі запит передається до відповідного функціонального компонента, який працює зі службовим каталогом .vcs. Основні дані зберігаються у файловій системі, а службові метадані подаються у форматі JSON. Така архітектура є достатньою для реалізації навчального DVCS-прототипу, оскільки забезпечує відокремлення командного інтерфейсу від логіки збереження об’єктів і керування станом репозиторію.

Модель збереження даних у програмному прототипі ґрунтується на використанні службового каталогу .vcs. Цей каталог створюється під час ініціалізації репозиторію та містить усі внутрішні дані, необхідні для роботи системи контролю версій. Робочі файли проєкту залишаються у звичайному

каталозі користувача, а службові об’єкти, покажчики, індекс і конфігурація зберігаються окремо.

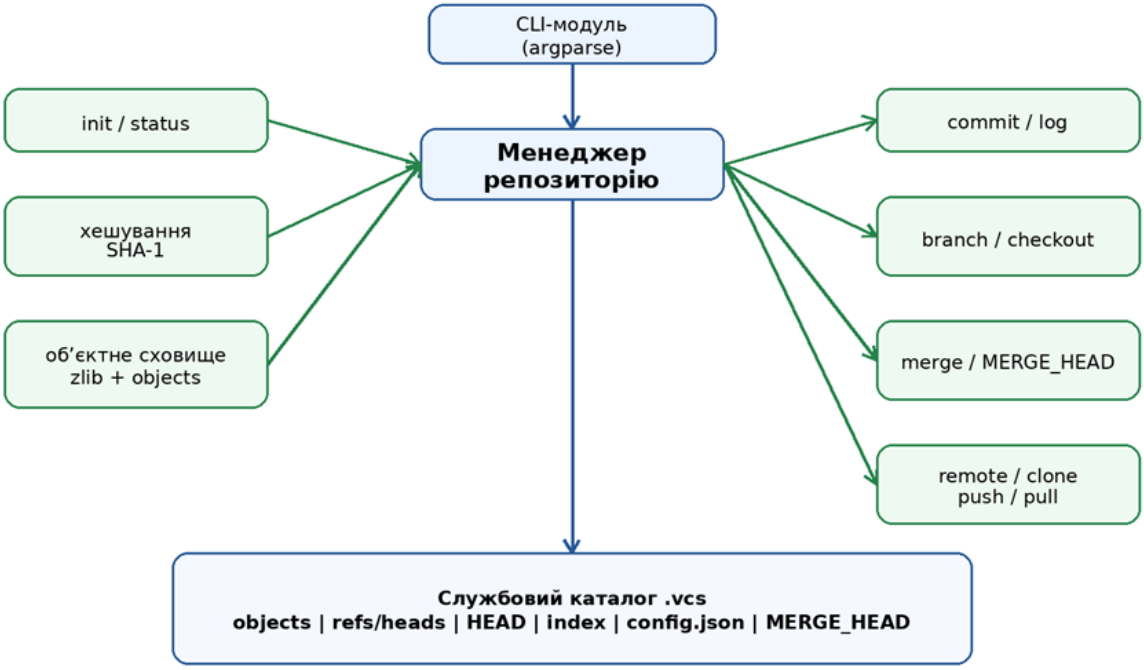

Рисунок 2.2 – Організаційна схема програмного прототипу системи контролю версій

Основними елементами службового каталогу є objects, refs/heads, HEAD, index, config.json і MERGE_HEAD. Каталог objects містить збережені об’єкти репозиторію. До них належать вміст файлів та службові структури комітів. Каталог refs/heads використовується для збереження покажчиків гілок. Файл HEAD визначає поточну активну гілку. Файл index виконує роль області підготовки змін до коміту. Файл config.json зберігає налаштування віддаленого репозиторію. Файл MERGE_HEAD створюється під час незавершеного злиття та містить ідентифікатор коміту гілки, яка зливається.

Структуру службового каталогу .vcs локального репозиторію подано на рис. 2.3. Він виконує роль внутрішнього сховища системи. Каталог objects містить стиснені об’єкти, адресовані SHA-1-хешем. Каталог refs/heads містить файли гілок, кожен із яких указує на останній коміт відповідної гілки. Файл HEAD визначає активну гілку або поточний стан. Файл index містить перелік

файлів, підготовлених до наступного коміту. Файл config.json використовується для збереження шляху до віддаленого репозиторію origin. Файл MERGE_HEAD застосовується для фіксації стану незавершеного злиття.

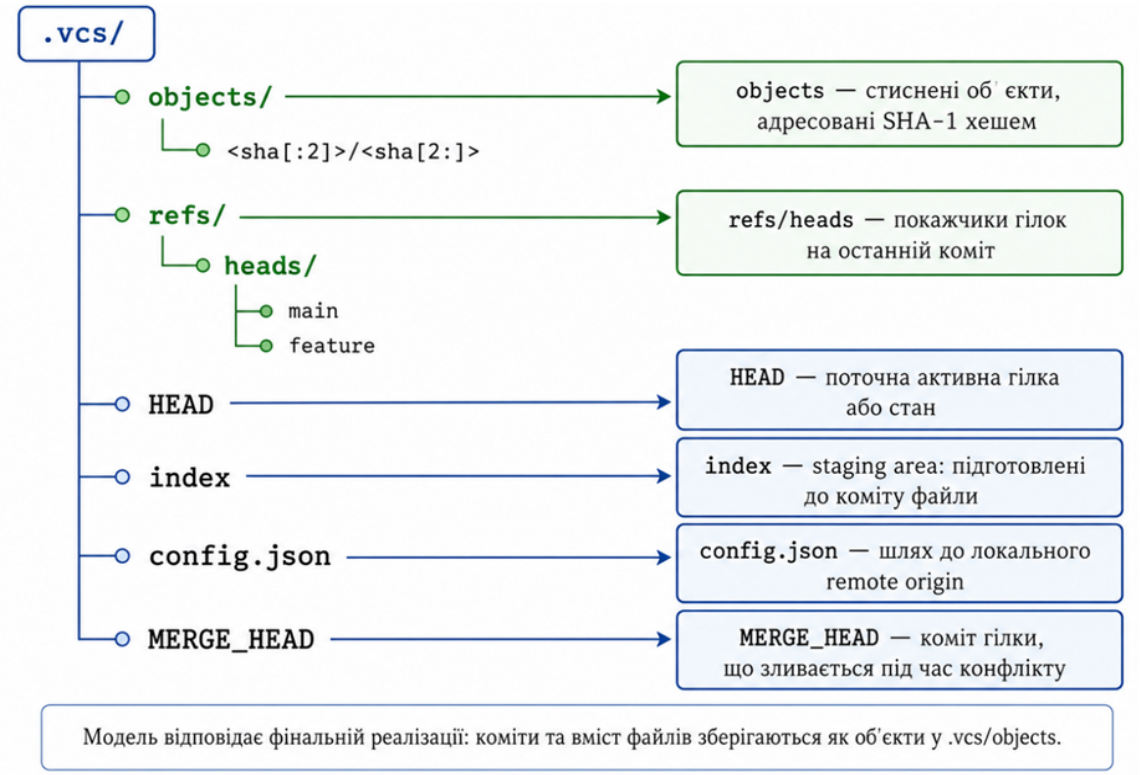

Рисунок 2.3 – Структура службового каталогу локального репозиторію

Збереження об’єктів виконується за хеш-адресованою моделлю. Спочатку система зчитує вміст файлу або службової структури, після чого обчислює хеш за алгоритмом SHA-1. Отриманий хеш використовується як ідентифікатор об’єкта. Для зручності зберігання перші символи хешу можуть використовуватися як назва підкаталогу, а решта — як назва файлу. Це зменшує кількість файлів в одному каталозі й наближує модель до принципів організації об’єктного сховища у промислових VCS [6; 12; 13].

Перед фізичним записом у сховище об’єкт стискається засобами zlib. Така операція зменшує обсяг службових даних і демонструє принцип внутрішнього пакування об’єктів. Під час читання об’єкта система знаходить його за хешем, зчитує байтові дані, виконує розпакування та повертає початковий вміст для подальшої обробки [14; 15].

Коміт зберігається як службова структура, що містить метадані зафіксованого стану. До складу коміту входять повідомлення користувача, дата створення, автор, посилання на попередній коміт або кілька батьківських комітів, а також перелік файлів і відповідних хешів. Така структура дозволяє відновити стан робочої копії, оскільки для кожного файлу відомо, який об’єкт у каталозі objects відповідає його вмісту [5; 6].

Індекс змін є проміжним шаром між робочою копією та комітом. Він дозволяє не фіксувати всі зміни автоматично, а підготувати до коміту лише ті файли, які користувач явно додав командою add. Після створення коміту index очищується або оновлюється відповідно до нового зафіксованого стану. Це дає змогу підтримувати логічну завершеність комітів і відокремлювати підготовлені зміни від непідготовлених [5].

Модель гілок реалізовано через файлові покажчики. Кожен файл у refs/heads містить ідентифікатор останнього коміту відповідної гілки. Створення гілки полягає у створенні нового файлу-покажчика, що посилається на поточний коміт. Перемикання гілки змінює HEAD і приводить робочу копію до стану, визначеного останнім комітом обраної гілки [5].

Віддалена синхронізація в прототипі реалізована у спрощеному вигляді. Віддалений репозиторій не є сервером, а подається як інший локальний каталог. Команда remote add записує шлях до цього каталогу в config.json. Команди push і pull виконують копіювання необхідних об’єктів і оновлення покажчиків між локальним і віддаленим репозиторіями. Така модель не охоплює мережеві протоколи, але дозволяє продемонструвати принципи обміну змінами між репозиторіями [4].

2.3 Алгоритми основних операцій системи контролю версій

Основні операції програмного прототипу побудовані навколо роботи з файловою системою, об’єктним сховищем і службовими покажчиками. Кожна команда виконує чітко визначену послідовність дій, а результат її роботи відображається у зміні службових файлів репозиторію або стану робочої копії.

Команда init створює службовий каталог .vcs. На цьому етапі формується базова структура репозиторію: каталог objects для об’єктів, каталог refs/heads для покажчиків гілок, файл HEAD для визначення активної гілки, файл index для області підготовки змін і файл config.json для майбутніх налаштувань віддаленої синхронізації. Після ініціалізації система може виконувати інші команди тільки в межах створеного репозиторію [5].

Команда add виконує підготовку файлу до коміту. Спочатку система перевіряє наявність репозиторію та існування вказаного файлу. Далі вміст файлу зчитується, для нього обчислюється SHA-1-хеш, після чого об’єкт стискається та записується у сховище objects. Після цього в index додається запис, який пов’язує шлях файлу з відповідним хешем. Якщо файл уже був збережений раніше й його вміст не змінився, хеш залишається тим самим [6; 12; 14].

Команда status аналізує поточний стан робочої копії. Для цього система зчитує останній зафіксований стан, поточний index і фактичний вміст робочого каталогу. Далі порівнюються хеші файлів. Якщо файл існує в робочій копії, але відсутній у попередньому стані, він визначається як невідстежуваний. Якщо файл був зафіксований, але його поточний хеш відрізняється від збереженого, він позначається як змінений. Якщо файл був зафіксований, але відсутній у робочій копії, він позначається як видалений. Якщо файл додано до index, він відображається як підготовлений до коміту.

Команда commit фіксує підготовлені зміни в історії репозиторію. Перед створенням коміту система перевіряє, чи існує репозиторій і чи містить index підготовлені файли. Якщо змін немає, команда завершується без створення нового коміту. Якщо зміни наявні, система формує метадані коміту: дату, автора, повідомлення, перелік файлів, посилання на попередній коміт і, за потреби, посилання на другий батьківський коміт після злиття. Після цього коміт зберігається як об’єкт, а покажчик поточної гілки оновлюється до нового ідентифікатора.

Алгоритм створення коміту подано на рис. 2.4. Процес починається з перевірки наявності репозиторію та аналізу підготовлених змін. Якщо index

порожній, система повідомляє користувача про відсутність змін для фіксації. Якщо зміни наявні, система зберігає відповідні об’єкти, формує метадані коміту, записує коміт до сховища, оновлює покажчик поточної гілки та очищує index. Таким чином, коміт стає новим зафіксованим станом локального репозиторію.

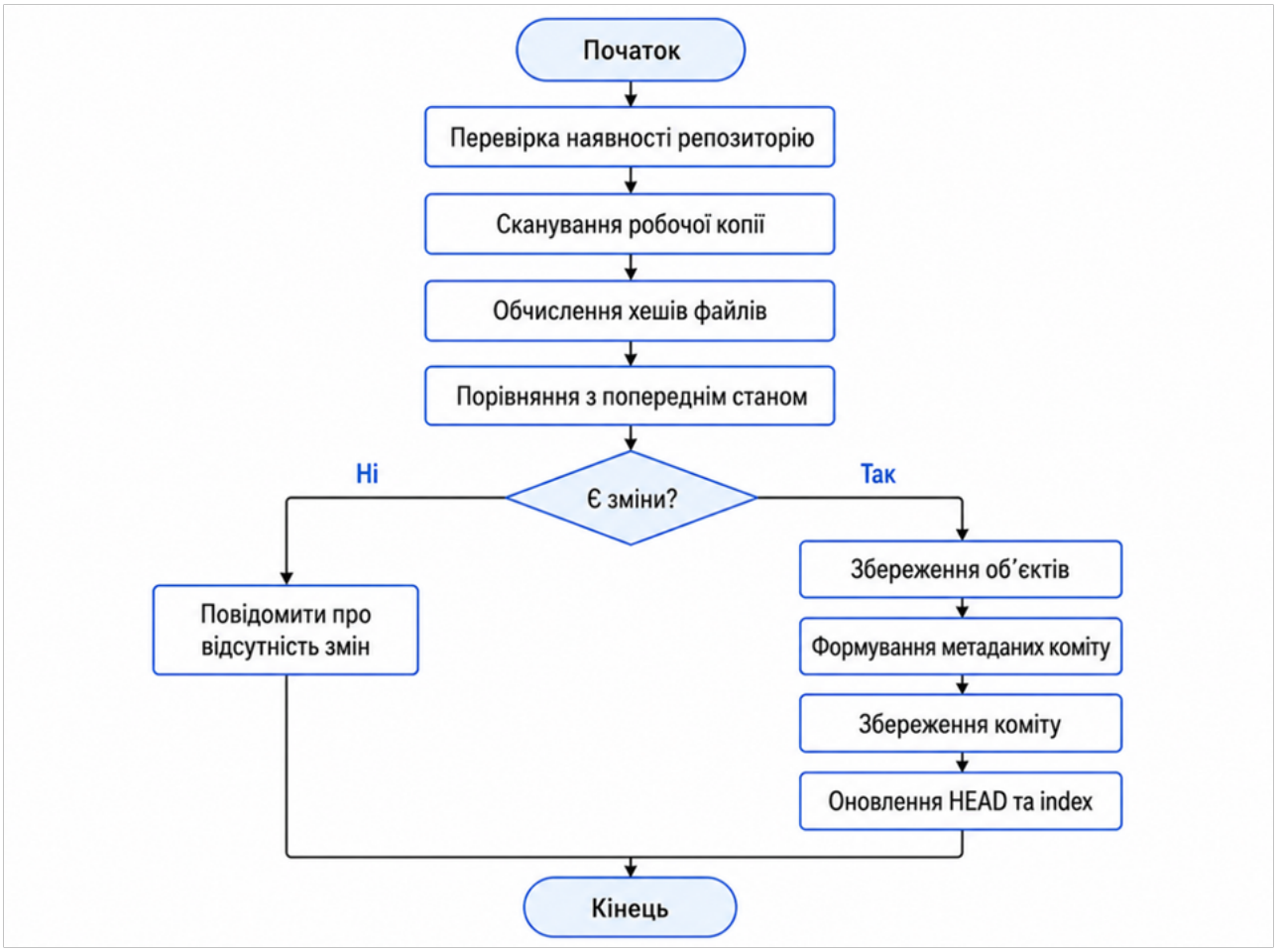

Рисунок 2.4 – Алгоритм створення коміту в локальній системі контролю версій

Команда log використовується для перегляду історії. Система визначає поточну гілку через HEAD, отримує ідентифікатор останнього коміту з refs/heads і послідовно переходить до батьківських комітів. Для кожного коміту виводяться його ідентифікатор, дата, автор, повідомлення та перелік зафіксованих файлів. Такий підхід дозволяє простежити історію розвитку проєкту в межах активної гілки [5].

Команда branch створює нову гілку або відображає список існуючих гілок. Якщо користувач вказує назву нової гілки, система перевіряє наявність

поточного коміту та створює відповідний файл у refs/heads. Значенням цього файлу стає ідентифікатор коміту, на якому була створена гілка. Якщо гілка з такою назвою вже існує, команда завершується з повідомленням про помилку [5].

Команда checkout використовується для перемикання між гілками або відновлення робочої копії до стану певного коміту. Система перевіряє, чи існує вказана гілка, зчитує її останній коміт, отримує перелік файлів і відповідних об’єктів, після чого оновлює робочий каталог. Якщо гілка не існує, команда завершується без зміни HEAD і без модифікації робочої копії. Такий підхід забезпечує безпечну обробку помилкових сценаріїв [5].

Алгоритм відновлення стану робочої копії подано на рис. 2.5. Відновлення починається з перевірки наявності репозиторію та цільової гілки. Якщо гілку знайдено, система зчитує відповідний коміт, отримує перелік файлів і хешів, відновлює відстежувані файли у робочій копії, оновлює index і змінює HEAD. Якщо цільова гілка не існує, команда завершується з повідомленням про помилку без зміни поточного стану репозиторію. Команда merge реалізує злиття змін з однієї гілки в іншу. Якщо поточна гілка може бути просто переміщена до коміту гілки-джерела, виконується швидке злиття. Якщо обидві гілки мають власні незалежні зміни, система формує новий коміт злиття. У випадку, коли один і той самий файл було змінено в обох гілках несумісним способом, система створює конфліктні маркери у файлі та записує службову інформацію в MERGE_HEAD. Після ручного вирішення конфлікту користувач може додати виправлений файл до index і створити коміт злиття. Команди remote add, push, pull і clone реалізують спрощену локальну модель віддаленої синхронізації. Команда remote add записує шлях до віддаленого репозиторію в config.json. Команда push передає локальні об’єкти та покажчики до віддаленого каталогу. Команда pull отримує дані з віддаленого репозиторію до локального. Команда clone створює копію існуючого репозиторію в новому каталозі. Така модель дозволяє продемонструвати принципи розподіленої системи контролю версій без реалізації мережевого сервера [4; 5].

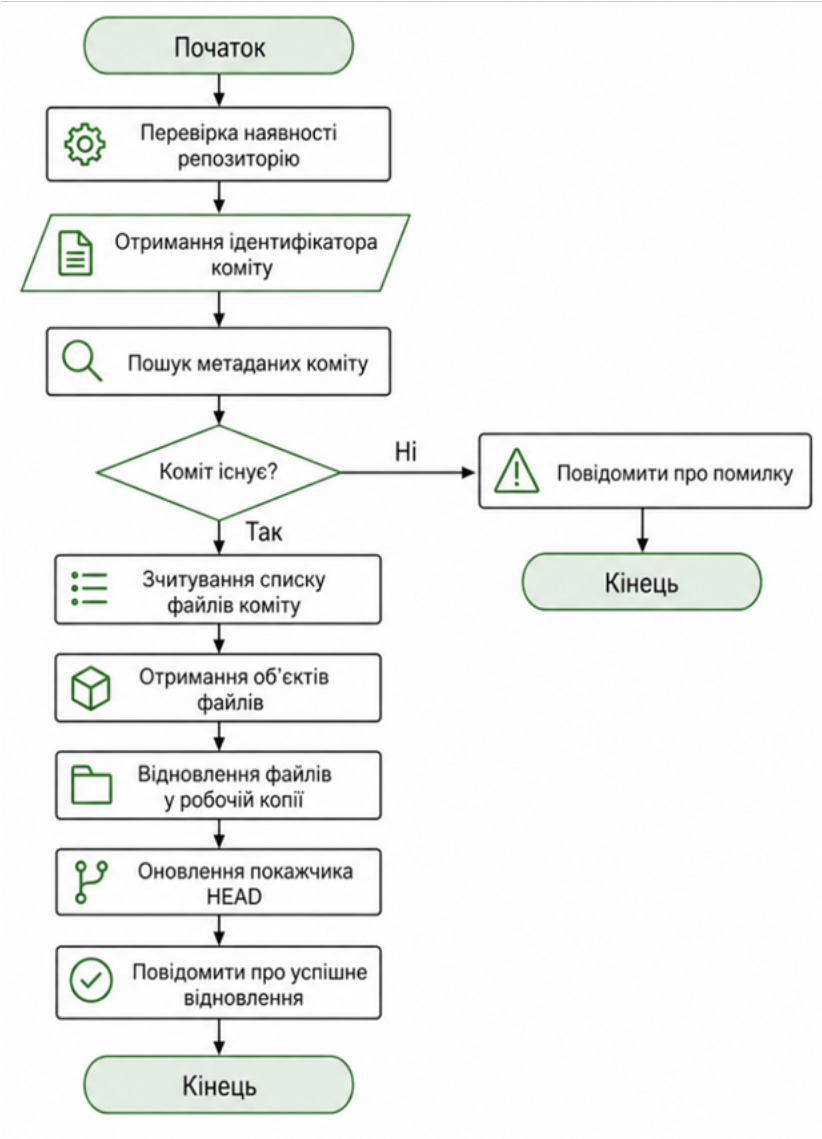

Рисунок 2.5 – Алгоритм відновлення стану робочої копії за комітом

Отже, в другому розділі було сформульовано вимоги до програмного прототипу локальної розподіленої системи контролю версій. Визначено основні сценарії використання, функціональні, нефункціональні та технологічні вимоги. На основі цих вимог обґрунтовано вибір мови програмування Python і стандартних бібліотек для роботи з файловою системою, хешуванням, стисненням, JSON-метаданими, командним інтерфейсом та автоматизованим тестуванням.

Також було спроєктовано компонентну архітектуру програмного прототипу, у межах якої виділено модуль командного інтерфейсу, модуль керування репозиторієм, об’єктне сховище, індекс змін, модуль комітів, модуль

гілок, модуль злиття та модуль віддаленої синхронізації. Визначено структуру службового каталогу .vcs, що включає objects, refs/heads, HEAD, index, config.json і MERGE_HEAD.

Окремо було описано алгоритми основних операцій системи контролю версій: init, add, status, commit, log, branch, checkout, merge, remote add, push, pull і clone. Отримані результати другого розділу є основою для перевірки працездатності програмного прототипу, формування тестових сценаріїв і проведення модульного та інтеграційного тестування у наступному розділі.

РОЗДІЛ 3 ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Вибір методів тестування

Тестування програмного прототипу локальної розподіленої системи контролю версій проводиться з метою перевірки відповідності реалізованої функціональності вимогам, визначеним у другому розділі. Оскільки розроблена система працює з файловою системою, службовим каталогом .vcs, об’єктним сховищем, індексом змін, комітами, гілками та локальною синхронізацією між репозиторіями, тестування має охоплювати як окремі алгоритмічні механізми, так і повні сценарії роботи користувача через командний інтерфейс [23].

Для перевірки програмного продукту обрано два основні методи тестування: модульне та інтеграційне тестування. Модульне тестування використовується для перевірки окремих функцій, які мають чітко визначений вхід і результат. До таких функцій належать генерація SHA-1-хешу, запис об’єкта до службового сховища, читання об’єкта зі сховища, стиснення даних і відновлення початкового вмісту. Саме ці механізми є базовими для роботи об’єктного сховища, тому їхня помилка може вплинути на всі подальші операції системи [23].

Інтеграційне тестування застосовується для перевірки взаємодії між кількома частинами програмного прототипу. У межах таких тестів оцінюється не окрема функція, а повний сценарій роботи системи: ініціалізація репозиторію, додавання файлів до індексу, створення коміту, створення гілки, перемикання між гілками, перевірка стану робочої копії, злиття змін, обробка конфліктів, передавання та отримання змін між локальними репозиторіями. Такий підхід дозволяє перевірити, чи коректно узгоджені між собою службові файли HEAD, index, refs/heads, objects, config.json і MERGE_HEAD.

Для автоматизації тестування використано стандартний фреймворк unittest. Його застосування є доцільним, оскільки він входить до стандартної бібліотеки Python і не потребує встановлення зовнішніх залежностей. Це відповідає загальній логіці розробки прототипу, у якому основна

функціональність реалізується засобами стандартних бібліотек Python. Використання unittest дає змогу групувати перевірки за класами тестів, виконувати їх автоматично та отримувати формалізований результат проходження тестового набору [19; 23].

Для ізоляції тестів використано тимчасові каталоги, створені засобами tempfile. Кожен тест виконується в окремому середовищі, де створюється власний робочий каталог, службовий каталог .vcs, тестові файли та, за потреби, додатковий локальний репозиторій для перевірки синхронізації. Такий підхід унеможливлює вплив тестів на реальні файли користувача та забезпечує повторюваність результатів [20].

Для перевірки сценаріїв, які мають імітувати поведінку реального користувача, застосовується subprocess. Цей інструмент дозволяє запускати програму через командний рядок і перевіряти результат її виконання: код завершення, стандартний вивід, повідомлення про помилки та зміни у файловій структурі репозиторію. Таким чином тестування охоплює не лише внутрішні функції, а й фактичну роботу командного інтерфейсу [21].

Обраний набір методів і засобів тестування відповідає характеру розробленої системи. Модульні тести перевіряють коректність базових алгоритмів, інтеграційні тести підтверджують правильність роботи команд у межах повного сценарію, а використання тимчасових каталогів забезпечує безпечність і відтворюваність тестового процесу.

Як видно з таблиці 3.1, тестування охоплює як внутрішні механізми збереження даних, так і повні сценарії взаємодії з програмним прототипом. Така структура перевірки є достатньою для оцінювання працездатності навчальної DVCS-системи, оскільки дозволяє підтвердити коректність хешування, об’єктного сховища, створення комітів, роботи з гілками, злиття змін і локальної синхронізації між репозиторіями.

Таблиця 3.1 – Методи та засоби тестування програмного прототипу

Метод тестування	Об’єкт перевірки	Засіб реалізації	Очікуваний результат
Модульне тестування	SHA-1-хешування, запис і читання об’єктів	unittest	Окремі функції повертають коректний результат
Модульне тестування	zlib-стиснення та відновлення даних	unittest	Об’єкт після відновлення збігається з початковими даними
Інтеграційне тестування	init, add, commit, log	unittest, subprocess	Команди узгоджено змінюють стан репозиторію
Інтеграційне тестування	branch, checkout, status	unittest, subprocess	Гілки, HEAD та робоча копія оновлюються коректно
Інтеграційне тестування	merge і обробка конфліктів	unittest, tempfile	Злиття виконується коректно або створюються конфліктні маркери
Інтеграційне тестування	remote add, push, pull, clone	unittest, subprocess, tempfile	Дані передаються між локальними репозиторіями
Перевірка відтворюваності	Ізольоване тестове середовище	tempfile	Кожен тест виконується незалежно від інших

3.2 Формування тест-плану

Для перевірки програмного прототипу було сформовано тест-план, який охоплює основні функціональні можливості локальної розподіленої системи контролю версій. Тест-план побудовано відповідно до вимог, визначених у другому розділі, та спрямовано на перевірку як окремих внутрішніх механізмів, так і повних користувацьких сценаріїв роботи з репозиторієм [23].

Основна увага під час формування тест-плану приділялася операціям, які безпосередньо впливають на цілісність історії змін: генерації хешів, збереженню та читанню об’єктів, створенню комітів, роботі з гілками, перемиканню станів, злиттю змін і локальній синхронізації між репозиторіями. Оскільки система працює з файловою структурою, окремо перевіряється коректність створення службових каталогів, оновлення покажчиків HEAD і refs/heads, очищення index після коміту та збереження службових файлів config.json і MERGE_HEAD.

Тестові сценарії згруповано за функціональними блоками. Такий підхід дозволяє відокремити перевірку базових службових функцій від перевірки повних командних сценаріїв. У тест-план включено 12 груп тестів TC001–TC012,

які разом охоплюють 27 автоматизованих перевірок. Узагальнений тест-план наведено в таблиці 3.2 [23].

Таблиця 3.2 – Узагальнений тест-план програмного прототипу

ID	Об’єкт перевірки	Основна перевірка	Очікуваний результат
TC001	SHA-1-хешування	Формування стабільного хешу для вхідних даних	Хеш має довжину 40 символів і збігається з результатом hashlib
TC002	Об’єктне сховище	Запис, стиснення та читання об’єктів	Дані після читання збігаються з початковим вмістом
TC003	Створення коміту	Фіксація підготовлених змін	Створено коміт, оновлено refs/heads/main, index очищено
TC004	Робота з гілками	Створення нової гілки	У refs/heads створено файл нової гілки
TC005	Помилковий checkout	Перехід на неіснуючу гілку	Система повертає помилку, HEAD не змінюється
TC006	Команда status	Визначення стану файлів	Відображаються modified, deleted, staged та untracked файли
TC007	Швидке злиття	Fast-forward merge	Покажчик поточної гілки оновлюється до цільового коміту
TC008	Злиття без конфлікту	Об’єднання незалежних змін	Зміни інтегруються без конфліктних маркерів
TC009	Конфлікт злиття	Зміна одного файла у двох гілках	Створюються конфліктні маркери та MERGE_HEAD
TC010	Коміт злиття	Фіксація результату merge	Коміт містить посилання на батьківські коміти
TC011	Push та pull	Локальна синхронізація репозиторіїв	Об’єкти й покажчики передаються між репозиторіями
TC012	Clone	Клонування локального репозиторію	Створено копію репозиторію зі службовою структурою

Тест-план охоплює всі ключові групи функціональності програмного прототипу. Перші дві групи тестів перевіряють базові службові механізми: хешування, стиснення, запис і читання об’єктів. Наступні групи перевіряють роботу з комітами, гілками, станом робочої копії та злиттям. Окремо виділено перевірку локальної синхронізації та клонування, оскільки ці операції демонструють розподілену модель роботи системи.

Для деталізації тест-плану було сформовано набір тестових випадків, у яких зазначено послідовність дій, вхідні дані та очікуваний результат. Основні тестові випадки наведено в таблиці 3.3.

Таблиця 3.3 – Основні тестові випадки програмного прототипу

ID	Опис кроків	Вхідні дані	Очікуваний результат
TC 001	Викликати функцію хешування для текстового вмісту та порівняти результат із hashlib.sha1	Рядок test data	Отримано SHA-1-хеш довжиною 40 символів
TC 002	Записати текстовий файл у сховище, після цього зчитати його за хешем	Файл text.txt	Після відновлення вміст збігається з початковим
TC 002	Записати бінарні дані у сховище та перевірити можливість їх читання	Бінарний файл	Дані коректно стискаються і відновлюються
TC 003	Виконати init, add, commit і перевірити refs/heads/main	Файл file.txt, повідомлення first commit	Створено коміт, main містить SHA-1 ідентифікатор
TC 003	Перевірити стан index після створення коміту	Підготовлені зміни в index	Після коміту index очищено
TC 004	Створити нову гілку від поточного коміту	Назва гілки feature	У refs/heads створено файл feature
TC 004	Спробувати створити гілку з уже наявною назвою	Назва існуючої гілки	Система повертає повідомлення про помилку
TC 005	Виконати checkout для неіснуючої гілки	Назва гілки unknown	Команда завершується з помилкою, HEAD не змінюється
TC 006	Змінити файл після коміту та виконати status	Відстежуваний файл	Файл відображається як Modified
TC 006	Додати новий файл без коміту та виконати status	Новий файл new.txt	Файл відображається як Untracked
TC 007	Виконати злиття, коли поточна гілка не має власних нових комітів	Гілки main і feature	Виконується швидке злиття
TC 009	Змінити один файл у двох гілках і виконати merge	Конфліктний файл	У файлі створюються конфліктні маркери
TC 011	Налаштувати remote, виконати push і pull між двома локальними репозиторіями	Два локальні каталоги	Дані синхронізуються між репозиторіями
TC 012	Виконати clone існуючого локального репозиторію	Шлях до репозиторію	Створено копію з каталогом .vcs та робочими файлами

Наведені тестові випадки перевіряють не лише позитивні сценарії, а й помилкові ситуації. Зокрема, окремо перевіряється спроба перейти на неіснуючу гілку, створити дубльовану гілку, прочитати неіснуючий об’єкт або виконати

злиття з конфліктом. Такий підхід дозволяє оцінити не тільки працездатність основних команд, а й стійкість системи до некоректних дій користувача.

Усі тести виконуються в ізольованих тимчасових каталогах. Це означає, що кожен тест створює власне середовище, ініціалізує репозиторій, формує тестові файли та після завершення не впливає на результати інших перевірок. Така організація тестування забезпечує відтворюваність і дозволяє багаторазово запускати весь тестовий набір без ручного очищення робочої директорії.

Результатом виконання тест-плану має бути підтвердження того, що програмний прототип коректно реалізує основні механізми системи контролю версій: хешування даних, об’єктне збереження, створення комітів, роботу з гілками, перевірку стану файлів, злиття, обробку конфліктів і локальну синхронізацію між репозиторіями.

3.3 Організація комп’ютерних експериментів

Комп’ютерні експерименти проводилися з метою практичної перевірки працездатності програмного прототипу локальної розподіленої системи контролю версій. Перевірка виконувалася у середовищі командного рядка шляхом запуску автоматизованих тестів, написаних із використанням стандартного фреймворку unittest. Додатково застосовувалися тимчасові каталоги, що дозволило створювати ізольовані тестові репозиторії без впливу на реальні файли користувача.

Перед запуском тестування було підготовлено робочий каталог проєкту, який містив основний файл реалізації vcs.py, файл тестів test_vcs.py та набір службових і тестових файлів. Після ініціалізації репозиторію система створювала службовий каталог .vcs, у якому зберігалися об’єкти, покажчики гілок, файл HEAD, index і конфігураційні дані. Така структура дозволила перевірити не лише роботу окремих команд, а й зміну внутрішнього стану репозиторію після кожної операції.

На рис. 3.1 показано структуру робочого каталогу, у якому виконувалося тестування. Наявність файлів vcs.py і test_vcs.py підтверджує розділення

програмної реалізації та тестового коду. Службовий каталог .vcs використовується для збереження внутрішнього стану репозиторію, зокрема об’єктів, покажчиків гілок, індексу та активної гілки.

Рисунок 3.1 – Структура тестового середовища програмного прототипу

Перший етап експериментальної перевірки був спрямований на тестування базових службових механізмів. До них належать генерація SHA-1-хешів, запис об’єктів до сховища, zlib-стиснення, читання об’єктів і відновлення початкового вмісту. Ці операції є основою роботи всієї системи, оскільки кожен файл і кожен коміт у прототипі ідентифікується через хеш-значення та зберігається як об’єкт у службовому каталозі.

Як показано на рис. 3.2, тести груп TC001 і TC002 завершилися успішно. Це підтверджує коректність генерації SHA-1-хешу, стабільність результату для однакових вхідних даних, правильне збереження об’єктів у каталозі objects, а також можливість відновлення текстових і бінарних даних після стиснення.

Рисунок 3.2 – Результат проходження тестів хешування та об’єктного сховища

Другий етап тестування охоплював перевірку команд створення комітів, роботи з гілками та обробки помилкових сценаріїв. Під час перевірки створювався тестовий репозиторій, додавалися файли до індексу, виконувалася фіксація змін, створювалися гілки та перевірялася реакція системи на спробу переходу до неіснуючої гілки. Окремо перевірялося, що після невдалої операції checkout файл HEAD не змінюється.

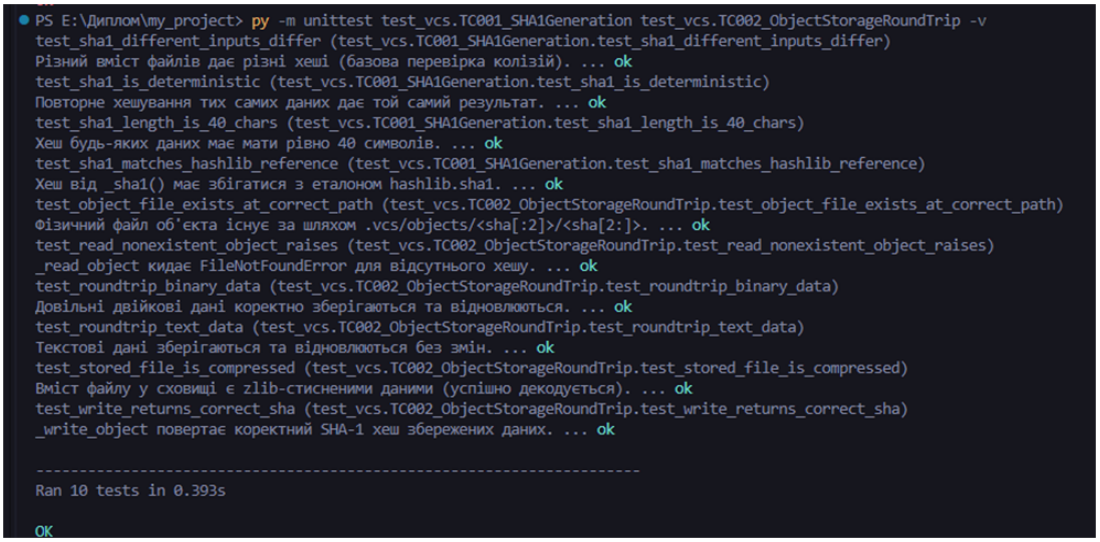

Рисунок 3.3 – Результат проходження тестів створення комітів і роботи з гілками

Результати, наведені на рис. 3.3, підтверджують, що команда commit створює службовий об’єкт коміту, оновлює покажчик поточної гілки та очищує index після успішної фіксації. Також підтверджено коректність створення нових гілок у каталозі refs/heads і правильну обробку помилки під час спроби перейти до гілки, якої не існує.

Третій етап комп’ютерних експериментів був спрямований на перевірку розширеної функціональності: визначення стану робочої копії, швидке злиття, звичайне злиття, обробку merge-конфліктів, створення коміту злиття, локальну синхронізацію між репозиторіями та клонування. Саме ці перевірки дозволяють оцінити прототип як спрощену локальну модель розподіленої системи контролю версій.

На рис. 3.4 показано результат запуску повного набору автоматизованих тестів. Усього виконано 27 перевірок, згрупованих у 12 тестових сценаріїв TC001–TC012. Усі перевірки завершилися успішно, що підтверджує коректність реалізації основних команд програмного прототипу та узгодженість роботи службових структур репозиторію.

```
PS E:\Диплом\my_project> py -m unittest test_vcs.TC003_CommitCreation test_vcs.TC004_BranchCreation test_vcs.TC005_CheckoutNonexistentBranch -v
test_commit_message_matches_input (test_vcs.TC003_CommitCreation.test_commit_message_matches_input)
Поле 'message' у коміті збігається з переданим рядком. ... ok
test_commit_object_has_required_fields (test_vcs.TC003_CommitCreation.test_commit_object_has_required_fields)
Об'єкт коміту у .vcs/objects/ містить усі обов'язкові поля. ... ok
test_first_commit_has_null_parent (test_vcs.TC003_CommitCreation.test_first_commit_has_null_parent)
Перший коміт у гілці має parent == null (немає попереднього коміту). ... ok
test_index_is_cleared_after_commit (test_vcs.TC003_CommitCreation.test_index_is_cleared_after_commit)
Індекс очищується після успішного коміту. ... ok
test_main_ref_contains_sha_after_commit (test_vcs.TC003_CommitCreation.test_main_ref_contains_sha_after_commit)
Після коміту файл refs/heads/main містить 40-символьний SHA-1. ... ok
test_branch_file_is_created (test_vcs.TC004_BranchCreation.test_branch_file_is_created)
Після 'branch feature' файл .vcs/refs/heads/feature існує. ... ok
test_branch_list_shows_new_branch (test_vcs.TC004_BranchCreation.test_branch_list_shows_new_branch)
Команда 'branch' (без аргументів) показує нову гілку у списку. ... ok
test_branch_name_with_dash (test_vcs.TC004_BranchCreation.test_branch_name_with_dash)
Гілки з дефісом у назві (feature-x) створюються коректно. ... ok
test_duplicate_branch_returns_error (test_vcs.TC004_BranchCreation.test_duplicate_branch_returns_error)
Повторне створення гілки з тим самим ім'ям завершується з помилкою. ... ok
test_new_branch_points_to_current_commit (test_vcs.TC004_BranchCreation.test_new_branch_points_to_current_commit)
Хеш нової гілки 'feature' збігається з хешем поточного коміту main. ... ok
test_checkout_unknown_branch_fails (test_vcs.TC005_CheckoutNonexistentBranch.test_checkout_unknown_branch_fails)
'checkout unknown' завершується з ненульовим кодом повернення. ... ok
test_checkout_unknown_branch_prints_error_message (test_vcs.TC005_CheckoutNonexistentBranch.test_checkout_unknown_branch_prints_error_message)
Виведене повідомлення інформує про відсутність гілки. ... ok
test_checkout_without_init_fails (test_vcs.TC005_CheckoutNonexistentBranch.test_checkout_without_init_fails)
'checkout' без ініціалізованого репозиторію завершується з помилкою. ... ok
test_head_points_to_main_initially (test_vcs.TC005_CheckoutNonexistentBranch.test_head_points_to_main_initially)
HEAD вказує на гілку 'main' після init. ... ok
test_head_unchanged_after_failed_checkout (test_vcs.TC005_CheckoutNonexistentBranch.test_head_unchanged_after_failed_checkout)
HEAD залишається незмінним після невдалого checkout. ... ok

----------------------------------------------------------------------
Ran 15 tests in 3.197s

OK
```

Рисунок 3.4 – Результат проходження повного набору автоматизованих тестів

Узагальнені результати комп’ютерних експериментів наведено в таблиці 3.4. Усі групи тестових сценаріїв завершилися успішно. Найбільша кількість перевірок припадає на команду status, обробку конфліктів, коміт злиття та локальну синхронізацію, оскільки ці операції мають найбільшу кількість станів і помилкових сценаріїв. Результати тестування підтверджують, що програмний прототип коректно працює з об’єктним сховищем, службовими покажчиками, індексом змін, гілками та локальними репозиторіями.

Таблиця 3.4 – Результати виконання тестових сценаріїв

ID	Об’єкт перевірки	Кількість перевірок	Результат
TC001	SHA-1-хешування	2	Успішно
TC002	Об’єктне сховище	1	Успішно
TC003	Створення коміту	2	Успішно
TC004	Створення гілки	2	Успішно
TC005	Checkout і помилковий перехід	2	Успішно
TC006	Команда status	5	Успішно
TC007	Швидке злиття	1	Успішно
TC008	Злиття без конфлікту	1	Успішно
TC009	Конфлікт злиття	3	Успішно
TC010	Коміт злиття	3	Успішно
TC011	Push, pull і remote	3	Успішно
TC012	Clone	2	Успішно
Разом	Повний набір тестів	27	Успішно

Окреме значення має перевірка конфліктів злиття. Під час тестування було змодельовано ситуацію, у якій один і той самий файл змінювався у двох різних гілках. У такому випадку система не перезаписує дані автоматично, а створює конфліктні маркери у файлі та зберігає службову інформацію у MERGE_HEAD. Це підтверджує коректну реакцію прототипу на суперечливі зміни.

Також було перевірено локальну віддалену синхронізацію. Для цього створювалися два тимчасові репозиторії, між якими виконувалися операції remote add, push і pull. Результати перевірки показали, що об’єкти та покажчики гілок можуть передаватися між локальними репозиторіями, а команда clone створює копію репозиторію зі службовою структурою та робочими файлами.

Отже, проведені комп’ютерні експерименти підтвердили працездатність розробленого програмного прототипу. Система коректно виконує основні операції локальної DVCS: ініціалізацію репозиторію, додавання файлів до індексу, створення комітів, перегляд історії, роботу з гілками, перемикання станів, злиття, обробку конфліктів, клонування та локальну синхронізацію між репозиторіями.

Загалом, у третьому розділі було проведено тестування програмного прототипу локальної розподіленої системи контролю версій. Для перевірки реалізованої функціональності обрано модульне та інтеграційне тестування, що дозволило оцінити як окремі службові функції, так і повні сценарії роботи користувача через командний інтерфейс. Було сформовано тест-план, який охоплює 12 груп тестових сценаріїв TC001-TC012. У межах цих сценаріїв перевірено генерацію SHA-1-хешів, збереження та читання об’єктів, zlib-стиснення, створення комітів, роботу з гілками, команду status, checkout, merge, обробку конфліктів, remote add, push, pull і clone. Загалом виконано 27 автоматизованих перевірок. Проведене тестування засвідчило, що розроблений програмний прототип відповідає поставленим функціональним вимогам і може бути використаний як навчальний інструмент для демонстрації внутрішніх механізмів систем контролю версій.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було проаналізовано принципи функціонування систем контролю версій та розроблено програмний прототип локальної розподіленої системи контролю версій мовою Python. Розроблене рішення призначене для демонстрації базових інженерних механізмів VCS: збереження історії змін, ідентифікації вмісту файлів, створення комітів, підтримки гілок, виконання злиття та локальної синхронізації між репозиторіями.

У першому розділі було проаналізовано предметну область систем контролю версій, визначено їхню роль у процесі розроблення програмного забезпечення та розглянуто основні поняття, на яких ґрунтується робота таких систем. Окрему увагу приділено порівнянню централізованої та розподіленої моделей контролю версій, а також аналізу наявних рішень Git, Subversion і Mercurial. На основі цього було сформульовано задачу розроблення локального програмного прототипу, який відтворює ключові механізми DVCS у спрощеному навчально-прикладному вигляді.

У другому розділі було сформовано функціональні, нефункціональні та технологічні вимоги до програмного продукту. Визначено основні сценарії використання системи, обґрунтовано вибір мови програмування Python і стандартних бібліотек для роботи з файловою системою, SHA-1-хешуванням, zlib-стисненням, JSON-метаданими, командним інтерфейсом і автоматизованим тестуванням. Також було спроєктовано архітектуру програмного прототипу, описано структуру службового каталогу .vcs та алгоритми основних операцій системи контролю версій.

У межах практичної реалізації було створено командний інтерфейс, що підтримує ініціалізацію репозиторію, додавання файлів до індексу, створення комітів, перегляд історії, перевірку стану робочої копії, створення гілок, перемикання між гілками, злиття, обробку конфліктів, додавання віддаленого репозиторію, push, pull і clone. Внутрішня модель прототипу ґрунтується на

службовому каталозі .vcs, у якому зберігаються об’єкти, покажчики гілок, файл HEAD, index, config.json і MERGE_HEAD.

У третьому розділі було проведено тестування програмного прототипу. Для перевірки реалізованої функціональності використано модульне та інтеграційне тестування із застосуванням стандартного фреймворку unittest. Тестовий набір охоплює 12 груп тестових сценаріїв TC001–TC012, у межах яких виконано 27 автоматизованих перевірок. Під час тестування підтверджено коректність генерації SHA-1-хешів, запису та читання об’єктів, zlib-стиснення, створення комітів, роботи з гілками, команди status, checkout, merge, обробки конфліктів, push, pull і clone.

Результати тестування показали, що програмний прототип виконує поставлені функціональні вимоги та коректно працює з основними службовими структурами репозиторію. Система забезпечує збереження зафіксованих станів проєкту, відновлення вмісту файлів за об’єктами, підтримку гілкування, фіксацію результатів злиття та локальний обмін даними між репозиторіями.

Розроблений програмний продукт має навчально-прикладне значення, оскільки дозволяє простежити внутрішню логіку роботи систем контролю версій без надлишкової складності промислових інструментів. Прототип може бути використаний для демонстрації принципів побудови DVCS, аналізу службової структури репозиторію та вивчення базових алгоритмів збереження історії змін.

Подальший розвиток роботи може передбачати реалізацію мережевої синхронізації через HTTP або SSH, додавання механізмів автентифікації користувачів, керування правами доступу, створення графічного інтерфейсу, оптимізацію об’єктного сховища та вдосконалення алгоритмів рядкового злиття для великих файлів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chacon S., Straub B. Pro Git. 2nd ed. New York : Apress, 2014. URL: https://git-scm.com/book/en/v2 (дата звернення: 15.06.2026).

2. Collins-Sussman B., Fitzpatrick B. W., Pilato C. M. Version Control with Subversion. For Subversion 1.8. URL: https://svnbook.red-bean.com/en/1.8/svn-book.html (дата звернення: 05.06.2026).

3. Mercurial SCM. Mercurial SCM: Work easier, Work faster. URL: https://www.mercurial-scm.org/ (дата звернення: 05.06.2026).

4. Git. Reference documentation. URL: https://git-scm.com/docs (дата звернення: 05.06.2026).

5. Git. gitglossary Documentation. URL: https://git-scm.com/docs/gitglossary (дата звернення: 15.06.2026).

6. Chacon S., Straub B. Git Internals – Git Objects. In: Pro Git. 2nd ed. URL: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects (дата звернення: 05.06.2026).

7. ISO/IEC/IEEE 29148:2018. Systems and software engineering — Life cycle processes — Requirements engineering. URL: https://www.iso.org/standard/72089.html (дата звернення: 05.06.2026).

8. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley Professional, 2021. URL: https://www.oreilly.com/library/view/software-architecture-in/9780136885979/ (дата звернення: 05.06.2026).

9. Python Software Foundation. The Python Standard Library. Python 3 documentation. URL: https://docs.python.org/3/library/index.html (дата звернення: 05.06.2026).

10. Python Software Foundation. pathlib — Object-oriented filesystem paths. Python 3 documentation. URL: https://docs.python.org/3/library/pathlib.html (дата звернення: 05.06.2026).

11. Python Software Foundation. os — Miscellaneous operating system interfaces. Python 3 documentation. URL: https://docs.python.org/3/library/os.html (дата звернення: 05.06.2026).

12. Python Software Foundation. hashlib — Secure hashes and message digests. Python 3 documentation. URL: https://docs.python.org/3/library/hashlib.html (дата звернення: 05.06.2026).

13. National Institute of Standards and Technology. Secure Hash Standard (SHS) : FIPS PUB 180-4. Gaithersburg : NIST, 2015. URL: https://csrc.nist.gov/pubs/fips/180-4/upd1/final (дата звернення: 05.06.2026).

14. Python Software Foundation. zlib — Compression compatible with gzip. Python 3 documentation. URL: https://docs.python.org/3/library/zlib.html (дата звернення: 05.06.2026).

15. Deutsch P., Gailly J.-L. ZLIB Compressed Data Format Specification version 3.3 : RFC 1950. IETF, 1996. URL: https://datatracker.ietf.org/doc/html/rfc1950 (дата звернення: 05.06.2026).

16. Python Software Foundation. json — JSON encoder and decoder. Python 3 documentation. URL: https://docs.python.org/3/library/json.html (дата звернення: 05.06.2026).

17. Bray T. The JavaScript Object Notation (JSON) Data Interchange Format : RFC 8259. IETF, 2017. URL: https://datatracker.ietf.org/doc/html/rfc8259 (дата звернення: 05.06.2026).

18. Python Software Foundation. argparse — Parser for command-line options, arguments and sub-commands. Python 3 documentation. URL: https://docs.python.org/3/library/argparse.html (дата звернення: 05.06.2026).

19. Python Software Foundation. unittest — Unit testing framework. Python 3 documentation. URL: https://docs.python.org/3/library/unittest.html (дата звернення: 05.06.2026).

20. Python Software Foundation. tempfile — Generate temporary files and directories. Python 3 documentation. URL: https://docs.python.org/3/library/tempfile.html (дата звернення: 05.06.2026).

21. Python Software Foundation. subprocess — Subprocess management. Python 3 documentation. URL: https://docs.python.org/3/library/subprocess.html (дата звернення: 05.06.2026).

22. Python Software Foundation. shutil — High-level file operations. Python 3 documentation. URL: https://docs.python.org/3/library/shutil.html (дата звернення: 05.06.2026).

23. ISO/IEC/IEEE 29119-1:2022. Software and systems engineering — Software testing — Part 1: General concepts. URL: https://www.iso.org/standard/81291.html (дата звернення: 05.06.2026).

ДОДАТКИ

Додаток А – Фрагменти програмного коду автоматизованих тестів

У додатку наведено фрагменти програмного коду автоматизованих тестів, які використовувалися для перевірки програмного прототипу локальної розподіленої системи контролю версій. Тести реалізовано мовою Python із використанням стандартного фреймворку unittest. Для ізоляції перевірок застосовувалися тимчасові каталоги, а для запуску команд програмного прототипу використовувався командний інтерфейс.

У лістингу А.1 наведено фрагмент тесту TC001, який перевіряє коректність генерації SHA-1-хешу. У межах перевірки контролюється довжина хешу, відповідність результату еталонному значенню hashlib.sha1 та повторюваність результату для однакових вхідних даних.

Лістинг А.1 – Перевірка генерації SHA-1-хешу

```
class TC001_SHA1Generation(VCSTestBase):
     def test_sha1_length_is_40_chars(self) -> None:
         _sha1, _, _ = self._import_helpers()
         samples = [
         b"Hello, VCS!",
         b"",
         b"\x00\xff\xfe" * 100,
         "Привіт, світ!".encode("utf-8"),
         ]
        for data in samples:
             with self.subTest(data=data[:20]):
                 result = _sha1(data)
                 self.assertIsInstance(result, str)
                 self.assertEqual(len(result), 40)

     def test_sha1_matches_hashlib_reference(self) -> None:
         _sha1, _, _ = self._import_helpers()
         payload = b"Diploma project test file content"
         expected = hashlib.sha1(payload).hexdigest()
         actual = _sha1(payload)

         self.assertEqual(actual, expected)

     def test_sha1_is_deterministic(self) -> None:
         _sha1, _, _ = self._import_helpers()
         data = b"Repeated hashing must be stable"
         first = _sha1(data)

         self.assertEqual(first, _sha1(data))
         self.assertEqual(first, _sha1(data))
```

У лістингу А.2 наведено фрагмент тесту TC002, який перевіряє роботу об’єктного сховища. Тест підтверджує, що дані зберігаються у службовому каталозі .vcs/objects, розміщуються за шляхом, сформованим на основі SHA-1-хешу, стискаються засобами zlib та можуть бути відновлені без втрати вмісту.

Лістинг А.2 – Перевірка збереження та відновлення об’єктів

```
class TC002_ObjectStorageRoundTrip(VCSTestBase):
      def setUp(self) -> None:
            super().setUp()
            self._init_repo()
      def test_roundtrip_text_data(self) -> None:
          _, _write_object, _read_object = self._import_helpers()
          original = "Вміст тестового файлу для перевірки zlib.".encode("utf-8")
          sha = _write_object(original)

          self.assertEqual(original, _read_object(sha))

      def test_object_file_exists_at_correct_path(self) -> None:
          _, _write_object, _ = self._import_helpers()
          data = b"Check physical file existence"
          sha = _write_object(data)

          expected = Path(self._tmpdir.name) / ".vcs" / "objects" / sha[:2] /
      sha[2:]
          self.assertTrue(expected.exists())

      def test_stored_file_is_compressed(self) -> None:
          _, _write_object, _ = self._import_helpers()
          data = b"This content will be compressed by zlib."
          sha = _write_object(data)

          raw_path = Path(self._tmpdir.name) / ".vcs" / "objects" / sha[:2] /
      sha[2:]
          decompressed = zlib.decompress(raw_path.read_bytes())

          self.assertEqual(decompressed, data)
```

У лістингу А.3 наведено фрагмент тесту TC003, який перевіряє створення коміту. Тест підтверджує, що після виконання команд init, add і commit покажчик гілки main містить SHA-1-ідентифікатор нового коміту, індекс очищується, а об’єкт коміту містить необхідні службові поля.

Лістинг А.3 – Перевірка створення коміту

```
class TC003_CommitCreation(VCSTestBase):
      def _do_commit(self, message: str = "initial commit") -> str:
           self._init_repo()
           self._write_file("test.txt", "Hello, VCS!")
           self._run("add", "test.txt")
           self._run("commit", "-m", message)
```

```
            ref = Path(self._tmpdir.name) / ".vcs" / "refs" / "heads" / "main"

            return ref.read_text(encoding="utf-8").strip()

        def test_main_ref_contains_sha_after_commit(self) -> None:
            sha = self._do_commit()

            self.assertEqual(len(sha), 40)
            self.assertTrue(all(c in "0123456789abcdef" for c in sha))

        def test_index_is_cleared_after_commit(self) -> None:
            self._do_commit()
            index_path = Path(self._tmpdir.name) / ".vcs" / "index"

            self.assertEqual(json.loads(index_path.read_text(encoding="utf-8")),
{})

        def test_commit_object_has_required_fields(self) -> None:
            sha = self._do_commit(message="перевірка полів коміту")
            obj_path = Path(self._tmpdir.name) / ".vcs" / "objects" / sha[:2] /
sha[2:]
            commit =
json.loads(zlib.decompress(obj_path.read_bytes()).decode("utf-8"))

            for field in {"tree", "parent", "author", "timestamp", "message"}:
                with self.subTest(field=field):
                    self.assertIn(field, commit)
```

У лістингу А.4 наведено фрагмент тесту TC004, який перевіряє створення нової гілки. Тест підтверджує, що гілка створюється як окремий файл у каталозі .vcs/refs/heads, вказує на поточний коміт, а повторне створення гілки з тією самою назвою завершується помилкою.

Лістинг А.4 – Перевірка створення гілки

```
class TC004_BranchCreation(VCSTestBase):
        def setUp(self) -> None:
            super().setUp()
            self._init_repo()
            self._write_file("test.txt", "branch test content")
            self._run("add", "test.txt")
            self._run("commit", "-m", "базовий коміт")

        def _get_main_sha(self) -> str:
            ref = Path(self._tmpdir.name) / ".vcs" / "refs" / "heads" / "main"
            return ref.read_text(encoding="utf-8").strip()

        def test_branch_file_is_created(self) -> None:
            result = self._run("branch", "feature")

            self.assertEqual(result.returncode, 0)

            branch_file = Path(self._tmpdir.name) / ".vcs" / "refs" / "heads" /
"feature"
            self.assertTrue(branch_file.exists())

        def test_new_branch_points_to_current_commit(self) -> None:
```

```
        main_sha = self._get_main_sha()
        self._run("branch", "feature")

        branch_file = Path(self._tmpdir.name) / ".vcs" / "refs" / "heads" /
"feature"
        feature_sha = branch_file.read_text(encoding="utf-8").strip()

        self.assertEqual(main_sha, feature_sha)

    def test_duplicate_branch_returns_error(self) -> None:
        self._run("branch", "feature")
        result = self._run("branch", "feature")

        self.assertNotEqual(result.returncode, 0)
```

У лістингу А.5 наведено фрагмент тесту TC005, який перевіряє обробку помилкового сценарію під час переходу на неіснуючу гілку. Тест підтверджує, що команда завершується з помилкою, виводить діагностичне повідомлення та не змінює файл HEAD. Це означає, що некоректна команда не пошкоджує поточний стан репозиторію.

Лістинг А.5 – Перевірка помилкового переходу на неіснуючу гілку

```
class TC005_CheckoutNonexistentBranch(VCSTestBase):
    def setUp(self) -> None:
        super().setUp()
        self._init_repo()
        self._write_file("test.txt", "checkout error test")
        self._run("add", "test.txt")
        self._run("commit", "-m", "коміт для тесту checkout")

    def _read_head(self) -> str:
        head_path = Path(self._tmpdir.name) / ".vcs" / "HEAD"
        return head_path.read_text(encoding="utf-8").strip()

    def test_checkout_unknown_branch_fails(self) -> None:
        result = self._run("checkout", "unknown")

        self.assertNotEqual(result.returncode, 0)

    def test_checkout_unknown_branch_prints_error_message(self) -> None:
        result = self._run("checkout", "unknown")
        combined = (result.stdout + result.stderr).lower()
        indicators = ["unknown", "не знайден", "not found", "помилка",
"error"]

        self.assertTrue(any(kw in combined for kw in indicators))

    def test_head_unchanged_after_failed_checkout(self) -> None:
        head_before = self._read_head()
        self._run("checkout", "unknown")

        self.assertEqual(head_before, self._read_head())
```

Додаток Б – Посилання на репозиторій

Програмний код розробленого проєкта розміщено за адресою https://github.com/hellstrvck/vcs-diploma-baziuk.